Uma implementação da busca em largura com estrutura bag e OpenMP

S. L. Gonzaga de Oliveira[1], M. I. Santana[1], D. Brandão[2], C. Osthoff[3]

[1]Universidade Federal de Lavras (UFLA)

sanderson@ufla.br, marcio.santana@estudante.ufla.br

[2]Centro Federal de Educação Tecnológica Celso Suckow da Fonseca (CEFET/RJ)

diego.brandao@eic.cefet-rj.br

[3]Laboratório Nacional de Computação Científica

osthoff@lncc.br

Abstract. *This paper shows the results of an OpenMP-based implementation of the breadth-first search procedure using the bag data structure. The code relied on the C++ programming language. Furthermore, this paper reimplements an existing proposal coded using the discontinued Cilk++ programming language. The experiments relied on ten undirected graphs and ten digraphs in executions performed on a machine containing eight cores and two threads per core. Regarding the serial version, the new parallel implementation yielded speedup from 3.2x to 5.7x when using eight threads and from approximately 3x to 8x when using 16 threads.*

Resumo. *Neste artigo, são mostrados resultados de uma re-implementação da busca em largura na linguagem C++ com estrutura bag e interface OpenMP. A implementação é baseada em uma proposta existente na bibliografia que utilizou a linguagem Cilk++, que foi descontinuada. Para os experimentos realizados neste presente trabalho, foram utilizados 10 grafos não direcionados e 10 digrafos em uma máquina composta de oito núcleos, com duas threads por núcleo. Em relação à versão serial, a nova implementação apresentou aceleração de 3,2 a 5,7x ao utilizar oito threads e de aproximadamente 3 a 8x ao utilizar 16 threads.*

1. Introdução

Um dos grandes desafios é a análise do grande e crescente volume de dados provenientes de tecnologias como telefones celulares, Medicina, Biologia, Física, interações nas mídias sociais, classificação de páginas da web, pesquisas genômicas, experimentos científicos de diversas áreas etc. A busca em largura é um dos algoritmos em grafos mais importantes e estudados, por ser um algoritmo fundamental para percorrer grafos. Esse algoritmo é base de diversos outros métodos relevantes, com aplicações em diversas áreas da ciência e da indústria. Como exemplos, percorrer os vértices do grafo é

um componente chave em aplicações práticas como interação de proteínas, transporte terrestre, redes sociais, cálculo da centralidade entre diferenças, identificação de componentes conexos, agrupamento de grafos, *pathminer*, detecção de estrutura de comunidade, pesquisa de padrões em sequências de DNA/RNA usando Trie, biologia computacional, automação de *design* eletrônico, conectividade em grafos etc. [Hong et al. 2011, Chhugani et al. 2012, Tithi et al. 2013]. Como outro exemplo, os algoritmos de baixo custo computacional no estado da arte para o problema de redução de largura de banda de matrizes, um importante problema de otimização combinatória, são baseados na busca em largura [Gonzaga de Oliveira and Silva 2020b, Gonzaga de Oliveira and Silva 2020a, Gonzaga de Oliveira and Silva 2021].

Em geral, suítes de *benchmark* que visam aplicações com grafos incluem a busca em largura como um elemento principal [Hong et al. 2011]. Como exemplo, o *benchmark* utilizado no Graph500[1] enfatiza o subsistema de comunicação do sistema, em vez de contar o ponto flutuante de precisão dupla. A suíte de *benchmark* Graph500 classifica supercomputadores com base em seu desempenho em aplicativos com utilização massiva de dados. O *benchmark* é baseado na busca em largura aplicada em um grande grafo não direcionado - um modelo de grafo de Kronecker com grau médio de 16. Um dos núcleos de computação no *benchmark* executa uma busca em largura paralela a partir de vértices aleatórios [Suzumura et al. 2011]. A busca em largura utilizada é síncrona e realizada por nível de vértices em relação ao vértice inicial.

A versão serial da busca em largura, implementada com fila, é rápida e eficiente, demandando tempo $\mathcal{O}(|V|+|E|)$ para o grafo $G = (V, E)$, composto de um conjunto V de vértices e de um conjunto E de arestas. As constantes escondidas pela notação assintótica são pequenas por causa da grande simplicidade das operações com filas: inserir no final da fila e remoção do início da fila são operações $\Theta(1)$.

A paralelização da busca em largura não é trivial porque é um algoritmo irregular e dependente de dados. Apesar de eficiente na versão serial, a implementação da busca em largura com fila é o maior obstáculo para a implementação paralela do algoritmo. A paralelização da busca em largura implementada com fila permite paralelismo muito pequeno em grafos esparsos, i.e., para $|E| \approx |V|$ [Leiserson and Schardl 2010].

