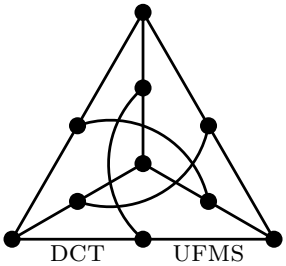
Algoritmos BSP/CGM para o Fecho Transitivo

Cristiano Costa Argemon Vieira

Dissertação de Mestrado

Orientação:Prof. Dr. Edson Norberto Cáceres

Área de Concentração: Ciência da Computação



Departamento de Computação e Estatística
Centro de Ciências Exatas e Tecnologia
Universidade Federal de Mato Grosso do Sul
15 de setembro de 2005

Algoritmos BSP/CGM para o Fecho Transitivo

Cristiano Costa Argemon Vieira

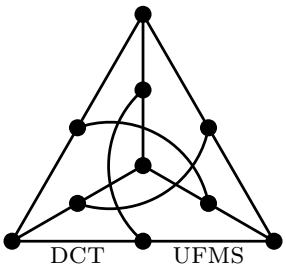
Dissertação de Mestrado

Orientação:Prof. Dr. Edson Norberto Cáceres

Área de Concentração: Ciência da Computação

Dissertação apresentada como requisito parcial para a obtenção do título de mestre em Ciência da Computação.

Durante a elaboração deste trabalho o autor recebeu apoio financeiro da CAPES.



Departamento de Computação e Estatística
Centro de Ciências Exatas e Tecnologia
Universidade Federal de Mato Grosso do Sul
15 de setembro de 2005

Algoritmos BSP/CGM para o Fecho Transitivo

Este exemplar corresponde à redação final da dissertação de mestrado devidamente corrigida e defendida por Cristiano Costa Argemon Vieira e aprovada pela comissão julgadora.

Campo Grande-MS, 15 de setembro de 2005

Banca Examinadora:

prof. Dr. Edson Norberto Cáceres - DCT/UFMS (Orientador)

prof. Dr. Henrique Mogelli - DCT/UFMS

prof. Dr. Siang Wun Song - IME/USP

Para minha esposa e para minha mãe.

Agradecimentos

Todos os objetivos que alcancei na vida exigiram muito esforço, dedicação e, em algumas situações, privações. Em nenhuma destas conquistas fui o único a dar algo em troca. Sempre existiram pessoas comprometidas a tornarem as coisas possíveis. Por isso, neste momento que brindo mais uma conquista, quero compartilhar esta felicidade com as pessoas que me ajudaram a vivenciar este momento singular. Agradeço a Deus por me dar saúde e forças para lutar pelo que quero e pelo que acredito. Agradeço a minha mãe Mara e a minha esposa Gretta por existirem e por serem. Meu agradecimento a estas pessoas é bem maior do que tudo posto.

Agradeço ao pai Edson Vicente pelos incentivos psicológicos e apoios financeiros. Agradeço ao meu sogro por permitir que eu acampasse em sua casa por algum tempo e a minha sogra por me tratar tão bem. Agradeço a minha vó Otília por ser a melhor vó do mundo. Agradeço ao meu pai Luiz Gonzaga Argemon Vieira. Agradeço ao meu irmão Rodrigo por ser meu melhor amigo e companheiro.

Agradeço ao Professor Edson Norberto Cáceres pela orientação neste trabalho, pela paciência que teve em várias circunstâncias e por ter acreditado que chegaríamos até aqui. Agradeço também a ele pelas experiências acadêmicas que me proporcionou durante este período as quais me fizeram amadurecer e despertar o senso de pesquisa.

Agradeço também ao IC-UNICAMP por permitir que executássemos nossas implementações no *Beowulf* para a obtenção dos resultados descritos neste trabalho, a CAPES pelo apoio financeiro e aos professores do DCT/UFMS pelos anos que estivemos juntos.

Aos amigos Anderson, Bianca, Carlos Juliano, Cláudio e Márcio, agradeço pelos momentos divertidíssimos que tivemos durante os almoços e durante os encontros na sala dos professores substitutos do DCT/UFMS. Estes momentos serviram como um refúgio à descontração e com certeza seriam ótimos enredos para bons livros. Sempre lembrarei destes momentos com grande alegria.

A todas estas pessoas, ofereço a felicidade e a alegria que sinto com mais esta conquista.

Cristiano Costa Argemon Vieira

Resumo

Apresentamos duas estratégias e dois algoritmos BSP/CGM para computar o fecho transitivo de um digrafo. Nossas idéias foram obtidas através da avaliação dos resultados obtidos pelos algoritmos BSP/CGM de *Alves et al.* e *Castro Jr.*. Melhoramos o desempenho destes algoritmos diminuindo o tamanho das mensagens trocadas entre os processadores, a computação local e a quantidade de rodadas de comunicação entre os processadores. Os resultados obtidos através da implementação das nossas estratégias e algoritmos foram melhores que os resultados apresentados por outros autores.

Abstract

We show two approaches and two BSP/CGM algorithms to compute a transitive closure of digraphs. Our ideas are based in a detailed observation of BSP/CGM algorithms presented by *Alves et al.* and *Castro Jr.*. We improved the algorithms performance by decreasing the message length changed between processors, the local computation and the number of communication rounds. The results obtained by implementations of our approaches and algorithms are better than the results obtained by other authors.

Conteúdo

Capítulo 1

Introdução

Este capítulo contém um breve resumo das principais soluções apresentadas para computar o fecho transitivo utilizando algoritmos seqüenciais e algoritmos paralelos. Além disso, apresentamos a notação e as definições que serão utilizadas no decorrer do texto e o modelo computacional utilizado.

1.1 O Problema Fecho Transitivo

Considere um grafo direcionado (digrafo) D=(V, E), onde V é o conjunto de vértices e E é o conjunto de arestas. O fecho transitivo de D é o digrafo D^t=(V, E^t) tal que para todos os pares de vértices v_i, v_j em V, existe uma aresta de v_i para v_j em E^t se e somente se, v_j pode ser alcançável a partir de v_i. A Figura 1.1 apresenta o fecho transitivo de um digrafo.

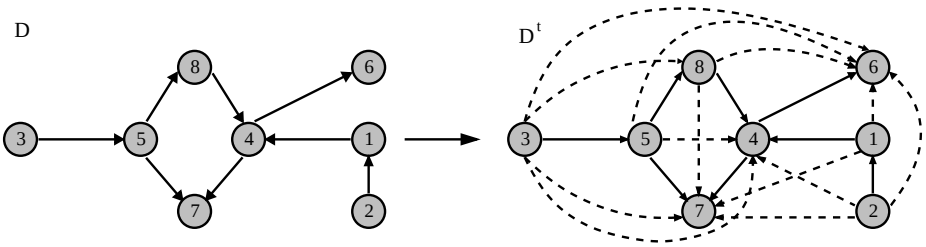


Figura 1.1: *Fecho Transitivo D^t do digrafo D.*

A computação eficiente do fecho transitivo de um digrafo é exigida em muitas aplicações computacionais, tais como na análise de fluxo e dependência em digrafos que representam sistemas paralelos e distribuídos, no projeto de compiladores e como um sub-problema importante na avaliação de consultas recursivas em bancos de dados.

Diversos algoritmos seqüenciais e paralelos foram propostos para computar o fecho transitivo. A computação do fecho transitivo de um digrafo foi estudada em 1959 por *Roy* [27]. Utilizando uma matriz de adjacências para armazenar o digrafo, *Warshall* [34] apresentou um algoritmo seqüencial com complexidade $O(n^3)$. Representando o digrafo

através de uma matriz de adjacência de bits, *Baase* [3] descreve uma variação do algoritmo de *Warshall* com complexidade de tempo $O(\frac{n^3}{\alpha})$, onde α é a quantidade de bits que podem ser armazenados em um tipo de dado primitivo.

Usando uma busca em profundidade (BP) ou uma busca em largura (BL), encontramos todos os vértices alcançáveis a partir do vértice v. Aplicando uma busca para cada um dos vértices do digrafo, computamos o fecho transitivo. Ambas buscas podem ser realizadas com complexidade de tempo $O(n + m)$. Conseqüentemente, o fecho transitivo pode ser computado em $O(n(n + m))$ [12].

Existem outras estratégias para computar o fecho transitivo de um digrafo [15, 22]. O melhor algoritmo seqüencial para resolver o problema para grafos esparsos tem complexidade de tempo $O(nm)$ (*Habib et al.* [13], *Koubková e Koubek* [19] e *Simon* [28]). Para grafos densos, o fecho transitivo pode ser computado seqüencialmente utilizando multiplicação de matrizes em tempo $O(n^{2.376})$ [7]. Embora os algoritmos seqüenciais, baseados na multiplicação de matrizes, para computar o fecho transitivo tenham boa complexidade assintótica, estes algoritmos não apresentam bons resultados na prática. Isso se deve ao fato de existirem algumas constantes agregadas à complexidade.

No modelo PRAM (Parallel Random Access Machine), *JáJá* [16] descreveu um algoritmo paralelo para computar o fecho transitivo de um digrafo. O algoritmo computa o fecho transitivo através da multiplicação de matrizes com complexidade de tempo $O(\lg n)$, usando $O(n^3 \lg n)$ processadores no modelo CRCW PRAM e com complexidade de tempo $O(\lg^2 n)$, usando $O(M(n) \lg n)$ processadores no modelo CREW PRAM, onde $M(n)$ é o melhor algoritmo seqüencial conhecido para multiplicar duas matrizes de tamanho n x n. Uma outra solução no modelo PRAM foi apresentada por *Karp e Ramachandran* [17].

Utilizando o modelo BSP (Bulk Synchronous Parallel) *Tiskin* [32] apresentou um algoritmo para computar o menor caminho entre todos os pares de vértices com complexidade de tempo $O(\frac{n^3}{p})$ e $O(\lg p)$ rodadas de comunicação. Para computar o fecho transitivo utilizando este algoritmo basta atribuir o custo 1 para as arestas existentes.

Baseados no algoritmo seqüencial de *Warshall* [34], *Kumar et al.* [20] apresentaram um algoritmo paralelo onde a matriz de adjacências que representa o digrafo é dividida em p sub-matrizes de tamanho $\frac{n}{\sqrt{p}}$ x $\frac{n}{\sqrt{p}}$, cada sub-matriz é atribuída a um processador. Cada processador armazena $O(\frac{n^2}{p})$ dados da matriz de adjacências. Na k-ésima iteração do laço mais externo do algoritmo seqüencial de *Warshall* (utilizado na computação local) cada processador envia seus dados para os outros $\sqrt{p} - 1$ processadores que contém os dados da k-ésima linha e da k-ésima coluna da matriz de adjacências. O passo de sincronização requer tempo $O(\lg p)$. Conseqüentemente, a complexidade de comunicação é $O(\frac{n^2}{\sqrt{p}} \lg p)$. A complexidade de tempo deste algoritmo é $O(\frac{n^3}{p})$.

No modelo de grafo de dependências *Pagourtzis et al* [24, 25] apresentaram algoritmos paralelos para computar o fecho transitivo e implementaram esses algoritmos utilizando PVM (Parallel Virtual Machine) em uma rede de estações de trabalho.

Utilizando uma variação do modelo BSP, o BSP/CGM (Bulk Synchronous Parallel/Coarse Grained Multicomputer), *Cáceres et al.* [5] apresentaram um algoritmo para

computar o fecho transitivo de um digrafo acíclico usando $\lg p + 1$ rodadas de comunicação, onde p é o número de processadores. Este algoritmo possui complexidade $O(\frac{nm}{p})$ de computação local e requer $O(\frac{nm}{p})$ espaço em cada processador. Utilizando as idéias do algoritmo de *Cáceres et al.* [5] e do algoritmo seqüencial de *Warshall* [34], *Alves et al.* [2] e *Castro Jr.* [6] apresentaram um algoritmo BSP/CGM para computar o fecho transitivo de um digrafo com complexidade $O(\frac{n^3}{p})$ de computação local e $O(\frac{n^2}{p})$ espaço em cada processador. O algoritmo apresentado por *Alves et al.* [2] e *Castro Jr.* [6] computa o fecho transitivo utilizando no máximo $O(p)$ rodadas de comunicação. Nos experimentos realizados pelos autores para digrafos gerados aleatoriamente, $\lg p + 1$ rodadas foram suficientes para o cálculo do fecho transitivo e os tempos experimentais obtidos apresentaram bons *speed-ups*. Um exemplo em que este algoritmo pode usar mais que $\lg p + 1$ rodadas de comunicação foi apresentado por *Alves et al.* [2].

Neste trabalho apresentamos dois novos algoritmos BSP/CGM para computar o fecho transitivo de um digrafo. Estes algoritmos foram elaborados a partir de estudos realizados nas características da computação local e da comunicação das soluções apresentados por outros autores. O primeiro algoritmo diminui a computação local atualizando o fecho transitivo após a inserção de uma aresta no digrafo. Desta maneira, evita computar algumas arestas que já existem no digrafo que representa o fecho transitivo. O segundo algoritmo evita a troca de mensagens entre os processadores. Além disso, apresentamos duas estratégias para diminuir a computação local e o tamanho das mensagens trocadas pelo algoritmo de *Alves et al.* [2]. Apresentamos também, um comparativo entre os resultados obtidos por *Alves et al.* [2] e os que obtivemos a partir da implementação dos nossos algoritmos e estratégias.

