MÁRCIO INÁCIO SANTANA

BUSCA EM LARGURA COM ESTRUTURA BAG E PADRÕES OPENMP E MPI EM MÁQUINAS DE ALTO DESEMPENHO

LAVRAS – MG

2021

MÁRCIO INÁCIO SANTANA

BUSCA EM LARGURA COM ESTRUTURA BAG E PADRÕES OPENMP E MPI EM MÁQUINAS DE ALTO DESEMPENHO

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

Prof. DSc. Sanderson L. Gonzaga de Oliveira
Orientador

LAVRAS – MG

2021

MÁRCIO INÁCIO SANTANA

BUSCA EM LARGURA COM ESTRUTURA BAG E PADRÕES OPENMP E MPI EM MÁQUINAS DE ALTO DESEMPENHO

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

APROVADO em 19 de novembro de 2021.

Prof. DSc. Sanderson L. Gonzaga de Oliveira	UFLA
Prof. DSc. Diego Nunes Brandão	CEFET-RJ
Prof. DSc. Denilson Alves Pereira	UFLA

Prof. DSc. Sanderson L. Gonzaga de Oliveira
Orientador

LAVRAS – MG
2021

Dedico aos meus pais, Antônio e Conceição, e a minha namorada, Aline.

AGRADECIMENTOS

Agradeço a Deus, por me permitir experienciar a existência.

Agradeço aos meus pais, Antônio e Conceição, por serem os seres humanos com as qualidades mais nobres que já conheci e sem os quais eu nada seria.

Agradeço a minha namorada, Aline, por ser minha fortaleza e fazer de mim uma pessoa melhor.

Agradeço ao meu Orientador, prof. Sanderson, pelo assencial apoio no desenvolvimento deste trabalho, pelo conhecimento transmitido, ajuda e pelas entrelinhas de seus ensinamentos, que me tornaram mais forte.

Agradeço à profa. Carla Osthoff pelo imprescindível apoio no projeto, disponibilizando acesso ao supercomputador SDumont do LNCC.

Agradeço ao prof. Diego Nunes pelo também imprescindível apoio no projeto, disponibilizando acesso à máquina "SKL" do CEFET-RJ e por suas valorosas contribuições para o melhoramento deste trabalho.

Agradeço ao prof. Denilson Alves por suas importantes contribuições para o aprimoramento do texto deste trabalho.

RESUMO

O desenvolvimento de implementações paralelas eficientes da busca em largura é de interesse fundamental, pois esse algoritmo é largamente utilizado como base em aplicações na ciência e na indústria. O objetivo deste trabalho foi desenvolver implementações paralelas da busca em largura utilizando a API *OpenMP* e a biblioteca *MPI* e a realização de amplo conjunto de testes em máquinas de alto desempenho para medir e analisar a performance dessas implementações. A metodologia consistiu na utilização da estrutura *bag*, por meio de adaptação do algoritmo *PBFS* de Leiserson e Schardl para duas versões de paralelização, uma com *OpenMP* puro ($PBFS_O$) e outra com *MPI* e *OpenMP* ($PBFS_H$). Foram aferidos os *speedups* das implementações paralelas, quando executadas sobre 63 grafos de teste em 2 máquinas de alto desempenho distintas. Os resultados obtidos por este trabalho foram o alcance de implementações funcionais do algoritmo *PBFS*, a descrição detalhada das versões adaptadas e a realização de testagem ampla em máquinas de alto desempenho com a medição dos *speedups* alcançados. As implementações desenvolvidas neste trabalho se mostraram, quando executadas sequencialmente, mais rápidas do que o algoritmo tradicional de busca em largura sequencial com fila (*BFS*). As médias geométricas dos tempos de execução sequenciais dos algoritmos $PBFS_O$ e $PBFS_H$, foram 5,93% e 4,82% menores, respectivamente, que a média geométrica dos tempos de execução do algoritmo *BFS*. O algoritmo $PBFS_O$ atingiu *speedups* máximos de 1,0 a 45,4, enquanto o $PBFS_H$ atingiu *speedups* máximos entre 1,0 e 23,3. A implementação que emprega *MPI* e *OpenMP* apresentou desempenho geral pior que a versão paralelizada por *OpenMP* puro, porém os resultados dos testes indicam um melhor uso do *hyperthreading* por parte daquela versão.

Palavras-chave: Algoritmos Paralelos. Busca em Largura. Estrutura bag. MPI. OpenMP. Teoria e algoritmos em grafos. Máquinas de alto desempenho.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Este capítulo tem como objetivos apresentar o tema abordado neste trabalho (*Seção 1.1*), bem como introduzir o problema juntamente com os objetivos (*Seção 1.2*), a justificativa para a pesquisa realizada (*Seção 1.3*) e, por fim, descrever como o restante deste trabaho encontra-se estruturado (*Seção 1.4*).

1.1 Contextualização e tema

Existe um grande e crescente volume de dados que são provenientes de tecnologias como telefones celulares, dados médicos, interações nas mídias sociais, experimentos científicos, etc. Um dos grandes desafios da atualidade é a realização da análise de dados originários de fontes que incluem redes sociais como Google, Facebook e Twitter, sistemas biológicos, classificação de páginas da web, pesquisa genômica e de experimentos da física de partículas, por exemplo. Tais dados podem ser modelados, para sua manipulação computacional por diversos algoritmos, como estruturas de dados *grafo*.

A busca em largura (*BFS*, de *Breadth-First Search Procedure*) é um dos algoritmos mais importantes estudados, por ser um algoritmo fundamental para percorrimento de *grafos*, sendo assim utilizado como base para diversos outros algoritmos com aplicações em diversas áreas da ciência e da indústria. A seguir são listadas algumas aplicações do algoritmo *BFS*:

- *Caminho mínimo* e *árvore geradora mínima* para grafos não ponderados: em um grafo não ponderado, o caminho mínimo é o caminho com o menor número de arestas. Com o algoritmo de busca em largura, sempre se alcança um vértice a partir de um vértice de origem percorrendo-se o menor número de arestas possível.

  Uma árvore T é chamada de *árvore geradora* de um grafo G se T é um subgrafo de G que possui todos os vértices de G, além disso, para grafos não ponderados, qualquer *árvore geradora* é uma *árvore geradora mínima* (aquela com custo total mínimo), que pode ser encontrada por meio da busca em largura.

- Redes *Peer to Peer(P2P)*: em redes *P2P*, como o *BitTorrent* (um sistema descentralizado de compartilhamento de arquivos), a busca em largura é utilizada para encontrar todos os vizinhos de um nó da rede.

- *Web crawlers* em ferramentas de busca: *Web crawlers* constroem índices usando a busca em largura. A ideia é começar a partir de uma página inicial, seguir todos os links encontrados e repetir o processo iterativamente nas novas páginas encontradas. A busca em profundidade também pode ser usada nos *web crawlers*, mas a vantagem no uso da busca em largura está no fato de que a altura da árvore construída pode ser limitada facilmente.

- Redes sociais: nas redes sociais, pode-se encontrar pessoas que estão a uma distância *'d'* de outra pessoa (distância no sentido de relações de amizade dentro da rede social) usando o algoritmo de busca em largura e iterando por *'d'* níveis.

- Sistemas de navegação por *GPS*: nesses sistemas de navegação a busca em largura pode ser utilizada para encontrar todas as localizações vizinhas de uma dada localização.

- *Broadcast* em uma rede: em redes de computadores, *broadcast* significa o envio de um pacote de informação a partir de um nó dessa rede para todos os nós receptores simultaneamente. Um pacote enviado via *broadcast* segue o algoritmo de busca em largura para chegar até todos os nós receptores.

- *Coleta de lixo*: em ciência da computação, a *coleta de lixo* é uma forma de gerenciamento automático de memória. O *coletor de lixo* tenta recuperar a memória que foi alocada pelo programa, mas não é mais referenciada - também chamada de *lixo*. A *coleta de lixo* foi inventada pelo cientista da computação americano John McCarthy por volta de 1959 para simplificar o gerenciamento manual de memória da linguagem de programação Lisp (MCCARTHY, 1960). A busca em largura é usada para copiar a *coleta de lixo* usando o algoritmo de Cheney, descrito em Cheney (1970).

- Redução de *largura de banda* em matrizes: 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 (OLIVEIRA; SILVA, 2019; OLIVEIRA; SILVA, 2020).

Sendo assim, o desenvolvimento de algoritmos de busca em largura rápidos e eficientes é de fundamental interesse. Neste trabalho, serão apresentadas duas implementações de variantes do algoritmo de busca em largura paralelo desenvolvido por Leiserson e Schardl (2010), algoritmo este denominado por esses autores de *PBFS*. Uma das versões foi desenvolvida empregando paralelização por *OpenMP* e a outra com paralelização híbrida por *MPI* e *OpenMP*.

Essas variações do algoritmo *PBFS* foram desenvolvidas com foco em sistemas de memória compartilhada, assim, para o caso da versão híbrida, o *MPI* foi utilizado para paralelização inter-soquete e o *OpenMP* para paralelização intra-soquete e, para a outra versão, o *OpemMP* foi utilizado para ambos os casos. Foram realizados experimentos com o objetivo de aferir-se os *speedups* alcançados pelas implementações em 2 máquinas de alto desempenho com 63 diferentes grafos de teste.

1.2 Problema e objetivos

Esta seção apresenta o problema e os objetivos que foram abordados neste trabalho. O propósito fundamental da *PBFS* é a execução de uma busca em largura, que pode ser definida da seguinte forma: dado um grafo $G(V,E)$, onde V é o conjunto de vértices e E o conjunto de arestas de G, o procedimento da busca em largura consiste em explorar os vértices de G a partir de um vértice de origem $s \in V$ de maneira crescente em relação à distância em número de arestas à qual os vértices se encontram da origem, ou seja, o primeiro vértice a ser explorado é s, em seguida, explora-se os vértices que podem ser alcançados percorrendo-se uma aresta a partir de s, na sequência os vértices que estão a duas arestas de distância de s são visitados e assim sucessivamente até que todos os vértices alcançáveis a partir da origem sejam visitados, produzindo-se assim uma árvore de busca em largura.

1.2.1 Problema

O problema da busca em largura paralela, consiste em resolver o problema da busca em largura utilizando multiplos processadores. Este estudo teve como base o algoritmo *PBFS* de Leiserson e Schardl (2010), que utilizou a estrutura de dados *bag* em substituição à estrutura *fila*. Assim, o problema abordado por este trabalho é expresso pela seguinte pergunta: Como criar novas versões do algoritmo *PBFS* de Leiserson e Schardl (2010) para desenvolver implementações da busca em largura paralela utilizando paralelização por *OpenMP* e por *MPI e OpenMP*, na linguagem de programação *C++*, e qual a performance dessas implementações?

1.2.2 Objetivo geral

O objetivo geral foi realizar um trabalho experimental que consistiu em desenvolver implementações de variações do algoritmo *PBFS* de Leiserson e Schardl (2010) na linguagem

de programação *C++* e nas quais a paralelização fosse feita por *OpenMP* e por *MPI* em conjunto com *OpenMP*, além da realização de um amplo conjunto de testes em máquinas de alto desempenho para medir a performance dessas implementações.

1.2.3 Objetivos específicos

Os objetivos específicos deste trabalho consistiram em:

1. Estudar conceitos necessários ao entendimento e adaptação do algoritmo *PBFS*;
2. Revisar estudos de trabalhos correlacionados ao tema;
3. Desenvolver uma versão do algoritmo *PBFS* com paralelização por *OpenMP* e descrever as adaptações feitas;
4. Desenvolver uma versão do algoritmo *PBFS* com paralelização por *MPI* e *OpenMP* e descrever as adaptações feitas;
5. Testar as implementações dos algoritmos em máquinas de alto desempenho aferindo-se os *speedups* alcançados.
6. Analisar os resultados dos testes em função das diferentes implementações realizadas.

1.3 Justificativa

Atualmente, o algoritmo de busca em largura é utilizado na composição de diversos algoritmos que têm extrema relevância na manipulação de dados de diversas fontes e que são utilizados em aplicações comerciais e também são fundamentais em muitas áreas de pesquisa. Neste sentido, um incremento da capacidade de processamento de dados contribui significativamente para o aumento da eficiência de aplicações utilizadas, por exemplo, na indústria e também para o avanço dos estudos em diversos campos (BRANDÃO et al., 2019). Esforços que visam aumentar a velocidade e a eficiência de algoritmos de busca geralmente se dão pela utilização de novas estruturas de dados e por novas técnicas de paralelização.

O algoritmo de busca em largura sequencial clássico (*Algoritmo 1*) utiliza uma estrutura de dados auxiliar chamada fila, uma estrutura muito eficiente cuja complexidade de suas operações de enfileirar e desenfileirar são da ordem de $O(1)$. Um dos desafios de implementações alternativas é encontrar uma estrutura que mantenha o tempo de acesso aos dados em $O(1)$,

mas na qual seja possível realizar-se o acesso paralelo; por isso, neste trabalho optou-se pela utilização da estrutura *bag*.

O algoritmo proposto por Leiserson e Schardl (2010) também utiliza a estrutura *bag*, porém com o foco em uma implementação em *Cilk++*, uma extensão às linguagens *C* e *C++* para suportar paralelismo de dados e de tarefas que foi descontinuada. Portanto, o desenvolvimento de implementações da *PBFS* com ferramentas de paralelização amplamente empregadas atualmente é relevante. Brandão et al. (2019) apresenta um estudo da utilzação do *OpenMP* na paralelização de um algoritmo de busca em largura, porém, o foco desse trabalho não foi na realização de testagem ampla ou de descrever adaptações feitas no algoritmo *PBFS*, mas sim de ser um verdadeiro tutorial da publicação de Leiserson e Schardl (2010).

Pelo exposto, a pesquisa deste trabalho se justifica no fato de haver uma lacuna de conhecimento em relação à descrição detalhada de adaptações da busca em largura paralela com *bags* direcionada para uma implementação em C++ com paralelização por *OpenMP* e *MPI*, além da realização de testagem ampliada em máquinas de alto desempenho.

1.4 Estrutura do trabalho

O *Capítulo 2* descreve as bases teóricas da *BFS* e da *PBFS*, exibe definições da estrutura de dados *bag* e sua estrutura componente, denominada *pennant*, bem como apresenta operações que podem ser aplicadas a essas estruturas de maneira a se obter as manipulações necessárias à *PBFS* e revisa trabalhos prévios correlacionados ao problema abordado neste trabalho. O *Capítulo 3* apresenta um breve resumo do roteiro de desenvolvimento do trabalho e datalha as adaptações feitas no algoritmo *PBFS* original. O *Capítulo 4* apresenta a classificação da pesquisa realizada, descreve as implementações da *BFS* e das variações da *PBFS*, mostra detalhes dos grafos de teste selecionados e seus critérios de seleção, detalha as espeficações das máquinas utilizadas e descreve os testes realizados e sobre quais parâmetros ocorreram. No *Capítulo 5*, são apresentados e analisados os resultados do trabalho e dos testes das implementações nas diferentes máquinas. Por fim, no *Capítulo 6*, são discutidos fatos relevantes sobre o desenvolvimento do trabalho e é realizado o direcionamento para futuras contribuições.

2 REFERENCIAL TEÓRICO

Este capítulo objetiva-se a apresentar suporte para os conceitos cuja compreensão foi necessária e serviram como fundamento para o desenvolvimento deste trabalho (*Seção 2.1*), além de apresentar o conhecimento prévio fruto de trabalhos correlatos (*Seção 2.2*).

2.1 Conceitos

A seguir, na *Seção 2.1.1*, são apresentados conceitos relacionados à busca em largura sequencial(*BFS*); na *Seção 2.1.2*, apresenta-se conceitos relativos à estrutura de dados (*bag*) e sua estrutura de dados componente chamada *pennant*, por fim, na *Seção 2.1.3*, são descritos conceitos referentes à busca em largura paralela(*PBFS*).

2.1.1 A busca em largura (*BFS*)

> A busca em largura é um dos algoritmos de busca em grafos mais simples e é o arquétipo para muitos algoritmos importantes. O algoritmo de árvore geradora mínima de Prim e o algoritmo de caminhos mínimos com fonte única de Dijkstra usam ideias similares àquelas da busca em largura. (CORMEN et al., 2009).

> Dado um grafo $G = (V, E)$ com o conjunto de vértices $V = V(G)$ e o conjunto de arestas $E = E(G)$, o problema da busca em largura(BFS) é calcular para cada vértice $v \in V$ a distância $Dist[v]$ à qual v se encontra de um vértice de origem $s \in V$. Mede-se a distância pelo número mínimo de arestas em um caminho de s para v em G. (CORMEN et al., 2009).

Observando o *Algoritmo 1*, temos a seguinte descrição:

> O procedimento serial padrão de busca em largura, [...], supõe que o grafo de entrada $G = (V, E)$ é representado usando uma lista de adjacências. Ele acrescenta vários atributos adicionais a cada vértice no grafo. Armazena-se a cor de cada vértice $u \in V$ no atributo $u.color$ e o predecessor de u no atributo $u.\pi$. Se u não tem predecessor(por exemplo, se $u = s$ ou u ainda não foi descoberto), então $u.\pi = NIL$. O atributo $u.d$ armazena a distância da fonte s até o vértice u computada pelo algoritmo. (CORMEN et al., 2009).

Complexidade da BFS

Ainda em relação ao *Algoritmo 1*, tem-se a seguinte análise:

> Após a inicialização, a busca em largura nunca colore um vértice de branco, e por isso o teste na linha 13 assegura que cada vértice é enfileirado no máximo uma vez, e portanto desenfileirado no máximo uma vez. As operações de

Algoritmo 1: Peseudocódigo do algoritmo BFS

```
BFS(G, s)
1   for each vertex u ∈ G.V − {s}
2       u.color = WHITE
3       u.d = ∞
4       u.π = NIL
5   s.color = GRAY
6   s.d = 0
7   s.π = NIL
8   Q = ∅
9   ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

enfileirar e desenfileirar levam tempo $O(1)$, então o tempo total devido a operações na pilha é $O(V)$. Porque o procedimento escaneia a lista de adjacências de cada vértice somente quando o vértice é desenfileirado, ele escaneia cada lista de adjacências no máximo uma vez. Como a soma dos comprimentos de todas as listas de adjacências é $O(E)$, o tempo gasto na inicialização é $O(V)$ e assim o tempo de execução total do procedimento *BFS* é $O(V+E)$. Portanto, a busca em largura executa em tempo linear no tamanho da lista de adjacências que representa G. (CORMEN et al., 2009).

2.1.2 A estrutura de dados *bag*

Esta seção tem por objetivo descrever a estrutura de dados *bag*, que é constituída por uma coleção de outras estruturas de dados chamadas *pennants*, assim sendo, primeiramente serão apresentados os conceitos relativos a esta estrutura e, na sequência, conceitos relacionados àquela estrutura de dados.

Pennants

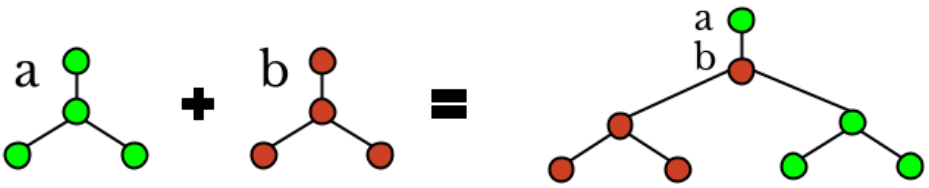
Um *pennant* é uma árvore de 2^k nós, onde k é um número inteiro não negativo. Cada nó x nesta árvore contém dois ponteiros para filhos: $x.left$ e $x.right$. A raiz da árvore tem apenas um filho esquerdo, que é uma árvore binária completa dos elementos restantes. (LEISERSON; SCHARDL, 2010).

Segundo Leiserson e Schardl (2010), "dois *pennants* x e y de tamanho 2^k podem ser combinados para formar um único *pennant* de tamanho 2^{k+1} em tempo $O(1)$ usando a função *PennantUnion*."

Segundo Leiserson e Schardl (2010), "a função *PennantSplit* realiza a operação inversa da função *PennantUnion* em tempo $O(1)$. Assume-se que o *pennant* de entrada contenha pelo menos 2 elementos."

Seja a um *pennant* com pelo menos dois elementos, assim a função $PennantSplit(a)$ pode ser descrita da seguinte forma: cria-se um novo *pennant* b que aponta para o filho da esquerda de a, o filho da esquerda de a passa a apontar para o filho da direita de b, o filho da direita de b aponta para $NULL$ e, por fim, retorna-se b. Pode-se visualizar a função $PennantSplit$ lendo a *Figura 2.1* da direita para a esquerda.

Figura 2.1 – União de *pennants*, dois *pennants* de tamanho 2^k se unem para formar um *pennant* de tamanho 2^{k+1}



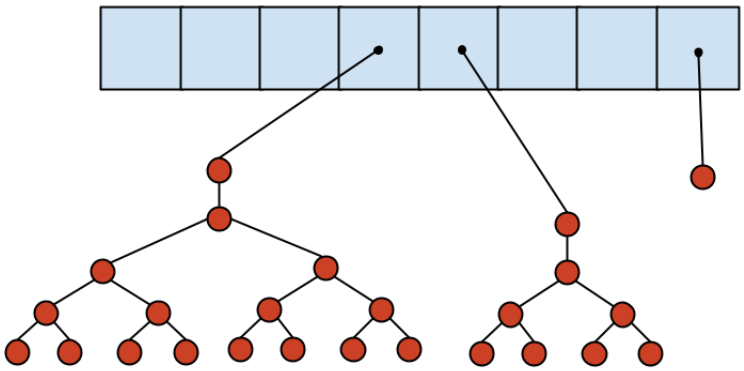
Fonte: Do autor(2021)

Bags

> Uma *bag* é uma coleção de *pennants*, dentre os quais não há dois de mesmo tamanho. A *PBFS* representa uma *bag* S usando um array de tamanho fixo $S[0..r]$, chamado de *backbone*, onde 2^{r+1} excede o número máximo de elementos que podem ser armazenados em uma bag. Cada entrada $S[k]$ no *backbone* contém um ponteiro nulo ou um ponteiro para um *pennant* de tamanho 2^k. (LEISERSON; SCHARDL, 2010).

A *Figura 2.2* apresenta uma representação gráfica de uma estrutura de dados *bag*.

Figura 2.2 – Uma *bag* com 3 *pennants* e com $25 = 00011001_2$ elementos



Fonte: Do autor(2021)

2.1.3 A busca em largura paralela (*PBFS*) com *bags*

> Desenvolvemos uma implementação multithread de busca em largura (BFS) em um grafo esparso usando as extensões Cilk++ para C++. Nosso programa *PBFS* em um único processador é executado tão rapidamente quanto uma implementação padrão em C++ da busca em largura. A *PBFS* alcança alta eficiência de trabalho usando uma nova implementação de uma estrutura de dados de multiconjuntos, chamada de "bag", no lugar da fila, normalmente empregada em algoritmos de busca em largura seriais. Para uma variedade de grafos de entrada de referência cujos diâmetros são significativamente menores do que o número de vértices - uma condição atendida por muitos grafos do mundo real - a *PBFS* demonstra bom speedup com o número de núcleos de processamento. (LEISERSON; SCHARDL, 2010).

Considerando o *Algoritmo 2*, segundo Leiserson e Schardl (2010): "Após a inicialização, a *PBFS* começa o laço de repetição **while** na linha 7 que iterativamente chama a função auxiliar *PROCESS-LAYER* para processar as camadas $d = 0, 1, ..., D$, onde D é o diâmetro do grafo de entrada G."

> Para processar $V_d = in\text{-}bag$, *PROCESS-LAYER* extrai cada vértice u contido em *in-bag* em paralelo e examina cada aresta (u, v) em paralelo. Se v ainda não tiver sido visitado - $v.dist$ é infinito (linha 17) - então a linha 18 atribui $v.dist = d + 1$ e a linha 19 insere v na bag de nível $d + 1$. (LEISERSON; SCHARDL, 2010).

Como pode ser observado no *Algoritmo 2*, a atualização de $v.dist$ na linha 18 causa uma condição de corrida, pois dois ou mais vértices que estejam sendo explorados em paralelo podem ambos conter o mesmo vértice v em sua vizinhança e, portanto, $v.dist$ pode estar sendo atualizado simultaneamente por duas *threads* diferentes. 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 *v.dist* para um mesmo valor, causando apenas retrabalho (LEISERSON; SCHARDL, 2010).

Uma segunda condição de corrida ocorre na linha 19 devido a inserções paralelas de vértices em $V_{d+1} = out\text{-}bag$. Para evitar essa condição de corrida foi empregada a funcionalidade redutor (*reducer*), fazendo de V_{d+1} um redutor de *bags*. onde *BAG-UNION* é a operação associativa utilizada pelo mecanismo de redução. A identidade para *BAG-UNION* - uma bag vazia - é criada por *BAG-CREATE* (LEISERSON; SCHARDL, 2010).

Algoritmo 2: Peseudocódigo do algoritmo *PBFS*

```
    PBFS(G, v_0)
1       parallel for each vertex v ∈ V(G) − {v_0}
2           v.dist = ∞
3       v_0.dist = 0
4       d = 0
5       V_0 = BAG-CREATE()
6       BAG-INSERT(V_0, v_0)
7       while ¬BAG-IS-EMPTY(V_d)
8           V_{d+1} = new reducer BAG-CREATE()
9           PROCESS-LAYER(revert V_d, V_{d+1}, d)
10          d = d + 1

    PROCESS-LAYER(in-bag, out-bag, d)
11      parallel for k = 0 to ⌊lg(BAG-SIZE(in-bag))⌋
12          if in-bag[k] ≠ NULL
13              PROCESS-PENNANT(in-bag[k], out-bag, d)

