Aumentando a Eficiência na Execução de Algoritmos de Grafos em HPC

Marcelo K. Moori[1], Hiago Mayk G. de A. Rocha[1], Janaina Schwarzrock[1], Arthur F. Lorenzon[2] e Antonio Carlos S. Beck[1]

[1]Universidade Federal do Rio Grande do Sul – Porto Alegre – RS – Brasil

[2]Universidade Federal do Pampa – Alegrete – RS – Brasil

mkmoori@inf.ufrgs.br

Abstract. *The growing need for extracting information from massive data - structured as graphs - has been pushing the development of even more robust parallel algorithms for such processing. However, the algorithms' communication-bound behavior and the highly irregular structure of the currently used graphs are hurdles to achieving the same levels of performance and efficiency as observed for other parallel applications. In this paper, we show that the scalability of different graph applications varies according to the used algorithms and their databases and, in most cases, using all available cores (i.e., all available processor's cores for execution) is not the best option in terms of efficiency. Based on that, we propose* ***MultGraph****, a framework that enables the simultaneous processing of several algorithms/graphs, distributing them in a non-uniform way among the cores, instead of executing them serially (i.e., one after another) with the maximum available parallelism. MultGraph works in two steps: (i) characterizing the algorithms/graphs by their efficiency levels; (ii) defining the allocations (clusters of algorithms to be executed concurrently), number of threads for each of them, and clusters' execution order. Experimental results on three multicore processors (Intel and AMD) show that MultGraph improves in up to* $9.21\times$ *and on average* $4.52\times$ *the algorithm's execution time compared to the default execution of applications in HPC systems.*

Resumo. *A crescente necessidade de extrair informações de dados massivos - estruturados como grafos - tem impulsionado o desenvolvimento de algoritmos paralelos cada vez mais robustos para este processamento. No entanto, o comportamento voltado à comunicação e a estrutura altamente irregular dos grafos usados cotidianamente são obstáculos para alcançar os mesmos níveis de desempenho e eficiência como os observados em outras aplicações paralelas. Neste artigo, nós mostramos que a escalabilidade de diferentes aplicações de grafos variam de acordo com o algoritmo usado e a sua base de dados e que, em muitos casos, utilizar todos recursos disponíveis (i.e. todos os núcleos do processador para a execução) não é a melhor opção em termos de eficiência. Com base nisso, nós propomos o* ***MultGraph****, um framework que permite o processamento simultâneo de vários algoritmos/grafos, distribuindo-os de maneira não uniforme entre os núcleos, ao invés de executá-los serialmente (i.e. um após o outro) com o máximo paralelismo disponível. MultGraph funciona em dois passos: (i) caracterizando os algoritmos/grafos pelos seus níveis de eficiência; (ii) definindo as alocações (agrupamentos de algoritmos e entradas a serem executados concorrentemente), número de threads para cada um deles, e a ordem de execução destes grupos. Resultados experimentais em três processadores multicore (Intel e AMD) mostram que o MultGraph melhora em até* $9,21\times$ *e* $4,52\times$

em média o tempo de execução das aplicações em relação à execução padrão de aplicações em sistemas HPC.

1. Introdução

O aumento no número de núcleos e memória disponível nos Computadores de Alto Desempenho (do inglês, *High Performance Computing* – HPC) vem possibilitando a análise de uma quantidade massiva de dados estruturados em forma de grafos, extraídos de fontes como Google, Facebook e Twitter [Yan et al. 2017, Mofrad et al. 2020]. Assim, por meio de algoritmos como o PageRank e Caminhos Mínimos, é possível realizar operações como o ordenamento das páginas mais visitadas em uma busca de conteúdo no Google ou até a realização de computações mais complexas no âmbito da Inteligência Artificial. Enquanto a necessidade de processar os grafos do mundo real de forma eficiente impulsionou o desenvolvimento de vários *frameworks*, como Ligra e Polymer [Shun and Blelloch 2013, Zhang et al. 2015], as características intrínsecas dos sistemas de HPC (e.g., hierarquia de memória complexa) e dos grafos (e.g., estrutura irregular e dimensão) têm desafiado os programadores a extrair todo o potencial dos recursos de processamento disponíveis [Rocha et al. 2021].

Tradicionalmente, a maneira de executar aplicações paralelas (que inclui processamento de grafos) nos sistemas de HPC é submeter *jobs* que executem a aplicação com o número de *threads* igual ao máximo número de recursos de processamento disponíveis no sistema. Porém, nem todas as aplicações se beneficiam disso, o que significa que usar o máximo número de *threads* pode não ser eficiente [1] [Lorenzon and Beck Filho 2019], isto é, não utilizar os recursos computacionais disponíveis de maneira ótima. Essa falta de escalabilidade está relacionada tanto a fatores de *hardware* (e.g, saturação de unidades funcionais e barramento de comunicação) quanto de *software* (e.g, sincronização de dados e acessos concorrentes à memória compartilhada) [Raasch and Reinhardt 2003, Suleman et al. 2008, da Silva et al. 2020]. Além do mais, aplicações de grafos são intrinsecamente orientadas à comunicação: possuem uma baixa fração de computação em relação às comunicações realizadas entre as *threads* [Mofrad et al. 2020], que envolve acessos extras à memória e/ou troca de mensagens [Beamer et al. 2015b]. Essa característica intensifica os problemas acima mencionados, uma vez que as unidades de processamento tendem a ficar ociosas por causa das sincronizações excessivas.

Para ilustrar melhor a discussão acima, a Tabela 1 mostra o número de *threads* (#Thr) que apresenta o melhor tempo de execução seguindo da eficiência do sistema (Eff) quando os algoritmos *Betweenness Centrality* (BC) e *Breadth-First Search* (BFS) executam diferentes grafos (discutidos em detalhes na seção 2). Pode-se observar que o melhor número de *threads* para executar cada aplicação varia de acordo com o grafo processado; e que o mesmo grafo pode influenciar na escalabilidade das aplicações, e.g., o BFS executando Texas é escalável até 18 *threads*, enquanto que o BC com Texas escala até 30 *threads*. Outra limitação que destacamos é com relação a eficiência do sistema: o comportamento ideal (desejado) para uma aplicação que escala é apresentar uma suave redução na eficiência do sistema conforme aumenta o número de *threads* [Eager et al. 1989], como é o caso da BC-orkut com 35%, fazendo uso do máximo de *threads*. Contudo, na maioria dos casos, acontece uma redução brusca na eficiência.

