Analisando a Escalabilidade e a Acurácia de Implementações Paralelas e Distribuídas para a Detecção de Comunidades em Grafos

Gabriel G. Santos[1], **César A. F. De Rose**[1], **Kartik Lakhotia**[2]

[1]Escola Politécnica – PUCRS
Porto Alegre – RS – Brasil

[2]Intel Labs
Santa Clara – CA – E.U.A.

gabriel.giordani@edu.pucrs.br, cesar.derose@pucrs.br,

kartik.lakhotia@intel.com

Resumo. *Detecção de comunidades em grafos é um tipo de análise amplamente utilizada por aplicações de diversas áreas do conhecimento. Com o crescente aumento do volume de dados surge a necessidade de implementações paralelas e distribuídas para que o tempo de processamento não aumente a ponto de limitar sua aplicabilidade. Este estudo analisa a escalabilidade e a acurácia dos métodos para detecção de comunidades mais utilizados atualmente em ambientes paralelos e distribuídos. A partir de uma análise detalhada do impacto que a comunicação entre os processos teve no tempo de execução de cada implementação são feitas recomendações em relação aos algoritmos mais promissores em relação a melhoria da escalabilidade nestes ambientes.*

1. Introdução

Uma comunidade dentro de um grafo pode ser definida como um subgrafo que possui mais conexões internas do que externas. O processo de encontrar a estrutura de comunidades de um grafo é chamado de detecção de comunidades. Diversos métodos para detectar comunidades foram desenvolvidos, envolvendo diferentes necessidades de quantidade de informação a respeito do grafo [Fortunato and Hric 2016]. Existem diversas aplicações em que este tipo de análise pode ser utilizada. Ela é encontrada, principalmente, em sistemas de recomendação, análise de redes sociais e aplicações de bioinformática como análise de cadeias de proteínas.

Devido ao aumento do volume de dados de diversas aplicações, existe uma necessidade de implementações paralelas para algoritmos de grafos, para que o tempo de processamento não aumente a ponto de limitar a aplicabilidade destas técnicas. Em muitos casos, como o de aplicações de redes sociais, os grafos atuais já não podem ser processados em apenas uma máquina devido a limitações de memória, aumentando a necessidade de implementações distribuídas.

Devido a natureza do problema de detecção de comunidades, se faz necessário o compartilhamento de informações sobre a estrutura de todo o grafo durante o processamento de cada vértice, de forma que implementações paralelas em memória compartilhada (multiprocessadores) e distribuídas que trocam mensagens (multicomputadores)

precisam utilizar otimizações para evitar comunicações excessivas e seções críticas entre os processos envolvidos. Em função disto, existe uma perda inevitável da qualidade da solução, que irá variar de acordo com a quantidade e tipo de otimizações utilizadas.

Este estudo propõe uma análise das principais implementações paralelas e distribuídas para detecção de comunidades com o objetivo de verificar a eficiência de cada otimização utilizada e o impacto resultante na qualidade da solução. Como carga de entrada foram selecionados grafos de aplicações reais com diferentes tamanhos e densidades, obtidos em repositórios públicos, como SNAP e Network Repository e amplamente usados em trabalhos relacionados. Tendo esses pontos em vista, as principais contribuições deste estudo são:

- Apresentação das principais soluções paralelas e distribuídas para detecção de comunidades encontradas na literatura detalhando as otimizações utilizadas por cada uma;
- Análise da escalabilidade e acurácia de cada implementação em um multiprocessador (multicore) e em um multicomputador (cluster) usando grafos reais de diferentes tamanhos e densidades;
- Análise detalhada do impacto que a comunicação entre os processos teve no tempo de execução de cada implementação e recomendações em relação aos algoritmos e otimizações mais promissores em relação a escalabilidade nestes ambientes.

2. Referencial Teórico

Existem diversos métodos de detecção de comunidades [Fortunato and Hric 2016] e diversas implementações paralelas e distribuídas de cada um destes métodos. Para este estudo foram selecionados os três métodos mais estudados na literatura e as suas implementações *open-source* disponíveis em repositórios públicos. Todas as implementações utilizam a linguagem C++ e as bibliotecas OpenMP e MPI para processamento paralelo e distribuído. A seguir, cada método e suas respectivas implementações são apresentados em detalhe.

