PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

GABRIEL GIORDANI DOS SANTOS

UMA ANÁLISE SOBRE A ACURÁCIA E A ESCALABILIDADE DE ALGORITMOS PARALELOS DE DETECÇÃO DE COMUNIDADES EM GRAFOS

Porto Alegre
2022

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UMA ANÁLISE SOBRE A ACURÁCIA E A ESCALABILIDADE DE ALGORITMOS PARALELOS DE DETECÇÃO DE COMUNIDADES EM GRAFOS

GABRIEL GIORDANI DOS SANTOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Dr. César Augusto Fonticielha De Rose

Porto Alegre
2022

Ficha Catalográfica

S237a Santos, Gabriel Giordani dos

Uma análise sobre a acurácia e a escalabilidade de algoritmos paralelos de detecção de comunidades em grafos / Gabriel Giordani dos Santos. – 2022.

62 p.

Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. César Augusto Fonticielha De Rose.

1. Detecção de comunidades. 2. Computação paralela. 3. Teoria dos grafos. I. De Rose, César Augusto Fonticielha. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS com os dados fornecidos pelo(a) autor(a).

Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

GABRIEL GIORDANI DOS SANTOS

UMA ANÁLISE SOBRE A ACURÁCIA E A ESCALABILIDADE DE ALGORITMOS PARALELOS DE DETECÇÃO DE COMUNIDADES EM GRAFOS

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação, Escola Politécnica da Pontifícia Universidade Católica do Rio Grande do Sul.

Aprovado(a) em 29 de Agosto de 2022.

BANCA EXAMINADORA:

Prof. Dr. Renato Antônio Celso Ferreira (PPGCC/UFMG)

Prof. Dr. Duncan Dubugras Alcoba Ruiz (PPGCC/PUCRS)

Prof. Dr. César Augusto Fonticielha De Rose (PPGCC/PUCRS - Orientador)

AGRADECIMENTOS

Agradeço ao meu orientador, o Dr. César A. F. De Rose, e ao Dr. Kartik Lakhotia por me guiarem na construção desta pesquisa.

Agradeço à CAPES por prover apoio financeira à esta pesquisa e ao Laboratório de Alto Desempenho da Pontifícia Universidade Católica do Rio Grande do Sul (LAD-IDEIA/PUCRS, Brasil) por fornecer suporte e recursos tecnológicos, que contribuíram para o desenvolvimento do presente projeto e para os resultados reportados nesta pesquisa.

UMA ANÁLISE SOBRE A ACURÁCIA E A ESCALABILIDADE DE ALGORITMOS PARALELOS DE DETECÇÃO DE COMUNIDADES EM GRAFOS

RESUMO

Detecção de comunidades é um tipo de análise topológica amplamente utilizada em análise de grafos de diversas áreas como análise de redes sociais, bioinformática e sistemas de recomendação. O problema compreende detectar componentes que apresentam alta densidade interna e baixa densidade externa. Devido ao rápido crescimento do volume de dados de diversas aplicações e à ampla utilização deste tipo de análise, diversas pesquisas em abordagens paralelas e distribuídas para resolver o problema de detecção de comunidades surgiram.

Alguns algoritmos possuem maior popularidade, resultando em uma extensa quantidade de pesquisa dentro de otimizações para processamento paralelo. Outros algoritmos, mesmo possuindo bons resultados de acurácia em testes, não apresentam o mesmo nível de profundidade de pesquisa em suas versões paralelas e distribuídas.

Esta pesquisa aborda a acurácia e escalabilidade de três algoritmos de detecção de comunidades. A partir dos experimentos realizados são propostas diretrizes para a utilização de cada algoritmo de acordo com as necessidades do usuário. Além disso, é explorado o comportamento das abordagens paralelas e possíveis melhorias são propostas.

Palavras-Chave: Detecção de comunidades, Computação paralela, Teoria dos grafos.

AN ANALYSIS ON THE ACCURACY AND SCALABILITY OF PARALLEL COMMUNITY DETECTION ALGORITHMS IN GRAPHS

ABSTRACT

Community detection is a type of topological analysis widely used in graph analysis in several fields such as social network analysis, bioinformatics and recommendation systems. The problem involves detecting components that have high internal density and low external density. Due to the rapid growth in the volume of data from a variety of applications and the wide use of this type of analysis, several researches in parallel and distributed approaches to solve the problem of community detection have emerged.

Some algorithms are more popular, resulting in an extensive amount of research on optimizations for parallel processing. Other algorithms, which posses better accuracy results in tests, do not present the same level of research depth in their parallel and distributed versions.

This research addresses the accuracy and scalability of three community detection algorithms. User guidelines are proposed based on the experiments results. In addition, the behavior of the parallel approaches is explored and possible improvements are proposed.

Keywords: Community detection, Parallel computing, Graph theory.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

Com o crescimento do volume de dados produzidos mundialmente [36], métodos para analisar e armazenar estes dados se tornaram cada vez mais importantes. Uma das estruturas de dados mais utilizadas para tais funções é o grafo. Esta estrutura é amplamente utilizada em diversas aplicações que necessitam do armazenamento de informações de um elemento assim como informações das relações deste elemento com outros, como redes sociais, bioinformática e redes de computadores [46, 33, 62, 43]. Embora o grafo apresente diversas vantagens em sua utilização, ele também apresenta desafios para serem analisados.

Atualmente, os grafos de diversas aplicações apresentam milhões de nodos e bilhões de arestas, o que torna o processamento dessas estruturas muito custoso. Além disso, alguns destes grafos (como por exemplo os de redes sociais) estão em constante mudança, portanto, algumas análises somente são relevantes se são obtidas dentro de um certo período de tempo. Em função disso, algoritmos de processamento paralelo para grafos têm ganhado muita atenção nos últimos anos.

Existem dois paradigmas para abordagens paralelas, processamento em memória compartilhada e em memória distribuída. Estes paradigmas apresentam vantagens e limitações distintas. Memória compartilhada possui uma implementação mais simples e apresenta ganhos facilmente, mas sua escalabilidade está atrelada a máquina disponível. Memória distribuída, embora possua mais complicações em sua implementação e não funcione para qualquer problema, pode ser escalada facilmente.

Existem diversos tipos de análises topológicas que podem ser realizadas sobre um grafo, desde propriedades de seus nodos e arestas até propriedades estruturais do grafo. Uma propriedade estrutural que está sendo pesquisada é a detecção de comunidades, que são definidas por Fortunato e Hric [10] como subconjuntos altamente densos de um grafo. De acordo com Fortunato e Hric [10], diversos métodos foram criados para este fim, com diferentes abordagens e necessidades. Tais métodos, assim como diversos algoritmos de processamento de grafos, apresentam uma necessidade de abordagens paralelas, já que muitos métodos apresentam um tempo de processamento que inviabiliza sua utilização sequencial em grafos atuais.

Algoritmos de processamento de grafos tendem a apresentar melhores performances em ambientes de memória compartilhada devido a sua estrutura [34]. Abordagens em ambientes de memória distribuída possuem um grande custo de comunicação em função da distribuição de vértices. Este alto custo de comunicação resulta em uma baixa eficiência na utilização de recursos por aplicações. Tendo em vista estes comportamentos, optou-se por explorar o comportamento de algoritmos de detecção de comunidades em ambientes de

memória compartilhada, uma vez que se estes ambientes não apresentarem bons ganhos de performance, ambientes distribuídos dificilmente serão utilizáveis.

Este trabalho explora três algoritmos de detecção de comunidades (Louvain, Label Propagation e Infomap) e algumas de suas abordagens paralelas em ambientes de memória compartilhada, tendo como principal objetivo averiguar seus desempenhos em ambientes atuais. Para este fim foram realizados experimentos de acurácia em grafos sintéticos e experimentos de escalabilidade em grafos reais.

A partir dos experimentos realizados foi constatado que o algoritmo Infomap apresenta a melhor acurácia, mas suas versões sequencial e paralela apresentam os maiores tempos de execução dentre os algoritmos abordados. O algoritmo Label Propagation é o mais rápido mas possui uma pior acurácia em grande parte dos casos abordados. O algoritmo Louvain apresenta-se entre os dois outros algoritmos tanto em tempo de execução quanto em acurácia. Com estes resultados são demonstrados os casos onde cada um desses algoritmos pode ser aplicado e quais são as possíveis melhorias que suas versões paralelas podem explorar para atingirem menores tempos de execução.

Nos próximos capítulos serão apresentados os conceitos necessários para o entendimento deste volume (Capítulo 2), os algoritmos abordados e suas respectivas implementações paralelas (Capítulo 3), os experimentos realizados (Capítulo 4), uma discussão em relação aos resultados desta pesquisa (Capítulo 5) e, finalmente, as conclusões obtidas a partir desta pesquisa (Capítulo 6).

2. REFERENCIAL TEÓRICO

Este capítulo apresenta as definições necessárias para o entendimento do restantes deste volume. Inicialmente, serão estabelecidos os conceitos de Paralelismo e Comunidades em Grafos. Em sequência, é apresentada uma breve revisão da literatura acerca da criação de abordagens paralelas e distribuídas para algoritmos de detecção de comunidades em grafos.

2.1 Paralelismo

Arquiteturas de computadores tradicionais apresentam apenas um processador. Arquiteturas paralelas utilizam de diversos processadores. Tanto arquiteturas tradicionais quanto arquiteturas paralelas já adotam placas *manycore* como GP-GPUs para acelerar o processamento de tarefas específicas com um custo energético menor. Estas arquiteturas, no entanto, apresentam um grande problema de memória, uma vez que a quantidade de memória que pode ser colocada em uma única máquina sem que ocorram problemas de I/O é muito limitada levando em consideração a quantidade de memória necessária para executar os problemas computacionais dos dias de hoje.

Para contornar o problema de memória apresentado por arquiteturas paralelas se optou pela criação de máquinas compostas por diversas outras máquinas, comumente chamadas de *clusters*. Um *cluster* de computadores é composto por diversas máquinas (nós) que apresentam arquiteturas iguais e se comunicam através de uma rede. Desta forma a memória disponível para utilização é a soma de todas as máquinas, uma vez que o problema será dividido entre elas. No entanto, este tipo de abordagem possui um grande problema, a necessidade de realizar trocas de mensagens entre os nós para passar informações relevantes para a solução do problema sendo processado. Em função disso, apenas dois tipos de problemas computacionais são recomendados para o paradigma de processamento distribuído: problemas que apresentam uma grande necessidade computacional por máquina, compensando o custo da troca de mensagens; e problemas que apresentam um custo de memória muito alto, impossibilitando que sejam processados em apenas uma máquina paralela.

2.1.1 Métricas de Desempenho de Programas Paralelos

Para realizar a avaliação da performance de programas paralelos são utilizadas, usualmente, duas métricas, Eficiência e Speed-Up. A métrica de Speed-Up diz respeito ao

ganho de velocidade no tempo de execução da implementação paralela relativo ao tempo de execução da implementação sequencial. Esta métrica é expressa da seguinte forma:

$$SpeedUp = \frac{T_s}{T_p} \tag{2.1}$$

onde T_s é o tempo de execução da implementação sequencial e T_p é o tempo de execução da implementação paralela. Esta métrica costuma ser calculada para os diferentes números de *threads* possíveis na máquina que está sendo utilizada para que seja possível identificar até quando se tem um ganho de velocidade por utilização de recursos. Toda implementação paralela possui um limite de ganho de velocidade e, ao chegar neste limite, qualquer tentativa de aumentar o número de *threads* sendo utilizada não irá apresentar ganhos e, em muitos casos, poderá apresentar resultados piores que os atingidos por execução com menos *threads*.

A métrica de eficiência diz respeito a eficiência do consumo de recursos de uma máquina. Esta métrica é calculada da seguinte forma:

$$Eficiencia = \frac{SpeedUp_n}{N} \tag{2.2}$$

onde $SpeedUp_n$ é o Speed-Up atingido ao utilizar n *threads* e N é o número de *threads* utilizado. Idealmente, o Speed-Up atingido por usar N *threads* deve ser N, pois assim os recursos disponíveis estão sendo utilizados totalmente. No entanto, usualmente, implementações paralelas contém sessões críticas e custos de acesso a certas informações, o que diminui o tempo em que cada *thread* está ativamente processando o problema em questão, o que influencia no Speed-Up atingido e, portanto, resulta em uma perda de Eficiência.

2.2 Comunidades em Grafos

Um grafo é uma estrutura que consiste em um conjunto de vértices e arestas. Vértices são pontos únicos no grafo, podendo ou não conter informações. Arestas são ligações entre dois nodos. Desta forma podemos definir um grafo G como $G(V, E)$, onde V é o conjunto de vértices dentro do grafo e E é o conjunto de arestas que ligam estes vértices.

Grafos e seus elementos possuem diversas propriedades. A propriedade que define as estruturas de comunidades é o grau de um vértice, que é definido como o número de arestas conectadas ao vértice. Além disso é necessário o conceito de grau interno e externo. O grau interno de um vértice corresponde ao número de arestas que o conectam aos

outros vértices de sua comunidade, já o grau externo diz respeito ao número de conexões que um vértice tem com elementos fora de sua comunidade.

Uma estrutura de comunidade pode ser definida de forma genérica como um módulo denso de um grafo, onde existem mais arestas internas, que conectam vértices da comunidade com outros vértices da comunidade, do que externas, que conectam vértices da comunidade com vértices fora da comunidade. Fortunato e Hric [10] apresentam duas definições de comunidade, uma clássica e outra moderna. Ambas definições dizem respeito a uma comunidade C que é um subgrafo $C(V_c, E_c)$ de um grafo $G(V, E)$, onde $V_c \subset V$ e $E_c \subset E$.

Na visão clássica, C apresenta uma estrutura fraca de comunidades se a média do grau interno dos vértices é maior do que a médio do grau externo. Se todos os vértices de C apresentam um grau interno maior do que seu grau externo, então C apresenta uma estrutura forte de comunidade. A Figura 2.1 apresenta um exemplo de estrutura de comunidade.

Na visão moderna, C apresenta uma estrutura fraca de comunidade se a média da probabilidade de cada vértice se conectar com outro vértice de C é maior do que a média da probabilidade de cada vértice se conectar com vértices externos. C é considerado uma comunidade forte se todos os seus vértices possuem uma probabilidade maior de estarem conectados entre si do que com vértices externos.

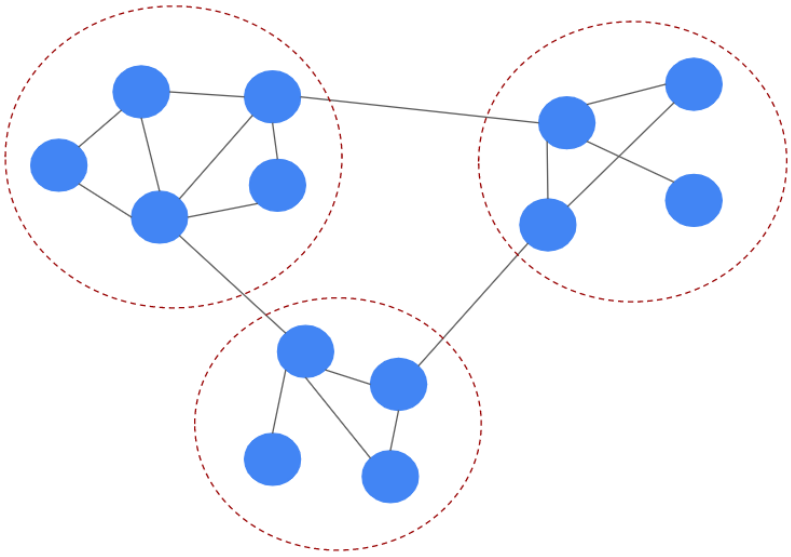


Figura 2.1: Exemplo de comunidades em um grafo. Cada comunidade está destacada por um círculo tracejado.

Métodos de detecção de comunidades podem ser divididos em quatro categorias, métodos espectrais, métodos de inferência estatística, métodos de otimização e métodos de simulações de dinâmica.

Métodos espectrais utilizam *eigenvectors* para construir uma projeção dos vértices do grafo. Os pontos de tal projeção são agrupados em comunidades através de outros métodos de detecção.

Métodos de inferência estatística propõe modelos generativos para os grafos. O método mais popular dentro desta categoria é *Stochastic Block Model*. Estes métodos não apresentam muitas implementações paralelas.

Métodos de otimização procuram maximizar uma função de otimização de comunidades. O método mais comum desta categoria é a otimização da função de modularidade, criada por Newman e Girvan [37]. Este método procura maximizar a modularidade de todas as possíveis comunidades de um grafo. Blondel [4] criou um dos algoritmos mais populares para maximizar a função de modularidade, chamado de Louvain.

Métodos de simulação utilizam simulações sobre a estrutura do grafo para descobrir comunidades. Estes métodos abrangem diversos tipos de simulação, como difusão, caminhamento, sincronização, etc. Uma das simulações mais comuns é o de um caminho pelo grafo, a partir da utilização de um *Random Walker*. Este tipo de método parte do pressuposto de que um processo que caminha sobre uma comunidade tende a se manter nela, já que uma comunidade é densamente conectada, e com isso é possível descobrir quais vértices pertencem a uma determinada comunidade a partir da probabilidade de um *Random Walker* andar por determinado vértice.

2.2.1 Validação de Algoritmos de Detecção de Comunidades

Existem diversas métricas que são usadas na literatura para avaliar a qualidade das soluções obtidas através de algoritmos de detecção de comunidades. Alguns autores optam por utilizar funções de qualidade, como a equação da modularidade, para avaliar soluções [35]. No entanto, a grande maioria dos estudos desta área opta por utilizar *benchmarks* de grafos sintéticos e/ou reais para realizar a avaliação das soluções.

No caso de *benchmarks* de grafos sintéticos, os grafos são gerados a partir de um modelo de geração. Um *benchmark* muito utilizado é o proposto por Lancichinetti et al. [25]. *Benchmarks* de grafos reais possuem suas estruturas de comunidades definidas por metadata, e não por cálculos topológicos.

Existem diferentes métodos para comparar o *ground truth* das comunidades dos *benchmarks* com as comunidades geradas por algoritmos. Uma das métricas comumente utilizadas é o *Normalized Mutual Information* (NMI), que verifica se é possível extrair a mesma quantidade de informação de duas variáveis. Outra métrica, proposta pelo Graph Challenge [22], é *pairwise Precision-Recall* dos vértices.

2.3 Paralelismo em Detecção de Comunidades

Devido a estrutura heterogênea dos grafos de aplicações reais, algoritmos de processamento de grafos apresentam diversos desafios para a implementação de paralelismo. Independente do paradigma paralelo utilizado é necessário se preocupar com os métodos de atualização das informações do grafo e, em alguns casos, a ordem de processamento dos vértices e arestas. Ao implementar abordagens distribuídas, ainda deve ser levado em consideração o alto custo de trocas de mensagens pela aplicação. Como os vértices e as arestas do grafo estão distribuídos em diversos nós computacionais pode ser necessária uma troca constante de informações pela rede ou fases de sincronização dos nós. Em ambos os casos o custo da troca de mensagens pode se tornar superior ao resto do processamento.

O problema de detectar comunidades em grafos pode se tornar muito custoso dependendo do tipo de algoritmo sendo utilizado. Além disso, o crescimento de volumes de dados torna cada vez maiores os grafos que devem ser processados. Em função desses pontos, algumas técnicas de detecção de comunidades se tornam inviáveis de serem executadas sequencialmente.

A utilização de abordagens paralelas e distribuídas se tornou essencial para a detecção de comunidades em grafos atuais. A utilização de paralelismo permite a execução de grandes grafos em intervalos de tempo aceitáveis. Além disso, no caso de grafos que não podem ser comportados em apenas uma máquina, sistemas distribuídos permitem seu processamento.

Diversas implementações paralelas e distribuídas de algoritmos de detecção de comunidades foram criadas ao longo dos anos. Tais implementações utilizam de diversas técnicas para atingir bons tempos de execução, alterando o comportamento dos algoritmos sequenciais quando necessário. Algumas implementações podem ser vistas na Tabela 2.1. É possível verificar que soluções paralelas em ambientes de memória compartilhada e distribuída estão sendo produzidas para diversos algoritmos, principalmente para os algoritmos de Louvain e Label Propagation.