Uma vez que nem todas as aplicações de grafos se beneficiam ao usar o máximo de recursos do sistema, como visto nos cenários anteriores, ajustar adequadamente o número de *threads* resulta no aumento de eficiência dessas aplicações. Assim, ao mesmo tempo

[1] Um sistema é 100% eficiente quando o seu ganho em desempenho é equivalente ao número de núcleos usados para executar sua versão paralelizada (e.g., usando-se 16 núcleos obtém-se 16x de *speedup*), mas ele apresenta uma eficiência de 50% se ao usar 16 cores obtém-se um *speedup* de 8x.

Tabela 1. Variação do melhor número de *threads* e eficiência do sistema nas execuções dos algoritmos BC e BFS conforme a variação dos grafos de entrada. Experimentos executados no Intel Xeon E5-2640v2.

	Amazon	**California**	**Google**	**Pennsyl.**	**Youtube**	**Berk**	**Texas**	**Wikitalk**	**Orkut**	**Patent**
BC (#Thr / Eff.)	10 / 21%	30 / 16%	32 / 26%	26 / 14%	28 / 11%	18 / 16%	30 / 11%	30 / 22%	32 / 35%	14 / 30%
BFS (#Thr / Eff.)	6 / 43%	30 / 13%	30 / 20%	18 / 18%	16 / 18%	14 / 20%	18 / 17%	24 / 20%	30 / 23%	14 / 21%

em que uma aplicação está sendo executada com um pequeno número de *threads*, porém com uma eficiência maior, pode-se utilizar os recursos que estão ociosos para executar outra aplicação. Desta maneira, este artigo baseia-se no princípio que executar simultaneamente múltiplos algoritmos de grafos de modo que a eficiência total do sistema aumente e, portanto, o tempo de execução total de todos os grafos a serem executados diminua. Para isto, neste trabalho propomos o *framework* MultGraph: uma estratégia de multiprogramação que identifica, dado o conjunto de algoritmos/grafos e o multiprocessador alvo, a melhor combinação de aplicações que deve ser executada simultaneamente. O algoritmo usado no framework trabalha em dois passos, dado um conjunto de grafos a serem executados: *(i)* Caracteriza as aplicações de acordo com o número de *threads* que apresenta melhor eficiência; e *(ii)* Define a alocação e a ordem de execução das aplicações nos núcleos do sistema.

Em resumo, as principais contribuições deste trabalho são:

- Análise de escalabilidade dos algoritmos de grafos, mostrando que dependendo do tipo de computação e da topologia do grafo, tem-se diferentes número de *threads* que entregam o melhor tempo de execução e eficiência do sistema;
- O *framework* MultGraph: uma ferramenta para execução simultânea de aplicações de grafos que oferece ganhos significativos tanto no tempo de execução quanto na eficiência do sistema;
- Resultados experimentais considerando três multiprocessadores diferentes, mostraram que o algoritmo MultGraph consegue melhorar em até $9.21\times$ e em média $4,52\times$ o tempo de execução, quando comparada com a execução padrão em sistemas HPC.

O restante deste artigo está organizado como segue: A seção 2 apresenta os conceitos básicos e as características dos algoritmos e grafos avaliados neste trabalho; Os detalhes do algoritmo MultGraph são apresentados na seção 3; A metodologia utilizada é detalhada na seção 4; A seção 5 apresenta os resultados; Os trabalhos relacionados são apresentados na seção 6. Por fim, a seção 7 conclui este trabalho.

2. Background

2.1. Grafos

Grafos são formados por uma quantidade massiva de dados e variam bastante em suas topologias. O modo como os grafos são lidos e processados está intrinsecamente ligado à distribuição dos seus nodos e arestas em sua estrutura; assim, podemos caracterizá-los com base nas seguintes métricas extraídas do *Stanford Network Analysis Platform* (SNAP) [Leskovec and Krevl 2014]: ***Weakly Connected Components* (WCC):** é um subconjunto maximal dos vértices do grafo com a restrição de que para quaisquer pares de vértices deve existir um caminho entre os vértices do par, ignorando a sua direção; ***Strongly Connected Components* (SCC):** é um subconjunto maximal dos vértices do grafo tal qual o WCC, mas com sua direção importando nesse caso (e.g., para um grafo não direcionado o SCC e WCC são equivalentes); ***Coeficiente Médio de Clusterização (CMC)*:** é a média dos coeficientes de agrupamentos locais de cada vértice do grafo, o que é definido pela proporção de ligações entre os vértices da sua vizinhança e o número de ligações

Tabela 2. Características dos grafos.

	Amazon	California	Google	Pennsyl.	Youtube	Berk	Texas	Wikitalk	Orkut	Patent
#Vértices	3,3e5	1,9e6	8,7e5	1,0e6	1,1e6	6,8e5	1,3e6	2,3e6	3,0e6	3,7e6
#Arestas	9,2e5	2,7e6	5,3e6	1,5e6	2,9e6	7,6e6	1,9e6	5,0e6	1,1e8	1,6e7
#Vért. no maior WCC	1	0,996	0,977	1	1	0,956	0,979	0,998	1	0,997
#Ares. no maior WCC	1	0,998	0,993	1	1	0,987	0,978	0,999	1	1
#Vért. no maior SCC	1	0,996	0,497	1	1	0,489	0,979	0,047	1	0
#Ares. no maior SCC	1	0,998	0,67	1	1	0,595	0,978	0,297	1	0
CMC	3,9e-1	4,6e-2	5,1e-1	4,6e-2	8,0e-2	5,9e-1	4,7e-2	5,2e-2	1,6e-1	7,5e-2
NT	6,6e5	1,2e5	1,3e7	6,7e4	3,0e6	6,4e7	8,2e4	9,2e6	6,2e8	7,5e6
FTF	7,9e-2	2,0e-2	1,9e-2	2,0e-2	2,0e-3	2,7e-3	2,0e-2	1,1e-3	1,4e-2	2,0e-2
Diâmetro	44	849	21	786	20	514	1054	9	9	22
90-PDE	15	500	8,1	530	6,5	9,9	670	4	4,8	9,4

Tabela 3. Características dos algoritmos.