Percorrer os vértices de um grafo apresenta vários desafios interessantes quando executado em plataformas de processadores modernos. O problema normalmente envolve acessos de memória de longa latência seguidos por pouca computação aritmética, levando à utilização ineficaz de recursos de computação e largura de banda. É difícil contornar essa latência de acesso à memória principal devido à natureza espacialmente incoerente dos acessos. Além disso, os requisitos de largura de banda desses acessos também são altos, levando a uma perda significativa de desempenho. Por fim, as arquiteturas com vários *sockets* requerem tratamento especial, pois precisam de otimizações com reconhecimento de localidade para garantir que o tráfego baixo entre *sockets* seja mantido. Todavia, isso pode entrar em conflito com a necessidade de manter todos os *sockets* ativos para balanceamento de carga. Nesses casos, é necessário tomar uma decisão cuidadosa para equilibrar esses dois fatores para obter o melhor desempenho [Chhugani et al. 2012].

Quando muitos processadores estão disponíveis, a árvore formada pela busca em

[1] https://graph500.org.

largura é construída em camadas, níveis ou fronteiras, a partir do vértice inicial, usando a sincronização de camadas. Quando a quantidade de vértices em uma nível é grande, as tarefas de identificação de vértices do nível seguinte podem ser divididas entre os processadores. O desafio é projetar uma estrutura de dados adequada para armanezar os vértices dos níveis de forma que permita leitura e gravação eficiente em paralelo [Belova and Ouyang 2017].

Nossa intenção no presente trabalho é obter uma versão paralela eficiente da busca em largura pela interface OpenMP. Neste contexto, eficiência significa uma aceleração linear, em que o número total de operações realizadas é o mesmo em relação a um fator constante ao que é comparável com a versão serial do algoritmo. Isso já foi alcançado, por exemplo, com a linguagem de programação Cilk++ [Leiserson and Schardl 2010]. Entretanto, essa linguagem de programação foi descontinuada. Com isso, nossa intenção de obter uma implementação eficiente com OpenMP é porque essa interface de programação de aplicativos é possivelmente a mais utilizada para se projetar algoritmos paralelos em sistemas de computação com memória compartilhada.

Neste trabalho, mostramos resultados de nossa nova implementação da busca em largura com a estrutura bag na linguagem de programação C++ com OpenMP. Nossa implementação é uma versão modificada em C++ e OpenMP da implementação por Cilk++ apresentada por Leiserson e Schardl [Leiserson and Schardl 2010].

Este texto está estruturado da seguinte forma. Trabalhos relacionados são descritos na Seção 2. A busca em largura serial é apresentada na Seção 3. Nossa implementação é descrita na Seção 4. Os testes são descritos na Seção 5. Os resultados são apresentados na Seção 6. Por fim, as conclusões são abordadas na Seção 7.

2. Trabalhos relacionados

Uma quantidade muito grande de versões paralelas da busca em largura já foi explorada. Em nosso trabalho, focamos na paralelização desse algoritmo em sistemas com processadores compostos de mais de um núcleo e com memória compartilhada, de forma que o grafo seja carregado na memória principal do sistema, isto é, não são realizados acessos à memória secundária. Neste contexto, Hassaan et al. [Hassaan et al. 2010] implementaram várias otimizações para conseguir aceleração quase linear em relação à versão serial em uma máquina com 16 threads.

Substituir a fila por outra estrutura para se paralelizar a busca em largura pode também comprometer o desempenho em relação à versão serial, pois o algoritmo serial é simples e rápido por justamente utilizar a estrutura fila. Assim, Leiserson e Schardl [Leiserson and Schardl 2010] projetaram uma estrutura de dados multiconjuntos chamada bag. Na versão paralela, a estrutura bag substitui a estrutura fila, ou seja, a estrutura bag mantém os vértices do próximo nível a serem processados. Essa é uma estrutura livre de bloqueios e permite a inserção de elementos na estrutura de forma tão rápida quanto na fila. A estrutura bag também permite procedimentos de divisão e união da estrutura de forma eficiente. A implementação dos autores buscou reduzir o tempo de sincronização do algoritmo. O algoritmo utiliza laços de repetição aninhados para evitar realizar iterações em ordem. O algoritmo mantém duas áreas de trabalho, uma para o nível corrente e outra para o próximo nível.

Esse tipo de algoritmo é chamado de *wavefront* e *top-down*, em que o algoritmo

percorre o grafo em níveis (ou camadas) em relação ao vértice inicial. Esse tipo de algoritmo também é chamado de algoritmo paralelo síncrono em massa (*Bulk Synchronous Parallel algorithm*). Chhugani et al. [Chhugani et al. 2012] explicaram que os métodos síncronos para a busca em largura percorrem o grafo em uma sequência de etapas de bloqueio delimitadas por pontos de sincronização. Cada etapa executa atualizações de todos os vértices em uma só profundidade. Abordagens assíncronas eliminam o uso de pontos de sincronização, embora isso possa resultar em várias atualizações para um só vértice e, consequentemente, ineficiência no trabalho. As abordagens síncronas funcionam bem para muitos grafos cujos diâmetros são pequenos em comparação com o número de vértices. Muitos grafos oriundos de aplicações reais apresentam essa característica. Portanto, concentramo-nos na abordagem síncrona.