Alguns trabalhos são propostos dentro do contexto de uma ferramenta de análise. Ghosh et al. [11] apresentam uma ferramenta que possui a implementação distribuída do algoritmo paralelo de Louvain produzida pelo mesmo grupo [12]. Slota et al. [49] apresenta uma ferramenta de particionamento de grafos baseada em uma implementação distribuída de Label Propagation. Bader e Madduri [1] apresentam uma ferramenta para particionamento e análise de grafos com três algoritmos de detecção de comunidades baseados em otimização de modularidade no paradigma de memória compartilhada. Staoudt e Meyerhenke [55] propõe implementações paralelas dos algoritmos Louvain e Label Propagation para uma biblioteca de análise de grafos.

Tabela 2.1: Artigos com abordagens paralelas e distribuídas para algoritmos de detecção de comunidades

Artigo	**Algoritmo**	**Paradigma**
Lu et al. [31]	Louvain	Memória Compartilhada
Chavarria-Miranda et al. [6]	Louvain	Memória Compartilhada
Halappanavar et al. [15]	Louvain	Memória Compartilhada
Bhowmik e Vadhiyar [3]	Louvain	Memória Compartilhada
Ghosh et al. [12]	Louvain	Memória Distribuída
Zeng e Yu [63]	Louvain	Memória Distribuída
Sattar e Arifuzzaman [45]	Louvain	Memória Distribuída
Wickramaarachchi [58]	Louvain	Memória Distribuída
Zeng e Yu [64]	Louvain	Memória Distribuída
Staoudt e Meyerhenke [55]	Louvain, Label Propagation	Memória Compartilhada
Khlopotine et al. [23]	Label Propagation	Memória Compartilhada
Soman e Narang [51]	Label Propagation	Memória Compartilhada
Ovelgonne [38]	Label Propagation	Memória Distribuída
Liu et al. [29]	Label Propagation	Memória Distribuída
Sedighpour e Bagheri [47]	Label Propagation	Memória Distribuída
Song et al [52]	Label Propagation	Memória Distribuída
Slota et al. [49]	Label Propagation	Memória Distribuída
He et al. [17]	Infomap, Label Propagation	Memória Compartilhada
Bae et al. [2]	Infomap	Memória Compartilhada
Faysal e Arifuzzaman [9]	Infomap	Memória Distribuída
Fathi et al. [8]	Random Walk	Memória Distribuída
Panyala et al. [39]	Otimização de Modularidade	Memória Compartilhada
Bader e Madduri [1]	Otimização de Modularidade	Memória Compartilhada
Bóta et al. [5]	K-means	Memória Compartilhada
Gregori et al. [14]	K-Clique	Memória Compartilhada
Lu et al. [32]	Maximal Clique	Memória Distribuída
Xu et al. [61]	Maximal Clique	Memória Distribuída
Sun et al. [56]	Swarm Intelligence	Memória Compartilhada
Souravlas et al. [54]	Path Analysis	Memória Compartilhada
Souravlas et al. [53]	Path Analysis	Memória Compartilhada
Zhou et al. [65]	Similaridade de Vértices	Memória Compartilhada
Qiao et al. [40]	Otimização	Memória Distribuída
Jain et al. [19]	CEIL	Memória Distribuída
Resende et al. [41]	Simulação de Partículas	Memória Compartilhada
Joldos e Technical [21]	Evolução	Memória Compartilhada
He et al. [16]	Simulação de Distância	Memória Compartilhada
Wu [59]	Simulação de Distância	Memória Compartilhada
Vo et al. [57]	Simulação de Distância	Memória Distribuída
Ghoshal et al. [13]	Algoritmo Genético	Memória Compartilhada
Weidong [7]	Difusão de Informação	Memória Distribuída
Soliman et al. [50]	Simulação de Difusão	Memória Distribuída
Liu e Wei [28]	GDS	Memória Distribuída
Liang et al [27]	Evolução	Memória Distribuída
Xu et al. [60]	Stochastic Block Model	Memória Distribuída
Sarswat e Guddeti [44]	CFM, Nash Equilibrium	Memória Compartilhada

Outros trabalhos focam na criação de abordagens paralelas em diferentes ambientes. Khlopotine et al. [23] e Bhowmik e Vadhiyar [3] apresentam implementações em memória compartilhada do algoritmo de Louvain. No entanto, Bhowmik e Vadhiyar [3] apresentam uma abordagem utilizando CPU e GPU, enquanto Khlopotine et al. [23] e Staoudt e Meyerhenke [55] apresentam uma implementação apenas em CPU. Bae et al. [2] apresenta a implementação paralela do algoritmo Infomap que é considerada o estado da arte para paradigmas de memória compartilhada.

2.4 Trabalhos Relacionados

Diversos pesquisadores já construíram diferentes tipos de análises de algoritmos de detecção de comunidades. A Tabela 2.2 apresenta alguns destes trabalhos, demonstrando se o estudo em questão aborda algoritmos sequenciais ou paralelos e se são realizados experimentos sobre os algoritmos.

Fortunato e Hric [10] apresentam um guia a respeito do conteúdo, onde apresentam diversas definições de estruturas de comunidades e algoritmos sequenciais para resolver o problema. O foco do trabalho construído por eles está no esclarecimento da área de detecção de comunidades, de forma a demonstrar as diversas formas de detectar comunidades e quais são suas vantagens. São realizados experimentos de acurácia com um conjunto de algoritmos.

Lancichinetti e Fortunato [24] apresentam um estudo de qualidade de solução de algoritmos sequenciais de detecção de comunidades. Os autores têm como foco determinar quais algoritmos possuem uma melhor acurácia. Dentro deste estudo é constatado que, para diversas métricas, os algoritmos Louvain e Infomap apresentam os melhores resultados.

Mothe et al. [35] também apresenta um estudo de acurácia de alguns algoritmos sequenciais de detecção de comunidades. No entanto, os autores utilizam apenas a métrica de modularidade para determinar os melhores algoritmos dentro de um conjunto de grafos sintéticos gerados a partir de um *Stochastic Block Model*. Este estudo conclui que, dentre 6 algoritmos, o Louvain apresenta a melhor qualidade.

Jin et al. [20] apresentam um estudo de algoritmos sequenciais de detecção de comunidades. Os algoritmos são divididos em diversas categorias e explorados dentro de suas teorias. O trabalho resulta em um compilado de diversos algoritmos, categorizados com informações referentes a suas abordagens, focos e limitações.

Naik et al. [36] explora algoritmos paralelos de detecção de comunidades. Os autores realizam uma revisão da literatura para averiguar quais técnicas estão sendo utilizadas para paralelizar a solução do problema. No entanto, não são realizados experimentos, o

estudo se limita a apenas compilar trabalhos que desenvolvem algoritmos de detecção de comunidades utilizados dentro do campo de análise de redes sociais.

Embora todos os trabalhos mencionados apresentem, em diferentes níveis, comparações entre algoritmos de detecção de comunidades, nenhum deles possui análises entre algoritmos sequenciais e suas versões paralelas. Além disso, a escalabilidade de algoritmos paralelos não é explorada pelos estudos que os analisam. Entender o funcionamento dos algoritmos paralelos e como eles diferem de suas versões sequenciais é crucial para que seja possível produzir implementações que atinjam os melhores resultados de acurácia e escalabilidade. Além disso, verificar o comportamento de técnicas utilizadas em um tipo de algoritmo permite iniciar estudos para utiliza-las em outros algoritmos.

Tabela 2.2: Estudos comparativos de algoritmos de detecção de comunidades

Estudo	**Tipo dos Algoritmos**	**Experimentos**
Fortunato e Hric [10]	Sequencial	Sim
Lancichinetti e Fortunato [24]	Sequencial	Sim
Mothe et al. [35]	Sequencial	Sim
Jin et al. [20]	Sequencial	Não
Naik et al. [36]	Paralelo	Não

3. ALGORITMOS ABORDADOS

Existe uma ampla quantidade de pesquisa acerca de implementações paralelas e distribuídas de algoritmos de detecção de comunidades, embora um grande número dessas pesquisas se concentrem nos mesmos algoritmos. Como algoritmos de processamento de grafos tendem a apresentar uma melhor eficiência em ambientes de memória compartilhada [34], espera-se que as abordagens paralelas de algoritmos de detecção de comunidades apresentem um grande nível de refino, já que diversos pesquisadores criaram abordagens paralelas e distribuídas. Em função desses pontos optou-se por explorar abordagens paralelas de algoritmos de detecção de comunidades, uma vez que se as mesmas não apresentarem bons resultados suas alternativas distribuídas provavelmente sofrerão de problemas similares somados aos altos custos de comunicação.

Foram selecionados três algoritmos para serem analisados em termos de acurácia e escalabilidade em ambientes paralelos de memória compartilhada. O primeiro algoritmo selecionado foi o de Louvain, já que ele apresenta um grande volume de pesquisa em soluções paralelas, assim como códigos sequenciais e paralelos disponíveis para utilização. O segundo algoritmo selecionado foi Label Propagation pois este também apresenta diversos trabalhos em versões paralelas e um amplo uso. O terceiro algoritmo selecionado foi o Infomap, que, embora não apresente muita pesquisa em ambientes paralelos, é considerado um dos melhores algoritmos em termos de acurácia [10].

Este capítulo entra em detalhes do funcionamento de cada um dos algoritmos abordados e suas respectivas versões paralelas.

3.1 Louvain

O algoritmo de Louvain foi proposto por Blondel et al. [4]. Para determinar a qualidade de uma comunidade é utilizada a equação da modularidade proposta por Newman e Girvan [37], que define a qualidade da divisão de um grafo em módulos. A equação da modularidade é definida como

$$Q = \frac{1}{2m} \sum_{ij} (A_{ij} - P_{ij}) \delta(C_i, C_j) \tag{3.1}$$

onde m é o número de arestas do grafo, o somatório percorre todos os pares de vértices i e j, $A_{i,j}$ é o elemento da matriz de adjacências, $P_{i,j}$ representa a matriz de adjacência média de um conjunto de grafos randomizados a partir do grafo original e C_i e C_j representam as comunidades de i e j, respectivamente. A eficácia desta equação se baseia na ideia de que um grafo randomizado a partir dos vértices do grafo original

não irá manter a estrutura de comunidades do mesmo, portanto, estruturas que existem no grafo original mas não existem nos grafos randomizados possuem uma chance mais alta de representarem comunidades.

O algoritmo apresenta uma abordagem aglomerativa, onde é executado diversas vezes em cima de suas soluções prévias. A Figura 3.1 demonstra os passos executados pelo algoritmo. Inicialmente, cada vértice do grafo é atribuído à sua própria comunidade. Em seguida, para cada vértice, é calculado se, ao mover-se para uma comunidade vizinha, ocorre um ganho de modularidade. Dentre todos os possíveis movimentos que resultam em um ganho de modularidade é executado aquele que apresenta o maior ganho. Este passo é repetido até que não seja possível executar um movimento que resulte em um ganho de modularidade. Com isso encerra-se a primeira fase do algoritmo e o primeiro nível da solução.

Após ter concluído o primeiro nível da solução, são criados super-vértices a partir das comunidades presentes no primeiro nível. As arestas de todos os vértices de cada comunidade são unidas para compor as arestas do novo super-vértice, criando arestas entre super-vértices para arestas que conectavam a comunidade em questão com outras comunidades. Com isto é criado um novo grafo, onde os super-vértices possuem suas próprias comunidades, assim como no início do algoritmo.

O algoritmo, então, é executado sobre este novo grafo. Em uma ordem aleatória os super-vértices tentarão mover-se para a comunidade vizinha que resulta no maior ganho de modularidade. Quando não houverem mais movimentos que possam ser feitos no grafo dá-se por completa a segunda fase da algoritmo e o segundo nível da solução.

O algoritmo segue estes passos até que, para um grafo gerado a partir da estrutura de comunidades de um nível da solução, não seja possível realizar nenhum movimento desde o início da execução da fase, ou seja, atingiu-se um nível de solução que é exatamente igual ao nível anterior. Com isso o algoritmo termina, apresentando um resultado hierárquico da distribuição de vértices em comunidades.

3.1.1 Grappolo

Para paralelizar o algoritmo de Louvain, Lu et al. [30] criaram a ferramenta Grappolo. Esta ferramenta utiliza diversas abordagens com o intuito de aumentar a performance da implementação sem sacrificar grandes partes da qualidade da solução. Os tipos de otimização utilizados impactam, principalmente, a ordem com que vértices são considerados para movimentos, assim como os movimentos em si. A Figura 3.2 mostra os passos realizados pelo algoritmo, onde a Figura 3.2a demonstra a bordagem base e a Figura 3.2b demonstra a abordagem com pré-processamento. Todos os estados brancos separados por reticências representam execuções simultâneas por múltiplas *threads*.

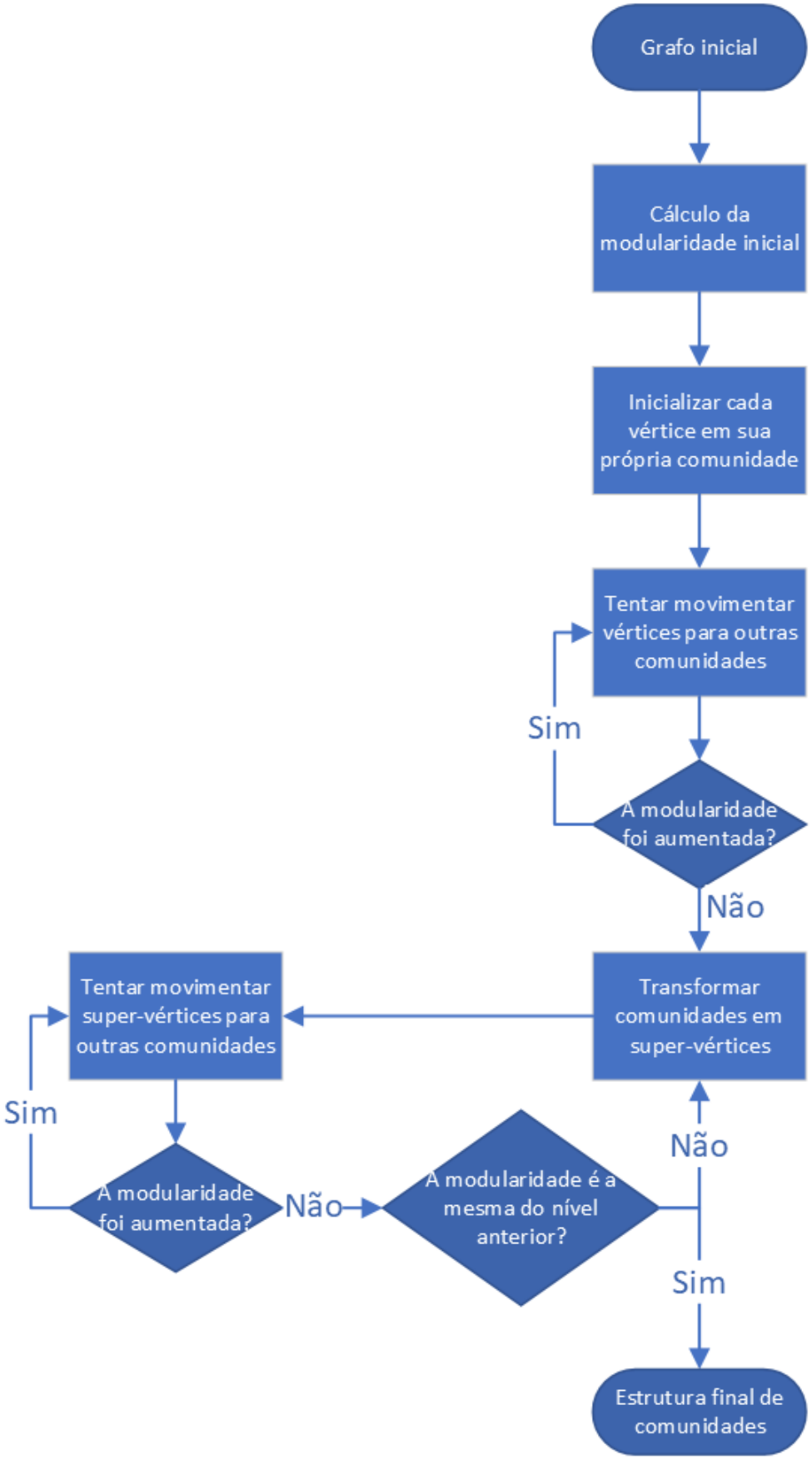


Figura 3.1: Flowchart do algoritmo Louvain.

O processamento de vértices do grafo é realizado em paralelo, de forma que cada *thread* calcule os movimentos ótimos de vértices. Como a execução de um movimento altera os valores de modularidade do grafo, pode-se cogitar que permitir que movimentos ocorram de forma assíncrona possa resultar em movimentos circulares e/ou sub-ótimos. No entanto, como demonstrado por Shi et al. [48], a execução de movimentos através de *atomics* apresenta melhores resultados em termos de performance do que a utilização de uma seção crítica ou *lock*. Lu et al. [30] utiliza de cláusulas *atomic* para a realização de movimentos por *threads*.

Lu et al. [30] também propõe um estágio de pré-processamento do grafo com o objetivo de diminuir o número de computações necessárias para chegar em uma solução. Este passo consiste em colorir o grafo com uma distância $d - 1$, de forma que nenhum vértice possua a mesma cor que algum de seus vizinhos. As *threads*, então, processam uma cor de vértices por vez, reduzindo a chance de movimentos sub-ótimos ocorrerem.

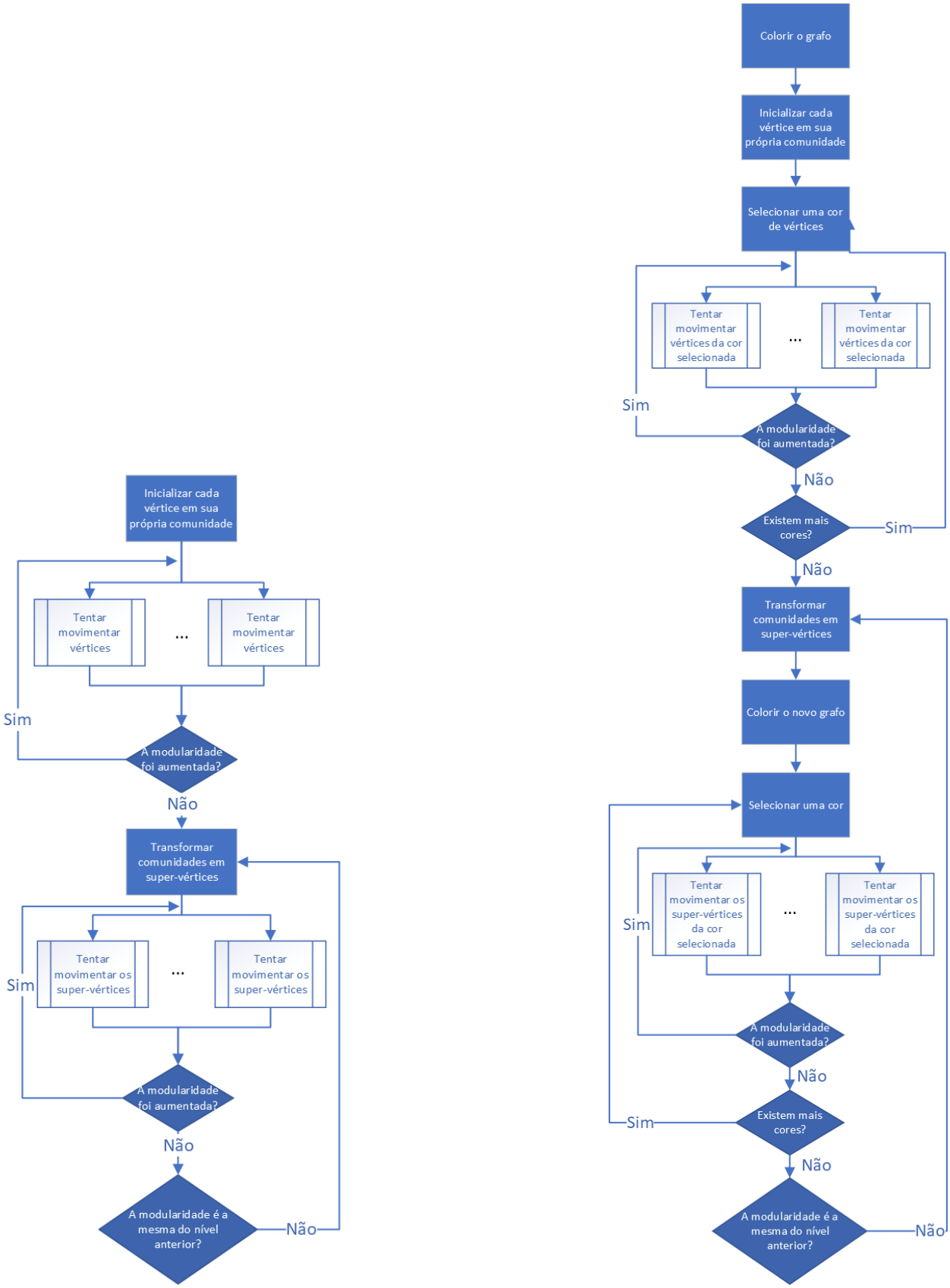
3.2 Label Propagation

O algoritmo de Label Propagation consiste em propagar comunidades pelo grafo de forma que a admissão às comunidade é feita por similaridade com vértices vizinhos. Inicialmente, alguns vértices do grafo, de forma aleatória, são atribuídos à suas próprias comunidades. Em seguida, estes vértices irão propagar suas comunidades para seus vizinhos, que irão assumir a comunidade da maioria de seus vizinhos. No caso de um empate entre duas comunidades, o critério de decisão torna-se decisão do autor. No caso da implementação por Staoudt e Meyerhenke [55], é utilizado o critério de modularidade das comunidades em questão, sendo que o vértice se junta a comunidade que resulte em um maior ganho da função de modularidade.