	Fonte Única /Todo Grafo	(Não-)Ponderado	#Vértices por Iteração	Percorrimento /Computação
BC	Fonte Única	Não-Ponderado	Parte	Computação
BFS	Fonte Única	Não-Ponderado	Parte	Percorrimento
CC	Todo Grafo	Não-Ponderado	Parte	Percorrimento
PR	Todo Grafo	Não-Ponderado	Todo	Computação
SSSP	Fonte Única	Ponderado	Parte	Percorrimento
TC	Todo Grafo	Não-Ponderado	Parte	Percorrimento

que poderiam existir entre estes; ***Número de Triângulos (NT)*****:** é o número de triplas (combinação de três vértices) conectadas, ignorando a sua direção; ***Fração de Triângulos Fechados (FTF)*****:** é a razão entre o número de triângulos e o total de triplas (todas as combinações de três vértices conectadas ou não); ***Diâmetro*****:** o mais longo menor caminho entre qualquer par de vértices do grafo; ***90-Percentil de Diâmetro Efetivo (90-PDE)*****:** é a menor distância para qual 90% dos pares de uma amostra de 1000 vértices estão distanciados no máximo por esse valor. Na Tabela 2 nós destacamos essas características nos diferentes grafos avaliados neste trabalho (extraídos do SNAP).

2.2. Frameworks e Algoritmos para Processamento de Grafos

Para a extração eficiente de informações em grafos (e.g., conectividade, caminhos mínimos e visibilidade dos vértices), vários trabalhos da literatura propõem *frameworks* para otimizar a execução paralela de algoritmos de processamento de grafos [Shun and Blelloch 2013, Mofrad et al. 2020]. Tais *frameworks* apresentam implementações de alto desempenho de algoritmos amplamente utilizados pela indústria, como por exemplo o *PageRank* (PR) e o *Breadth-First Search* (BFS), com aplicabilidade em diversas áreas (e.g., química, medicina, comercio e logística) [Gleich 2014].

Os algoritmos amplamente estudados na literatura de grafos e que a maioria dos *frameworks* implementam são [Heidari et al. 2018]: ***Betweenness Centrality (BC)*** aproxima a medida de intermediação do grafo por um subconjuntos de vértices, tendo que a intermediação de um vértice v é dada pela razão entre os caminhos mínimos que atravessam v e todos os caminhamentos mínimos do conjunto considerado; ***Breadth-First Search (BFS)*** realiza a travessia do grafo a partir de um vértice raiz, visitando todos os vértices de mesmo nível antes de avançar para o próximo; ***Connected Components (CC)*** computa todos os componentes conexos do grafo, que é definido por um conjunto de vértices no qual existe um caminho não direcionado entre qualquer par de vértices do conjunto; ***PageRank (PR)*** iterativamente calcula o valor de *PageRank* para todos os vértices com um fator de amortização d de 0,85; ***Single-Source Shortest Paths (SSSP)*** calcula todos os caminhamentos mínimos alcançáveis a partir de um dado vértice raiz; ***Triangle Counting (TC)*** computa o total de triângulos do grafo.

Além das diferenças topológicas dos grafos do mundo real (ver Tabela 2), os al-

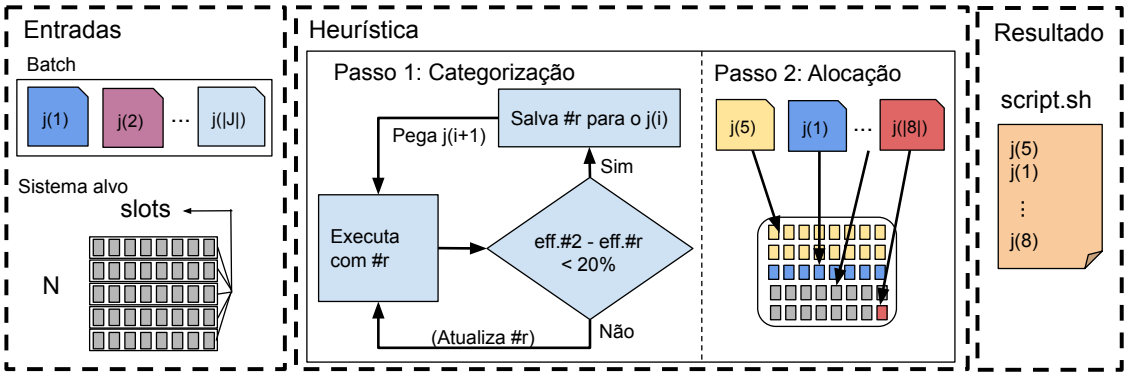
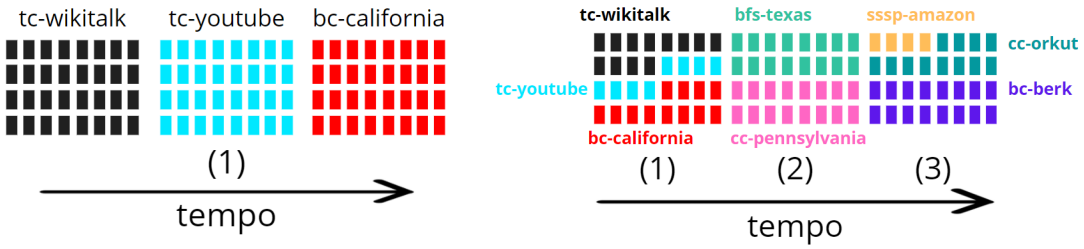


Figura 1. Fluxo de otimização do MultGraph.



(a) Execução Tradicional (b) MultGraph

Figura 2. Exemplo de comparação entre a (a) execução tradicional de aplicações de HPC e (b) a solução gerada pelo MultGraph.

goritmos também possuem diferenças em termos de (ver Tabela 3): requerer ou não um vértice inicial (BC, BFS e SSSP); fração de vértices que são computadas por iteração; e se o processamento é focado no percorrimento do grafo (BFS, CC, SSSP e TC) ou no cálculo de propriedades dos vértices (BC e PR). Além disso, diferentes algoritmos têm diferentes aplicabilidades (e.g. SSSP em GPS's para fins de traçamento de caminhos, CC em recomendação de conexões em redes sociais como Twitter e Facebook, etc).