Conforme descrito por St. John et al. [St. John et al. 2012], a implementação paralela de Leiserson e Schardl [Leiserson and Schardl 2010] não está livre de problemas. Ao se utilizar um sistema com memória compartilhada, vértices que pertencem a um conjunto são processados em paralelo. Como é possível que dois vértices no conjunto compartilhem um vizinho em comum, a atualização em outro conjunto deve ser realizada de forma atômica. Isso gera uma fonte de contenção no processamento dos elementos e pode criar gargalos na execução.

Belova e Ouyang [Belova and Ouyang 2017] substituíram a estrutura bag por vetores de bits. Uma implementação trivial com vetores de bits da mesma forma que descrito em Leiserson e Schardl [Leiserson and Schardl 2010] é apenas 38% mais lenta que o *software* Ligra [Shun and Blelloch 2013a, Shun and Blelloch 2013b], um dos códigos da busca em largura mais rápidos para execução em um único computador [Belova and Ouyang 2017].

Tivemos uma experiência em estudo anterior [Brandão et al. 2019], em que implementamos exatamente a mesma versão paralela de Leiserson e Schardl [Leiserson and Schardl 2010], mas em linguagem de programação C++ com OpenMP. Nesse trabalho, foi realizado apenas um experimento, com desaceleração de 0,04 da versão paralela em relação à versão serial. Estudos aprofundados no código nos fizeram descartá-la. Assim, o código foi totalmente re-implementado e apresentamos nossa experiência no trabalho presente. A contribuição da publicação anterior foi por ser um verdadeiro tutorial da publicação de Leiserson e Schardl [Leiserson and Schardl 2010]. Além disso, o trabalho anterior [Brandão et al. 2019] contém mais detalhes da implementação da busca em largura com a estrutura bag na linguagem de programação C++ com OpenMP do que a própria publicação de Leiserson e Schardl [Leiserson and Schardl 2010], que utilizou a linguagem de programação Cilk++.

3. Busca em largura serial

No Algoritmo 1, mostra-se o pseudocódigo da busca em largura serial. O algoritmo recebe um grafo conexo e um vértice inicial v_0.

O algoritmo inicializa as distâncias dos demais vértices do grafo em relação ao vértice v_0 com ∞ na linha 1. O vértice v_0 é inserido na fila F na linha 3.

O laço de repetição na linha 4 percorre os demais vértices do grafo. Na linha 5, o algoritmo desenfileira um vértice da fila F e o atribui a u.

Algoritmo 1: Busca em largura serial.

```
   Entrada: grapo conexo G = (V, E); vértice v_0 ∈ V
1  início
2      para cada vértice u ∈ V − {v_0} faça Dist[u] ← ∞
3      Dist[v_0] ← 0
4      Enfileira(F,v_0)
5      enquanto F ≠ ∅
6          u ← Desenfileira(F)
7          para cada vértice v ∈ V tal que (u, v) ∈ E faça
8              se Dist[v] = ∞, então,
9                  Dist[v] ← Dist[u] + 1
10                 Enfileira(F,v)
```

O laço de repetição na linha 6 percorre a lista de adjacências do vértice u. Se a condição na linha 7 é satisfeita, então, o vértice v adjacente ao vértice u ainda não foi visitado. Assim, a distância do vértice v é atualizada para a distância do vértice u mais um e o vértice v é inserido na fila F.

4. Estrutura bag modificada

Para deixar o texto completo, reproduzimos a proposta de Leiserson e Schardl [Leiserson and Schardl 2010]. No decorrer do texto, comentamos as modificações realizadas no presente estudo em relação à proposta original desses autores.

Uma estrutura bag é um conjunto não ordenado, ou seja, é um *backbone* para estruturas *pennants*. Especificamente, uma estrutura bag é constituída por uma coleção de *pennants* implementada em um *array*, em que cada posição contém um ponteiro para uma estrutura *pennant*. Uma estrutura *pennant* é uma árvore em que cada nó possui dois filhos, com exceção da raiz, que possui apenas o filho da esquerda. A partir desse filho, a estrutura *pennant* é uma árvore binária balanceada. Um *pennant* sempre conterá 2^k elementos, para $k \in \mathbb{N}^*$. As árvores *pennants* podem ser unidas e divididas de forma eficiente. As operações Divide-Bag e Une-Bags são executadas em tempo $\mathcal{O}(lg|V|)$.

As principais modificações em relação à proposta de Leiserson e Schardl, implementadas neste trabalho, são descritas a seguir.

1. Criação da estrutura bag - método construtor: ao se criar a estrutura bag, é criado um *array* em que todos os ponteiros são inicializados com nulo.
2. Inserção na estrutura bag: é criada uma nova estrutura *pennant* (*pennant* atual) com o vértice a ser inserido. Então, percorre-se o vetor de *pennants* da estrutura. Para cada *pennant* encontrado, isto é, a entrada da estrutura bag não contém um ponteiro nulo, é realizada a união com a estrutura *pennant* atual e a posição correspondente passa a conter um ponteiro nulo.
3. União de duas estruturas *bags*: a estrutura bag chamadora conterá a união resultante e a estrutura bag passada como argumento será incorporada à estrutura bag chamadora.