    PROCESS-PENNANT(in-pennant, out-bag, d)
14      if PENNANT-SIZE(in-pennant) < GRAINSIZE
15          for each u ∈ in-pennant
16              parallel for v ∈ G.Adj[u]
17                  if v.dist == ∞
18                      v.dist = d + 1
19                      BAG-INSERT(out-bag, v)
20      else
21          new-pennant = PENNANT-SPLIT(in-pennant)
22          spawn PROCESS-PENNANT(new-pennant, out-bag, d)
23          PROCESS-PENNANT(in-pennant, out-bag, d)
24          sync
```

Complexidade da PBFS

Nossa análise teórica da *PBFS* limita o trabalho adicional devido ao redutor de *bag* quando a condição de corrida é resolvida usando bloqueios

> de exclusão mútua. Teoricamente, em um gráfico G com um conjunto de vértices $V = V(G)$, um conjunto de arestas $E = E(G)$, diâmetro D e grau limitado, esta versão de "bloqueio" da *PBFS* realiza a BFS em tempo $O((V+E)/P + Dlg^3(V/D))$ em P processadores e exibe paralelismo efetivo $\Omega((V+E)/Dlg^3(V/D))$, que é considerável quando $D \ll V$, mesmo se o grafo é esparso. (LEISERSON; SCHARDL, 2010).

2.2 Trabalhos correlatos

Uma quantidade muito grande de versões paralelas da busca em largura já foi explorada. Neste trabalho, o foco foi a 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, Burtscher e Pingali (2010) implementaram várias otimizações para conseguir aceleração quase linear em relação à versão sequencial 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 sequencial, pois o algoritmo sequencial é simples e rápido por justamente utilizar a estrutura *fila*. Assim, Leiserson e 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 no 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*).

> Os métodos síncronos para a busca em largura percorrem o grafo em uma sequência de etapas de bloqueio delimitadas por pontos de sincronização, com cada etapa executando atualizações de todos os vértices em uma só profundidade. Por outro lado, abordagens assíncronas eliminam o uso de pontos de sincronização, embora isso possa resultar em várias atualizações para um só vértice e, consequente, ineficiência no trabalho. As abordagens síncronas funcionam bem para muitos grafos cujos diâmetros são pequenos em comparação

> com o número de vértices. Muitos grafos do mundo real apresentam essa característica. (CHHUGANI et al., 2012).

A implementação paralela de Leiserson e 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 durante a execução. (ST. JOHN; DENNIS; GAO, 2012).

Abordagens substituíram a estrutura *bag* por vetores de bits (BELOVA; OUYANG, 2017). Implementação trivial com vetores de bits da mesma forma que descrito em Leiserson e Schardl (2010) é apenas 38% mais lenta que o software *Ligra* (SHUN; BLELLOCH, 2013a; SHUN; BLELLOCH, 2013b), um dos códigos da busca em largura mais rápidos para execução em um único computador (BELOVA; OUYANG, 2017).

Em Brandão et al. (2019) implementou-se exatamente a mesma versão paralela de Leiserson e 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 sequencial. Brandão et al. (2019) é um verdadeiro tutorial da publicação de Leiserson e Schardl (2010), além disso, esse trabalho 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 (2010), que utilizou a linguagem de programação *Cilk++*.

3 DESENVOLVIMENTO

Este capítulo tem como objetivo apresentar as implementações desenvolvidas neste trabalho. A *Seção 3.1* primeiramente apresenta um breve roteiro sobre as etapas do desenvolvimento e, em seguida, na *Seção 3.1.1*, faz-se a descrição do algoritmo $PBFS_O$; na *Seção 3.1.2*, descreve-se o algoritmo $PBFS_H$ e, por fim, na *Seção 3.1.3*, são descritas as diferenças entre as implementações apresentadas em relação ao algoritmo *PBFS* de Leiserson e Schardl (2010).

3.1 Implementações

Primeiramente, buscou-se o estudo dos conceitos teóricos que envolvem o problema abordado e, posteriormente, realizou-se a investigação de publicações anteriores que abordaram o mesmo problema. Grande parte do estudo foi realizado com uma abordagem similar àquela feita por Leiserson e Schardl (2010), porém o algoritmo *PBFS* foi adaptado para dois tipos de implementação em *C++*, uma com paralilização utilizando apenas *OpenMP* e outra com palelização híbrida por *MPI* e *OpenMP*. Foi implementada uma versão serial clássica do algoritmo de busca em largura, que neste trabalho denota-se apenas como *BFS*, para servir como um comparativo de tempo de execução para as implementações da *PBFS* desenvolvidas.

A implementação do algoritmo *BFS* seguiu exatamente a estrutura do algoritmo original (*Algoritmo 1*) descrito em Cormen et al. (2009), enquanto as implementações do algoritmo *PBFS* tiveram como base o algoritmo, com todas as otimizações, descrito por Leiserson e Schardl (2010), mas houve modificações para que se tornasse viável a sua implementação em *C++* em suas duas versões de paralelização. A versão da PBFS com *OpenMP* puro, será denotada a seguir como $PBFS_O$, enquanto a versão híbrida será nomeada no restante deste trabalho de $PBFS_H$. O Algoritmo 3 apresenta o pseudocódigo do algoritmo $PBFS_O$ e o Algortimo 8 exibe o pseudocódigo do algoritmo $PBFS_H$.

3.1.1 Algoritmo $PBFS_O$

O algoritmo $PBFS_O$ é um algoritmo que busca descrever a implementção paralelizada da busca em largura que foi desenvolvida neste trabalho utilizando a linguagem de programação *C++* e a interface de programação *OpenMP* para implementar o paralelismo via múltiplas *threads* de execução. Esse algoritmo utiliza a estrutura de dados *bag*, conforme descrito em 2.1.2, para armazenar os vértices a serem processados. O $PBFS_O$ é nível-síncrono, o que significa que

o paralelismo é empregado aos vértices do nível de exploração mais atual durante a execução, mas cada nível de exploração é descoberto pelo algoritmo de forma sequencial.

O pseudocódigo do $PBFS_O$ está representado no *Algoritmo 3*. Nas linhas 1-4, o vetor que armazenará as distâncias de cada vértice inicial é criado (linha 1) e inicializado (linhas 2-4); todas as distâncias são inicializadas com ∞, exceto a do vértice inicial da busca (v_0), que é inicializada com 0. O nível atual é inicializado com 0 na linha 5. Na linha 6, cria-se e inicializa-se com uma *bag* vazia a *bag* que será responsável por armazenar os vértices que serão explorados no nível atual (B_{nivel_atual}) e, na linha 7, insere-se o vértice inicial da busca em B_{nivel_atual}. Na linha 8, cria-se o vetor de *bags* que conterá uma *bag* exclusiva para cada *thread* e que será responsável por aramazenar os vértices do próximo nível a ser explorado ($B_{proximo\_nivel}$). Nas linhas 9 e 10, essas *bags* são incializadas em paralelo, cada uma recebendo uma *bag* vazia.

O laço de repetição que se inicia na linha 11 é responsável por explorar os vértices do nível atual enquanto ele contiver algum vértice. A linha 12 chama o procedimento *PROCESSAR-NIVEL* que será responsável por explorar os vértices do nível atual e armazenar os vértices a serem explorados no nível seguinte em $B_{proximo\_nivel}$, esse procedimento será descrito em detalhes mais a diante. A linha 13 reinicializa o *hopper* de B_{nivel_atual} com um *pennant* vazio e a linha 14 faz de B_{nivel_atual} uma *bag* vazia (o que, na prática, significa fazer seu tamanho igual a 0). Nas linhas 15-17, ocorre a união de cada uma das $B_{proximo\_nivel}$ de cada *thread* com B_{nivel_atual}, o que significa que B_{nivel_atual} agora contém os vértices a serem explorados no próximo nível. A linha 18 incrementa o nível atual, ou seja, o nível atual agora é o que antes era o próximo nível.

Procedimento PROCESSAR-NIVEL

O procedimento *PROCESSAR-NIVEL* é responsável por explorar os vértices do nível atual, encontrando vértices adjacentes a eles e que devem ser armazenados para que possam ser explorados no nível seguinte. Esse procedimento tem como parâmetros, $bag_{entrada}$, que contém os vértices a serem explorados no nível atual, bag_{saida}, que é onde os vértices de exploração no nível seguinte serão armazenados e, n, que representa o nível atual da exploração.

As linhas 20-22, criam uma tarefa em paralelo que tem como objetivo executar o procedimento *PROCESSAR-PENNANT* no *hopper* de $bag_{entrada}$. O procedimento *PROCESSAR-PENNANT* é responsável por dividir um *pennant* em *pennants* de tamanho unitário (<= *GRAIN-*

Algoritmo 3: Peseudocódigo do algoritmo $PBFS_O$

entrada: Grafo $G(V,E)$ e vértice inicial v_0
saída : $Dists$ - Vetor de distâncias da árvore BFS

$PBFS_O(G, v_0)$
1 **declare** $Dists : vetor[1...|V|]$ de inteiros
2 **para em paralelo** $v \in V(G) - \{v_0\}$ **faça**
3 $Dists[v] \leftarrow \infty$
4 $Dists[v_0] \leftarrow 0$
5 $nivel \leftarrow 0$
6 $B_{nivel\_atual} \leftarrow CRIAR\text{-}BAG()$
7 $B_{nivel\_atual}.Inserir(v_0)$
8 **declare** $B_{proximo\_nivel} : vetor[0...|threads|-1]$ de bags
9 **seção paralela**
10 $B_{proximo\_nivel}[thread\_id] \leftarrow CRIAR\text{-}BAG()$
11 **enquanto** $B_{nivel\_atual} \neq \emptyset$ **faça**
12 $PROCESSAR\text{-}NIVEL\left(G, B_{nivel\_atual}, B_{proximo\_nivel}, nivel\right)$
13 $B_{nivel\_atual}.hopper \leftarrow CRIAR\text{-}PENNANT()$
14 $B_{nivel\_atual} \leftarrow \emptyset$
15 **seção paralela**
16 **seção crítica**
17 $B_{nivel\_atual}.UNIAO\left(B_{proximo\_nivel}[thread\_id]\right)$
18 $nivel \leftarrow nivel + 1$
19 **retorna** $Dists$

$PROCESSAR\text{-}NIVEL(bag_{entrada}, bag_{saida}, n)$
20 **seção paralela**
21 **single**
22 **task** $PROCESSAR\text{-}PENNANT(bag_{entrada}.hopper, bags_{saida}, n)$
23 **para** $k \leftarrow 0$ **até** $\left\lfloor lg\left(\frac{|bag_{entrada}|-|bag_{entrada}.hopper|}{GRAINSIZE}\right)\right\rfloor$ **faça**
24 **se** $bag_{entrada}[k] \neq NULL$ **então**
25 $PROCESSAR\text{-}PENNANT(bag_{entrada}[k], bags_{saida}, n)$
26 $bag_{entrada}[k] \leftarrow NULL$

$PROCESSAR\text{-}PENNANT(pennant_{entrada}, bags_{saida}, n)$
27 **se** $|pennant_{entrada}| \leq GRAINSIZE$ **então**
28 **para** $u \in pennant_{entrada}$ **faça**
29 **para** $v \in G.Adj[u]$ **faça**
30 **se** $Dists[v] = \infty$ **então**
31 $Dists[v] \leftarrow n+1$
32 $bags_{saida}[thread\_id].Inserir(v)$
33 **senão**
34 $novoPenn \leftarrow pennant_{entrada}.Dividir()$
35 **task** $PROCESSAR\text{-}PENNANT(novoPenn, bags_{saida}, n)$
36 $PROCESSAR\text{-}PENNANT(pennant_{entrada}, bags_{saida}, n)$

SIZE) para então explorá-los, esse procedimento será descrito em detalhes mais adiante. As linhas 23-26, são responsáveis por percorrer o *backbone* de $bag_{entrada}$ em paralelo e chamar o procedimento *PROCESSAR-PENNANT* em cada *pennant* não nulo encontrado.

Procedimento PROCESSAR-PENNANT

O procedimento *PROCESSAR-PENNANT* recebe os seguintes parâmetros: o *pennant* a ser processado ($pennant_{entrada}$), bag_{saida} (o vetor de *bags* que aramazenará os vértices que serão processados no nível seguinte) e n (o nível atual). Esse procedimento divide $pennant_{entrada}$ paralela e recursivamente (linhas 34-36) até que os *pennants* resultantes possuam tamanho menor ou igual ao tamanho unitário definido (*GRAINSIZE*). A linha 27 verifica se o *pennant* que está sendo processado já atingiu um tamanho menor ou igual a *GRAINSIZE*. Se o tamanho de $pennant_{entrada}$ for menor que *GRAINSIZE*, então, a linha 28 percorre cada vértice u desse *pennant* e a linha 29 percorre todos os vértices v adjacentes a u, para todo u. A linha 30, verifica se v já foi visitado anteriormente e, em caso positivo, as linhas 31 e 32 atualizam a distância de v em relação a v_0 para $n+1$ e inserem v na bag_{saida} da *thread* que visitou v, respectivamente. Se o tamanho de $pennant_{entrada}$ for maior que *GRAINSIZE*, as linhas 34-36 continuam a dividi-lo.

Método CRIAR-BAG

O método *CRIAR-BAG* é responsável por instanciar uma nova *bag* vazia. O pseudocódigo desse método está representado no *Algoritmo 4*. Na linha 1, uma referência para uma nova instância de um objeto *bag* é criada. Na linha 2, o *hopper* da nova *bag* é criado e inicializado com um *pennant* vazio. Na linha 2, o *backbone* da nova *bag* é criado, esse *backbone* é um vetor com capacidade para $r+1$ ponteiros para *pennants*, onde $r = \left\lfloor lg\left(\frac{|V|}{GRAINSIZE}\right)\right\rfloor$ e V é o conjunto de vértices do grafo de entrada do $PBFS_O$. Nas linhas 4-5, todas as posições do *backbone* recém criado são inicializadas paralelamente, cada uma recebendo um ponteiro nulo (NULL). Por fim, na linha 7, a referência para a *bag* recém criada é retornada.

Método UNIAO

UNIAO é um método com a finalidade de unir duas *bags*. A *bag* chamadora desse método conterá, ao final da operação, o resultado da união, enquanto, a bag passada como argumento estará vazia, ou seja, com todos os ponteiros para *pennants* de seu *backbone* iguais a *NULL* e com seu *hopper* apontando para um *pennant* vazio.

Algoritmo 4: Peseudocódigo do método *CRIAR-BAG*

```
   saída  : Bag vazia recém criada
   CRIAR-BAG()
1      bag {
2          hopper ← CRIAR-PENNANT()
3          declare backbone : vetor[0...r] de pennants
4          para em paralelo k ← 0 até r faça
5              backbone[k] ← NULL
6      }
7      retorna bag
```

O *Algoritmo 5* apresenta o pseudocódigo do método *UNIAO*. Nas linhas 1 e 2, dois *hoppers* (ponteiros para *pennants*), são criados e inicializados com ponteiros nulos. O objetivo desses *hoppers* é armazenar uma referência para o *hopper*, dentre os *hoppers* das duas *bags* sendo unidas, com o menor número de vértices (em H_{menos_cheio}) e com o maior número de vértices (em H_{mais_cheio}); isso é feito pelas linhas 3-8.

Nas linhas 9-11, os elementos do *hopper* menos cheio são copiados para o *hopper* mais cheio até que o *hopper* menos cheio fique sem elementos ou até que o *hopper* mais cheio esteja totalmente cheio. Na linha 12, *y* (o carry para a função *FA* descrita em Leiserson e Schardl (2010)) é inicializado com um ponteiro nulo. A liha 13 verifica se o *hopper* menos cheio ficou sem elementos, em caso positivo, na linha 14, o *hopper* da *bag* chamadora passa a apontar para o *hopper* mais cheio, em caso negativo, na linha 16, *y* passa a apontar para o *hopper* mais cheio e, na linha 17, o *hopper* da *bag* chamadora passa a apontar para o *hopper* menos cheio.

As linhas 18-19 percorrem os *backbones* das duas *bags* participantes da união até a máxima posição possivelmente ocupada pelo resultado dessa operação e executa a união utilizando a função *FA*, conforme descrita em Leiserson e Schardl (2010). A linha 20 cria um novo *hopper* vazio para a *bag* passada como argumento.

Método CRIAR-PENNANT

O método *CRIAR-PENNANT* instancia um novo *pennant* e retorna um ponteiro para ele. O *Algoritmo 6* representa o pseudocódigo do método *CRIAR-PENNANT*. Na linha 1, uma referência para um novo *pennant* é criada. A linha 2 cria o *noh* raiz do *pennant* recém criado e a linha 4 retorna um ponteiro para esse *pennant*.

Algoritmo 5: Peseudocódigo do método *UNIAO*

entrada: Bag B_2, que será unida à *bag* chamadora
$UNIAO(B_2)$
1 $H_{menos\_cheio} \leftarrow NULL$
2 $H_{mais\_cheio} \leftarrow NULL$
3 **se** $|this.hopper| < |B_2.hopper|$ **então**
4 $H_{menos\_cheio} \leftarrow this.hopper$
5 $H_{mais\_cheio} \leftarrow B_2.hopper$
6 **senão**
7 $H_{menos\_cheio} \leftarrow B_2.hopper$
8 $H_{mais\_cheio} \leftarrow this.hopper$
9 **enquanto** $H_{menos\_cheio} \neq \emptyset \wedge |H_{mais\_cheio}| < r$ **faça**
10 $H_{mais\_cheio}[|H_{mais\_cheio}|] \leftarrow H_{menos\_cheio}[|H_{menos\_cheio}| - 1]$
11 $H_{menos\_cheio}[|H_{menos\_cheio}| - 1] \leftarrow NULL$
12 $y \leftarrow NULL$
13 **se** $H_{menos\_cheio} = \emptyset$ **então**
14 $this.hopper \leftarrow H_{mais\_cheio}$
15 **senão**
16 $y \leftarrow H_{mais\_cheio}$
17 $this.hopper \leftarrow H_{menos\_cheio}$
18 **para** $k \leftarrow 0$ **até** $\left\lfloor lg\left(\frac{|this|+|B_2|}{GRAINSIZE}\right)\right\rfloor$ **faça**
19 $(this.backbone[k], y) \leftarrow FA(this.backbone[k], B_2.backbone[k], y)$
20 $B_2.hopper \leftarrow$ *CRIAR-PENNANT* ( )

Algoritmo 6: Peseudocódigo do método *CRIAR-PENNANT*

saída : *Pennant* vazio recém criado
CRIAR-PENNANT ()
1 **pennant** {
2 $raiz \leftarrow$ *CRAR-NOH* ( )
3 }
4 **retorna pennant**

Método CRIAR-NOH

O método *CRIAR-NOH* instancia um novo *noh*, que é a estrutura de dados onde os vértices estão armazenados e também são as unidades que formam os *pennants*. O *Algoritmo 7* apresenta o pseudocódigo do algoritmo *CRIAR-NOH*. Na linha 1, uma referência para um novo *noh* é criada. A linha 2 cria o vetor de vértices (inteiros) de tamanho *GRAINSIZE* do *noh* e as linhas 3 e 4 criam e inicializam, com ponteiros nulos, os ponteiros para os filhos esquerdo e direito do *noh* recém criado. A linha 5 retorna um ponteiro para o novo *noh* criado. Note que, em *C++*, os vetores de inteiros recém alocados no *heap* têm as suas posições inicializadas por

padrão com o valor 0, assim, não há necessidade de inicializar cada posição do vetor de vértices do novo *noh* criado.

Algoritmo 7: Peseudocódigo do método *CRIAR-NOH*

```
   saída  : Noh vazio recém criado
   CRIAR-NOH ()
1      noh {
2          declare vertices : vetor[1...GRAINSIZE] de inteiros
3          esquerdo ← NULL
4          direito ← NULL
5      }
6      retorna noh
```

3.1.2 Algoritmo $PBFS_H$

O algoritmo $PBFS_H$ é similar ao algoritmo $PBFS_O$ descrito anteriormente, porém com algumas diferenças importantes. O $PBFS_H$ utiliza paralelização híbrida por *MPI* e *OpenMP*, nele são utilizados dois processos *MPI* que devem executar em uma mesma máquina, pois esse algoritmo pressupõe a utilização de memória compartilhada. Cada um dos processos *MPI* podem disparar várias *threads* via a interface *OpenMP*. O algoritmo $PBFS_H$ não é nível-síncrono, pois é possível que um dos processos esteja explorando um nível diferente do outro em um determinado momento da execução. O trabalho é dividido entre os processos da seguinte forma: o conjunto de vértices do nível 1 é dividido igualmente entre os dois processos antes da exploração iniciar, isso faz com que o trabalho seguinte de exploração dos demais níveis também seja dividido, no entanto, é possível que haja intersecção entre o conjunto de vértices explorado por cada processo nos níveis seguintes.

O pseudocódigo do $PBFS_H$ está representado no *Algoritmo 8*. Nas linhas 1 e 2, são declaradas e inicializadas com o valor 0 as variáveis para armazenar o tamanho dos vetores de distâncias que cada processo irá alocar e, nas linhas 3 e 4, esses vetores são criados e inicializados com ponteiros nulos. As linhas 5 e 6 garantem que o vetor de distâncias que o processo 0 alocará terá tamanho igual ao número de vértices do grafo e as linhas 7 e 8 fazem o mesmo mas para o processo 1. Nas linhas 9 e 10, cada um dos processos aloca seu respectivo vetor de distâncias *BFS* em uma janela de memória compartilhada. Na linha 11, é criada e inicializada com NULL a variável *Dists*, que será responsável por apontar para o vetor de distâncias do processo corrente. Nas linhas 12-17, cada processo obtém uma referência para o vetor de distâncias *BFS*

do outro processo consultando a janela de memória alocada por este e a variável *Dists* é apontada para o vetor de distâncias do processo corrente. Nas linhas 18 e 19 cada processo inicializa em paralelo seu respectivo vetor de distâncias com o valor ∞ em cada posição. Na linha 20, a execução dos dois processos é sincronizada. Da linha 21 até a linha 35, a execução ocorre da mesma forma como descrito para as linhas 4-18 do *Algoritmo 3* ($PBFS_O$), com a diferença que esse trecho estará sendo executado simultâneamente em dois processos *MPI* diferentes.

O procedimento *PROCESSAR-NIVEL* do $PBFS_H$ é idêntico ao procedimento de mesmo nome do *Algoritmo 3* ($PBFS_O$), enquanto o procedimento *PROCESSAR-PENNANT* é diferente no $PBFS_H$. No algoritmo $PBFS_H$, assim como no $PBFS_O$, o procedimento *PROCESSAR-PENNANT* também divide $pennant_{entrada}$ paralela a recursivamente até que os *pennants* resultantes possuam um tamanho menor ou igual a *GRAINSIZE* (linhas 61-64). As linhas 50-60 do algoritmo $PBFS_H$ também são responsáveis por explorar os vértices dos *pennants* de tamanho menor ou igual a *GRAINSIZE*, assim como descrito para as linhas 27-32 do algoritmo $PBFS_O$, porém, no $PBFS_H$, as linhas 51-56 dividem igualmente os vértices a serem explorados no nível 1 entre os processos 0 e 1.

Ao atingir-se o ponto de sincronização entre os processos na linha 36, cada processo explorou seu conjunto parcial de vértices do grafo G de entrada, conforme a divisão a partir nível 1 mencionada anteriormente, e, portanto, eles preencheram parcialmente seus respectivos vetores de distâncias *BFS*. Assim, as linhas 37-42 são responsáveis por fazer a união desses vetores em $Dists_0$, gerando o vetor com todas as distâncias *BFS* correspondentes a cada vértice de G, que é retornado na linha 43.

3.1.3 Diferenças em relação ao algoritmo *PBFS*

A principal diferença entre as implementações desenvolvidas neste trabalho e o algoritmo *PBFS* de Leiserson e Schardl (2010) é que as implementações apresentadas aqui utilizaram a linguagem de programação *C++* com *OpenMP* e/ou *MPI* enquanto a implementação desenvolvida por aqueles autores utilizou a linguagem de programação *Cilk++*. Além disso, o $PBFS_O$ e o $PBFS_H$ têm algumas diferenças estruturais significativas em relação ao *PBFS*, conforme descrito a seguir.

Utilizamos um vetor de *bags*, uma *bag* exclusiva para cada *thread*, para armazenar os vértices do próximo nível a ser explorado pela busca onde Leiserson e Schardl (2010) emprega-

ram um *hiperobjeto redutor de bags*. Esse vetor de *bags* possibilita as inserções sem bloqueios ou seções críticas observadas nas linhas 32 e 60 dos *Algoritmos 3 e 8*, respectivamente.

Tanto no algoritmo $PBFS_O$ quanto no algoritmo $PBFS_H$, optou-se por não se paralelizar o laço de repetição que visita a lista de adjências de um vértice u que é explorado pelo procedimento que tem objetivo de processar *pennants*, ou seja, não se paralelizou o laço de repetição da linha 29 do *Algoritmo 3* nem o laço de repetição da linha 57 do *Algoritmo 8*; a razão dessa adaptação é que, empiricamente, observou-se a obtenção de melhores resultados com a não paralelização dos referidos laços de repetição.

Algoritmo 8: Peseudocódigo do algoritmo $PBFS_H$

entrada: Grafo $G(V,E)$ e vértice inicial v_0
saída : $Dists$ - Vetor de distâncias da árvore BFS
$PBFS_H(G, v_0)$

1 $tamDists_0 \leftarrow 0$
2 $tamDists_1 \leftarrow 0$
3 $Dists_0 \leftarrow NULL$
4 $Dists_1 \leftarrow NULL$
5 **se** $RANK = 0$ **então**
6 $\quad tamDists_0 \leftarrow |V|$
7 **senão**
8 $\quad tamDists_1 \leftarrow |V|$
9 $MEM_0 \leftarrow ALOCAR\text{-}MEM\text{-}COMPARTILHADA(Dists0, tamDists_0, inteiro)$
10 $MEM_1 \leftarrow ALOCAR\text{-}MEM\text{-}COMPARTILHADA(Dists1, tamDists_1, inteiro)$
11 $Dists \leftarrow NULL$
12 **se** $RANK = 0$ **então**
13 $\quad Dists \leftarrow Dists_0$
14 $\quad Dists_1 \leftarrow CONSULTAR\text{-}MEM\text{-}COMPARTILHADA(MEM_1)$
15 **senão**
16 $\quad Dists \leftarrow Dists_1$
17 $\quad Dists_0 \leftarrow CONSULTAR\text{-}MEM\text{-}COMPARTILHADA(MEM_0)$
18 **para em paralelo** $v \in V(G) - \{v_0\}$ **faça**
19 $\quad Dists[v] \leftarrow \infty$
20 **sincronizar processos**
21 $Dists[v_0] \leftarrow 0$
22 $nivel \leftarrow 0$
23 $B_{nivel\_atual} \leftarrow CRIAR\text{-}BAG()$
24 $B_{nivel\_atual}.Inserir(v_0)$
25 **declare** $B_{proximo\_nivel} : vetor[0...|threads|-1]$ de bags
26 **seção paralela**
27 $\quad B_{proximo\_nivel}[thread\_id] \leftarrow CRIAR\text{-}BAG()$
28 **enquanto** $B_{nivel\_atual} \neq \emptyset$ **faça**
29 $\quad PROCESSAR\text{-}NIVEL\left(G, B_{nivel\_atual}, B_{proximo\_nivel}, nivel\right)$
30 $\quad B_{nivel\_atual}.hopper \leftarrow CRIAR\text{-}PENNANT()$
31 $\quad B_{nivel\_atual} \leftarrow \emptyset$
32 $\quad$ **seção paralela**
33 $\quad\quad$ **seção crítica**
34 $\quad\quad\quad B_{nivel\_atual}.UNIAO\left(B_{proximo\_nivel}[thread\_id]\right)$
35 $\quad nivel \leftarrow nivel + 1$
36 **sincronizar processos**
37 $minVertice \leftarrow 1 + RANK \times \left\lceil \frac{|V|}{2} \right\rceil$
38 $maxVertice \leftarrow MIN\left((RANK+1) \times \left\lceil \frac{|V|}{2} \right\rceil, |V|\right)$
39 **para em paralelo** $i \leftarrow minVertice$ **até** $maxVertice$ **faça**
40 $\quad$ **se** $(Dists_0[i] \neq \infty) \wedge (Dists_1[i] < Dists_0[i])$ **então**
41 $\quad\quad Dists_0[i] \leftarrow Dists_1[i]$

```
42  sincronizar processos
43  retorna Dists_0

    PROCESSAR-NIVEL(bag_entrada, bags_saida, n)
44  seção paralela
45      single
46          task PROCESSAR-PENNANT (bag_entrada.hopper, bags_saida, n)
47      para k ← 0 até ⌊lg((|bag_entrada| − |bag_entrada.hopper|) / GRAINSIZE)⌋ faça
48          se bag_entrada[k] ≠ NULL então
49              PROCESSAR-PENNANT(bag_entrada[k], bags_saida, n)
50              bag_entrada[k] ← NULL