2.1. Método Louvain

O método de Louvain [Blondel et al. 2008] propõe a otimização da função da modularidade [Newman and Girvan 2004], que quantifica a conectividade de um conjunto de sub-grafos. O algoritmo realiza a otimização da modularidade em múltiplos níveis do grafo. Inicialmente, cada vértice é considerado como pertencente a sua própria comunidade. Em seguida, cada vértice, em uma ordem aleatória, verifica se uma mudança para uma comunidade de um de seus vizinhos proverá um melhor valor de modularidade para o grafo. O vértice, então, é movido para a comunidade que proverá o melhor valor de modularidade e não se move caso não seja possível uma melhora. Quando não existem mais movimentos que melhorem a modularidade do grafo cada comunidade é convertida em um super-vértice e um novo grafo é formado. O algoritmo, então, é aplicado novamente neste novo grafo. Quando um grafo não é capaz de gerar uma melhora no valor da modularidade o algoritmo encerra.

Lu et al. [Lu et al. 2015] propuseram uma implementação do algoritmo Louvain chamada Grappolo que utiliza paralelismo em memória compartilhada e aplica otimizações referentes a ordem de processamento dos vértices e na independência de

seu processamento. Cada *thread* OpenMP é responsável por processar um conjunto de vértices, realizando todos os cálculos e movimentos. Como efetuar movimentos implica na alteração de duas comunidades, é necessário utilizar uma forma de sincronismo entre as *threads*. Os autores evitam a utilização de uma seção crítica para efetuar movimentos trocando-a por um conjunto de operações *atomic*. Para evitar conflitos entre *threads* o grafo é colorido e cada conjunto de cores é processado separadamente.

Gosh et al. [Ghosh et al. 2018] propuseram uma implementação híbrida MPI/OpenMP do algoritmo de Louvain chamada Vite, que roda tanto em multiprocessadores quanto em multicomputadores. O grafo é dividido entre as máquinas do multicomputador utilizando processos MPI, que por sua vez chamam diretivas OpenMP para explorar os vários núcleos de cada máquina. Além disso, duas otimizações são utilizadas para acelerar a convergência da solução. A primeira otimização se chama *Threshold Cycling* e pode ser utilizada durante todo o processamento. Ela consiste em modificar o valor da condição de parada do algoritmo de forma iterativa, juntamente com as construções dos grafos. Os autores utilizam um valor maior para as primeiras iterações do algoritmo, onde o grafo ainda é muito grande. Conforme comunidades são formadas e o grafo é transformado em um novo grafo e, portanto, diminuindo de magnitude, a condição de parada se torna menor. Desta forma, computações excessivas nas iterações iniciais são evitadas. A segunda otimização se chama *Early Termination*. Ela define uma probabilidade de um vértice deixar de ser elegível para ser processado. Esta probabilidade aumenta conforme o vértice permanece na mesma comunidade, até que ele se torna inativo, impedindo que seja processado novamente na iteração atual. Desta forma, o processamento excessivo de vértices que não irão se mover é cortado.

2.2. Método Infomap

O método Infomap [Rosvall et al. 2009] introduz uma versão otimizada da Map Equation criada pelos próprios autores para detectar comunidades. A equação se baseia em uma caminhada aleatória, calculando qual seria a menor quantidade de bits possível para descrever um caminho pelo grafo. Comunidades são utilizadas como prefixos para permitir a reutilização de IDs de vértices. A equação, então, otimiza o *Minimum Description Length* (MDL) de um grafo. O método possui uma estrutura similar ao Louvain, mas apresenta mais fases para prover um refino da solução. Inicialmente, é realizada uma detecção seguindo os mesmos passos do algoritmo de Louvain onde uma solução inicial é criada e então um novo grafo é gerado. O algoritmo é aplicado novamente nesse grafo e assim sucessivamente até que não ocorram melhoras na Map Equation. Após essa fase inicial o algoritmo executa dois tipos de refino sobre a solução encontrada. Primeiro, é permitido que vértices se movam individualmente, como na primeira fase, mas suas comunidades permanecem sendo as mesmas que foram definidas anteriormente. Isto permite que o algoritmo inicial seja re-computado sobre o grafo, corrigindo movimentos individuais. O segundo refino age sobre cada comunidade, dividindo-as em sub-comunidades e aplicando o algoritmo inicial em grupos de vértices.

Bae et al. [Bae et al. 2013] propuseram uma implementação paralela para memória compartilhada do método Infomap usando OpenMP. Os autores a nomearam de Relaxmap, pois são propostos relaxamentos de *thresholds* e computações em relação ao algoritmo original. A principal proposta é a aceleração da convergência do método Infomap por meio de *thresholds* mais altos para as condições de parada do algoritmo. Além

disso, o processamento de vértices é dividido entre múltiplas *threads*, que realizam os cálculos de movimento de forma independente e utilizam uma seção crítica para efetuar os melhores movimentos encontrados.

Faysal et al. [Faysal et al. 2021] propuseram uma implementação híbrida MPI/OpenMP do método Infomap chamada HyPC-Map, que roda tanto em multiprocessadores quanto em multicomputadores. O grafo é dividido entre as máquinas do multicomputador utilizando processos MPI, que por sua vez chamam diretivas OpenMP para explorar os vários núcleos de cada máquina, seguindo o mesmo modelo do Relaxmap. Os processos MPI se comunicam após cada iteração, enviando o posicionamento final de seus vértices em comunidades.