No restante deste capítulo, descrevemos a notação que utilizamos e o modelo computacional no qual estamos propondo nossos algoritmos. No capítulo seguinte descrevemos algoritmos seqüenciais e algoritmos paralelos que serão utilizados em nosso trabalho. No terceiro capítulo apresentamos as estratégias propostas e os dois novos algoritmos paralelos para computar o fecho transitivo de um digrafo. Apresentamos, no quarto capítulo, os resultados obtidos através da implementação dos algoritmos que estamos propondo. No quinto capítulo apresentamos nossas conclusões.

1.2 Notação

A seguir apresentamos algumas definições e a notação utilizadas neste trabalho que são baseadas em [1, 14, 30].

Um **grafo** $D = (V, E)$ é um conjunto finito não-vazio V e um conjunto E de pares não-ordenados de elementos distintos de V, tal que $|V| = n$ e $|E| = m$. Os elementos de V são os vértices e os elementos de E são as arestas de D, respectivamente. Cada aresta é representada por um par não-ordenado $e = \{v, w\}$. Os vértices v e w, denominados adjacentes, são os extremos da aresta e.. Quando cada aresta pertencente ao conjunto E é representada por um par ordenado, dizemos que o grafo é um grafo direcionado (digrafo) e a aresta é denominada **direcionada** e incidente em apenas um vértice, o

vértice de chegada.

Uma seqüência de vértices $v_1, ..., v_k$ tal que $(v_i, v_{i+1}) \in E$, $1 \leq i \leq k-1$, é denominado **caminho** de v_1 a v_k. Diz-se então, que v_i alcança v_k e v_k é alcançável a partir de v_i. Um caminho de k vértices é formado por $k-1$ arestas (v_1,v_2), (v_2,v_3), ..., (v_{k-1},v_k), onde $k-1$ é o **comprimento** do caminho. Se todos os vértices do caminho forem distintos, a seqüência recebe o nome de **caminho simples**. O caminho contendo arestas direcionadas é denominado **caminho direcionado**.

O caminho entre os vértices $v_i, v_j \in V$ é denotado como segue:

(i) $v_i \rightarrow v_j$, um caminho direcionado consistindo de somente uma aresta iniciando no vértice v_i e terminando no vértice v_j.

(ii) $v_i \rightsquigarrow v_j$, um caminho direcionado iniciando no vértice v_i e terminando no vértice v_j sem quantificar ou definir os vértices intermediários.

(iii) $v_i \rightsquigarrow^k v_j$, um caminho direcionado iniciando no vértice v_i e terminando no vértice v_j com k vértices intermediários ($k+1$ arestas).

(iv) $v_i \rightsquigarrow^{...k} v_j$, um caminho direcionado iniciando no vértice v_i e terminando no vértice v_j com $v_1, v_2, ..., v_k$ como vértices intermediários.

Em algumas ocasiões $v_i \rightarrow v_j$ é denotado como sendo a aresta e_{ij} ou (v_i, v_j).

O **fecho transitivo** de um digrafo D é o digrafo $D^t = (V, E^t)$, onde $E^t = \{(v_i \rightarrow v_j : \exists\ v_i \rightsquigarrow v_j\ em\ D)\}$. Isto é, para todos os pares de vértices v_i, v_j em V, existe uma aresta de v_i para v_j em E^t se, v_j pode ser alcançável a partir de v_i.

Utilizamos duas maneiras para representar um digrafo: matriz de adjacências e lista de adjacências. A Figura 1.2 apresenta a matriz de adjacências e a lista de adjacências de um digrafo D.

1.3 O Modelo Computacional Utilizado

Existem vários modelos propostos para a computação paralela [26, 9, 8, 33]. Não há um modelo que seja amplamente aceito (da mesma forma que o modelo seqüencial RAM) para o projeto e análise de algoritmos paralelos.

Neste trabalho, utilizamos o modelo BSP/CGM (Bulk Synchronous Parallel/Coarse Grained Multicomputer) [9, 10, 33]. Seja N o tamanho da entrada do problema. Um computador BSP/CGM consiste de um conjunto de p processadores, cada um com memória local. Cada processador é conectado por um roteador através do qual podem ser enviadas/recebidas mensagens para/de outros processadores. Um algoritmo BSP/CGM consiste de rodadas alternadas de computação e comunicação separadas por uma barreira de sincronização.

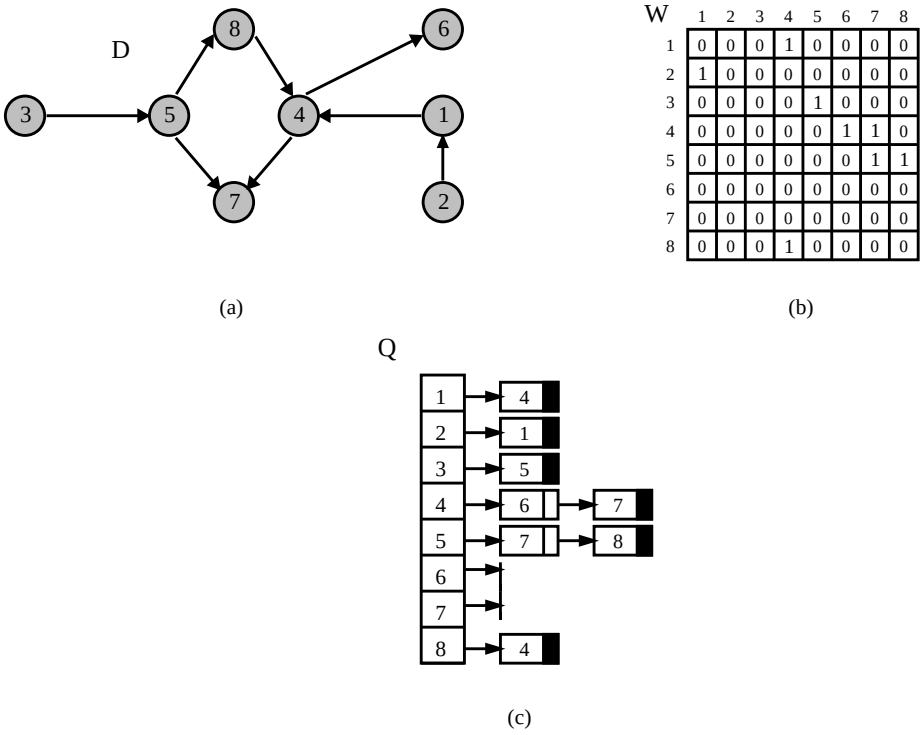


Figura 1.2: a) D=(V, E), b) Matriz de adjacências de D e c) Lista de Adjacências de D.

Em uma rodada de computação, geralmente, utilizamos o melhor algoritmo seqüencial para processar os dados locais. As mensagens entre os processadores são enviadas em uma rodada de comunicação. No modelo BSP/CGM, o custo de comunicação é modelado pelo número de rodadas de comunicação. Cada processador pode enviar/receber no máximo $O(\frac{N}{p})$ dados em cada rodada de comunicação.

Encontrar um algoritmo eficiente no modelo BSP/CGM é equivalente a minimizar o número de rodadas de comunicação, bem como o tempo total de computação local. A grande vantagem deste modelo é que produz resultados próximos dos obtidos pelas máquinas paralelas disponíveis no mercado.

1.4 Ambiente de Troca de Mensagens

O paradigma de troca de mensagens (*Message Passing*) consiste em um conjunto de métodos que torna possível a criação, a gerência, a comunicação e a sincronização entre processos quando não existe memória compartilhada. Para permitir que linguagens como C e Fortran incorporassem esse paradigma, foram definidas extensões para essas linguagens, geralmente na forma de bibliotecas, chamadas de ambientes de troca de mensagens. Neste trabalho utilizamos a biblioteca LAM-MPI (http://www.lam-mpi.org/). Informações mais detalhadas sobre a biblioteca podem ser encontradas em [4, 23, 29].

1.5 Discussão

Neste capítulo apresentamos, de maneira resumida, os resultados seqüenciais e os resultados paralelos existentes para computar o fecho transitivo de um digrafo. Além disso, descrevemos a notação e terminologia e fizemos uma breve descrição do modelo BSP/CGM que será utilizado neste trabalho.

Capítulo 2

Algoritmos para Computar o Fecho Transitivo

Neste capítulo, descrevemos quatro algoritmos seqüenciais e dois paralelos para computar o fecho transitivo. Estes algoritmos contêm idéias que utilizaremos para formular nossas contribuições.

2.1 Algoritmos Seqüenciais

Como mencionado anteriormente, *Warshall* [34] apresentou um algoritmo com complexidade $O(n^3)$ utilizando uma matriz de adjacências para representar o digrafo. Utilizando o algoritmo de *Warshall* e representando a matriz de adjacências por uma matriz de bits, *Baase* [3] descreveu um algoritmo $O(\frac{n^3}{\alpha})$, onde α é a quantidade de bits que podem ser armazenados em um tipo de dado primitivo.

Uma outra forma de computar o fecho transitivo é utilizar uma busca no digrafo para encontrar todos os vértices v_j alcançáveis a partir do vértice v_i e criar arestas e_{ij}. Encontrando todos os vértices que um vértice alcança, computamos o fecho transitivo. Este algoritmo possui complexidade $O(n(n+m))$.

Os algoritmos dinâmicos para manter o fecho transitivo após a inserção de uma aresta podem ser utilizados para computar fecho transitivo. Basta realizar a atualização do fecho após a inserção de cada aresta.

Para digrafos densos, o fecho transitivo pode ser computado utilizando a multiplicação de matrizes em tempo $O(n^{2.376})$ [7].

Outras técnicas para computar o fecho transitivo são descritas na literatura. A seguir descreveremos com mais detalhes os quatro algoritmos que são utilizados em nossos algoritmos paralelos.

2.1.1 O Algoritmo de Warshall

Seja um digrafo $D = (V, E)$, representado pela matriz de adjacências W, da qual cada posição w_{ij} representa a existência da aresta e_{ij} entre os vértices v_i e v_j. Considere $w_{ij} \in \{0, 1\}$, 1 se existir a aresta e_{ij}, caso contrário 0.

O Algoritmo 1 descreve os passos do algoritmo de *Warshall*. A entrada do algoritmo é um digrafo D representado por uma matriz de adjacências W. O Algoritmo 1 transforma W em uma matriz de adjacências W^t que representa o fecho transitivo D^t de D. Denotamos por w^t_{ij} cada posição de W^t

Algoritmo 1: Warshall-I

Entrada: D = Digrafo representado pela matriz de adjacências W.
Saída: D^t = fecho transitivo de D representado pela matriz de adjacências W^t.

```
1:  n ← |V|
2:  W^t ← W
3:  para k ← 1 até n faça
4:      para i ← 1 até n faça
5:          se  i ≠ k ∧ w^t_ik então
6:              para j ← 1 até n faça
7:                  w^t_ij ← w^t_ij ∨ w^t_kj
8:              fim para
9:          fim se
10:     fim para
11: fim para
12: Retorne D^t
```

Considere um estágio do Algoritmo 1 como sendo a k-ésima iteração do laço mais externo. No primeiro estágio, verifica-se a existência de um caminho $v_i \rightsquigarrow^{\cdots 1} v_j$. No segundo estágio, verifica-se a existência de um caminho $v_i \rightsquigarrow^{\cdots 2} v_j$. E assim sucessivamente até o n-ésimo estágio onde verifica-se a existência de um caminho $v_i \rightsquigarrow^{\cdots n} v_j$. Apenas as arestas entre vértices que já foram computadas pelo fecho transitivo são consideradas no caminho. Conseqüentemente, w_{ij} não sofrerá alterações. Isto deve-se ao fato do algoritmo verificar se o vértice k está entre todos os pares de vértices (v_i, v_j), e não, se todos os vértices k estão entre o par de vértices (v_i, v_j). Note que $1 \leq k, i, j \leq n$.

Para computar a aresta e_{ij} deve existir o caminho $v_i \rightsquigarrow v_j$. O algoritmo de *Warshall* verifica a existência deste caminho tentando construí-lo gradativamente com a inserção de todos os vértices entre os vértices v_i e v_j. O laço mais externo é responsável pela construção sistemática deste caminho.

O tempo de execução do pior caso do Algoritmo 1 é $O(n^3)$ e o tempo de execução do melhor caso é $\Theta(n^2)$. Sua implementação é muito simples. A condição na Linha 5 verifica se $v_i \rightarrow v_k$. A aresta é gerada somente se $(w^t_{ik} \wedge w^t_{kj})$ for verdadeiro. Então, caso w^t_{ij} seja falso, não é necessário realizar as iterações do laço mais interno.

Embora a complexidade de computação seja maior a de outros algoritmos seqüenciais

existentes, como os apresentados por *Habib et al.* [13], *Koubková e Koubek* [19], *Simon* [28] e *Coppersmith e Winograd* [7], este algoritmo pode ser utilizado para qualquer tipo de grafo e além disso estamos interessados em sua estrutura, pois possui a característica de que no k-ésimo estágio podem ser geradas arestas em todas as posições da matriz de adjacências, no entanto, as arestas utilizadas na geração estão contidas na k-ésima linha e na k-ésima coluna. Esta importante característica pode ser ser utilizada na implementação paralela. A Figura **2.1** apresenta, na região mais clara, as posições da matriz de adjacências nas quais podem ser geradas novas arestas durante um determinado estágio. Na região mais escura estão contidas as arestas utilizadas para computar as novas. No k-ésimo estágio, $0 < k \leq 4$, apenas as arestas e_{ik} e e_{kj} são consultadas.