    PROCESSAR-PENNANT (pennant_entrada, bags_saida, n)
51  se |pennant_entrada| ≤ GRAINSIZE então
52      minVertice ← 1
53      maxVertice ← tamPenn
54      se n = 1 então
55          minVertice ← 1 + RANK × ⌈|pennant_entrada| / 2⌉
56          maxVertice ← MIN((RANK + 1) × ⌈|pennant_entrada| / 2⌉, |pennant_entrada|)
57      para u ← minVertice até maxVertice faça
58          para v ∈ G.Adj[u] faça
59              se Dists[v] = ∞ então
60                  Dists[v] ← n + 1
61                  bags_saida[thread_id].Inserir(v)
62  senão
63      novoPenn ← pennant_entrada.Dividir()
64      task PROCESSAR-PENNANT(novoPenn, bag_saida, n)
65      PROCESSAR-PENNANT(pennant_entrada, bag_saida, n)
```

4 METODOLOGIA

Este capítulo tem como objetivo apresentar a classificação da pesquisa realizada (*Seção 4.1*) e a descrição dos materiais e métodos empregados na elaboração deste trabalho (*Seção 4.2*).

4.1 Classificação da pesquisa

A pesquisa desenvolvida neste trabalho é classificada, quanto à natureza, em pesquisa básica, pois seus produtos não são diretamente aplicados mas sim usados por outros algoritmos que podem ou não ser diretamente empregados em aplicações práticas. Quanto à abordagem, em pesquisa quantitativa, pois fundamenta-se essencialmente em quantidades numéricas, linguagem matemática e técnicas estatísticas para descrever os fenômenos observados. Quanto aos objetivos, em pesquisa explicativa, pois utiliza-se de métodos experimentais e conhecimentos anteriores para adaptar um algoritmo preexistente e busca explicar, em certo nível, o seu comportamento. Quanto aos procedimentos técnicos, em pesquisa experimental, pois utiliza-se de equipamentos com variáveis que podem ser controladas para determinar a influência destas no objeto de estudo; classifica-se também em pesquisa bibliográfica, pois realizou-se o estudo de conceitos teóricos para embasar o conhecimento a ser produzido bem como de trabalhos correlatos.

4.2 Procedimentos metodológicos

As implementações dos algoritmos $PBFS_O$ e $PBFS_H$ desenvolvidos nesta pesquisa foram realizadas utilizando a linguagem de programação C++, a interface *OpenMP* e o framework *MPI*, conforme foi explicado no *Capítulo 3*. A *Seção 4.2.1* apresenta os grafos utilizados para testar esses algoritmos. A *Seção 4.2.2* descreve as máquinas onde os testes foram realizados. Por fim, a *Seção 4.2.3* apresenta uma descrição detalhada dos testes realizados.

4.2.1 Grafos de teste

As implementações dos algoritmos BFS, $PBFS_O$ e $PBFS_H$ foram testadas em 63 grafos divididos em dois conjuntos, aqueles com apenas uma componente fortemente conectada, que neste trabalho denominaremos de grafos *conexos* e aqueles com mais de uma componente fortemente conectada, que chamaremos de grafos *não conexos*. O conjunto dos grafos *conexos*

possui 32 grafos e, o dos grafos *não conexos*, 31; a razão para quantidades diferentes de grafos nos dois conjuntos é que houve 2 grafos *conexos* e 1 *não conexo* que executaram com o algoritmo $PBFS_O$ mas não com o $PBFS_H$, devido a limitações de memória principal em uma das máquinas de teste. Todos os grafos foram selecionados da base de dados *SuiteSparse Matrix Collection*, apresentada por Davis e Hu (2011). Os critérios de seleção dos grafos foram os seguintes:

- **Grafos conexos direcionados:** Os 15 maiores da base de dados em número de arestas.
- **Grafos conexos não direcionados:** Os 15 maiores da base de dados em número de arestas, com adição do 16º e 17º maiores para testes na máquina *SKL* (4.2.2), já que o 1º e o 2º maiores grafos *conexos* não direcionados não executaram com o algoritmo $PBFS_H$ nessa máquina.
- **Grafos não conexos direcionados:** Os 15 maiores da base de dados em número de arestas.
- **Grafos não conexos não direcionados:** Do 2º ao 16º maiores da base de dados em número de arestas, com adição do 17º maior para testes na máquina *SKL* (4.2.2), já que o 2º maior grafo não conexo não direcionado não executou com o algoritmo $PBFS_H$ nessa máquina. O 1º maior grafo desse tipo não foi incluído porque ele não executa com nenhum dos algoritmos implementados, em nenhuma das máquinas de teste, devido a sua grande ocupação de memória principal.

O motivo de se ter incluído grafos de teste *não conexos* foi pelo fato de que esses grafos são, em geral, maiores que os grafos *conexos* na base de dados utilizada e, como queríamos testar as implementações com o máximo de carga (no sentido de utilização de CPU e de memória principal) possível nas máquinas de teste, então inicialmente acreditamos que incluir os grafos *não conexos* era uma escolha razoável. No entanto, ao longo do desenvolvimento, foi observado que a escolha do vértice inicial poderia afetar significativamente o desempenho das implementações uma vez que esse vértice definiria a componente fortemente conectada (CFC) que seria explorada pela busca em largura. Uma CFC muito pequena pode produzir um desempenho ruim das implementações porque a pouca quantidade de vértices e arestas a serem percorridas impactaria negativamente no paralelismo, por exemplo. Em geral, as CFCs percorridas pelas implementações nos testes realizados com os grafos não conexos neste trabalho

foram significativamente menores em número de vértices e arestas do que o grafo do qual elas fazem parte.

As *Tabelas* 4.1 e 4.2 exibem algumas características do conjunto de grafos *conexos* e do conjunto de grafos *não conexos*, respectivamente.

Tabela 4.1 – Grafos *conexos*

Nº	**Nome**	**|V|**	**|A|**	**Orientação**	**Mem. ocupada**
1	cage15	5.154.859	99.199.551	direcionado	9.04G
2	wiki-topcats	1.791.489	28.511.807	direcionado	2.68G
3	cage14	1.505.785	27.130.349	direcionado	2.69G
4	rajat31	4.690.002	20.316.253	direcionado	2.70G
5	fem_hifreq_circuit	491.100	20.239.237	direcionado	2.69G
6	kim2	456.976	11.330.020	direcionado	2.37G
7	atmosmodl	1.489.752	10.319.760	direcionado	2.66G
8	torso1	116.158	8.516.500	direcionado	2.44G
9	cage13	445.315	7.479.343	direcionado	2.46G
10	ohne2	181.343	6.869.939	direcionado	2.47G
11	Chevron4	711.450	6.376.412	direcionado	2.49G
12	marine1	400.320	6.226.538	direcionado	2.51G
13	Hamrle3	1.447.360	5.514.242	direcionado	2.56G
14	Chebyshev4	68.121	5.377.761	direcionado	2.58G
15	largebasis	440.020	5.240.084	direcionado	2.60G
16	GAP-urand	134.217.728	4.294.966.740	não direcionado	160.00G
17	com-Friendster	65.608.366	3.612.134.270	não direcionado	134.00G
18	mycielskian20	786.431	2.710.370.560	não direcionado	83.00G
19	mycielskian19	393.215	903.194.710	não direcionado	33.70G
20	nlpkkt240	27.993.600	760.648.352	não direcionado	32.60G
21	nlpkkt200	16.240.000	440.225.632	não direcionado	19.90G
22	mycielskian18	196.607	300.933.832	não direcionado	11.20G
23	com-Orkut	3.072.441	234.370.166	não direcionado	8.94G
24	nlpkkt160	8.345.600	225.422.112	não direcionado	9.14G
25	Flan_1565	1.564.794	114.165.372	não direcionado	5.34G
26	europe_osm	50.912.018	108.109.320	não direcionado	6.47G
27	delaunay_n24	16.777.216	100.663.202	não direcionado	6.18G
28	mycielskian17	98.303	100.245.742	não direcionado	3.83G
29	nlpkkt120	3.542.400	95.117.792	não direcionado	3.86G
30	dielFilterV3real	1.102.824	89.306.020	não direcionado	2.95G
31	channel-500x100x100-b050	4.802.000	85.362.744	não direcionado	3.25G
32	audikw_1	943.695	77.651.847	não direcionado	3.55G

Fonte: Do autor(2021)

Tabela 4.2 – Grafos *não conexos*

Nº	Nome	|V|	|A|	Orientação	Mem. ocupada
1	sk-2005	50.636.154	1.949.412.601	direcionado	87.30G
2	GAP-web	50.636.151	1.930.292.948	direcionado	85.00G
3	twitter7	41.652.230	1.468.365.182	direcionado	62.40G
4	GAP-twitter	61.578.415	1.468.364.884	direcionado	60.00G
5	it-2004	41.291.594	1.150.725.436	direcionado	52.60G
6	webbase-2001	118.142.155	1.019.903.190	direcionado	56.00G
7	uk-2005	39.459.925	936.364.282	direcionado	51.00G
8	arabic-2005	22.744.080	639.999.458	direcionado	33.20G
9	stokes	11.449.533	349.921.980	direcionado	30.20G
10	uk-2002	18.520.486	298.113.762	direcionado	22.50G
11	HV15R	2.017.169	283.073.458	direcionado	21.70G
12	indochina-2004	7.414.866	194.109.311	direcionado	14.50G
13	vas_stokes_4M	4.382.246	131.577.616	direcionado	11.10G
14	ML_Geer	1.504.002	110.686.677	direcionado	9.90G
15	ljournal-2008	5.363.260	79.023.142	direcionado	6.89G
16	GAP-kron	134.217.726	4.223.264.644	não direcionado	146.60G
17	mawi_201512020330	226.196.185	480.047.894	não direcionado	18.20G
18	kmer_V1r	214.005.017	465.410.904	não direcionado	17.60G
19	kmer_A2a	170.728.175	360.585.172	não direcionado	13.70G
20	Queen_4147	4.147.110	316.548.962	não direcionado	14.80G
21	kmer_P1a	139.353.211	297.829.984	não direcionado	11.40G
22	mawi_201512020130	128.568.730	270.234.840	não direcionado	10.40G
23	rgg_n_2_24_s0	16.777.216	265.114.400	não direcionado	9.90G
24	kron_g500-logn21	2.097.152	182.082.942	não direcionado	6.95G
25	mawi_201512020030	68.863.315	143.414.960	não direcionado	5.63G
26	kmer_U1a	67.716.231	138.778.562	não direcionado	5.46G
27	Bump_2911	2.911.419	127.729.899	não direcionado	5.96G
28	rgg_n_2_23_s0	8.388.608	127.002.786	não direcionado	4.85G
29	Cube_Coup_dt6	2.164.760	124.406.070	não direcionado	5.79G
30	Cube_Coup_dt0	2.164.760	124.406.070	não direcionado	5.79G
31	kmer_V2a	55.042.369	117.217.600	não direcionado	4.65G

Fonte: Do autor(2021)

4.2.2 Máquinas de teste

Os testes de *speedup* foram realizados em máquinas de alto desempenho cujas características estão detalhadas a seguir.

Máquina SKL

Máquina com 2 soquetes, cada um contendo um processador Intel Xeon Gold 5120 de 2.2GHz, com 14 núcleos físicos e 14 núcleos lógicos, cache L1 de 896KiB, cache L2 de 14MiB

e cache L3 de 19MiB; memória RAM total de 192GiB(6x32GiB DIMM DDR4 2400MHz). O Sistema Operacional da máquina é o Ubuntu 19.10 e o compilador utilizado foi o gcc 9.2.1. [1]

Máquina CLX

Um dos nós computacionais de expansão(Bull Sequana X1120) do supercomputador Santos Dumont(SDumont), contendo 2 soquetes, cada um com um processador Intel Xeon Cascade Lake Gold 6252 de 2.10GHz, com 24 núcleos físicos e 24 núcleos lógicos, cache L1 de 750KiB, cache L2 de 23MiB e cache L3 de 34MiB; memória RAM total de 357GiB. O Sistema Operacional da máquina é o RedHat Linux 7.6 e o compilador utilizado foi o gcc 7.4.0. [1]

4.2.3 Descrição dos testes

Os testes foram realizados com objetivo de obter-se os tempos médios de execução das versões do algoritmo *PBFS* que foram implementadas e seus *speedups* para diferentes condições de paralelismo definidas pelo número de processos *MPI* e/ou número de *threads OpenMP* utilizadas em cada execução.

Testes com o algoritmo $PBFS_O$

Estes testes foram realizados com todos os grafos nas 2 máquinas e ocorreu da seguinte forma: cada arquivo de texto que contém os dados de um grafo foi carregado para a memória principal e foi construída uma estrutura de dados *Grafo*, utilizando-se uma lista de adejacências, para representar o respectivo grafo. Para cada um dos grafos, o primeiro teste constituiu-se em executar o algoritmo *BFS* e computar-se o tempo de execução, na sequência foram executados n testes, onde cada um deles foi constituído de uma execução do $PBFS_O$ com i *threads OpenMP*, $i \in \{1,...,n\}$. Para a máquina *SKL*, utilizou-se $n = 56$ e, para a máquina *CLX*, $n = 48$. Em cada um dos testes do $PBFS_O$, calculou-se o *speedup* obtido por esse algoritmo da seguinte forma:

$$speedup_i = \frac{timePBFS_{O_1}}{timePBFS_{O_i}}$$

[1] Em todas as máquinas, em todos os testes, a política de vinculação de *threads* utilizada foi a *default/-none*, que siginifica que, por padrão, as *threads* não são vinculadas a nenhum contexto em particular; no entanto, se o sistema operacional suportar uma política de afinidade de *threads*, o compilador ainda utilizará a interface de afinidade de *threads* do *OpenMP* para determinar a topologia da máquina.

Onde $speedup_i$ representa o *speedup* alcançado pelo $PBFS_O$ no teste i (com i *threads*), $timePBFS_{O_1}$ é o tempo médio de execução do $PBFS_O$ no teste 1 (com 1 *thread*), e $timePBFS_{O_i}$ é o tempo médio de execução do $PBFS_O$ no teste i. Os tempos médios refereridos anteriormente foram calculados segundo a média geométrica de 10 execuções do respectivo algoritmo com os parâmetros conforme especificados. [2]

Testes com o algoritmo $PBFS_H$

Este teste foi realizado com todos os grafos na máquina *CLX* e com 60 dos 63 grafos de teste na máquina *SKL*, uma vez que os grafos *com-Friendster, GAP-kron e GAP-urnad* demandam mais memória principal no teste com o $PBFS_H$ do que o que há disponível nesta máquina. O teste procedeu-se como se segue: optou-se por uma paralelização inter-soquete por *MPI* e uma paralelização intra-soquete por *OpenMP*. Foram realizados n testes, onde, no primeiro deles, o algoritmo $PBFS_H$ foi executado com apenas um processo *MPI* e uma *thread OpenMP* e computou-se o tempo médio de execução; para o $i-$ésimo dos $n-1$ demais testes, utilizou-se 2 processos *MPI*, sendo cada um destes vinculado a cada soquete disponível, cada processo *MPI* foi responsável por disparar $i-1$ *threads OpenMP*, $i \in \{2,3,...,n\}$; para cada teste foi calculado o tempo médio de execução. Para a máquina *SKL*, $n = 29$, para a máquina *CLX*, $n = 25$. Os tempos médios de execução foram obtidos através da média geométrica dos tempos de 10 execuções suscessivas do algoritmo $PBFS_H$ com os parâmetros especificados anteriormente em cada teste. Os *speedups* foram calculados da seguinte forma:

$$speedup_i = \frac{timePBFS_{H_1}}{timePBFS_{H_i}}$$

Onde $speedup_i$ representa o *speedup* obtido no teste i, $timePBFS_{H_1}$ é tempo médio de execução obtido no teste 1 e $timePBFS_{H_i}$ é o tempo de execução médio obtido no teste i. [2]

[2] Logo antes de cada um dos testes com cada um dos grafos com os algoritmos $PBFS_O$ e $PBFS_H$, foi realizado também um teste desse grafo com o algoritmo BFS para que fosse possível se analisar a performance das versões paralelas da busca em largura implementadas em relação à versão sequencial com fila desse algoritmo.

5 RESULTADOS E ANÁLISE

Neste capítulo são descritos todos os resultados dos testes feitos para se obter os *speedups* dos algoritmos $PBFS_O$ e $PBFS_H$ nas diferentes máquinas de teste de alta performance utilizadas, conforme as especificações e a metodologia descritas no *Capítulo 4*. Também mostra-se os tempos de execução obtidos pelo algoritmo BFS no teste de cada grafo, em cada máquina de teste.

Em todos os gráficos apresentados neste capítulo, o eixo *x* representa o número de *threads* utilizado em cada rodada de testes e o eixo *y* representa o *speedup* médio obtido.

5.1 Testes na máquina *SKL*

Os resultados dos testes na máquina *SKL* estão descritos a seguir. A *Seção 5.1.1* apresenta os resultados dos testes dos grafos *conexos* e a *Seção 5.1.2* apresenta os resultados dos testes para grafos *não conexos* nessa máquina.

