IMPLEMENTAÇÕES PARALELAS PARA FECHO TRANSITIVO

Raphael de Aquino Gomes
Departamento de Áreas Acadêmicas IV – Instituto Federal de Educação, Ciência e Tecnologia de Goiás
Câmpus Goiânia, Rua 75, nº 46, Centro, CEP: 74055-110, Goiânia – GO, Brasil
Instituto de Informática – Universidade Federal de Goiás
Bloco IMF I, Câmpus Samambaia, Caixa Postal 131, CEP 74001-970, Goiânia – GO, Brasil
raphael@ifg.edu.br

Elisângela Silva Dias
Instituto de Informática – Universidade Federal de Goiás
Bloco IMF I, Câmpus Samambaia, Caixa Postal 131, CEP 74001-970, Goiânia – GO, Brasil
elisangela@inf.ufg.br

Márcia Rodrigues Cappelle Santana
Instituto de Informática – Universidade Federal de Goiás
Bloco IMF I, Câmpus Samambaia, Caixa Postal 131, CEP 74001-970, Goiânia – GO, Brasil
marcia@inf.ufg.br

Edson Norberto Cáceres
Faculdade de Computação – Universidade Federal de Mato Grosso do Sul
Cidade Universitária, Caixa Postal 549, CEP 79070-900, Campo Grande – MS, Brasil
edson@facom.ufms.br

Wellington Santos Martins
Instituto de Informática – Universidade Federal de Goiás
Bloco IMF I, Câmpus Samambaia, Caixa Postal 131, CEP 74001-970, Goiânia – GO, Brasil
wellington@inf.ufg.br

Resumo

A computação do fecho transitivo de um grafo é um problema que foi considerado pela primeira vez em 1959. Muitos algoritmos sequenciais para solução deste problema foram propostos e algoritmos paralelos foram considerados a partir de 1990. Apresentamos um algoritmo paralelo para computação do fecho transitivo em grafos gerais (orientados ou não). O algoritmo proposto utiliza o modelo BSP/CGM, tendo como base a utilização de busca em largura em grafos (BFS). Apresentamos três implementações paralelas baseadas no algoritmo BFS: uma em MPI, uma em OpenMP e uma híbrida (MPI/OpenMP). Para avaliar a eficiência do algoritmo proposto e das implementações desenvolvidas, foram realizados experimentos com grafos de tamanhos e características variadas. A complexidade computacional do algoritmo é $O(\frac{n^2}{p}(n+m))$.

PALAVRAS-CHAVE. Fecho Transitivo. Algoritmos Paralelos. MPI/OpenMP.

Abstract

The computation of the transitive closure of a graph is an problem that was first considered in 1959. Many sequential algorithms for solving this problem have been proposed and parallel algorithms have been considered since 1990. We present a parallel algorithm for computing the transitive closure in general graphs (oriented or not). The proposed algorithm uses the BSP/CGM model, based on the use of breadth-first search in graphs (BFS). We present three parallel implementations based on the BFS algorithm: one for MPI, one for OpenMP and one hybrid (MPI/OpenMP). To evaluate the efficiency of the algorithm and developed implementations, experiments were performed with graphs of different sizes and characteristics. The computational complexity of the algorithm is $O(\frac{n^2}{p}(n+m))$.

KEYWORDS. Transitive Closure. Parallel Algorithms. MPI/OpenMP.

1 Introdução

A computação paralela surgiu com o intuito de resolver problemas computacionalmente difíceis devido à sua complexidade e/ou tamanho de entrada. É uma técnica que utiliza o conceito de que um problema grande e difícil pode ser particionado em subproblemas menores e menos difíceis. Esses subproblemas menores, por sua vez, podem ser distribuídos e resolvidos paralelamente. Os resultados parciais obtidos podem ser combinados a fim de obtermos a solução do problema original. Esse processo de distribuição, ou seja, comunicação e processamento pode ter que ser repetido até que a solução final seja encontrada.

Conforme vários estudos já demonstraram, o principal gargalo da computação paralela é a comunicação. Logo, um algoritmo eficiente deve considerar um equilíbrio entre as rodadas de comunicação e as rodadas de processamento.

Um dos problemas interessantes da área de Teoria dos Grafos é a computação do Fecho Transitivo. Este é um problema cuja complexidade é polinomial. Pode ser definido da seguinte forma: seja G um grafo acíclico com conjunto de vértices $V(G)$ e conjunto de arestas $E(G)$ com $|V| = n$ e $|E| = m$. O *fecho transitivo* de G é o grafo G', obtido de G pela adição de uma aresta (v_i, v_j) se há em G um caminho entre os vértices v_i e v_j, para todo $1 \leq i, j \leq n$. Na Figura 1 temos um grafo G em (a) e seu respectivo fecho transitivo G' em (b). As arestas pontilhadas são as que devem ser acrescentadas ao grafo original G.

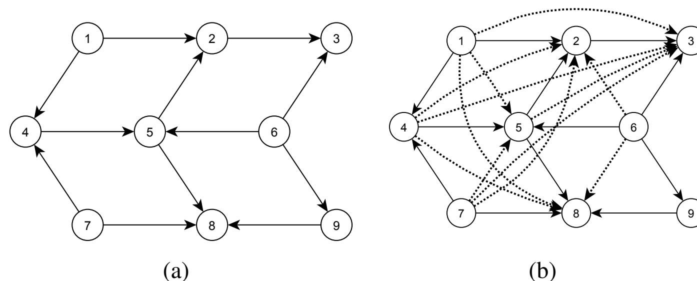

Figura 1: Um grafo orientado G (a) e seu fecho transitivo G' (b).