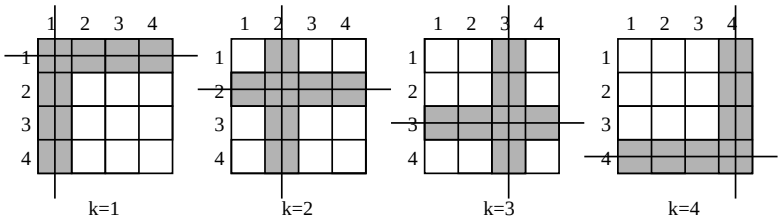


Figura 2.1: Estágios do algoritmo de *Warshall*

2.1.2 O Algoritmo de Warshall Utilizando uma Matriz de Bits

Como mencionado anteriormente, *Baase* [3] descreveu uma maneira de melhorar o algoritmo de *Warshall* utilizando uma matriz de bits para representar a matriz de adjacências. Nesta solução, cada posição da matriz de adjacência é representada por um bit.

A principal idéia desta solução está baseada no fato do laço mais interno alterar apenas a coluna da linha i. Isto é, na k-ésima iteração do laço mais externo, o algoritmo de *Warshall* verifica quais vértices v_i possuem uma aresta para o vértice v_k ($v_i \rightarrow v_k$) e cria uma aresta $w_{ij}(v_i \rightarrow v_j)$, onde j pertence ao conjunto de vértices para os quais v_k possui uma aresta ($v_k \rightarrow v_j$). As linhas $\overline{w_i}$ e $\overline{w_k}$ representam os vértices nos quais v_i e v_k incidem, respectivamente. Para criarmos as arestas w_{ij} de v_i para v_j que possuam v_k como vértice intermediário basta fazermos um OU lógico entre as linhas i e k. O Algoritmo 2 representa os passos desta solução.

No máximo n^2 instruções lógicas OU são feitas sobre as linhas de W^t. No entanto, uma linha de W^t pode não caber em uma única unidade de memória do computador e mais do que uma instrução lógica é realizada na linha 6. Cada posição w_{ij} da matriz de adjacências W deve armazenar apenas a informação de que a aresta do vértice v_i para o vértice v_j existe ou não. Esta informação pode ser armazenada em apenas um *bit*. Utilizamos os bits de um tipo de dado primitivo τ qualquer, $|\tau|=\alpha$ bits, para representar a informação da arestas. Ao fazermos um OU lógico entre duas variáveis do tipo τ computamos até α arestas no digrafo em $O(1)$ utilizando o paralelismo do processador. Desta forma, a matriz que representa o digrafo contém n linhas e $\frac{n}{\alpha}$ colunas. Pelo fato de realizarmos um OU lógico entre duas variáveis deste tipo em $O(1)$, são necessárias $\frac{n}{\alpha}$ operações para computar a linha 6. A complexidade do Algoritmo 2 é $O(n^2)$ no melhor caso e $O(\frac{n^3}{\alpha})$ no

Algoritmo 2: Warshall-II

Entrada: D = Digrafo representado pela matriz de adjacências (em bits) W.
Saída: D^t = fecho transitivo de D representado pela matriz de adjacências (em bits) W^t.

1: $n \leftarrow |V|$
2: $W^t \leftarrow W$
3: **para** $k \leftarrow 1$ até n **faça**
4: **para** $i \leftarrow 1$ até n **faça**
5: **se** w^t_{ik} **então**
6: $\overline{w^t_i} \leftarrow \overline{w^t_i} \vee \overline{w^t_k}$
7: **fim se**
8: **fim para**
9: **fim para**
10: **Retorne** D^t

pior caso.

A Figura **2.2** ilustra a computação das arestas do vértice v_i para os vértices aos quais v_k possui uma aresta durante a execução do Algoritmo 2. Isto significa fazer o vértice v_i alcançar diretamente (através de uma única aresta), todos os vértices que v_k alcança diretamente.

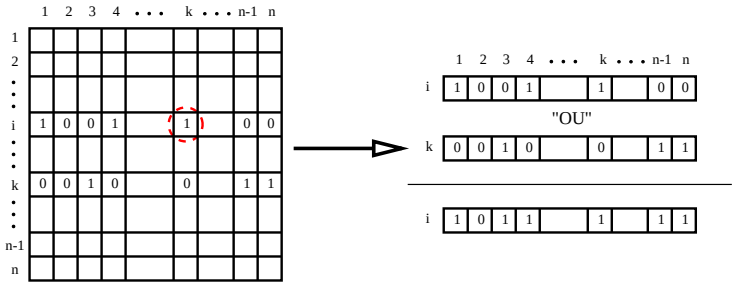


Figura 2.2: Ilustração da operação lógica "OU"entre as linhas i e k.

Embora a complexidade dos Algoritmos 1 e 2 seja $O(n^3)$, pois α é uma constante, a implementação do Algoritmo 2 apresenta resultados muito melhores que os resultados apresentados pela implementação do Algoritmo 1. A Tabela 2.1 apresenta uma comparação entre os resultados obtidos a partir da implementação dos Algoritmos 1 e 2 utilizando um microcomputador contendo 384 MB de RAM e uma UCP Intel Pentium III 700 MHz.

Verificamos os tempos utilizando $\alpha = 8$ e $\alpha = 32$. Os resultados mostram o bom desempenho do Algoritmo 2 quando comparado ao Algoritmo 1. Os digrafos foram gerados aleatoriamente.

2.1.3 Um Algoritmo Utilizando Busca em Digrafos

Como mencionado anteriormente, os Algoritmos 1 e 2 possuem complexidade $O(n^3)$ e podem ser utilizados para digrafos cíclicos e acíclicos. Ambos algoritmos utilizam uma

Número Vértices	Número Arestas	Tempo(s)		
		Alg 1	Alg 2($\alpha = 8$)	Alg 2($\alpha = 32$)
1024	5000	68.2580	1.9530	0.6810
	12500	65.9150	2.5540	0.9310
	25000	65.5340	2.4240	1.0010
	50000	65.2540	2.9640	1.0880
	100000	65.1640	3.0150	1.2100
	200000	66.5960	3.1250	1.3920

Tabela 2.1: Comparação entre os Algoritmos 1 e 2.

matriz de adjacências para representar o digrafo e computam o fecho transitivo gradativamente. Uma outra estratégia para computar o fecho transitivo é verificar quais vértices um determinado vértice alcança. Utilizando uma busca em profundidade ou uma busca em largura encontramos todos os vértices alcançáveis a partir de v_i. Seja Γ^{v_i} a árvore construída através da busca. Todos os vértices v_j presentes em Γ^{v_i} são alcançáveis a partir de v_i. Construímos Γ^{v_i}, $\forall\ i \in V$ adicionando a arestas e_{ij} ao fecho transitivo final. A busca em profundidade ou em largura a partir de um determinado vértice pode ser realizada em $O(n+m)$. Com isso, computamos o fecho transitivo em $O(n\ (n+m))$ de tempo e $O(n+m^t)$ de espaço [12], onde m^t é a quantidade de arestas do fecho transitivo.

O Algoritmo 3 apresenta os passos para computar o fecho transitivo D^t de um digrafo D baseado em uma busca em profundidade.

Algoritmo 3: FT-Busca

Entrada: D = Digrafo representado pela lista de adjacências A.
Saída: D^t = Fecho transitivo de D representado pela lista de adjacências A^t.

1: $A^t \leftarrow A$
2: **para** $i \leftarrow 1$ até n **faça**
3: Compute Γ^{v_i}.
4: Crie a aresta $a^t_{i,j} = \{(v_i \rightarrow v_j) : j \in \Gamma^{v_i}\}$
5: **fim para**
6: **Retorne** D^t

Considere um estágio do Algoritmo 3 como sendo a i-ésima iteração do laço da linha 2. Neste estágio Γ^{v_i} possui todos os vértices aos quais v_i deve atingir diretamente no fecho transitivo final. As arestas geradas em um estágio não influenciam a computação em nenhum outro estágio. A Figura **2.3** ilustra o digrafo D antes e depois do estágio $i = 3$ do Algoritmo 3. O digrafo representado por (a) refere-se ao fecho transitivo parcial de D após a realização da computação dos estágios 1 e 2. O digrafo representado por (b) refere-se a Γ^{v_3}, a árvore construída a partir do vértice v_3 utilizando uma busca em profundidade em D. O digrafo representado por (c) refere-se ao fecho transitivo parcial de D após a computação do estágio 3.

Comparando o Algoritmo 1 (*Warshall-I*) e o Algoritmo 3 (*FT-BP*) é possível observar

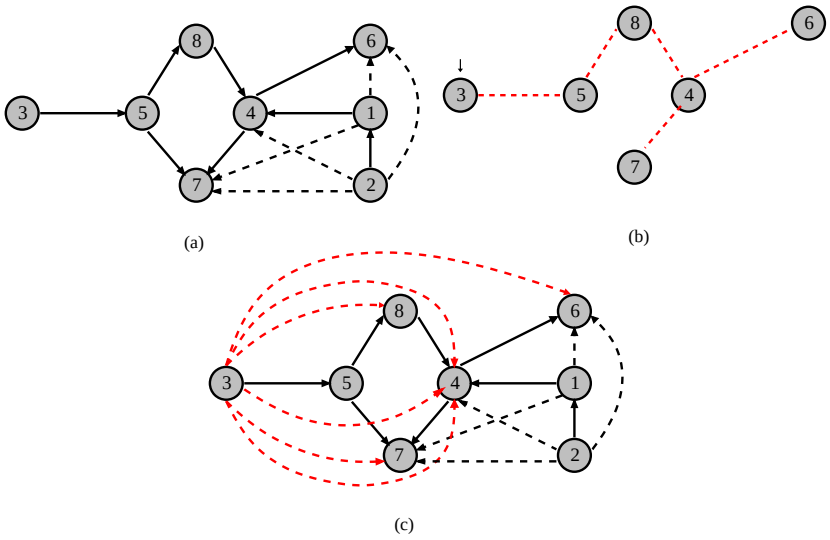


Figura 2.3: Execução do Algoritmo 3 durante a computação do estágio 3.

que para digrafos densos ambos possuem a mesma complexidade $O(n^3)$, pois, $m = O(n^2)$. Por outro lado, para digrafos esparsos, o algoritmo que utiliza busca em digrafos possui complexidade menor que a do *Warshall*.

As Tabelas 2.2 e 2.3 apresentam os resultados obtidos pela implementação dos Algoritmos 1 e 3 para computar do fecho transitivo de vários digrafos utilizando um microcomputador contendo 384 MB de RAM e uma UCP Intel Pentium III 700 MHz. Os digrafos foram gerados aleatoriamente. Variamos a quantidade de arestas em digrafos com $|V| = 512$ e $|V| = 1024$ com o objetivo de verificar em quais situações os algoritmos se destacam. É importante observar que os resultados práticos refletem os resultados teóricos dos algoritmos. Para digrafos densos o algoritmo de *Warshall* é mais rápido e possui os tempos aproximados semelhantes independente da quantidade de arestas presente no digrafo. Embora existam algumas constantes envolvidas na implementação do Algoritmo 3, os resultados obtidos a partir da implementação deste algoritmo, quando submetido a digrafos esparsos, são melhores que os resultados obtidos através da implementação do algoritmo 1.

Número Vértices	**Número Arestas**	**Tempo(s)**	
		Alg 1	**Alg 3**
512	1500	8.3920	0.0300
	20000	8.2620	1.1320
	40000	8.2820	2.6340
	80000	8.3020	5.2070
	160000	8.2920	10.1140

Tabela 2.2: Resultados obtidos para grafos com 512 vértices.

Número Vértices	Número Arestas	Tempo(s)	
		Alg 1	Alg 3
1024	4000	68.2580	0.1600
	8000	65.9150	0.2500
	16000	65.5340	1.1520
	32000	65.2540	4.4170
	64000	65.1640	8.6220
	128000	66.5960	16.3730
	256000	65.2140	32.1570
	512000	64.8030	63.2610

Tabela 2.3: Resultados obtidos para grafos com 1024 vértices.

2.1.4 Manutenção do Fecho Transitivo

A manutenção do fecho transitivo de um digrafo deve ser efetuada sempre que uma operação de inserção ou remoção de arestas for realizada. Seja, $D^t = (V, E^t)$ o fecho transitivo de $D = (V, E)$. Após a realização de uma operação de inserção ou remoção, o conjunto de arestas E de D sofrerá alterações, conseqüentemente, será necessário computarmos novamente D^t. Vários algoritmos foram apresentados para manter o fecho transitivo após uma operação que altera o digrafo. Um algoritmo é denominado *completamente dinâmico* se ele mantém o fecho transitivo após operações de inserção e remoção de uma aresta no digrafo que representa o fecho. Quando um algoritmo trata apenas uma das operações, é denominado *parcialmente dinâmico*: incremental ou decremental, para operações de inserção ou remoção, respectivamente. Algoritmos *completamente dinâmicos* são apresentados por *Demetrescu* e *Italiano* [11] e *King* e *Sagert* [18].