5.1.1 Grafos *conexos*

A *Tabela 5.1* apresenta os tempos de execução sequenciais obtidos pelos algoritmos BFS, $PBFS_O$ e $PBFS_H$ para os grafos *conexos*, na máquina *SKL*. A coluna v_0 dessa tabela representa o vértice inicial utilizado pelos algoritmos testados. Observa-se que, em geral, os algoritmos paralelos executados sequencialmente foram mais rápidos que o algoritmo BFS. O $PBFS_O$ foi mais lento que o BFS em apenas 4 dos 32 grafos testados e a média geométrica dos tempos de execução sequenciais nesses grafos foi 3,34% menor (mais rápida) no $PBFS_O$. O algoritmo $PBFS_H$ foi mais lento que o BFS em apenas 1 dos 30 grafos testados e a média geométrica dos tempos de execução sequenciais nesses grafos foi 4,15% menor (mais rápida) no $PBFS_H$.

Tabela 5.1 – Tempos de execução sequenciais (em segundos) dos algoritmos *BFS*, $PBFS_O$, e $PBFS_H$ nos grafos *conexos*, na máquina *SKL*.

Nº	Grafo	BFS*	$PBFS_O$	BFS**	$PBFS_H$	v_0
1	cage15	2,26	2,26	2,22	2,21	4.850.801
2	wiki-topcats	0,88	0,80	0,85	0,78	1.166.027
3	cage14	0,58	0,57	0,58	0,56	1.457.234
4	rajat31	0,60	0,55	0,54	0,54	1.251
5	fem_hifreq_circuit	0,30	0,28	0,29	0,27	28.187
6	kim2	0,14	0,13	0,14	0,13	1.355
7	atmosmodl	0,33	0,27	0,33	0,27	39.404
8	torso1	0,08	0,08	0,09	0,08	4.161
9	cage13	0,15	0,15	0,15	0,14	411.097
10	ohne2	0,16	0,16	0,15	0,15	59.989
11	Chevron4	0,11	0,09	0,10	0,09	512
12	marine1	0,14	0,12	0,13	0,11	20.908
13	Hamrle3	0,36	0,30	0,34	0,29	736.801
14	Chebyshev4	0,05	0,05	0,05	0,05	1
15	largebasis	0,09	0,16	0,09	0,16	40.003
16	GAP-urand	479,80	465,37	-	-	57.337.431
17	com-Friendster	265,55	249,41	-	-	40.553.569
18	mycielskian20	25,58	25,19	28,79	25,30	786.431
19	mycielskian19	8,30	8,03	8,09	7,97	393.215
20	nlpkkt240	13,23	13,23	13,10	12,73	116.645
21	nlpkkt200	7,22	7,28	7,33	7,12	81.205
22	mycielskian18	2,67	2,66	2,67	2,66	196.607
23	com-Orkut	5,20	4,49	4,39	4,33	43.608
24	nlpkkt160	3,59	3,62	3,53	3,52	52.165
25	Flan_1565	1,17	1,16	1,16	1,14	19
26	europe_osm	13,21	12,33	13,56	12,39	36.470.776
27	delaunay_n24	4,42	3,96	4,56	4,00	3.142.594
28	mycielskian17	0,90	0,91	0,96	0,92	98.303
29	nlpkkt120	1,60	1,59	1,65	1,55	29.525
30	dielFilterV3real	1,14	1,11	1,16	1,11	16.481
31	channel-500x100x100-b050	1,26	1,24	1,26	1,22	2.452
32	audikw_1	0,96	0,96	0,97	0,93	3.178

* Obtidos imediatamente antes de cada teste com o algoritmo $PBFS_O$.
** Obtidos imediatamente antes de cada teste com o algoritmo $PBFS_H$.

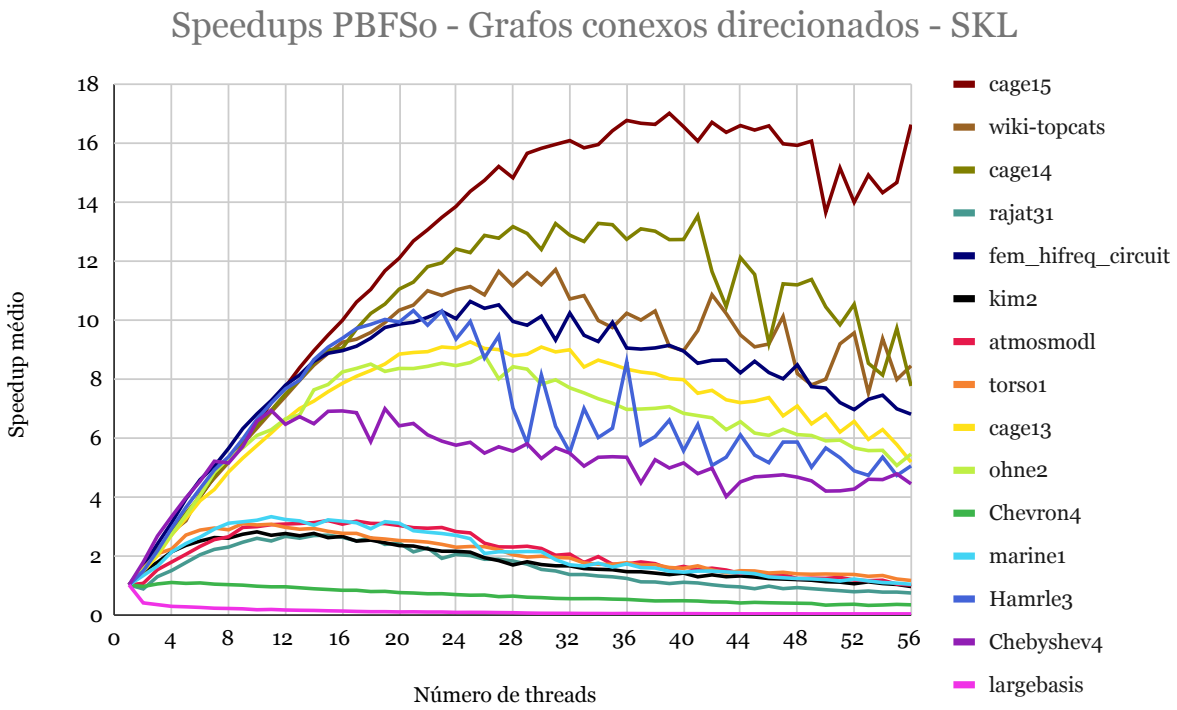
Fonte: Do autor(2021)

5.1.1.1 *Speedups* do algoritmo $PBFS_O$

A *Figura 5.1* e a *Tabela 5.2* apresentam os *speedups* alcançados pelo algoritmo $PBFS_O$ nos grafos *conexos* direcionados, na máquina *SKL*. Houve *speedups* significativos na maioria dos grafos desse teste, onde apenas 2 dos 15 grafos testados apresentaram *slowdown* predominante para os diferentes números de *threads* usados. Os maiores *speedups* foram acançados pelos grafos *cage14*, *cage15* e *wiki-topcats*. Os piores *speedups* foram obtidos nos grafos *Che-*

vron4, *largebasis* e *rajat31*. Em geral o perfil das curvas de *speedup* foi de decrescimento para um número de *threads* maior que 28, o que indica que a utilização do *hyperthreading* impactou negativamente o desempenho do algoritmo $PBFS_O$ neste teste.

Figura 5.1 – Gráfico com os *speedups* obtidos pelo algoritmo $PBFS_O$ nos grafos *conexos* direcionados, na máquina *SKL*.



Fonte: Do autor(2021)

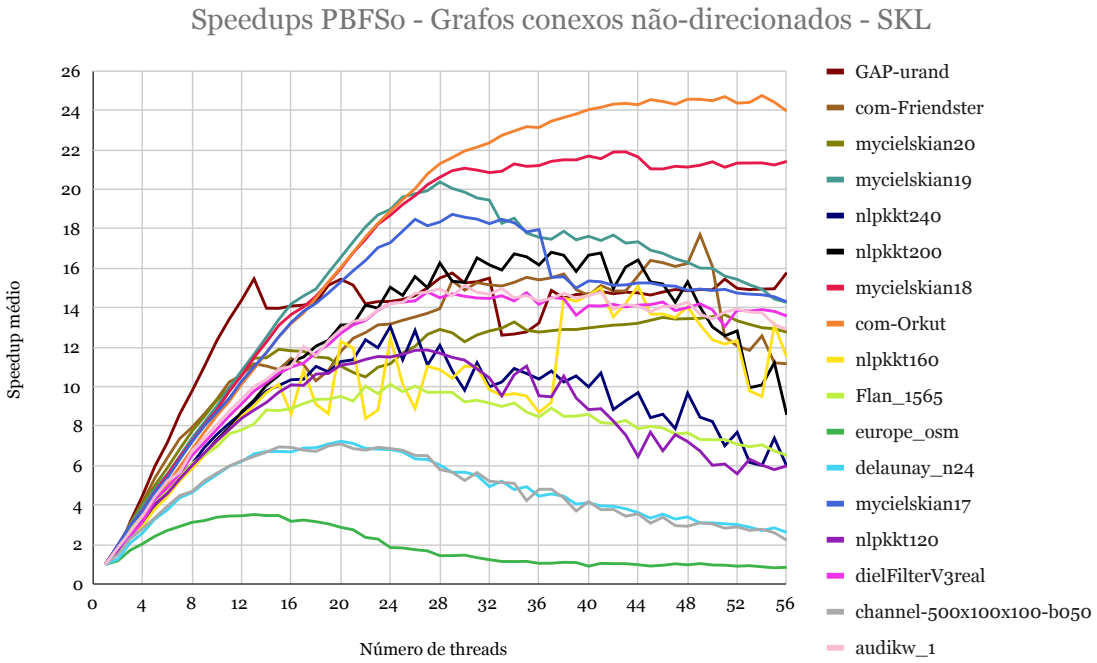
Tabela 5.2 – *Speedups* obtidos pelo algoritmo $PBFS_O$ nos grafos *conexos* direcionados, na máquina *SKL*

Nº de threads	cage15	wiki-topcats	cage14	rajat31	fem_hifreq_circuit	kim2	atmosmodl	torso1	cage13	ohne2	Chevron4	marine1	Hamrle3	Chebyshev4	largebasis
2	1,4	1,4	1,4	0,9	1,7	1,5	1,1	1,6	1,5	1,5	0,9	1,3	1,5	1,8	0,4
4	2,7	2,8	2,8	1,5	3,1	2,1	1,8	2,2	2,7	2,7	1,1	2,1	2,9	3,3	0,3
6	4,0	4,1	4,0	2,0	4,6	2,5	2,3	2,9	3,9	4,0	1,1	2,6	4,3	4,5	0,2
8	5,3	5,2	5,2	2,3	5,6	2,6	2,6	2,9	4,8	5,3	1,0	3,1	5,4	5,1	0,2
10	6,5	6,3	6,4	2,6	6,8	2,8	3,0	3,0	5,7	6,1	1,0	3,2	6,6	6,6	0,2
12	7,7	7,4	7,5	2,7	7,8	2,7	3,1	3,0	6,6	6,6	0,9	3,2	7,7	6,5	0,1
14	9,0	8,5	8,5	2,7	8,6	2,7	3,1	2,9	7,2	7,6	0,9	3,0	8,6	6,5	0,1
16	10,0	9,2	9,0	2,6	9,0	2,6	3,1	2,8	7,8	8,2	0,8	3,2	9,4	6,9	0,1
18	11,0	9,6	10,2	2,6	9,4	2,5	3,1	2,6	8,3	8,5	0,8	2,9	9,8	5,9	0,1
20	12,1	10,3	11,0	2,4	9,9	2,3	3,0	2,5	8,8	8,3	0,7	3,1	9,9	6,4	0,1
22	13,1	11,0	11,8	2,3	10,1	2,2	2,9	2,5	8,9	8,4	0,7	2,8	9,8	6,1	0,1
24	13,9	11,0	12,4	2,0	10,0	2,1	2,8	2,3	9,0	8,4	0,7	2,7	9,3	5,7	0,1
26	14,7	10,8	12,9	1,9	10,4	1,9	2,4	2,3	9,0	8,8	0,7	2,1	8,7	5,5	0,1
28	14,8	11,2	13,2	1,8	9,9	1,7	2,3	2,0	8,8	8,4	0,6	2,1	7,0	5,5	0,1
30	15,8	11,2	12,4	1,5	10,1	1,7	2,2	2,0	9,1	7,8	0,6	2,1	8,1	5,3	0,0
32	16,1	10,7	12,9	1,4	10,2	1,6	2,0	1,9	9,0	7,7	0,5	1,7	5,5	5,5	0,0
34	16,0	10,0	13,3	1,3	9,3	1,5	2,0	1,7	8,6	7,3	0,5	1,7	6,0	5,3	0,0
36	16,8	10,2	12,7	1,2	9,0	1,4	1,7	1,8	8,3	7,0	0,5	1,7	8,6	5,3	0,0
38	16,6	10,3	13,0	1,1	9,0	1,4	1,7	1,7	8,2	7,0	0,5	1,6	6,0	5,3	0,0
40	16,6	9,0	12,7	1,1	8,9	1,4	1,6	1,6	8,0	6,8	0,5	1,4	5,6	5,1	0,0
42	16,7	10,9	11,6	1,0	8,6	1,3	1,6	1,5	7,6	6,7	0,4	1,5	5,1	5,0	0,0
44	16,6	9,5	12,1	0,9	8,2	1,3	1,4	1,5	7,2	6,5	0,4	1,4	6,1	4,5	0,0
46	16,6	9,2	9,2	1,0	8,2	1,2	1,3	1,4	7,4	6,1	0,4	1,3	5,1	4,7	0,0
48	15,9	8,2	11,2	0,9	8,5	1,2	1,2	1,4	7,1	6,1	0,4	1,2	5,8	4,7	0,0
50	13,7	8,0	10,4	0,8	7,7	1,1	1,2	1,4	6,8	5,9	0,3	1,2	5,6	4,2	0,0
52	14,0	9,6	10,5	0,8	7,0	1,0	1,2	1,4	6,6	5,7	0,3	1,2	4,9	4,3	0,0
54	14,3	9,4	8,1	0,8	7,4	1,0	1,2	1,3	6,3	5,6	0,3	1,1	5,3	4,6	0,0
56	16,6	8,4	7,8	0,7	6,8	0,9	1,0	1,1	5,2	5,4	0,3	1,0	5,0	4,4	0,0

Fonte: Do autor(2021)

A *Figura 5.2* e a *Tabela 5.3* apresentam os *speedups* médios alcançados pelo algoritmo $PBFS_O$ nos grafos *conexos* não direcionados, na máquina *SKL*. Todas os grafos apresentaram *speedup* significativamente acima de 1 para a maioria dos números de *threads* utilizados. O único grafo onde obteve-se *slowdown* foi o *europ-osm*, mas isso ocorreu em apenas alguns poucos casos para números de *threads* acima de 50. Os maiores *speedups* máximos foram obtidos pelos grafos *com-Orkut*, *mycielskian18* e *mycielskian19* e os grafos *channel-500x100x100-b050*, *delaunay_n24* e *europe_osm* alcançaram os menores *speedups* máximos deste teste.

Figura 5.2 – Gráfico com os *speedups* obtidos pelo algoritmo $PBFS_O$ nos grafos *conexos* não direcionados, na máquina *SKL*.



Fonte: Do autor(2021)

Os grafos *com-Orkut* e *mycielskian18* apresentaram algum crescimento/estabilidade no perfil de suas curvas de *speedup* para números de *threads* acima de 28, mas, no geral, o que se observa na *Figura 5.2* é que o perfil dessas curvas foi de decrescimento para números de *threads* acima desse valor, o que indica que, de forma geral, a utilização do *hyperthreading* não produziu melhores desempenhos neste teste.

Tabela 5.3 – *Speedups* obtidos pelo algoritmo $PBFS_O$ nos grafos *conexos* não direcionados, na máquina *SKL*

Nº de threads	GAP-urand	com-Friendster	mycielskian20	mycielskian19	nlpkkt240	nlpkkt200	mycielskian18	com-Orkut	nlpkkt160	Flan_1565	europe_osm	delaunay_n24	mycielskian17	nlpkkt120	dielFilterV3real	channel-500x100x100-b050	audikw_1
2	1,6	1,7	1,9	2,0	1,6	1,6	2,0	1,8	1,6	1,8	1,2	1,3	2,0	1,7	1,8	1,5	1,8
4	4,5	4,2	4,1	3,9	3,2	3,0	3,7	3,3	2,9	3,6	2,0	2,6	3,7	3,2	3,1	2,8	3,5
6	7,2	6,4	5,9	5,6	4,7	4,6	5,5	5,1	4,4	4,7	2,7	3,8	5,4	4,6	4,9	4,0	5,2
8	9,8	7,9	7,8	7,4	6,1	6,0	7,3	6,8	5,9	6,1	3,1	4,6	7,2	6,0	6,5	4,7	6,8
10	12,3	9,4	9,3	9,0	7,6	7,3	8,8	8,5	7,2	7,0	3,4	5,5	8,7	7,3	7,8	5,6	8,1
12	14,4	10,6	10,7	10,8	8,7	8,7	10,6	10,1	8,5	7,8	3,5	6,2	10,3	8,4	9,1	6,3	9,4
14	14,0	11,0	11,5	12,5	9,8	10,1	12,3	11,7	9,7	8,8	3,5	6,7	11,7	9,2	10,2	6,7	10,4
16	14,1	11,4	11,8	14,2	10,4	11,3	13,7	13,2	8,6	8,9	3,2	6,7	13,3	10,1	11,0	6,9	11,0
18	14,3	10,3	11,5	15,0	11,0	12,1	14,6	14,5	9,1	9,4	3,2	6,9	14,2	10,6	11,7	6,7	11,6
20	15,4	11,8	11,0	16,5	11,3	13,1	16,0	16,0	12,3	9,5	2,9	7,2	15,3	11,1	12,7	7,1	12,9
22	14,2	12,7	10,5	18,1	12,4	14,1	17,5	17,6	8,4	10,0	2,4	6,9	16,4	11,4	13,4	6,8	13,4
24	14,3	13,2	11,2	19,0	13,0	15,0	18,7	18,9	12,5	10,1	1,9	6,8	17,3	11,5	14,2	6,9	14,2
26	14,6	13,5	12,0	19,8	12,8	15,6	19,7	20,0	8,9	10,0	1,8	6,3	18,5	11,8	14,3	6,5	14,7
28	15,5	14,0	12,9	20,4	12,1	16,3	20,6	21,3	10,9	9,7	1,4	6,0	18,3	11,7	14,5	5,8	15,0
30	15,3	14,9	12,3	19,9	9,8	15,3	21,1	21,9	11,0	9,2	1,5	5,7	18,6	11,3	14,6	5,3	15,1
32	15,5	15,2	12,8	19,4	10,0	16,2	20,8	22,4	9,8	9,2	1,2	4,9	18,3	10,4	14,5	5,2	14,7
34	12,7	15,3	13,3	18,5	10,9	16,7	21,3	23,0	9,7	9,2	1,1	4,8	18,3	10,6	14,3	5,1	14,5
36	13,2	15,4	12,8	17,6	10,4	16,2	21,2	23,1	8,7	8,5	1,1	4,5	18,0	9,5	14,2	4,8	14,3
38	14,5	15,7	12,9	17,9	10,2	16,7	21,5	23,6	14,5	8,5	1,1	4,5	15,6	10,5	14,5	4,4	14,7
40	14,7	14,6	13,0	17,6	10,0	16,7	21,7	24,0	14,6	8,6	0,9	4,1	15,4	8,8	14,1	4,2	14,6
42	14,7	14,9	13,1	17,7	8,8	15,0	21,9	24,3	13,5	8,1	1,0	3,9	15,1	8,3	14,2	3,8	13,9
44	14,8	15,6	13,2	17,3	9,7	16,4	21,6	24,3	15,1	7,9	1,0	3,6	15,3	6,5	14,1	3,6	14,1
46	14,8	16,3	13,5	16,8	8,6	15,2	21,0	24,4	13,7	7,9	1,0	3,6	15,1	6,8	14,3	3,4	14,1
48	14,9	16,2	13,4	16,3	9,7	15,3	21,1	24,6	14,1	7,7	1,0	3,4	14,9	7,2	14,0	2,9	14,3
50	14,9	16,1	13,5	16,0	8,2	13,0	21,4	24,5	12,4	7,3	1,0	3,1	14,9	6,0	13,9	3,1	13,4
52	15,0	12,1	13,3	15,4	7,7	12,8	21,3	24,4	12,3	7,1	0,9	3,0	14,7	5,6	13,9	2,9	13,9
54	14,9	12,6	13,0	14,9	6,0	10,1	21,3	24,7	9,5	7,1	0,9	2,7	14,7	6,0	13,9	2,8	13,7
56	15,8	11,2	12,8	14,3	6,0	8,6	21,4	24,0	11,5	6,5	0,9	2,6	14,3	6,0	13,6	2,2	12,9

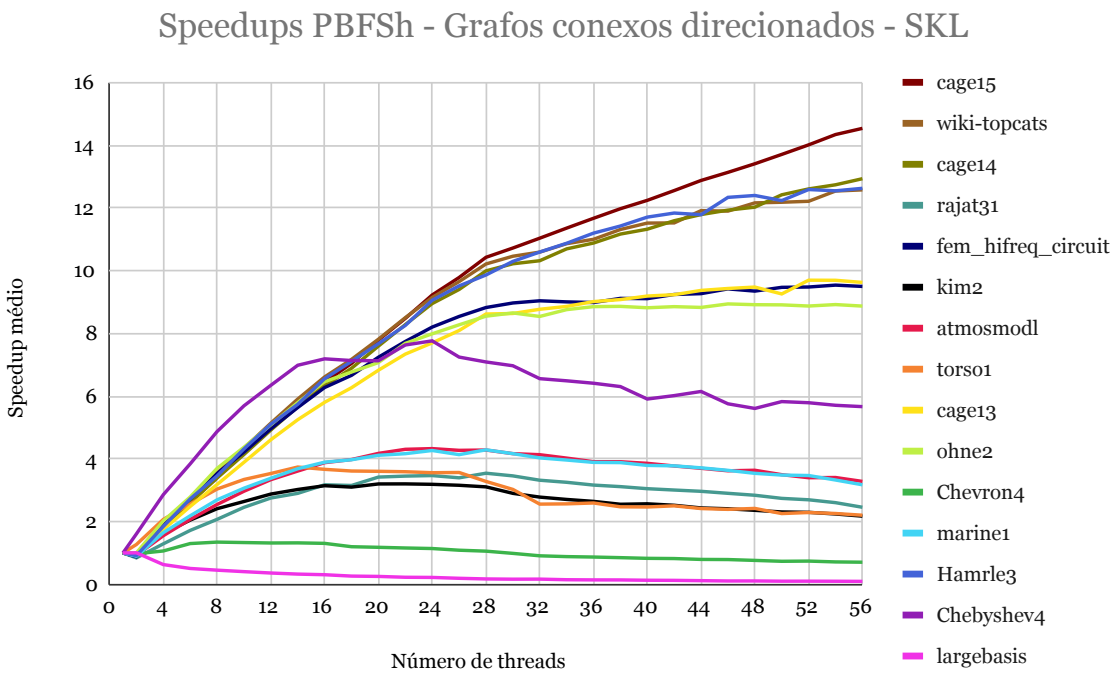
Fonte: Do autor(2021)

5.1.1.2 *Speedups* do algoritmo $PBFS_H$

A *Tabela 5.4* e a *Figura 5.3* exibem os resultados obtidos, na máquina *SKL*, pelo algoritmo $PBFS_H$ nos grafos *conexos* direcionados. 14 dos 15 grafos testados apresentaram *speedup* para um número de *threads* até 28 e o grafo *largebasis* apresentou predominantemente *slow-*

down para a maioria dos números de *threads* utilizados. Os melhores *speedups* máximos foram obtidos pelos grafos *cage14*, *cage15* e *wiki-topcats* e, os piores, pelos grafos *Chevron4*, *kim2* e *largebasis*.

Figura 5.3 – Gráfico com os *speedups* obtidos pelo algoritmo $PBFS_H$ nos grafos conexos direcionados, na máquina *SKL*.



Fonte: Do autor(2021)

O perfil das curvas de *speedup* visto na *Figura 5.3* foi de crescimento para 7 dos 15 grafos testados e de leve decrescimento para os demais. Os grafos *cage14*, *cage15*, *Hamrle3* e *wiki-topcats* tiveram aumento significativo no *speedup* para um número de *threads* acima de 28, enquanto os grafos *cage13*, *fem_hifreq_circuit* e *ohne2* apresentaram leve crescimento para essas quantidades de *threads*; assim, a utilização do *hyperthreading* produziu um melhor desempenho em 46,67% dos grafos deste teste.

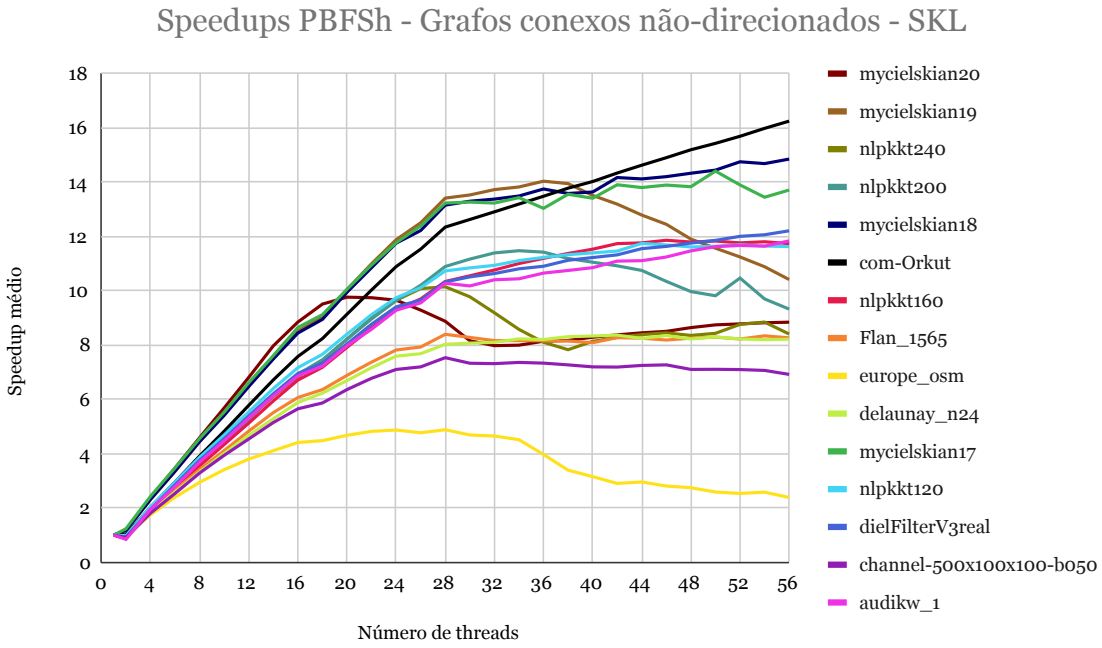
Tabela 5.4 – *Speedups* do algoritmo $PBFS_H$ nos grafos *conexos* direcionados, na máquina *SKL*

Nº de threads	cage15	wiki-topcats	cage14	rajat31	fem_hifreq_circuit	kim2	atmosmodl	torso1	cage13	ohne2	Chevron4	marine1	Hamrle3	Chebyshev4	largebasis
2	0,9	0,9	0,9	0,8	0,9	0,9	0,9	1,3	0,9	1,0	1,0	0,9	0,9	1,6	1,0
4	1,8	1,9	1,8	1,3	1,9	1,6	1,5	2,1	1,7	2,0	1,1	1,6	1,8	2,9	0,6
6	2,6	2,7	2,6	1,7	2,7	2,0	2,1	2,6	2,5	2,8	1,3	2,2	2,7	3,8	0,5
8	3,4	3,5	3,4	2,1	3,5	2,4	2,5	3,0	3,2	3,7	1,3	2,7	3,5	4,9	0,5
10	4,2	4,3	4,1	2,5	4,2	2,6	3,0	3,3	3,9	4,4	1,3	3,1	4,3	5,7	0,4
12	4,9	5,1	4,9	2,8	5,0	2,9	3,3	3,5	4,6	5,1	1,3	3,4	5,1	6,3	0,4
14	5,7	5,9	5,7	2,9	5,6	3,0	3,6	3,7	5,2	5,8	1,3	3,7	5,7	7,0	0,3
16	6,4	6,6	6,3	3,2	6,3	3,1	3,9	3,7	5,8	6,5	1,3	3,9	6,6	7,2	0,3
18	7,0	7,2	6,9	3,2	6,7	3,1	4,0	3,6	6,3	6,8	1,2	4,0	7,1	7,1	0,3
20	7,8	7,8	7,6	3,4	7,2	3,2	4,2	3,6	6,8	7,1	1,2	4,1	7,7	7,1	0,3
22	8,5	8,5	8,3	3,4	7,7	3,2	4,3	3,6	7,3	7,7	1,2	4,2	8,3	7,6	0,2
24	9,2	9,1	8,9	3,5	8,2	3,2	4,3	3,6	7,7	8,0	1,1	4,3	9,1	7,8	0,2
26	9,8	9,6	9,4	3,4	8,5	3,2	4,3	3,6	8,1	8,3	1,1	4,1	9,5	7,2	0,2
28	10,4	10,2	10,0	3,5	8,8	3,1	4,3	3,3	8,6	8,5	1,1	4,3	9,9	7,1	0,2
30	10,7	10,5	10,2	3,5	9,0	2,9	4,2	3,0	8,6	8,7	1,0	4,2	10,3	7,0	0,2
32	11,0	10,6	10,3	3,3	9,0	2,8	4,1	2,6	8,8	8,5	0,9	4,0	10,6	6,6	0,2
34	11,4	10,9	10,7	3,3	9,0	2,7	4,0	2,6	8,9	8,8	0,9	4,0	10,9	6,5	0,1
36	11,7	11,0	10,9	3,2	9,0	2,6	3,9	2,6	9,0	8,9	0,9	3,9	11,2	6,4	0,1
38	12,0	11,3	11,2	3,1	9,1	2,6	3,9	2,5	9,1	8,9	0,8	3,9	11,4	6,3	0,1
40	12,2	11,5	11,3	3,0	9,1	2,6	3,9	2,5	9,2	8,8	0,8	3,8	11,7	5,9	0,1
42	12,6	11,5	11,6	3,0	9,2	2,5	3,8	2,5	9,2	8,9	0,8	3,8	11,8	6,0	0,1
44	12,9	11,9	11,8	3,0	9,3	2,4	3,7	2,4	9,4	8,8	0,8	3,7	11,8	6,1	0,1
46	13,1	11,9	11,9	2,9	9,4	2,4	3,6	2,4	9,4	8,9	0,8	3,6	12,3	5,8	0,1
48	13,4	12,2	12,0	2,8	9,3	2,4	3,6	2,4	9,5	8,9	0,8	3,5	12,4	5,6	0,1
50	13,7	12,2	12,4	2,7	9,5	2,3	3,5	2,3	9,3	8,9	0,7	3,5	12,2	5,8	0,1
52	14,0	12,2	12,6	2,7	9,5	2,3	3,4	2,3	9,7	8,9	0,7	3,5	12,6	5,8	0,1
54	14,3	12,5	12,7	2,6	9,5	2,2	3,4	2,3	9,7	8,9	0,7	3,3	12,5	5,7	0,1
56	14,5	12,6	12,9	2,5	9,5	2,2	3,3	2,2	9,6	8,9	0,7	3,2	12,6	5,7	0,1

Fonte: Do autor(2021)

A *Figura 5.4* e a *Tabela 5.5* apresentam os *speedups* médios obtidos pelo algoritmo $PBFS_H$ nos grafos *conexos* não direcionados, na máquina *SKL*. Todos os grafos deste teste apresentaram *speedups* significativos. Os maiores *speedups* máximos foram obtidos pelos grafos *com-Orkut*, *mycielskian17* e *mycielskian18* e, os menores, pelos grafos *channel-500x100x100-b050*, *delaunay_n24* e *europe_osm*. O perfil das curvas de *speedup* foi, em geral, de crescimento para quantidades de *threads* acima de 28, o que indica que a utilização do *hyperthreading* impactou positivamente no desempenho do $PBFS_H$ neste teste.

Figura 5.4 – Gráfico com os *speedups* obtidos pelo $PBFS_H$ nos grafos conexos não direcionados, na máquina *SKL*



Fonte: Do autor(2021)

Tabela 5.5 – *Speedups* do algoritmo $PBFS_H$ nos grafos *conexos* não direcionados, na máquina *SKL*

Nº de threads	mycielskian20	mycielskian19	nlpkkt240	nlpkkt200	mycielskian18	com-Orkut	nlpkkt160	Flan_1565	europe_osm	delaunay_n24	mycielskian17	nlpkkt120	dielFilterV3real	channel-500x100x100-b050	audikw_1
2	1,2	1,2	0,9	1,0	1,2	1,0	0,9	0,9	1,0	0,9	1,2	1,0	0,9	0,9	0,8
4	2,4	2,3	1,9	1,9	2,3	2,0	1,9	1,9	1,7	1,9	2,4	2,0	1,9	1,8	2,0
6	3,5	3,4	2,7	2,7	3,4	3,0	2,7	2,7	2,4	2,6	3,5	3,0	2,9	2,5	2,8
8	4,6	4,4	3,6	3,6	4,4	3,9	3,6	3,4	2,9	3,3	4,6	3,9	3,8	3,3	3,7
10	5,7	5,4	4,4	4,4	5,4	4,8	4,3	4,1	3,4	4,0	5,5	4,7	4,6	3,9	4,5
12	6,8	6,5	5,3	5,3	6,5	5,8	5,1	4,8	3,8	4,7	6,6	5,5	5,4	4,5	5,3
14	8,0	7,5	6,1	6,2	7,5	6,7	5,9	5,5	4,1	5,3	7,6	6,4	6,2	5,1	6,1
16	8,8	8,5	6,8	6,9	8,4	7,6	6,7	6,1	4,4	5,9	8,6	7,2	7,0	5,7	6,9
18	9,5	9,0	7,4	7,5	8,9	8,2	7,2	6,4	4,5	6,2	9,1	7,7	7,3	5,9	7,2
20	9,8	10,0	8,2	8,2	9,9	9,1	7,9	6,9	4,7	6,7	10,1	8,4	8,0	6,4	8,0
22	9,7	11,0	9,0	9,0	10,9	10,0	8,6	7,4	4,8	7,2	11,0	9,1	8,8	6,8	8,6
24	9,7	11,9	9,6	9,6	11,8	10,9	9,3	7,8	4,9	7,6	11,8	9,7	9,4	7,1	9,3
26	9,3	12,5	10,1	10,2	12,2	11,5	9,7	7,9	4,8	7,7	12,4	10,1	9,7	7,2	9,6
28	8,9	13,4	10,1	10,9	13,2	12,3	10,3	8,4	4,9	8,0	13,2	10,7	10,3	7,5	10,3
30	8,2	13,5	9,8	11,2	13,3	12,6	10,6	8,3	4,7	8,1	13,3	10,8	10,5	7,3	10,2