3.3 Infomap

Rosvall et al. [42] propuseram o problema de descrever, da menor forma possível, um caminho por um grafo. Os autores exploram a utilização de codificações dos vértices do grafo e eventualmente mapeiam o problema para detecção de comunidades, já que, como apresentado pelos autores, é possível diminuir o tamanho dos códigos utilizados pelos vértices a partir de suas comunidades.

Rosvall et al. [42] propuseram a métrica de *Minimum Description Lenght* (MDL), que quantifica a menor descrição possível de um caminho pelo grafo. A forma mais simples de descrever um caminho é apenas enumerar todos os vértices visitados, no entanto, este método gera uma descrição grande. Os autores, então, propõem reutilizar identifica-



(a) Grappolo base.

(b) Grappolo com coloração do grafo.

Figura 3.2: Flowcharts das abordagem utilizadas pela ferramenta Grappolo.

dores de acordo com a formação de comunidades. Dois vértices podem possuir o mesmo identificador desde que pertençam à comunidades diferentes.

Para codificar vértices e comunidades é utilizado o código de Huffman [18], onde o tamanho dos códigos que representam vértices e comunidades é baseado na frequência com que um *random walk* infinito visita estes vértices e comunidades. Os códigos para cada comunidade são guardados em um livro global, e os códigos dos vértices de cada comunidade são guardados dentro de livros próprios de cada comunidade. Ao descrever um caminho pelo grafo é possível utilizar o identificador da comunidade apenas uma vez para informar que todos os vértices após este identificador devem ser decodificados a partir do livro de códigos da comunidade em questão, e, quando houver o identificador de que o caminho irá trocar de comunidade, deve-se utilizar o livro global para saber qual livro deve ser usado para decodificar a próxima sequência de vértices.

Para descobrir a estrutura de comunidades de um grafo Rosvall et al. [42] criaram a *map equation*, que é definida como

$$L(M) = q_{\curvearrowright} H(Q) + \sum_{i=1}^{m} p^i_{\circlearrowright} H(P^i) \tag{3.2}$$

onde $L(M)$ é o menor código atingível (MDL) por um módulo M de vértices do grafo. $H(Q)$ e $H(P^i)$ são os tamanhos médios de códigos dentro do livro de comunidades e do livro da comunidade i. $q_{\curvearrowright}$ é a frequência com que o livro de comunidades é utilizado, ou seja, a frequência com que o *random walker* troca de comunidade. $p^i_{\circlearrowright}$ é a frequência com que o livro da comunidade i é utilizado, ou seja, a frequência com que o *random walker* se mantém em uma mesma comunidade.

3.3.1 Algoritmo

O algoritmo Infomap pode ser dividido em duas etapas, uma de detecção e outra de refinamento. A Figura 3.3 apresenta o *Flowchart* do algoritmo.

A primeira etapa do algoritmo possui uma execução semelhante ao algoritmo de Louvain. Inicialmente todos os vértices são adicionados a suas próprias comunidades e, em sequência, vértices tentarão, em ordem aleatória, se mover para comunidades vizinhas, executando o movimento que resulta na maior redução no MDL do grafo. Estas tentativas de movimentos são realizadas até que não haja um movimento que reduza o MDL do grafo. Em seguida, todas as comunidades estabelecidas são transformadas em super-vértices e um novo grafo é criado tendo seus vértices como as comunidades do grafo anterior e suas arestas formadas pela junção das arestas pertencentes as comunidades. Neste novo grafo o algoritmo tenta realizar movimentos que reduzam o MDL. Quando não for possível realizar

novos movimentos o algoritmo irá transformar as comunidades atuais em super-vértices e continuará subindo de nível até que seja atingido um grafo em que nenhum movimento reduza o MDL.

Uma vez encerrada a etapa de detecção o algoritmo inicializa a etapa de refinamento, que é composta de duas partes que são executadas de forma intercalada. A primeira parte, nomeada de *Fine Tune*, envolve permitir que vértices no primeiro nível do grafo possam se mover novamente, tendo as informações adquiridas por todos os níveis. A segunda parte, nomeada de *Coarse Tune*, inicia no mais alto nível da atual solução e então divide as comunidades em sub-módulos, e então em sub-submódulos e assim por diante, até que os módulos possuam no mínimo dois vértices. Em seguida estes módulos podem se mover livremente entre comunidades desde que ocorra uma redução no MDL. Quando um *Fine Tune* ou um *Coarse Tune* além dos primeiros não gera uma redução no MDL o algoritmo encerra.

3.3.2 Relaxmap

O algoritmo Relaxmap foi proposto como uma versão paralela do algoritmo Infomap para ambientes de memória compartilhada. A abordagem dos autores é simples e não utiliza de otimizações propostas em trabalhos envolvendo outros algoritmos. A Figura 3.4 ilustra os passos realizados pelo algoritmo Relaxmap. Todos os estados brancos separados por reticências representam execuções simultâneas por múltiplas *threads*, enquanto todos os estados vermelhos representam uma seção crítica, ou seja, apenas uma *thread* pode executar tal passo por vez.

Para a etapa de detecção o método que computa e decide movimentos é realizado em paralelo, possuindo uma seção crítica para finalizar movimentos. O trabalho de iterar sobre os vértices do grafo é dividido entre as *threads* disponíveis, que irão procurar por movimentos ótimos. Sempre que uma *thread* encontra um movimento ótimo ela irá solicitar acesso a seção crítica e, uma vez dentro, irá conferir se o movimento ainda é válido (outras *threads* podem ter realizado movimentos que inviabilizam o movimento atual) e irá realizar o movimento.

A proposta de usar uma seção crítica surge da possibilidade de ocorrerem movimentos cíclicos no grafo, onde dois ou mais vértices trocam entre comunidades ao mesmo tempo. Isto faz com que o MDL resultante não seja o previsto pelas *threads* ao realizarem o movimento.

Para a etapa de refinamento é utilizada a mesma abordagem de seção crítica para execução dos movimentos. Além disso a divisão de comunidades em módulos da parte *Coarse Tune* é realizada em paralelo.

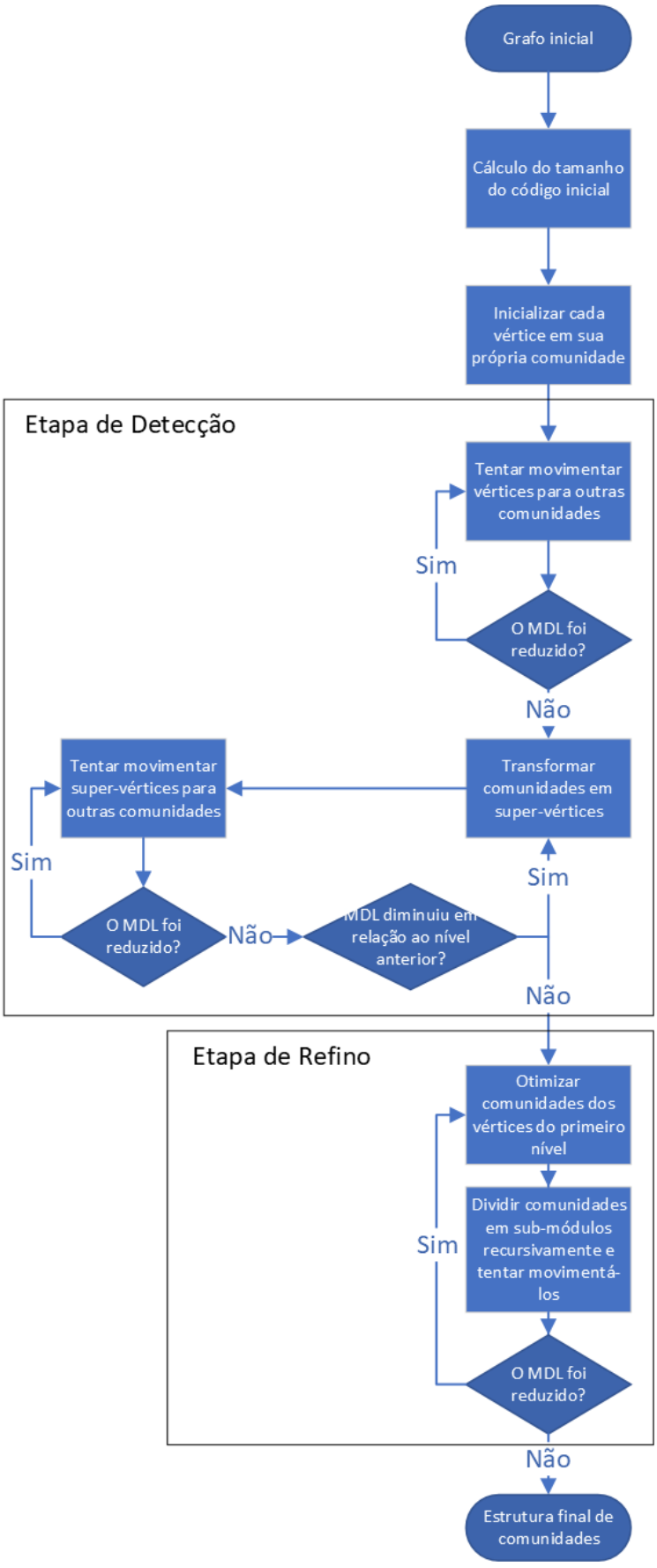
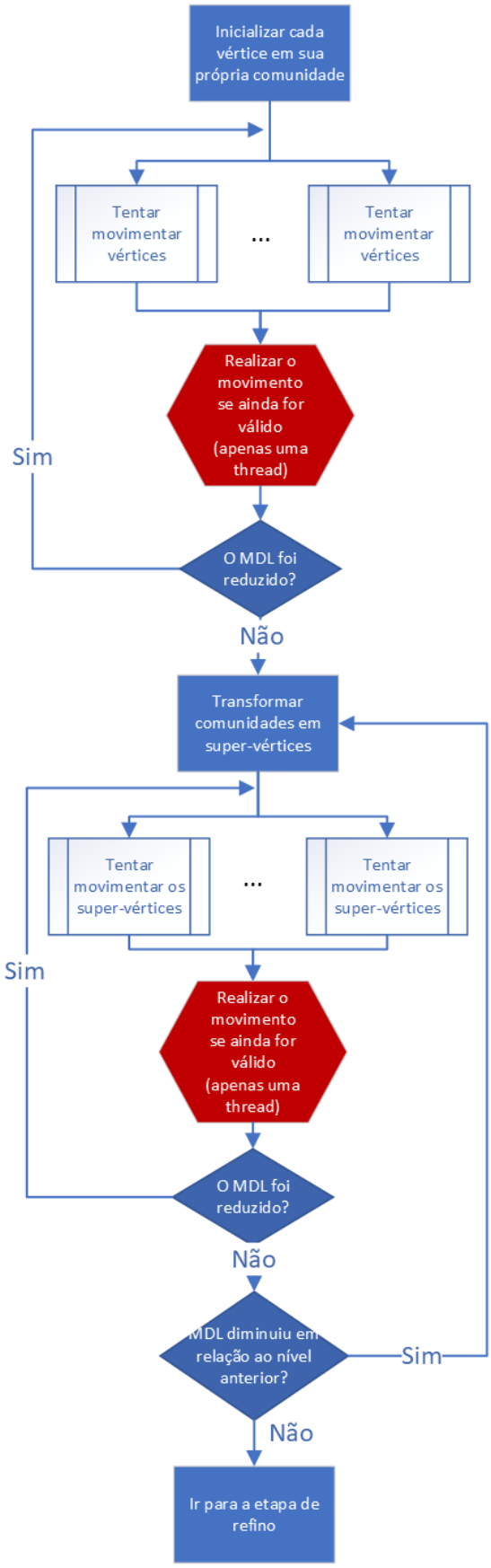
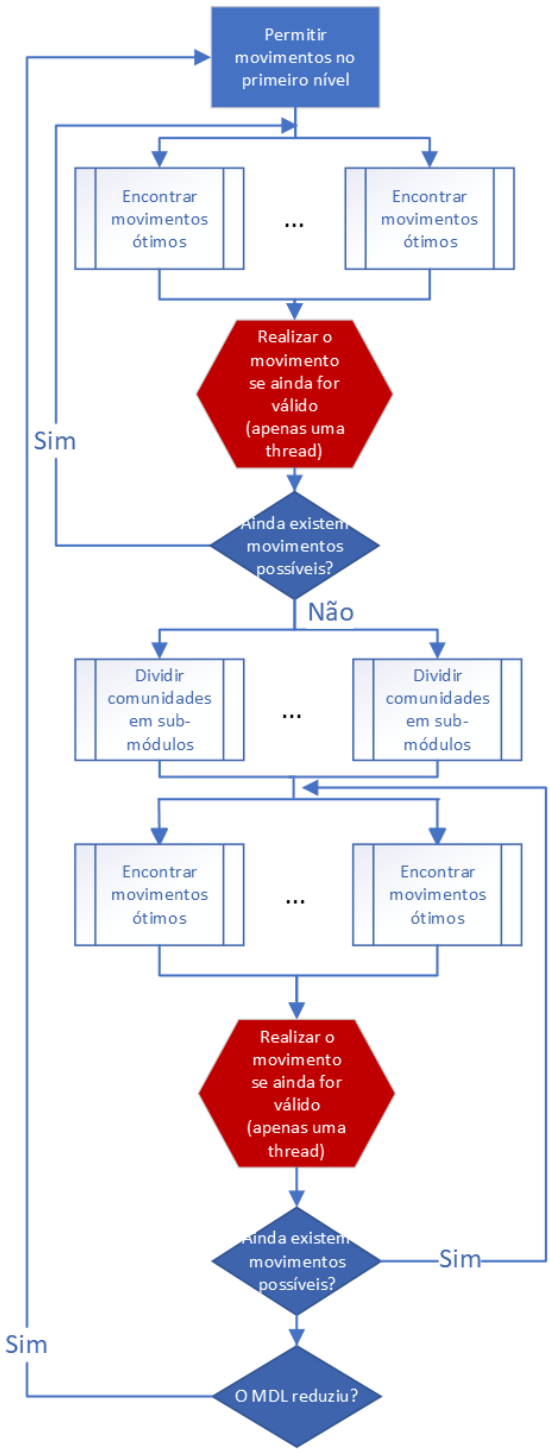


Figura 3.3: Flowchart do algoritmo Infomap.



(a) Etapa de Detecção



(b) Etapa de Refino

Figura 3.4: Flowcharts da abordagem utilizada pelo algoritmo Relaxmap

4. AVALIAÇÃO DA ACURÁCIA E ESCALABILIDADE DOS ALGORITMOS

Com o intuito de explorar a eficácia de abordagens paralelas de algoritmos de detecção de comunidades foram realizados experimentos sobre os algoritmos apresentados no Capítulo 3. Neste capítulo, tais experimentos e os grafos utilizados por eles serão demonstrados. Foram realizados dois tipos de experimentos. Primeiramente foi verificada a acurácia dos algoritmos sequenciais e suas versões paralelas. Em seguida foram realizados experimentos de performance.

Os experimentos de acurácia visam verificar a qualidade das soluções obtidas pelos algoritmos em grafos que apresentem diferentes densidades e distribuições de comunidades, de forma a descobrir se algum algoritmo apresenta uma piora de qualidade em função de algum atributo do grafo. Além disso, é necessário verificar se as abordagens paralelas propostas apresentam perdas de qualidade em troca de um aumento de performance.

Os experimentos de performance visam descobrir a eficiência das abordagens paralelas, ou seja, se os ganhos de tempo de execução são condizentes com os recursos utilizados pelas implementações. Além disso, espera-se descobrir o comportamento dos algoritmos para verificar se existem pontos que podem ser otimizados.

Os experimentos de acurácia foram realizados em grafos sintéticos gerados com o *benchmark* criado por Lancichinetti et al. [25]. Os experimentos de performance foram realizados sobre um conjunto de grafos reais obtidos no banco de dados do Stanford Network Analysis Project (SNAP) [26].

Todos os experimentos foram realizados em um Servidor Dell EMC PowerEdge R740 com duas soquetes, cada uma com um processador Xeon Gold 5118 (12 Cores de 2.30 GHz cada) com 12 Mbytes de cache L2 por core físico e 16.5 Mbytes de cache L3 por processador com 322 Gbytes de memória principal.

4.1 Grafos Utilizados

Os grafos utilizados nos experimentos podem ser divididos em dois grupos, grafos sintéticos e grafos reais. Todos os grafos sintéticos foram gerados a partir do *benchmark* criado por Lancichinetti et al. [25]. Tais grafos apresentam variações na disposição de suas estruturas, mas possuem sempre o mesmo número de vértices. Os grafos reais podem ser conferidos na Tabela 4.1. Tais grafos foram selecionados baseados em seus tamanhos, assim como a necessidade de apresentarem arestas bidirecionais.

4.1.1 Grafos Sintéticos

Ao construir o *benchmark* LFR, Lancichinetti et al. [25] propuseram uma série de parâmetros que regem a estrutura dos grafos gerados. Tais parâmetros definem aspectos como o grau médio dos vértices, a distribuição de vértices em comunidades, o percentual de arestas de um vértice que o conecta com outros vértices em sua comunidade e a existência de vértices que pertencem a mais de uma comunidade. Desta forma, grafos gerados a partir deste *benchmark* apresentam uma estrutura heterogênea, podendo apresentar uma boa representação de grafos que podem existir em aplicações reais.

Para construir os grafos sintéticos utilizados nos experimentos desta pesquisa foram seguidas algumas das variáveis utilizadas por Lancichinetti et al. [25] ao apresentar e testar seu *benchmark*. Os valores utilizados para os exponentes que definem a distribuição de graus de vértices e a distribuição do tamanho de comunidades foram 2 e 1, respectivamente. Estes valores são definidos no *benchmark* LFR como esperados em grafos reais. Além disso, para determinar o grau médio dos vértices em cada grafo foram utilizados múltiplos de 5.

Todos os grafos gerados apresentam $10,000$ vértices. O grau médio dos vértices varia entre 5 e 30, com incrementos de 5, com o intuito de proporcionar grafos esparsos (grau médio ≤ 10), grafos medianos (grau médio entre 15 e 20) e grafos densos (grau médio ≥ 25). Para cada grau médio foram criados grafos com *mixing parameter* entre 0.1 e 0.6, com incrementos de 0.1. A variável do *mixing parameter* determina o percentual médio das arestas de um vértice que se encontram fora da comunidade a qual o vértice pertence. Desta forma, um *mixing parameter* de 0.1 representa que, em média, 10% das arestas de um vértice se conectam com vértices fora da comunidade. Isto significa que, conforme o *mixing parameter* aumenta, a estrutura de comunidade do grafo se torna menos perceptível. Como demonstrado por Fortunato e Hric [10], algoritmos de detecção de comunidades apresentam um rápido declínio de qualidade de solução a partir de um *mixing parameter* de 0.5. Portanto, optou-se por realizar testes de acurácia com valores apenas um pouco maiores.

