Processamento Distribuído de Grafos: Modelagem de Desempenho e Escalonamento de Tarefas Moldáveis

Daniel Presser[1], **Frank Siqueira**[1,2]

[1]Programa de Pós-Graduação em Ciência da Computação (PPGCC)
[2]Departamento de Informática e Estatística (INE) – Centro Tecnológico (CTC)
Universidade Federal de Santa Catarina (UFSC) – Florianópolis - SC, Brasil

Abstract. *The ever growing datasets observed in modern applications also applies to datasets modeled as graphs. Several large scale distributed graph processing models, such as Pregel, have been proposed. These models are designed to run in large clusters, where the resources must be allocated efficiently. In this paper we present a prediction model and a scheduler for distributed graph processing jobs. The scheduler treats the jobs as moldable tasks and, based on the predictions, allocates the best number of workers to each job in order to minimize makespan. Experimental results show that the prediction model has accuracy close to 90%, which allows the scheduler to work within the theoretical approximation limits of the optimal makespan.*

Resumo. *O crescimento contínuo das bases de dados observado em aplicações atuais também se aplica àquelas modeladas como grafos. Neste contexto, diversos modelos para processamento distribuído de grafos de larga escala foram propostos, como o Pregel. Estes modelos assumem o uso de clusters de computadores, nos quais as tarefas precisam ser alocadas de maneira eficiente. Neste trabalho é apresentado um modelo de predição de desempenho e um escalonador de jobs de processamento de grafos. O escalonador trata os jobs a escalonar como tarefas moldáveis, encontrando a melhor alocação de processadores, com base nas predições, para otimizar o tempo total de processamento (makespan). São apresentados resultados experimentais demonstrando que o modelo de desempenho tem precisão média de 90% que permite ao escalonador se manter dentro dos limites teóricos de aproximação do makespan ótimo.*

1. Introdução

Grafos são uma maneira natural de representar dados que consistem de entidades conectadas e suas relações. Esta estrutura de dados é comum em muitas aplicações do mundo real, incluindo redes sociais, a Internet, e sistemas biológicos, para citar alguns exemplos. Seguindo o crescimento contínuo das bases de dados observado nos últimos anos, tanto pesquisadores quanto a indústria vem propondo modelos para processar grandes volumes de dados modelados como grafos. Nos últimos anos, muitos modelos deste tipo foram propostos, como Pregel [Malewicz et al. 2010], GraphX [Xin et al. 2013] e PowerGraph [Gonzalez et al. 2012]. Estes sistemas são planejados para serem executados em grandes *clusters* de computadores, que podem chegar aos milhares de nós e processar grafos da ordem de bilhões de vértices e arestas.

Considerando o custo de manter grandes *clusters*, provedores de computação em nuvem têm se mostrado uma forma econômica para organizações serem capazes de lidar

com grandes volumes de dados. Com a evolução destes provedores, novos serviços, caracterizados por oferecerem uma infraestrutura gerenciada elástica para uso de plataformas específicas, vem sendo propostos. Existem alguns serviços deste tipo que oferecem plataformas para processamento distribuído de larga escala. Como exemplo é possível citar o Amazon Elastic MapReduce[1], e o IBM BigInsights[2] para aplicações do Hadoop [White 2012] e Spark [Zaharia et al. 2016]. A Google também oferece o Google DataFlow[3], com suporte a diversos modelos de processamento. Em serviços deste tipo, o usuário submete código escrito para a plataforma em questão (Hadoop, Spark e outros) e os dados que deseja analisar, podendo usar mais ou menos recursos de acordo com a necessidade. Neste contexto, diversos usuários competem pelos mesmos recursos, por isso é necessário escalonar o uso dos recursos de maneira eficiente.

O surgimento deste tipo de serviço despertou o interesse de pesquisadores, resultando na criação de vários modelos de predição de desempenho e escalonamento voltados para modelos de processamento paralelo e distribuído de propósito geral, como Hadoop e Spark. No entanto, há poucos trabalhos na literatura que tratam destas questões no contexto de processamento distribuído de grafos. A pesquisa de modelos de predição de desempenho está limitada à avaliação de custo/benefício considerando limitações de rede, o que é incomum em cenários de nuvem. Já os modelos de escalonamento limitam-se a tratar *jobs* de processamento de grafos como tarefas rígidas, nas quais o número de processadores utilizado é conhecido pelo usuário de antemão.