32	8,0	13,7	9,2	11,4	13,4	12,9	10,8	8,2	4,7	8,1	13,2	10,9	10,6	7,3	10,4
34	8,0	13,8	8,6	11,5	13,5	13,2	11,0	8,2	4,5	8,2	13,4	11,1	10,8	7,4	10,4
36	8,1	14,0	8,1	11,4	13,8	13,5	11,2	8,2	4,0	8,2	13,0	11,2	10,9	7,3	10,7
38	8,2	13,9	7,8	11,2	13,6	13,8	11,4	8,1	3,4	8,3	13,6	11,3	11,1	7,3	10,7
40	8,3	13,5	8,1	11,1	13,6	14,0	11,5	8,1	3,2	8,3	13,4	11,4	11,2	7,2	10,9
42	8,4	13,2	8,3	10,9	14,2	14,3	11,7	8,3	2,9	8,4	13,9	11,5	11,3	7,2	11,1
44	8,5	12,8	8,4	10,8	14,1	14,6	11,8	8,3	3,0	8,2	13,8	11,7	11,6	7,3	11,1
46	8,5	12,4	8,5	10,3	14,2	14,9	11,9	8,2	2,8	8,4	13,9	11,7	11,6	7,3	11,2
48	8,6	11,9	8,4	10,0	14,3	15,2	11,8	8,3	2,7	8,2	13,8	11,6	11,8	7,1	11,5
50	8,7	11,6	8,4	9,8	14,4	15,4	11,8	8,3	2,6	8,3	14,4	11,6	11,9	7,1	11,6
52	8,8	11,2	8,8	10,5	14,8	15,7	11,8	8,2	2,5	8,2	13,9	11,7	12,0	7,1	11,7
54	8,8	10,9	8,8	9,7	14,7	16,0	11,8	8,3	2,6	8,2	13,4	11,6	12,1	7,1	11,6
56	8,8	10,4	8,4	9,3	14,9	16,3	11,7	8,3	2,4	8,2	13,7	11,6	12,2	6,9	11,8

Fonte: Do autor(2021)

5.1.2 Grafos *não conexos*

Esta seção apresenta os resultados dos testes dos algoritmos $PBFS_O$ (*Seção 5.1.2.1*) e $PBFS_H$ (*Seção 5.1.2.2*) nos grafos *não conexos*, na máquina *SKL*. A *Tabela 5.6* exibe os tempos de execução sequenciais dos algoritmos BFS, $PBFS_O$ e $PBFS_H$ nesses testes. Para cada um dos grafos, a *Tabela 5.6* também exibe os detalhes da componente fortemente conectada (*CFC*)

utilizada, onde v_0 representa o vértice inicial da busca em largura utilizado pelos algoritmos e, portanto, esse vértice é também o identificador da *CFC* testada, e $|V|_{CFC}$ e $|A|_{CFC}$ representam o número de vértices e arestas dessa *CFC*, respectivamente.

Tabela 5.6 – Tempos de execução sequenciais (em segundos) dos algoritmos *BFS*, $PBFS_O$, e $PBFS_H$ nos grafos *não conexos*, na máquina *SKL*.

Nº	Grafo	BFS*	$PBFS_O$	BFS**	$PBFS_H$	v_0	$\|V\|_{CFC}$	$\|A\|_{CFC}$
1	sk-2005	0,11	0,13	0,11	0,13	19.879.528	16.016	104.502
2	GAP-web	0,11	0,13	0,11	0,13	19.879.528	16.016	91.686
3	twitter7	75,90	75,32	110,72	107,96	4.597.022	35.016.137	1.415.799.538
4	GAP-twitter	79,57	79,99	109,18	107,85	19.058.682	35.016.137	1.415.799.261
5	it-2004	0,15	0,19	0,16	0,18	1.906.277	9.995	40.365
6	webbase-2001	0,49	0,57	0,46	0,52	86.002.303	3.842	3.841
7	uk-2005	0,08	0,10	0,09	0,10	21.049.533	5.609	21.961
8	arabic-2005	0,05	0,06	0,05	0,06	2.960.954	9.965	19.920
9	stokes	5,95	5,74	6,00	5,73	10.901.764	11.303.355	349.175.802
10	uk-2002	6,27	5,49	6,12	5,37	15.353.975	17.994.090	289.864.964
11	HV15R	3,42	3,37	3,44	3,37	265.192	2.017.169	283.073.458
12	indochina-2004	0,43	0,43	0,43	0,43	6.086.394	6.987	48.228.945
13	vas_stokes_4M	2,27	2,19	2,24	2,15	3.429.948	4.309.660	131.456.523
14	ML_Geer	1,10	1,10	1,10	1,08	43	1.504.002	110.879.972
15	ljournal-2008	2,63	2,36	2,58	2,32	5.178.078	4.815.948	77.879.760
16	GAP-kron	315,48	307,35	-	-	71.328.661	63.032.893	2.111.611.751
17	mawi_201512020330	19,31	12,92	20,87	13,63	104.492.306	213.682.593	231.407.318
18	kmer_V1r	125,99	106,64	125,03	110,69	2	214.004.392	232.704.832
19	kmer_A2a	104,89	99,93	100,13	88,17	21.095.627	170.372.459	179.941.739
20	Queen_4147	3,22	3,30	3,22	3,32	19	4.147.110	166.823.197
21	kmer_P1a	76,03	67,14	78,06	68,08	414.528	138.896.082	148.465.346
22	mawi_201512020130	11,14	7,46	11,59	7,58	58.782.752	121.594.511	130.268.466
23	rgg_n_2_24_s0	9,07	8,53	9,01	8,39	2.421.853	16.777.215	132.557.200
24	kron_g500-logn21	3,18	3,12	3,30	3,26	1.930.587	1.543.901	91.041.917
25	mawi_201512020030	6,08	4,09	6,24	4,10	31.413.874	64.912.184	69.026.990
26	kmer_U1a	34,76	29,38	34,18	29,58	10.669.071	64.678.340	66.393.629
27	Bump_2911	1,68	1,59	1,65	1,58	1.035.677	2.852.430	65.261.670
28	rgg_n_2_23_s0	4,28	3,99	4,20	3,84	1.210.948	8.388.601	63.501.390
29	Cube_Coup_dt6	1,32	1,34	1,30	1,31	181	2.164.760	64.685.452
30	Cube_Coup_dt0	1,32	1,38	1,30	1,31	181	2.164.760	64.685.452
31	kmer_V2a	26,23	23,02	26,20	22,72	4.105.631	53.500.237	57.076.126

* Obtidos imediatamente antes de cada teste com o algoritmo $PBFS_O$.
** Obtidos imediatamente antes de cada teste com o algoritmo $PBFS_H$.

Fonte: Do autor(2021)

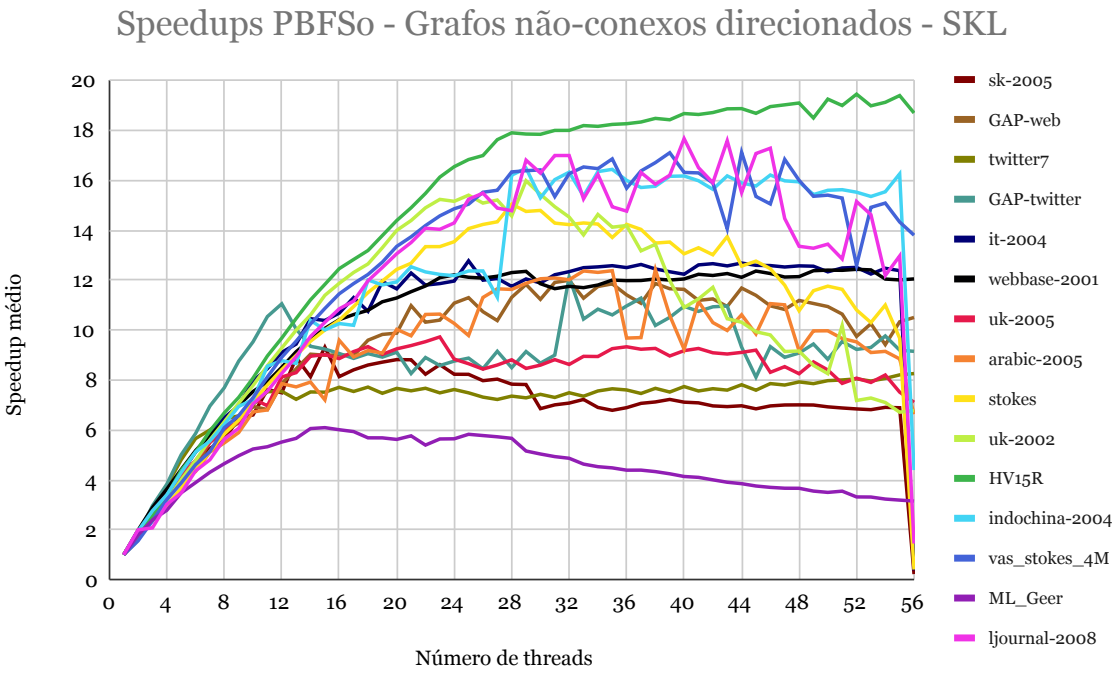
Na *Tabela 5.6*, observa-se que, em geral, os algoritmos paralelos executados sequencialmente foram mais rápidos que o algoritmo *BFS*. O $PBFS_O$ foi mais lento que o *BFS* em 10 dos 31 grafos testados e a média geométrica dos tempos de execução sequenciais nesses grafos foi 3,59% menor (mais rápida) no $PBFS_O$. O algoritmo $PBFS_H$ foi mais lento que o *BFS* em 9

dos 30 grafos testados e a média geométrica dos tempos de execução sequenciais nesses grafos foi 4,73% menor (mais rápida) no $PBFS_H$.

5.1.2.1 *Speedups* do algoritmo $PBFS_O$

Na *Figura 5.5* e na *Tabela 5.7*, tem-se os *speedups* médios obtidos pelo algoritmo $PBFS_O$ nos grafos *não conexos* direcionados, na máquina *SKL*. Todos os grafos deste teste apresentaram *speedups* significativos e, em geral, nehum grafo apresentou *slowdown* para a grande maioria dos números de *threads* utilizados.

Figura 5.5 – Gráfico com os *speedups* do $PBFS_O$ nos grafos *não conexos* direcionados, na máquina *SKL*.



Fonte: Do autor(2021)

Os melhores *speedups* máximos foram alcançados pelos grafos *HV15R*, *ljournal-2008* e *vas_stokes_4M* e, os piores, pelos grafos *ML_Geer*, *sk-2005* e *twitter7*. O perfil das curvas de *speedup* foi, de maneira generalizada, de estabilidade para números de *threads* acima de 28, indicando que a utilização do *hyperthreading* parece não ter afetado de forma significativa o desempenho do algoritmo $PBFS_O$ neste teste.

Tabela 5.7 – *Speedups* obtidos pelo algoritmo $PBFS_O$ nos grafos *não conexos* direcionados, na máquina *SKL*

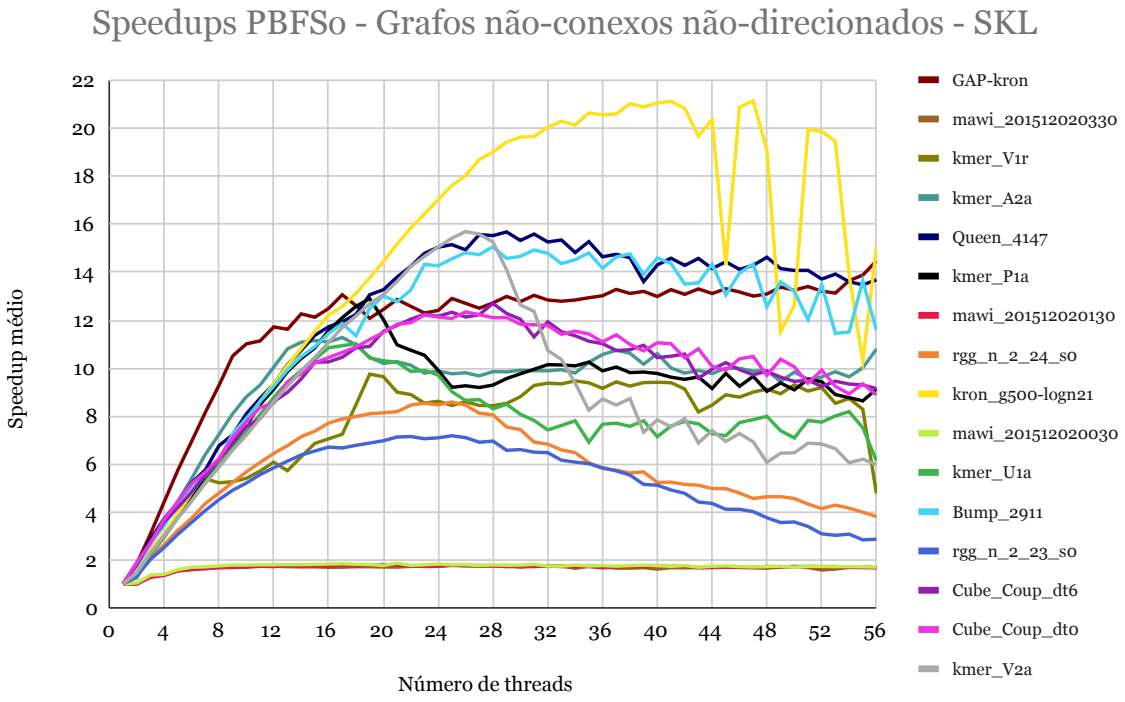
Nº de threads	sk-2005	GAP-web	twitter7	GAP-twitter	it-2004	webbase-2001	uk-2005	arabic-2005	stokes	uk-2002	HV15R	indochina-2004	vas_stokes_4M	ML_Geer	ljournal-2008
2	1,9	1,8	1,6	1,8	2,0	2,0	1,8	1,8	1,9	1,6	1,8	2,0	1,6	1,8	2,0
4	3,4	3,3	3,8	3,8	3,6	3,6	3,4	3,3	3,0	3,2	3,4	3,4	3,2	2,8	3,0
6	4,7	4,5	5,7	5,9	5,1	5,2	4,8	4,3	4,4	4,8	5,1	5,1	4,6	3,9	4,4
8	5,6	5,9	6,4	7,7	6,5	6,3	6,1	5,5	5,8	6,3	6,7	6,2	6,1	4,6	5,6
10	6,6	6,9	7,0	9,5	8,0	7,5	7,3	6,7	7,1	7,8	8,1	7,2	7,3	5,2	7,0
12	7,5	7,9	7,5	11,1	9,1	8,5	8,1	7,9	8,4	9,3	9,7	8,8	8,9	5,5	8,3
14	8,1	9,0	7,5	9,3	10,5	9,6	9,0	7,9	9,6	10,6	11,2	10,4	10,2	6,1	9,7
16	8,1	9,0	7,7	9,1	10,7	10,4	8,9	9,6	10,4	11,9	12,5	10,3	11,5	6,0	10,9
18	8,6	9,6	7,7	9,1	10,8	10,8	9,3	9,2	11,5	12,6	13,2	12,0	12,2	5,7	12,0
20	8,8	9,9	7,7	9,1	11,7	11,3	9,2	10,0	12,4	14,0	14,4	11,9	13,4	5,6	13,1
22	8,2	10,3	7,7	8,9	11,8	11,8	9,5	10,6	13,3	14,9	15,5	12,3	14,2	5,4	14,1
24	8,2	11,1	7,6	8,8	12,0	12,2	8,8	10,3	13,5	15,2	16,5	12,2	14,9	5,6	14,3
26	8,0	10,7	7,3	8,5	12,0	12,1	8,4	11,3	14,2	15,1	17,0	12,4	15,5	5,8	15,5
28	7,8	11,3	7,4	8,5	11,7	12,3	8,8	11,6	15,0	14,6	17,9	16,2	16,3	5,7	14,8
30	6,9	11,2	7,4	8,7	11,9	11,9	8,6	12,1	14,8	15,4	17,8	15,3	16,4	5,0	16,3
32	7,1	12,0	7,5	12,1	12,3	11,8	8,6	12,0	14,2	14,5	18,0	16,3	16,3	4,9	17,0
34	6,9	11,7	7,6	10,8	12,5	11,8	8,9	12,3	14,2	14,6	18,2	16,3	16,5	4,5	16,2
36	6,9	11,4	7,6	11,0	12,5	12,0	9,3	9,7	14,2	14,2	18,3	16,0	15,7	4,4	14,8
38	7,1	11,9	7,7	10,2	12,4	12,0	9,3	12,4	13,5	13,4	18,5	15,8	16,7	4,3	15,8
40	7,1	11,6	7,7	11,0	12,2	12,0	9,2	9,2	13,1	10,9	18,7	16,2	16,3	4,1	17,7
42	7,0	11,2	7,7	10,9	12,7	12,2	9,1	10,3	13,0	11,7	18,7	15,6	15,9	4,0	15,9
44	7,0	11,7	7,8	9,3	12,7	12,1	9,1	10,6	12,6	10,3	18,9	15,9	17,1	3,8	15,5
46	7,0	11,0	7,9	9,3	12,6	12,3	8,3	11,1	12,5	9,8	19,0	16,2	15,1	3,7	17,3
48	7,0	11,2	7,9	9,1	12,6	12,1	8,3	9,2	10,8	9,2	19,1	15,9	16,0	3,7	13,4
50	6,9	10,9	8,0	8,8	12,3	12,4	8,4	10,0	11,8	8,3	19,3	15,6	15,4	3,5	13,4
52	6,8	9,7	8,0	9,2	12,5	12,4	8,1	9,5	10,8	7,2	19,4	15,5	12,6	3,3	15,1
54	6,9	9,4	8,1	9,8	12,5	12,0	8,2	9,1	11,0	7,1	19,1	15,5	15,1	3,2	12,2
56	0,2	10,5	8,3	9,1	0,4	12,0	7,1	6,6	0,4	6,8	18,7	4,4	13,8	3,1	1,5

Fonte: Do autor(2021)

A *Figura 5.6* e a *Tabela 5.8* mostram os *speedups* médios obtidos pelo algoritmo $PBFS_O$ nos grafos *não conexos* não direcionados, na máquina *SKL*. Todos os grafos apresentaram *speedup* significativo e nenhum grafo apresentou *slowdown*. Os melhores *speedups* máximos foram obtidos pelos grafos *kmer_V2a*, *kron_g500-logn21* e *Queen_4147*, enquanto os piores foram obtidos pelos grafos *mawi_201512020030*, *mawi_201512020130* e *mawi_201512020330*. O perfil das curvas de *speedup* foi, em geral, de decrescimento para um número de *threads* maior

que 28, indicando que a utilização do *hyperthreading* aftetou negativamente o desempenho do $PBFS_O$ neste teste.

Figura 5.6 – Gráfico com os *speedups* obtidos pelo algoritmo $PBFS_O$ nos grafos *não conexos* não direcionados, na máquina *SKL*.



Fonte: Do autor(2021)

Tabela 5.8 – *Speedups* obtidos pelo algoritmo $PBFS_O$ nos grafos *não conexos* não direcionados, na máquina *SKL*

Nº de threads	GAP-kron	mawi_201512020330	kmer_V1r	kmer_A2a	Queen_4147	kmer_P1a	mawi_201512020130	rgg_n_2_24_s0	kron_g500-logn21	mawi_201512020030	kmer_U1a	Bump_2911	rgg_n_2_23_s0	Cube_Coup_dt6	Cube_Coup_dt0	kmer_V2a
2	1,8	1,0	1,4	1,7	1,8	1,4	1,0	1,4	1,6	1,0	1,2	1,7	1,3	1,8	1,9	1,5
4	4,4	1,4	3,0	3,4	3,7	3,1	1,3	2,6	3,1	1,4	2,9	3,5	2,5	3,6	3,7	3,0
6	6,9	1,6	4,7	5,4	5,2	4,5	1,6	3,7	4,8	1,7	4,5	5,0	3,6	4,8	5,2	4,5
8	9,3	1,7	5,2	7,2	6,8	6,1	1,7	4,8	6,1	1,7	5,9	6,2	4,5	6,2	6,2	5,9
10	11,0	1,7	5,4	8,8	8,1	7,6	1,7	5,7	7,7	1,8	7,4	7,9	5,2	7,6	7,8	7,2
12	11,7	1,7	6,1	10,0	9,3	9,1	1,7	6,4	9,3	1,8	8,8	9,2	5,8	8,6	8,5	8,6
14	12,3	1,7	6,3	11,1	10,7	10,4	1,7	7,1	10,8	1,8	9,9	10,5	6,4	9,5	9,8	9,9
16	12,5	1,7	7,0	11,1	11,7	11,6	1,7	7,7	12,2	1,8	10,8	11,4	6,7	10,3	10,4	11,1
18	12,6	1,7	8,5	11,0	12,2	12,6	1,7	8,0	13,1	1,8	11,0	11,3	6,8	10,9	10,9	12,2
19	12,1	1,7	9,7	10,5	13,1	12,9	1,8	8,1	13,8	1,8	10,4	12,5	6,9	10,9	11,2	12,7
20	12,4	1,7	9,6	10,2	13,3	12,0	1,8	8,1	14,4	1,8	10,3	13,0	7,0	11,5	11,5	13,1
22	12,6	1,7	8,9	10,1	14,2	10,7	1,7	8,5	15,8	1,8	9,9	13,3	7,1	12,0	11,9	14,2
24	12,4	1,7	8,6	9,9	15,0	9,9	1,7	8,5	17,0	1,8	9,7	14,3	7,1	12,2	12,1	15,1
26	12,7	1,7	8,6	9,8	14,9	9,3	1,8	8,5	18,0	1,8	8,7	14,8	7,1	12,1	12,3	15,7
28	12,7	1,7	8,4	9,9	15,5	9,3	1,7	8,0	19,0	1,8	8,3	15,1	7,0	12,7	12,1	15,3
30	12,8	1,7	8,8	9,9	15,3	9,8	1,7	7,4	19,6	1,8	8,1	14,7	6,6	12,0	11,8	12,6
32	12,8	1,7	9,4	9,9	15,3	10,1	1,8	6,8	20,1	1,7	7,4	14,8	6,5	11,9	11,8	10,7
34	12,8	1,6	9,5	9,8	14,8	10,1	1,7	6,5	20,1	1,8	7,8	14,5	6,1	11,4	11,5	9,4
36	13,0	1,7	9,2	10,6	14,6	9,9	1,7	5,8	20,6	1,8	7,7	14,2	5,8	11,0	11,1	8,7
38	13,1	1,7	9,2	10,6	14,6	9,8	1,7	5,6	21,0	1,8	7,6	14,8	5,5	10,8	11,0	8,7
40	13,0	1,6	9,4	10,6	14,3	9,8	1,7	5,2	21,1	1,8	7,1	14,6	5,1	10,4	11,1	7,8
42	13,1	1,7	9,1	9,8	14,3	9,5	1,7	5,1	20,8	1,8	7,8	13,5	4,8	10,6	10,4	7,9
44	13,1	1,7	8,5	9,8	14,2	9,2	1,7	5,0	20,4	1,7	7,3	14,3	4,4	10,0	10,1	7,4
46	13,2	1,7	8,8	10,0	14,1	9,2	1,7	4,8	20,9	1,7	7,7	14,0	4,1	9,9	10,4	7,3
48	13,1	1,6	9,1	9,9	14,6	9,0	1,7	4,6	19,1	1,7	8,0	12,6	3,8	9,9	9,7	6,1
50	13,3	1,7	9,3	9,8	14,1	9,1	1,7	4,6	12,6	1,7	7,1	13,2	3,6	9,4	10,1	6,5
52	13,2	1,6	9,2	9,6	13,7	9,4	1,7	4,1	19,9	1,7	7,7	13,5	3,1	9,2	9,9	6,8
54	13,7	1,7	8,7	9,6	13,6	8,8	1,7	4,2	13,9	1,7	8,2	11,5	3,1	9,3	8,9	6,1
56	14,5	1,7	4,8	10,8	13,7	9,1	1,7	3,8	15,0	1,7	6,1	11,6	2,9	9,1	8,9	6,0

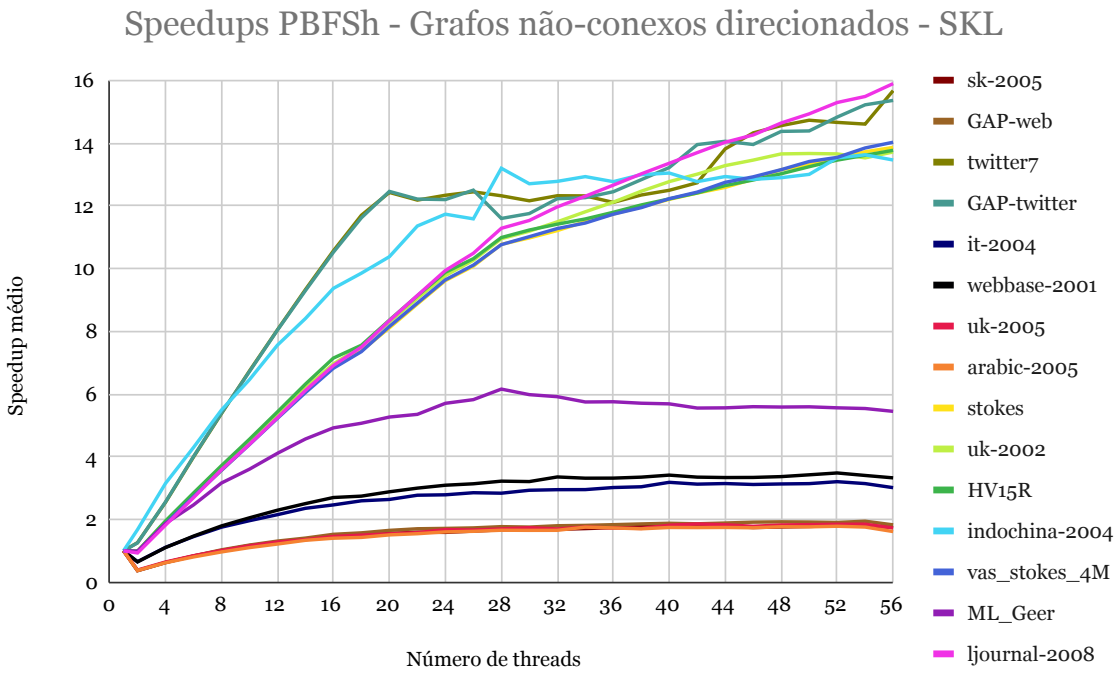
Fonte: Do autor(2021)

5.1.2.2 *Speedups* do algoritmo $PBFS_H$

A *Figura 5.7* e a *Tabela 5.9* apresentam os *speedups* obtidos pelo algoritmo $PBFS_H$ nos grafos *não conexos* direcionados, na máquina *SKL*. Todos os grafos apresentaram *speedup* significativo e nenhum grafo apresentou *slowdown* predominante para a maioria dos diferentes números de *threads* usados. Os melhores *speedups* máximos foram obtidos pelos grafos *GAP-*

twitter, *ljournal-2008* e *twitter7*, enquanto os piores foram obtidos pelos grafos *arabic-2005*, *GAP-web* e *sk-2005*. O perfil das curvas de *speedup* foi, em geral, de crescimento para um número de *threads* maior que 28, indicando que a utilização do *hyperthreading* aftetou positivamente o desempenho do $PBFS_H$ neste teste.

Figura 5.7 – Gráfico com os *speedups* obtidos pelo algoritmo $PBFS_H$ nos grafos *não conexos* direcionados, na máquina *SKL*.



Fonte: Do autor(2021)

Tabela 5.9 – *Speedups* obtidos pelo algoritmo $PBFS_H$ nos grafos *não conexos* direcionados, na máquina *SKL*

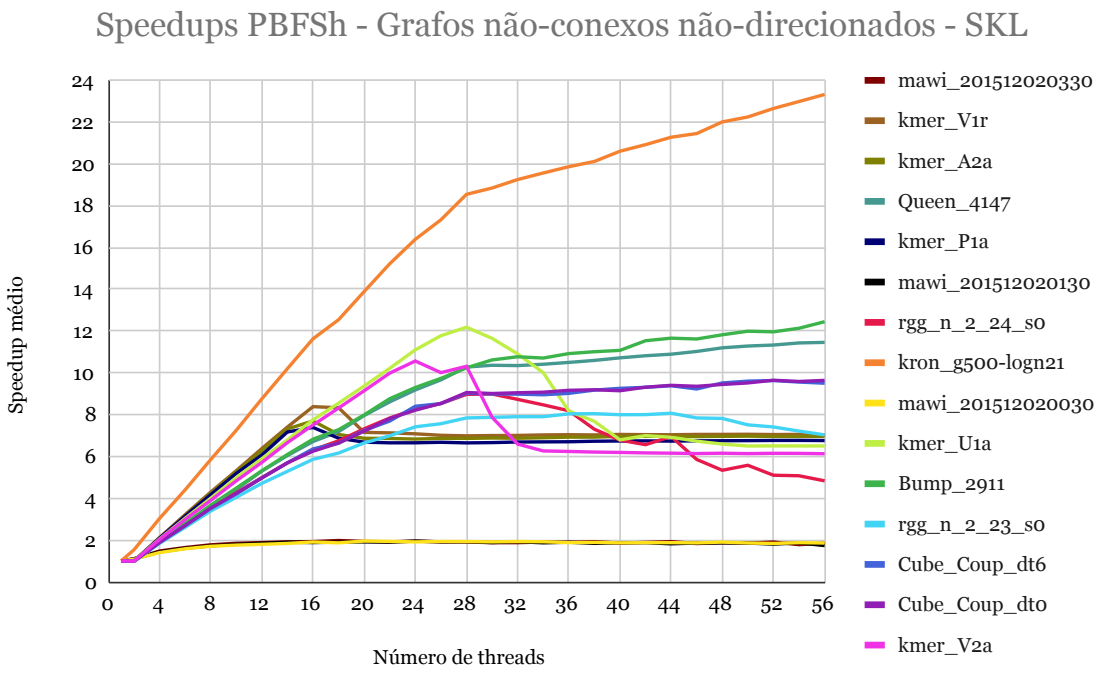
Nº de threads	sk-2005	GAP-web	twitter7	GAP-twitter	it-2004	webbase-2001	uk-2005	arabic-2005	stokes	uk-2002	HV15R	indochina-2004	vas_stokes_4M	ML_Geer	ljournal-2008
2	0,4	0,4	1,3	1,3	0,7	0,6	0,4	0,4	1,0	1,0	1,0	1,7	1,0	1,0	0,9
4	0,6	0,6	2,5	2,6	1,1	1,1	0,6	0,6	1,9	1,9	1,9	3,1	1,9	1,9	1,8
6	0,8	0,8	4,0	4,0	1,5	1,5	0,8	0,8	2,8	2,8	2,8	4,3	2,7	2,5	2,7
8	1,0	1,0	5,4	5,4	1,8	1,8	1,0	1,0	3,6	3,7	3,7	5,5	3,6	3,2	3,6
10	1,2	1,2	6,7	6,7	2,0	2,1	1,2	1,1	4,4	4,5	4,6	6,5	4,4	3,6	4,4
12	1,3	1,3	8,0	8,0	2,2	2,3	1,3	1,2	5,3	5,3	5,4	7,6	5,2	4,1	5,2
14	1,4	1,4	9,3	9,3	2,4	2,5	1,4	1,3	6,1	6,2	6,3	8,4	6,0	4,6	6,1
16	1,4	1,5	10,6	10,5	2,5	2,7	1,5	1,4	6,8	7,0	7,1	9,4	6,8	4,9	6,9
18	1,5	1,6	11,7	11,6	2,6	2,7	1,5	1,4	7,4	7,5	7,6	9,8	7,3	5,1	7,5
20	1,6	1,7	12,4	12,5	2,6	2,9	1,5	1,5	8,1	8,3	8,4	10,4	8,1	5,3	8,3
22	1,6	1,7	12,2	12,2	2,8	3,0	1,6	1,5	8,9	9,0	9,1	11,4	8,9	5,3	9,1
24	1,6	1,7	12,3	12,2	2,8	3,1	1,7	1,6	9,6	9,8	9,9	11,7	9,6	5,7	9,9
26	1,6	1,7	12,4	12,5	2,9	3,1	1,7	1,6	10,1	10,3	10,3	11,6	10,1	5,8	10,5
28	1,7	1,8	12,3	11,6	2,8	3,2	1,7	1,7	10,8	10,9	11,0	13,2	10,8	6,2	11,3
30	1,7	1,8	12,2	11,7	2,9	3,2	1,7	1,7	11,0	11,2	11,2	12,7	11,0	6,0	11,5
32	1,7	1,8	12,3	12,2	2,9	3,4	1,7	1,7	11,2	11,5	11,4	12,8	11,3	5,9	12,0
34	1,7	1,8	12,3	12,3	3,0	3,3	1,8	1,8	11,5	11,8	11,6	12,9	11,4	5,7	12,3
36	1,7	1,8	12,1	12,4	3,0	3,3	1,7	1,7	11,7	12,1	11,8	12,8	11,7	5,8	12,7
38	1,8	1,9	12,3	12,8	3,0	3,3	1,7	1,7	12,0	12,5	12,0	13,0	11,9	5,7	13,0
40	1,8	1,9	12,5	13,2	3,2	3,4	1,8	1,7	12,2	12,8	12,2	13,0	12,2	5,7	13,4
42	1,8	1,9	12,7	13,9	3,1	3,3	1,9	1,7	12,4	13,0	12,4	12,8	12,4	5,5	13,7
44	1,8	1,9	13,8	14,1	3,1	3,3	1,8	1,7	12,6	13,3	12,6	12,9	12,7	5,6	14,0
46	1,7	1,9	14,3	14,0	3,1	3,3	1,8	1,7	12,8	13,5	12,8	12,8	12,9	5,6	14,3
48	1,8	1,9	14,6	14,4	3,1	3,4	1,8	1,8	13,1	13,7	13,0	12,9	13,1	5,6	14,6
50	1,8	1,9	14,7	14,4	3,1	3,4	1,8	1,8	13,3	13,7	13,2	13,0	13,4	5,6	14,9
52	1,8	1,9	14,7	14,8	3,2	3,5	1,8	1,8	13,5	13,7	13,5	13,5	13,5	5,6	15,3
54	1,8	1,9	14,6	15,2	3,1	3,4	1,8	1,8	13,7	13,5	13,6	13,6	13,8	5,5	15,5
56	1,7	1,8	15,7	15,4	3,0	3,3	1,7	1,6	13,9	13,7	13,8	13,5	14,0	5,4	15,9

Fonte: Do autor(2021)

Na *Figura 5.8* e na *Tabela 5.10*, estão representados os *speedups* obtidos pelo algoritmo $PBFS_H$ nos grafos *não conexos* não direcionados, na máquina *SKL*. Todos os grafos apresentaram *speedup* significativo e nenhum grafo apresentou *slowdown* para nenhum dos diferentes números de *threads* usados. Os melhores *speedups* máximos foram obtidos pelos grafos *Bump_2911*, *kmer_U1a* e *kron_g500-logn21*, enquanto os piores foram obtidos pelos grafos *mawi_201512020030*, *mawi_201512020130* e *mawi_201512020330*. O perfil das curvas de *speedup* foi, em geral, de leve crescimento/estabilidade para um número de *threads* maior