Os Algoritmos 2, 3 e 4 mostram os pseudocódigos da busca em largura em paralelo baseado no padrão OpenMP, adaptado da versão em Cilk++ proposta por Leiserson e Schardl. Após a inicialização, o Algoritmo 2 começa o laço de repetição **enquanto** mostrado na linha 7. Esse laço de repetição chama iterativamente a função auxiliar Processa-Nível (Algoritmo 3) para processar os níveis $l = 0, 1, \cdots, D$, em que D é o diâmetro do grafo [Leiserson and Schardl 2010].

Algoritmo 2: Busca em largura em paralelo [Leiserson and Schardl 2010].

```
   Entrada: G = (V, E); vértice inicial v_0 ∈ V
   Output: vetor de distâncias Dist
1  início
2      parallel para cada vértice v ∈ V \ {v_0} faça Dist[v] ← ∞
3      Dist[v_0] ← 0
4      l ← 0
5      F_0 ← Cria-Bag()
6      Insere-Bag(F_0, v_0)
7      enquanto F_l ≠ ∅
8          F_{l+1} ← Cria-Bag()
9          Processa-Nível(G, F_l, F_{l+1}, Dist, l)
10         l ← l + 1
11     retorne Dist
```

Algoritmo 3: Processa-Nível [Leiserson and Schardl 2010].

```
   Entrada: estrutura in-bag, estrutura out-bag, Dist, l
1  início
2      parallel para k ← 0 até ⌊lg(|in-bag|)⌋
3          Se in-bag[k] ≠ NULL, então, Processa-Pennant(in-bag[k], out-bag, Dist, l)
```

Algoritmo 4: Processa-Pennant utilizada na implementação baseada no padrão OpenMP e adaptada da proposta em Cilk++ de [Leiserson and Schardl 2010].

```
   Entrada: estrutura in-pennant, estrutura out-bag, Dist, l
1  início
2      se |in-pennant| > GRAINSIZE, então,
3          aux-bag ← Cria-Bag()
4          para cada vértice u ∈ in-pennant faça
5              para cada vértice v ∈ Adj[u] faça
6                  se Dist[v] = ∞, então,
7                      Dist[v] ← l + 1
8                      Insere-Bag(aux-bag, v)
9          seção crítica Une-Bags(out-bag,aux-bag)
10         retorne
11     seção crítica se in-pennant ≠ NULL, então, new-pennant ← Divide-Pennant(in-pennant)
12     spawn Processa-Pennant(new-pennant, out-bag, l)
13     Processa-Pennant(in-pennant, out-bag, l)
14     sync
```

Para processar a estrutura *in-bag* F_l, a função Processa-Pennant (Algoritmo 4) verifica cada vértice $u \in$ *in-pennant* em paralelo (na linha 4) e examina em paralelo cada aresta (u, v). Uma importante diferença de nossa implementação em relação à implementação de Leiserson e Schardl, e também em relação à implementação anterior [Brandão et al. 2019], é que não paralelizamos o laço de repetição **para** interno na linha 5 da função Process-Pennant (Algoritmo 4). Isso foi feito porque gerou resultados melhores em investigação exploratória.

Se o vértice v ainda não foi visitado, isto é, Dist[v] = ∞ na linha 6, então, Dist[v] recebe o nível $l + 1$ na linha 7. Em seguida, a linha 8 insere v na estrutura *out-bag*, ou seja, no nível $l + 1$ da estrutura bag auxiliar, criada na linha 3. Essa bag auxiliar é responsável por armazenar temporariamente os vértices que são visitados localmente por cada thread. Essa modificação em relação à proposta de Leiserson e Schardl foi necessária pelo seguinte motivo. Se esses vértices fossem inseridos diretamente na bag de saída, que é compartilhada por todas as threads, poderia ocorrer conflito de concorrência durante a

operação de inserção. Portanto, para resolver esse conflito, seria necessário realizar essa operação em uma seção crítica que, por sua vez, se encontraria dentro de dois laços de repetição aninhados. Assim, o vértice recém-visitado é inserido na bag auxiliar e, na linha 9, ocorre a união da bag de saída com a estrutura bag auxiliar. Essa união de bags é melhor explicada adiante.

A atualização de Dist[v] na linha 7 do Algoritmo 4 causa uma condição de corrida, pois dois ou mais vértices que estejam sendo explorados em paralelo podem conter o mesmo vértice v em sua vizinhaça. Portanto, Dist[v] pode estar sendo atualizado simultaneamente por mais de uma thread. Porém, esse conflito não causa problemas e não precisa de bloqueios ou sessões críticas para solucioná-lo, uma vez que essas threads estariam explorando um mesmo nível e portanto atualizariam Dist[v] para um mesmo valor, causando apenas retrabalho [Leiserson and Schardl 2010].