Neste trabalho é apresentado um modelo de predição de desempenho para o GPS [Salihoglu and Widom 2013], um modelo de processamento distribuído de grafos baseado no Pregel, da Google. Este modelo tem por objetivo produzir predições de desempenho a serem usadas por um escalonador para *jobs* de processamento de grafos, de forma que um conjunto de *jobs* possa ser escalonado de maneira moldável: ou seja, ajustando-se o número de processadores a serem usados para otimizar o tempo total (*makespan*) de computação do conjunto de *jobs* de processamento de grafos.

As principais contribuições deste trabalho são: (a) Um modelo de predição de desempenho para o GPS; (b) Um escalonador para *jobs* de processamento de grafos, tratando tais tarefas como moldáveis; (c) Uma avaliação experimental do modelo e do escalonador realizada na Amazon Web Services.

O restante deste artigo é organizado da seguinte forma. A Seção 2 apresenta os principais conceitos sobre processamento distribuído de grafos. A Seção 3 apresenta o modelo de desempenho proposto, enquanto a Seção 4 descreve o mecanismo de escalonamento de tarefas moldáveis. Os experimentos realizados são descritos na Seção 5. Na seção 6 são apresentados os trabalhos relacionados a este. Por fim, nas seção 7 são apresentadas as conclusões deste trabalho e as possibilidades de trabalhos futuros.

2. Processamento Distribuído de Grafos

O crescimento das bases de dados observado nos últimos anos deu origem a diversos modelos de processamento distribuído e paralelo cujo objetivo é realizar análises e processamento em grandes volumes de dados. No entanto, os primeiros modelos de propósito

[1] `https://aws.amazon.com/pt/emr/`

[2] `https://www.ibm.com/analytics/us/en/technology/biginsights/`

[3] `https://cloud.google.com/dataflow/`

geral, como MapReduce e Spark, não se mostraram adequados para dados modelados como grafos [Malewicz et al. 2010]. Esta observação levou à criação de modelos de processamento paralelo e distribuído específicos para dados modelados como grafos. Dentre os modelos propostos, Pregel [Malewicz et al. 2010], da Google, é um dos mais importantes. Embora seja uma solução proprietária, diversos modelos de código aberto baseados no Pregel também foram propostos, como GPS [Salihoglu and Widom 2013], Apache Giraph [Avery 2011] e Apache Hama [Seo et al. 2010]. A seguir serão detalhadas as características do Pregel e do GPS, a implementação usada neste trabalho.

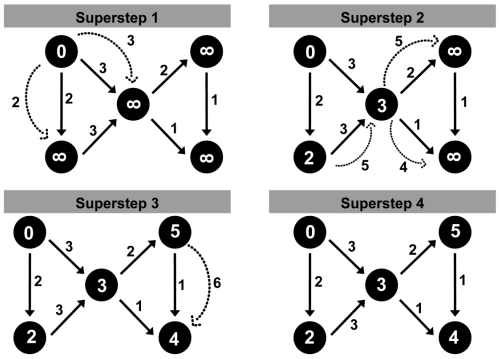
Pregel segue um modelo de programação paralela baseado no *Bulk Syncrhonous Parallel* (BSP). O BSP é caracterizado pela existência de uma barreira de sincronização, chamada *superstep*. No BSP, o programa executa um passo de processamento sobre dados numa memória local, depois há um passo de comunicação no qual os processos distribuem o resultado do passo anterior e atualizam sua memória local. Apenas quando todos os processos finalizaram estes dois passos um novo *superstep* é iniciado. Este modelo reduz as preocupações por parte do programador com condições de corrida e outros problemas de sincronização no código. No entanto, balanceamento de carga se torna preocupação, já que se um processo levar mais tempo que os demais para realizar um passo, todos os outros processadores podem ficar ociosos.

No Pregel, um grafo é composto por um conjunto de vértices (ou nós). Cada vértice é composto por um identificador único, um estado (ativo ou inativo), um valor, uma fila de mensagens e uma lista de adjacências. A lista de adjacências representa as arestas direcionadas que ligam um vértice aos outros do grafo, e cada aresta também pode ter um valor definido pelo usuário.