2.3. Método Label Propagation

O método Label Propagation apresenta maior simplicidade em relação aos métodos anteriores. Sua ideia principal é atribuir uma comunidade única para cada vértice em um subconjunto de vértices do grafo e fazer com que eles propaguem suas comunidades para seus vizinhos. Vértices mudam de comunidade baseado na quantidade de vizinhos que pertencem a outra comunidade. O algoritmo realiza diversas iterações de propagação até que seja atingida uma convergência nas atribuições de comunidade de cada vértice.

Devido a simplicidade do método, existem diversas implementações paralelas para memória compartilhada do mesmo. Para este estudo foi selecionada a implementação do *toolkit* NetworKit [Staudt et al. 2016], chamada PLP [Staudt and Meyerhenke 2016], pois a mesma é implementada na mesma linguagem das outras implementações abordadas (C++) e utiliza a mesma biblioteca de paralelização (OpenMP). O PLP, assim como o algoritmo original, apresenta uma implementação bastante simples. Os vértices do grafo são distribuídos entre múltiplas *threads*, que propagam as comunidades dos vértices.

O CDLP é uma implementação do método Label Propagation do *benchmark* LDBC Graph Analytics [Iosup et al. 2016] . A implementação faz pequenas alterações no algoritmo original para torná-lo determinístico. Para este estudo foi utilizada a versão do CDLP disponível na biblioteca GRAPE [Fan et al. 2017], que é implementada em C++ e MPI/OpenMP e roda em multicomputadores.

3. Experimentos

Para verificar a escalabilidade das implementações apresentadas na Seção 2 e os impactos das otimizações na qualidade das comunidades detectadas, foram realizados os seguintes experimentos:

- **Qualidade:** Comparação dos resultados obtidos a partir dos algoritmos sequenciais com os resultados de cada implementação utilizando suas diferentes otimizações.
- ***Speed-Up:*** Comparação do tempo de execução dos algoritmos sequenciais com as implementações paralelas e distribuídas utilizando diferentes números de processos.
- **Comunicação:** Análise do impacto que a comunicação entre os processos teve no tempo de execução.

Todos os experimentos foram realizados em um cluster com quatro máquinas multicore Dell EMC PowerEdge R740 com duas soquetes, cada uma com um processador

Xeon Gold 5118 (12 Cores de 2.30 GHz e suporte a hyperthreading de grau 2) totalizando 24 cores capazes de executar 48 *threads* por máquina. Cada core físico possui 12 Mbytes de cache L2 e uma cache L3 compartilhada de 16.5 Mbytes por processador, com 192 Gbytes de memória principal por máquina. As máquinas do cluster estão interligadas por 4 redes Gigabit-Ethernet cuja capacidade agregada é utilizada pelo MPI, atingindo uma vazão de até $3,2$Gbits/s e latência de 35 micro segundos. Os tempos sequenciais foram medidos em um núcleo desta máquina, as execuções das implementações paralelas com memória compartilhada (OpenMP) em uma máquina deste cluster, e as implementações distribuídas baseadas em troca de mensagem (MPI) utilizando todas as máquinas do cluster. Todos os resultados apresentados nesta Seção são referentes às médias de 5 execuções, utilizando um intervalo de confiança de 90%.

Para os experimentos foram selecionados sete grafos de aplicações reais. A Tabela 1 apresenta os grafos e o tempo de execução sequencial em segundos nos três métodos analisados neste estudo. Os grafos foram obtidos nos bancos de dados do SNAP [Leskovec and Krevl 2014] e do NetworkRepository [Rossi and Ahmed 2015]. Os grafos wiki-topcats e web-uk-2005 são, respectivamente, topologias da Wikipédia e de sites do Reino Unido. O restante são redes sociais de diferentes países.

Tabela 1. Características dos grafos reais utilizados neste estudo e seus tempos de execução sequencial

Grafo	Vértices	Arestas	Tempo de Execução (s)		
			Louvain	Infomap	Label Propagation
wiki-topcats	1.791.489	28.511.807	79,2	454,08	3,6
soc-pokec-relationships	1.632.803	30.622.564	69,2	631,57	3,95
com-livejournal	3.997.962	34.681.189	323,8	721,79	8
soc-LiveJournal1	4.847.571	68.993.773	266,6	1.269,99	8,64
com-orkut	3.072.441	117.185.083	622,6	2.080,28	19,96
soc-twitter-2010	21.297.772	265.025.809	1.079,6	10.247,96	66,29
web-uk-2005	39.459.925	936.364.282	1.783,2	28.791,7	98,16

3.1. Qualidade da Solução

Inicialmente, foram executados experimentos relativos a qualidade das comunidades geradas por cada implementação. Para tais experimentos foram criados grafos sintéticos com o *benchmark* LFR [Lancichinetti et al. 2008]. O *benchmark* permite a criação de grafos com diferentes tamanhos e distribuições de arestas e comunidades, provendo um arquivo de *ground-truth* das comunidades do grafo gerado. Os grafos criados para estes experimentos seguem os parâmetros propostos pelos autores.