Neste trabalho, estamos interessados apenas em manter o fecho transitivo após a inserção de uma aresta. Algoritmos seqüenciais incrementais com complexidade $O(n)$ amortizado foram apresentados por *Italiano* [15] e *La Poutre* e *van Leeuwen* [21]. Podemos utilizar estes algoritmos para computar o fecho transitivo inserindo todas as arestas de D em um digrafo inicialmente sem nenhuma aresta. Para cada aresta inserida, basta utilizar um dos algoritmos para manter o fecho transitivo. Conseqüentemente, podemos computar o fecho transitivo em $O(nm)$ amortizado em grafos esparsos ou densos.

2.2 Algoritmos BSP/CGM

Baseados no modelo BSP/CGM, *Cáceres et al.* [5] apresentaram um algoritmo para computar o fecho transitivo de um digrafo acíclico. Utilizando as idéias desse algoritmo *Alves et al.* [2] e *Castro Jr.* [6] apresentaram e implementaram algoritmos BSP/CGM para computar o fecho transitivo de um digrafo.

A seguir descreveremos os algoritmos apresentados por *Cáceres et al* [5] e por *Alves et al* [2].

2.2.1 O Algoritmo de *Cáceres et al*

O algoritmo paralelo para computar o fecho transitivo apresentado por *Cáceres et al.* [5] recebe como entrada um digrafo acíclico distribuído pelos p processadores. A distribuição é feita conforme a rotulação obtida através da computação de uma **extensão linear** L de D. Seja $D = (V, E)$ um digrafo acíclico, com $|V| = n$ vértices e $|E| = m$ arestas. Uma extensão linear de D é uma seqüência $v_1, ..., v_n$ de seus vértices, tal que se existe uma aresta do vértice v_i para o vértice v_j, então, v_i deve aparecer antes de v_j em L. Esta distribuição garante que $O(\lg p)$ rodadas de comunicação sejam suficientes para computar D^t.

Seja $S \subseteq V$, $D(S)$ denota o digrafo formado exatamente pelas arestas de D que possuem pelo menos um de seus extremos em S.

Sendo A um caminho em D, $|A|$ denota o seu comprimento. O Algoritmo 4 apresenta os passos definidos pelo algoritmo BSP/CGM apresentado por *Cáceres et al.* [5].

Algoritmo 4: FTP - Cáceres et al

Entrada: D = Digrafo acíclico, p = processadores.
Saída: D^t = Fecho transitivo de D.

(1) Encontre uma extensão linear L de D;
(2) Seja $S_0, ..., S_{p-1}$ uma partição de $V(D)$, cujas partes tenham cardinalidades tão iguais quanto possível, e onde cada S_j seja formado por vértices consecutivos em L. Para $j = 0, ..., (p-1)$, atribua os vértices de S_j ao processador p_j;
(3) Em paralelo, cada processador p_j seqüencialmente:
  (3.1) Construa o digrafo $D(S_j)$ de D
  (3.2) Compute o fecho transitivo $D^t(S_j)$ de $D(S_j)$
  (3.3) Inclua em D as arestas $D^t(S_j) \backslash D(S_j)$
(4) Após todos os processadores terem completado o passo 3, verifique se $D^t(S_j) = D(S_j)$, para todos os processadores p_j . Se verdadeiro, o algoritmo é finalizado e D é o fecho transitivo do digrafo de entrada. Caso contrário, repita o passo 3.

Teorema 1 *O Algoritmo 4 computa corretamente o fecho transitivo de um digrafo. Além disso, ele requer no máximo* $1 + \lceil \lg p \rceil$ *iterações do passo 3.*

Prova Denote por D_i o digrafo D ao final da i-ésima iteração do passo 3. Seja D_0 o digrafo de entrada e D_i^t o fecho transitivo de D_i, i=0, 1, ..., i, onde $i \leq \lg p$. Sendo $D_i(S_j)$ um digrafo de D_i, todas as arestas do fecho transitivo $D_i^t(S_j)$ de $D_i(S_j)$ também pertencem a D_i^t, e conseqüentemente a D_0^t. Para mostrar que o algoritmo computa corretamente o fecho transitivo D_0^t de D_0 é suficiente mostrar que toda aresta de D_0^t é também uma aresta de algum $D_i^t(S_j)$. Com este propósito, seja $(v,w) \in E(D_0^t)$. Mostraremos que $(v,u) \in E(D_i^t(S_j))$, para algum i e j. Pelo fato de (v,w) ser uma aresta de D_0^t, D_0 contém um caminho z_1, ..., z_l, de v=z_1 para w=z_l. Para cada k, $1 \leq k \leq l$, denote por

$P(z_k)$ o processador para o qual z_k é atribuído. Pelo fato desta distribuição de vértices obedecer uma extensão linear, segue que $P(z_1)$, ..., $P(z_l)$ é uma seqüência não-decrescente. Além disso, os vértices atribuídos ao mesmo processador são consecutivos na seqüência. Conseqüentemente, após completar a primeira iteração do passo 3, sabemos que, D_1 contém um caminho A_1 do vértice v para o vértice w, formado somente pelos vértices de z_1,...,z_l, atribuídos a processadores distintos. Por esta razão, $|A_1| \leq p$. Se $|A_1| = 1$, então $(v, w) \in E(D_1^t(S_j))$, implicando que o algoritmo está correto. Caso contrário, seja $z_{k-1}^{'}, z_k^{'}, z_{k+1}^{'}$ três vértices consecutivos em A. Considere $P(z_k^{'}) = j$. Conseqüentemente, $(z_{k-1}^{'}, z_k^{'}), (z_k^{'}, z_{k+1}^{'}) \in E(D_1(S_j))$. Isso significa que ao final da segunda iteração do passo 3 teremos $(z_{k-1}^{'}, z_{k+1}^{'}) \in E(D_2)$. Conseqüentemente, D_2 contém um caminho A_2 do vértice v para o vértice w, formado por um subconjunto de vértices de A_1 satisfazendo $|A_2| = \left\lceil \frac{|A_1|}{2} \right\rceil$. Por indução, segue que $|A_{\lceil \lg |A_1|+1 \rceil}| = 1$, isto é, $(v, w) \in E(D^t_{\lceil \lg |A_1|+1 \rceil}(S_j))$ como requerido. Além disso, não mais do que $1 + \lg p$ iterações do passo 3 são necessárias. ∎

Não é conhecido um algoritmo BSP/CGM que compute uma extensão linear de D em tempo $O(\frac{n+m}{p})$ com p processadores. Considerando que o algoritmo do fecho transitivo apresentado por *Cáceres et al.* é $O(\frac{nm}{p})$, o passo 1 pode ser computado seqüencialmente em $O(n + m)$ por um dos processadores desde que $p \leq \frac{n^2}{m+n}$. Isto é, a memória local do processador deve ser suficiente para armazenar a lista de adjacências inteira.

2.2.2 O Algoritmo de *Alves et al*

Baseados no algoritmo de *Cáceres et al.* [5] e no algoritmo seqüencial de *Warshall* [34], *Alves et al.* [2] e *Castro Jr.* [6] apresentaram uma nova solução no modelo BSP/CGM para computar o fecho transitivo. Este algoritmo não computa a extensão linear L e distribui os dados de acordo com a numeração inicial dos vértices do digrafo D.

O algoritmo distribui a matriz de adjacências por p processadores. A distribuição é feita particionando W em p faixas horizontais e p faixas verticais formadas por $\frac{n}{p}$ linhas consecutivas e $\frac{n}{p}$ colunas consecutivas, respectivamente. Por simplicidade e sem perda de generalidade, assumimos que n é divisível por p. A i-ésima faixa horizontal e a i-ésima faixa vertical são atribuídas ao processador p_i. Observe que quase todos os elementos w_{ij} são armazenados em dois processadores.

Com esta distribuição podemos utilizar o algoritmo seqüencial de *Warshall* em cada um dos processadores. Considere um estágio do algoritmo de *Alves et al.* como sendo a k-ésima iteração do laço mais externo. Em cada estágio, o algoritmo verifica se o vértice k pertence ao caminho $v_i \rightsquigarrow^{\cdots k} v_j$. Observe que para isso, apenas a k-ésima linha e k-ésima coluna precisam estar disponíveis no processador, pois, são necessários os elementos w_{ik} (estes são todos os elementos da k-ésima coluna da matriz) e também os elementos w_{kj} (estes são todos os elementos da k-ésima linha da matriz). A aresta gerada, em uma rodada de computação, que pertence a um outro processador é enviada na próxima rodada de comunicação. O algoritmo termina em duas situações:

- Após uma rodada de computação, se nenhum processador gerar uma aresta perten-

cente a outro processador.

- Após uma rodada de comunicação, se nenhum processador receber uma aresta que deva ser inserida no digrafo (aresta nova).

O Algoritmo 5 apresenta o algoritmo de *Alves et al.* [2].

Algoritmo 5: FTP-*Alves et al.*

Entrada: W = Matriz de adjacências armazenada em p processadores: cada processador $q(1 \leq q \leq p)$ armazena as submatrizes $W[(q-1)\frac{n}{p}+1..q\frac{n}{p}][1..n]$ e $W[1..n][(q-1)\frac{n}{p}+1..q\frac{n}{p}]$

Saída: W = Matriz de adjacências do fecho transitivo do digrafo D distribuída pelos p processadores.

1: **repita**
2: **para** $k \leftarrow (q-1)\frac{n}{p}$ até $q\frac{n}{p}$ **faça**
3: **para** $i \leftarrow 0$ até $n-1$ **faça**
4: **para** $j \leftarrow 0$ até $n-1$ **faça**
5: **se** $w_{ik} = 1$ e $w_{kj} = 1$ **então**
6: $w_{ij} \leftarrow 1$(se w_{ij} pertence a um processador diferente de q, armazene e envie na próxima rodada de comunicação.)
7: **fim se**
8: **fim para**
9: **fim para**
10: **fim para**
11: Envie os dados correspondentes a outros processadores.
12: Receba os dados pertencentes ao processador q.
13: **até** que nenhuma aresta seja criada.

Pelo fato de não utilizar a rotulação de uma extensão linear na distribuição dos vértices entre os processadores, o limite de $\lg p$ rodadas de comunicação do Teorema 1 não pode ser aplicado, sendo necessário, no pior caso, $p-1$ rodadas. No entanto, os experimentos realizados pelos autores para digrafos gerados aleatoriamente, $\lg p + 1$ rodadas foram suficientes para o cálculo do fecho transitivo e os tempos obtidos apresentaram bons *speed-ups*. Um exemplo que requer mais de $\lg p + 1$ rodadas de comunicação e foi descrito por *Alves et al.*. Considere p=4 e um digrafo que é uma lista linear com n=16 vértices (cada vértice é rotulado com dois dígitos): $11 \rightarrow 31 \rightarrow 41 \rightarrow 21 \rightarrow 32 \rightarrow 12 \rightarrow 42 \rightarrow 33 \rightarrow 22 \rightarrow 43 \rightarrow 13 \rightarrow 23 \rightarrow 34 \rightarrow 14 \rightarrow 24 \rightarrow 44$. Assuma que cada vértice rotulado com os dígitos ij está armazenado no processador i. O Algoritmo 5 computa o fecho transitivo deste digrafo utilizando mais que $\lg p + 1 = 3$ rodadas de comunicação.

Teorema 2 *O Algoritmo 5 computa o fecho transitivo de um digrafo com complexidade de tempo* $\Theta(\frac{n^3}{p})$, $O(p)$ *rodadas de comunicação e requer* $O(\frac{n^2}{p})$ *memória em cada processador.*

Prova [2] ∎

2.3 Discussão

Vários algoritmos seqüenciais e paralelos foram propostos nos últimos anos para computar o fecho transitivo. Para digrafos acíclicos, *Habib et al.* [13], *Koubková e Koubek* [19] e *Simon* [28] apresentaram algoritmos seqüenciais com complexidade $O(nm)$. Para digrafos densos, o fecho transitivo pode ser computado utilizando multiplicação de matrizes em tempo $O(n^{2.376})$ [7]. Com complexidade $O(n^3)$ *Warshall* [34] apresentou um algoritmo seqüencial para computar o fecho transitivo. Embora possua complexidade maior, este algoritmo pode ser facilmente implementado e possui uma característica que favorece sua paralelização: no k-ésimo estágio, apenas a k-ésima linha e a k-ésima devem estar presentes no processador. Utilizando uma matriz de bits para representar a matriz de adjacências, *Baase* [3] descreveu uma maneira de melhorar o algoritmo de *Warshall* [34], diminuindo sua complexidade para $O(\frac{n^3}{\alpha})$, onde α é a quantidade de bits que podem ser armazenados em um tipo de dado primitivo.

Cáceres et al. apresentaram um algoritmo BSP/CGM com complexidade $O(\frac{nm}{p})$ de computação e $O(\lg p)$ rodadas de comunicação. Este algoritmo distribui os vértices aos processadores baseado em uma rotulação obtida através da computação de uma extensão linear do digrafo. No entanto, não é conhecido um algoritmo BSP/CGM que compute uma extensão linear de um digrafo em tempo $O(\frac{n+m}{p})$ com p processadores. Considerando que o algoritmo do fecho transitivo apresentado por *Cáceres et al.* é $O(\frac{nm}{p})$, o passo 1 pode ser computado seqüencialmente por um dos processadores. Baseados no algoritmo de *Cáceres et al.* e no algoritmo seqüencial de *Warshall*, *Alves et al.* apresentaram um novo algoritmo BSP/CGM para computar o fecho transitivo de um digrafo com complexidade de tempo $\Theta(\frac{n^3}{p})$ e $O(p)$ rodadas de comunicação. Este algoritmo utiliza a numeração inicial do digrafo para realizar da distribuição das arestas aos processadores. Mesmo realizando a distribuição desta maneira, para todos os digrafos utilizados nos experimentos realizados por *Alves et al.*, o fecho transitivo foi computado em até $\lg p + 1$ rodadas de comunicação.