De acordo com Koubková e Koubek (2002), os algoritmos que resolvem este problema podem ser divididos em dois grupos. O primeiro grupo utiliza a representação do grafo por matriz de adjacências e calcula o resultado através de operações de multiplicação da matriz (como pode ser visto em Fischer e Meyer (1971)). Apesar de ser possível explorar o paralelismo neste grupo de algoritmos, a representação usada é um complicador ao considerar grafos com uma grande quantidade de vértices. Já os algoritmos do segundo grupo são baseados em técnicas de busca em grafos e permitem usar uma representação do grafo através de listas de adjacências, que requer menos espaço, o que permite sua implementação para grafos maiores.

Neste artigo, apresentamos um algoritmo paralelo para computação do fecho transitivo em grafos gerais. O algoritmo proposto utiliza o modelo BSP/CGM (Valiant (1990), Dehne (1999)), representando os grafos de acordo com o primeiro grupo discutido acima. Também utilizamos técnicas do segundo grupo, tendo como base a utilização de busca em largura em grafos (*Breadth First Search* – BFS), procurando explorar o paralelismo de arquiteturas *multicore* em CPUs. Detalhes do algoritmo BFS podem ser encontrados em Thomas et al. (2001). Apresentamos três implementações paralelas baseadas no algoritmo BFS: uma em MPI, uma em OpenMP e uma híbrida (MPI/OpenMP). Para avaliar a eficiência do algoritmo proposto e das implementações desenvolvidas, foram realizados experimentos com grafos de tamanhos e características variadas.

O restante do artigo está organizado da seguinte forma: na seção 2 apresentamos um algoritmo BSP/CGM que utiliza a busca em largura para a computação do fecho transitivo; na seção 3 des-

crevemos as implementações realizadas; na seção 4 discutimos os resultados obtidos através de experimentos e, finalmente, as considerações finais são apresentadas na seção 5.

2 Algoritmos para Fecho Transitivo

A computação do fecho transitivo de um grafo é um problema que foi considerado pela primeira vez por Roy (1959). Desde então, muitos algoritmos sequenciais para solução deste problema foram propostos. Um deles é o algoritmo de Warshall (1962) que calcula o fecho transitivo de um grafo representado por uma matriz de adjacências em tempo $O(n^3)$. Algoritmos paralelos foram propostos a partir de 1990, como o de Karp e Ramachandran (1990).

O algoritmo 1, de Warshall (1962), tem uma ideia simples e complexidade $O(n^3)$. O interesse neste algoritmo específico se dá pelo fato de que a forma em que o fecho é computado possibilita o projeto do algoritmo paralelo. A entrada para o algoritmo de Warshall é uma matriz W, $n \times n$, e será produzida uma matriz W^t de mesma dimensão. Inicialmente $W^t = W$ e $w^t_{ij} = 1$ se há uma aresta de i para j em G. Em caso contrário $w^t_{ij} = 0$. A ideia é verificar quando há entradas $w^t_{ik} = 1$ e $w^t_{kj} = 1$. Neste caso, como o vértice j é alcançável a partir de i, deve-se fazer $w^t_{i,j} = 1$.

Algoritmo 1: Warshall

Entrada: Grafo G orientado representado por uma matriz W, $n \times n$
Saída: Grafo G^t representado por uma matriz W^t

```
1  n ← |V|;
2  W^t ← W;
3  para k ← 1 até n faça
4      para i ← 1 até n faça
5          para j ← 1 até n faça
6              se w^t_ik = 1 e w^t_kj = 1 então
7                  w^t_ij ← 1;
8              fim
9          fim
10     fim
11 fim
12 Retorne G^t;
```

Baseados no algoritmo de Warshall, Alves et al. (2003) apresentaram um algoritmo para computação do fecho transitivo em grafos orientados gerais. Embora os autores tenham apresentado bons resultados experimentais, com até $\log p + 1$ rodadas de comunicação, não é possível garantir o limite de $O(\log p)$ rodadas. O algoritmo distribui a matriz de adjacências de G entre os p processadores. A distribuição é feita particionando a matriz W em p faixas horizontais e p faixas verticais formadas, respectivamente, por $\frac{n}{p}$ linhas e $\frac{n}{p}$ colunas. O algoritmo de Warshall é utilizado, então, em cada um dos processadores. Em cada estágio, o algoritmo verifica se o vértice k pertence ao caminho de v_i para v_j, sendo necessárias apenas a k-ésima linha e a k-ésima coluna.

Uma forma de calcular o fecho transitivo sequencialmente consiste na utilização de um algoritmo de busca em grafos. Para cada $v \in V(G)$ podemos utilizar a busca para encontrar o conjunto T de todos os vértices de G acessíveis a partir de v e então acrescentar uma aresta de v a w para todo $w \in T$. Neste caso, o grafo pode ser representado por uma matriz de adjacências ou por uma lista de adjacências. A complexidade deste algoritmo é $O(n^2(n+m))$.

As implementações apresentadas neste artigo seguem um algoritmo proposto que utiliza uma versão modificada da busca em largura em grafos, conforme o algoritmo 2.

Algoritmo 2: BFS-Modificado

Entrada: Grafo $G = (V,E)$ e $s,t \in V(G)$
Saída: Grafo G modificado com arestas de s para todo vértice no caminho de s a t

```
1  coloque s na fila Q;
2  enquanto Q ≠ ∅ faça
3      seja v o primeiro vértice da fila Q;
4      para cada vértice w adjacente a v faça
5          E ← E ∪ {(s,w)};
6          se w = t então
7              retorna;
8          fim
9          se w não está marcado então
10             marque w;
11             insira w no final da fila Q ;
12         fim
13     fim
14     retire v da fila Q;
15 fim
```

Nesta versão da busca em largura, são recebidos como entrada dois vértices, s e t, e o algoritmo irá verificar se há um caminho com origem em s e término em t em G. Para todo vértice w acessível a partir de s, o algoritmo insere uma aresta de s a w. O algoritmo termina quando o vértice t é encontrado ou quando todos os vértices acessíveis a partir de s forem verificados. Assim como o algoritmo BFS original, a complexidade deste algoritmo é também $O(n+m)$.