3. MultGraph

MultGraph é uma ferramenta *offline* que otimiza a execução simultânea de quaisquer conjunto de aplicações de grafos dado como entrada, encontrando os números de *threads* e ordem de execução ideais que melhorem o tempo de execução das aplicações e a eficiência do sistema. O fluxo de otimização o MultGraph é apresentado na Figura 1 e ele trabalha da seguinte forma: (*i-Entradas*) O usuário provê o conjunto de aplicações (algoritmos/grafos) assim como o sistema em que ele será executado; (*ii-Heurística*) O *framework* aplica uma heurística de dois passos onde: *Passo 1*, realiza uma categorização das aplicações com o objetivo de encontrar o melhor número de *threads* para cada uma delas baseando-se nos valores de eficiência. Também é feita a ordenação das aplicações de forma decrescente (do maior número de *threads* para o menor); e *Passo 2*, o MultGraph pega a lista de aplicações ordenadas e define a alocação das *threads* nos núcleos do sistema; (*iii-Resultado*) Por fim, o *framework* retorna um *script* com a solução obtida. Este *script* é usado para iniciar a execução paralela dos algoritmos/grafos dados em (*i-Entradas*), cada um com seu número de *threads* devidamente configurado e agrupados na ordem dada como solução. A Figura 2 mostra um exemplo de solução gerada pelo MultGraph para o conjunto genérico de aplicações (TC-wikitalk, TC-youtube, BC-california, etc) em comparação com na maneira Tradicional de execução de aplicações em HPCs. Note que no MultGraph as aplicações são executadas simultaneamente, ao passo que na execução tradicional elas esperam em uma fila de até que as aplicações que chegaram antes terminem as suas execuções.

O MultGraph considera no *Passo 1* da (*ii-Heurística*) o seguinte cálculo de eficiência [Eager et al. 1989]: $E = \frac{t_1}{n \times t_n}$, onde t_1 é o tempo da execução serial, t_n é o tempo da execução paralela usando n *threads*. Considerando essa fórmula, o framework busca categorizar as aplicações de acordo com o número de *threads* que não implica em uma redução muito significativa na eficiência do sistema. Além disso, para o gerenciamento do número de *threads* e as suas alocações nos núcleos do sistema, o MultGraph usa as variáveis ambiente *OMP_NUM_THREADS* e *GOMP_CPU_AFFINITY*. Na seção seguinte, apresentamos os detalhes da heurística proposta.

3.1. Heurística

O Algoritmo 1 apresenta o pseudocódigo da heurística usada pela otimização do MultGraph. Ele toma como entradas o conjunto de aplicações a serem executadas J, o tamanho dos saltos no cálculo das eficiências $slots$ e as configurações do multiprocessador alvo $system$. O algoritmo começa inicializando a variável que armazena as categorizações do Passo 1, opt_eff, com vazio ([ ]), o *script* da solução final também como vazio ($\oslash$) e a variável $nucleos$ com o número de núcleos disponíveis no sistema (linhas 1-3).

Após isso, o algoritmo inicia o Passo 1 (linhas 5-15): Para cada aplicação j na lista de aplicações a serem executadas J, o algoritmo coleta suas eficiências na execução com $N - i * slot$ *threads*, com i variando no intervalo $[0, \frac{|nucleos|}{slot} - 1]$ (e.g, considerando $nucleos = 32$ e com $slot = 8$ poderiam ser avaliadas as execução com 32, 24, 16, 8, 4 e 2 *threads*, nessa ordem). Dessa forma, a partir do maior número de *threads* aplicável ao número do tamanho do *slot*, testa-se a diferença de eficiência para com a de 2 *threads* e, caso essa seja menor que 20% (porcentagem alcançada por meio de uma análise experimental, que se mostrou ideal para uma boa variedade categórica), esse fica definido como o número otimizado de *threads* para essa aplicação (linha 9). No caso da diferença de eficiência nunca ser abaixo de 20%, define-se a execução serial como sendo a melhor opção para a execução da aplicação (linha 13). No fim do Passo 1, a lista opt_eff é ordenada de forma decrescente (do maior valor para o menor) e inicia-se o Passo 2;

No *Passo 2* (linha 17-25), o algoritmo define as execuções das aplicações por avaliar a lista opt_eff enquanto ainda existem aplicações a serem mapeadas nos núcleos do sistema (linha 17) e enquanto o sistema ainda tiver núcleos disponíveis (linha 18). Neste caso, o algoritmo obtém uma aplicação j cujo seu número de *threads* com melhor eficiência r (i.e., a tupla $< j, r >$) cabe no atual número de núcleos disponíveis no sistema $|nucleos|$ (linhas 19). Após isso, define-se a afinidade de *threads* para os núcleos do sistema (linha 20) e escreve a solução no arquivo *script* (linha 21). No caso de não existirem núcleos disponíveis mesmo ainda tendo aplicações a serem executadas, uma nova rodada

Algorithm 1 *Heurística*

Input: $J \leftarrow$ Conjunto de aplicações $\{j_1, j_2, ..., j_k\}$,
$slot \leftarrow$ Tamanho dos *slots*,
$system \leftarrow$ Sistema alvo.
Ensure: $script$ contendo a solução final.
1: $opt\_eff \leftarrow [\,]$
2: $script \leftarrow \oslash$
3: $nucleos \leftarrow AvailableCores(system)$
4: // Passo 1: Caracteriza as aplicações.
5: **for** $j \in J$ **do**
6: **for** $i \in [0, \frac{|nucleos|}{slot} - 1]$ **do**
7: $r \leftarrow N - i * slot$
8: **if** $(|eff(j.2) - eff(j.r))| < 20\%)$ **then**
9: $opt\_eff.insert(< j, r >)$
10: $break$
11: **end if**
12: **end for**
13: $opt\_eff.insert(< j, 1 >)$
14: **end for**
15: $opt\_eff \leftarrow DecreasingSort(opt\_eff)$
16: // Passo 2: Define mapeamento e ordem de execução.
17: **while** $(opt\_eff \neq \oslash)$ **do**
18: **if** $(AvailableAny(nucleos)))$ **then**
19: $< j, r > \leftarrow opt\_eff.getProximo(|nucleos|)$
20: $map \leftarrow DefineAfinidade(j, r, nucleos)$
21: $Write(j, r, map, script)$
22: **else**
23: $MakeAvailable(nucleos)$
24: **end if**
25: **end while**

Tabela 4. Configurações das máquinas usadas.