Uma segunda condição de corrida ocorreria na linha 8 do Algoritmo 4 devido a inserções paralelas de vértices em *out-bag*. Para evitar essa condição de corrida, Leiserson e Schardl empregaram um redutor (*reducer*) utilizando o mecanismo União. A identidade para União, uma bag vazia, é criada por Cria-Bag [Leiserson and Schardl 2010]. Um redutor é um hiperobjeto implementado na linguagem Cilk++. Um hiperobjeto é um mecanismo na linguagem que permite diferentes ramificações de um programa multithread para manter visões locais coordenadas da mesma variável não local. Neste presente estudo, o redutor foi implementado por meio da criação de uma estrutura bag auxiliar na linha 3, inserção na bag auxiliar na linha 8 e, finalmente, união de bags na linha 9, que é realizada de forma atômica.

Para melhorar o desempenho da operação de inserção na estrutura bag, foram realizadas duas modificações importantes nas estruturas *pennant* e bag em relação à implementação de Leiserson e Schardl e, portanto, também em relação à implementação anterior [Brandão et al. 2019]. Primeiro, além de seus ponteiros, cada nó da estrutura *pennant* na bag armazena um *array* de tamanho constante igual a GRAINSIZE elementos, em vez de apenas um só elemento. Foi utilizado GRAINSIZE = 128 nos testes. Segundo, além do *backbone*, isto é, o *array* de *pennants*, a estrutura bag também mantém uma estrutura *pennant* adicional de tamanho GRAINSIZE chamado *hopper*, que ela preenche gradualmente. A seguir, enumeramos os impactos dessas modificações nas operações da estrutura bag.

1. O construtor da estrutura bag aloca espaço adicional para o *hopper*. Esse *overhead* é pequeno e ocorre apenas uma vez em cada estrutura bag.
2. A operação de inserção na estrutura bag primeiro tenta inserir o vértice no *hopper*. Se estiver cheio, então, ela insere o *hopper* no seu *backbone* e cria um novo *hopper* no qual ela insere o vértice. Essa otimização não muda o tempo de execução assintótico da operação de inserção e faz com que o programa execute significativamente mais rapidamente. Na maioria das vezes, a operação de inserção na estrutura bag simplesmente insere o elemento no *hopper*, cujo código é aproximadamente idêntico a inserir um elemento em um *array*. Apenas uma vez, a cada GRAINSIZE inserções, a inserção dispara inserção do *hopper*, que estará cheio, no *backbone* da estrutura bag.
3. Em uma operação de união de duas estruturas *bags* B_1 e B_2, primeiro determina-se qual estrutura bag tem o *hopper* menos cheio. Supondo-se que seja B_1, a

implementação modificada copia os elementos do *hopper* da estrutura bag B_1 para o *hopper* da estrutura bag B_2 até que este esteja cheio ou aquele fique sem elementos. Se o *hopper* da estrutura bag B_1 ficar sem elementos, a operação de união une os elementos dos *backbones* das estruturas *bags* B_1 e B_2 para produzir o *backbone* da estrutura bag resultante e utiliza o *hopper* da estrutura bag B_2 como o *hopper* da estrutura bag resultante. Todavia, se o *hopper* de B_2 ficar cheio, a operação de união de estruturas *bags* atribui esse *hopper* ao *carry* e o insere no *backbone* da estrutura bag resultante. Ainda, o *hopper* da estrutura bag B_1, agora contendo menos elementos, forma o *hopper* da estrutura bag resultante. No restante, a operação de união de estruturas *bags* é realizada conforme descrito por Leiserson e Schardl.

4. O laço de repetição **para** paralelo da linha 2 da função Processa-Nível (Algoritmo 3) chama a função Processa-Pennant com um *pennant* $F_l[k]$ (ou estrutura *in-bag*$[k]$) como argumento e o divide paralela e recursivamente até que as partes tenham tamanho unitário. Então, o conjunto de *pennants* unitárias é processado paralelamente.

5. Descrição dos testes

Nos experimentos, cada grafo foi carregado para a memória principal. Tanto a versão serial quanto a versão paralela, executada de duas até o número de *threads* disponíveis na máquina, foram executadas dez vezes em cada grafo, obtendo-se assim os tempos médios de execução, bem como os *speedups* da versão paralela em relação à versão serial. Os speedups foram calculados com T_1/T_i, variando i de 2 a 16, em que T_i e T_1 denotam os tempos médios da versão paralela e serial com a estrutura bag, respectivamente. Em particular, a versão serial com a estrutura bag foi mais rápida que a versão com fila.

Foram utilizados 20 grafos conexos obtidos da base de matrizes SuiteSparse [Davis and Hu 2011], sendo 10 digrafos e 10 grafos não direcionados. Os testes foram realizados em uma máquina com processador Intel® Core i9-9900K de 3,6 GHz, composta de oito núcleos físicos e oito núcleos lógicos, com 64 GiB de memória principal (4x16 GiB DIMM DDR4 2666MHz). O sistema operacional instalado nessa máquina era o Ubuntu 20.04.1 LTS. Foi utilizada a afinidade de thread default, isto é, noverbose, respect, granularity=core, tipo=none, permute=0 e offset=0.