Utilizando o algoritmo de busca em largura modificado, apresentamos o algoritmo 3. Neste algoritmo, os vértices são divididos, de forma mais igual possível, entre os p processadores. Cada processador irá computar e acrescentar as arestas com ponta inicial em vértices sob sua responsabilidade. Ainda assim, a matriz de adjacências do grafo G é armazenada na memória local de cada um dos p processadores, já um vértice v pode alcançar qualquer outro vértice do grafo. Consideramos que n é divisível por p e então os vértices são divididos igualmente entre os processadores para que possam realizar o cálculo do fecho transitivo de G.

No Teorema seguinte mostramos a corretude do algoritmo 3.

Teorema 1 *O algoritmo 3 computa corretamente o fecho transitivo de um grafo. Além disso, sua complexidade computacional é* $O(\frac{n^2}{p}(n+m))$.

Prova. Seja $v \in S_j$, onde S_j $(1 \leq j \leq p)$ é uma partição dos vértices de G, e seja W_j a matriz modificada pelo processador j. Todas as arestas do fecho transitivo do grafo G com ponta inicial em v serão computadas e inseridas na matriz W_j, já que o BFS-modificado é chamado para o par (v,t), para todo $t \in V(G)$, o que é garantido pela repetição das linhas 3 e 4. Cada processador irá computar as arestas independentemente e não irá acessar as arestas acrescentadas por outro processador. Observe que não haverá criação de arestas duplicadas pelos processadores, já que cada processador j irá acrescentar apenas as arestas com ponta inicial em S_j. Para cada $v \in S_j$, o processador j irá executar o algortimo BFS-modificado no máximo n vezes (repetição da linha 4), pois há uma verificação de adjacência de v em relação à cada vértice $t \in V(G)$ antes que o BFS-modificado seja chamado. Assim, no pior caso, cada processador irá executar o algoritmo de busca no máximo $\frac{n^2}{p}$ vezes o que resultará em uma complexidade de tempo de $O(\frac{n^2}{p}(n+m))$. □

Algoritmo 3: Fecho por BFS

Entrada: Grafo G representado por uma matriz W, $n \times n$, armazenado inteiramente em cada um dos p processadores
Saída: Grafo G^t representado por uma matriz W^t

```
1  Seja S_1, ..., S_p uma partição de V(G), cujas partes tem cardinalidades iguais;
2  para cada processador j, j = 1, ..., p faça em paralelo
3  |   para cada v ∈ S_j faça
4  |   |   para cada t ∈ V(G) faça
5  |   |   |   se v ≠ t e v não é adjacente a t então
6  |   |   |   |   execute BFS-Modificado (v,t)
7  |   |   |   fim
8  |   |   fim
9  |   fim
10 fim
11 Combine as matrizes de todos os processadores;
12 Retorne G^t;
```

O algoritmo proposto minimiza a comunicação entre os processadores. Haverá comunicação para armazenar uma cópia da matriz do grafo em cada um dos processadores e, quando cada processador j termina de calcular as arestas do conjunto S_j, há uma etapa final de comunicação para combinar as matrizes dos processadores. Posteriormente apresentaremos os resultados dos experimentos com este algoritmo.

3 Implementações Paralelas

Apresentamos três implementações paralelas que resolvem o problema do fecho transitivo baseadas no algoritmo de busca em largura (BFS): uma em MPI, uma em OpenMP e uma híbrida (MPI/ OpenMP). Algumas particularidades sobre estas implementações são descritas a seguir.

3.1 Implementação em MPI

Segundo McBryan (1994), *Message Passing Interface* (MPI) é uma tentativa de padronização para ambientes de trocas de mensagem. Nos programas em MPI cada processador executa uma cópia do mesmo programa.

Usamos a linguagem C e a biblioteca MPICH2 (Gropp (2002)) para a implementação da técnica proposta no algoritmo 3. A matriz original é enviada a todos os processos. Os vértices a serem computados são distribuídos de forma o mais igualitária possível entre os processos. Após o processamento, as matrizes resultantes são combinadas no processo mestre.

3.2 Implementação em OpenMP

OpenMP é um conjunto de diretivas de compilação e uma biblioteca de rotinas que estende as linguagens Fortran, C ou C++ para permitir paralelismo usando memória compartilhada (Dagum e Menon (1998)). Assim como na versão MPI, a linguagem C foi utilizada para a implementação do algoritmo 3. A diferença é que a matriz W que representa o grafo G está acessível a todos os processos e, portanto, não há comunicação.

Como várias *threads* acessam uma mesma memória, esta implementação pôde tirar maior proveito da estratégia de utilização da busca em largura. Quando uma *thread* modifica o grafo, gerando

arestas pela execução do algoritmo de busca em largura modificado, outras *threads* com execução posterior poderão utilizar estas arestas para facilitar o cálculo do fecho transitivo para os vértices sob sua responsabilidade. É importante ressaltar que não foi realizado nenhum controle de concorrência para o acesso à memória compartilhada, uma vez que as tarefas são independentes e trabalham com conjuntos de dados disjuntos.

3.3 Implementação híbrida (MPI/OpenMP)

Nesta implementação, buscamos avaliar a integração da tecnologia MPI (memória distribuída) e da OpenMP (memória compartilhada).

Utilizamos MPI para ler o arquivo de entrada contendo a matriz de adjacências do grafo e para distribuir os dados para os processos. Em cada um dos processos é paralelizado o cálculo do fecho transitivo utilizando OpenMP, ou seja, cada *thread* realiza a busca em largura em um subconjunto de pares de vértices. Como determinante da quantidade de *threads* criadas, usamos o número de núcleos da máquina onde o processo estava executando. Por fim, as matrizes são combinadas pelo processo mestre usando a biblioteca MPI. Com este modelo híbrido obtemos um nível duplo de paralelização, separando o trabalho a ser realizado entre processos executando em máquinas diferentes e entre *threads* em cada processo.