	Intel32	Intel64	AMD128
Processador	Intel Xeon E5-2640v2	Intel Xeon X7550	AMD Ryzen 3990X
Frequência base	2,0GHz	2,0GHz	2,9GHz
#Núcleos / *#Threads*	16 / 32	32 / 64	64 / 128
Cache L1	32 KB	32 KB	32 KB
Cache L2	256 KB	256 KB	512 KB
Cache L3	20 MB	18 MB	16 MB
Memória RAM	128 GB	128 GB	32 GB

do algoritmo inicia. Os núcleos são liberados (linha 23) e as aplicações são mapeadas nos núcleos para serem executadas assim que as aplicações previamente alocadas terminem suas execuções.

Complexidade do Tempo de Execução da Heurística: Para obter a complexidade assintótica do tempo de execução do algoritmo, devemos analisar cada um de seus passos: No *Passo 1*, temos que as linhas 5-12 tem complexidade O($|J| * |nucleos|$), assumindo o caso onde $slot = 1$ (valor para a execução de pior caso). A ordenação realizada na linha 15 tem complexidade O($|J| * log(|J|)$). Com isso, a complexidade do Passo 1 fica em O($|J| * [|nucleos| + log(|J|)]$). Para o *Passo 2* temos uma complexidade de O($|J| * |nucleos|$), visto que dentro do *loop* que cobre as linhas 17-25, sempre 1 elemento é removido da lista opt_eff e todos os núcleos devem ser avaliados para saber se existe algum disponível. Portanto, a complexidade de pior caso do algoritmo é definida pelo Passo 1 e resulta em O($|J| * [|nucleos| + log(|J|)]$). Assim, considerando o custo polinomial na execução de pior caso, o MultGraph trabalha de forma eficiente, possibilitando ao usuário submeter *batches* de diferentes tamanhos a serem otimizados em sistemas com número variado de núcleos.

4. Metodologia

Algoritmos: Neste trabalho nós avaliamos os algoritmos disponíveis no GAP *Benchmark Suite* (GAPBS) [Beamer et al. 2015a]. O GAPBS disponibiliza a implementação de alto desempenho de um conjunto representativo de algoritmos de processamento de grafos, sendo um *baseline* representativo do estado da arte para as pesquisas nessa área. A ferramenta também conta com uma metodologia padronizada para a execução das aplicações, oferecendo confiabilidade nos dados extraídos em suas avaliações e comparações. Nós descrevemos todos os algoritmos avaliados na seção 2 (ver Tabela 3). Todos eles são implementados em C++11, paralelizados usando OpenMP e suas implementações consideram a estratégia de otimização que é mais adequada para cada um deles. Em nossos experimentos, nós compilamos as aplicações usando GNU g++ 10.1.0 e OpenMP versão 4.5 com a *flag* de otimização -O3.

Grafos: Nós avaliamos 10 grafos extraídos da SNAP [Leskovec and Krevl 2014]. Eles são descritos na seção 2 (ver Tabela 2). Nós consideramos um conjunto representativo de grafos do mundo real cobrindo as duas grandes classes de topologias (malhas e redes sociais) com diferentes características e tamanho (variando de 300 mil até 4 milhões de nós).

Ambiente de Execução: Nós realizamos os experimentos nos sistemas *multicore* apresentados na Tabela 4 usando o Sistema Operacional (SO) Linux com *kernel* v. 4.19.

Configurações e Análises Realizadas: Para analisar o comportamento dos algoritmos executando os grafos descritos anteriormente, nós consideramos tanto as métricas de processamento (e.g., tempo de execução e eficiência) quanto as que descrevem as características dos algoritmos e grafos (ver seção 2 para mais detalhes). Inicialmente, nós analisamos a escalabilidade das aplicações (ver seção 5.1), realizando execuções na versão serial e paralela, onde o número de *threads* varia de 2 ao máximo disponível no sistema

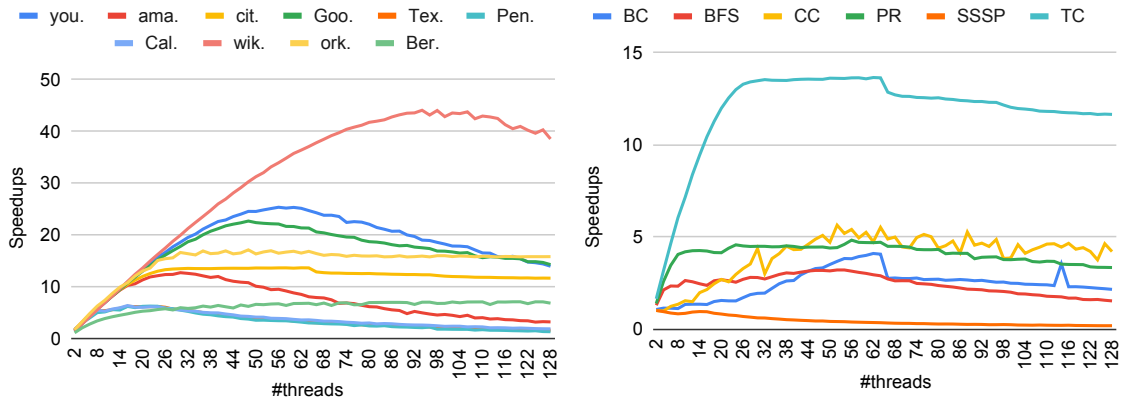
(32, 64 e 128). Nessas execuções, o SO é responsável por decidir o mapeamento de *threads* aos núcleos. Todos os experimentos foram realizados com o número de execuções padrão especificado em [Beamer et al. 2015a]. Logo após, nós analisamos o desempenho do MultGraph (ver seção 5.2), mostrando as categorizações e os ganhos de desempenho e eficiência alcançados. Para essa análise, nós selecionamos aleatoriamente 4 *batches* (conjunto de algoritmos/grafos) com tamanhos variando de 3 a 39. Essa metodologia de criação dos *batches* destaca a capacidade do MultGraph em trabalhar com cenários reais, onde a otimização deve ser feita conforme as aplicações chegam, sem nenhum conhecimento *a priori* das mesmas. Os resultados do MultGraph são comparados com a execução padrão das aplicações em sistemas de HPC (**baseline**): as aplicações são executadas em sequencia, sempre usando o máximo de recursos de processamento disponíveis.