Os programas no Pregel são expressos como uma função definida pelo usuário que recebe um vértice como argumento. Esta função é executada em todos os vértices do grafo, de maneira paralela, durante cada *superstep* da computação. Nesta função o usuário pode modificar o estado do vértice, modificar o valor do vértice e receber e enviar mensagens para outros vértices. As mensagens enviadas no *superstep* S são recebidas pelo vértice de destino apenas no superstep $S + 1$, de acordo com o modelo BSP. Isso garante o paralelismo da computação de cada vértice durante um *superstep*.

Um *superstep* termina quando a função tiver sido executada em todos os vértices do grafo e todas as mensagens enviadas durante o *superstep* tiverem sido recebidas nos destinos. A função definida pelo usuário só é executada em vértices que estiverem **ativos** no *superstep*. Um vértice pode ser desativado pela função definida pelo usuário, entretanto, caso um vértice inativo receba uma mensagem, ele é ativado automaticamente pelo sistema. A computação continua até que todos os vértices do grafo estejam inativos.

A figura 2 apresenta o algoritmo *Single Source Shortest Path* (SSSP) no modelo do Pregel e um exemplo de execução dos *supersteps* em um grafo. No primeiro *superstep*, todos os vértices estão ativos, mas apenas o vértice de origem 0 envia mensagens para seus vértices adjacentes, somando seu valor ao da aresta que os liga. No próximo *superstep*, os vértices recebem essa mensagem, atualizam seu valor com o valor recebido (caso seja menor que o valor atual) e enviam novas mensagens para seus vizinhos, somando o novo valor ao valor da aresta que os liga. O processo continua até que todos os vértices tenham sido atualizados com a menor distância do vértice de origem.



Algorithm 1 Single Source Shortest Path

```
1: function COMPUTE(vertex)
2:     minDist ← origem(vertex.id) ? 0 : +∞
3:     for msginvertex.messageQueue do
4:         minDist ← min(msg.value, minDist)
5:     if minDist < vertex.value then
6:         vertex.value ← minDist
7:         for edgeinvertex.neighbors do
8:             msgValue ← vertex.value + edge.value
9:             sendToNeighbor(edge, msgValue)
10:    vertex.active ← False
```

Figure 1. SSSP no modelo do Pregel sendo executado em um grafo de exemplo

Pregel é composto por um processo *master* e vários processos *workers*. O processo *master* é responsável por coordenar a computação, indicando quando cada novo *superstep* deve começar. Já os *workers* realizam o processamento propriamente dito, executado a função definida pelo usuário nos vértices sob sua responsabilidade. No Pregel o grafo deve ser carregado na memória dos *workers* para ser processado, o que impõe limites mínimos no número de *workers* necessários dependendo do tamanho do grafo. Ao iniciar a computação, o *master* particiona o grafo de entrada de acordo com o número de *workers* disponíveis. Cada *worker* carrega sua partição, que contém os vértices que estarão sob sua responsabilidade. Após a carga finalizar, os *supersteps* iniciam e são executados enquanto houver vértices ativos ou mensagens a serem enviadas.

GPS [Salihoglu and Widom 2013] é uma implementação *open source* de um modelo de processamento de grafos baseado no Pregel. GPS oferece as mesmas abstrações e segue o mesmo modelo de computação do Pregel. Entretanto, GPS oferece novas funcionalidades, dentre as quais se destaca o mecanismo de *Large Adjacency List Partition* (LALP), que particiona as listas de adjacências de vértices de grau muito alto entre os *workers* da computação. Este mecanismo auxilia no balanceamento de carga do sistema.

3. Modelagem de Desempenho

Estimar o tempo de execução de algoritmos no modelo do Pregel é desafiador. Algoritmos em grafos normalmente apresentam baixa localidade de dados e um alto volume de acesso a dados em relação ao processamento realizado [Lumsdaine et al. 2007]. Entretanto, ao analisar cuidadosamente as características dos algoritmos e do modelo do Pregel/GPS é possível chegar em um modelo estatístico preciso o suficiente para estimar o desempenho no contexto de um escalonador moldável. Nesta seção, inicialmente são apresentados os detalhes dos algoritmos estudados neste trabalho. Depois, os detalhes do modelo de previsão são descritos.