que 28, indicando que a utilização do *hyperthreading* aftetou positivamente o desempenho do $PBFS_H$ neste teste.

Figura 5.8 – Gráfico com os *speedups* obtidos pelo algoritmo $PBFS_H$ nos grafos *não conexos* não direcionados, na máquina *SKL*.



Fonte: Do autor(2021)

Tabela 5.10 – *Speedups* obtidos pelo algoritmo $PBFS_H$ nos grafos *não conexos* não direcionados, na máquina *SKL*

Número de threads	mawi_201512020330	kmer_V1r	kmer_A2a	Queen_4147	kmer_P1a	mawi_201512020130	rgg_n_2_24_s0	kron_g500-logn21	mawi_201512020030	kmer_U1a	Bump_2911	rgg_n_2_23_s0	Cube_Coup_dt6	Cube_Coup_dt0	kmer_V2a
2	1,1	1,0	1,0	1,0	1,0	1,1	1,0	1,5	1,1	1,0	1,0	1,0	1,0	1,0	1,0
4	1,5	2,1	2,0	1,9	2,1	1,4	1,9	3,0	1,4	2,0	1,9	1,8	1,9	1,9	2,0
6	1,6	3,2	3,2	2,8	3,1	1,6	2,7	4,4	1,6	3,0	2,8	2,6	2,7	2,7	3,0
8	1,8	4,3	4,2	3,6	4,2	1,7	3,6	5,8	1,7	4,0	3,7	3,4	3,5	3,5	3,9
10	1,8	5,3	5,3	4,4	5,2	1,8	4,3	7,2	1,8	5,0	4,5	4,0	4,2	4,2	4,8
12	1,9	6,4	6,3	5,3	6,1	1,8	5,0	8,8	1,8	5,9	5,3	4,7	5,0	5,0	5,7
14	1,9	7,4	7,3	6,0	7,2	1,9	5,7	10,2	1,8	6,8	6,1	5,3	5,7	5,7	6,7
16	1,9	8,4	7,7	6,7	7,4	1,9	6,3	11,6	1,9	7,8	6,8	5,9	6,4	6,2	7,5
18	2,0	8,3	7,0	7,2	6,9	1,9	6,7	12,5	1,9	8,5	7,3	6,2	6,6	6,6	8,3
20	1,9	7,2	6,9	8,0	6,7	1,9	7,3	13,9	1,9	9,4	8,0	6,6	7,2	7,2	9,1
22	1,9	7,1	6,9	8,6	6,7	1,9	7,8	15,2	1,9	10,2	8,8	7,0	7,7	7,8	10,0
24	1,9	7,1	6,8	9,2	6,7	1,9	8,3	16,4	1,9	11,1	9,3	7,4	8,4	8,2	10,6
26	1,9	7,0	6,9	9,7	6,7	1,9	8,5	17,3	1,9	11,8	9,7	7,6	8,5	8,5	10,0
28	1,9	7,0	6,9	10,3	6,6	1,9	9,0	18,5	1,9	12,2	10,2	7,8	9,0	9,1	10,3
30	1,9	7,0	6,9	10,4	6,7	1,9	9,0	18,8	1,9	11,7	10,6	7,9	9,0	9,0	7,9
32	1,9	7,0	6,9	10,3	6,7	1,9	8,7	19,3	1,9	10,9	10,8	7,9	9,0	9,0	6,6
34	1,9	7,0	6,9	10,4	6,7	1,9	8,5	19,6	1,9	10,0	10,7	7,9	9,0	9,1	6,3
36	1,9	7,0	6,9	10,5	6,7	1,9	8,2	19,9	1,9	8,2	10,9	8,0	9,0	9,2	6,2
38	1,9	7,0	6,9	10,6	6,7	1,8	7,3	20,1	1,9	7,7	11,0	8,0	9,2	9,2	6,2
40	1,9	7,1	7,0	10,7	6,7	1,9	6,8	20,6	1,9	6,8	11,1	8,0	9,3	9,1	6,2
42	1,9	7,0	7,0	10,8	6,7	1,9	6,6	20,9	1,9	7,0	11,5	8,0	9,3	9,3	6,2
44	1,9	7,0	7,0	10,9	6,7	1,8	7,0	21,3	1,9	6,9	11,7	8,1	9,4	9,4	6,2
46	1,8	7,1	6,9	11,0	6,7	1,9	5,9	21,5	1,9	6,7	11,6	7,8	9,2	9,4	6,1
48	1,9	7,1	7,0	11,2	6,8	1,9	5,3	22,0	1,9	6,6	11,8	7,8	9,5	9,4	6,2
50	1,8	7,1	7,0	11,3	6,8	1,9	5,6	22,2	1,9	6,5	12,0	7,5	9,6	9,5	6,1
52	1,9	7,0	7,0	11,3	6,8	1,8	5,1	22,7	1,8	6,5	12,0	7,4	9,6	9,6	6,1
54	1,8	7,1	7,0	11,4	6,8	1,9	5,1	23,0	1,9	6,5	12,1	7,2	9,6	9,6	6,1
56	1,9	7,0	7,0	11,5	6,8	1,7	4,8	23,3	1,9	6,5	12,4	7,0	9,5	9,6	6,1

Fonte: Do autor(2021)

5.2 Testes na máquina CLX

Os resultados dos testes na máquina *CLX* estão descritos a seguir. A *Seção 5.2.1* apresenta os resultados dos testes dos grafos *conexos* e a *Seção 5.2.2* apresenta os resultados dos testes para os grafos *não conexos* nessa máquina.

5.2.1 Grafos *conexos*

A *Tabela 5.11* apresenta os tempos de execução sequenciais obtidos pelos algoritmos BFS, $PBFS_O$ e $PBFS_H$ para os grafos *conexos*, na máquina *CLX*. A coluna v_0 dessa tabela representa o vértice inicial utilizado pelos algoritmos testados. Observa-se que, em geral, os algoritmos paralelos executados sequencialmente foram mais rápidos que o algoritmo BFS. O $PBFS_O$ foi mais lento que o BFS em apenas 4 dos 30 grafos testados e a média geométrica dos tempos de execução sequenciais nesses grafos foi 5,97% menor (mais rápida) no $PBFS_O$. O algoritmo $PBFS_H$ foi mais lento que o BFS em apenas 1 dos 30 grafos testados e a média geométrica dos tempos de execução sequenciais nesses grafos foi 6,17% menor (mais rápida) no $PBFS_H$.

Tabela 5.11 – Tempos de execução sequenciais (em segundos) dos algoritmos *BFS*, $PBFS_O$, e $PBFS_H$ nos grafos *conexos*, na máquina *CLX*.

Nº	Grafo	BFS*	$PBFS_O$	BFS**	$PBFS_H$	v_0
1	cage15	2,01	1,93	2,01	1,93	4.850.801
2	wiki-topcats	0,72	0,63	0,71	0,63	1.166.027
3	cage14	0,51	0,49	0,52	0,49	1.457.234
4	rajat31	0,51	0,42	0,52	0,43	1.251
5	fem_hifreq_circuit	0,27	0,24	0,26	0,25	28.187
6	kim2	0,13	0,11	0,13	0,12	1.355
7	atmosmodl	0,30	0,24	0,31	0,24	39.404
8	torso1	0,08	0,07	0,07	0,07	4.161
9	cage13	0,13	0,12	0,13	0,12	411.097
10	ohne2	0,12	0,11	0,12	0,12	59.989
11	Chevron4	0,09	0,08	0,09	0,08	512
12	marine1	0,11	0,09	0,11	0,10	20.908
13	Hamrle3	0,31	0,26	0,30	0,25	736.801
14	Chebyshev4	0,05	0,04	0,05	0,04	1
15	largebasis	0,09	0,12	0,08	0,12	40.003
16	GAP-urand	306,41	332,62	249,59	228,49	57.337.431
17	com-Friendster	150,01	138,71	162,78	143,61	40.553.569
18	mycielskian20	20,81	20,46	20,67	20,58	786.431
19	mycielskian19	6,84	6,82	6,89	6,85	393.215
20	nlpkkt240	11,01	10,66	11,53	11,06	116.645
21	nlpkkt200	6,24	6,17	6,19	6,18	81.205
22	mycielskian18	2,31	2,26	2,30	2,29	196.607
23	com-Orkut	3,63	3,52	3,64	3,52	43.608
24	nlpkkt160	3,04	3,06	3,21	3,21	52.165
25	Flan_1565	1,01	1,00	1,03	1,00	19
26	europe_osm	11,49	9,89	11,46	9,77	36.470.776
27	delaunay_n24	4,18	3,55	4,40	3,74	3.142.594
28	mycielskian17	0,77	0,77	0,77	0,76	98.303
29	nlpkkt120	1,24	1,26	1,24	1,23	29.525
30	dielFilterV3real	1,00	0,96	1,01	0,97	16.481

* Obtidos imediatamente antes de cada teste com o algoritmo $PBFS_O$.
** Obtidos imediatamente antes de cada teste com o algoritmo $PBFS_H$.

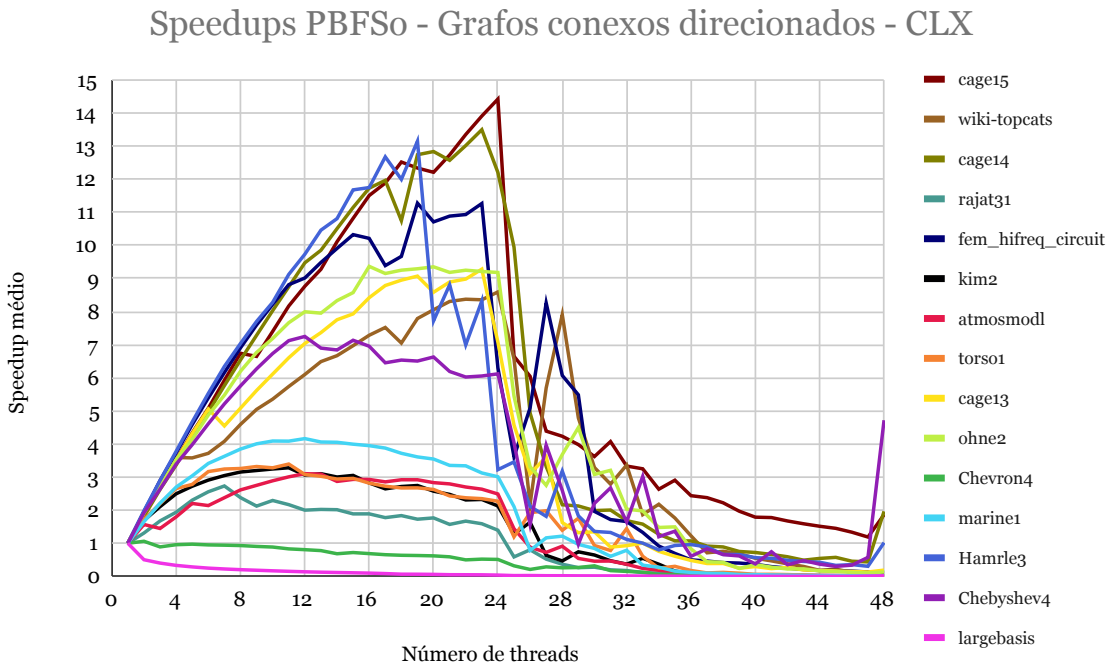
Fonte: Do autor(2021)

5.2.1.1 *Speedups* do algoritmo $PBFS_O$

A *Figura 5.9* e a *Tabela 5.12* apresentam os *speedups* obtidos pelo algoritmo $PBFS_O$ nos grafos *conexos* direcionados, na máquina *CLX*. 13 dos 15 grafos testados apresentaram *speedup* significativo para algum dos números de *threads* utilizados e 2 desses grafos apresentaram majoritariamente *slowdown*. Os melhores *speedups* máximos foram obtidos pelos grafos *cage14*, *cage15* e *Hamrle3*, enquanto os piores foram obtidos pelos grafos *Chevron4*, *largebasis* e *rajat31*. O perfil das curvas de *speedup* foi, em geral, de decrescimento acentuado para um número

de *threads* maior que 24, indicando que a utilização do *hyperthreading* aftetou negativamente o desempenho do $PBFS_O$ neste teste.

Figura 5.9 – Gráfico com os *speedups* obtidos pelo algoritmo $PBFS_O$ nos grafos *conexos* direcionados, na máquina *CLX*.



Fonte: Do autor(2021)

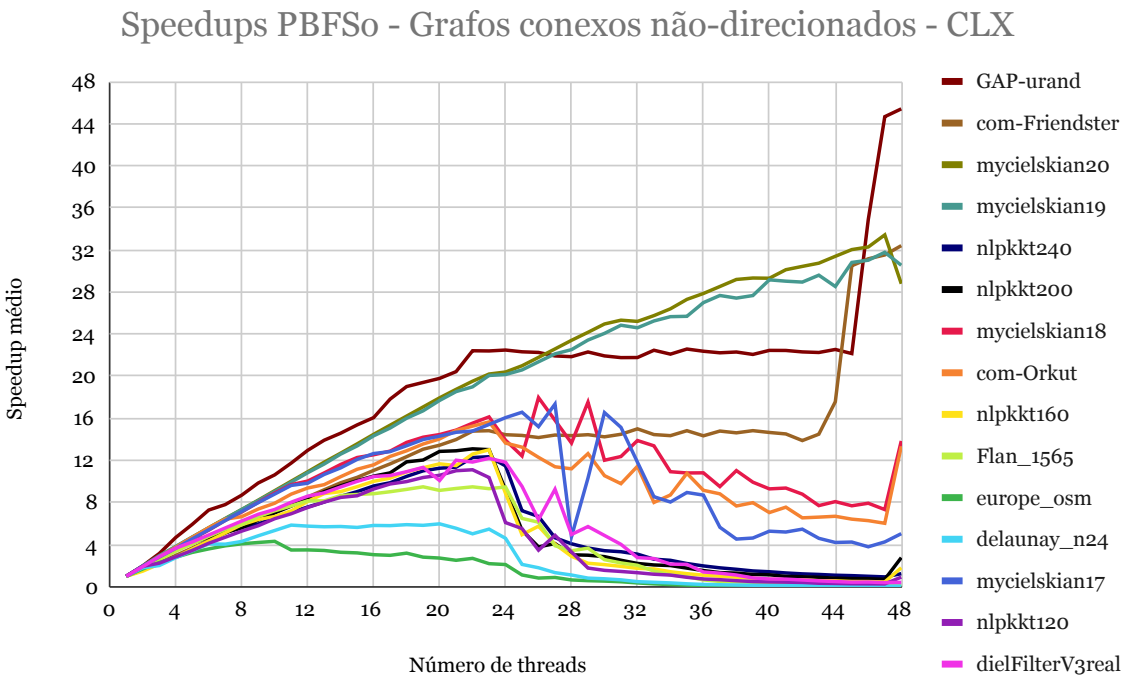
Tabela 5.12 – *Speedups* obtidos pelo algoritmo $PBFS_O$ nos grafos *conexos* direcionados, na máquina *CLX*

Nº de threads	cage15	wiki-topcats	cage14	rajat31	fem_hifreq_circuit	kim2	atmosmodl	torso1	cage13	ohne2	Chevron4	marine1	Hamrle3	Chebyshev4	largebasis
2	1,9	1,9	1,8	1,3	2,0	1,7	1,6	1,7	1,9	1,9	1,1	1,7	2,0	1,9	0,5
4	3,5	3,6	3,4	1,9	3,7	2,5	1,8	2,7	3,5	3,5	1,0	2,7	3,8	3,4	0,3
6	5,1	3,7	5,0	2,6	5,4	2,9	2,1	3,2	5,1	4,9	1,0	3,4	5,5	4,6	0,2
8	6,7	4,6	6,5	2,4	6,9	3,2	2,6	3,3	5,1	6,2	0,9	3,8	7,0	5,8	0,2
10	7,4	5,4	8,0	2,3	8,2	3,2	2,9	3,3	6,1	7,2	0,9	4,1	8,3	6,7	0,2
12	8,8	6,1	9,5	2,0	9,0	3,1	3,1	3,1	7,0	8,0	0,8	4,2	9,7	7,3	0,1
14	10,1	6,7	10,5	2,0	9,9	3,0	2,9	2,9	7,8	8,3	0,7	4,1	10,8	6,8	0,1
16	11,5	7,3	11,7	1,9	10,2	2,8	2,9	2,8	8,4	9,4	0,7	3,9	11,8	7,0	0,1
18	12,5	7,1	10,8	1,8	9,7	2,7	2,9	2,7	9,0	9,3	0,6	3,7	12,0	6,5	0,1
20	12,2	8,1	12,8	1,8	10,7	2,6	2,8	2,6	8,6	9,4	0,6	3,5	7,7	6,6	0,1
22	13,4	8,4	13,0	1,7	10,9	2,3	2,7	2,4	9,0	9,3	0,5	3,3	7,0	6,0	0,0
24	14,4	8,6	12,2	1,4	6,3	2,1	2,5	2,3	7,1	9,2	0,5	3,0	3,2	6,1	0,0
26	6,0	2,2	4,9	0,8	5,1	1,6	0,9	1,9	3,1	3,3	0,2	0,8	2,1	1,6	0,0
28	4,2	7,9	2,2	0,4	6,1	0,5	0,9	1,4	1,6	3,7	0,3	1,2	3,2	2,6	0,0
30	3,6	3,3	2,0	0,3	2,0	0,6	0,5	0,9	1,3	3,1	0,3	0,8	1,4	2,2	0,0
32	3,3	3,4	1,7	0,2	1,7	0,4	0,4	1,4	0,9	2,0	0,1	0,8	1,1	1,6	0,0
34	2,6	2,2	1,3	0,1	0,9	0,4	0,2	0,2	0,8	1,5	0,0	0,3	0,8	1,2	0,0
36	2,4	1,2	1,1	0,1	0,5	0,1	0,1	0,2	0,5	0,8	0,0	0,1	1,0	0,6	0,0
38	2,2	0,8	0,9	0,0	0,4	0,1	0,1	0,1	0,4	0,4	0,0	0,1	0,7	0,7	0,0
40	1,8	0,5	0,7	0,0	0,4	0,0	0,0	0,1	0,3	0,4	0,0	0,1	0,6	0,4	0,0
42	1,7	0,4	0,6	0,0	0,3	0,0	0,0	0,1	0,3	0,2	0,0	0,0	0,5	0,4	0,0
44	1,5	0,2	0,5	0,0	0,2	0,0	0,0	0,0	0,2	0,2	0,0	0,0	0,4	0,4	0,0
46	1,3	0,1	0,5	0,0	0,2	0,0	0,0	0,0	0,1	0,1	0,0	0,0	0,4	0,3	0,0
48	1,9	0,1	2,0	0,0	0,1	0,0	0,0	0,0	0,2	0,1	0,0	0,0	1,0	4,7	0,0

Fonte: Do autor(2021)

A *Figura 5.10* e a *Tabela 5.13* apresentam os *speedups* obtidos pelo algoritmo $PBFS_O$ nos grafos *conexos* não direcionados, na máquina *CLX*. Todos os grafos apresentaram *speedup* significativo e nenhum grafo apresentou *slowdown* de forma majoritária para os diferentes números de *threads*. Os melhores *speedups* máximos foram obtidos pelos grafos *com-Friendster*, *GAP-urand* e *mycielskian20*, enquanto os piores foram obtidos pelos grafos *delaunay_n24*, *europe_osm* e *Flan_1565*. O perfil das curvas de *speedup* foi, em geral, de decrescimento para um número de *threads* maior que 24, embora 4 grafos tenham apresentado crescimento significativo a partir dessa quantidade de *threads*, assim a utilização do *hyperthreading* aftetou negativamente o desempenho do $PBFS_H$ na maioria dos grafos deste teste.

Figura 5.10 – Gráfico com os *speedups* obtidos pelo algoritmo $PBFS_O$ nos grafos *conexos* não direcionados, na máquina *CLX*.



Fonte: Do autor(2021)

Tabela 5.13 – *Speedups* obtidos pelo algoritmo $PBFS_O$ nos grafos *conexos* não direcionados, na máquina *CLX*

Número de threads	GAP-urand	com-Friendster	mycielskian20	mycielskian19	nlpkkt240	nlpkkt200	mycielskian18	com-Orkut	nlpkkt160	Flan_1565	europe_osm	delaunay_n24	mycielskian17	nlpkkt120	dielFilterV3real
2	2,0	1,6	2,0	2,0	1,9	1,9	1,9	1,9	1,5	1,8	1,7	1,8	2,0	1,9	1,9
4	4,7	3,0	3,8	3,8	3,6	3,6	3,7	3,8	2,8	3,4	2,8	2,8	3,7	2,8	3,7
6	7,3	4,4	5,6	5,6	4,8	4,7	5,4	5,6	4,3	4,8	3,6	4,1	5,4	4,1	4,9
8	8,7	5,9	7,3	7,3	5,7	5,6	7,1	6,7	5,9	5,9	4,1	4,3	7,1	5,3	6,2
10	10,6	7,2	9,1	9,1	6,6	7,0	8,9	7,9	6,7	7,2	4,3	5,3	8,8	6,5	7,3
12	13,0	8,5	10,9	10,8	7,6	8,4	10,0	9,4	8,1	8,2	3,5	5,8	9,8	7,5	8,6
14	14,6	9,9	12,7	12,6	8,6	9,6	11,6	10,5	9,0	8,7	3,3	5,7	11,3	8,5	9,5
16	16,1	11,1	14,5	14,3	9,6	10,5	12,5	11,6	10,0	8,8	3,0	5,8	12,7	9,3	10,5
18	19,0	12,4	16,2	16,0	10,5	11,9	13,7	12,9	10,9	9,3	3,2	5,9	13,4	10,0	10,9
20	19,8	13,4	17,9	17,7	11,3	12,9	14,5	14,0	11,7	9,2	2,7	6,0	14,3	10,6	10,1
22	22,4	14,8	19,5	19,0	12,3	13,1	15,5	15,3	12,6	9,5	2,7	5,0	14,8	11,1	11,8
24	22,5	14,4	20,4	20,2	11,5	9,2	13,9	13,6	8,9	9,5	2,1	4,6	16,1	6,1	11,8
26	22,3	14,2	21,8	21,4	6,7	3,8	18,0	12,3	5,8	6,1	0,8	1,8	15,2	3,5	6,4
28	21,9	14,4	23,4	22,5	4,1	3,0	13,7	11,2	2,9	3,4	0,6	1,1	4,8	3,3	5,0
30	21,9	14,2	25,0	24,1	3,4	2,9	12,0	10,6	2,1	2,6	0,5	0,8	16,5	1,6	4,9
32	21,8	15,0	25,2	24,6	3,1	2,2	13,9	11,4	1,8	2,0	0,4	0,5	11,8	1,3	2,8
34	22,1	14,4	26,4	25,7	2,5	2,0	10,9	8,7	1,4	1,1	0,2	0,4	8,0	1,1	2,2
36	22,4	14,3	27,9	27,0	2,0	1,5	10,8	9,2	1,1	0,6	0,1	0,2	8,7	0,7	1,4
38	22,3	14,6	29,2	27,4	1,7	1,3	11,0	7,7	0,9	0,5	0,1	0,2	4,5	0,6	1,1
40	22,5	14,7	29,3	29,2	1,4	1,1	9,3	7,0	0,7	0,3	0,0	0,1	5,3	0,5	0,7
42	22,3	13,9	30,4	28,9	1,2	1,0	8,8	6,5	0,6	0,3	0,0	0,1	5,5	0,4	0,6
44	22,5	17,6	31,4	28,5	1,1	0,8	8,1	6,7	0,6	0,2	0,0	0,1	4,2	0,3	0,5
46	34,9	31,2	32,3	31,0	1,0	0,8	7,9	6,3	0,5	0,2	0,0	0,1	3,8	0,3	0,4
48	45,4	32,4	28,8	30,5	1,2	2,8	13,8	13,3	1,8	0,1	0,0	0,1	5,0	0,9	0,4

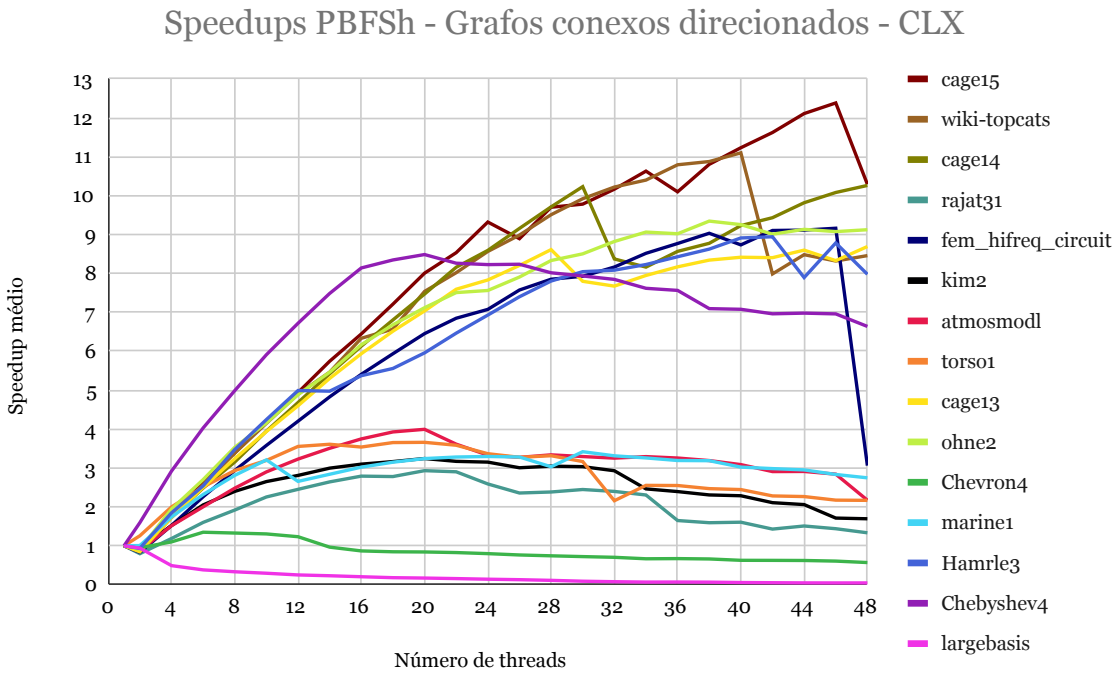
Fonte: Do autor(2021)

5.2.1.2 *Speedups* do algoritmo $PBFS_H$

A *Figura 5.11* e a *Tabela 5.14* apresentam os *speedups* obtidos pelo algoritmo $PBFS_H$ nos grafos *conexos* direcionados, na máquina *CLX*. 13 dos 15 grafos testados apresentaram *speedup* significativo e 2 grafos apresentaram majoritariamente *slowdown*. Os melhores *speedups* máximos foram obtidos pelos grafos *cage14*, *cage15* e *wiki-topcats*, enquanto os piores foram obtidos pelos grafos *Chevron4*, *largebasis* e *rajat31*. O perfil das curvas de *speedup* foi de crescimento em 7 grafos e de decrescimento leve nos demais para um número de *threads* maior que

24, assim a utilização do *hyperthreading* aftetou positivamente o desempenho do $PBFS_H$ neste teste em 46,67% dos grafos testados.

Figura 5.11 – Gráfico com os *speedups* obtidos pelo algoritmo $PBFS_H$ nos grafos *conexos* direcionados, na máquina *CLX*.



Fonte: Do autor(2021)

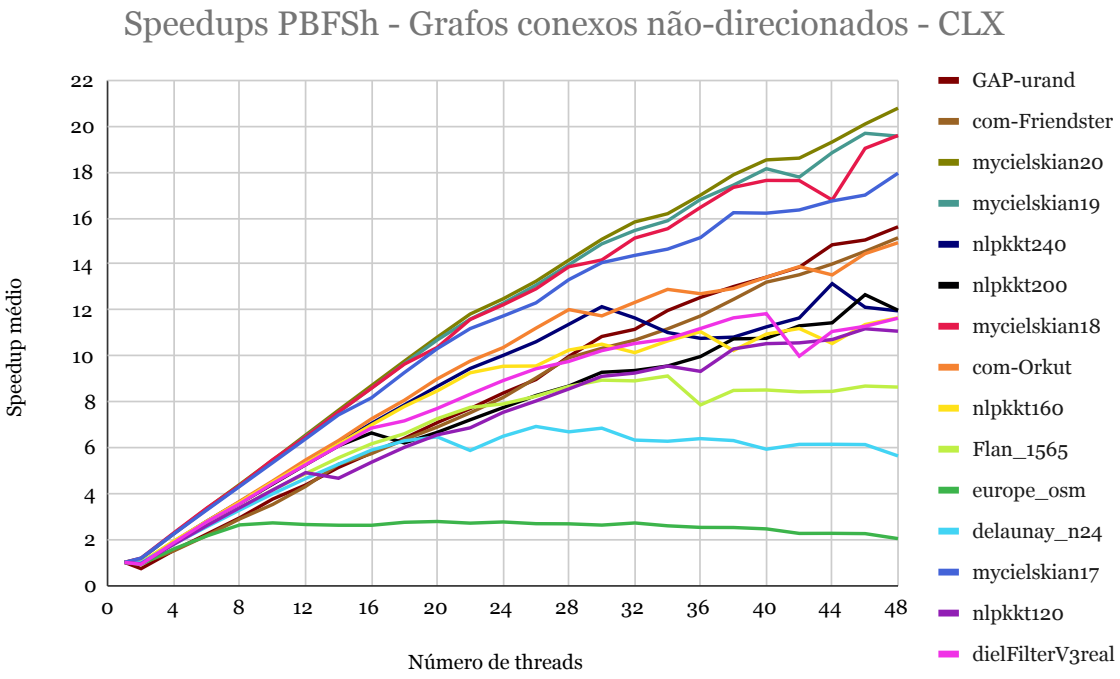
Tabela 5.14 – *Speedups* obtidos pelo algoritmo $PBFS_H$ nos grafos *conexos* direcionados, na máquina *CLX*

Nº de threads	cage15	wiki-topcats	cage14	rajat31	fem_hifreq_circuit	kim2	atmosmodl	torso1	cage13	ohne2	Chevron4	marine1	Hamrle3	Chebyshev4	largebasis
2	0,9	0,9	0,8	0,8	0,8	0,9	0,9	1,3	0,8	0,9	0,9	1,0	0,9	1,6	0,9
4	1,8	1,7	1,7	1,2	1,5	1,5	1,5	2,0	1,7	1,9	1,1	1,7	1,8	2,9	0,5
6	2,6	2,6	2,5	1,6	2,3	2,1	2,0	2,5	2,5	2,7	1,3	2,3	2,5	4,0	0,4
8	3,4	3,4	3,2	1,9	2,9	2,4	2,5	2,9	3,2	3,5	1,3	2,8	3,5	5,0	0,3
10	4,2	4,1	3,9	2,3	3,6	2,7	2,9	3,2	3,9	4,2	1,3	3,2	4,2	5,9	0,3
12	4,9	4,9	4,7	2,5	4,2	2,8	3,2	3,6	4,6	4,9	1,2	2,7	5,0	6,7	0,3
14	5,7	5,5	5,4	2,6	4,8	3,0	3,5	3,6	5,3	5,5	1,0	2,8	5,0	7,5	0,2
16	6,4	6,3	6,1	2,8	5,4	3,1	3,7	3,5	5,9	6,2	0,9	3,0	5,4	8,1	0,2
18	7,2	6,6	6,8	2,8	5,9	3,2	3,9	3,7	6,5	6,7	0,8	3,1	5,6	8,3	0,2
20	8,0	7,5	7,5	2,9	6,5	3,2	4,0	3,7	7,0	7,1	0,8	3,2	6,0	8,5	0,2
22	8,5	8,0	8,2	2,9	6,8	3,2	3,6	3,6	7,6	7,5	0,8	3,3	6,5	8,3	0,2
24	9,3	8,6	8,6	2,6	7,1	3,2	3,3	3,4	7,8	7,6	0,8	3,3	6,9	8,2	0,1
26	8,9	9,0	9,2	2,4	7,6	3,0	3,3	3,3	8,2	7,9	0,8	3,3	7,4	8,2	0,1
28	9,7	9,5	9,7	2,4	7,9	3,0	3,3	3,3	8,6	8,3	0,7	3,0	7,8	8,0	0,1
30	9,8	9,9	10,2	2,4	7,9	3,0	3,3	3,2	7,8	8,5	0,7	3,4	8,0	7,9	0,1
32	10,2	10,2	8,4	2,4	8,2	2,9	3,3	2,2	7,7	8,8	0,7	3,3	8,1	7,8	0,1
34	10,6	10,4	8,2	2,3	8,5	2,5	3,3	2,6	7,9	9,1	0,7	3,3	8,2	7,6	0,1
36	10,1	10,8	8,6	1,6	8,8	2,4	3,3	2,5	8,2	9,0	0,7	3,2	8,4	7,6	0,1
38	10,8	10,9	8,8	1,6	9,0	2,3	3,2	2,5	8,3	9,3	0,7	3,2	8,6	7,1	0,1
40	11,2	11,1	9,2	1,6	8,7	2,3	3,1	2,4	8,4	9,3	0,6	3,0	8,9	7,1	0,1
42	11,6	8,0	9,4	1,4	9,1	2,1	2,9	2,3	8,4	9,0	0,6	3,0	8,9	7,0	0,1
44	12,1	8,5	9,8	1,5	9,1	2,1	2,9	2,3	8,6	9,1	0,6	3,0	7,9	7,0	0,0
46	12,4	8,3	10,1	1,4	9,2	1,7	2,8	2,2	8,3	9,1	0,6	2,8	8,8	7,0	0,0
48	10,3	8,5	10,3	1,3	3,1	1,7	2,2	2,2	8,7	9,1	0,6	2,7	8,0	6,6	0,0

Fonte: Do autor(2021)

A *Figura 5.12* e a *Tabela 5.15* apresentam os *speedups* obtidos pelo algoritmo $PBFS_H$ nos grafos *conexos* não direcionados, na máquina *CLX*. Todos os grafos apresentaram *speedup* significativo e nenhum grafo apresentou majoritariamente *slowdown* para os diferentes números de *threads* usados. Os melhores *speedups* máximos foram obtidos pelos grafos *mycielskian18*, *mycielskian19* e *mycielskian20*, enquanto os piores foram obtidos pelos grafos *delaunay_n24*, *europe_osm* e *Flan_1565*. O perfil das curvas de *speedup* foi, em geral, de crescimento para um número de *threads* maior que 24, indicando que a utilização do *hyperthreading* aftetou positivamente o desempenho do $PBFS_H$ neste teste.

Figura 5.12 – Gráfico com os *speedups* obtidos pelo algoritmo $PBFS_H$ nos grafos *conexos* não direcionados, na máquina *CLX*.



Fonte: Do autor(2021)

Tabela 5.15 – *Speedups* obtidos pelo algoritmo $PBFS_H$ nos grafos *conexos* não direcionados, na máquina *CLX*