A dificuldade em comparar com a implementação de Leiserson e Schardl [Leiserson and Schardl 2010] refere-se a utilizar a linguagem Cilk++, que foi descontinuada. Assim, para deixar este texto completo, na Tabela 1, são mostrados os oito grafos utilizados por Leiserson e Schardl. Nessa tabela, também constam os speedups alcançados pela proposta dos autores implementada na linguagem Cilk++. A máquina dos autores era composta de oito núcleos físicos com hyperthreading desabilitado.

Leiserson e Schardl obtiveram os grafos Grid3D200 e RMat23 de referências, mas não encontramos esses grafos. Assim, não temos as informações se são digrafos e seus números de componentes fortemente conectadas.

Os demais seis grafos utilizados por Leiserson e Schardl estão disponíveis na base SuiteSparse. Três desses grafos têm mais do que um componente fortemente conectado. Os autores não informaram qual o componente utilizado. Assim, nossa implementação poderia executar em um componente diferente do que aquele utilizado pelos autores. Dessa forma, preferimos não utilizar esses grafos.

Tabela 1. Grafos utilizados por Leiserson e Schardl [Leiserson and Schardl 2010], em que Compon. é o número de componentes fortemente conectadas.

Grafo	$\|V\|$	$\|E\|$	Speedup	Digrafo	Compon.
nlpkkt160	8.345.600	225.422.112	6,0	não	1
Grid3D200	8.000.000	55.800.000	4,9	não inf.	não inf.
cage15	5.154.859	99.199.551	5,3	sim	1
wikipedia-20070206	3.566.907	45.030.389	6,4	sim	1.203.340
Freescale1	3.428.755	17.052.626	5,2	sim	1.061
RMat23	2.300.000	77.900.000	6,5	não inf.	não inf.
kkt_power	2.063.494	12.771.361	6,0	não	9.611
cage14	1.505.785	27.130.349	5,3	sim	1

Nesse sentido, preferimos utilizar somente grafos conexos em nossos testes. Utilizamos os 10 maiores digrafos conexos da base SuiteSparse. Também utilizamos os 10 maiores grafos conexos não direcionados da base, com menos do que 3,6 bilhões de arestas. Assim, utilizamos os grafos cage14 e cage15 para comparar os resultados de nossa implementação com a proposta de Leiserson e Schardl, apesar de a comparação não ser direta por terem sido executadas em máquinas diferentes. Nesse sentido, não utilizamos o grafo nlpkkt160 porque utilizamos outros 10 grafos não direcionados maiores. O menor grafo não direcionado que utilizamos é composto de mais de 16 milhões de vértices.

6. Resultados

Na Tabela 2, são mostrados os speedups com oito e 16 threads de nossa implementação em C++ com OpenMP aplicada a 10 grafos não direcionados e 10 digrafos. Nas Figuras 1 e 2, são mostrados os speedups utilizando de 2 a 16 threads da nova implementação em C++ com OpenMP aplicada a 10 grafos não direcionados e 10 digrafos, respectivamente. Para uma melhor apresentação, os gráficos são apresentados em símbolos e linhas.

Tabela 2. Speedups com oito e 16 threads da nova implementação em C++ com OpenMP aplicada a 10 grafos não direcionados e 10 digrafos.

Grafo	$\|V\|$	$\|E\|$	#8	#16
europe_osm	50.912.018	108.109.320	4,2	5,3
nlpkkt240	27.993.600	760.648.352	5,6	7,8
GAP-road	23.947.347	57.708.624	4,5	5,4
road_usa	23.947.347	57.708.624	4,5	5,4
hugebubbles-00020	21.198.119	63.580.358	4,9	6,5
hugebubbles-00010	19.458.087	58.359.528	4,9	6,3
hugebubbles-00000	18.318.143	54.940.162	4,4	5,5
delaunay_n24	16.777.216	100.663.202	5,1	6,8
nlpkkt200	16.240.000	440.225.632	5,6	7,8
hugetrace-00020	16.002.413	47.997.626	4,6	5,8
Digrafo	$\|V\|$	$\|E\|$	#8	#16
cage15	5.154.859	99.199.551	4,8	6,7
rajat31	4.690.002	20.316.253	4,9	4,6
wiki-topcats	1.791.489	28.511.807	5,6	7,9
cage14	1.505.785	27.130.349	5,3	7,4
atmosmodl	1.489.752	10.319.760	4,4	5,0
atmosmodm	1.489.752	10.319.760	4,4	5,0
Hamrle3	1.447.360	5.514.242	5,7	6,8
atmosmodj	1.270.432	8.814.880	4,4	5,1
atmosmodd	1.270.432	8.814.880	4,4	5,0
tmt_unsym	917.825	4.584.801	3,2	2,9

O speedup de 5,3x da nova implementação em C++ por OpenMP ao ser aplicado no grafo cage14 foi o mesmo que o alcançado na proposta em Cilk++ de Leiserson e