5. Resultados Experimentais

5.1. Analise de escalabilidade

Nesta seção, nós mostramos experimentalmente os dois princípios nos quais o MultGraph se baseia: (*i*) nem sempre usar o máximo de recursos do sistema entrega o melhor desempenho para as aplicações de grafos; e que (*ii*) não existe um número de *threads* ideal que resulta no melhor tempo de execução para todos os algoritmos e grafos. Nesse sentido, a Figura 3 apresenta um exemplo representativo do comportamento dos *speedups* sobre a versão serial para *(a)* o algoritmo TC com relação a variação dos grafos e *(b)* para o grafo cit-Patents com a variação dos algoritmos. Os experimentos foram executados no sistema *AMD128*. Desses dados, destacamos as seguintes observações: *(i)* O comportamento dos *speedups* na maioria dos algoritmos/grafos segue um padrão de crescimento até um ponto de saturação, onde não há mais ganhos significativos ou até a degradação do desempenho (e.g., TC-wik. satura com 94 *threads* com 44,10 de *speedup*, já quando executada com 128 *threads* tem *speedup* de 38,56); *(ii)* A magnitude dos ganhos em *speedup* varia conforme o algoritmo, grafo de entrada e sistema. Por exemplo, a execução TC-you. apresenta *speedup* de até 25,36 , porém a TC-Cal. apresenta apenas 6,36. Já com relação a variação nos algoritmos, temos que o TC-cit. apresenta 13,66 e o SSSP-cit tem apenas 1,00 de *speedup* (i.e., não tem ganhos na execução paralela).

Tendo em vista a saturação dos *speedups* em pontos bem baixos do gráfico (32 *threads* para a maioria dos grafos em *(a)*, sendo que o TC é o algoritmo que mais provê escalabilidade), mostra que executar um conjunto de aplicações serialmente - uma após a outra com a total utilização do sistema - pode ser muito desvantajoso. Essa variação dos *speedups* mostrada na Figura 3 impacta diretamente no melhor número de *threads* para



(a) TC (b) cit-Patents

Figura 3. *Speedups* do algoritmo (a) TC e do grafo (b) cit-Patents quando executados na máquina AMD128.

Tabela 5. *#threads* com melhor tempo de execução para cada algoritmo/grafo.

	Intel32						Intel64						AMD128					
	BC	**BFS**	**CC**	**PR**	**SSSP**	**TC**	**BC**	**BFS**	**CC**	**PR**	**SSSP**	**TC**	**BC**	**BFS**	**CC**	**PR**	**SSSP**	**TC**
youtube	28	16	26	28	4	32	64	62	64	64	14	62	16	16	62	8	6	56
amazon	10	6	28	18	4	22	64	62	64	64	62	62	8	14	18	18	2	30
citPatents	14	14	32	32	6	32	64	62	62	64	62	62	62	52	52	56	2	62
Google	32	30	28	30	14	30	64	64	62	62	12	64	16	16	32	12	8	48
Texas	30	18	32	30	2	20	24	44	64	62	2	62	8	8	46	58	2	16
Pennsylvania	26	18	32	30	2	16	28	64	62	62	2	62	8	8	40	58	2	22
California	30	30	30	30	2	26	38	58	62	64	2	64	8	8	50	112	2	16
wikitalk	30	24	14	26	22	30	62	64	62	62	34	58	16	14	12	10	8	94
orkut	32	30	30	32	22	32	62	64	64	64	20	64	24	16	66	12	16	48
Berk	18	14	24	32	2	32	8	62	62	64	4	62	8	8	66	34	6	120

executar cada combinação algoritmo/grafo. Na Tabela 5 nós sumarizamos o número de *threads* que apresentou o melhor tempo de execução para cada algoritmo/grafo nos três sistemas avaliados (*Intel32*, *Intel64* e *AMD128*). Alguns algoritmos tendem a ter melhor performance com um maior número de *threads* (CC e TC em todos os sistemas), porém também existem aqueles que requerem apenas de um número pequeno de *threads* (SSSP). Vários aspectos influenciam esse comportamento: os diferentes algoritmos realizam diferentes tipos de computação (ver Tabela 3); os grafos do mundo real possuem diferentes topologias (ver Tabela 2); e, os diversos multiprocessadores apresentam diferente número de núcleos e hierarquia de memória (ver seção 4). Todos esses aspectos devem ser considerados quando se ajusta o número de *threads* para melhorar a execução dos algoritmos de grafos, pois uma otimização específica para um algoritmo, grafo, ou sistema, não será necessariamente efetiva se qualquer uma dessas variáveis mudar. Com isso, a vantagem do *MultGraph* é a capacidade de contornar os aspectos acima mencionados, otimizando a execução simultânea das aplicações de grafos com base na categorização das aplicações.

5.2. Performance do MultGraph

5.2.1. Passo 1: Categorização dos algoritmos/grafos:

Nessa seção nós apresentamos o resultado da aplicação do *Passo 1* da Heurística do MultGraph (ver seção 3). Esse passo é responsável por categorizar as aplicações com relação ao número de *threads* que entrega a melhor eficiência do sistema. Na Tabela 6, nós detalhamos o melhor número de *threads* para cada sistema avaliado (Intel32, Intel64 e AMD128).