Capítulo 3

Novos Algoritmos e Outras Contribuições

Neste capítulo, apresentaremos duas estratégias para diminuir o computação local e o tamanho das mensagens trocadas nos algoritmos propostos por *Alves et al.* [2] e *Castro Jr.* [6]. Além disso, apresentamos dois novos algoritmos BSP/CGM para computar o fecho transitivo de um digrafo. O primeiro algoritmo evita computar arestas já existentes no digrafo. O segundo algoritmo evita a troca de mensagens entre os processadores.

3.1 Avaliação dos Algoritmos de *Alves et al.* e *Castro Jr.*

Fizemos uma avaliação detalhada dos algoritmos propostos por *Alves et al.* [2] e *Castro Jr.* [6]. Executamos a implementação destes algoritmos a fim de determinar a quantidade de arestas geradas (**G**), a quantidade de arestas geradas que pertencem a outros processadores (**PE**), a quantidade de arestas enviadas (**E**), a quantidade de arestas recebidas (**R**), a quantidade de arestas recebidas que o processador receptor não possuía (**RN**) e também a quantidade de arestas duplicadas (**RC**). As arestas duplicadas são as arestas recebidas que o processador receptor já continha ou que vários processadores enviaram na mesma rodada de comunicação. Uma mesma aresta e_{ij} pode ser enviada ao processador p_k por p-1 outros processadores. Uma arestas é enviada pelo mesmo processador em apenas uma rodada de comunicação. Seja $\Phi_q = \{(q)\frac{n}{p}+1, (q)\frac{n}{p}+2, \ldots, (q)\frac{n}{p}+\frac{n}{p}\}$ o conjunto de vértices que o processador q utiliza como vértices intermediários na computação do fecho transitivo segundo a idéia do algoritmo seqüencial de *Warshall*. Em cada rodada de computação o processador p_k pode gerar até quatro tipos de arestas e_{ij}:

a) $\{i, j\} \in \Phi_k$: Esta aresta não deve ser enviada para outro processador.

b) $i \in \Phi_k$ e $j \notin \Phi_k$: Esta aresta deve ser enviada para outro processador.

c) $i \in \Phi_z$ e $j \in \Phi_x, z \neq x \neq k \neq z$: Esta aresta deve ser enviada para os processadores p_z e p_x.

d) $\{i, j\} \in \Phi_z, z \neq k$: Esta aresta deve ser enviada para o processador p_z.

A Tabela 3.1 apresenta a quantidade de arestas envolvidas em cada uma das rodadas de computação e comunicação utilizando um digrafo com 512 vértices e 53.000 arestas com 2, 4, 8, 16 e 32 processadores. Podemos observar que a quantidade de arestas envolvidas em uma rodada de comunicação é muito grande. Além disso, a quantidade de arestas (**G**) é sempre maior que a quantidade de arestas (**AP**), pois, apenas as arestas do tipo *b*, *c* e *d* devem ser enviadas. Quando a quantidade de arestas (**RN**) é igual a zero, significa que nenhum processador gerou arestas novas e o fecho transitivo está computado.

Procs.	**Rod.**	**G**	**PE**	**E**	**R**	**RN**	**RC**
2	1	261632	196352	196352	196352	0	196352
4	1	261632	245376	343680	343680	0	343680
8	1	261120	257080	428736	428736	64	428672
	2	266479	257535	429567	429567	0	429567
16	1	238517	237605	433543	433543	2690	430853
	2	258941	258029	473069	473069	0	473069
32	1	118191	117996	225907	225907	9301	216606
	2	252331	252236	490316	490316	0	490316

Tabela 3.1: Quantidade de arestas envolvidas na computação e comunicação

Os valores estão muito próximos pelo fato de estarmos apresentando a média das quantidades. Os resultados apresentados pela Tabela 3.1 foram obtidos utilizando o mesmo grafo dos resultados apresentados no próximo capítulo. Este grafo é denso e por isso requer poucas rodadas de comunicação.

3.2 Diminuição da Computação Local e do Tamanho da Mensagem

Para diminuirmos a computação do algoritmo de *Alves et al.* [2], utilizamos a idéia descrita por *Baase* [3]. Como mencionado anteriormente, na k-ésima iteração do laço mais externo, o algoritmo de *Warshall* verifica quais vértices v_i possuem uma aresta para o vértice v_k ($v_i \rightarrow v_k$) e cria uma aresta w_{ij} ($v_i \rightarrow v_j$), onde j pertence ao conjunto de vértices para os quais v_k possui uma aresta ($v_k \rightarrow v_j$). As linhas $\overline{w_i}$ e $\overline{w_k}$ representam os vértices nos quais v_i e v_k incidem, respectivamente. Para criarmos as arestas w_{ij} de v_i para v_j que possuem v_k como vértice intermediário basta realizar um OU lógico entre as linhas i e k. Esta foi a idéia em [3] para melhorar o algoritmo de *Warshall* e que utilizamos para melhorar o algoritmo de *Alves et al.* [2]. Para utilizarmos esta estratégia, distribuímos a matriz de adjacências (em bits) W do digrafo D inteira para todos os processadores. Cada posição

w_{ij} de W representa a existência da aresta e_{ij}. A matriz W possui n linhas e $\frac{n}{\alpha}$ colunas, onde α é a quantidade de bits capaz de ser armazenado em um tipo de dado primitivo. Conseqüentemente, a matriz de adjacência é representada utilizando $O(\frac{n^2}{\alpha})$ espaço. Cada aresta é armazenada em um bit.

Como descrito anteriormente, seja $\Phi_q = \{(q)\frac{n}{p}+1, (q)\frac{n}{p}+2, \ldots, (q)\frac{n}{p}+\frac{n}{p}\}$ o conjunto de vértices que o processador q utiliza como vértices intermediários na computação do fecho transitivo segundo a idéia do algoritmo seqüencial de *Warshall*. Cada processador p_q fica responsável por computar o fecho transitivo dos $\frac{n}{p}$ vértices pertencentes ao conjunto Φ_q. As arestas geradas em uma rodada de computação são enviadas na próxima rodada de comunicação. Apenas as arestas compartilhadas devem ser enviadas.

Como os todos os processadores armazenam a matriz de adjacências inteira, cada processador p_q computa uma extensão linear $L = \{l_1, l_2, l_3, ..., l_n\}$, $l_i \in V$, dos vértices de D e atribui para Φ_q o conjunto de $\frac{n}{p}$ vértices consecutivos de L ($\Phi_q = \{l_{q\frac{n}{p}+1}, l_{q\frac{n}{p}+2}, ..., l_{q\frac{n}{p}+\frac{n}{p}}\}$, $0 \leq q < p$). A extensão linear L pode ser computada em tempo $O(n+m)$ [31].

Durante as rodadas de comunicação o processador p_q envia as arestas geradas na z-ésima faixa vertical e na z-ésima faixa horizontal para o processador p_z. Para cada aresta w_{ij}, $0 \leq i, j < n$, devem ser enviados os valores de i e j (linha e coluna). As arestas podem ser enviadas de duas maneiras:

i) Para cada aresta, enviamos um único valor a computado como sendo $a = i * n + j$. A memória necessária para enviar cada aresta para um outro processador é c bits.

ii) Enviamos a i-ésima faixa vertical e a i-ésima faixa horizontal inteiras para o processador p_i armazenando cada aresta em um bit. O processador p_q envia $O(\frac{n^2}{p})$ bits para cada processador p_z, $z \neq q$.

Seja Q^q a quantidade de arestas geradas e que devem ser enviadas pelo processador p_q ao processador p_w, $w \neq q$. Antes de cada rodada de comunicação, verificamos se o tamanho da mensagem é menor enviando apenas as arestas geradas ou as faixas inteiras. Isto, é enviamos apenas as arestas geradas quando $Q^q < \frac{n^2}{c}$. Esta é a estratégia que estamos apresentando para diminuirmos o tamanho das mensagens trocadas entre os processadores.

A Tabela 3.2 apresenta a quantidade máxima de arestas que o processador p_q pode enviar em uma rodada de comunicação sem que haja necessidade de enviar as faixas inteiras. Para estes valores utilizamos $c = 32$. Para computar D^t de D com $n = 512$, são enviadas apenas as arestas geradas (e que pertencem a outros processadores) se $Q^q < 8192$. Caso contrário, são enviadas as faixas horizontal e vertical inteiras.

O Algoritmo 6 ilustra o algoritmo de *Alves et al.* incluindo nossas estratégias para diminuir a computação local e também o tamanho das mensagens trocadas entre os processadores.

Algoritmo 6: FTP-Bits

Entrada: D representado por uma matriz de adjacências de bits W de tamanho $\frac{n^2}{\alpha}$; p o número de processadores; e cada processador p_z $(0 \leq z \leq p-1)$ armazena uma cópia de W.

Saída: D^t representado como uma matriz W^t distribuída por p processadores.

1: **se** z=0 **então**
2: Compute L e $\Phi_{0,\ldots,p-1}$
3: **fim se**
4: **repita**
5: **para todo** $k \in \Phi_z$ em ordem crescente **faça**
6: **para** $i \leftarrow 1$ **até** n **faça**
7: **se** $w_{ik} = 1$ **então**
8: **para** $j = 1$ **até** $\frac{n}{\alpha}$ **faça**
9: $w_{ij} \leftarrow w_{ij} \vee w_{kj}$
10: **fim para**
11: **fim se**
12: **fim para**
13: **fim para**
14: **se** $Q^k > \frac{n^2}{c}$ **então**
15: Envie/Receba somente as arestas geradas para/de outros processadores
16: **senão**
17: Envie/Receba as faixas para/de outros processadores
18: **fim se**
19: **até** que nenhuma aresta pertencente a outro processador seja gerada
20: **retorne** D^t

Número	**Qtde**
128	512
256	2048
512	8192
1024	32768
2048	131072
...	...
n	$\frac{n^2}{c}$

Tabela 3.2: Valores de Q^q em relação a quantidade de arestas.

A complexidade de tempo do Algoritmo 6 é determinada pelos três laços aninhados. O primeiro laço possui $\frac{n}{p}$ iterações. O segundo laço possui n iterações. O terceiro laço possui $\frac{n}{\alpha}$ iterações. Conseqüentemente, a complexidade é dada por: $\frac{n}{p} * n * \frac{n}{\alpha} = O(\frac{n^3}{p\alpha})$. Pelo fato de Φ_q ser computado utilizando a rotulação obtida através de uma extensão linear L de D, $O(\lg p)$ rodadas de comunicação são suficientes para computar o fecho transitivo de um digrafo acíclico. Em uma rodada de comunicação o processador p_q envia no máximo $O(\frac{n^2}{p})$ bits para o processador p_z, $q \neq z$.

Teorema 3 *O Algoritmo 6 computa o fecho transitivo de um digrafo acíclico em* $O(\frac{n^3}{p\alpha})$ *de computação local com* $O(\lg p)$ *rodadas de comunicação e* $O((MAX(n+m, \frac{n^2}{\alpha}))$ *espaço.*

Prova A complexidade de computação é α vezes mais rápida que a do Algoritmo 5 pelo fato de realizarmos um OU lógico entre as linhas $\overline{w_i}$ e $\overline{w_i}$. Nosso algoritmo utiliza no máximo $O(\lg p)$ rodadas de comunicação para computar o fecho transitivo pelo fato de distribuir os vértices aos processadores segundo a idéia do Algoritmo 4. A complexidade de espaço local é a complexidade de espaço necessário para computar L e armazenar W. Para grafos esparsos, a espaço necessário para computar a extensão linear é menor que $O(\frac{n^2}{\alpha})$. Para grafos densos, a computação da extensão linear requer espaço maior que $O(\frac{n^2}{\alpha})$. Conseqüentemente, o Algoritmo 6 requer $O((\text{MAX}(n+m, \frac{n^2}{\alpha}))$ espaço local. ∎

Os resultados obtidos através da implementação destas estratégias são apresentados no Capítulo quatro.

3.3 Diminuição do Tamanho das Mensagens

Nas implementações apresentadas por *Alves et al.* [2] e *Castro Jr.* [6] o custo relacionado ao envio das arestas geradas é muito grande (apresentado na Tabela 3.1). Isso acontece por que não é feito nenhum tipo de tratamento antes do envio. Utilizamos a estratégia de diminuição do tamanho das mensagens trocadas entre os processadores no algoritmo de *Alves et al.* [2]. O Algoritmo 7 apresenta o algoritmo incluindo esta estratégia.