Para medir a acurácia de cada algoritmo, eles foram executados com os grafos sintéticos e seus resultados foram comparados com o *ground-truth* seguindo a métrica de *precision-recall* proposta pelo Graph Challange [Kao et al. 2017]. A Figura 1 mostra os valores de F-score para cada algoritmo sequencial em função do percentual de arestas dos vértices de cada comunidade que estão conectadas a vértices de outras comunidades, onde $0,1$ representa comunidades bem definidas (10% de arestas fora da comunidade) e valores iguais ou superiores a $0,5$ representam comunidades mal definidas. O algoritmo Infomap apresenta a melhor acurácia dentro dos grafos gerados, apenas perdendo a precisão em casos onde comunidades são mal formadas. O algoritmo Louvain apresenta resultados piores em todos os casos. O algoritmo Label Propagation apresenta os piores resultados, sendo capaz de ultrapassar 50% de acurácia apenas em um caso.

Para medir o impacto das implementações paralelas e distribuídas na qualidade da solução foram utilizadas as métricas de otimização de cada algoritmo. Para as implementações Grappolo e Vite foi verificado a diferença produzida na modularidade. Para as implementações Relaxmap e HyPC-Map foi verificado o MDL final produzido. Não foi possível executar o grafo web-uk-2005 com a implementação HyPC-Map devido a limites de tempo de execução atingidos durante a leitura do arquivo do grafo. As versões paralelas tendem a apresentar pequenas diferenças com relação aos resultados das execuções sequenciais, sendo o Relaxmap a implementação com maior diferença no resultado. As versões distribuídas, por sua vez, apresentam diferenças maiores, mas que ainda estão dentro de um valor aceitável, sendo o HyPC-Map a implementação com maior diferença no resultado.

3.2. Fator de Aceleração (*Speed-Up*)

Para verificar a escalabilidade das implementações foi calculado o fator de aceleração (*Speed-Up*) onde cada implementação foi comparada com a execução sequencial dos algoritmos originais mantendo a carga e aumentando o número de processos trabalhando em paralelo (escalabilidade forte). Para o ambiente de memória compartilhada foram realizadas execuções com 6, 12, 24 e 48 *threads* geradas pelo OpenMP, onde a última configuração utiliza *hyperthreading*. Para o ambiente distribuído, como todas as implementações são híbridas (MPI/OpenMP), foram utilizadas diversas configurações envolvendo número de processos MPI e número de *threads* por processo. Dentre todas as configurações testadas, é apresentada a com melhores resultados de *Speed-Up* gerais, que consiste na utilização de um processo MPI por máquina do cluster e 24 *threads* por processo em cada máquina.

Para as implementações em memória compartilhada foram calculadas as médias geométricas do fator de aceleração atingido nas execuções de cada grafo em cada configuração de *threads*. As curvas de aceleração produzidas podem ser vistas na Figura 2. A baixa escalabilidade da implementação PLP pode ser atribuída ao fato do algoritmo Label Propagation ser muito rápido, deixando pouco espaço para ganhos de tempo de execução em ambientes paralelos. A implementação Grappolo apresenta resultados ruins de *Speed-Up* quando executada sem suas otimizações. No entanto, utilizar a otimização de coloração do grafo sempre gera ganhos de *Speed-Up*, principalmente para execuções com 6 *threads*, onde eficiências paralelas de 100% são atingidas. O Relaxmap, embora apresente os melhores resultados gerais de *Speed-Up*, é capaz de atingir no máximo 50% de eficiência paralela para alguns grafos em 24 *threads*, mesmo sendo a implementação que mais sacrifica o resultado final. O método Infomap é o mais lento entre os abordados neste estudo, fazendo com que exista mais espaço para ganhos de tempo de execução do que nos outros algoritmos abordados.