4.1.2 Grafos Reais

O objetivo em utilizar grafos reais para os experimentos de performance é verificar o desempenho dos algoritmos em testes que representem possíveis utilizações em aplicações. Para isto, foram priorizadas duas características ao selecionar grafos:

1. Tamanho: É necessário que os grafos selecionados possuam um tamanho (número de vértices e arestas) que gerem um tempo de execução que permita que as abordagens paralelas consigam um ganho de desempenho.

2. Tipo: Foram selecionados grafos que representam dados que compreendem um possível uso de algoritmos de detecção de comunidades em aplicações reais.

Dentre os grafos reais existentes no repositório SNAP [26] foram selecionados quatro grafos de redes sociais (Friendster, Orkut, Live Journal e Youtube), um grafo de produtos (Amazon), um grafo de topologia da internet (Skitter) e um grafo de estradas (RoadCA). Alguns atributos topológicos destes grafos podem ser vistos na Tabela 4.1. Todos os grafos apresentam arestas bidirecionais para que as análises feitas sobre os algoritmos sejam executadas utilizando o mesmo tipo de modelo de solução, uma vez que, para resolver grafos com arestas unidirecionais, são utilizados cálculos diferentes.

Tabela 4.1: Grafos reais utilizados

Grafo	**Vértices**	**Arestas**	**Grau Médio**
Friendster	65.608.366	1.806.067.135	55
Orkut	3.072.626	11.7185.083	76
Live Journal	4.036.538	34.681.189	17
Youtube	1.157.827	2.987.624	5
Skitter	1.696.415	11.095.298	13
Amazon	548.551	925.872	3
RoadCA	1.971.281	2.766.607	2

4.2 Análise da Acurácia dos Algoritmos

Ao construir implementações paralelas e distribuídas para algoritmos sequenciais é necessário levar em consideração a necessidade de similaridade de resultado entre as diferentes versões. No caso de algoritmos de detecção de comunidades, é de extrema importância que as abordagens paralelas utilizadas mantenham um nível similar de qualidade ao da solução do algoritmo original. Em alguns casos, pode ser preferível um algoritmo que sacrifique qualidade por velocidade, mas isto deve ser considerado quando se está decidindo qual algoritmo utilizar, e não quando se está utilizando uma implementação paralela de um algoritmo.

Com o intuito de verificar o impacto que as abordagens paralelas têm sobre a qualidade da solução, todos os algoritmos e suas versões paralelas foram executadas sobre todos os grafos sintéticos descritos anteriormente. Para verificar a qualidade dos resultados foi utilizada a métrica de *pairwise Precision-Recall* sugerida pelo Graph Challenge [22].

A Figura 4.1 mostra a qualidade da solução alcançada pelo algoritmo de Louvain e duas versões da abordagem utilizada pela ferramente Grappolo, onde cada linha representa grafos de mesmo grau médio mas com variação de *mixing parameter*. A primeira abordagem, ilustrada pela Figura 4.1b, compreende a utilização de *threads* para o processamento dos movimentos de vértices no grafo, utilizando cláusulas *atomic* para realizar os movimentos. A segunda abordagem, ilustrada pela Figura 4.1c, compreende, além da utilização de *threads*, a utilização de uma etapa de pré-processamento do grafo, onde todos os vértices são atribuídos a uma cor de forma que vértices não possuam a mesma cor que seus vizinhos.

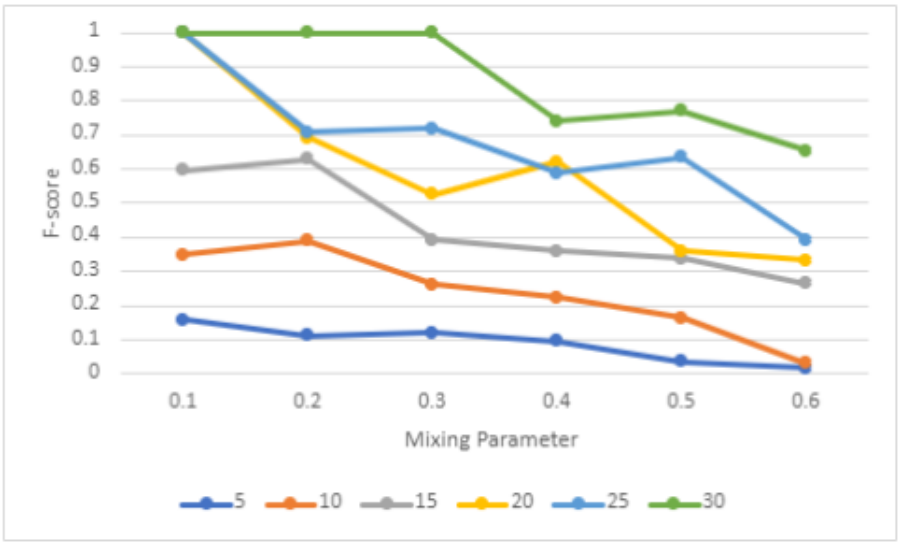
Estudos acerca da equação da modularidade apontam a existência de um limite de solução proveniente da utilização da equação para detectar comunidades [10]. Este limite é referente ao tamanho das comunidades do grafo, onde comunidades pequenas não são corretamente encontradas por algoritmos que utilizam a equação. Este atributo é verificável nas execuções do algoritmo sequencial e da versão paralela simples, onde a acurácia é diretamente proporcional a densidade do grafo. Como os todos os grafos possuem o mesmo número de vértices, aqueles com um menor grau médio acabam por possuir um número maior de comunidades pequenas, uma vez que são esparsos.

Os resultados obtidos nos testes da abordagem que utiliza um pré-processamento foram inesperados. Embora a modularidade final obtida pelas execuções seja muito próxima a obtida através da abordagem simples, os resultados em termos de acurácia com o *ground truth* das comunidades são os piores encontrados neste estudo. Os motivos para tais resultados não podem ser inferidos apenas com os experimentos de acurácia, uma vez que é necessário entender o comportamento geral do algoritmo para tal.

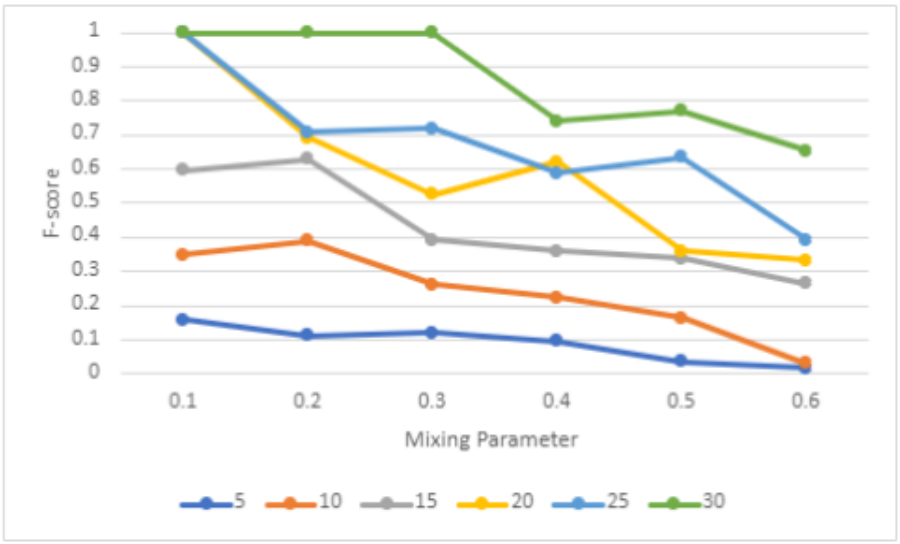
A Figura 4.2 mostra a qualidade da solução alcançada pelo algoritmo Label Propagation e sua versão paralela. A principal atração deste algoritmo é seu tempo de execução, que, como será demonstrado nas próximas seções, é muito menor do que outros algoritmos comumente utilizados. No entanto, este algoritmo apresenta uma clara troca de qualidade por velocidade, principalmente para grafos que possuem suas estruturas de comunidades menos definidas. Além disso, a densidade do grafo aparenta estar diretamente relacionada com a perda de qualidade de solução.

É importante mencionar que a abordagem paralela do algoritmo Label Propagation possui uma acurácia igual ou maior do que o algoritmo sequencial em todos os grafos utilizados. Isto é um comportamento que já foi observado [55], e está ligado, principalmente, ao fato de que execuções paralelas tendem a não gerar máximos locais.

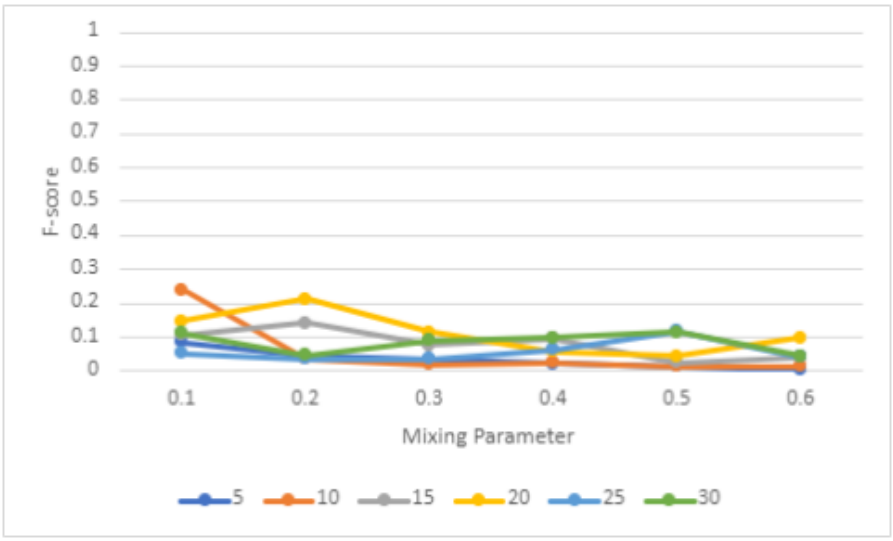
A Figura 4.3 mostra a precisão alcançada pelo algoritmo Infomap e por uma execução de 24 *threads* do algoritmo Relaxmap nos gráficos sintéticos criados. Ambos os algoritmos foram executados usando o mesmo *threshold* ($1e^{-3}$) e semente (1). No geral, o algoritmo Infomap apresenta uma qualidade de solução pior que o Relaxmap para grafos com menor grau médio. É importante ter em mente que o Infomap tende a ter um MDL final



(a) Louvain

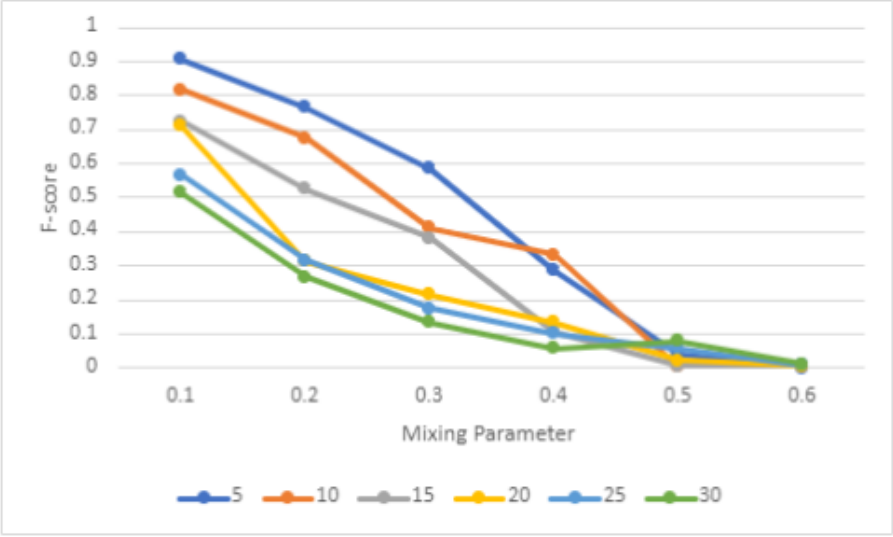


(b) Grappolo

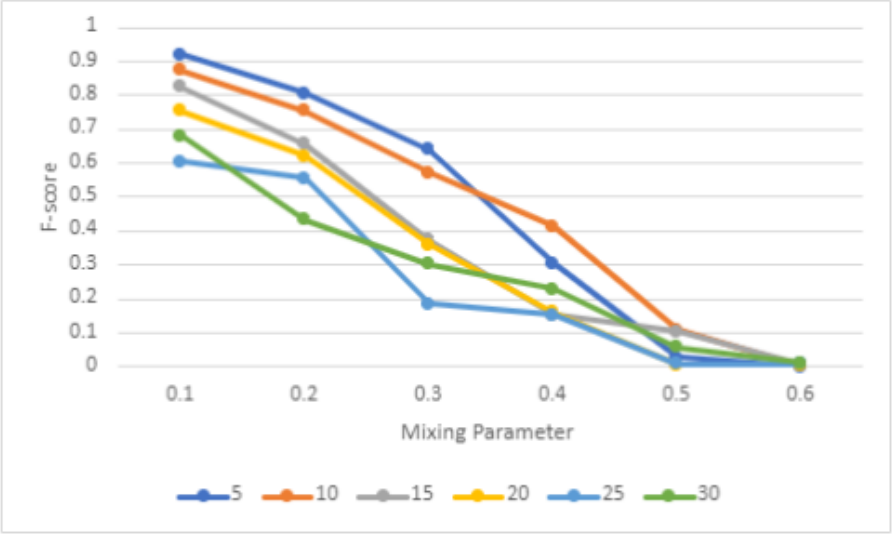


(c) Grappolo com coloração

Figura 4.1: F-score obtido da execução do algoritmo Louvain e sua versão paralela.



(a) Label Propagation

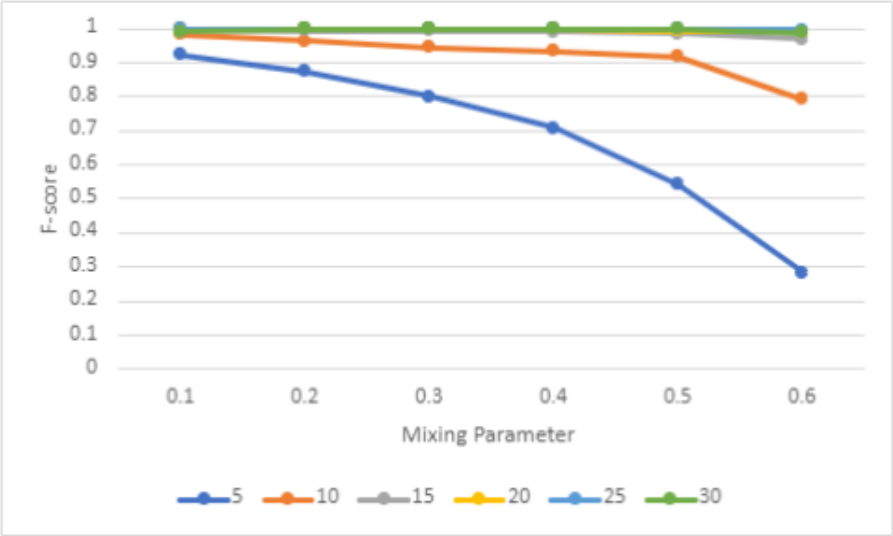


(b) Multithread Label Propagation

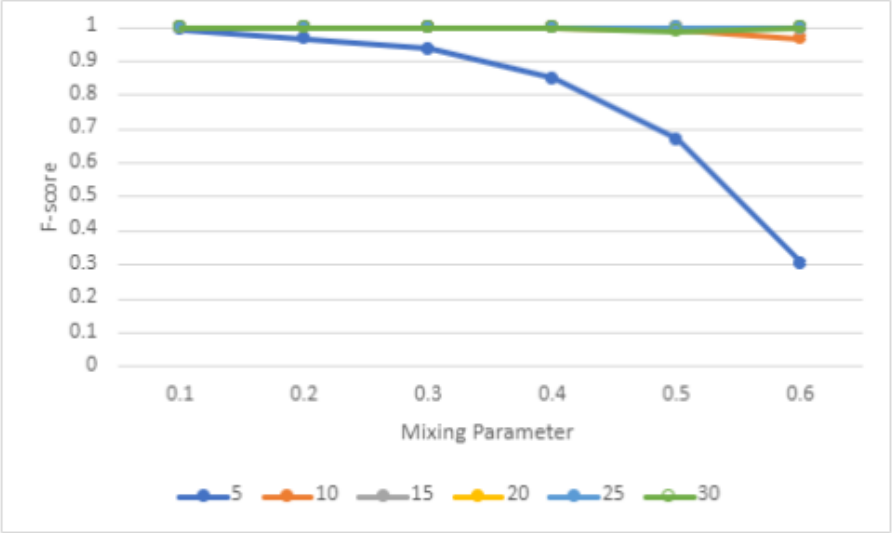
Figura 4.2: F-score obtido da execução do algoritmo Label Propagation e sua versão paralela.

menor, o que significa que o MDL final não é uma referência perfeita para a qualidade da solução. Também é importante apontar que, embora o Infomap apresente uma solução de qualidade inferior ao Relaxmap para alguns grafos, existem outras métricas que não foram utilizadas neste estudo, sendo uma delas a hierarquia de comunidades do grafo. Com isso em mente não é possível estabelecer que o Relaxmap seja necessariamente melhor que o Infomap, mas é possível observar que sacrificar algumas melhorias no MDL ainda pode levar a boas soluções.

A partir dos experimentos de acurácia é possível determinar que, em termos de qualidade da detecção de comunidades, o algoritmo Infomap e sua versão paralela Relaxmap são as melhores opções dentre os algoritmos analisados. No entanto, esta não é a única métrica que deve ser utilizada para escolher qual algoritmo utilizar. É necessário que



(a) Infomap



(b) Relaxmap

Figura 4.3: F-score obtido pela da execução dos algoritmos Infomap e Relaxmap.

os algoritmos produzam suas soluções em tempo hábil a depender das necessidades da aplicação em que estão sendo utilizados.

Para verificar a questão de tempo de execução dos algoritmos é necessário realizar dois tipos de análise. Para algoritmos sequenciais deve-se analisar o tempo de execução para grafos de diferentes tamanhos, uma vez que alguns algoritmos podem apresentar tempos aceitáveis para grafos pequenos, mas se tornarem inviáveis a partir de um certo tamanho de grafo por criarem um número excessivo de computações. Para as abordagens paralelas também é necessário averiguar a escalabilidade da implementação com base no ganho de velocidade de acordo com o número de *threads* utilizadas.

4.3 Escalabilidade das Abordagens Paralelas

Realizar comparações entre execuções dos algoritmos sequenciais com execuções das abordagens paralelas nem sempre é fiel ao desempenho real obtido pelos algoritmos paralelos. Isto ocorre porque, muitas vezes, alterações são feitas para que os algoritmos paralelos obtenham um melhor tempo, desde alterações no funcionamento de certas partes do código até cortes de processos que não possuem um grande impacto na qualidade da solução.

Em função disso, é necessário realizar comparações entre as execuções sequenciais dos algoritmos paralelos e as execuções em múltiplas *threads* dos mesmos. Desta forma pode-se averiguar qual o real impacto da utilização do paradigma paralelo em questão. A Figura 4.4 mostra uma comparação entre o *speedup* e a eficiência atingidos pelo algoritmo Relaxmap em relação a uma execução do algoritmo Infomap e uma execução sequencial do algoritmo Relaxmap.

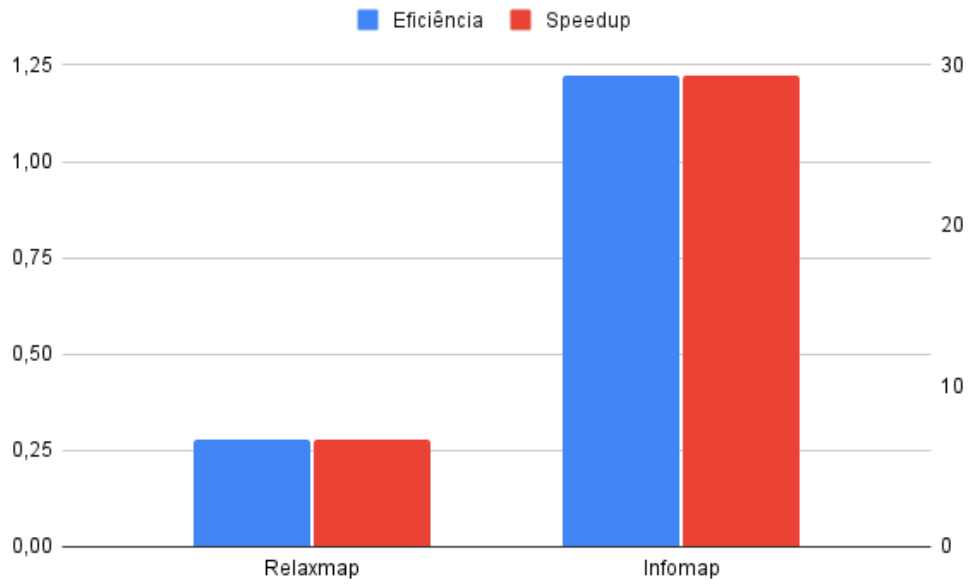


Figura 4.4: Comparação entre o *speedup* sobre o grafo LiveJournal gerado com base em uma execução sequencial do Relaxmap e uma execução do Infomap.