Algoritmo 7: FTP-Alves et al. - II

Entrada: D representada por uma matriz W de tamanho n x n; p o número de processadores; e cada processador p_z $(0 \leq z \leq p-1)$ armazena a sub-matriz $W[(z\frac{n}{p}+1..(z+1)\frac{n}{p}][1..n]$ e $W[1..n][z\frac{n}{p}+1..(z+1)\frac{n}{p}]$

Saída: D^t representado por W^t distribuído pelos p processadores.

```
1:  repita
2:    para k ← l(n/p) + 1 até (l + 1)(n/p) − 1 faça
3:      para i ← 1 até n faça
4:        para j ← 1 até n faça
5:          se w_ik = 1 e w_kj = 1 então
6:            w_ij ← 1
7:          fim se
8:        fim para
9:      fim para
10:   fim para
11:   se Q^k > n^2/c então
12:     Envie/Receba somente as arestas geradas para/de outros processadores
13:   senão
14:     Envie/Receba as faixas para/de outros processadores
15:   fim se
16: até que nenhuma aresta pertencente a outro processador seja gerada
```

Teorema 4 *O Algoritmo 7 computa o fecho transitivo D^t do digrafo D no modelo BSP/CGM com complexidade $O(\frac{n^3}{p})$ de computação local com $O(p)$ rodadas de comunicação e requer $O(\frac{n^2}{p})$ espaço.*

Prova [6] ∎

Embora esta seja uma alteração bastante simples, os resultados experimentais foram bastante satisfatórios. Não alteramos a computação local, conseqüentemente, este algoritmo gera e envia as mesmas arestas geradas e enviadas no Algoritmo 5. Cada processador envia no máximo $\frac{n^2}{p}$ bits para os outros processadores.

A quantidade de rodadas de comunicação pode ser diminuída para $O(\lg p)$ se um processador computar uma extensão linear e utilizar a rotulação dos vértices para distribuí-los aos processadores, desde que, $p \leq \frac{n^2}{m+n}$. Isto é, a memória local do processador deve ser suficiente para armazenar a lista de adjacências inteira.

3.4 Diminuição da Computação Local Atualizando o Fecho Transitivo

Com o Algoritmo 6 diminuímos expressivamente a computação local do Algoritmo 5. Uma outra maneira para diminuirmos o trabalho local é evitando computar arestas já existentes no digrafo. No algoritmo 5 cada processador p_q é responsável por computar o fecho transitivo de um sub-digrafo local D_{lq}, $D_{lq} \subseteq D$, contendo todas as arestas de saída e entrada dos vértices existentes em Φ_q. Isto é, D_{lq} contém a q-ésima faixa vertical e a q-ésima faixa horizontal da matriz que representa o digrafo original. Ao final da primeira rodada de computação, cada processador p_q possui o fecho transitivo das arestas contidas no digrafo local. A complexidade da primeira rodada de computação é $\Theta(\frac{n^3}{p})$. A segunda rodada de computação tem a mesma complexidade, pois, é computado novamente o fecho transitivo para todos os vértices de D_{lq}.

Apresentamos um novo algoritmo para computar o fecho transitivo de um digrafo baseado nas idéias contidas no algoritmo apresentado por *Alves et al.* [2] e na manutenção do fecho transitivo após a inserção de uma aresta. Nosso algoritmo evita computar arestas que já existam no digrafo local. Após a primeira rodada de comunicação o processador p_q não precisa computar o fecho transitivo novamente para todos os vértices presentes em D_{lq} (não precisa executar novamente os três laços aninhados), basta adicionar as arestas recebidas durante a rodada de comunicação e atualizar o fecho transitivo local após a inserção de cada aresta. Utilizamos esta idéia para computar o fecho transitivo também na primeira rodada de computação. Retiramos todas as arestas de D_{lq} e em seguida, inserimos novamente atualizando o fecho transitivo após a inserção de cada aresta.

Algoritmos seqüenciais para manter o fecho transitivo após a inserção de uma aresta foram apresentados por *Italiano* [15] e *La Poutre* e *van Leeuwen* [21] com complexidade de tempo $O(n)$ amortizado.

Utilizamos o algoritmo do *Italiano* [15] para realizar a atualização do fecho transitivo. Seja Ψ um conjunto de arestas. Inicialmente, todas as arestas são inseridas em Ψ e são retiradas todas as arestas do sub-digrafo local D_{lq}. Em seguida, as arestas contidas em Ψ são inseridas em D_{lq} mantendo a propriedade de fecho transitivo.

Algoritmo 8 ilustra os passos do nosso algoritmo.

Utilizamos a computação de Q^q para diminuir o tamanho da mensagem trocada entre os processadores. Nosso algoritmo é eficiente no senso amortizado: uma particular operação pode ser lenta, mas, uma seqüência de operações deve ser mais rápida. A complexidade da computação local é $O(nr)$ amortizado, onde r é a quantidade de arestas recebidas pelo processador em uma rodada de comunicação e que devem ser inseridas no digrafo. Note que $r = O(\frac{n^2}{p})$, o tamanho das faixas vertical e horizontal. Em uma rodada de comunicação, vários processadores podem enviar a mesma aresta ao processador q, apenas uma é inserida no digrafo local. Além disso, essa aresta é inserida apenas se ainda não existir. Para a primeira rodada de computação, r é a quantidade de arestas do digrafo local inicial.

Algoritmo 8: FTP-Adicionando Arestas

Entrada: D_{lq}: grafo local representado pela q-ésima faixa vertical e a q-ésima faixa horizontal de W ($W[q\frac{n}{p}+1...q\frac{n}{p}][1...n]$ e $W[1...n][q\frac{n}{p}+1...q\frac{n}{p}]$), e_{ij} : é uma aresta de D_{lq}, $0 \leq q < p$.

Saída: W = Matriz de adjacências do fecho transitivo do digrafo D representada por D_{lq} distribuída pelos p processadores.

1: $\Psi \leftarrow$ Todas as arestas de D_{lq}.
2: $D_{lq} \leftarrow \emptyset$
3: **repita**
4: **para todo** $e_{ij} \in \Psi$ **faça**
5: Insira e_{ij} em D_{lq}
6: Utilize o algoritmo do *Italiano* para atualizar o fecho transitivo.
7: **fim para**
8: $\Psi \leftarrow \emptyset$
9: **se** $Q^q > \frac{n^2}{c}$ **então**
10: Envie/Receba somente as arestas geradas para/de outros processadores
11: **senão**
12: Envie/Receba as faixas para/de outros processadores
13: **fim se**
14: Insira as arestas recebidas, que não existem no digrafo, em Ψ.
15: **até** que nenhuma aresta pertencente a outro processador seja gerada

Teorema 5 *O Algoritmo 8 computa o fecho transitivo D^t do digrafo D no modelo BSP/CGM com complexidade $O(nr)$ amortizado, $O(p)$ e $O(\frac{n^2}{p})$ de computação, comunicação e espaço, respectivamente.*

Prova Seja D_q^i o sub-digrafo contido no processador q antes do início da i-ésima rodada de computação, $1 \leq i \leq p$. Logo, D_q^1 é o sub-digrafo atribuído ao processador q antes do início do algoritmo. Na primeira rodada computação, retire todas as aresta de D_{lq} e insira em D_q^1 (uma a uma) realizando a manutenção do fecho transitivo em D_q^1 após cada inserção. Na segunda rodada de comunicação, apenas as arestas recebidas que ainda não existem são inseridas em D_q^2, e assim por diante. O sub-digrafo local pode conter no máximo $\frac{n^2}{p}$ arestas. Conseqüentemente, no máximo $r = \frac{n^2}{p}$ vezes será realizada a atualização do fecho transitivo. As mesmas arestas do Algoritmo 5 são geradas e enviadas. ∎

O algoritmo apresentado por *Alves et al.* possui complexidade $O(\frac{n^3}{p})$ em todas as rodadas de computação. Nosso algoritmo, possui no pior caso complexidade $\frac{n^3}{p}$ amortizado. No entanto, esta complexidade está relacionada a quantidade de rodadas de comunicação. As arestas inseridas em uma rodada de computação não são inseridas em outra rodada. A diferença relacionada a quantidade de operações realizadas nos dois algoritmos em uma rodada de computação fica acentuada quando são necessárias várias rodadas de comunicação. Neste caso, nosso algoritmo tende a obter um desempenho melhor.

As arestas geradas e enviadas deste algoritmo são as mesmas geradas e enviadas pelo

algoritmo de *Alves et al.* [2]. Podemos relaxar a complexiade de espaço do Algoritmo 8 para $O(n^2)$ com o objetivo de computar a extensão linear L e realizarmos a distribuição dos vértices obedecendo a rotulação obtida em L. Neste caso, um processador computa a extensão e distribui os vértices aos outros processadores. Utilizando esta distribuição a complexidade de comunicação é $O(\lg p)$ rodadas [5].

3.5 Evitando a Troca de Mensagens Entre os Processadores.

O algoritmo e as estratégias que propusemos anteriormente diminuem a computação local e o tamanho das mensagens trocadas entre os processadores. Verificamos nas implementações descritas anteriormente, de *Alves et al.* [2], *Castro Jr.* [6] e na implementação dos nossos algoritmos (6 e 7), que a troca de mensagens compromete bastante o desempenho. Nesta seção, apresentamos um algoritmo BSP/CGM com complexidade $O(\frac{n}{p}(n+m))$ de computação local e sem rodada de comunicação. Este algoritmo utiliza as idéias contidas no algoritmo seqüencial 3 (*FT-Busca*). A lista de adjacências é distribuída inteiramente para todos os processadores. Desta forma, cada processador possui os dados necessários para computar o fecho transitivo. Para este algoritmo, denotamos Φ_q como sendo o conjunto de $\frac{n}{p}$ vértices que o processador q utiliza como vértices de origem. Através de uma busca em profundidade no digrafo encontramos todos os vértices j alcançáveis a partir de um vértice i, $i \in \Phi_q$ e criamos a aresta e_{ij}. A complexidade de espaço é $O(n+m)$.

O Algoritmo 9 apresenta os passos do nosso algoritmo.

Algoritmo 9: FTP-Busca

Entrada: Q = Lista de adjacências do digrafo D armazenada inteiramente em cada um dos p_q processador, p_q $(0 \leq q \leq p-1)$.

Saída: D^t = Fecho transitivo do digrafo D representado por Q^t, a lista de adjacências de D^t distribuída por p processadores: cada processador p_q possui a lista de adjacências dos vértices $v_i, i \in \Phi_q$

1: **para** $i \leftarrow l\frac{n}{p}+1$ até $(l+1)\frac{n}{p}$ **faça**
2: $\quad Q_i^t \leftarrow NULL$
3: $\quad$ Compute Γ^i.
4: $\quad$ Crie a aresta $q_{i,j}^t = \{e_{ij} : j \in \Gamma^i\}$
5: **fim para**

Considere um estágio do Algoritmo 9 como sendo a i-ésima iteração do laço da linha 1. Ao final deste estágio Γ^i possui todos os vértices aos quais v_i deve incidir no fecho transitivo final. Durante a computação do Algoritmo 9 as arestas geradas a partir de Γ^i não são utilizadas na computação das outras arvores. Por este motivo, nosso algoritmo não necessita de comunicação entre os processadores. O algoritmo que estamos propondo possui uma boa escalabilidade. No entanto, a computação realizada por um processador não é utilizada por outros, isto pode provocar um desbalanceamento na computação local.

O pior caso deste desbalanceamento é, por exemplo, na computação do fecho transitivo de um digrafo D onde D é uma lista. Isto é $D = (V, E)$ com $n = |V|$ e $m = |E|$ onde $n-2$ vértices possuem apenas uma aresta de saída e uma aresta de entrada, um vértice possui apenas uma aresta de entrada e um vértice possui apenas uma aresta de saída. Em um digrafo deste tipo, dependendo de como Φ_q for computado, um processador realiza uma computação muito maior que a computação realizada por outro processador. A carga de trabalho do processador p_q é dada pelo somatório da quantidade de vértices contidos em Γ^i, $\forall\ i \in \Phi_q$.

Ao final da computação o fecho transitivo está distribuído pelos processadores.

Teorema 6 *O Algoritmo 9 computa o fecho transitivo D^t do digrafo D no modelo BSP/CGM com complexidade $O(\frac{n}{p}(n+m))$, $O(n+m)$ e $O(1)$ de computação local, espaço e comunicação, respectivamente.*

Prova O conjunto Φ_q possui $\frac{n}{p}$ vértices. Para cada vértice presente em Φ_q é realizado uma busca em profundidade (ou largura) utilizando $O(n+m)$, conseqüentemente, a complexidade de comutação local é $O(\frac{n}{p}(n+m))$. A memória local é $O(n+m)$ pelo fato de armazenar a lista de adjacências inteira. ∎

Embora este algoritmo seja *simples*, não encontramos trabalhos que utilizem estas idéias. Os resultados experimentais são expressivos.

3.6 Discussão

Avaliamos as implementações dos algoritmos BSP/CGM de *Alves et al.* e *Castro Jr.* para computar o fecho transitivo de um digrafo. Verificamos a quantidade de arestas geradas, enviadas, recebidas por cada processador e também quais arestas eram geradas por processadores distintos em uma mesma rodada de computação. Para melhorar o desempenho destas implementações propusemos estratégias e algoritmos que diminuem o tamanho das mensagens trocadas entre os processadores, diminuem a computação local e evitam a troca de mensagens.

Utilizando as idéias do algoritmo de *Alves et al.* e uma matriz de bits para representar o digrafo, apresentamos uma estratégia para diminuir a computação local de $O(\frac{n^3}{p})$ para $O(\frac{n^3}{p\alpha})$, onde α é a quantidade de bits que podem ser armazenados em um tipo de dado primitivo. Nesta estratégia a matriz de adjacências de bits (de tamanho $\frac{n^2}{\alpha}$) é distribuída inteira para todos os processadores. Em uma rodada de comunicação, cada processador envia $\frac{n^2}{p}$ bits para cada processador. Este algoritmo utiliza o paralelismo intra-processador e inter-processador.