Para as implementações distribuídas, os grafos são explorados de forma singular, já que suas diferenças de tamanho podem afetar a proporção de tempo gasto com troca de mensagens, diminuindo o valor do *Speed-Up*. Devido a limitação de páginas, são mostrados apenas os quatro grafos que possuem os maiores tempos de execução sequencial. Assim como nos experimentos de acurácia, não foi possível executar o grafo web-uk-2005 para a implementação HyPC-Map. A Figura 3 apresenta o *Speed-Up* das implementações distribuídas (multicomputadores) em função do número de *threads* geradas pelo OpenMP, onde um processo MPI é utilizado para cada 24 *threads*. As legendas da implementação

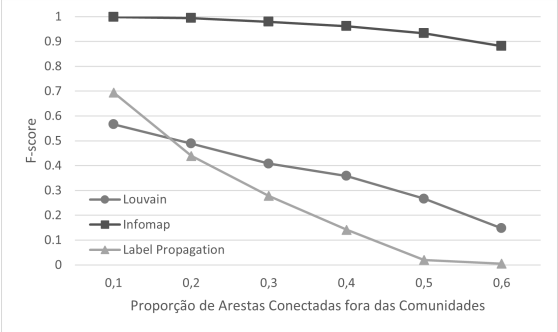

Figura 1. Acurácia (F-score) da implementação sequencial de cada método.

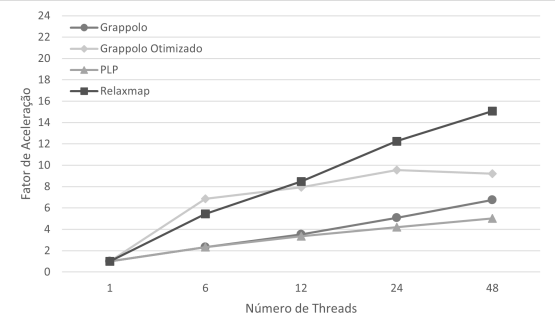

Figura 2. Fator de aceleração (*Speed-Up*) das implementações OpenMP em uma máquina do cluster.

Vite se referem a otimização utilizada: sem adição de otimizações (Vite), *Threshold Cycling* (Vite TC) e *Early Termination* (Vite ET). Também foram executados experimentos com as duas otimizações ativadas, mas os resultados obtidos foram similares a utilização de apenas *Threshold Cycling*, portanto, eles não são demonstrados.

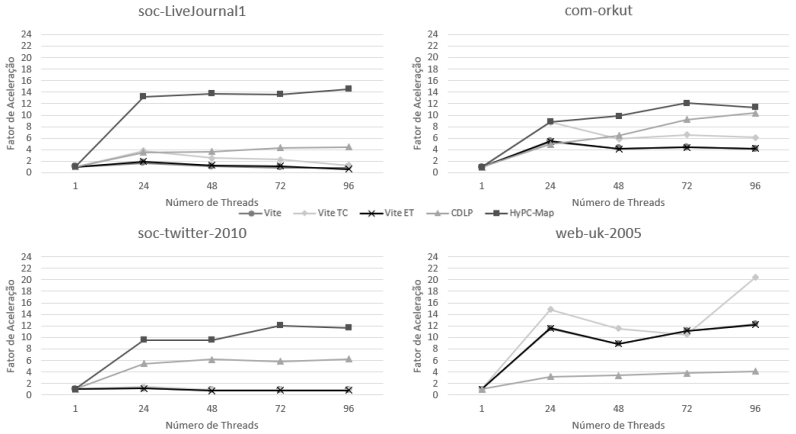

Figura 3. Fator de aceleração (*Speed-Up*) das implementações em memória distribuída (multicomputadores).

Para a implementação Vite ocorre uma perda de *Speed-Up* com o aumento do número de máquinas utilizadas para quase todos os grafos.Apenas execuções com quatro máquinas do grafo web-uk-2005 são capazes de superar os ganhos de execuções em memória compartilhada. É evidente a existência de problemas de escalabilidade, embora não seja possível atribuí-los a algo específico sem a verificação do desempenho de cada componente da implementação. A implementação HyPC-Map apresenta ganhos em relação a execuções em memória compartilhada, mas tais ganhos são mínimos com o aumento de máquinas. Para alguns grafos algumas configurações de máquinas já apresentam quedas de *Speed-Up*, indicando que a implementação pode estar em seu limite de escalabilidade dentro do ambiente utilizado. Em função disso é evidente o baico desempenho em um ambiente distribuído. A implementação CDLP apresenta ganhos mínimos de *Speed-Up* em todos os grafos, possuindo quase o mesmo tempo de execução independente do número de processos MPI utilizados. Isto, no entanto, pode ser devido ao baixo tempo de execução do método Label Propagation.

As implementações, no geral, não apresentam bons valores de *Speed-Up* e eficiência paralela, sendo piores em ambientes distribuídos. No caso das implementações do Label Propagation tais resultados são esperados, uma vez que o algoritmo sequencial é muito rápido, deixando pouco espaço para ganhos de tempo de execução. No entanto, para as outras implementações, existe a possibilidade de ganhos de desempenho.