É possível verificar uma drástica diferença entre as duas comparações. Ao comparar o algoritmo Relaxmap com o algoritmo Infomap, é possível ter a impressão de que a abordagem paralela utilizada pelo Relaxmap entrega excelentes resultados, tendo casos onde sua eficiência é maior do que $1,0$. No entanto, ao comparar o algoritmo Relaxmap consigo mesmo, é claro que o modo com que o algoritmo utilizada *threads* representa apenas uma pequena parte da redução do tempo de execução. A grande diferença entre o Relaxmap e o Infomap está, na verdade, na redução de trabalho. Na Seção 4.5 este com-

portamento ficará mais claro, onde será demonstrado que o algoritmo Relaxmap possui um número menor de iterações devido ao relaxamento dos *thresholds* utilizados.

Em função da diferença de comportamento das versões paralelas, os experimentos para verificar a escalabilidade de tais versões foram realizados através da comparação de uma execução sequencial dos algoritmos paralelos e uma execução paralela. Estas execuções foram feitas sobre os grafos reais, com o intuito de demonstrar o desempenho alcançado pelos algoritmos em situações análogas às que ocorreriam em aplicações reais.

Como apresentado na Tabela 4.1, um dos grafos selecionados (Friendster) possui um número de vértices e arestas muito superior aos outros grafos. Tal grafo foi selecionado para representar uma aplicação de grande porte. No entanto, algumas execuções deste grafo apresentam tempos de execução muito altos, tornando-o inviável para alguns dos experimentos realizados posteriormente. Além disso, este é o único grafo que possui um número de execuções alternativo. Todas as medições envolvendo os outros grafos reais apresentados são uma média de 5 execuções do algoritmo, ou seja, 5 execuções sequenciais e 5 execuções paralelas para cada número de *threads*. No caso específico do grafo Friendster, foram realizadas apenas uma execução sequencial e uma execução para cada número de *threads*. A Tabela 4.2 mostra os tempos de execução sequenciais e paralelos da detecção de cada algoritmo para os grafos reais, ou seja, não estão inclusos tempos de operações de I/O.

Tabela 4.2: Tempo de execução sequencial e paralelo em 24 *threads* dos algoritmos paralelos para os grafos reais.

Grafo	**Tempo de Execução (s)**					
	Sequencial			**24 *Threads***		
	Grappolo	**LP**	**Relaxmap**	**Grappolo**	**LP**	**Relaxmap**
Friendster	49.077, 1	409, 8	111.073	4.925, 9	85, 2	13.537, 4
Orkut	1.109, 3	20, 8	2.461, 5	40, 3	5, 1	350, 4
Live Journal	526, 6	7, 6	590, 2	24	5	88, 3
Youtube	21	0, 8	57, 7	2, 6	0, 7	9, 2
Skitter	65, 2	1, 8	113, 1	4, 1	1, 3	17, 7
Amazon	6, 7	0, 3	6, 6	0, 5	0, 08	1, 7
RoadCA	10, 5	0, 8	37, 4	2, 2	0, 4	11, 7

O grafo Friendster é uma excelente representação da necessidade de abordagens paralelas para algoritmos de detecção de comunidades. Ele apresenta tempos de execução sequencias muito altos, principalmente para os algoritmos que apresentam maior acurácia (ultrapassando 24 horas de processamento para o algoritmo Relaxmap). Estes pontos o tornam ideal para iniciar a exploração do desempenho das abordagens paralelas. Por possuir um grande tempo de processamento, existe um grande espaço para melhorias. A Figura 4.5 ilustra o desempenho das abordagens paralelas de acordo com o número de *threads* utilizado para processar o grafo Friendster. Cada linha representa o speedup de cada algoritmo

e cada coluna representa a eficiência. Para este grafo a versão da ferramenta Grappolo utilizada é a simples, que não possui pré-processamento. Dentro deste contexto, todos os algoritmos apresentam uma perda de eficiência conforme o número de *threads* aumenta, sendo que apenas a ferramenta Grappolo consegue se manter em eficiências próximas ou superiores a 50%. Isto mostra que os algoritmos paralelos possuem uma sub-utilização de recursos.

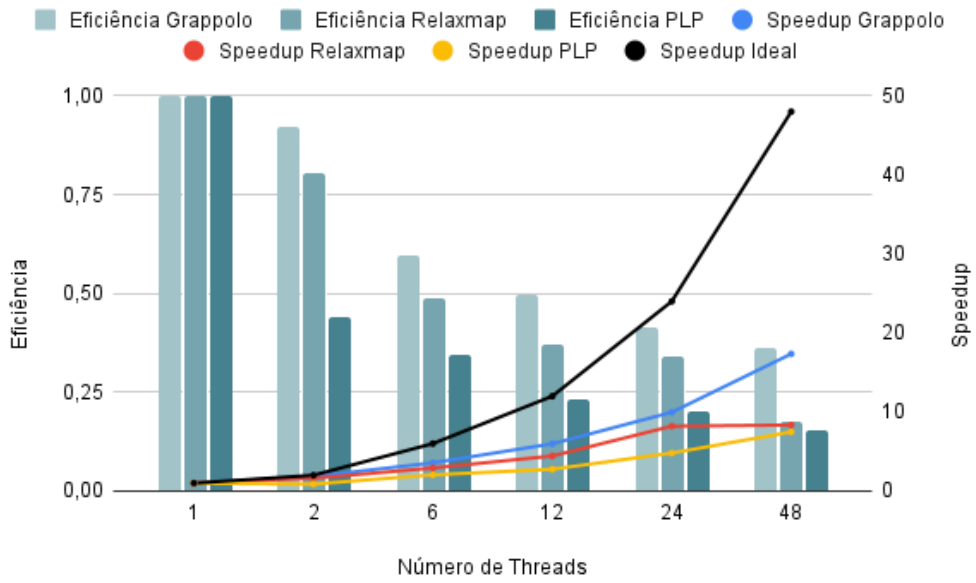
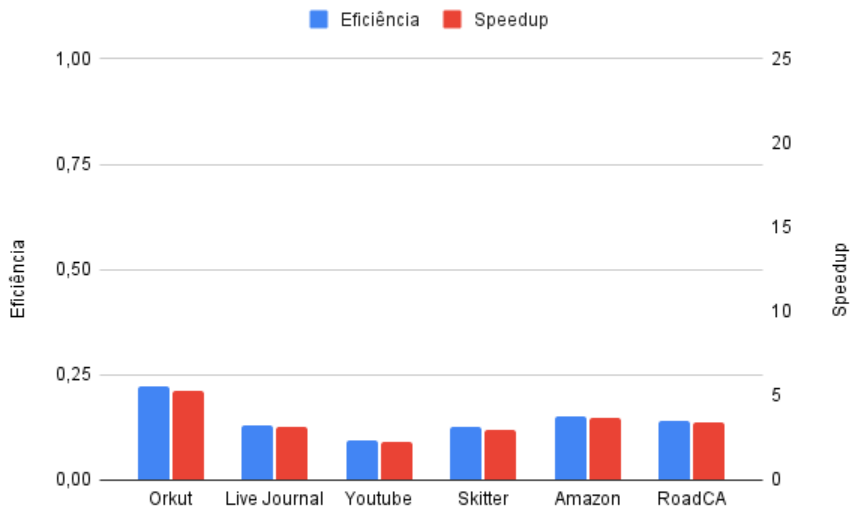


Figura 4.5: *Speedup* e eficiência das abordagens paralelas no grafo Friendster quando comparadas com execuções sequenciais delas mesmas.

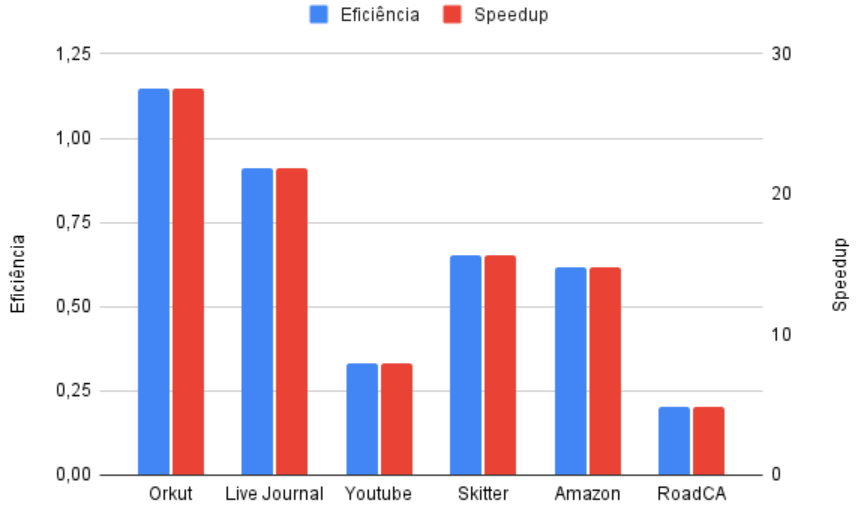
A Figura 4.5 demonstra uma limitação de escalabilidade do algoritmo Relaxmap. O ganho de velocidade é desprezível ao aumentar o número de *threads* de 24 para 48. Isto indica que o algoritmo já atingiu seu limite de escalabilidade. A partir desse fato, foi decidido que os experimentos seriam realizados sobre execuções de 24 *threads*.

A Figura 4.6 mostra o *speedup* e a eficiência da ferramenta Grappolo nos grafos reais restantes. A abordagem sem pré-processamento não apresenta resultados satisfatórios, apresentando eficiências inferiores a obtida na execução do grafo Friendster. A abordagem com coloração, por outro lado, apresenta eficiências altas para alguns grafos, e eficiências semelhantes a abordagem sem pré-processamento para outros. A partir disso é possível verificar que a utilização de coloração pode gerar um grande ganho de velocidade. No entanto, como visto anteriormente, esta abordagem prejudica a qualidade da solução.

A Figura 4.7 mostra o *speedup* e a eficiência atingidos pelos algoritmos Relaxmap e Label Propagation nos demais grafos reais. O algoritmo Label Propagation apresenta os piores resultados dentre todos os algoritmos, assim como nas execuções sobre o grafo Friendster. No entanto, este algoritmo apresenta os menores tempos de execução dentre os algoritmos abordados, independente de serem execuções sequenciais ou paralelas.



(a) Grappolo simples



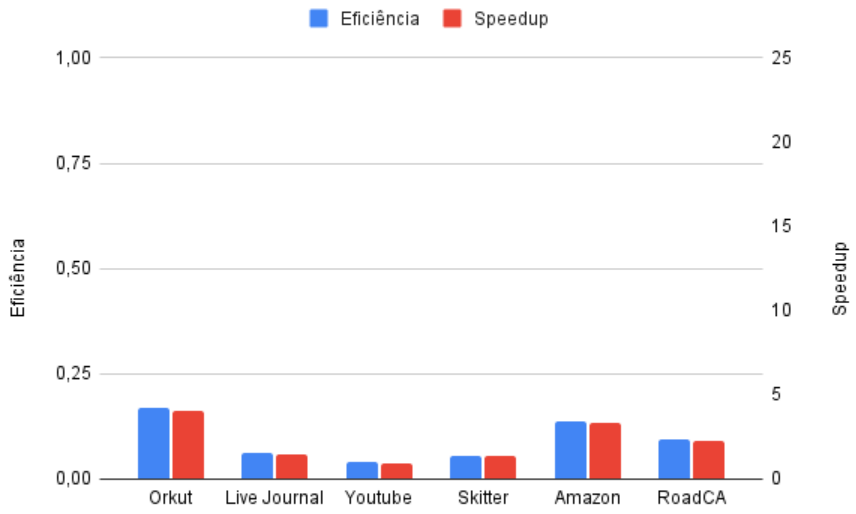
(b) Grappolo com coloração

Figura 4.6: *Speedup* e eficiência das duas abordagens da ferramenta Grappolo para execuções em 24 *threads*.

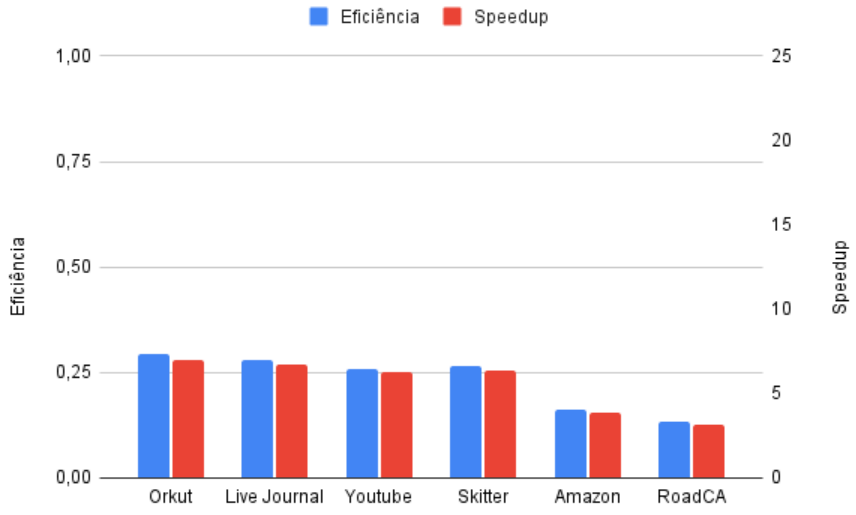
Como o algoritmo Label Propagation é muito simples e naturalmente muito rápido, não existem muitas melhorias que poderiam ser realizadas em seu processamento. A suposta falta de escalabilidade do algoritmo pode ser apenas o resultado da utilização de grafos que não são grandes o suficiente para permitir que o algoritmo apresente altos *speedups*. E, mesmo que o algoritmo não possua valores de eficiência acima de 50% para grafos maiores (como é o caso do grafo Friendster), seu tempo de execução continua sendo muito menor do que outros algoritmos.

O algoritmo Relaxmap apresenta resultados piores do que os atingidos sobre o grafo Friendster. Isso indica que o algoritmo, embora tenha resultados melhores para grafos maiores, possui problemas sérios de escalabilidade. Como o Relaxmap apresenta os

maiores tempos de execução sequenciais ele deveria possuir o maior potencial para ganhos de *speedup*.



(a) Label Propagation



(b) Relaxmap

Figura 4.7: *Speedup* e eficiência dos algoritmos Label Propagation e Relaxmap para execuções em 24 *threads*.

Todos os algoritmos paralelos apresentam problemas de escalabilidade, com exceção da versão com pré-processamento da ferramenta Grappolo. No entanto, para conseguir entender os possíveis motivos desses problemas é necessário realizar uma análise do comportamento dos algoritmos. No caso específico do algoritmo Label Propagation, mesmo uma análise mais profunda não irá proporcionar resultados relevantes, uma vez que o algoritmo, além de simples, possui tempos de execução tão baixos que a real diferença de processamento estará na máquina sendo utilizada.

No caso da ferramenta Grappolo é necessário entender quais seriam os possíveis motivos da versão com pré-processamento estar resultando em soluções de qualidade in-

ferior. Como a versão em questão já atinge bons resultados de eficiência e *speedup* para a maioria dos grafos, verificar que tipos de comportamentos diferentes são gerados. Desta forma, pode-se supor quais ajustes podem beneficiar a versão com pré-processamento. Mesmo ajustes que aumentem o tempo de execução podem se tornar benéficos, uma vez que a versão já possui bons resultados de escalabilidade.

No caso do algoritmo Relaxmap é necessário entender como seu comportamento muda em relação ao algoritmo Infomap. Como o Relaxmap realiza relaxamentos para diminuir seu tempo de execução, é possível que os pontos do algoritmo que precisam ser trabalhados sejam diferentes do algoritmo Infomap. Desta forma, entender quais são as diferenças de comportamento podem auxiliar em uma evolução específica do algoritmo Relaxmap.

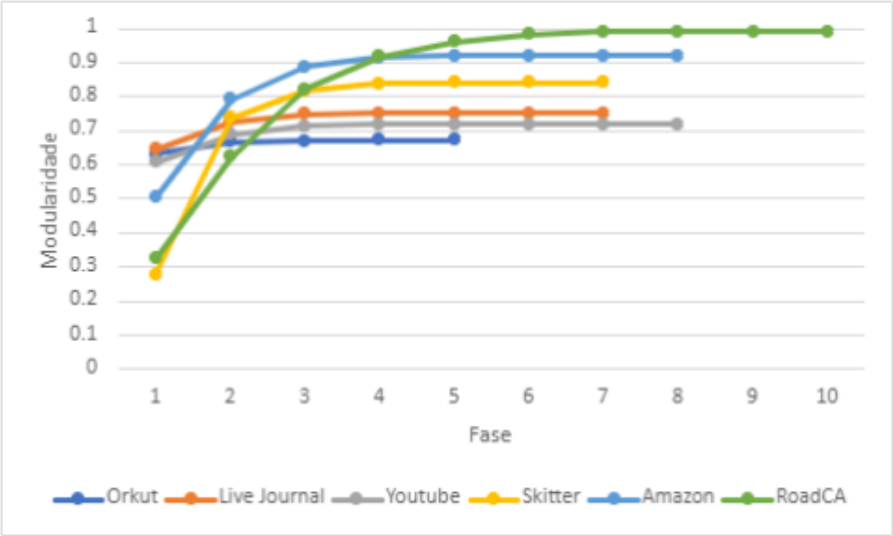
As próximas seções tratam da caracterização da ferramenta Grappolo e dos algoritmos Infomap e Relaxmap, com o intuito de explorar as possíveis melhorias que podem ser construídas para cada algoritmo.

4.4 Caracterização da Ferramenta Grappolo

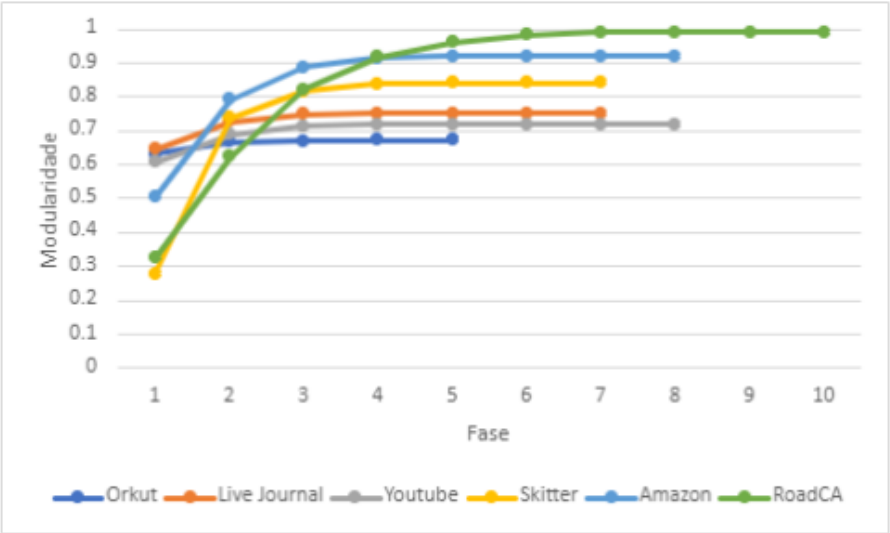
Para entender as diferenças de comportamento da abordagem paralela que utiliza de coloração do grafo é necessário, inicialmente, entender como as execuções sequencial e da abordagem simples da ferramenta Grappolo se comportam. Para isso, cada fase do algoritmo foi analisada, obtendo-se diferentes informações. Entende-se por fase a detecção de um nível do algoritmo, ou seja, o procedimento de detectar comunidades em um grafo e, em seguida, construir super-vértices a partir destas comunidades para, finalmente, construir um novo grafo. Desta forma, cada fase representa um dos níveis da solução final.