Alteramos o algoritmo de *Alves et al.* para limitar o tamanho das mensagens trocadas entre os processadores, cada processador envia no máximo $\frac{n^2}{p}$ bits aos outros processadores.

Após a primeira rodada de comunicação, cada processador possui o fecho transitivo parcial computado utilizando os vértices pertencentes ao conjunto Φ_q como vértices intermediários. A partir da segunda rodada de computação não é necessário computar novamente, basta, inserir as arestas recebidas e atualizar o fecho transitivo após a inserção de cada aresta. Isso pode ser feito também na primeira rodada de computação. Baseados nesta idéia, apresentamos um algoritmo BSP/CGM com $O(nr)$ amortizado de computação local, r é a quantidade de arestas recebidas e que devem ser inseridas no digrafo. Este algoritmo requer $O(p)$ rodadas de comunicação. As arestas inseridas em uma rodada de computação não são inseridas em outras rodadas.

Geralmente, em uma implementação paralela, o custo relacionado à comunicação é o grande delimitador do tempo no total. Baseado em uma busca em profundidade (ou em largura) no digrafo, apresentamos um algoritmo com complexidade $O(\frac{n}{p}(n+m))$ de computação sem rodadas de comunicação. Para evitar as comunicações, distribuímos a lista de adjacências para todos os processadores. A complexidade de memória é $O(n+m)$. Este algoritmo pode apresentar um desbalanceamento da computação local em alguns casos.

Capítulo 4

Implementações

Neste capítulo descrevemos os resultados obtidos a partir das implementações das estratégias e algoritmos apresentados no capítulo anterior. Nossas implementações foram executadas em dois clusters Beowulf. O primeiro com 64 nós consistindo de microcomputadores de baixo custo contendo 256MB de RAM, 256MB de memória *swap*, UCP Intel Pentium III 448.956 MHz, 512KB cache. Além disso, dois nós de acesso consistindo de dois microcomputadores, cada um contendo 512MB de RAM, 512MB de memória *swap*, UCP Intel Pentium IV 2.00 GHz e 512KB cache. Este também foi o cluster utilizado na implementação de *Alves et al.* e *Castro Jr.* O cluster está dividido em dois blocos de 32 nós cada. Os nós de cada bloco estão conectados através de um *switch* de 100 MB. Cada um dos nós de acesso está conectado ao *switch* que conecta aos blocos. O segundo cluster é constituído de 12 nós, sendo 6 microcomputadores Pentium 4 de 1.7GHz e 6 microcomputadores AMD Athlon de 1.6GHz. Os nós deste cluster são conectados por um *switch* de 1 Gb. Nossas implementações foram feitas utilizando ANSI C++ e a biblioteca LAM-MPI[23, 29].

4.1 Gerador de Digrafos

Os digrafos utilizados nos testes foram gerados aleatoriamente com probabilidade de existir uma aresta entre dois vértices variando de 15% à 20%. Implementamos um gerador de digrafos a partir da implementação feita por *Castro Jr.* [6]. Incluímos as funcionalidades de geração de digrafos acíclicos e conexos. Além disso, nosso gerador recebe como argumento a quantidade de arestas do digrafo ou define uma quantidade aleatoriamente.

4.2 Resultados Obtidos

Implementamos os algoritmos 6, 7 e 9. Além disso, fizemos uma avaliação dos tempos de comunicação dos Algoritmos 5 e 7. Para observar melhor a eficiência de nossos algoritmos, comparamos nossos resultados com os apresentados por *Alves et al.*. A Tabela 4.1

apresenta os resultados obtidos por eles para digrafos com 512, 1024 e 1920 vértices. Não há registro da quantidade de arestas dos digrafos e também da quantidade de rodadas de comunicação envolvidas na computação do fecho transitivo. A quantidade de arestas do digrafo é fundamental para determinar o tempo das implementações, pois, quanto menor a quantidade de arestas, maior é o número de rodadas de comunicação necessárias para computar o fecho transitivo. Ao longo desta seção apresentaremos nossos resultados.

Procs.	**512**	**1024**	**1920**
1	25.4	143.2	1614
2	9.3	123.1	603
4	8.3	84.4	257
8	3.9	36.4	123
16	2.6	18.0	69
32	1.9	15.6	68
64	3.3	12.1	49

Tabela 4.1: Melhores resultados apresentados por Alves et al. [2]

Os tempos que estamos apresentando utilizando apenas um processador representam somente o tempo de computação, não existe comunicação nenhuma envolvida.

4.2.1 Algoritmo *FTP-BITS*

O Algoritmo 6 utiliza o pipeline do processador para melhorar a computação local. Durante a implementação utilizamos uma matriz de adjacências onde cada elemento é representado por um caracter. Conseqüentemente, cada elemento reflete a informação de até oito arestas ($\alpha = 8$) do digrafo. Com isto, $\frac{n}{\alpha}$ operações lógicas de OU entre as linhas w_i e w_k são realizadas, $\forall\ i, k \in \{1, 2, ..., n\}$.

Como mencionado anteriormente, o processador q envia no máximo $\frac{n^2}{p}$ bits para cada processador k, $0 \leq k < p$. Isso se deve ao fato de computarmos Q^q. A Tabela 4.2 apresenta os resultados, em segundos, obtidos através da implementação do Algoritmo 6, enviando apenas as arestas geradas (**A**), computando Q^q (**B**), enviando as faixas horizontal e vertical inteiras (**C**) no primeiro cluster. Além disso, apresenta também os resultados obtidos no segundo cluster computando Q^q (**D**). Utilizamos digrafos com 512, 1.024 e 2.048 vértices e 53.000, 210.000 e 800.000 arestas, respectivamente.

Através da Tabela 4.2 é possível observar que quando a quantidade de rodadas de comunicação é aumentada, o desempenho diminui. Para digrafos densos, nossa implementação apresenta speedup quase linear. Isso acontece pelo fato da computação local ser expressiva mesmo quando utilizamos uma quantidade grande de processadores. Isso não acontecerá se aumentarmos o tamanho de α, pois, diminuiremos a carga de trabalho em cada processador.

A implementação do Algoritmo 6 é mais complexa que a implementação dos algoritmos 5, 7 e 9.

	512 x 512				1024 x 1024				2048 x 2048			
Procs.	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
1	0.9	0.9	0.9	0.3	7.4	7.4	7.4	2.1	61.5	61.5	61.5	17.2
2	0.8^{1}	0.7	0.5^{1}	0.2	5.1^{1}	4.1	3.9^{1}	1.3	36.2^{1}	33.0	31.6^{1}	10.0
4	0.7^{1}	0.5	0.3^{1}	0.1	3.9^{1}	2.9	2.0^{1}	0.7	23.2^{1}	18.9	15.9^{1}	5.3
8	0.7^{1}	0.3	0.1^{1}	0.0	1.9^{1}	1.2	1.0^{1}	0.3	16.6^{1}	12.8	8.1^{1}	2.7
16	1.1^{2}	0.4	0.2^{2}	-	1.9^{1}	1.0	0.8^{1}	-	8.9^{1}	5.1	4.6^{1}	-
32	1.1^{2}	0.7	0.5^{2}	-	2.5^{2}	1.5	1.4^{2}	-	5.4^{1}	3.8	2.4^{1}	-

Tabela 4.2: Resultados obtidos a partir do implementação do Algoritmo **FTP-BITS**.

Pelo fato da implementação do algoritmo seqüencial de *Warshall* utilizando bits ser muito mais rápida que a implementação sem bits, é difícil obtermos uma implementação paralela eficiente e escalável. Em um determinado momento a computação fica muito rápida e os tempos são delimitados principalmente pelas trocas de mensagens.

Quando aumentamos o número de processadores, diminuímos a carga de trabalho de cada um deles. Além disso, a nova configuração das arestas armazenadas em cada processador, pode propiciar uma quantidade maior de rodadas de comunicação necessárias para computar o fecho transitivo. Conseqüentemente, o ganho obtido durante a computação local é menor que o custo agregado pela necessidade de mais uma rodada. Isso acontece sempre que uma nova rodada for necessária. A matriz de adjacências pode ser representada por qualquer tipo de dado primitivo. Quanto maior a quantidade de bits do tipo de dado, mais rápida é a computação local. O tempo gasto com a troca de mensagens não é alterado, pois, cada aresta ainda é representada por um bit.

A Figura **4.1** apresenta o tempo de comunicação utilizado pelo Algoritmo 5 (I) e o tempo utilizado pelo algoritmo 7 (II) no primeiro cluster (PC) e no segundo cluster (SC). Quando aumentamos a quantidade de processadores, a quantidade de arestas que devem ser enviadas a mais de um processador aumenta, conseqüentemente, o tempo de comunicação também. Os valores de I-PC e II-SC possuem pouca variação pelo fato de enviarmos mensagens sempre do mesmo tamanho. Os valores foram obtidos para um grafo com 2048 vértices. O tempo de comunicação utilizado pelo Algoritmo 7 é bem menor.

Os resultados obtidos foram significativamente melhores que os resultados de outros autores, tais como os apresentados por *Castro Jr.* [6], *Alves et al.* [2], *Pagourtzis et al.* [24, 25].

4.2.2 Algoritmo *FTP-Alves et al.-II*

Os resultados obtidos através da implementação do Algoritmo 7 mostraram que a idéia de limitar o tamanho das mensagens contribui positivamente ao desempenho do algoritmo. Nossa implementação é semelhante à implementação apresentada por *Alves et al.*. A única diferença está no envio das arestas aos outros processadores. A Tabela 4.3 apresenta os resultados obtidos a partir da implementação do Algoritmo 7 no primeiro cluster (**A**) e

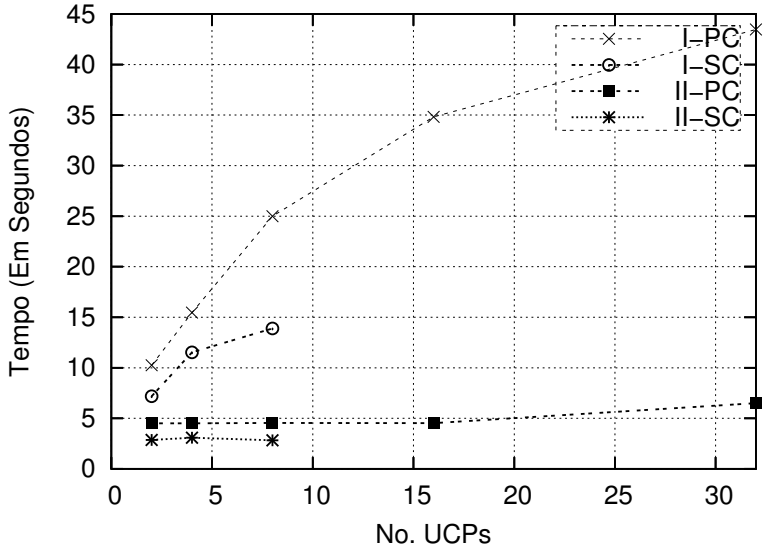


Figura 4.1: Comparação entre os tempos de comunicação do Algoritmo 5 e 7.

no segundo cluster (**B**). Computamos Q^k antes da troca de mensagens.

	1024		**2048**		**4096**	
Procs.	**A**	**B**	**A**	**B**	**A**	**B**
1	86	23	688	190	5300	1521
2	44	12	348	100	2780	788
4	22	6	176	50	1427	407
8	12	3	91	27	723	206
16	6	-	48	-	377	-
32	5	-	26	-	205	-

Tabela 4.3: Algoritmo 7 computando Q^k (tempo em segundos.).

Utilizamos digrafos com 1.024, 2.048 e 4096 vértices e 210.000 e 800.000 e 3.300.000 arestas, respectivamente.

4.2.3 Algoritmo *FTP-Busca*

Grande parte do tempo total das três implementações que apresentamos anteriormente é referente à troca de mensagens entre os processadores. O Algoritmo 9 evita a troca de mensagens utilizando uma busca em profundidade (ou largura) no digrafo. Para isso, a lista de adjacência está inteira em todos os processadores.

Os resultados obtidos através da implementação do Algoritmo 9 comprovam que, embora possa haver um desbalanceamento de computação local, a ausência de comunicação entre os processadores produz resultados muito bons. Utilizamos uma busca em profundidade para computar os vértices alcançáveis a partir do vértice v_i. A Tabela 4.4 apresenta

os resultados obtidos, em segundos, para digrafos com 512, 1.024, 2.048, 4.096 e 6.144 vértices e 53.000, 210.000, 800.000, 2.000.000 e 4.000.000 arestas, respectivamente. Os resultados foram obtidos no primeiro (**A**) e no segundo (**B**) cluster.