Número de threads	GAP-urand	com-Friendster	mycielskian20	mycielskian19	nlpkkt240	nlpkkt200	mycielskian18	com-Orkut	nlpkkt160	Flan_1565	europe_osm	delaunay_n24	mycielskian17	nlpkkt120	dielFilterV3real
2	0,7	1,0	1,2	1,2	1,0	1,0	1,2	1,0	1,0	1,0	0,9	1,0	1,2	0,9	0,9
4	1,5	1,5	2,3	2,2	1,9	1,9	2,3	1,9	1,9	1,8	1,6	1,8	2,2	1,8	1,9
6	2,2	2,2	3,4	3,3	2,8	2,7	3,4	2,8	2,8	2,6	2,2	2,6	3,3	2,6	2,8
8	2,9	2,9	4,4	4,3	3,6	3,6	4,4	3,7	3,6	3,4	2,6	3,3	4,3	3,4	3,6
10	3,8	3,5	5,5	5,4	4,5	4,4	5,5	4,5	4,5	4,2	2,7	4,0	5,3	4,2	4,4
12	4,4	4,3	6,5	6,5	5,3	5,3	6,5	5,5	5,3	4,9	2,6	4,7	6,4	4,9	5,3
14	5,1	5,3	7,6	7,5	6,2	6,1	7,6	6,3	6,1	5,6	2,6	5,3	7,4	4,7	6,1
16	5,8	5,7	8,7	8,6	7,1	6,6	8,6	7,3	7,0	6,2	2,6	5,9	8,2	5,4	6,9
18	6,4	6,3	9,8	9,7	7,9	6,2	9,6	8,1	7,8	6,6	2,7	6,3	9,3	6,0	7,2
20	7,1	6,9	10,8	10,7	8,7	6,7	10,4	9,0	8,5	7,3	2,8	6,5	10,3	6,6	7,7
22	7,7	7,5	11,8	11,6	9,4	7,2	11,6	9,8	9,3	7,8	2,7	5,9	11,2	6,9	8,3
24	8,4	8,2	12,5	12,3	10,0	7,7	12,2	10,4	9,5	7,9	2,8	6,5	11,7	7,5	8,9
26	9,0	9,0	13,2	13,1	10,6	8,3	12,9	11,2	9,6	8,2	2,7	6,9	12,3	8,0	9,4
28	10,0	9,9	14,2	14,0	11,4	8,7	13,9	12,0	10,2	8,7	2,7	6,7	13,3	8,6	9,8
30	10,8	10,3	15,1	14,9	12,1	9,3	14,2	11,7	10,5	8,9	2,6	6,8	14,1	9,1	10,2
32	11,2	10,7	15,8	15,5	11,6	9,4	15,1	12,3	10,1	8,9	2,7	6,3	14,4	9,2	10,5
34	12,0	11,2	16,2	15,9	11,0	9,6	15,5	12,9	10,6	9,1	2,6	6,3	14,6	9,6	10,7
36	12,5	11,7	17,0	16,8	10,8	10,0	16,5	12,7	11,1	7,9	2,5	6,4	15,2	9,3	11,2
38	13,0	12,5	17,9	17,4	10,8	10,7	17,3	12,9	10,2	8,5	2,5	6,3	16,2	10,3	11,7
40	13,4	13,2	18,5	18,2	11,3	10,8	17,6	13,4	10,9	8,5	2,5	5,9	16,2	10,5	11,8
42	13,9	13,5	18,6	17,8	11,6	11,3	17,6	13,9	11,2	8,4	2,3	6,1	16,4	10,6	10,0
44	14,8	14,0	19,3	18,9	13,1	11,4	16,8	13,5	10,5	8,5	2,3	6,1	16,7	10,7	11,1
46	15,0	14,6	20,1	19,7	12,1	12,7	19,0	14,5	11,4	8,7	2,3	6,1	17,0	11,2	11,3
48	15,6	15,1	20,8	19,6	12,0	12,0	19,6	14,9	11,7	8,6	2,0	5,6	18,0	11,1	11,6

Fonte: Do autor(2021)

5.2.2 Grafos *não conexos*

Esta seção apresenta os resultados dos testes dos algoritmos $PBFS_O$ (*Seção 5.2.2.1*) e $PBFS_H$ (*Seção 5.2.2.2*) nos grafos *não conexos*, na máquina *CLX*. A *Tabela 5.16* exibe os tempos de execução sequenciais dos algoritmos *BFS*, $PBFS_O$ e $PBFS_H$ nesses testes. Para cada um dos grafos, a *Tabela 5.16* também exibe os detalhes da componente fortemente conectada (*CFC*) utilizada, onde v_0 representa o vértice inicial da busca em largura utilizado pelos algoritmos e que é também o identificador da *CFC* testada, $|V|_{CFC}$ e $|A|_{CFC}$ representam o número de vértices e arestas dessa *CFC*, respectivamente.

Tabela 5.16 – Tempos de execução sequenciais (em segundos) dos algoritmos *BFS*, $PBFS_O$, e $PBFS_H$ nos grafos *não conexos*, na máquina *CLX*.

Nº	Grafo	BFS*	$PBFS_O$	BFS**	$PBFS_H$	v_0	$\|V\|_{CFC}$	$\|A\|_{CFC}$
1	sk-2005	0,13	0,11	0,09	0,11	19.879.528	16.016	104.502
2	GAP-web	0,13	0,11	0,10	0,12	19.879.528	16.016	91.686
3	twitter7	47,45	47,18	52,35	48,05	4.597.022	35.016.137	1.415.799.538
4	GAP-twitter	49,50	48,06	49,78	47,42	19.058.682	35.016.137	1.415.799.261
5	it-2004	0,13	0,12	0,10	0,12	1.906.277	9.995	40.365
6	webbase-2001	0,40	0,37	0,28	0,34	86.002.303	3.842	3.841
7	uk-2005	0,10	0,09	0,07	0,09	21.049.533	5.609	21.961
8	arabic-2005	0,06	0,05	0,04	0,05	2.960.954	9.965	19.920
9	stokes	5,30	4,99	5,25	4,99	10.901.764	11.303.355	349.175.802
10	uk-2002	5,49	4,70	5,46	4,79	15.353.975	17.994.090	289.864.964
11	HV15R	3,04	2,96	2,99	2,99	265.192	2.017.169	283.073.458
12	indochina-2004	0,38	0,38	0,37	0,37	6.086.394	6.987	48.228.945
13	vas_stokes_4M	1,99	1,88	1,96	1,87	3.429.948	4.309.660	131.456.523
14	ML_Geer	0,95	0,95	0,95	0,93	43	1.504.002	110.879.972
15	ljournal-2008	2,16	1,87	2,11	1,87	5.178.078	4.815.948	77.879.760
16	GAP-kron	159,01	152,58	161,11	154,39	71.328.661	63.032.893	2.111.611.751
17	mawi_201512020330	18,32	10,97	17,92	10,63	104.492.306	213.682.593	231.407.318
18	kmer_V1r	93,51	80,75	105,42	110,65	2	214.004.392	232.704.832
19	kmer_A2a	75,43	87,36	73,03	75,89	21.095.627	170.372.459	179.941.739
20	Queen_4147	2,82	2,91	2,80	2,90	19	4.147.110	166.823.197
21	kmer_P1a	59,84	51,62	58,91	49,93	414.528	138.896.082	148.465.346
22	mawi_201512020130	10,53	6,21	10,34	6,20	58.782.752	121.594.511	130.268.466
23	rgg_n_2_24_s0	7,91	7,19	7,90	7,25	2.421.853	16.777.215	132.557.200
24	kron_g500-logn21	2,89	2,78	2,61	2,60	1.930.587	1.543.901	91.041.917
25	mawi_201512020030	5,71	3,49	5,58	3,37	31.413.874	64.912.184	69.026.990
26	kmer_U1a	26,74	23,19	34,69	31,08	10.669.071	64.678.340	66.393.629
27	Bump_2911	1,46	1,37	1,44	1,35	1.035.677	2.852.430	65.261.670
28	rgg_n_2_23_s0	3,70	3,34	3,59	3,27	1.210.948	8.388.601	63.501.390
29	Cube_Coup_dt6	1,14	1,16	1,14	1,16	181	2.164.760	64.685.452
30	Cube_Coup_dt0	1,14	1,17	1,16	1,21	181	2.164.760	64.685.452

* Obtidos imediatamente antes de cada teste com o algoritmo $PBFS_O$.
** Obtidos imediatamente antes de cada teste com o algoritmo $PBFS_H$.

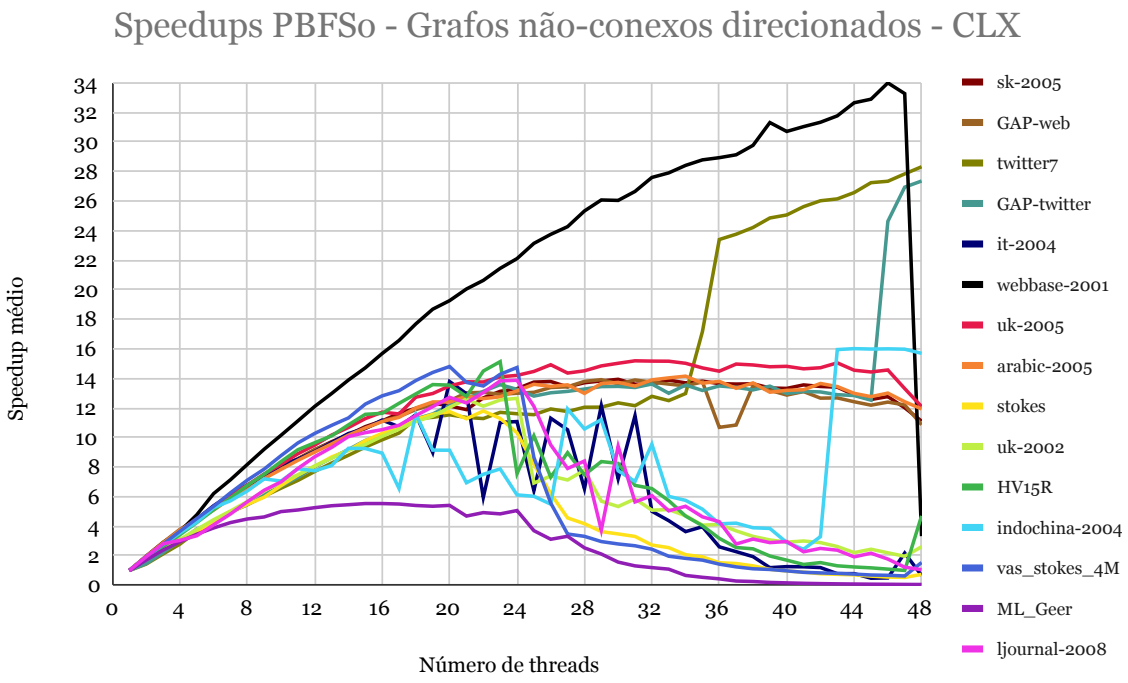
Fonte: Do autor(2021)

Na *Tabela 5.16*, observa-se que, em geral, os algoritmos paralelos executados sequencialmente foram mais rápidos que o algoritmo BFS. O $PBFS_O$ foi mais lento que o BFS em apenas 4 dos 30 grafos testados e a média geométrica dos tempos de execução sequenciais nesses grafos foi 10,62% menor (mais rápida) no $PBFS_O$. O algoritmo $PBFS_H$ foi mais lento que o BFS em 11 dos 30 grafos testados e a média geométrica dos tempos de execução sequenciais nesses grafos foi 4,23% menor (mais rápida) no $PBFS_H$.

5.2.2.1 *Speedups* do algoritmo $PBFS_O$

A *Figura 5.13* e a *Tabela 5.17* apresentam os *speedups* obtidos pelo algoritmo $PBFS_O$ nos grafos *não conexos* direcionados, na máquina *CLX*. Todos os grafos apresentaram *speedup* significativo e nenhum grafo apresentou majoritariamente *slowdown* para a maioria dos números de *threads* utilizados. Os melhores *speedups* máximos foram obtidos pelos grafos *GAP-twitter*, *twitter7* e *webbase-2001*, enquanto os piores foram obtidos pelos grafos *ML_Geer*, *stokes* e *uk-2002*. O perfil das curvas de *speedup* foi, em geral, de decrescimento/estabilidade para um número de *threads* maior que 24, indicando que a utilização do *hyperthreading* aftetou negativamente o desempenho do $PBFS_O$ neste teste.

Figura 5.13 – Gráfico com os *speedups* obtidos pelo algoritmo $PBFS_O$ nos grafos *não conexos* direcionados, na máquina *CLX*.



Fonte: Do autor(2021)

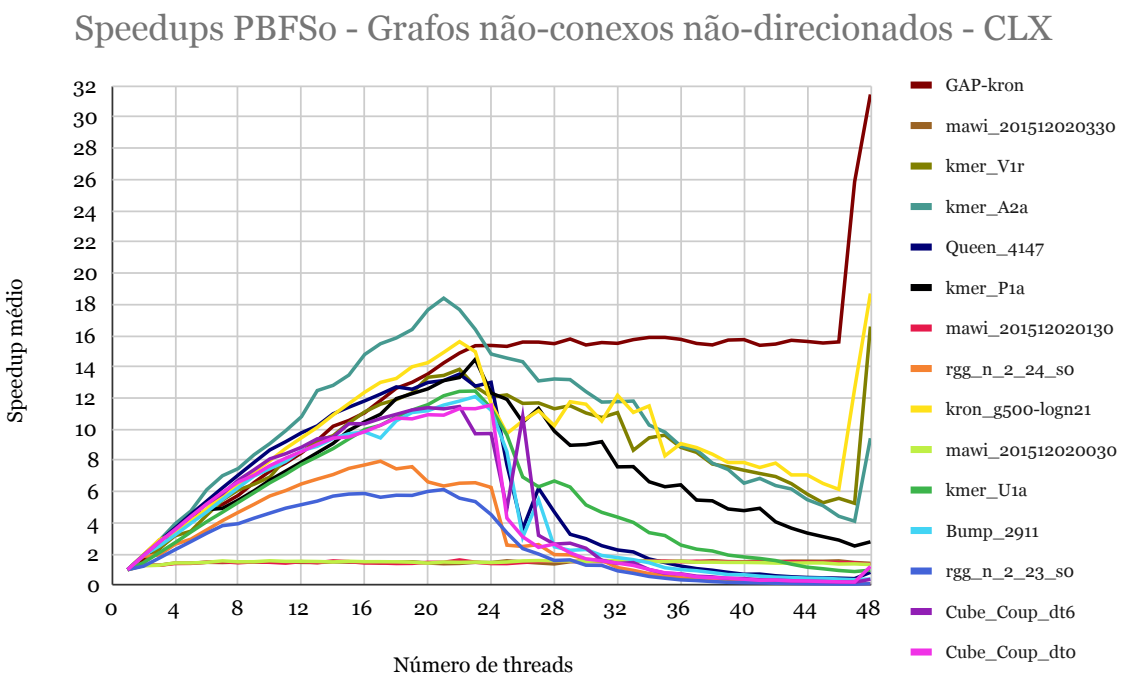
Tabela 5.17 – *Speedups* obtidos pelo algoritmo $PBFS_O$ nos grafos *não conexos* direcionados, na máquina *CLX*

Número de threads	sk-2005	GAP-web	twitter7	GAP-twitter	it-2004	webbase-2001	uk-2005	arabic-2005	stokes	uk-2002	HV15R	indochina-2004	vas_stokes_4M	ML_Geer	ljournal-2008
2	2,0	2,0	1,4	1,5	1,8	1,8	1,9	1,9	1,9	1,9	1,8	1,9	1,9	1,7	1,9
4	3,6	3,8	2,8	3,0	3,6	3,6	3,7	3,7	3,1	3,1	3,4	3,5	3,7	2,9	3,0
6	5,2	5,3	4,2	4,2	5,4	6,2	5,3	5,1	4,2	4,4	5,1	5,3	5,3	3,9	4,1
8	6,5	6,7	5,4	5,5	6,8	8,2	6,8	6,4	5,5	5,7	6,8	6,4	7,1	4,5	5,7
10	7,9	8,0	6,6	6,7	8,1	10,2	8,2	7,8	6,7	7,0	8,4	7,1	8,8	5,0	7,0
12	9,2	9,2	7,7	8,0	9,1	12,1	9,5	9,0	8,1	8,1	9,6	7,7	10,3	5,3	8,7
14	10,2	10,2	8,8	9,2	10,3	13,9	10,7	10,2	9,1	9,1	10,9	9,3	11,3	5,5	10,1
16	11,1	11,1	9,8	10,2	11,2	15,7	11,7	11,1	10,3	10,1	11,6	8,9	12,8	5,5	10,5
18	12,0	11,9	11,3	11,3	11,5	17,7	12,7	12,0	11,2	11,1	13,0	11,6	13,9	5,4	11,5
20	12,1	12,5	11,5	12,2	13,8	19,3	13,5	12,4	11,8	12,0	13,5	9,1	14,8	5,4	12,7
22	12,7	13,1	11,3	13,1	6,0	20,6	13,8	12,6	11,8	12,1	14,5	7,5	13,5	4,9	13,0
24	13,3	13,0	11,6	13,3	11,1	22,1	14,2	13,2	10,3	12,7	7,6	6,1	14,7	5,1	13,9
26	13,8	13,4	11,9	13,0	11,3	23,7	14,9	13,5	6,2	7,4	7,3	5,5	5,6	3,1	9,5
28	13,7	13,8	12,0	13,3	6,5	25,3	14,5	13,0	4,2	7,8	7,5	10,6	3,3	2,5	8,4
30	13,9	13,6	12,3	13,5	7,2	26,0	15,0	13,7	3,5	5,3	8,3	7,8	2,8	1,6	9,4
32	13,6	13,7	12,8	13,6	5,0	27,6	15,2	13,9	2,7	5,1	6,5	9,5	2,4	1,2	6,1
34	13,7	13,5	13,0	13,6	3,6	28,4	15,0	14,2	2,1	4,7	4,7	5,7	1,8	0,7	5,3
36	13,6	10,7	23,4	13,5	2,6	28,9	14,5	13,8	1,5	4,1	3,2	4,2	1,4	0,4	4,3
38	13,6	13,5	24,2	13,2	1,9	29,8	14,9	13,7	1,3	3,3	2,5	3,9	1,1	0,3	3,1
40	13,3	12,9	25,1	13,0	1,3	30,7	14,8	13,2	1,0	2,9	1,7	2,9	1,0	0,2	3,0
42	13,5	12,7	26,0	13,1	1,2	31,3	14,7	13,7	0,8	2,9	1,5	3,3	0,8	0,1	2,5
44	12,9	12,4	26,6	12,8	0,8	32,7	14,6	13,0	0,7	2,2	1,2	16,0	0,8	0,1	1,9
46	12,8	12,4	27,3	24,6	0,5	34,0	14,6	13,0	0,6	2,2	1,1	16,0	0,7	0,1	1,8
48	11,1	10,9	28,3	27,4	0,7	3,3	12,1	11,9	0,7	2,6	4,7	15,7	1,5	0,1	1,1

Fonte: Do autor(2021)

A *Figura 5.14* e a *Tabela 5.18* apresentam os *speedups* obtidos pelo algoritmo $PBFS_O$ nos grafos *não conexos* não direcionados, na máquina *CLX*. Todos os grafos apresentaram *speedup* significativo e nenhum grafo apresentou majoritariamente *slowdown* para a maioria dos números de *threads* utilizados. Os melhores *speedups* máximos foram obtidos pelos grafos *GAP-kron*, *kmer_A2a* e *kron_g500-logn21*, enquanto os piores foram obtidos pelos grafos *mawi_201512020030*, *mawi_201512020130* e *mawi_201512020330*. O perfil das curvas de *speedup* foi, em geral, de decrescimento para um número de *threads* maior que 24, indicando que a utilização do *hyperthreading* aftetou negativamente o desempenho do $PBFS_O$ neste teste.

Figura 5.14 – Gráfico com os *speedups* obtidos pelo algoritmo $PBFS_O$ nos grafos *não conexos* não direcionados, na máquina *CLX*.



Fonte: Do autor(2021)

Tabela 5.18 – *Speedups* obtidos pelo algoritmo $PBFS_O$ nos grafos *não conexos* não direcionados, na máquina *CLX*

Número de threads	GAP-kron	mawi_201512020330	kmer_V1r	kmer_A2a	Queen_4147	kmer_P1a	mawi_201512020130	rgg_n_2_24_s0	kron_g500-logn21	mawi_201512020030	kmer_U1a	Bump_2911	rgg_n_2_23_s0	Cube_Coup_dt6	Cube_Coup_dt0
2	1,6	1,3	1,4	1,9	1,9	1,9	1,3	1,9	2,0	1,3	1,4	1,9	1,2	1,9	1,9
4	3,3	1,4	3,2	3,9	3,7	3,4	1,4	2,6	3,4	1,4	2,7	3,2	2,3	3,6	3,6
6	4,7	1,5	4,5	6,1	5,4	4,8	1,5	3,5	4,9	1,5	4,1	4,7	3,3	5,2	5,2
8	5,8	1,5	6,1	7,5	7,1	5,5	1,4	4,7	6,5	1,5	5,3	6,3	3,9	6,7	6,6
10	7,3	1,5	6,9	9,1	8,7	6,7	1,5	5,7	8,0	1,6	6,6	7,5	4,6	8,1	7,7
12	8,5	1,5	8,5	10,8	9,8	7,9	1,5	6,5	9,5	1,5	7,7	8,6	5,2	8,8	8,6
14	10,2	1,5	9,6	12,8	11,0	9,1	1,6	7,1	10,9	1,5	8,8	9,4	5,7	9,5	9,4
16	11,1	1,5	11,1	14,8	11,8	10,5	1,4	7,7	12,4	1,5	10,0	9,8	5,9	10,4	9,8
18	12,6	1,5	11,9	15,8	12,7	12,0	1,4	7,5	13,3	1,5	10,8	10,5	5,8	11,0	10,7
20	13,5	1,4	13,3	17,6	13,0	12,6	1,5	6,6	14,3	1,4	11,6	11,2	6,0	11,4	10,9
22	14,9	1,4	13,8	17,7	13,5	13,3	1,6	6,5	15,6	1,5	12,4	11,8	5,6	11,5	11,3
24	15,4	1,5	12,1	14,8	13,0	12,3	1,4	6,3	11,7	1,5	11,3	11,2	4,5	9,7	11,5
26	15,6	1,5	11,7	14,3	3,6	10,4	1,5	2,5	10,5	1,6	6,9	3,1	2,4	10,8	3,1
28	15,5	1,4	11,3	13,2	4,7	9,9	1,5	2,0	10,2	1,6	6,7	2,5	1,6	2,6	2,6
30	15,4	1,5	11,0	12,4	3,0	9,0	1,5	1,7	11,6	1,5	5,2	2,3	1,3	2,4	1,7
32	15,5	1,6	11,1	11,8	2,3	7,6	1,5	1,1	12,2	1,6	4,4	1,8	0,9	1,4	1,4
34	15,9	1,6	9,4	10,3	1,7	6,6	1,5	0,7	11,5	1,5	3,4	1,4	0,6	1,0	1,0
36	15,8	1,5	8,8	8,9	1,2	6,4	1,5	0,5	9,1	1,5	2,6	1,0	0,3	0,7	0,7
38	15,4	1,6	7,8	7,8	1,0	5,4	1,5	0,4	8,4	1,5	2,2	0,8	0,2	0,5	0,5
40	15,7	1,5	7,4	6,5	0,7	4,8	1,5	0,2	7,9	1,5	1,8	0,7	0,2	0,4	0,4
42	15,5	1,5	7,0	6,4	0,6	4,1	1,5	0,2	7,8	1,4	1,6	0,5	0,1	0,3	0,3
44	15,6	1,5	5,8	5,5	0,5	3,3	1,5	0,1	7,1	1,4	1,2	0,5	0,1	0,3	0,3
45	15,5	1,5	5,3	5,1	0,5	3,1	1,5	0,1	6,5	1,4	1,1	0,5	0,1	0,2	0,2
46	15,6	1,6	5,6	4,4	0,5	2,9	1,5	0,1	6,1	1,4	0,9	0,4	0,1	0,2	0,2
48	31,5	1,4	16,6	9,4	0,8	2,8	1,3	0,1	18,7	1,3	1,0	0,4	0,1	0,4	1,2

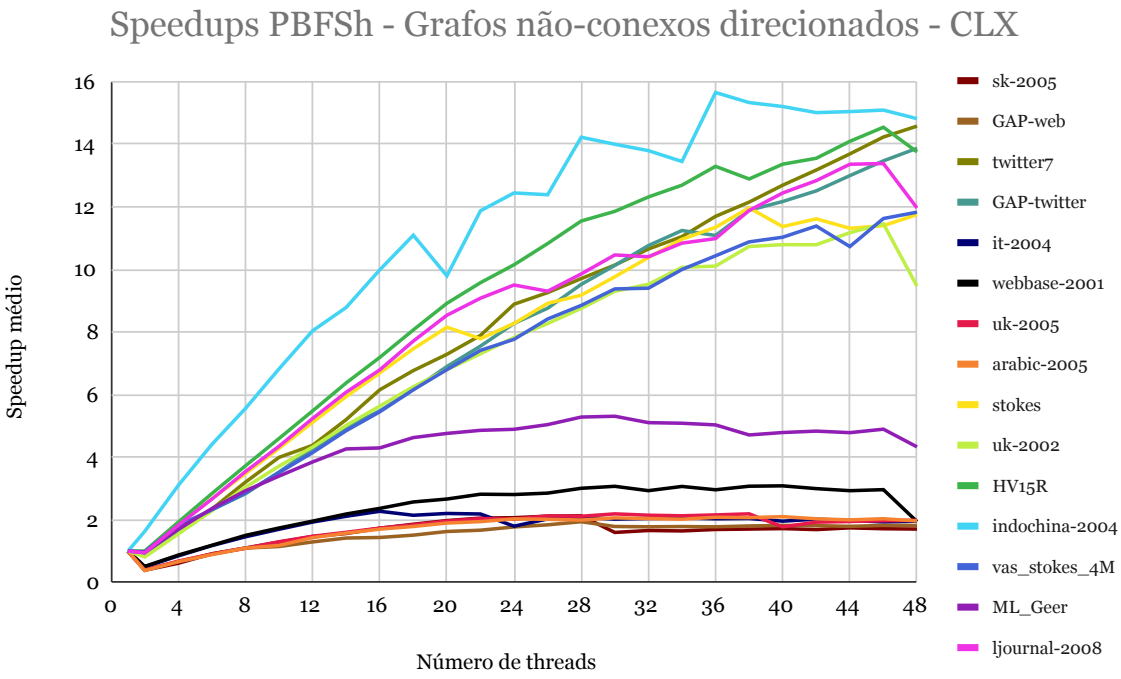
Fonte: Do autor(2021)

5.2.2.2 *Speedups* do algoritmo $PBFS_H$

A *Figura 5.15* e a *Tabela 5.19* apresentam os *speedups* obtidos pelo algoritmo $PBFS_H$ nos grafos *não conexos* direcionados, na máquina *CLX*. Todos os grafos apresentaram *speedup* significativo e nenhum grafo apresentou majoritariamente *slowdown* para a maioria dos números de *threads* utilizados. Os melhores *speedups* máximos foram obtidos pelos grafos *HV15R*, *indochina-2004* e *twitter7*, enquanto os piores foram obtidos pelos grafos *arabic-2005*, *sk-2005* e *uk-2005*. O perfil das curvas de *speedup* foi, em geral, de crescimento/estabilidade para um

número de *threads* maior que 24, indicando que a utilização do *hyperthreading* aftetou positivamente o desempenho do $PBFS_H$ neste teste.

Figura 5.15 – Gráfico com os *speedups* obtidos pelo algoritmo $PBFS_H$ nos grafos *não conexos* direcionados, na máquina *CLX*.



Fonte: Do autor(2021)

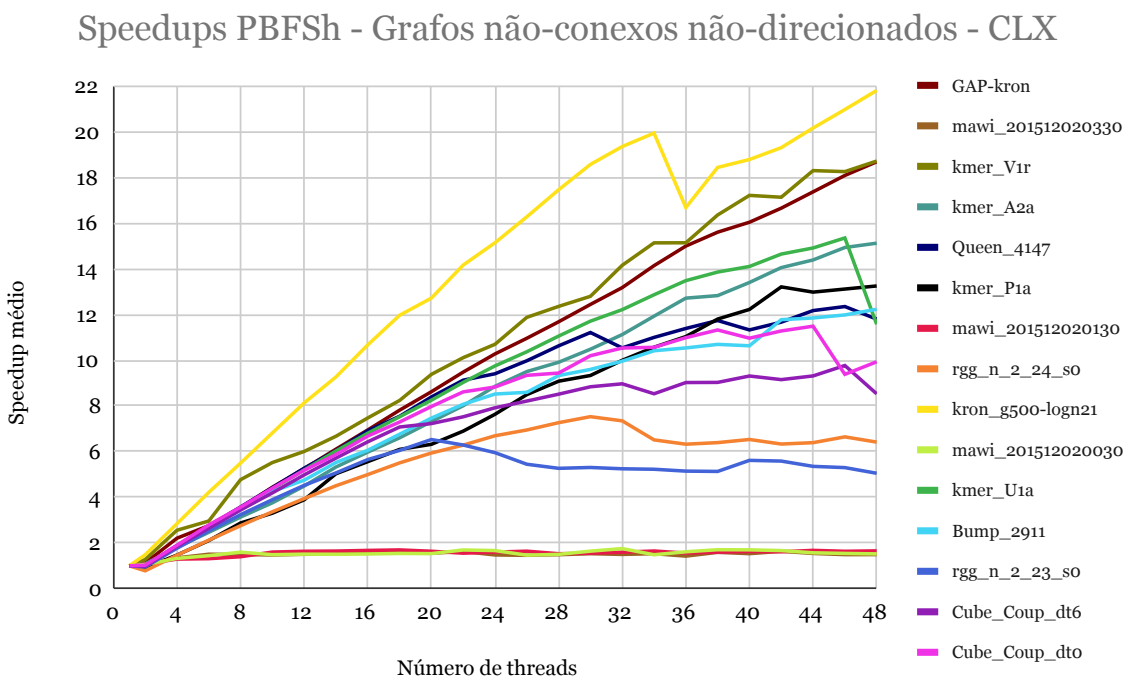
Tabela 5.19 – *Speedups* obtidos pelo algoritmo $PBFS_H$ nos grafos *não conexos* direcionados, na máquina *CLX*

Número de threads	sk-2005	GAP-web	twitter7	GAP-twitter	it-2004	webbase-2001	uk-2005	arabic-2005	stokes	uk-2002	HV15R	indochina-2004	vas_stokes_4M	ML_Geer	ljournal-2008
2	0,4	0,4	0,9	0,9	0,5	0,5	0,4	0,4	1,0	0,8	1,0	1,6	1,0	1,0	0,9
4	0,6	0,7	1,6	1,6	0,8	0,9	0,7	0,7	1,8	1,5	1,9	3,1	1,8	1,7	1,8
6	0,9	0,9	2,3	2,3	1,2	1,2	0,9	0,9	2,7	2,3	2,8	4,4	2,3	2,4	2,7
8	1,1	1,1	3,2	2,8	1,5	1,5	1,1	1,1	3,5	3,0	3,7	5,6	2,9	2,9	3,5
10	1,3	1,1	4,0	3,5	1,7	1,7	1,3	1,2	4,3	3,7	4,6	6,8	3,5	3,4	4,3
12	1,5	1,3	4,4	4,3	1,9	2,0	1,5	1,4	5,1	4,3	5,5	8,0	4,1	3,8	5,2
14	1,6	1,4	5,2	5,0	2,1	2,2	1,6	1,6	5,9	5,0	6,4	8,8	4,8	4,3	6,1
16	1,7	1,4	6,2	5,5	2,3	2,4	1,7	1,7	6,7	5,6	7,2	10,0	5,4	4,3	6,8
18	1,9	1,5	6,8	6,2	2,1	2,6	1,8	1,8	7,5	6,3	8,1	11,1	6,2	4,6	7,7
20	2,0	1,6	7,3	6,9	2,2	2,7	2,0	1,9	8,2	6,8	8,9	9,8	6,8	4,8	8,5
22	2,1	1,7	7,9	7,5	2,2	2,8	2,0	1,9	7,8	7,3	9,6	11,9	7,4	4,9	9,1
24	2,1	1,8	8,9	8,3	1,8	2,8	2,0	2,0	8,3	7,8	10,1	12,4	7,8	4,9	9,5
26	2,1	1,8	9,3	8,8	2,0	2,9	2,1	2,0	8,9	8,3	10,8	12,4	8,4	5,0	9,3
28	2,1	1,9	9,7	9,5	2,1	3,0	2,1	2,0	9,2	8,8	11,5	14,2	8,8	5,3	9,9
30	1,6	1,8	10,1	10,1	2,0	3,1	2,2	2,1	9,8	9,3	11,9	14,0	9,4	5,3	10,5
32	1,7	1,8	10,6	10,8	2,0	2,9	2,1	2,0	10,4	9,5	12,3	13,8	9,4	5,1	10,4
34	1,6	1,8	11,0	11,2	2,1	3,1	2,1	2,0	11,0	10,1	12,7	13,4	10,0	5,1	10,8
36	1,7	1,8	11,7	11,1	2,0	3,0	2,2	2,1	11,3	10,1	13,3	15,6	10,4	5,0	11,0
38	1,7	1,8	12,1	11,9	2,0	3,1	2,2	2,1	12,0	10,7	12,9	15,3	10,9	4,7	11,9
40	1,7	1,8	12,7	12,2	2,0	3,1	1,8	2,1	11,4	10,8	13,4	15,2	11,0	4,8	12,4
42	1,7	1,8	13,2	12,5	2,0	3,0	1,9	2,0	11,6	10,8	13,5	15,0	11,4	4,8	12,8
44	1,7	1,8	13,7	13,0	2,0	2,9	1,9	2,0	11,3	11,2	14,1	15,0	10,7	4,8	13,4
46	1,7	1,8	14,2	13,5	1,9	3,0	2,0	2,0	11,4	11,5	14,5	15,1	11,6	4,9	13,4
48	1,7	1,8	14,6	13,9	2,0	1,9	2,0	2,0	11,7	9,5	13,8	14,8	11,8	4,3	12,0

Fonte: Do autor(2021)

A *Figura 5.16* e a *Tabela 5.20* apresentam os *speedups* obtidos pelo algoritmo $PBFS_H$ nos grafos *não conexos* não direcionados, na máquina *CLX*. Todos os grafos apresentaram *speedup* significativo e nenhum grafo apresentou majoritariamente *slowdown* para a maioria dos números de *threads* utilizados. Os melhores *speedups* máximos foram obtidos pelos grafos *GAP-kron*, *kmer_V1r* e *kron_g500-logn21*, enquanto os piores foram obtidos pelos grafos *mawi_201512020030*, *mawi_201512020130* e *mawi_201512020330sk-2005*. O perfil das curvas de *speedup* foi, em geral, de crescimento/estabilidade para um número de *threads* maior que 24, indicando que a utilização do *hyperthreading* aftetou positivamente o desempenho do $PBFS_H$ neste teste.