O primeiro ponto a ser abordado é a modularidade atingida em cada fase de cada versão, que pode ser vista na Figura 4.8. O crescimento da modularidade da execução sequencial e da execução simples de 24 *threads* é igual para todos os grafos. No entanto, para a execução com coloração ocorrem alterações para as primeiras fases. A versão com coloração inicia com um valor maior de modularidade, realizando incrementos menores ao longo das fases. Isto indica que a versão com coloração gera uma solução diferente desde o primeiro nível, fazendo com que todos os níveis subsequentes também gerem uma solução diferente do algoritmo simples.

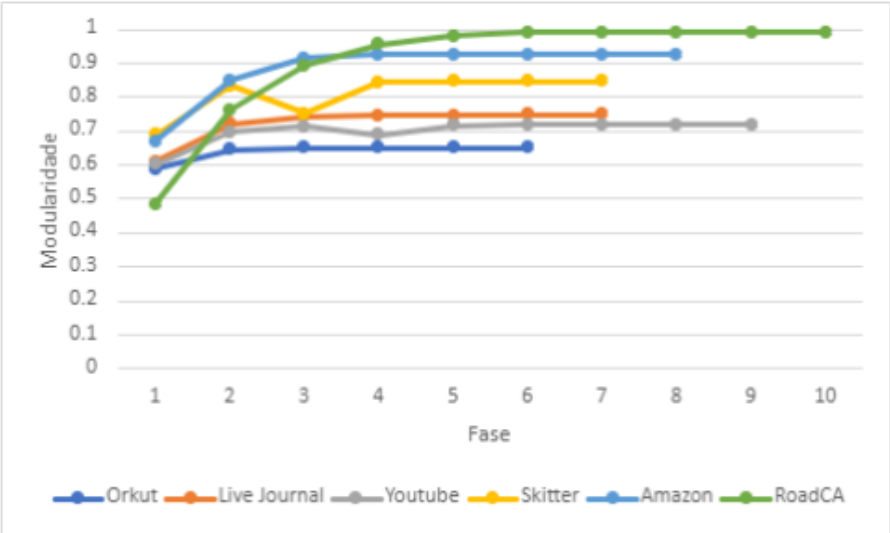
Com a informação de que as versões produzem soluções diferentes desde o início de suas execuções resta descobrir qual é o impacto gerado no tempo de execução geral. Para isso, foram analisadas as computações e os movimentos realizados por cada versão em cada fase. Computações são realizadas para procurar movimentos ótimos, sendo o processo de calcular a modularidade resultante de um possível movimento uma computação. Desta forma, quando uma *thread* está verificando qual seria o movimento ótimo de um



(a) Grappolo simples sequencial



(b) Grappolo simples com 24 *threads*



(c) Grappolo com coloração com 24 *threads*

Figura 4.8: Modularidade atingida em cada fase de cada versão da ferramenta Grappolo.

vértice ela irá realizar um número de computações equivalente a no máximo o número de vizinhos do vértice.

Para a execução sequencial e a execução simples são executados o mesmo número de computações e movimentos, enquanto para a versão com coloração são executados menos computações e movimentos. A Figura 4.9 mostra o percentual de computações que resultam em um movimento por fase. A versão com coloração apresenta uma distribuição de movimentos diferente das outras execuções.

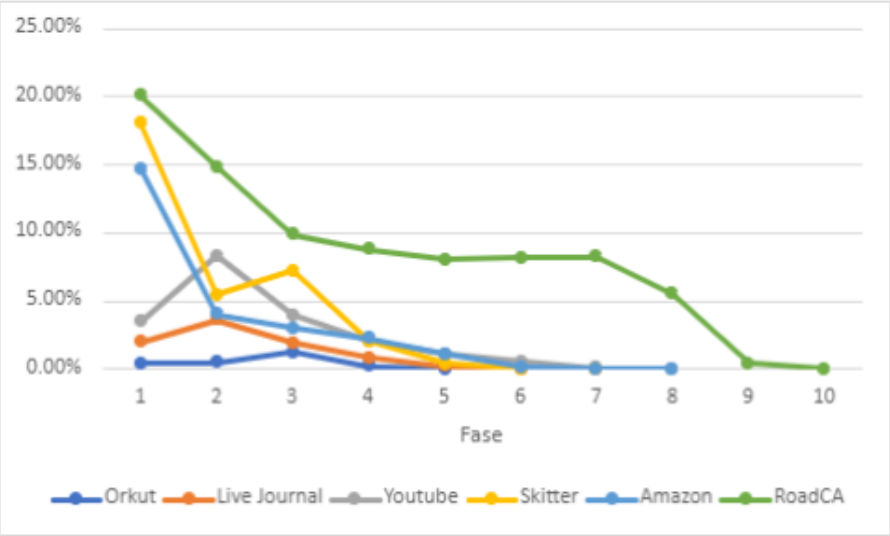
Os problemas de acurácia da versão com coloração provavelmente estão ligados aos comportamentos das fases iniciais do algoritmo, que produzem uma solução diferente do esperado. A solução encontrada nas fases iniciais propaga-se pelas fases futuras, prejudicando a detecção de comunidades. Um dos possíveis motivos para essas diferenças é a ordem de processamento dos vértices, que passa a ser ditada pelas cores do grafo.

Como a execução simples em paralelo mantém o mesmo comportamento e resultados da execução sequencial, pode-se afirmar que a utilização de cláusulas *atomic* para realizar o movimento dos vértices não tem impacto negativo na convergência da solução. No entanto, ainda resta verificar o motivo da versão simples, não escalar de forma satisfatória. Para isso, é necessário verificar a distribuição de tempo dentro do algoritmo.

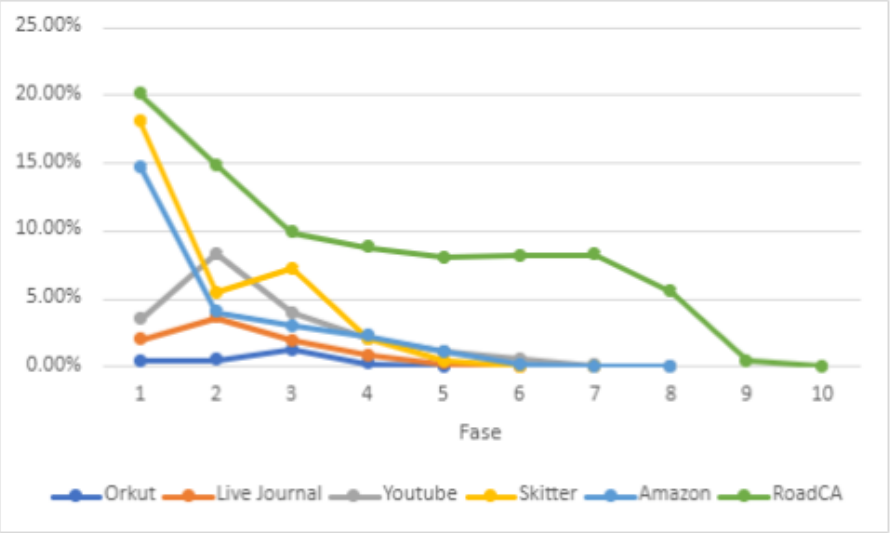
A Figura 4.10 mostra a distribuição de tempo durante os estágios de detecção do algoritmo. As operações contabilizadas são preparação, computação e movimentação. Preparação diz respeito ao tempo gasto com preparos para realizar computações, onde são calculadas as variáveis necessárias. Computação diz respeito ao tempo gasto com computações. Movimentação diz respeito ao tempo gasto com a movimentação de vértices entre comunidades. Para a Figura 4.10b são calculadas as médias de tempo de execução de cada *thread*.

No caso da execução sequencial o maior consumo de tempo está localizado nas preparações. Um dos fatores para isso ocorrer é que as preparações sempre são realizadas para todos os vértices, enquanto computações podem não ocorrer dependendo das comunidades às quais os vizinhos de cada vértice pertencem. O caso das execuções paralelas, no entanto, mostra diferenças.

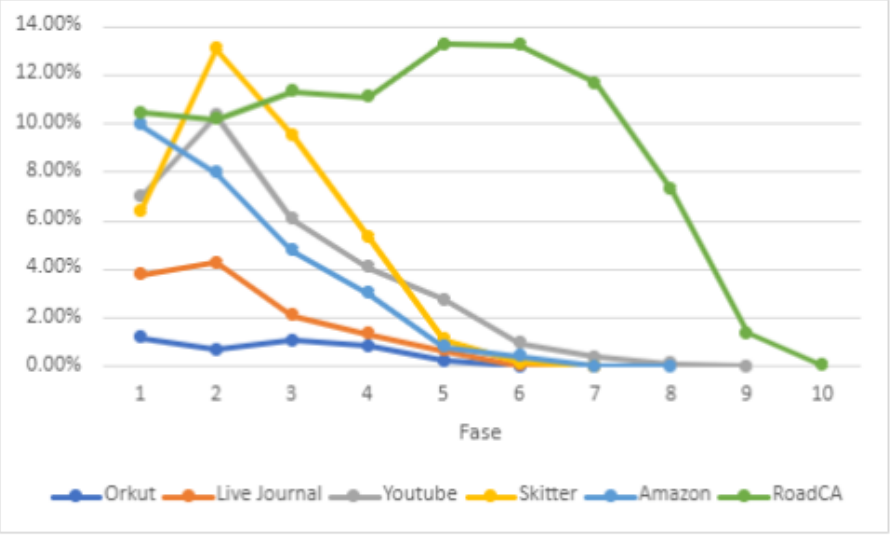
Como a execução simples paralela possui o mesmo comportamento da versão sequencial, é muito estranho que as duas versões possuam tempos relativos tão diferentes. Foi, então, verificado não somente o tempo médio que uma *thread* leva para completar as partes da detecção mas também qual é o tempo que a *thread* com maior tempo de execução leva. Com isso foi observado que o comportamento da execução paralela simples é igual a execução sequencial, indicando que a implementação paralela possui um problema de balanceamento de carga.



(a) Grappolo simples sequencial

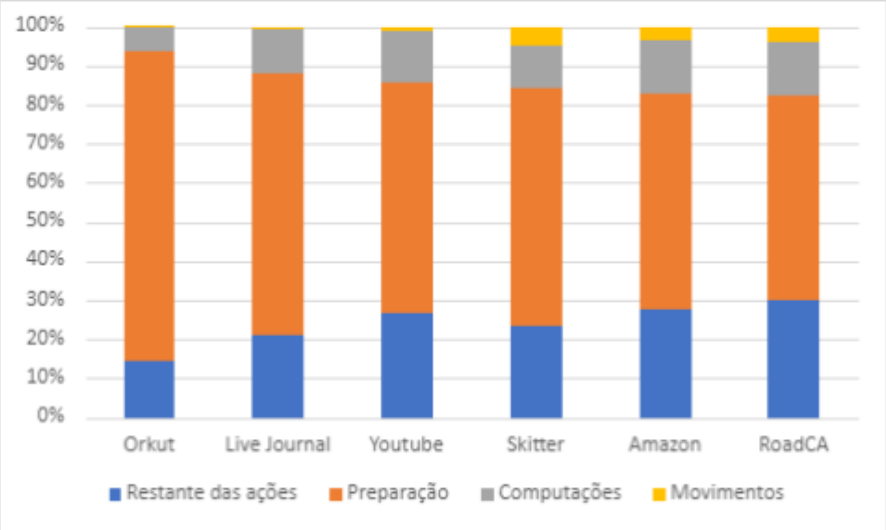


(b) Grappolo simples com 24 *threads*

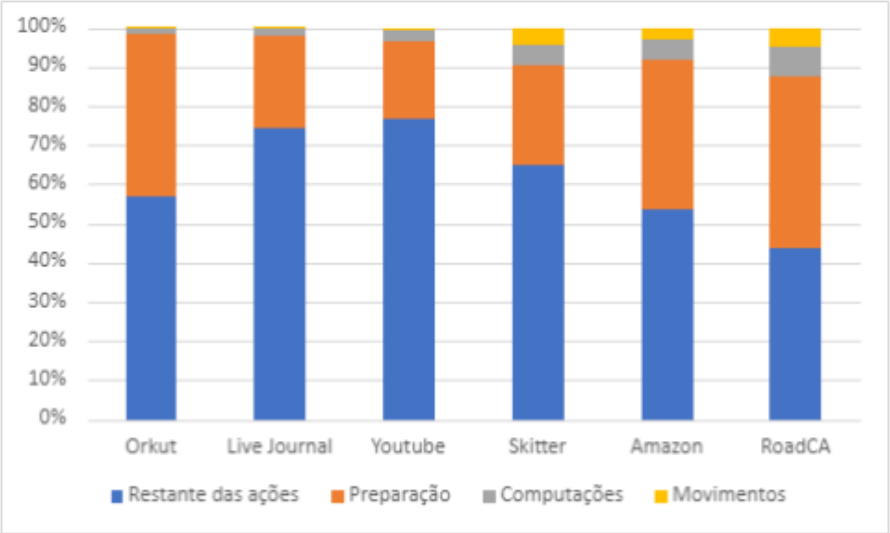


(c) Grappolo com coloração com 24 *threads*

Figura 4.9: Percentual de computações que resultam em um movimento.



(a) Grappolo simples sequencial

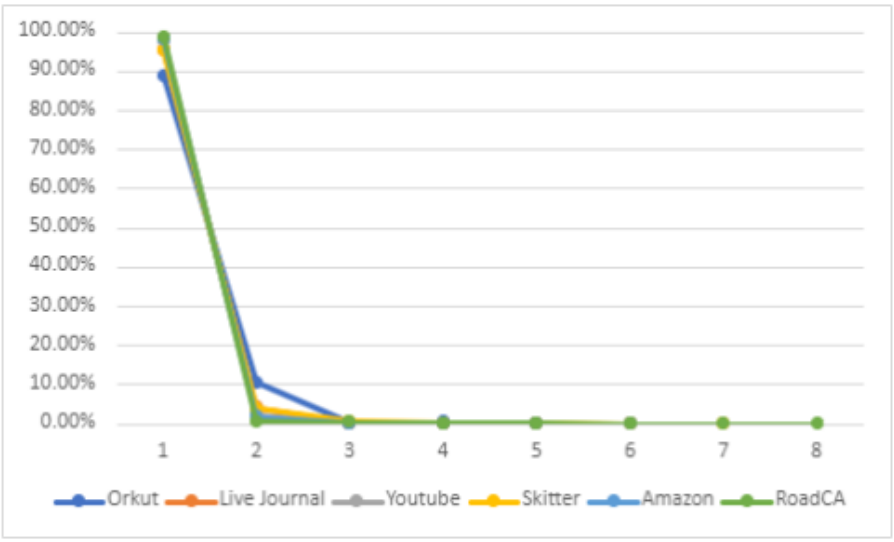


(b) Grappolo simples com 24 *threads*

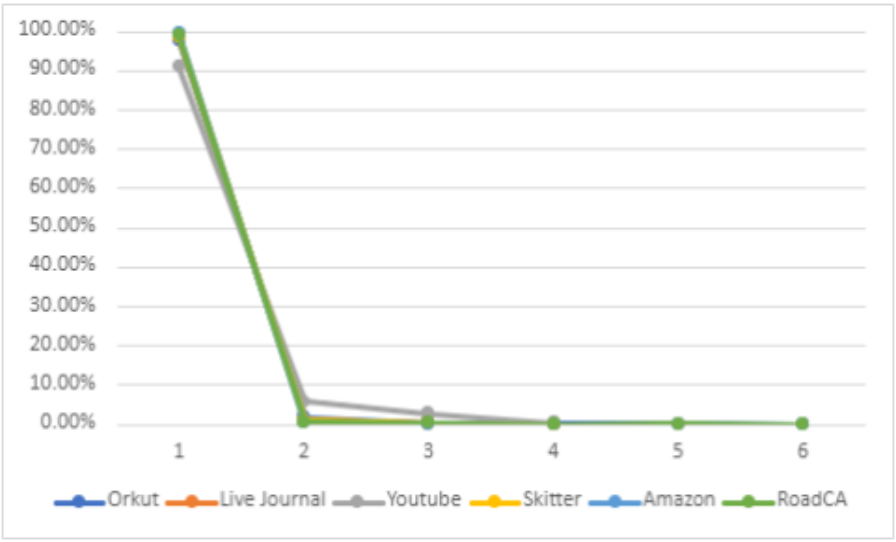
Figura 4.10: Tempo relativo gasto por cada ação da detecção dos algoritmos.

4.5 Caracterização dos Algoritmos Infomap e Relaxmap

Para verificar os problemas de escalabilidade dos algoritmos Infomap e Relaxmap, cada etapa de cada algoritmo foi analisada para que fosse possível entender como elas afetam a detecção de comunidades e o consumo geral de tempo. Inicialmente, para verificar a contribuição de cada iteração de ambos os algoritmos, foi calculado quanto da redução geral do MDL foi alcançada em cada iteração. A Figura 4.11a ilustra o algoritmo Infomap e a Figura 4.11b ilustra o algoritmo Relaxmap. É importante notar que a primeira iteração representa toda a etapa de detecção, enquanto as iterações pares subsequentes representam um *Fine Tune* e as iterações ímpares representam um *Coarse Tune*.



(a) Infomap

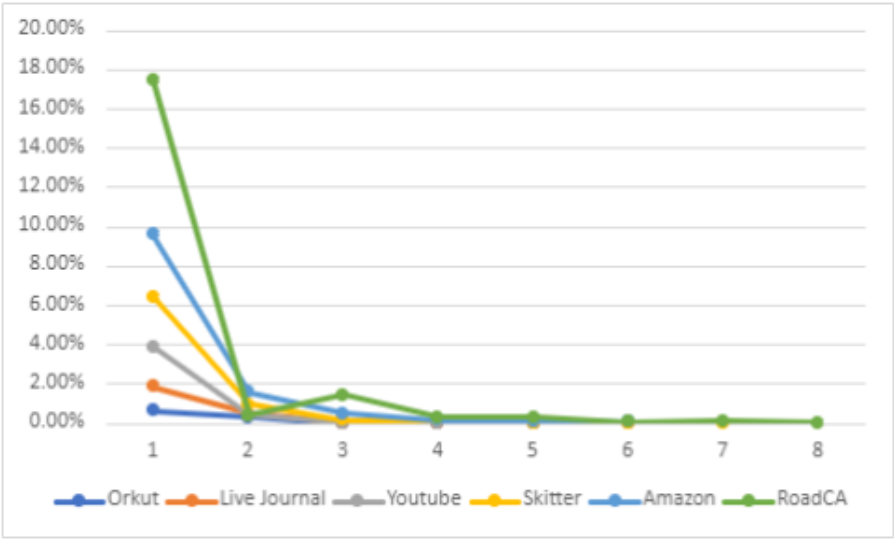


(b) Relaxmap

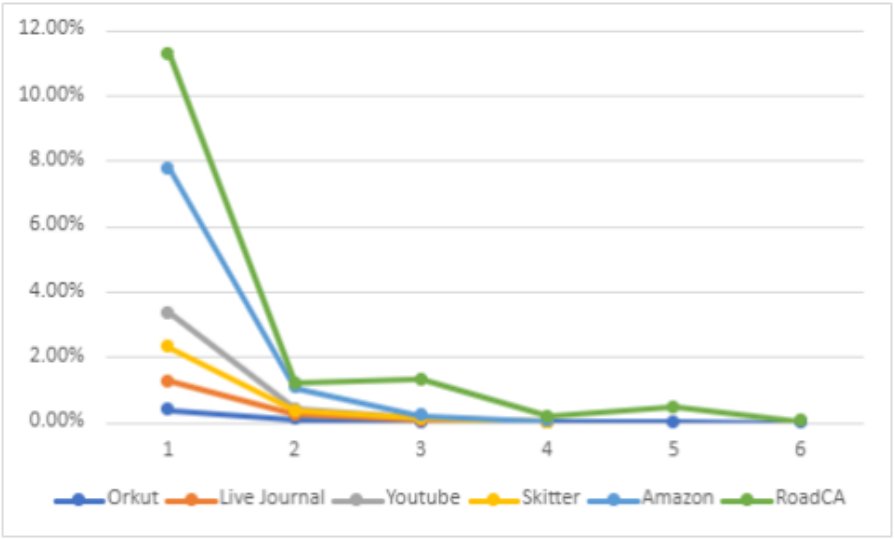
Figura 4.11: Percentual da redução do MDL por iteração em relação a redução total obtida pelos algoritmos Infomap e Relaxmap.