3.3. Impactos da Comunicação

Como não é possível descobrir o motivo dos problemas de escalabilidade apenas com comparações de tempo de execução, também foram realizadas medidas com relação ao impacto da comunicação entre os processos MPI das implementações distribuídas Vite e HyPC-Map. A implementação CDLP não foi abordada nestes testes pois executa em menos de um minuto para todos os grafos abordados, tornando investigações mais profundas irrelevantes.

Ambas implementações seguem um modelo de comunicação síncrona, onde cada processo MPI envia dados para todos os outros processos e apenas sai da seção de comunicação após receber dados de todos os outros processos. A implementação Vite possui duas seções de comunicação dentro do processo de detecção de comunidades. A primeira ocorre antes de toda fase do Louvain (computar e realizar movimentos), onde os processos MPI verificam se as estruturas de comunidades estão coerentes entre todos os processos. A segunda ocorre ao final de toda iteração do algoritmo, onde processos enviam informações relevantes a vértices de borda que estão divididos entre dois processos. Portanto, ambas seções de comunicação foram consideradas para a verificação do impacto da comunicação na implementação Vite.

A implementação HyPC-Map possui apenas uma seção de comunicação, que ocorre ao final de cada fase do algoritmo e consiste no envio do posicionamento final de todos os vértices em suas respectivas comunidades. Após a comunicação, cada processo atualiza seu grafo, resolvendo conflitos de posicionamento de vértices a partir de heurísticas, sem se comunicar com outros processos.

A Figura 4 mostra o tempo relativo da comunicação de cada implementação dentro da execução dos quatro maiores grafos. A comunicação da implementação Vite representa mais tempo relativo de execução do que a comunicação da implementação HyPC-Map. Isto é esperado, uma vez que o Vite realiza comunicações com mais frequência, enquanto o HyPC-Map possui apenas uma seção para cada fase do algoritmo.

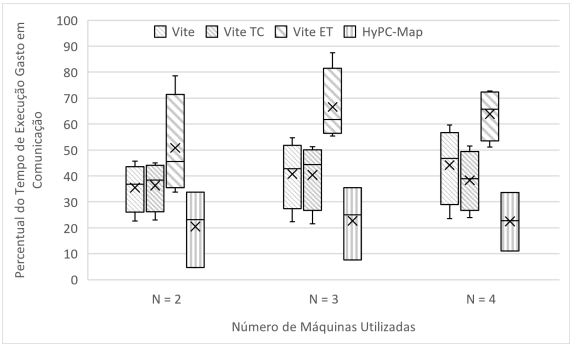

Figura 4. Percentual do tempo de execução gasto nas seções de comunicação das implementações Vite e HyPC-Map.

No caso do Vite, o grande *overhead* de comunicação é claramente um dos motivos de sua falta de escalabilidade. Em alguns casos, a comunicação excede 50% do tempo de execução, enquanto o restante do tempo é composto pela detecção de comunidades e a criação de novos grafos. Como a implementação foca suas otimizações na convergência do método Louvain e não na comunicação de informações referentes às comunidades, o desempenho paralelo do Vite torna-se dependente da rede.

O HyPC-Map, por possuir menos impacto na comunicação, possui problemas de escalabilidade provenientes, principalmente, do processamento de comunidades feito por meio do OpenMP (memória compartilhada). Como foi visto na Seção 2, ambas implementações do método Infomap utilizam abordagens simples para o paralelismo com OpenMP. Não são utilizadas otimizações para a convergência do algoritmo ou para o sincronismo entre as *threads*. Por apresentar um alto tempo de execução no processamento em OpenMP, o HyPC-Map apresenta um baixo impacto de comunicação.

4. Discussão

A partir dos experimentos descritos na Seção 3 é possível afirmar que as implementações abordadas possuem problemas de escalabilidade em ambientes paralelos e distribuídos. Em multiprocessadores a eficiência não passou de 50% quando utilizados todos os núcleos físicos, o que pode ser considerado baixo para este tipo de máquina. Em multicomputadores, a escalabilidade das implementações é ainda pior, gerando, em alguns casos, valores de *Speed-Up* mais baixos do que em ambientes de memória compartilhada. Nesta Seção, são abordados os problemas de cada implementação, assim como os casos em que elas devem ser utilizadas.

O método Louvain apresenta uma acurácia razoável, sendo melhor para grafos mais densos. Suas implementações paralela e distribuída possuem resultados com pouca variação de modularidade, mesmo com a utilização de diversas otimizações para acelerar a convergência do algoritmo. As implementações sofrem de problemas de escalabilidade principalmente no ambiente distribuído, que não apresentou ganhos em relação a execuções em memória compartilhada com a exceção do grafo soc-uk-2005. O motivo do baixo desempenho está ligado, principalmente, às fases de comunicação da implementação Vite, que em alguns casos representam mais de 50% do tempo de execução. Dados os resultados deste estudo, a utilização de uma implementação distribuída do algoritmo Louvain é recomendada apenas quando o grafo que deva ser processado não seja armazenável em apenas uma máquina.