4 Avaliação dos Resultados

Todos os experimentos foram realizados em máquinas com Sistema Operacional Linux, distribuição Ubuntu e configurações físicas descritas a seguir. Para os experimentos da implementação sequencial, utilizamos a máquina denominada `GRM` com processador AMD *Phenom*™ 9600 Quad-Core e 4 GB de memória RAM.

Nos experimentos em MPI, utilizamos um *cluster* com 4 máquinas, sendo 2 máquinas com a mesma configuração da máquina `GRM` e 2 máquinas com processador AMD *Phenom*™ II X4 B95 e memória RAM de 3 GB. A implementação foi feita usando gcc 4.4.3 e a biblioteca MPICH2 1.4.1p1. Executamos cada instância com 1, 2 e 4 processos. Esta quantidade foi adotada visando avaliar o impacto da comunicação no tempo total. A leitura dos dados foi realizada apenas pelo mestre (`GRM`) e posteriormente os dados foram divididos e distribuídos entre todos os escravos.

Para os experimentos em OpenMP foi utilizada a máquina denominada `SEDNA` com processador Intel® Xeon® X3450 2,67 GHz, 4 núcleos (permite 2 threads/núcleo) e 8MB L3 compartilhada (1MB L2, 128KB L1); e Memória RAM de 16 GB. A implementação foi feita usando gcc 4.4.3. Para cada instância avaliamos a execução com 1, 2, 4 e 7 *threads*, sendo esta quantidade escolhida de forma a facilitar a divisão exata de tarefas e a alocação de uma *thread* por núcleo.

Nos experimentos da implementação híbrida (MPI/OpenMP), foi utilizado o *cluster* já descrito. A implementação também foi feita usando gcc 4.4.3 e a biblioteca MPICH2 1.4.1p1. Foi criada a mesma quantidade de processos da implementação MPI e quantidade de *threads* de acordo com o número de núcleos da máquina em que estes estavam executando.

Foram realizados experimentos com vinte grafos diferentes com 280, 560, 1120, 2240 e 4480 vértices, que foram separados por grupos: orientado denso, orientado esparso, não orientado denso e não orientado esparso, sendo que para cada grupo há um grafo para cada quantidade de vértices. Não foram realizados testes para grafos maiores devido à limitação de memória RAM.

Para a criação destes grafos, foi utilizado um gerador de grafos acíclicos, sendo a quantidade de vértices informada pelo usuário e quantidade de arestas aleatória. Adaptamos o gerador usado no trabalho de Alves et al. (2003) para permitir grafos maiores e eliminar os laços nos vértices. O grafo resultante é representado como uma matriz de adjacências em um arquivo no formato de texto. Na execução das implementações, este grafo é informado como entrada e é gerado um arquivo no formato de texto com a matriz de adjacências do grafo representanto o fecho transitivo.

Foram realizados dez experimentos para cada grafo/ambiente e calculada a média aritmética para obter o tempo de processamento (TP) e o tempo total (TT) de execução. Devemos observar que o tempo total leva em consideração a entrada dos dados, o processamento e a comunicação.

Para efetuar a comparação entre as diversas implementações, foram executados também os códigos sequenciais referentes à implementação do algoritmo de Warshall e uma versão do Algoritmo 3 na máquina `GRM`, com um único processo, que é chamado de BFS-Sequencial nos resultados.

4.1 Resultados com MPI

Devido a problemas técnicos utilizamos neste experimento somente 4 máquinas disponíveis no *cluster*. Os resultados obtidos em relação ao TP para os grafos orientados estão na Tabela 1 e dos grafos não orientados na Tabela 2. Em todas as tabelas utilizaremos *D* para representar grafos densos e *E* para grafos esparsos.

Tabela 1: Tempos de processamento (μs) MPI para grafos orientados

Processos	280 vértices		560 vértices		1120 vértices		2240 vértices		4480 vértices	
	D	E	D	E	D	E	D	E	D	E
1	0,11	0,05	0,56	0,18	6,02	2,41	25,07	10,06	368,86	28,19
2	0,07	0,06	0,29	0,26	3,84	2,80	25,62	18,92	252,42	135,89
4	0,05	0,04	0,22	0,21	2,69	1,90	16,82	9,49	170,41	84,49

Tabela 2: Tempos de processamento (μs) MPI para grafos não orientados

Processos	280 vértices		560 vértices		1120 vértices		2240 vértices		4480 vértices	
	D	E	D	E	D	E	D	E	D	E
1	0,18	0,08	0,83	0,83	11,85	0,58	48,65	48,61	429,29	210,83
2	0,11	0,08	0,44	0,49	6,06	1,24	25,27	28,78	345,00	196,52
4	0,06	0,05	0,33	0,38	3,35	1,07	13,71	19,61	231,46	101,37

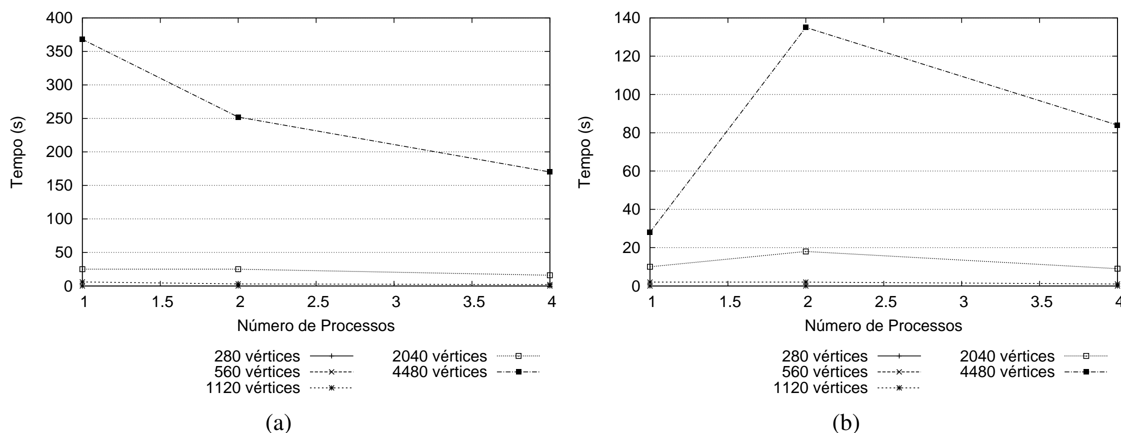

Figura 2: Tempos de processamento MPI em grafos orientados densos (a) e esparsos (b).