No geral, a maior parte das aplicações tem sua melhor execução com 1 e 8 *threads*. Mais especificamente, para o Intel32, 51,7% das aplicações são categorizadas com 1 *thread* e 58,3% e 60%, respectivamente, para o Intel64 e o AMD128. Nessa mesma ordem, as aplicações de 8 *threads* aparecem com uma porcentagem de 16,7%, 41,7% e 25%. Isso acontece principalmente nos algoritmos BC e SSSP em todas as máquinas e para os demais algoritmos (com exceção do TC nas máquinas com maior número de núcleos). Um maior número de *threads* implica em maior utilização dos resursos do sistema e maior *overhead* de sincronização nos acessos aos dados compartilhados. Se os recursos utilizados ficarem muito tempo ociosos quando na sincronização dos dados, a eficiência do sistema é prejudicada, fazendo com que a redução do número de *threads* apresente melhor resultado.

5.3. Passo 2: Execução multiprogramada

Esta seção discute os resultados da execução dos *scripts* gerados na aplicação do *Passo 2* do MultGraph. Esse passo é responsável por definir a alocação das aplicações nos núcleos do sistema assim como sua ordem de execução. Na Figura 4 nós apresentamos o tempo de execução nos 4 diferentes *batches*/conjuntos de aplicações (grupos de barras). Os resultados são apresentados para os diferentes sistemas e normalizados pelo *baseline*

Tabela 6. *#threads* com melhor eficiência para cada algoritmo/grafo.

	Intel32						Intel64						AMD128					
	BC	**BFS**	**CC**	**PR**	**SSSP**	**TC**	**BC**	**BFS**	**CC**	**PR**	**SSSP**	**TC**	**BC**	**BFS**	**CC**	**PR**	**SSSP**	**TC**
youtube	1	8	1	8	1	16	8	8	8	1	1	8	1	8	1	1	1	40
amazon	8	24	1	24	1	32	8	8	1	1	1	8	1	8	8	1	1	16
citPatents	8	8	1	24	1	32	8	8	1	8	8	1	1	1	1	8	1	16
Google	1	16	1	32	1	32	1	8	1	8	1	8	8	8	1	8	1	40
Texas	1	1	32	1	1	16	1	1	8	8	1	1	1	1	32	1	1	8
Pennsylvania	1	1	32	1	1	24	1	8	8	1	1	1	1	1	32	1	1	1
California	1	1	32	1	1	16	1	1	1	1	1	1	8	1	1	1	1	8
wikitalk	1	16	1	1	8	32	1	8	1	1	1	8	1	1	1	1	1	80
orkut	1	8	1	32	1	24	1	8	1	1	1	8	8	8	1	16	8	16
Berk	1	1	8	8	1	8	1	1	8	8	1	8	1	1	8	1	1	8

(as execuções das aplicações em sequência sempre usando o máximo número de *threads*), logo quanto maior as barras, melhor é o desempenho do MultGraph.

Como o MultGraph baseia seu processo de otimização no cálculo da eficiência, o melhor número de *threads* encontrado para cada aplicação vai ser reduzido na maioria dos casos (ver Tabela 6). Isso beneficia a execução de vários algoritmos de grafos simultaneamente, o que implica em melhora na eficiência do sistema e no tempo de execução total das aplicações. ***No geral***, considerando todos os conjuntos de aplicações avaliados, o MultGraph apresenta ganhos de $1,91\times$ no Intel32, $9,21\times$ no Intel64 e $2,45\times$ no AMD128 para os tempos de execução. Considerando cada conjunto de aplicações individualmente, o MultGraph melhora em até $3,06\times$, $16,53\times$, $2,10\times$ e $29,48\times$ o tempo de execução dos quatro respectivos conjuntos.

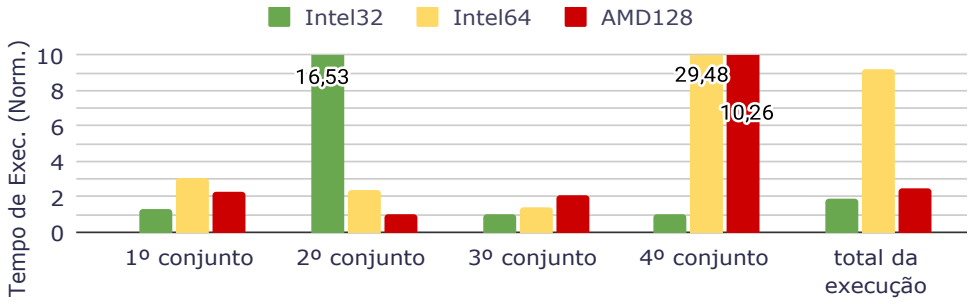


Figura 4. Resultados de tempo de execução nas soluções obtidas pelo MultGraph com relação ao *baseline* (valores acima de 1 indicam em quantas vezes o MultGraph é melhor).

6. Trabalhos Relacionados

Nessa seção, apresentamos trabalhos da literatura que focam sua otimização na: *(i)* execução multiprogramada de aplicações em sistemas multiprocessados; *(ii)* análise e otimização de algoritmos de grafos.

(i) Georgios Varisteas [Varisteas 2015] propõe uma solução com o objetivo de otimizar da taxa de transferência das aplicações que estão executando concorrentemente. Sudarsan e Ribbens [Sudarsan and Ribbens 2016] apresentam um *framework* que torna dinâmico o dimensionamento das aplicações. Este dimensionamento é baseado em cenários e estratégias onde prioriza-se o menor tempo de execução utilizando todos os recursos disponíveis e em uma outra situação prioriza-se as execuções de alta ou baixa prioridade utilizando os recursos dentro de um limite pré-determinado. Jorge González-Domínguez [Jorge González-Domínguez and Touriño 2012] desenvolve um algoritmo que usa o Servet *benchmark suite* para o mapeamento de aplicações paralelas em sistemas *multicore* e que analisa o seu impacto por meio de três modelos de programação paralela diferentes: troca de mensagens, memória compartilhada e espaço de memória global particionado. O trabalho de [da Silva et al. 2021] propõe o SRA (*Smart Resource*

Allocation), um algoritmo que otimiza tempo de execução das aplicações e o consumo de energia do sistema por executar simultaneamente várias aplicações, cada uma com os seus melhores número de *threads*.