Figura 5.16 – Gráfico com os *speedups* obtidos pelo algoritmo $PBFS_H$ nos grafos *não conexos* não direcionados, na máquina *CLX*.



Fonte: Do autor(2021)

Tabela 5.20 – *Speedups* obtidos pelo algoritmo $PBFS_H$ nos grafos *não conexos* não direcionados, na máquina *CLX*

Número de threads	GAP-kron	mawi_201512020330	kmer_V1r	kmer_A2a	Queen_4147	kmer_P1a	mawi_201512020130	rgg_n_2_24_s0	kron_g500-logn21	mawi_201512020030	kmer_U1a	Bump_2911	rgg_n_2_23_s0	Cube_Coup_dt6	Cube_Coup_dt0
2	1,1	1,1	1,3	1,1	1,0	0,9	1,1	0,8	1,5	1,1	1,0	1,0	0,9	1,0	1,0
4	2,2	1,3	2,6	1,8	1,9	1,5	1,3	1,5	2,8	1,3	1,8	1,8	1,8	1,8	1,9
6	2,8	1,5	3,0	2,5	2,7	2,1	1,3	2,1	4,2	1,4	2,7	2,6	2,5	2,6	2,8
8	3,5	1,5	4,8	3,1	3,6	2,9	1,4	2,8	5,5	1,6	3,5	3,5	3,2	3,4	3,6
10	4,4	1,5	5,5	3,8	4,4	3,3	1,6	3,4	6,8	1,5	4,4	4,3	3,9	4,2	4,4
12	5,3	1,5	6,0	4,5	5,3	3,9	1,6	3,9	8,1	1,5	5,2	4,7	4,5	5,0	5,2
14	6,1	1,5	6,7	5,3	6,0	5,0	1,6	4,5	9,3	1,5	6,0	5,5	5,1	5,7	5,9
16	6,9	1,6	7,5	6,0	6,9	5,6	1,7	5,0	10,7	1,5	6,8	6,1	5,7	6,4	6,7
18	7,8	1,6	8,2	6,6	7,5	6,1	1,7	5,5	12,0	1,5	7,5	6,8	6,1	7,1	7,3
20	8,6	1,5	9,4	7,3	8,4	6,3	1,6	5,9	12,7	1,5	8,3	7,5	6,5	7,2	8,0
22	9,5	1,6	10,1	8,0	9,1	6,9	1,6	6,3	14,2	1,7	9,0	8,1	6,3	7,5	8,6
24	10,3	1,5	10,7	8,9	9,4	7,6	1,6	6,7	15,1	1,7	9,8	8,5	6,0	7,9	8,8
26	11,0	1,5	11,9	9,5	10,0	8,5	1,6	6,9	16,3	1,5	10,4	8,6	5,4	8,2	9,3
28	11,7	1,5	12,4	9,9	10,6	9,1	1,5	7,3	17,5	1,5	11,1	9,3	5,3	8,5	9,4
30	12,4	1,5	12,8	10,5	11,2	9,3	1,5	7,5	18,6	1,6	11,7	9,6	5,3	8,8	10,2
32	13,2	1,5	14,2	11,1	10,5	10,0	1,6	7,3	19,4	1,8	12,2	10,0	5,2	9,0	10,5
34	14,1	1,5	15,1	11,9	11,0	10,6	1,6	6,5	19,9	1,5	12,9	10,4	5,2	8,5	10,6
36	15,0	1,4	15,1	12,7	11,4	11,0	1,6	6,3	16,7	1,6	13,5	10,5	5,1	9,0	11,0
38	15,6	1,6	16,4	12,8	11,7	11,8	1,6	6,4	18,4	1,7	13,9	10,7	5,1	9,0	11,3
40	16,0	1,5	17,2	13,4	11,3	12,2	1,6	6,5	18,8	1,7	14,1	10,6	5,6	9,3	11,0
42	16,7	1,6	17,1	14,1	11,7	13,2	1,6	6,3	19,3	1,7	14,6	11,8	5,6	9,2	11,3
44	17,4	1,5	18,3	14,4	12,2	13,0	1,7	6,4	20,2	1,6	14,9	11,9	5,4	9,3	11,5
46	18,1	1,5	18,3	14,9	12,4	13,1	1,6	6,6	21,0	1,5	15,4	12,0	5,3	9,8	9,4
48	18,7	1,5	18,7	15,1	11,8	13,3	1,7	6,4	21,8	1,5	11,6	12,2	5,1	8,5	9,9

Fonte: Do autor(2021)

5.3 Resumo

Os algoritmos $PBFS_O$ e $PBFS_H$ desenvolvidos neste trabalho, quando executados sequencialemente, são, em geral, mais rápidos que o algoritmo de busca em largura sequencial com fila (BFS). O $PBFS_O$ com 1 *thread* foi mais lento que o BFS em apenas 22 dos 123 testes feitos com esse algoritmo nas duas máquinas de teste, enquanto o $PBFS_H$ com 1 *thread* foi mais lento que o BFS em apenas 22 dos seus 120 testes nas máquinas *SKL* e *CLX*. A média geométrica dos tempos de execução sequenciais dos algoritmos $PBFS_O$ e $PBFS_H$, foi 5,93% e 4,82% menor que a do BFS, respectivamente.

As *Tabelas* 5.21 e 5.22 exibem os *speedups* máximos obtidos pelos algoritmos $PBFS_O$ e $PBFS_H$ em cada grafo nas máquinas *SKL* e *CLX*. Além disso, para que fosse possível a análise do impacto da escolha do vértice inicial no desempenho das implementações, na *Tabela* 5.21 foi incluída a coluna $e(v_0)$, que representa a excentricidade do vértice inicial usado, já na *Tabela* 5.22, além da coluna $e(v_0)$, foram incluídas também as colunas $|V_{CFC}|$ e $|A_{CFC}|$, que representam a quantidade de vértices e a quantidade de arestas da componente fortemente conectada usada, respectivamente.

Observa-se que o $PBFS_O$, em geral, obteve *speedups* máximos maiores que o $PBFS_H$. O $PBFS_O$ obteve *speedups* máximos maiores que o $PBFS_H$ em 35 dos 60 testes (2 testes por grafo, 1 em cada máquina) em que os dois algoritmos foram ambos testados. Assim, por essa métrica, pode-se dizer que o desempenho do algoritmo $PBFS_O$ foi melhor do que o desempenho do algoritmo $PBFS_H$. Como pode ser verificado pelas *Figuras* 5.1 a 5.16, o perfil geral das curvas de *speedup* foi de decrescimento para o algoritmo $PBFS_O$ e de crescimento para o algoritmo $PBFS_H$, para quantidades de *threads* acima da quantidade de núcleos físicos da máquina de realização dos testes, indicando que a utilização do *hyperthreading* aftetou negativamente o desempenho do $PBFS_O$ e positivamente desempenho do $PBFS_H$.

Em relação ao impacto da escolha do vértice inicial, observa-se o seguinte: parece que existe uma correleção de proporcionalidade inversa entre a excentricidade do vértice inicial e o desempenho dos algoritmos, isso é mais evidente de ser pecebido nos grafos *conexos*. A nossa hipótese que busca explicar esse comportamento reside no fato das implementações desenvolvidadas descobrirem os níveis do grafo de entrada de maneira sequencial ($PBFS_O$) e sequencial intra-processo ($PBFS_H$), sendo o paralelismo implementado essencialmente somente intra-nível. Assim, quanto maior a excentricidade do vértice inicial, maior será o trabalho sequencial realizado pelos algoritmos, implicando em um pior desempenho (*speedup*). Nos grafos *não conexos*, além do impacto de $e(v_0)$ também parece haver um outro impacto significativo advindo da escolha do vértice inicial no sentido de que esse vértice também definirá a componente fortemente conectada sobre a qual os algoritmos executarão e, quando esta componente tiver um tamanho muito pequeno, o paralelismo não será bem aproveitado, contribuindo negativamente para o desempenho.

Tabela 5.21 – *Speedups* máximos obtidos pelos algoritmos $PBFS_O$ e $PBFS_H$ nos grafos *conexos*, nas máquinas *SKL* e *CLX*.

		SKL	SKL	CLX	CLX	
Nº	**Grafo**	**$PBFS_O$**	**$PBFS_H$**	**$PBFS_O$**	**$PBFS_H$**	**$e(v_0)$**
1	cage15	17,0	14,5	14,4	12,4	30
2	wiki-topcats	11,7	12,6	8,6	11,1	254
3	cage14	13,5	12,9	13,5	10,3	25
4	rajat31	2,7	3,5	2,7	2,9	1253
5	fem_hifreq_circuit	10,6	9,5	11,3	9,2	46
6	kim2	2,8	3,2	3,3	3,2	337
7	atmosmodl	3,2	4,3	3,1	4,0	428
8	torso1	3,1	3,7	3,4	3,7	127
9	cage13	9,3	9,7	9,3	8,7	23
10	ohne2	8,8	8,9	9,4	9,3	27
11	Chevron4	1,1	1,3	1,1	1,3	1393
12	marine1	3,3	4,3	4,2	3,4	161
13	Hamrle3	10,3	12,6	13,2	8,9	18
14	Chebyshev4	7,0	7,8	7,3	8,5	1
15	largebasis	1,0	1,0	1,0	1,0	40001
16	GAP-urand	15,8	-	45,4	15,6	7
17	com-Friendster	17,7	-	32,4	15,1	21
18	mycielskian20	13,6	9,8	33,4	20,8	2
19	mycielskian19	20,4	14,0	31,8	19,7	2
20	nlpkkt240	13,1	10,1	12,4	13,1	240
21	nlpkkt200	16,8	11,5	13,1	12,7	200
22	mycielskian18	21,9	14,9	18,0	19,6	2
23	com-Orkut	24,7	16,3	15,7	14,9	6
24	nlpkkt160	15,1	11,9	13,0	11,7	160
25	Flan_1565	10,1	8,4	9,5	9,1	197
26	europe_osm	3,5	4,9	4,3	2,8	21894
27	delaunay_n24	7,2	8,4	6,0	6,9	1464
28	mycielskian17	18,7	14,4	17,4	18,0	2
29	nlpkkt120	11,9	11,7	11,1	11,2	120
30	dielFilterV3real	14,8	12,2	12,2	11,8	61
31	channel-500x100x100-b050	7,1	7,5	-	-	498
32	audikw_1	15,1	11,8	-	-	56

Fonte: Do autor(2021)

Tabela 5.22 – *Speedups* máximos obtidos pelos algoritmos $PBFS_O$ e $PBFS_H$ nos grafos *não conexos*, nas máquinas *SKL* e *CLX*.

		SKL	SKL	CLX	CLX			
Nº	**Grafo**	**$PBFS_O$**	**$PBFS_H$**	**$PBFS_O$**	**$PBFS_H$**	**$e(v_0)$**	**$\|V\|_{CFC}$**	**$\|A\|_{CFC}$**
1	sk-2005	9,3	1,8	13,9	2,1	5	16016	104502
2	GAP-web	12,0	1,9	13,9	1,9	5	16016	91686
3	twitter7	8,3	15,7	28,3	14,6	12	35016137	1415799538
4	GAP-twitter	12,1	15,4	27,4	13,9	12	35016137	1415799261
5	it-2004	12,8	3,2	13,8	2,3	4	9995	40365
6	webbase-2001	12,4	3,5	34,0	3,1	1	3842	3841
7	uk-2005	9,7	1,9	15,2	2,2	10	5609	21961
8	arabic-2005	12,4	1,8	14,2	2,1	4	9965	19920
9	stokes	15,0	13,9	11,8	12,0	237	11303355	349175802
10	uk-2002	16,0	13,7	12,7	11,5	39	17994090	289864964
11	HV15R	19,4	13,8	15,1	14,5	69	2017169	283073458
12	indochina-2004	16,5	13,6	16,0	15,6	2	6987	48228945
13	vas_stokes_4M	17,1	14,0	14,8	11,8	67	4309660	131456523
14	ML_Geer	6,1	6,2	5,5	5,3	499	1504002	110879972
15	ljournal-2008	17,7	15,9	13,9	13,4	39	4815948	77879760
16	GAP-kron	14,5	-	31,5	18,7	5	63032893	2111611751
17	mawi_201512020330	1,8	2,0	1,6	1,6	6	213682593	231407318
18	kmer_V1r	9,7	8,4	16,6	18,7	322	214004392	232704832
19	kmer_A2a	11,3	7,7	18,4	15,1	513	170372459	179941739
20	Queen_4147	15,7	11,5	13,5	12,4	175	4147110	166823197
21	kmer_P1a	12,9	7,4	14,5	13,3	486	138896082	148465346
22	mawi_201512020130	1,8	1,9	1,6	1,7	7	121594511	130268466
23	rgg_n_2_24_s0	8,6	9,0	8,0	7,5	1910	16777215	132557200
24	kron_g500-logn21	21,2	23,3	18,7	21,8	4	1543901	91041917
25	mawi_201512020030	1,9	1,9	1,6	1,8	8	64912184	69026990
26	kmer_U1a	11,0	12,2	12,4	15,4	889	64678340	66393629
27	Bump_2911	15,1	12,4	12,1	12,2	95	2852430	65261670
28	rgg_n_2_23_s0	7,2	8,1	6,1	6,5	1389	8388601	63501390
29	Cube_Coup_dt6	12,7	9,6	11,5	9,8	152	2164760	64685452
30	Cube_Coup_dt0	12,3	9,6	11,5	11,5	152	2164760	64685452
31	kmer_V2a	15,7	10,6	-	-	864	53500237	57076126

Fonte: Do autor(2021)

6 CONCLUSÃO

Neste trabalho foram desenvolvidas duas versões adaptadas do algoritmo *PBFS* de Leiserson e Schardl (2010). Uma das versões criadas, a $PBFS_O$, utilizou paralelização por *OpenMP* puro e, na outra versão, na $PBFS_H$, utilou-se paralelização híbrida por *MPI* e *OpenMP*. Essas implementações se mostraram adaptações funcionais e, em geral, apresentaram *speedup* significativo.

Foram realizados testes extensivos em 2 máquinas de alto desempenho, com 63 grafos distintos. Nesses testes, o algortimo $PBFS_O$ atingiu *speedups* máximos de 1,0 a 45,4, enquanto o $PBFS_H$ atingiu *speedups* máximos entre 1,0 e 23,3. Além disso, a média geométrica dos tempos de execução sequenciais dos algoritmos $PBFS_O$ e $PBFS_H$, foi 5,93% e 4,82% menor, respectivamente, do que a média geométrica dos tempos de execução do algoritmo de busca em largura sequencial com fila (*BFS*). Foi observado o fato de que existe a possibilidade de que a escolha do vértice inicial acarrete em *speedups* ruins, devido a este vértice possuir grande excentricidade e/ou pertencer a uma componente fortemente conectada que possui um pequeno número de vértices; no entanto, acreditamos que é necessária a realização de testes específicos para compravar e dimensionar esses impactos.

Não foi observada vantagem, no sentido de obtenção de melhores *speedups*, da versão com paralelização por *MPI* e *OpenMP* sobre a versão com paralelização por *OpenMP* puro, pelo contrário, foi observada uma ligeira piora de performance nesse caso, no entanto, observou-se que o $PBFS_H$ tende a fazer melhor uso do *hyperthreading* do que a $PBFS_O$.

6.1 Trabalhos futuros

Considerando-se as implementações da *PBFS* desenvolvidas neste trabalho, destaca-se como um ponto relevante passível de ser explorado em futuros estudos o desenvolvimento de uma implementação com *MPI* e possivelmente *OpenMP* de uma adaptação do algoritmo $PBFS_H$ que seja destinado à execução em múltiplos nós computacionais, essa versão da busca em largura paralela poderia fazer uso de uma estratégia de união de vetores de distâncias parciais (semelhante à estratégia utilizada para o algoritmo $PBFS_H$) que, ao final do processamento em cada nó, seriam enviados a um nó responsável por fazer a união dos mesmos e produzir o vetor de distâncias final.

REFERÊNCIAS

BELOVA, M.; OUYANG, M. Breadth-first search with a multi-core computer. In: **Proceedings - 2017 IEEE 31st International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2017**. Lake Buena Vista, FL, USA: Institute of Electrical and Electronics Engineers Inc., 2017. p. 579–587. ISBN 9781538634080. Disponível em: <https://ieeexplore.ieee.org/document/7965097>.

BRANDÃO, D. et al. Estudo sobre o uso do framework openmp na paralelização de um algoritmo para o problema de busca em largura paralela. In: **LI Simpósio Brasileiro de Pesquisa Operacional**. Campinas, SP, Brasil: Galoá, 2019. Disponível em: <encurtador.com.br/gDQ26>.

CHENEY, C. J. A nonrecursive list compacting algorithm. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 13, n. 11, p. 677–678, nov 1970. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/362790.362798>.

CHHUGANI, J. et al. Fast and efficient graph traversal algorithm for CPUs: Maximizing single-node efficiency. In: **Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IPDPS 2012**. Shanghai, China: IEEE, 2012. p. 378–389. ISBN 9780769546759. Disponível em: <https://ieeexplore.ieee.org/abstract/document/6267875>.

CORMEN, T. H. et al. **Introduction to Algorithms, Third Edition**. 3rd. ed. Cambridge, Massachusetts & London, England: The MIT Press, 2009. ISBN 0262033844.

DAVIS, T.; HU, Y. The university of florida sparse matrix collection. **ACM Transactions on Mathematical Software (TOMS)**, v. 38, n. 1, p. 1, 11 2011.

HASSAAN, M. A.; BURTSCHER, M.; PINGALI, K. Ordered and unordered algorithms for parallel breadth first search. In: **Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques**. New York, NY, USA: Association for Computing Machinery, 2010. (PACT '10), p. 539–540. ISBN 9781450301787. Disponível em: <https://doi.org/10.1145/1854273.1854341>.

LEISERSON, C. E.; SCHARDL, T. B. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In: **Annual ACM Symposium on Parallelism in Algorithms and Architectures**. New York, New York, USA: ACM Press, 2010. p. 303–314. ISBN 9781450300797. Disponível em: <http://portal.acm.org/citation.cfm?doid=1810479.1810534>.

MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, part i. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 3, n. 4, p. 184–195, apr 1960. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/367177.367199>.

OLIVEIRA, S. L. G. de; SILVA, L. M. Evolving reordering algorithms using an ant colony hyperheuristic approach for accelerating the convergence of the ICCG method. **Engineering with Computers**, Springer Science and Business Media Deutschland GmbH, v. 36, n. 4, p. 1857–1873, oct 2019. ISSN 14355663. Disponível em: <https://link.springer.com/article/10.1007/s00366-019-00801-5>.

OLIVEIRA, S. L. G. de; SILVA, L. M. An ant colony hyperheuristic approach for matrix bandwidth reduction. **Applied Soft Computing Journal**, Elsevier Ltd, v. 94, p. 106434, sep 2020. ISSN 15684946. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1568494620303744>.

SHUN, J.; BLELLOCH, G. E. Ligra: A lightweight graph processing framework for shared memory. In: **Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**. New York, NY, USA: Association for Computing Machinery, 2013. (PPoPP '13), p. 135–146. ISBN 9781450319225. Disponível em: <https://doi.org/10.1145/2442516.2442530>.

SHUN, J.; BLELLOCH, G. E. Ligra: A lightweight graph processing framework for shared memory. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 48, n. 8, p. 135–146, fev. 2013. ISSN 0362-1340. Disponível em: <https://doi.org/10.1145/2517327.2442530>.

ST. JOHN, T.; DENNIS, J. B.; GAO, G. R. Massively parallel breadth first search using a tree-structured memory model. In: **Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores**. New York, NY, USA: Association for Computing Machinery, 2012. (PMAM '12), p. 115–123. ISBN 9781450312110. Disponível em: <https://doi.org/10.1145/2141702.2141715>.