O método Infomap apresenta a melhor acurácia dentre os algoritmos abordados, embora também apresente os maiores tempos de execução sequencial. As implementações abordadas, Relaxmap e HyPC-Map, apresentam perdas de acurácia em relação ao algoritmo sequencial devido aos relaxamentos mencionados na Seção 2, sendo a implementação distribuída, HyPC-Map, a com os piores resultados. Ambas implementações não possuem boas curvas de *Speed-Up*, especialmente no ambiente distribuído, onde a adição de máquinas apresenta ganhos mínimos. A partir da medição do impacto que a comunicação entre processos MPI tem sobre o tempo de execução do HyPC-Map, conclui-se que o principal impeditivo para maiores ganhos de desempenho se encontra no processamento de movimentos de vértices, que é feito em memória compartilhada por meio de diretivas OpenMP. As implementações do Infomap são as

únicas abordadas neste estudo que utilizam uma seção crítica para atualizar informações de vértices e comunidades ao invés de diretivas *atomic*, que proporcionam um *overhead* de sincronização muito menor, podendo ser um dos motivos do baixo desempenho em memória compartilhada do Relaxmap e do HyPC-Map.

O método Label Propagation é um caso a parte dos outros abordados neste estudo devido a seu baixo tempo de execução sequencial, o que torna altos valores de *Speed-Up* difíceis de serem atingidos. Sua aplicabilidade encontra-se em casos onde velocidade é o fator mais importante, como em particionamento de grafos, uma vez que a qualidade das comunidades geradas é muito inferior a outros algoritmos. Dentro deste contexto, a utilização de implementações paralelas e distribuídas traz benefícios, uma vez que qualquer ganho de tempo de execução é muito bem-vindo.

Dados os resultados apresentados neste estudo, o método que possui o maior potencial para futuras otimizações paralelas e distribuídas é o Infomap. Ele apresenta a melhor qualidade de comunidades detectadas e também o maior tempo de execução. No entanto, não apresenta otimizações em suas implementações paralela e distribuída, que utilizam apenas modelos simples de paralelização. Atualmente, a implementação HyPC-Map não apresenta altas taxas de comunicação entre processos MPI, ao contrário da implementação Vite. Devido a isso, recomenda-se que sejam realizadas otimizações em relação a convergência do algoritmo e a sincronização das *threads* do OpenMP. Outros pontos que podem ser otimizados envolvem a distribuição de trabalho e o sincronismo entre *threads* e processos MPI, já que o modelo de comunicação utilizado pelo HyPC-Map é síncrono, e exige que processos MPI aguardem que todos os outros processos terminem suas partes do grafo, gerando uma barreira para a progressão do processamento das comunidades.

5. Trabalhos Relacionados

Agreste et al. [Agreste et al. 2016] propuseram uma comparação entre algoritmos sequenciais de detecção de comunidades em termos de acurácia e escalabilidade. Os autores focam em métodos que são capazes de processar grafos direcionados e concluem que o algoritmo Infomap possui o melhor *trade-off* entre acurácia e tempo de execução. Diferente do estudo proposto neste trabalho, os autores não abordam implementações paralelas e distribuídas, apenas as implementações sequenciais originais de cada algoritmo.

Sobin et al. [Sobin et al. 2017] propuseram uma revisão da literatura de métodos de detecção de comunidades e suas implementações paralelas. Os autores criam uma taxonomia para dividir os métodos em grandes grupos e abordam quais otimizações são utilizadas em cada implementação. Diferente do estudo proposto neste trabalho, que elege um algoritmo para futuras otimizações paralelas e distribuídas, os autores elegem um algoritmo para ser utilizado em ambientes paralelos dadas as implementações já disponíveis. Nesse contexto, os autores concluem que o algoritmo Louvain é a melhor escolha para utilização em diferentes ambientes por possuir a maior quantidade de pesquisa em métodos paralelos, abordando arquiteturas *multicore*, *manycore* e GPU.

Naik et al. [Naik et al. 2022] propuseram uma revisão metodológica de implementações de métodos de detecção de comunidades em ambientes distribuídos com foco em análise de redes sociais. Os autores focam na explicação dos métodos abordados e suas otimizações, sem realizar experimentos com os algoritmos. Não é recomendado um

algoritmo para utilização em ambientes distribuídos, apenas discutidas as características de diferentes implementações nestes ambientes.