Podemos observar que os menores TPs ocorreram para os grafos orientados esparsos. Também para estes grafos notamos que, na maioria dos casos, o melhor tempo ocorreu para o processamento com 4 processadores. Esta variação pode ser visualizada no gráfico da Figura 2. Da mesma forma, para os demais houve um ganho de desempenho quando aumentamos o número de processadores, principalmente para os grafos com maior quantidade de vértices. Um exemplo ocorreu com os grafos orientados densos, cuja variação no TP em relação à quantidade de processadores pode

ser vista no gráfico da Figura 2(a). Na versão implementada, a comunicação ocorreu somente na distribuição inicial da matriz e no final para a combinação dos resultados obtidos. Portanto, o tempo de comunicação cresceu proporcionalmente à quantidade de vértices do grafo e foi o mesmo para grafos do mesmo tamanho.

4.2 Resultados com OpenMP

Os TPs na implementação em OpenMP podem ser observados nas Tabelas 3 e 4. Observamos que para os grafos esparsos houve uma diferença considerável em relação aos grafos densos. A diferença maior ocorreu nos grafos esparsos orientados, cujo TP foi, em média, mais de 10 vezes menor do que o tempo para grafos densos e em média 6 vezes menor do que o tempo gasto no processamento para os grafos esparsos não orientados. Uma razão para esta grande diferença está no fato de ter sido utilizada a busca em largura que tem melhores tempos de execução para grafos esparsos. Podemos observar também uma média de 215% no ganho de processamento ao se comparar o melhor e o pior tempo para cada instância.

Tabela 3: Tempos de processamento (μs) OpenMP para grafos orientados

Thread	280 vértices		560 vértices		1120 vértices		2240 vértices		4480 vértices	
	D	E	D	E	D	E	D	E	D	E
1	0,14	0,07	0,51	0,16	8,80	3,19	33,68	13,26	528,20	34,00
2	0,08	0,03	0,37	0,10	5,21	1,93	21,82	7,75	347,06	20,54
4	0,05	0,02	0,23	0,10	4,11	1,53	17,84	6,52	326,66	21,87
7	0,05	0,01	0,41	0,12	4,80	1,93	20,24	7,24	349,94	20,67

Tabela 4: Tempos de processamento (μs) OpenMP para grafos não orientados

Thread	280 vértices		560 vértices		1120 vértices		2240 vértices		4480 vértices	
	D	E	D	E	D	E	D	E	D	E
1	0,27	0,11	1,12	1,14	17,84	0,58	71,17	69,94	640,38	288,79
2	0,13	0,05	0,59	0,63	10,76	0,43	46,08	46,27	414,57	181,84
4	0,09	0,03	0,52	0,56	8,83	0,32	36,91	38,02	431,10	207,03
7	0,17	0,08	0,58	0,54	10,25	0,41	43,49	44,11	418,48	196,98

4.3 Resultados com MPI/OpenMP

Com relação aos experimentos realizados em MPI (seção 4.1) e OpenMP (seção 4.2), observamos melhorias no TP com o uso do MPI para grafos não orientados (na maioria dos casos) e para grafos orientados densos. Já com o uso do OpenMP foram obtidas melhorias apenas para grafos orientados esparsos.

Tabela 5: Tempos de processamento (μs) MPI/OpenMP para grafos orientados

Processos	280 vértices		560 vértices		1120 vértices		2240 vértices		4480 vértices	
	D	E	D	E	D	E	D	E	D	E
1	0,04	0,03	0,15	0,07	1,63	0,64	6,81	2,74	98,75	7,63
2	0,03	0,03	0,10	0,14	1,35	1,01	8,57	4,93	83,84	44,84
4	0,02	0,02	0,06	0,18	0,69	0,52	4,27	2,41	43,14	21,79

Na implementação híbrida, observamos que o TP em grafos orientados esparsos sempre foi melhor do que o TP em grafos orientados densos (Tabela 5). A Figura 3 ilustra graficamente a diferença de tempo para os grafos orientados.

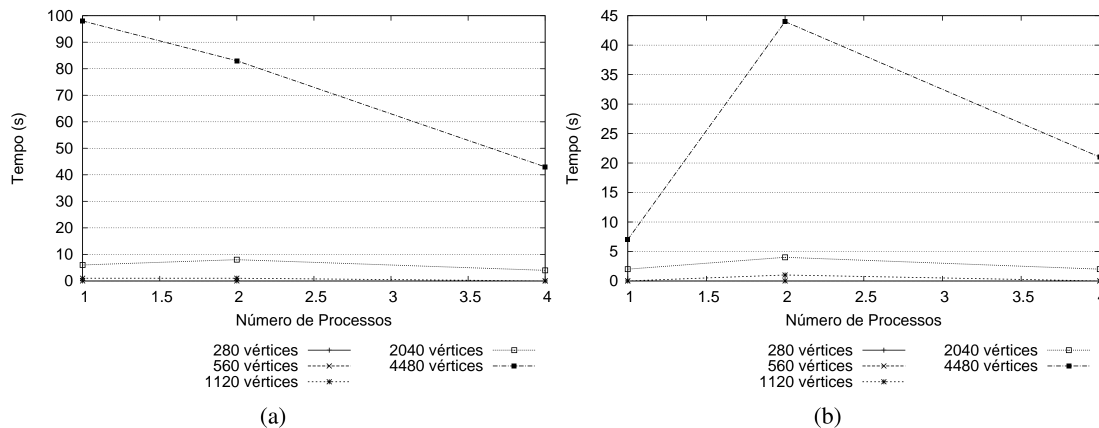

Figura 3: Tempos de processamento MPI/OpenMP para grafos orientados densos (a) e esparsos (b).