(ii) O trabalho de [Shun and Blelloch 2013] propõe Ligra, um modelo de programação de memória compartilhada para algoritmos de grafos que ajusta automaticamente a maneira como o grafo é avaliado de acordo com um limiar provido pelo programador. O trabalho de [Zhang et al. 2015] realiza um extenso estudo sobre os impactos da execução de algoritmos de grafos em sistemas NUMA. Com base nisso, eles propõe Polymer, um *framework* que otimiza o processamento baseando-se na localidade dos dados e em sua distribuição nas memórias do sistema. Amitabha Roy [Roy et al. 2013] propõe X-Stream, um *framework* para processamento de grafos que utiliza o modelo *scatter-gather* centrado em arestas. Por ser capaz de evitar acessos aleatórios à memória principal, X-Stream apresenta boa escalabilidade conforme o aumento no número de núcleos do sistema.

Contribuição: Por mais que diversos trabalhos tenham proposto a otimização da performance de aplicações paralelas, ao avaliar a sua execução simultânea em processadores *multicore (i)*, com base nos trabalhos avaliados, nós somos os primeiros a considerar a otimização de algoritmos de grafos, tendo em vista tanto o tempo de execução do algoritmo quanto a eficiência do sistema. Além do mais, nossa proposta ortogonaliza os esforços de *(ii)*, uma vez que o MultGraph melhora a performance dos algoritmos de processamento de grafos sem requerer qualquer mudança em seus códigos fonte (i.e., o algoritmo BFS nas diferentes implementações do Ligra, X-Stream ou Polymer pode ter melhor tempo de execução em utilizar o nosso *framework*).

7. Conclusão

Neste trabalho nós mostramos que limitações de escalabilidade dos algoritmos de processamento de grafos variam de acordo com as suas características e as topologias dos grafos processados. Com base nisso, nós propomos o MultGraph: uma estratégia de multiprogramação que, dado conjunto de aplicações e o multiprocessador alvo, encontra a melhor configuração de algoritmo/grafo que deve ser executada simultaneamente de forma a melhorar o tempo de execução e a eficiência do sistema. Os resultados experimentais considerando três diferentes processadores multicores (Intel e AMD) mostraram que o MultGraph melhora em até $9,21\times$ o tempo de execução dos algoritmos em relação à execução padrão de aplicações em sistemas HPC. Nossas perspectivas de trabalhos futuros incluem uma analise mais detalhada do comportamento de execução das aplicações com base em dados de baixo nível (de processamento e acessos à memória) e em uma análise do comportamento do MultGraph quando na otimização de diferentes *frameworks* de processamento de grafos.

Agradecimentos

Esse estudo foi financiado em parte pela Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, a Fundação de Amparo à Pesquisa do Estado do RS (FAPERGS) e o Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq). Alguns experimentos desse trabalho usaram a infraestrutura do PCAD, http://gppd-hpc.inf.ufrgs.br/, do INF/UFRGS.

Referências

Beamer, S., Asanović, K., and Patterson, D. (2015a). The gap benchmark suite. *arXiv preprint arXiv:1508.03619*.

Beamer, S., Asanovic, K., and Patterson, D. (2015b). Locality exists in graph processing: Workload characterization on an ivy bridge server. In *ISWC*, pages 56–65. IEEE.

da Silva, V., Medeiros, T., Rocha, H., Luizelli, M., Rossi, F., Beck, A. C., and Lorenzon, A. (2020). Análise da execução concorrente de aplicações paralelas em arquiteturas multicore. In *WSCAD*, pages 61–72. SBC.

da Silva, V. S., Nogueira, A. G., de Lima, E. C., de A. Rocha, H. M., Serpa, M. S., Luizelli, M. C., Rossi, F. D., Navaux, P. O., Beck, A. C. S., and Francisco Lorenzon, A. (2021). Smart resource allocation of concurrent execution of parallel applications. *CCPE*, page e6600.

Eager, D., Zahorjan, J., and Lazowska, E. (1989). Speedup versus efficiency in parallel systems. *IEEE TC*, 38(3):408–423.

Gleich, D. F. (2014). Pagerank beyond the web.

Heidari, S., Simmhan, Y., Calheiros, R., and Buyya, R. (2018). Scalable graph processing frameworks: A taxonomy and open challenges. *CSUR*, 51:1–53.

Jorge González-Domínguez, Guillermo L. Taboada, B. B. F. M. J. M. and Touriño, J. (2012). Automatic mapping of parallel applications on multicore architectures using the servet benchmark suite. *CEE*, 38:258–269.

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

Lorenzon, A. F. and Beck Filho, A. C. S. (2019). *Parallel computing hits the power wall: principles, challenges, and a survey of solutions*. Springer Nature.

Mofrad, M. H., Melhem, R., Ahmad, Y., and Hammoud, M. (2020). Graphite: a numa-aware hpc system for graph analytics based on a new mpi* x parallelism model. *Proceedings of the VLDB Endowment*, 13(6):783–797.

Raasch, S. E. and Reinhardt, S. K. (2003). The impact of resource partitioning on smt processors. In *PACT*, pages 15–25.

Rocha, H. M. G. d. A., Schwarzrock, J., Lorenzon, A. F., and Beck, A. C. S. (2021). Boosting graph analytics by tuning threads and data affinity on numa systems. In *PDP*, pages 161–168. IEEE.

Roy, A., Mihailovic, I., and Zwaenepoel, W. (2013). X-stream: Edge-centric graph processing using streaming partitions. page 472–488.

Shun, J. and Blelloch, G. E. (2013). Ligra: a lightweight graph processing framework for shared memory. In *ACM PPoPP*, pages 135–146.

Sudarsan, R. and Ribbens, C. J. (2016). Combining performance and priority for scheduling resizable parallel applications. *JPDC*, 87:55–66.

Suleman, M. A., Qureshi, M. K., and Patt, Y. N. (2008). Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps. *SIGARCH CAN*, 36(1):277–286.

Varisteas, G. (2015). *Effective cooperative scheduling of task-parallel applications on multiprogrammed parallel architectures*. PhD thesis. QC 20151016.

Yan, D., Bu, Y., Tian, Y., and Deshpande, A. (2017). Big graph analytics platforms. *FTD*, 7(1-2):1–195.

Zhang, K., Chen, R., and Chen, H. (2015). Numa-aware graph-structured analytics. *SIGPLAN Not.*, 50(8):183–193.