A etapa de detecção de ambos os algoritmos tem o maior impacto na redução do MDL, o que é esperado, pois o objetivo da etapa de refino é ajustar possíveis erros da etapa de detecção. Isso fica ainda mais claro na Figura 4.12a e Figura 4.12b, que mostram a porcentagem de cálculos feitos para encontrar movimentos ideais que realmente resultaram em movimentos. Esta é a primeira grande diferença entre os algoritmos: Infomap tem uma taxa mais alta de cálculos que resultam em movimentos. Isso pode ser resultado dos cálculos extras feitos pelo Relaxmap quando uma *thread* está decidindo se um movimento ainda é válido.

Para descobrir se o aumento nos cálculos tem um impacto negativo no desempenho do Relaxmap, foi verificado o tempo de execução de cada etapa de ambos os algoritmos. A Figura 4.13 apresenta a distribuição de tempo de cada etapa de cada algoritmo,



(a) Infomap



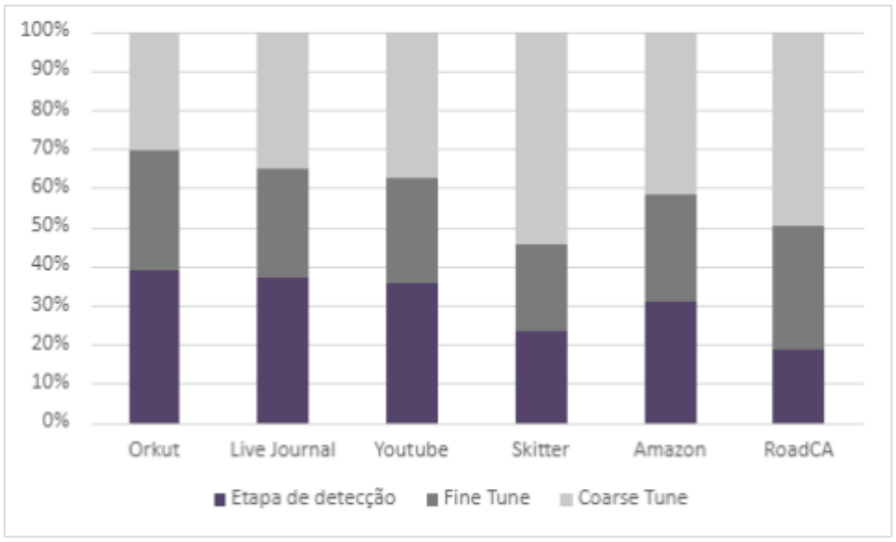
(b) Relaxmap

Figura 4.12: Percentual de cálculos de movimentos que resultam em um movimento por iteração nos algoritmos Infomap e Relaxmap.

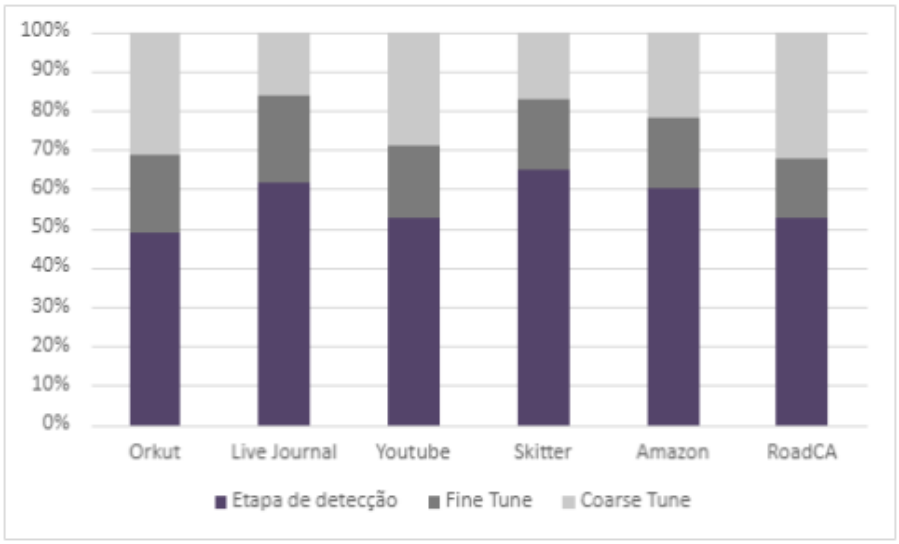
com a etapa de refino dividida em suas duas partes. O algoritmo Relaxmap tem um aumento notável no tempo relativo da etapa de detecção, e como a única abordagem paralela utilizada nesta etapa está ligada aos cálculos para encontrar movimentos ótimos, pode-se supor que a existência de uma seção crítica sobre os movimentos tem um grande impacto na etapa de detecção.

4.5.1 Análise das Iterações

É possível observar que a parte *Coarse Tune* geralmente representa a maior parte do tempo de processamento gasto na etapa de ajuste em ambos os algoritmos. Isso pode



(a) Infomap



(b) Relaxmap

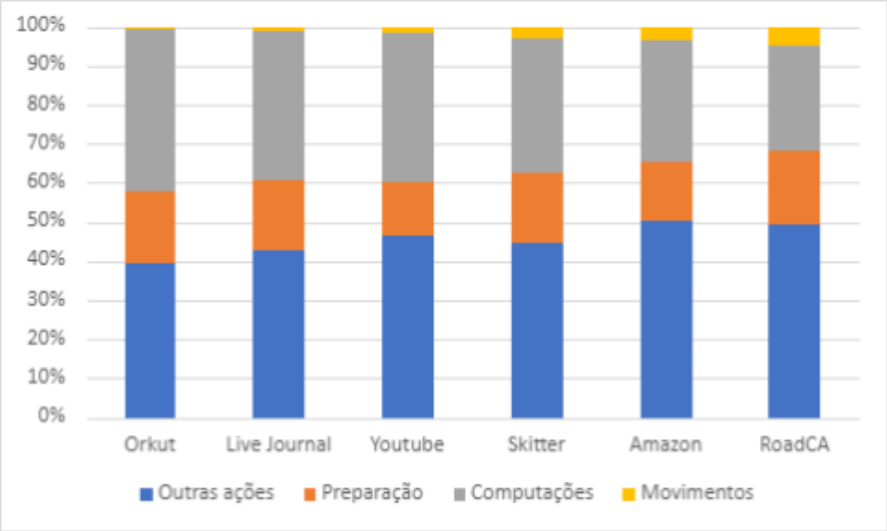
Figura 4.13: Distribuição de tempo consumido por cada etapa dos algoritmos Infomap e Relaxmap.

significar que o processo de dividir as comunidades em submódulos possui maior influência no tempo de processamento do que tentar encontrar movimentos ideais. Para explorar essa possibilidade foram verificados os tempos de execução das ações de cada etapa. Para cada passo foi verificado o tempo gasto nas ações envolvidas na realização dos movimentos, que são: movimentos, cálculos para movimentos ótimos e os passos preparatórios necessários para realizar os cálculos. O tempo de execução dessas ações foram comparados com o restante do tempo gasto em cada iteração. É importante notar que cada passo tem um conjunto diferente de ações fora do método de movimento. No caso da etapa de detecção existe a construção de super-vértices e novos grafos. Para o *Fine Tune* também existe a reatribuição de vértices para que possam movimentar-se entre comunidades novamente. Para o *Coarse Tune* também existe a divisão das comunidades em submódulos.

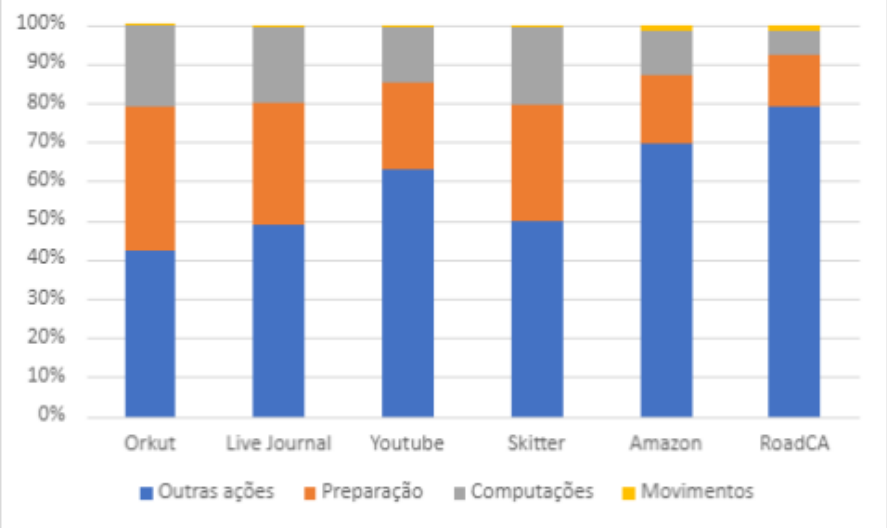
A Figura 4.14 apresenta o consumo de tempo relativo das ações da etapa de detecção. Esta etapa é responsável por quase todos os movimentos de ambos os algoritmos. Ao comparar os dois algoritmos, é possível observar que o algoritmo Relaxmap não apenas gasta menos tempo em movimentos do que o algoritmo Infomap, mas também gasta menos tempo em cálculos. A mudança no consumo de tempo das etapas preparatórias e dos cálculos é provavelmente atribuída às implementações específicas das etapas preparatórias de ambos os algoritmos. Por outro lado, o caso do tempo gasto em movimentos pode ser atribuído à espera pela entrada da seção crítica. Isso pode ser observado nos grafos que têm mais movimento relativo, RoadCA e Amazon. Como todo o tempo excedente do método de movimento de ambos os algoritmos está sendo captado pelo cronômetro utilizado para o restante da iteração e o tempo relativo gasto pelo restante da etapa de detecção teve o maior aumento nos grafos com mais movimentos, é possível inferir que o tempo de espera para acessar a seção crítica tem um impacto considerável no tempo de execução geral da etapa de detecção.

A Figura 4.15 descreve o tempo gasto em ações nas iterações do *Fine Tune*. Esta parte da etapa de ajuste não tem muita diferença entre os algoritmos, além da mudança nas etapas preparatórias, que foi discutida anteriormente. Como o comportamento desta parte é quase idêntico ao da etapa de detecção, mas com menos movimentos, o impacto da seção crítica existirá, mas em menor escala.

A etapa *Corse Tune* apresenta uma distribuição de tempo de processamento muito diferente das demais etapas de ambos os algoritmos. A Figura 4.16 ilustra o tempo gasto em cada ação. Como visto anteriormente, o *Corse Tune* representa a maior parte do tempo gasto na etapa de ajuste, mas não há um aumento significativo no movimento. Como no caso do *Fine Tune* a seção crítica não tem grande impacto, o mesmo pode ser inferido para o *Corse Tune*. Isso aparentemente é verdade, uma vez que o aumento das ações fora do método de movimento não é tão significativo quanto na etapa de detecção. No entanto, o aspecto interessante do *Corse Tune* é que o tempo gasto fora do método de movimento é maior do que nas outras etapas de ambos os algoritmos. Como a única diferença do *Corse Tune* é a divisão dos módulos em submódulos, é possível inferir que tal operação tem grande impacto no desempenho geral de ambos os algoritmos.

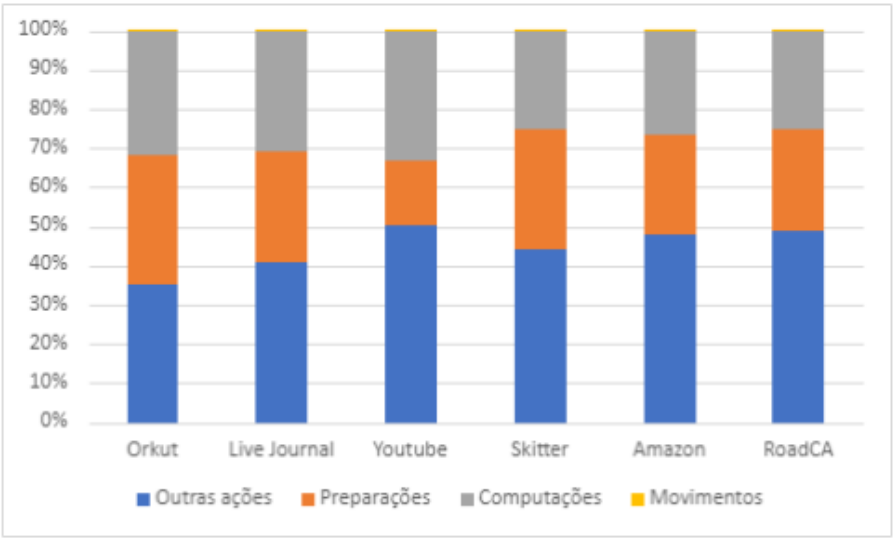


(a) Infomap

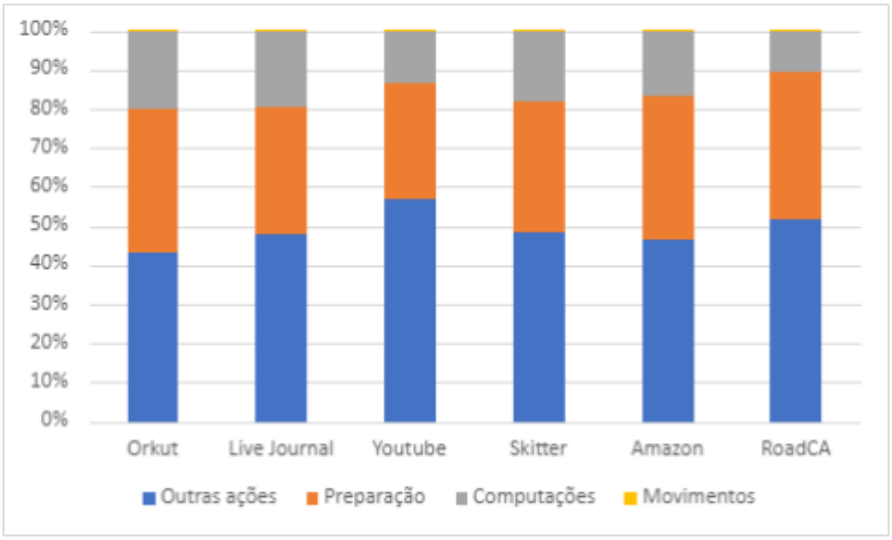


(b) Relaxmap

Figura 4.14: Tempo gasto por cada ação da etapa de detecção nos algoritmos Infomap e Relaxmap.

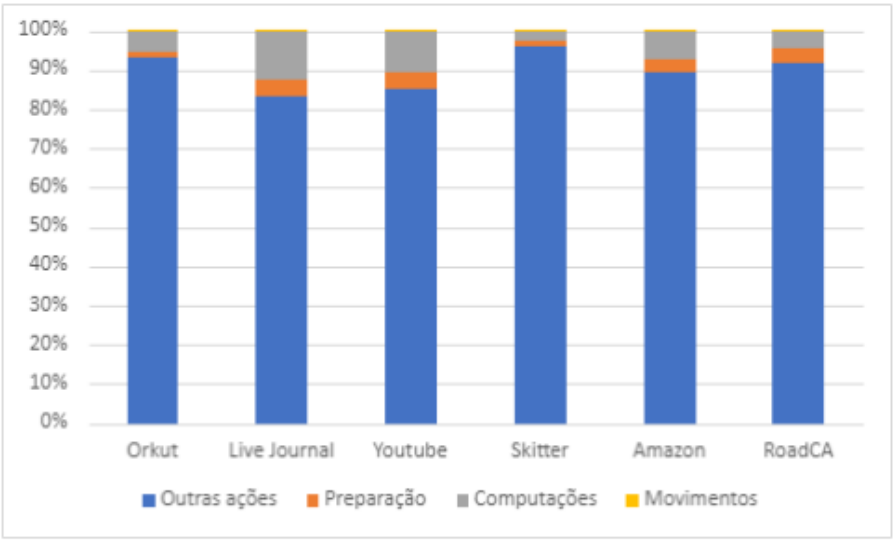


(a) Infomap

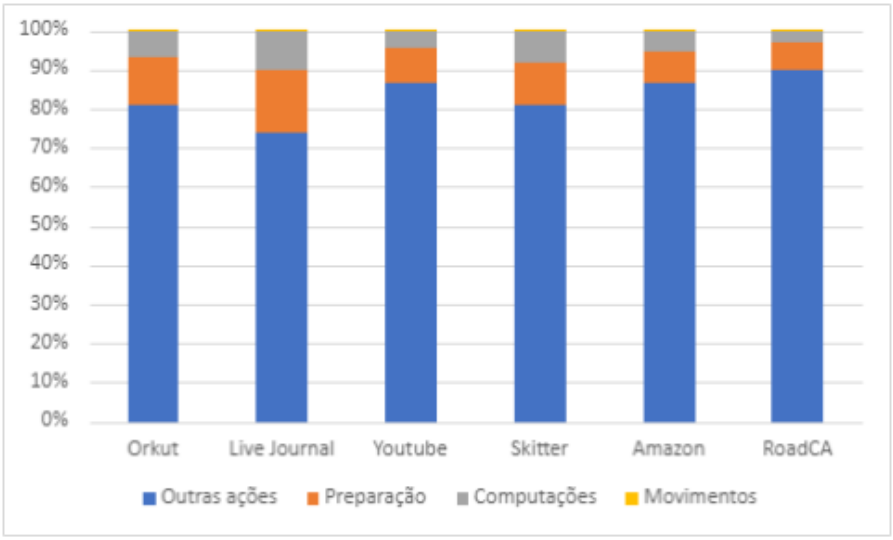


(b) Relaxmap

Figura 4.15: Tempo gasto por cada ação da parte *Fine Tune* nos algoritmos Infomap e Relaxmap.



(a) Infomap



(b) Relaxmap

Figura 4.16: Tempo gasto por cada ação da parte *Coarse Tune* nos algoritmos Infomap e Relaxmap.

5. DISCUSSÃO

A partir dos resultados obtidos através dos experimentos realizados neste estudo é possível construir um conjunto de diretrizes para a utilização dos algoritmos de detecção de comunidades abordados. Tais diretrizes levam em consideração dois aspectos: tempo disponível para a execução do algoritmo e acurácia obtida para cada densidade de grafo. Desta forma, espera-se prover um guia para a seleção de algoritmos de detecção de comunidades de acordo com as necessidades de cada usuário ou aplicação. A Tabela 5.1 mostra as diretrizes gerais.

O algoritmo Label Propagation apresenta os melhores tempos de execução tanto para execuções sequenciais quanto para paralelas, se tornando a escolha ideal para aplicações que dependem de um processamento rápido. Além disso, por não apresentar grandes valores de *speedup*, não é necessário possuir um grande número de recursos para usufruir das vantagens de tempo de execução do algoritmo. No entanto, por motivos de acurácia, recomenda-se utilizar versões paralelas do algoritmo, uma vez que ele apresenta resultados um pouco melhores. A partir desses pontos pode-se sugerir que a utilização do algoritmo Label Propagation deve ser realizada no contexto de aplicações que necessitam de resultados em tempo próximo ao real e que, preferencialmente, tratem de grafos esparsos. Um bom exemplo é o campo de análise de redes sociais: os grafos tendem a ser esparsos e convergir para estruturas de comunidades, além de necessitarem de um rápido processamento, uma vez que sofrem atualizações constantes.

O algoritmo Louvain está localizado como um meio-termo entre os algoritmos abordados tanto em termos de tempo de execução quanto em acurácia. Sua acurácia apresenta um comportamento oposto ao algoritmo Label Propagation em relação a densidade do grafo. O Louvain possui melhores resultados para grafos mais densos, tornando-se a pior escolha para aplicações que utilizam grafos esparsos. O tempo de execução do algoritmo pode se tornar um empecilho para grafos com tamanhos superiores a um bilhão de arestas, mas a utilização de paralelismo no nível da abordagem simples da ferramenta Grappolo consegue contornar tal problema desde que a aplicação em questão não necessite de processamento próximo a tempo real.

O algoritmo Infomap possui a melhor acurácia entre os algoritmos abordados. No entanto, seu tempo de execução pode torna-lo inviável para aplicações que necessitam de resultados em curtos períodos. Mesmo a melhor abordagem paralela, o Relaxmap, não consegue atingir bons tempos. Esses pontos posicionam o algoritmo Infomap como um bom candidato a aplicações que necessitam do melhor resultado possível e não possuam um tempo limite para o processamento do algoritmo, como é o caso de aplicações da área de bioinformática.