Conforme é apresentado na Tabela 6, constatamos que nem sempre o tempo de processamento em grafos não orientados esparsos é menor do que em grafos não orientados densos. Como exemplo, observe os grafos com 2240 vértices.

Tabela 6: Tempos de processamento (μs) MPI/OpenMP para grafos não orientados

Processos	280 vértices		560 vértices		1120 vértices		2240 vértices		4480 vértices	
	D	E	D	E	D	E	D	E	D	E
1	0,06	0,03	0,24	0,22	3,13	0,17	13,70	13,05	115,04	57,70
2	0,04	0,02	0,14	0,17	1,61	0,44	6,76	9,54	113,06	50,26
4	0,02	0,02	0,08	0,09	0,83	0,22	3,51	4,96	58,13	26,58

4.4 Discussão dos Resultados Gerais

Na Tabela 7, mostramos a comparação entre os tempos obtidos em todas as implementações paralelas, indicando qual teve o melhor e o pior desempenhos quanto ao TP e qual o ganho. É necessário ressaltar que selecionamos o melhor TP de cada uma das implementações e que o ganho de processamento foi calculado pela divisão do tempo máximo pelo mínimo.

Para grafos orientados densos, percebemos que a implementação híbrida teve um desempenho bem superior às outras paralelas e sequenciais. Para grafos com 4480 vértices, foi 8 vezes melhor do que a implementação do BFS-sequencial e aproximadamente 83 vezes melhor do que o algoritmo de Warshall. Este último apresentou o pior desempenho em quase todas as situações, com exceção de grafos não-orientados densos, onde o BFS-sequencial teve maior tempo de processamento.

Para grafos orientados esparsos, observamos que, exceto para grafos com 280 vértices em que os melhores melhor implementação foi em OpenMP, a implementação híbrida obteve melhor resultado. Para grafos com 560, 2240 e 4480 vértices, o ganho de tempo foi superior a 20 vezes. Para grafos com 4480 vértices, ela foi 83,31 vezes melhor que a implementação de Warshall. Este foi o maior ganho de desempenho.

Para grafos não orientados densos, a implementação híbrida foi a melhor para todos os grafos testados. A implementação de Warshall obteve pior desempenho para grafos com 560, 1120 e 4480 vértices. Para os demais grafos não orientados densos, o maior tempo de processamento observado foi o da implementação BFS-sequencial.

Para grafos não orientados esparsos, a implementação híbrida novamente obteve melhor desempenho e o de Warshall o pior desempenho.

Grafo		Tempo de processamento Warshall	Tempo de processamento BFS-Sequencial	Melhor tempo de Processamento MPI	Melhor tempo de Processamento OpenMP	Melhor tempo de Processamento MPI-OpenMP	Tempo de Processamento Mínimo	Implementação com processamento Mínimo	Tempo de processamento Máximo	Implementação com processamento Máximo	Ganho de Processamento
Orientado Denso	280 vértices	0,15	0,11	0,05	0,05	0,02	0,02	Híbrida	0,15	Warshall	6,15
	560 vértices	1,22	0,56	0,22	0,23	0,06	0,06	Híbrida	1,22	Warshall	20,46
	1120 vértices	10,09	6,06	2,69	4,11	0,69	0,69	Híbrida	10,09	Warshall	14,65
	2240 vértices	80,25	26,01	16,82	17,84	4,27	4,27	Híbrida	80,25	Warshall	18,81
	4480 vértices	642,38	370,18	170,41	326,66	43,14	43,14	Híbrida	642,38	Warshall	14,89
Orientado Esparso	280 vértices	0,15	0,05	0,04	0,01	0,02	0,01	OpenMP	0,15	Warshall	10,84
	560 vértices	1,18	0,18	0,18	0,10	0,05	0,05	Híbrida	1,18	Warshall	21,92
	1120 vértices	10,06	2,49	1,90	1,53	0,52	0,52	Híbrida	10,06	Warshall	19,27
	2240 vértices	80,06	10,15	9,49	6,52	2,41	2,41	Híbrida	80,06	Warshall	33,27
	4480 vértices	635,52	28,78	28,19	20,54	7,63	7,63	Híbrida	635,52	Warshall	83,31
Não-Orientado Denso	280 vértices	0,15	0,18	0,06	0,09	0,02	0,02	Híbrida	0,18	BFS-Seq	7,75
	560 vértices	1,21	0,83	0,33	0,52	0,08	0,08	Híbrida	1,21	Warshall	14,72
	1120 vértices	10,1	11,73	3,35	8,83	0,83	0,83	Híbrida	11,73	BFS-Seq	14,05
	2240 vértices	80,29	49,1	13,71	36,91	3,51	3,51	Híbrida	80,29	Warshall	22,86
	4480 vértices	643,95	431,8	231,46	414,57	58,13	58,13	Híbrida	643,95	Warshall	11,08
Não-Orientado Esparso	280 vértices	0,15	0,09	0,05	0,03	0,02	0,02	Híbrida	0,15	Warshall	6,56
	560 vértices	1,22	0,83	0,38	0,54	0,09	0,09	Híbrida	1,22	Warshall	13,19
	1120 vértices	9,72	0,6	0,58	0,32	0,17	0,17	Híbrida	9,72	Warshall	58,26
	2240 vértices	80,4	48,84	19,61	38,02	4,96	4,96	Híbrida	80,40	Warshall	16,21
	4480 vértices	640,01	213,79	101,37	181,84	26,58	26,58	Híbrida	640,01	Warshall	24,08
										Máximo ganho:	83,31

Tabela 7: Tempos de processamento (μs) para todas implementações com todos os grafos.