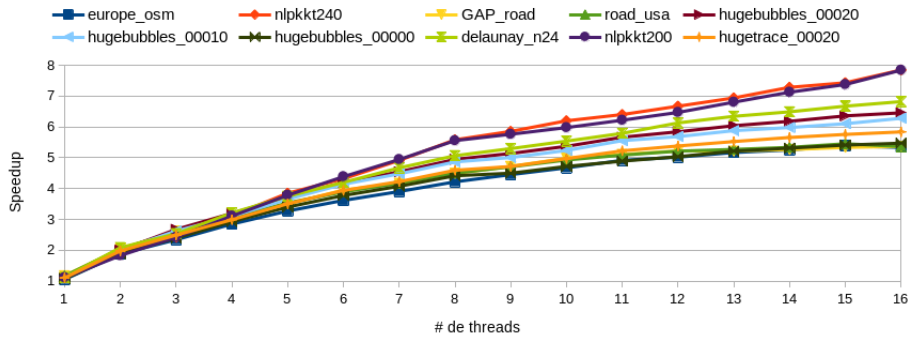

Figura 1. Speedups utilizando de 2 a 16 threads da nova implementação em C++ com OpenMP aplicada a 10 grafos não direcionados.

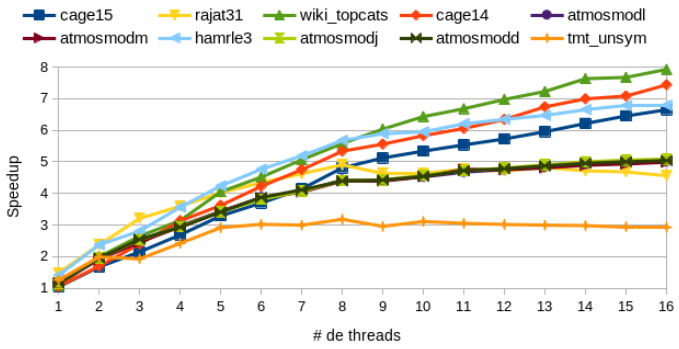

Figura 2. Speedups utilizando de 2 a 16 threads da nova implementação em C++ com OpenMP aplicada a 10 digrafos.

Schardl. A nova implementação em C++ por OpenMP obteve speedup de 4,8 ao ser aplicada no grafo cage15, enquando a proposta em Cilk++ de Leiserson e Schardl obteve speedup de 5,3x nesse grafo. Esses speedups são próximos e o demais speedups da nova implementação em C++ com OpenMP também foram competitivos com os speedups apresentados por Leiserson e Schardl.

Houve melhora do speedup com nossa implementação em quase todos os casos ao utilizar hyperthreading. A exceção foi no grafo tmt_unsym. Em particular, nossa implementação obteve speedups de 5,4x e 5,5x ao ser aplicada nos grafos europe_osm e road_usa com 15 threads, respectivamente.

7. Conclusões

Leiserson e Schardl [Leiserson and Schardl 2010] propuseram uma implementação da busca em largura em paralelo pela linguagem Cilk++ utilizando a estrutura bag em vez de utilizar a estrutura fila, tradicionalmente utilizada na versão serial desse algoritmo. Todavia, essa linguagem foi descontinuada. Pelo contrário, a linguagem C++ e o padrão OpenMP são largamente empregados em implementações paralelas. Assim, utilizamos a linguagem C++ e o padrão OpenMP para implementar a busca em largura com a estrutura bag proposta pelos autores. Para isso, fizemos algumas modificações em relação à proposta original de Leiserson e Schardl.

Nos testes em oito grafos, Leiserson e Schardl utilizaram uma máquina Intel Core

i7 composta de oito núcleos físicos, com *hyperthreading* desabilitado. A implementação dos autores apresentou speedup de ≈5x–6,5x em relação à versão serial. Nossos testes foram realizados em uma máquina i9 composta de oito núcleos físicos e oito núcleos lógicos. Em geral, a nova implementação em C++ por OpenMP obteve speedups de 4,4x a 5,7x com oito threads ao ser aplicada em 10 grafos não direcionados e 10 digrafos. Ao ser aplicada no grafo tmt_unsym, a implementação apresentou speedup de 3,2x e, no grafo europe_osm, a nova versão em C++ com OpenMP apresentou speedup de 4,2x. Em geral, a nova implementação apresentou speedup de 5x a 8x com 16 threads ao ser aplicada nos mesmos grafos. Ao ser aplicada no grafo tmt_unsym, a implementação apresentou speedup de 2,9x e, e no grafo rajat31, a nova versão em C++ com OpenMP apresentou speedup de 4,6x com 16 threads. Como continuação deste estudo, pretendemos investigar por que as acelerações nesses grafos ficaram menores que nos demais grafos. O speedup médio da implementação, com os grafos utilizados, foi de aproximadamente 5x com oito núcleos. Assim, podem ainda existir recursos a serem explorados. Precisamos investigar se é possível melhorar esse desempenho e verificar quais overheads estão limitando o desempenho. Precisamos analisar as características do grafo que limitam o speedup. Precisamos também investigar se é possível utilizar métricas para agrupar grafos de acordo com o desempenho.