Tabela 5.1: Guia de utilização dos algoritmos.

Algoritmo	Requerimento de Tempo			Tipo do Grafo		
	Rápido	**Médio**	**Lento**	**Esparso**	**Mediano**	**Denso**
Louvain		X	X			X
Label Propagation	X	X	X	X		
Infomap			X	X	X	X

5.1 Problemas de Escalabilidade e Possíveis Melhorias

A partir dos experimentos realizados neste trabalho pode-se apontar para uma falta de escalabilidade nas implementações paralelas dos algoritmos abordados. O algoritmo Label Propagation, no entanto, não apresenta uma necessidade de melhorias, uma vez que o algoritmo é simples, naturalmente rápido e só é competitivo em termos de acurácia para grafos com estruturas bem específicas.

A ferramenta Grappolo confirma que não é necessário utilizar uma seção crítica para realizar movimentos no grafo. No entanto, sua versão simples, que apresenta um comportamento igual a execução sequencial, possui problemas claros de escalabilidade. Com os resultados da análise feita sobre os algoritmos, pode-se apontar um problema de balanceamento de carga dentro da ferramenta. Mesmo que a média de tempo que a maioria das *threads* leva para processar a detecção de comunidades seja baixa, o tempo de execução total do algoritmo será determinado pela *thread* com o maior tempo de execução. Isso faz com que o algoritmo simples apresente eficiências muito baixas, uma vez que grande parte dos recursos disponíveis estão sendo sub-utilizados em função da espera pelo término de poucas *threads*.

A proposta de utilizar uma fase de pré-processamento para obter ganhos de eficiência e *speedup* gera bons resultados de escalabilidade para a ferramenta Grappolo. No entanto, ocorre uma troca drástica entre escalabilidade e qualidade da solução. A profundidade dos experimentos realizados neste trabalho não é capaz de identificar o ponto exato onde a coloração do grafo resulta em uma perda de acurácia. É possível identificar que o algoritmo se comporta de maneira diferente, tendo menos computações e distribuição de movimentos diferente, e, embora atinja valores quase iguais de modularidade, que o crescimento da modularidade é alterado. O problema enfrentado por essa versão do algoritmo de Louvain pode estar sendo causado por falsos movimentos ótimos feitos nas fases iniciais do algoritmo, que se propagam para fases posteriores. Como essa versão possui uma escalabilidade alta, existe espaço para explora-la, mesmo que para aumentar a acurácia seja necessário aumentar o tempo de processamento.

A abordagem do algoritmo Relaxmap para paralelizar o algoritmo Infomap possui problemas de escalabilidade devido a escolhas de design. A existência de uma seção

crítica para a realização de movimentos tem um alto impacto no tempo total de execução da primeira etapa do algoritmo. A etapa de ajuste não é tão afetada pela seção crítica porque os movimentos não ocorrem com a mesma frequência, mas é fortemente impactada pela divisão das comunidades em submódulos.

Uma vez que o algoritmo Infomap compartilha semelhanças com o algoritmo Louvain, técnicas desenvolvidas para alcançar implementações escaláveis do Louvain podem ser usadas para melhorar a etapa de detecção do Infomap e Relaxmap. Inicialmente, pode-se explorar utilizar as técnicas implementadas na ferramenta Grappolo. Trocar a seção crítica para movimentos por cláusulas *atomic* pode gerar um ganho de tempo de execução na fase de detecção do algoritmo. Além disso, utilizar a técnica de coloração utilizada pela ferramenta Grappolo pode ser beneficial ao Relaxmap devido a etapa de ajuste que o algoritmo possui. Qualquer movimento ótimo falso feito nos primeiros níveis da solução pode ser ajustado posteriormente pelos passos de refino. É possível que tais movimentos não ocorram, uma vez que a equação utilizada pelo algoritmo Relaxmap é diferente e pode não ser tão suscetível quanto a equação da modularidade.

Quaisquer melhorias na etapa de detecção do algoritmo terão efeito na etapa de *Fine Tune*, devido a similaridade de funcionamento das duas. Por outro lado, a etapa *Coarse Tune* precisa ser trabalhada. A divisão das comunidades em submódulos compõe o maior tempo gasto em todo a etapa de ajuste, que também é (no caso do Infomap) o maior tempo de processamento. O Relaxmap executa a divisão da comunidade em paralelo, mas a operação ainda tem um enorme impacto no consumo de tempo do *Coarse Tune*, o que significa que, para obter maiores *speedups*, pode ser necessário podar o trabalho gerado pelas divisões.

6. CONCLUSÃO

Esta pesquisa explorou o funcionamento de alguns algoritmos de detecção de comunidades e suas abordagens paralelas. A partir dos experimentos realizados observou-se que cada algoritmo apresenta vantagens e desvantagens, tornando-os atrativos para diferentes tipos de aplicações.

O algoritmo Label Propagation é atrativo para casos onde o tempo de execução é o mais importante. Tendo a possibilidade de gerar bons resultados para grafos esparsos. O algoritmo de Louvain apresenta problemas de acurácia atrelados a utilização da equação da modularidade, obtendo piores resultados para grafos que possuam comunidades pequenas. O algoritmo Infomap, por outro lado, apresenta uma boa acurácia independente da densidade do grafo, possuindo perdas apenas para grafos que tenham suas estruturas de comunidade mal definidas.

Ao analisar as versões paralelas dos algoritmos conclui-se que ainda existem pontos a serem trabalhados para que seja possível atingir bons resultados de eficiência e *speedup*. No caso da ferramenta Grappolo, o motivo do desbalanceamento de carga deve ser analisado, para que sua versão simples possa atingir melhores tempos de execução. A utilização de coloração deve ser repensada no contexto do algoritmo de Louvain. Para o algoritmo Relaxmap devem ser testadas as abordagens utilizadas pela ferramenta Grappolo, inclusive o pré-processamento de coloração, uma vez que é possível que as etapas de ajustes resolvam os problemas que podem surgir.

Os próximos passos desta pesquisa compreendem a modelagem e implementação de novas abordagens paralelas para detecção de comunidades. O algoritmo Infomap apresenta uma boa base para novas abordagens, já que, além de possuir a melhor acurácia, possui a maior complexidade de execução. Existem diversos aspectos do algoritmo Infomap que podem ser melhorados em futuras abordagens paralelas, podendo proporcionar uma competitividade de tempo de execução. Após o desenvolvimento de uma abordagem paralela em memória compartilhada que apresente bons resultados de acurácia e escalabilidade pretende-se criar uma abordagem híbrida, utilizando memória compartilhada e distribuída, com o intuito de prover uma abordagem que possa ser utilizada para os grafos de aplicações atuais de grande porte.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Bader, D. A.; Madduri, K. “Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks”. In: 2008 IEEE International Symposium on Parallel and Distributed Processing, 2008, pp. 1–12.

[2] Bae, S.-H.; Halperin, D.; West, J.; Rosvall, M.; Howe, B. “Scalable flow-based community detection for large-scale network analysis”. In: 2013 IEEE 13th International Conference on Data Mining Workshops, 2013, pp. 303–310.

[3] Bhowmik, A.; Vadhiyar, S. “Hydetect: A hybrid cpu-gpu algorithm for community detection”. In: 2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC), 2019, pp. 2–11.

[4] Blondel, V. D.; Guillaume, J.-L.; Lambiotte, R.; Lefebvre, E. “Fast unfolding of communities in large networks”, *Journal of statistical mechanics: theory and experiment*, vol. 2008–10, 2008, pp. P10008.

[5] Bóta, A.; Krész, M.; Zaválnij, B. “Adaptations of the k-means algorithm to community detection in parallel environments”. In: 2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2015, pp. 299–302.

[6] Chavarria-Miranda, D.; Halappanavar, M.; Kalyanaraman, A. “Scaling graph community detection on the tilera many-core architecture”. In: 2014 21st International Conference on High Performance Computing (HiPC), 2014, pp. 1–11.

[7] Chen, W. “Discovering communities by information diffusion”. In: 2011 Eighth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), 2011, pp. 1123–1132.

[8] Fathi, R.; Rahaman Molla, A.; Pandurangan, G. “Efficient distributed community detection in the stochastic block model”. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), 2019, pp. 409–419.

[9] Faysal, M. A. M.; Arifuzzaman, S. “Distributed community detection in large networks using an information-theoretic approach”. In: 2019 IEEE International Conference on Big Data (Big Data), 2019, pp. 4773–4782.

[10] Fortunato, S.; Hric, D. “Community detection in networks: A user guide”, *Physics reports*, vol. 659, 2016, pp. 1–44.

[11] Ghosh, S.; Halappanavar, M.; Tumeo, A.; Kalyanaraman, A.; Gebremedhin, A. H. “Scalable distributed memory community detection using vite”. In: 2018 IEEE High Performance extreme Computing Conference (HPEC), 2018, pp. 1–7.

[12] Ghosh, S.; Halappanavar, M.; Tumeo, A.; Kalyanaraman, A.; Lu, H.; Chavarrià-Miranda, D.; Khan, A.; Gebremedhin, A. “Distributed louvain algorithm for graph community detection”. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2018, pp. 885–895.

[13] Ghoshal, A. K.; Das, N.; Bhattacharjee, S.; Chakraborty, G. “A fast parallel genetic algorithm based approach for community detection in large networks”. In: 2019 11th International Conference on Communication Systems Networks (COMSNETS), 2019, pp. 95–101.

[14] Gregori, E.; Lenzini, L.; Mainardi, S. “Parallel k-clique community detection on large-scale networks”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 24–8, 2013, pp. 1651–1660.

[15] Halappanavar, M.; Lu, H.; Kalyanaraman, A.; Tumeo, A. “Scalable static and dynamic community detection using grappolo”. In: 2017 IEEE High Performance Extreme Computing Conference (HPEC), 2017, pp. 1–6.

[16] He, T.; Cai, L.; Meng, T.; Chen, L.; Deng, Z.; Cao, Z. “Parallel community detection based on distance dynamics for large-scale network”, *IEEE Access*, vol. 6, 2018, pp. 42775–42789.

[17] He, Y.; Xu, J.; Yuan, B. “Community structure analysis using label propagation and flow-based ensemble learning”. In: 2016 International Joint Conference on Neural Networks (IJCNN), 2016, pp. 720–726.

[18] Huffman, D. A. “A method for the construction of minimum-redundancy codes”, *Proceedings of the IRE*, vol. 40–9, 1952, pp. 1098–1101.

[19] Jain, A.; Nasre, R.; Ravindran, B. “Dceil: Distributed community detection with the ceil score”. In: 2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2017, pp. 146–153.

[20] Jin, D.; Yu, Z.; Jiao, P.; Pan, S.; He, D.; Wu, J.; Yu, P.; Zhang, W. “A survey of community detection approaches: From statistical modeling to deep learning”, *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[21] Joldos, M.; Technical, C. C. “A parallel evolutionary approach to community detection in complex networks”. In: 2017 13th IEEE International Conference on Intelligent Computer Communication and Processing (ICCP), 2017, pp. 247–254.

[22] Kao, E.; Gadepally, V.; Hurley, M.; Jones, M.; Kepner, J.; Mohindra, S.; Monticciolo, P.; Reuther, A.; Samsi, S.; Song, W.; Staheli, D.; Smith, S. “Streaming graph challenge: Stochastic block partition”. In: 2017 IEEE High Performance Extreme Computing Conference (HPEC), 2017, pp. 1–12.

[23] Khlopotine, A. B.; Sathanur, A. V.; Jandhyala, V. “Optimized parallel label propagation based community detection on the intel(r) xeon phi(tm) architecture”. In: 2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2015, pp. 9–16.

[24] Lancichinetti, A.; Fortunato, S. “Community detection algorithms: a comparative analysis”, *Physical review E*, vol. 80–5, 2009, pp. 056117.

[25] Lancichinetti, A.; Fortunato, S.; Radicchi, F. “Benchmark graphs for testing community detection algorithms”, *Physical review E*, vol. 78–4, 2008, pp. 046110.

[26] Leskovec, J.; Krevl, A. “SNAP Datasets: Stanford large network dataset collection”. Capturado em: http://snap.stanford.edu/data, Março 2022.

[27] Liang, S.; Li, H.; Gong, M.; Wu, Y.; Zhu, Y. “Distributed multi-objective community detection in large-scale and complex networks”. In: 2019 15th International Conference on Computational Intelligence and Security (CIS), 2019, pp. 201–205.

[28] Liu, J.; Wei, Z. “Community detection based on graph dynamical systems with asynchronous runs”. In: 2014 Second International Symposium on Computing and Networking, 2014, pp. 463–469.

[29] Liu, X.; Firoz, J. S.; Zalewski, M.; Halappanavar, M.; Barker, K. J.; Lumsdaine, A.; Gebremedhin, A. H. “Distributed direction-optimizing label propagation for community detection”. In: 2019 IEEE High Performance Extreme Computing Conference (HPEC), 2019, pp. 1–6.

[30] Lu, H.; Halappanavar, M.; Kalyanaraman, A. “Parallel heuristics for scalable community detection”, *Parallel Computing*, vol. 47, 2015, pp. 19–37.

[31] Lu, H.; Halappanavar, M.; Kalyanaraman, A.; Choudhury, S. “Parallel heuristics for scalable community detection”. In: 2014 IEEE International Parallel Distributed Processing Symposium Workshops, 2014, pp. 1374–1385.

[32] Lu, L.; Gu, Y.; Grossman, R. “dmaximalcliques: A distributed algorithm for enumerating all maximal cliques and maximal clique distribution”. In: 2010 IEEE International Conference on Data Mining Workshops, 2010, pp. 1320–1327.

[33] Mahmoud, H.; Masulli, F.; Rovetta, S.; Russo, G. “Community detection in protein-protein interaction networks using spectral and graph approaches”. In: International meeting on computational intelligence methods for bioinformatics and biostatistics, 2013, pp. 62–75.

[34] McSherry, F.; Isard, M.; Murray, D. G. “Scalability! but at what {COST}?” In: 15th Workshop on Hot Topics in Operating Systems (HotOS XV), 2015.

[35] Mothe, J.; Mkhitaryan, K.; Haroutunian, M. “Community detection: Comparison of state of the art algorithms”. In: 2017 Computer Science and Information Technologies (CSIT), 2017, pp. 125–129.

[36] Naik, D.; Ramesh, D.; Gandomi, A. H.; Gorojanam, N. B. “Parallel and distributed paradigms for community detection in social networks: A methodological review”, *Expert Systems with Applications*, vol. 187, 2022, pp. 115956.

[37] Newman, M. E. J.; Girvan, M. “Finding and evaluating community structure in networks”, *Phys. Rev. E*, vol. 69, Feb 2004, pp. 026113.

[38] Ovelgönne, M. “Distributed community detection in web-scale networks”. In: 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2013), 2013, pp. 66–73.

[39] Panyala, A.; Subasi, O.; Halappanavar, M.; Kalyanaraman, A.; Chavarria-Miranda, D.; Krishnamoorthy, S. “Approximate computing techniques for iterative graph algorithms”. In: 2017 IEEE 24th International Conference on High Performance Computing (HiPC), 2017, pp. 23–32.

[40] Qiao, S.; Han, N.; Gao, Y.; Li, R.-H.; Huang, J.; Guo, J.; Gutierrez, L. A.; Wu, X. “A fast parallel community discovery model on complex networks through approximate optimization”, *IEEE Transactions on Knowledge and Data Engineering*, vol. 30–9, 2018, pp. 1638–1651.

[41] Resende, H.; Fazenda, A. L.; Quiles, M. G. “Parallel algorithm for dynamic community detection”. In: 2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), 2017, pp. 55–60.

[42] Rosvall, M.; Axelsson, D.; Bergstrom, C. T. “The map equation”, *The European Physical Journal Special Topics*, vol. 178–1, 2009, pp. 13–23.

[43] Salathé, M.; Jones, J. H. “Dynamics and control of diseases in networks with community structure”, *PLoS computational biology*, vol. 6–4, 2010, pp. e1000736.

[44] Sarswat, A.; Guddeti, R. M. R. “A novel overlapping community detection using parallel cfm and sequential nash equilibrium”. In: 2018 10th International Conference on Communication Systems Networks (COMSNETS), 2018, pp. 649–654.

[45] Sattar, N. S.; Arifuzzaman, S. “Parallelizing louvain algorithm: Distributed memory challenges”. In: 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech), 2018, pp. 695–701.

[46] Satuluri, V.; Wu, Y.; Zheng, X.; Qian, Y.; Wichers, B.; Dai, Q.; Tang, G. M.; Jiang, J.; Lin, J. “Simclusters: Community-based representations for heterogeneous recommendations at twitter”. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2020, pp. 3183–3193.

[47] Sedighpour, N.; Bagheri, A. “Paslpa - overlapping community detection in massive real networks using apache spark”. In: 2018 9th International Symposium on Telecommunications (IST), 2018, pp. 233–240.

[48] Shi, J.; Dhulipala, L.; Eisenstat, D.; Łacki, J.; Mirrokni, V. “Scalable community detection via parallel correlation clustering”, *arXiv preprint arXiv:2108.01731*, 2021.

[49] Slota, G. M.; Root, C.; Devine, K.; Madduri, K.; Rajamanickam, S. “Scalable, multi-constraint, complex-objective graph partitioning”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 31–12, 2020, pp. 2789–2801.

[50] Soliman, A.; Rahimian, F.; Girdzijauskas, S. “Stad: Stateful diffusion for linear time community detection”. In: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), 2018, pp. 1074–1085.

[51] Soman, J.; Narang, A. “Fast community detection algorithm with gpus and multicore architectures”. In: 2011 IEEE International Parallel Distributed Processing Symposium, 2011, pp. 568–579.

[52] Song, Q.; Li, B.; Yu, W.; Li, J.; Shi, B. “Nslpa: A node similarity based label propagation algorithm for real-time community detection”. In: 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, 2014, pp. 896–901.

[53] Souravlas, S.; Sifaleras, A.; Katsavounis, S. “A parallel algorithm for community detection in social networks, based on path analysis and threaded binary trees”, *IEEE Access*, vol. 7, 2019, pp. 20499–20519.

[54] Souravlas, S.; Sifaleras, A.; Katsavounis, S. “Hybrid cpu-gpu community detection in weighted networks”, *IEEE Access*, vol. 8, 2020, pp. 57527–57551.

[55] Staudt, C. L.; Meyerhenke, H. “Engineering parallel algorithms for community detection in massive networks”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 27–1, 2016, pp. 171–184.

[56] Sun, H.; Jie, W.; Sauer, C.; Ma, S.; Han, G.; Xing, W. “A self-organizing algorithm for community structure analysis in complex networks”. In: 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2016, pp. 5–12.

[57] Vo, N.; Lee, K.; Tran, T. “Mrattractor: Detecting communities from large-scale graphs”. In: 2017 IEEE International Conference on Big Data (Big Data), 2017, pp. 797–806.

[58] Wickramaarachchi, C.; Frincu, M.; Small, P.; Prasanna, V. K. “Fast parallel algorithm for unfolding of communities in large graphs”. In: 2014 IEEE High Performance Extreme Computing Conference (HPEC), 2014, pp. 1–6.

[59] Wu, B.; Zhang, C.; Guo, Q. “A parallel network community detection algorithm based on distance dynamics”. In: 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), 2017, pp. 819–826.

[60] Xu, J.; Fu, L.; Gan, X.; Zhu, B. “Distributed community detection on overlapping stochastic block model”. In: 2020 International Conference on Wireless Communications and Signal Processing (WCSP), 2020, pp. 201–206.

[61] Xu, Y.; Cheng, J.; Fu, A. W.-C. “Distributed maximal clique computation and management”, *IEEE Transactions on Services Computing*, vol. 9–1, 2016, pp. 110–122.

[62] Zalmout, N.; Ghanem, M. “Multidimensional community detection in twitter”. In: 8th International Conference for Internet Technology and Secured Transactions (ICITST-2013), 2013, pp. 83–88.

[63] Zeng, J.; Yu, H. “Parallel modularity-based community detection on large-scale graphs”. In: 2015 IEEE International Conference on Cluster Computing, 2015, pp. 1–10.

[64] Zeng, J.; Yu, H. “A scalable distributed louvain algorithm for large-scale graph community detection”. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER), 2018, pp. 268–278.

[65] Zhou, Q.; Cai, S.-M.; Zhang, Y.-C. “Parallel heuristic community detection method based on node similarity”, *IEEE Access*, vol. 7, 2019, pp. 184145–184159.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br