6. Conclusão

Métodos de detecção de comunidades são amplamente utilizados em diversos tipos de aplicações de análise de grafos. Devido ao crescimento do volume de dados, faz-se necessária a utilização de implementações paralelas e distribuídas. No entanto, este estudo mostrou que tanto as versões paralelas para multiprocessadores quanto para multicomputadores mais citadas na literatura ainda apresentam significativos problemas de escalabilidade. Em multiprocessadores a eficiência não passou de 50% quando utilizados todos os núcleos físicos, o que pode ser considerado baixo para este tipo de máquina. Em multicomputadores, a escalabilidade das implementações é ainda pior, gerando, em alguns casos, valores de *Speed-Up* mais baixos do que em ambientes de memória compartilhada. No entanto, este estudo mostrou que cada implementação apresenta causas diferentes para esta baixa escalabilidade. Enquanto na Vite isto é resultado da grande necessidade de comunicação entre processos MPI, no caso da HyPC-Map é resultado da baixa eficiência do processamento em memória compartilhada feito pelo OpenMP. Isto faz com que futuras otimizações para cada implementação sigam estratégias diferentes. Baseado neste estudo, em trabalhos futuros pretende-se explorar otimizações para o método Infomap em ambientes distribuídos. Os principais focos destas otimizações serão na convergência do algoritmo e no sincronismo das *threads* em cada máquina do cluster. Também será explorada a utilização de um esquema assíncrono de comunicação entre as máquinas.

Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal Nivel Superior – Brasil (CAPES) – Código de Financiamento 001. Os autores agradecem 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.

Referências

Agreste, S., De Meo, P., Fiumara, G., Piccione, G., Piccolo, S., Rosaci, D., Sarne, G. M., and Vasilakos, A. V. (2016). An empirical comparison of algorithms to find communities in directed graphs and their application in web data analytics. *IEEE transactions on big data*, 3(3):289–306.

Bae, S.-H., Halperin, D., West, J., Rosvall, M., and Howe, B. (2013). Scalable flow-based community detection for large-scale network analysis. In *2013 IEEE 13th International Conference on Data Mining Workshops*, pages 303–310.

Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008.

Fan, W., Xu, J., Wu, Y., Yu, W., and Jiang, J. (2017). Grape: Parallelizing sequential graph computations. *Proceedings of the VLDB Endowment*, 10(12):1889–1892.

Faysal, M. A. M., Arifuzzaman, S., Chan, C., Bremer, M., Popovici, D., and Shalf, J. (2021). Hypc-map: A hybrid parallel community detection algorithm using information-theoretic approach. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE.

Fortunato, S. and Hric, D. (2016). Community detection in networks: A user guide. *Physics reports*, 659:1–44.

Ghosh, S., Halappanavar, M., Tumeo, A., Kalyanaraman, A., Lu, H., Chavarrià-Miranda, D., Khan, A., and Gebremedhin, A. (2018). Distributed louvain algorithm for graph community detection. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 885–895.

Iosup, A., Hegeman, T., Ngai, W. L., Heldens, S., Prat-Pérez, A., Manhardto, T., Chafio, H., Capotă, M., Sundaram, N., Anderson, M., et al. (2016). Ldbc graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *Proceedings of the VLDB Endowment*, 9(13):1317–1328.

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

Lancichinetti, A., Fortunato, S., and Radicchi, F. (2008). Benchmark graphs for testing community detection algorithms. *Physical review E*, 78(4):046110.

Leskovec, J. and Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`.

Lu, H., Halappanavar, M., and Kalyanaraman, A. (2015). Parallel heuristics for scalable community detection. *Parallel Computing*, 47:19–37.

Naik, D., Ramesh, D., Gandomi, A. H., and Gorojanam, N. B. (2022). Parallel and distributed paradigms for community detection in social networks: A methodological review. *Expert Systems with Applications*, 187:115956.

Newman, M. E. J. and Girvan, M. (2004). Finding and evaluating community structure in networks. *Phys. Rev. E*, 69:026113.

Rossi, R. A. and Ahmed, N. K. (2015). The network data repository with interactive graph analytics and visualization. In *AAAI*.

Rosvall, M., Axelsson, D., and Bergstrom, C. T. (2009). The map equation. *The European Physical Journal Special Topics*, 178(1):13–23.

Sobin, C., Raychoudhury, V., and Saha, S. (2017). A survey of parallel community detection algorithms. In *Handbook of Research on Applied Cybernetics and Systems Science*, pages 1–26. IGI Global.

Staudt, C. L. and Meyerhenke, H. (2016). Engineering parallel algorithms for community detection in massive networks. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):171–184.

Staudt, C. L., Sazonovs, A., and Meyerhenke, H. (2016). Networkit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530.