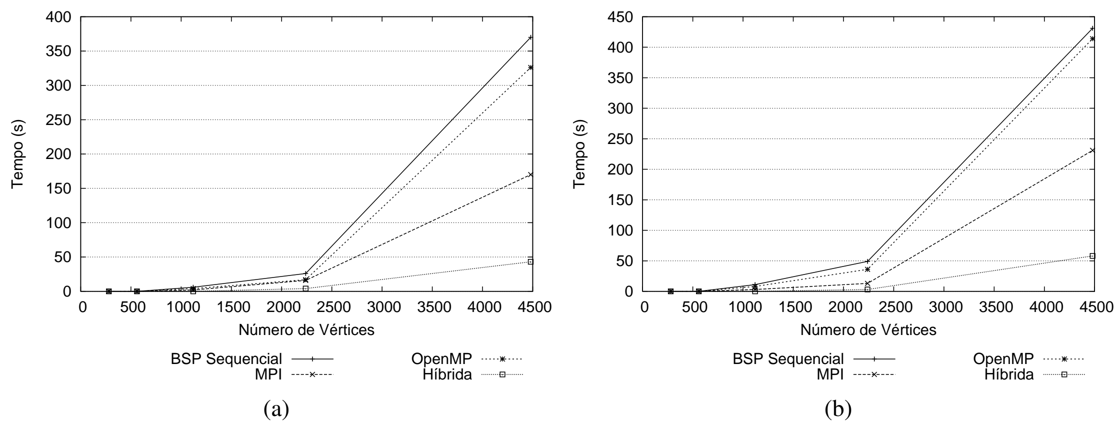

Figura 4: Tempos de processamento para grafos densos orientados (a) e não orientados (b).

Comparando somente as implementações baseadas no algoritmo paralelo proposto, a implementação híbrida obteve melhor desempenho, como podemos observar na Figura 4. O único caso em que ela não obteve um ganho superior foi para grafos orientados esparsos com 280 vértices. Como era esperado, também observamos que a implementação sequencial obteve os piores resultados.

Analisamos também o tempo total de execução. Os resultados podem ser vistos na Tabela 8 e mantiveram praticamente as mesmas relações observados com o tempo de processamento. Uma das mudanças observadas foi para grafos não-orientados esparsos com 280 vértices para os quais a melhor implementação passou a ser OpenMP. Outra mudança ocorreu na avaliação dos grafos não-orientados densos com 280 vértices, onde o pior tempo total passou a ser com MPI. Vale ressaltar também que o ganho de processamento máximo foi de 41,56 vezes melhor que o pior tempo observado, um valor expressivo.

A melhoria visualizada com a implementação híbrida (tanto com relação ao tempo de processamento quanto total) pode ser atribuída a um nível maior de paralelismo obtido com a união de MPI e OpenMP. Além disso, a comunicação e o uso de memória são diminuídas pelo compartilhamento

Grafo		Tempo de Processamento Warshall	Tempo total BFS-Sequencial	Melhor tempo Total MPI	Melhor tempo Total OpenMP	Melhor tempo total MPI-OpenMP	Tempo total Mínimo	Implementação com tempo total mínimo	Tempo total Máximo	Implementação com tempo total máximo	Ganho de Processamento
Orientado Denso	280 vértices	0,15	0,14	0,15	0,12	0,09	0,09	Híbrida	0,15	Warshall	1,76
	560 vértices	1,22	0,67	0,67	0,33	0,28	0,28	Híbrida	1,22	Warshall	4,32
	1120 vértices	10,09	6,49	4,93	4,74	2,12	2,12	Híbrida	10,09	Warshall	4,76
	2240 vértices	80,25	27,75	25,96	19,74	8,75	8,75	Híbrida	80,25	Warshall	9,17
	4480 vértices	642,38	377,09	199,46	335,13	79,88	79,88	Híbrida	642,38	Warshall	8,04
Orientado Esparso	280 vértices	0,15	0,08	0,09	0,05	0,07	0,05	OpenMP	0,15	Warshall	2,79
	560 vértices	1,18	0,29	0,31	0,21	0,20	0,20	Híbrida	1,18	Warshall	5,86
	1120 vértices	10,06	2,93	2,89	2,18	1,13	1,13	Híbrida	10,06	Warshall	8,90
	2240 vértices	80,06	11,9	11,98	8,56	4,69	4,69	Híbrida	80,06	Warshall	17,06
	4480 vértices	635,52	35,53	35,79	28,75	15,29	15,29	Híbrida	635,52	Warshall	41,56
Não-Orientado Denso	280 vértices	0,15	0,21	0,22	0,12	0,10	0,10	Híbrida	0,22	MPI	2,11
	560 vértices	1,21	0,95	0,82	0,70	0,37	0,37	Híbrida	1,21	Warshall	3,24
	1120 vértices	10,1	12,16	5,57	9,31	3,06	3,06	Híbrida	12,16	BFS-Seq	3,97
	2240 vértices	80,29	50,86	22,83	39,46	12,47	12,47	Híbrida	80,29	Warshall	6,44
	4480 vértices	643,95	438,68	268,20	421,71	94,79	94,79	Híbrida	643,95	Warshall	6,79
Não-Orientado Esparso	280 vértices	0,15	0,12	0,12	0,06	0,08	0,06	OpenMP	0,15	Warshall	2,49
	560 vértices	1,22	0,94	0,86	0,73	0,35	0,35	Híbrida	1,22	Warshall	3,45
	1120 vértices	9,72	1,03	1,06	0,93	0,65	0,65	Híbrida	9,72	Warshall	14,89
	2240 vértices	80,4	50,64	28,75	40,07	14,04	14,04	Híbrida	80,40	Warshall	5,73
	4480 vértices	640,01	220,59	137,92	189,48	63,16	63,16	Híbrida	640,01	Warshall	10,13
										Máximo ganho:	41,56

Tabela 8: Tempos totais (μs) para todas implementações com todos os grafos.

de dados entre as *threads*.