Obtendo-se melhor desempenho, pretendemos executar a implementação em máquinas com processadores escaláveis Intel® Xeon®. Essas máquinas são compostas por dois sockets. Assim, pretendemos utilizar o padrão MPI para comunicação inter sockets e o padrão OpenMP para comunicação intra socket, conforme realizado por Cabral et al. [Cabral et al. 2020].

Também como continuação deste estudo, pretende-se incluir uma estratégia de *work chunking* na implementação [Hassaan et al. 2011]. Ainda, Tithi et al. [Tithi et al. 2013] apresentaram dois tipos paralelos sem bloqueio da busca em largura, juntamente com suas variantes, com base em filas de tarefas centralizadas e na estratégia *work-stealing* distribuído aleatório. Os algoritmos foram implementados em Cilk++ e apresentaram resultados melhores que a proposta de Leiserson e Schardl. Pretendemos também implementar essas abordagens com a linguagem de programação C++ com OpenMP. Também planejamos avaliar as extensões implementadas por Belova e Ouyang [Belova and Ouyang 2017]. Pretendemos comparar os resultados também com a compressão implementada no software Ligra+ [Shun et al. 2015].

Referências

Belova, M. and Ouyang, M. (2017). Breadth-first search with a multi-core computer. In *IEEE Int. Parallel and Distributed Processing Symposium Workshops*, pages 579–587.

Brandão, D., Coutinho, R., Silva, P. H. G., Assis, L. S., Sá, F. P. G., and Gonzaga de Oliveira, S. L. (2019). Estudo sobre o uso do framework openmp na paralelização de um algoritmo para o problema de busca em largura. In *Anais do LI Simpósio Brasileiro de Pesquisa Operacional (SBPO 2019)*, volume 2, page 108262, Limeira, SP. Sobrapo.

Cabral, F. L., Gonzaga de Oliveira, S. L., Osthoff, C., Costa, G. P., Brandão, D. N., and Kischinhevsky, M. (2020). An evaluation of MPI and OpenMP paradigms in finite-difference explicit methods for PDEs on shared-memory multi- and manycore systems. *Concurrency and Computation: Practice and Experience*, 32(20):e5642.

Chhugani, J., Satish, N., Kim, C., Sewall, J., and Dubey, P. (2012). Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency. In *Proc. of the 2012 IEEE 26th Int. Parallel and Distributed Processing Symposium*, pages 378–389.

Davis, T. A. and Hu, Y. (2011). The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1–25.

Gonzaga de Oliveira, S. L. and Silva, L. M. (2020a). An ant colony hyperheuristic approach for matrix bandwidth reduction. *Applied soft computing*, 94:106434.

Gonzaga de Oliveira, S. L. and Silva, L. M. (2020b). Evolving reordering algorithms using an ant colony hyperheuristic approach for accelerating the convergence of the ICCG method. *Engineering with Computers*, 36:1857–1873.

Gonzaga de Oliveira, S. L. and Silva, L. M. (2021). Low-cost heuristics for matrix bandwidth reduction combined with a Hill-Climbing strategy. *Rairo - Operations Research*, 55(4):2247–2264.

Hassaan, M. A., Burtscher, M., and Pingali, K. (2010). Ordered and unordered algorithms for parallel breadth first search. In *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, pages 539–540.

Hassaan, M. A., Burtscher, M., and Pingali, K. (2011). Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms. In *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 3–12.

Hong, S., Oguntebi, T., and Olukotun, K. (2011). Efficient parallel graph exploration on multicore CPU and GPU. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*, pages 100–113.

Leiserson, C. E. and Schardl, T. B. (2010). A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proc. of the 22nd annual ACM Symp. on Parallelism in algorithms and architectures*, pages 303–314.

Shun, J. and Blelloch, G. E. (2013a). Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, pages 135–146, New York. ACM.

Shun, J. and Blelloch, G. E. (2013b). Ligra: A lightweight graph processing framework for shared memory. *ACM SIGPLAN Notices*, 48(8):135–146.

Shun, J., Dhulipala, L., and Blelloch, G. E. (2015). Smaller and faster: Parallel processing of compressed graphs with ligra+. In *Data Compression Conference*, pages 403–412.

St. John, T., Dennis, J. B., and Gao, G. R. (2012). Massively parallel breadth first search using a tree-structured memory model. In *Proceedings of the 2012 Int. Workshop on Programming Models and Applications for Multicores and Manycores*, pages 115–123.

Suzumura, T., Ueno, K., Sato, H., Fujisawa, K., and Matsuoka, S. (2011). Performance characteristics of Graph500 on large-scale distributed environment. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 149–158.

Tithi, J. J., Matani, D., Menghani, G., and Chowdhury, R. A. (2013). Avoiding locks and atomic instructions in shared-memory parallel BFS using optimistic parallelization. In *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, IPDPSW*, pages 1628–1637.