Procs.	**512 x 512**		**1024 x 1024**		**2048 x 2048**		**4096 x 4096**		**6144 x 6144**	
	A	**B**	**A**	**B**	**A**	**B**	**A**	**B**	**A**	**B**
1	3.7	1.7	30.5	13.1	232.9	97.7	1253.5	512.7	4962.8	1908.5
2	1.8	0.9	15.2	6.9	116.5	51.7	626.5	268.8	2482.4	912.6
4	0.9	0.5	7.6	3.3	58.3	24.7	313.3	142.5	1241.4	455.3
8	0.4	0.3	3.8	2.0	29.2	13.1	156.6	75.3	620.8	228.1
16	0.2	-	1.9	-	14.6	-	78.5	-	310.3	-
32	0.1	-	0.9	-	7.3	-	39.8	-	155.2	-
64	0.06	-	0.48	-	3.64	-	19.6	-	78.26	-

Tabela 4.4: Resultados do Algoritmo *FTP-Busca*

A Figura **4.2** apresenta o gráfico obtido para digrafos com 4096 e 6144 vértices com 2.000.000 e 4.000.000 arestas respectivamente. Gráfico gerado utilizando os valores presentes na Tabela 4.4.

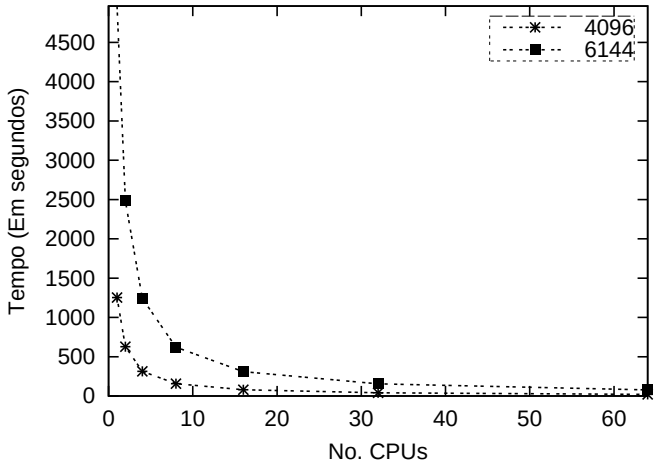


Figura 4.2: Resultados para digrafos maiores: 4096 e 6144 vértices com 2.000.000 e 4.000.000 arestas respectivamente

Os resultados obtidos são melhores que obtidos pelas implementações do Algoritmo 7. Seu desempenho só não é melhor que o desempenho do Algoritmo 6, mas, aproxima-se em algumas situações. No entanto, a implementação do Algoritmo 9 é bem mais simples.

4.3 Discussão

Neste capítulo apresentamos os ambientes nos quais realizamos nossos experimentos e também os resultados obtidos através da implementação dos algoritmos e idéias que es-

tamos propondo. Os digrafos foram gerados de maneira aleatória. Utilizamos o mesmo gerador e o primeiro cluster é mesmo o cluster utilizado nos experimentos de *Castro Jr* e *Alves et al.*.

Os resultados obtidos foram melhores que os apresentados anteriormente por outros autores. A Tabela 4.5 ilustra um comparativo entre os resultados obtidos por *Alves et al.* (**A**-Algoritmo 5) e os nossos resultados (**B**-Algoritmo 6, **C**-Algoritmo 7 e **D**-Algoritmo 9) para digrafos com 2048 vértices e 800000 arestas.

	2048 x 2048			
Procs.	**A***	**B**	**C**	**D**
1	1614	61.5	688	232.9
2	603	31.6	348	116.5
4	257	15.9	176	58.2
8	123	8.1	91	29.1
16	69	4.6	48	14.6
32	68	2.4	26	7.3

Tabela 4.5: Comparação entre os resultados de *Alves et al.* e nossos resultados (*com 1920 vértices). Tempo em segundos

A Figura apresenta a representação gráfica dos resultados contidos na Tabela 4.5.

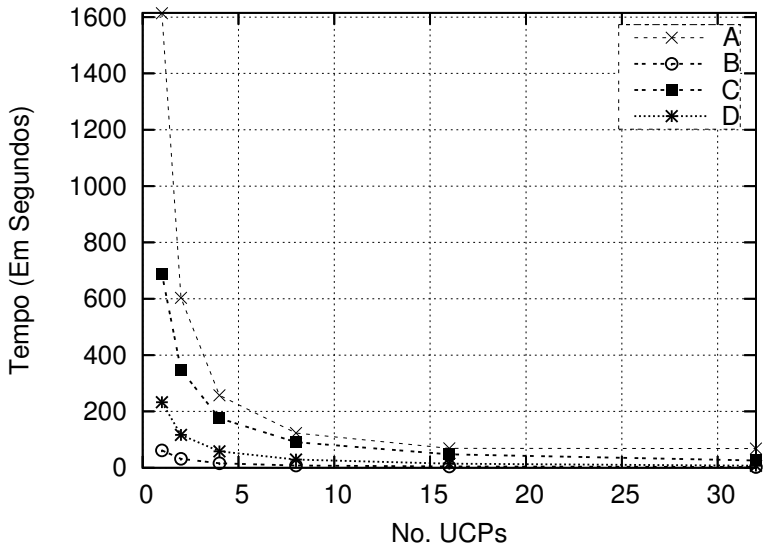


Figura 4.3: Representação gráfica da Tabela 4.5

Nosso melhor resultado foi obtido através do Algoritmo 6, em algumas situações, mais de vinte vezes mais rápido que os resultados obtidos por *Alves et al.*. Para digrafos maiores e densos, este algoritmo apresentou *speed-ups* quase lineares.

Mesmo podendo haver um desbalanceamento de carga e não existir a reutilização da computação já feita por algum outro processador, a implementação do Algoritmo 9

apresentou resultados também expressivamente melhores que os apresentados por outros autores.

Capítulo 5

Conclusões

Avaliamos as implementações do algoritmo BSP/CGM de *Cáceres et al.* apresentadas por *Alves et al.* [2] e *Castro Jr.* [6] para computar o fecho transitivo de um digrafo. Verificamos os tempos relacionados à computação local e à troca de mensagens. Melhoramos o desempenho destas implementações, diminuindo o tamanho das mensagens trocadas entre os processadores, a computação local e a quantidade de rodadas de comunicação entre os processadores.

Apresentamos dois algoritmos para diminuir a computação local. Utilizando uma matriz de bits para representar o digrafo e utilizando as idéias do algoritmo de *Alves et al.*, apresentamos um algoritmo BSP/CGM com complexidade de computação $O(\frac{n^3}{p\alpha})$ e $O(\log p)$ rodas de comunicação. Neste algoritmo a matriz de adjacências de bits é distribuída inteiramente a todos os processadores. Limitamos o tamanho das mensagens trocadas entre os processadores em $O(\frac{n^2}{p})$ bits, utilizando apenas um bit para representar cada aresta. A implementação deste algoritmo obteve, em algumas situações, tempo superior a vinte vezes mais rápido que os resultados de *Alves et al.*.

Com o objetivo de diminuir a computação local, evitamos computar arestas já existentes no digrafo. Cada aresta recebida em uma rodada de comunicação é inserida no digrafo e em seguida atualizamos o digrafo para que represente novamente o fecho transitivo. Baseados na manutenção do fecho transitivo após a inserção de uma aresta, apresentamos um algoritmo BSP/CGM com $O(nr)$ amortizado de computação local, r é a quantidade de arestas recebidas e que devem ser inseridas no grafo. Cada processador possui $O(\frac{n^2}{p})$ memória local. A complexidade de comunicação deste algoritmo é $O(p)$ rodadas.

Alteramos a implementação apresentada por *Alves et al.* para que cada processador envie no máximo $\frac{n^2}{p}$ bits aos outros processadores. Isso é feito armazenando cada arestas em apenas um bit. Os resultados obtidos a partir da implementação desta idéia foram significativamente melhores que os originais apresentados por *Alves et al.*. A complexidade de comunicação deste algoritmo é $O(p)$ rodadas de comunicação.

Verificamos que o custo relacionado à comunicação é o grande delimitador do tempo total da computação do fecho transitivo. Baseado em uma busca em profundidade (ou

largura), apresentamos um algoritmo com complexidade $O(\frac{n}{p}(n+m))$ de computação sem rodadas de comunicação. Para evitar as comunicações, distribuímos a lista de adjacências para todos os processadores. A complexidade de memória é $O(n+m)$. Mesmo podendo haver um desbalanceamento de carga e não existir a reutilização da computação já feita por algum outro processador, a implementação do Algoritmo que utiliza a busca em um digrafo apresentou resultados também expressivamente melhores que os apresentados por outros autores.

Dois de nossos algoritmos utilizam mais espaço local. No entanto, requerem menos rodadas de comunicação. Os resultados obtidos através da implementação dos nossos algoritmos foram melhores que os apresentados por outros autores. Parte deste trabalho foi publicado no "16th Symposiun on Computer Architecture and High Performance Computing".

Referências Bibliográficas

[1] A.V. Aho, J.E. Hopcroft, e J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1987.

[2] C.E.R. Alves, E.N. Cáceres, A.A. Castro Jr., S.W. Song, e J.L. Szwarcfiter. Efficient parallel implementation of transitive closure of digraphs. *10th Euro PVM/MPI 2003, Venice, Italy, September 29 - October 2, Lecture Notes in Computer Science*, 2840:126–133, 2003.

[3] S. Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, 1978.

[4] E.N. Cáceres, H. Mongelli, e S.W. Song. Algoritmos usando cgm/pvm/mpi: Uma introdução. *JAI-SBC*, 2001.

[5] E.N. Cáceres, S.W. Song, e J.L. Szwarcfiter. A parallel algorithm for transitive closure. *Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems - PDCS, Cambridge, USA, November 4-6*, páginas 116–118, 2002.

[6] A.A. Castro Jr. *Implementação e Avaliação de Algoritmos BSP/CGM para o Fecho Transitivo e Problemas Relacionados*. Tese de Mestrado, Universidade Federal de Mato Grosso do Sul - UFMS - Campo Grande/MS - Brasil, Março 2003.

[7] D. Coppersmith e S. Winograd. Matriz multiplication via arithmetic progression. *Proceedings of the 19th ACM Symposiun on Theory of Computing*, páginas 1 – 6, 1987.

[8] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, e T. von Eicken. Logp: Towards a realistic model of parallel computation. *ACM SIGPLAN: Symposium on Principles and Practice of Parallel Programming*, 4:1–12, 1993.

[9] F. Dehne. Guest editor's introduction: Coarse grained parallel algorithms. *Algorithmica*, 24(3/4):173 – 176, 1999.

[10] F. Dehne, A. Fabri, e A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *in: Proceedings ACM 9th Conference on Computational Geometry*, páginas 298 – 307, 1993.

[11] C. Demetrescu e G.F Italiano. Mantaining dynamic matrices for fully dynamic transitive closure. *CoRR*, cs.DS/0104001, 2001.

[12] M.T. Goodrich e R. Tamassia. *Algorithm Design*. IE-Wiley, 2001.

[13] M. Habib, M. Morvan, e J. Rampom. On the calculation of transitive reduction-closure of orders. *Discrete Mathematics*, 111:289–303, 1993.

[14] J.E. Hopcroft e R.E. Tarjan. Efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, 1973.

[15] G.F. Italiano. Amortized efficiency of a path retrieval data structure. *Theor. Comput. Sci.*, 48:273–281, 1986.

[16] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[17] R.M. Karp e V. Ramachandran. *Parallel Algorithms for Shared-Memory Machines - Handbook of Theoretical Computer Science*, volume A. The MIT PRESS/Elsevier, 1990.

[18] V. King e G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. *J. Comput. Syst. Sci.*, 65(1):150–167, 2002. ISSN 0022-0000.

[19] A. Koubková e V. Koubek. Algorithms for transitive closure. *Information Processing Letters*, 81:289–296, 2002.

[20] V. Kumar, A. Grama, A. Gupta, e G. Karypis. *An Introduction to Parallel Computing: Design and Analysis of Algorithms*. The Benjamin/Cummings Publishing Company, 1994.

[21] J.A. La Poutré e J. van Leeuwen. Maintenance of transitive closure and transitive reduction of graphs. *Proc. Workshop on Graph-Theoretic Concepts in Computer Science*, páginas 106–120, 1988.

[22] E. Nuutila. *Efficient Transitive Closure Computation in Large Digraphs*. Tese de Doutoramento, Helsinki University of Technology - Espoo, Finlândia, Junho 1995.

[23] P.S. Pacheco. *Parallel Programming with MPI*. Morgan-Kaufmann, 1997.

[24] A. Pagourtzis, I. Patapov, e W. Rytter. PVM computation of the transitive closure: The dependency graph approach. *Proceedings Euro PVM/MPI 2001, Lectures Notes in Computer Science*, 2131:249–256, 2001.

[25] A. Pagourtzis, I. Patapov, e W. Rytter. Observations on parallel computation of transitive and max-closure problems. *Proceedings Euro PVM/MPI 2002, Lectures Notes in Computer Science*, 2474:217–225, 2002.

[26] J.H. Reif. *Synthesis of Parallel Algorithms*. Morgan-Kaufmann, 1993.

[27] R. Roy. Transitivité et connexité. *C.R. Acad. Sci. Paris*, 249:216–218, 1959.

[28] K. Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theoretical Computer Science*, 58:325–346, 1988.

[29] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, e J. Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[30] J.L. Szwarcfiter. *Grafos e Algoritmos Computacionais*. Editora Campus, 1988.

[31] J.L. Szwarcfiter e L. Markenzon. *Estruturas de Dados e Seus Algoritmos - 2º Edição*. Editora LTC, 1994.

[32] A. Tiskin. All-pairs shortest paths computation in the BSP model. *28th ICALP, Lectures Notes in Computer Science*, 2076:178–189, 2001.

[33] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990.

[34] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.