Na Tabela 9 são apresentados os cálculos dos *speedups* para o TP, em relação ao tempo de execução com somente um processo/*thread*. Na computação paralela é desejável que se consiga obter o *speedup* linear ($S_p = p$). Apesar de apresentarmos *speedups* ruins em alguns casos, observamos ganhos em todas as implementações.

Processos	Grafos																			
	Orientado Denso					Orientado Esparso					Não-Orientado Denso					Não-Orientado Esparso				
	280	560	1120	2040	4080	280	560	1120	2040	4080	280	560	1120	2040	4080	280	560	1120	2040	4080
2	1,42	1,94	1,57	0,98	1,46	0,82	0,69	0,86	0,53	0,21	1,71	1,88	1,95	1,93	1,24	1,13	1,72	0,47	1,69	1,07
4	2,14	2,55	2,23	1,49	2,16	1,24	0,85	1,26	1,06	0,33	3,03	2,47	3,54	3,55	1,85	1,67	2,19	0,55	2,48	2,08

Implementação em MPI

Threads	Grafos																			
	Orientado Denso					Orientado Esparso					Não-Orientado Denso					Não-Orientado Esparso				
	280	560	1120	2040	4080	280	560	1120	2040	4080	280	560	1120	2040	4080	280	560	1120	2040	4080
2	1,74	1,41	1,69	1,54	1,52	2,55	1,54	1,66	1,71	1,66	2,18	1,92	1,66	1,54	1,54	2,15	1,81	1,34	1,51	1,59
4	2,76	2,25	2,14	1,89	1,62	3,00	1,56	2,08	2,03	1,55	3,12	2,15	2,02	1,93	1,49	3,43	2,03	1,83	1,84	1,39
7	2,97	1,25	1,83	1,66	1,51	5,22	1,29	1,65	1,83	1,64	1,63	1,93	1,74	1,64	1,53	1,31	2,11	1,41	1,59	1,47

Implementação em Open MP

Processos	Grafos																			
	Orientado Denso					Orientado Esparso					Não-Orientado Denso					Não-Orientado Esparso				
	280	560	1120	2040	4080	280	560	1120	2040	4080	280	560	1120	2040	4080	280	560	1120	2040	4080
2	1,36	1,54	1,21	0,79	1,18	1,13	0,75	0,63	0,56	0,17	1,67	1,69	1,95	2,03	1,02	1,01	1,31	0,38	1,37	1,15
4	1,73	2,53	2,37	1,60	2,29	1,68	1,39	1,22	1,14	0,35	2,56	2,97	3,76	3,90	1,98	1,44	2,39	0,75	2,63	2,17

Implementação Híbrida

Tabela 9: *Speedup* do tempo de processamento para grafos orientados.

5 Considerações Finais

Neste artigo, apresentamos um algoritmo BSP/CGM (algoritmo 3) para computação do fecho transitivo em grafos gerais (orientados ou não) que utiliza uma versão modificada da busca em largura, executada sequencialmente em cada um dos p processadores. O algoritmo tem complexidade de tempo $O(\frac{n^2}{p}(n+m))$, mas apresenta bons resultados na prática.

Para a realização de experimentos, implementamos o algoritmo 3 em diferentes modelos de computação paralela: MPI, OpenMP e junção dessas duas. Embora os experimentos tenham sido realizados em um ambiente restrito com um pequeno número de processadores/núcleos, obtivemos bons resultados que mostram que a implementação paralela pode ser vantajosa em vários casos, principalmente quando temos grafos com grande quantidade de vértices. Para os grafos esparsos, os resultados foram ainda melhores. Este fato pode ser explicado pela utilização da busca em largura que apresenta melhor desempenho para grafos em que a quantidade de arestas é $O(n)$, diminuindo a complexidade do algoritmo de computação do fecho transitivo para $O(\frac{n^3}{p})$.

Observamos na Figura 4 que a implementação híbrida obteve melhor desempenho para praticamente todos os tipos de grafos. Foram apresentados os cálculos dos *speedups* para o tempo de processamento e para o tempo total. Observamos um bom valor de *speedup* para alguns grafos.

Algumas modificações nos algoritmos poderiam melhorar seus desempenhos e são sugestões de trabalhos futuros. Como os algoritmos implementados podem receber como entrada grafos orientados e não orientados, um tratamento específico para grafos não orientados poderia apresentar um resultado melhor do que o encontrado neste trabalho. Uma outra proposta seria melhorar o algoritmo em MPI de forma a diminuir o volume de dados na comunicação. Além disso, o uso de *Graphics Processing Unit* (GPUs) para computação de propósito geral é um dos maiores atrativos atuais na computação paralela, de forma que a implementação do algoritmo usando esta tecnologia pode trazer ganhos satisfatórios, principalmente se desenvolvida de forma híbrida com as demais.

Referências

Alves, C., Cáceres, E., Castro, A., Song, S., e Szwarcfiter, J. (2003). Efficient parallel implementation of transitive closure of digraphs. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 126–133.

Dagum, L. e Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55.

Dehne, F. (1999). *Coarse-grained Parallel Algorithms*. Springer.

Fischer, M. e Meyer, A. (1971). Boolean matrix multiplication and transitive closure. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pages 129–131. IEEE.

Gropp, W. (2002). Mpich2: A new start for mpi implementations. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 37–42.

Karp, R. M. e Ramachandran, V. (1990). Parallel algorithms for shared-memory machines. *Handbook of Theoretical Computer Science*, A(17):869–941.

Koubková, A. e Koubek, V. (2002). Algorithms for transitive closure. *Information processing letters*, 81(6):289–296.

McBryan, O. (1994). An overview of message passing environments. *Parallel Computing*, 20(4):417–444.

Roy, B. (1959). Transitivité et connexité. *C. R.*, 249:216–218.

Thomas, H., Charles, E., e Ronald, L. (2001). *Introduction to Algorithms.* MIT press.

Valiant, L. (1990). A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111.

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