O modelo apresentado aqui supõe um *cluster* homogêneo. Essa é uma suposição razoável dada a natureza do BSP: em um *cluster* heterogêneo, devido à barreira de sincronização, máquinas mais lentas atrasariam toda a computação. Também se assume que cada máquina executará apenas um *worker*, portanto os termos *processador* e *worker* podem ser usados de maneira intercambiável.

3.1. Algoritmos para Análise de Dados

No modelo do Pregel, o objetivo de uma computação é realizar algum tipo de análise em um grafo de entrada. O tipo de análise é definido pelo algoritmo que será executado durante a computação. Os algoritmos são definidos como uma função a ser executada em cada vértice do grafo de entrada, durante cada *superstep*. De maneira geral, esta função costuma ter três partes: primeiro, mensagens recebidas do *superstep* anterior são computadas. Depois, possivelmente um novo valor é atribuído ao vértice e mensagens são enviadas para seus vizinhos para notificá-los da atualização. Esta observação inclusive deu origem a outras abstrações para processamento de grafos, como o modelo *Gather-Apply-Scatter* (GAS) do PowerGraph [Gonzalez et al. 2012].

Esta característica faz que diferentes algoritmos no modelo do Pregel apresentem padrões de processamento e envio de mensagens semelhantes. Por isso, os algoritmos comumente são agrupados em categorias [Han et al. 2014]. A existência destes padrões de processamento e envio de mensagens sugere que o modelo proposto aqui seja aplicável a outros algoritmos além dos estudados neste trabalho, bastando que sejam realizadas etapas de treinamento do modelo de previsão específicas para esses novos algoritmos.

Neste trabalho serão explorados os algoritmos *Single Source Shortest Path* (SSSP), *PageRank* (PR) e *Weak Connected Components* (WCC). O SSSP já foi detalhado na seção 2, e serve como um exemplo de algoritmo da categoria de busca sequencial. O PageRank [Page et al. 1999] pode ser visto no algoritmo 2 e é um exemplo de caminhada aleatória (*random walk*). Inicialmente os vértices calculam seu rank e propagam para seus vizinhos. Depois, seus ranks são atualizados a cada *superstep* com base nas mensagens recebidas, até que o número de *supersteps* estabelecido pelo usuário seja alcançado. O WCC pode ser visto no algoritmo 3 e é um exemplo de busca paralela. Inicialmente cada vértice começa com um número igual ao seu ID. Depois, cada vértice vai se juntando aos vizinhos que possuírem IDs menores até que sejam identificadas todas as componentes fracamente conexas do grafo.

Tanto no WCC quanto no SSSP, o número de *supersteps* necessários para completar uma computação depende da topologia do grafo de entrada. No caso do SSSP, depende também de qual vértice é a origem da busca. Já o PageRank tem um número fixo estabelecido pelo usuário. Os algoritmos também apresentam diferenças no número de vértices ativos durante a computação. No PageRank, todos os vértices permanecem ativos durante toda a computação. No WCC, todos os vértices começam ativos, mas com o tempo o número de vértices ativos vai caindo, conforme os vértices se juntam nas componentes conexas do grafo. No caso do SSSP, no início poucos vértices estão ativos. Com o tempo o número de vértices ativos cresce conforme a distância se propaga e, ao final, diminui até que todos os vértices tenham sido alcançados. Estas considerações sobre o número de vértices ativos são importantes, pois afetam diretamente o tempo necessário para completar uma computação.

3.2. Modelo de Previsão

Uma computação no Pregel/GPS pode ser divida em três passos: primeiro o grafo de entrada é particionado e carregado do HDFS para a memória dos *workers*. Depois, os *supersteps* são executados e finalmente o grafo resultante da computação é armazenado

Algorithm 2 PageRank

```
1: function COMPUTE(vertex)
2:     if superstepNo = 1 then
3:         vertex.value ← 1/vertex.numOfNeighbors
4:         sendToNeighbors(vertex.value)
5:     else
6:         sum ← 0
7:         for msginvertex.messageQueue do
8:             sum ← sum + msg.value
9:         vertex.value ← (0.85 × sum/vertex.numOfNeighbors) +
10:            (0.15/numOfVertices)
11:        sendToNeighbors(vertex.value)
12:        if superstepNo = maxIter then
13:            vertex.active ← False
```

Algorithm 3 Weak Connected Components

```
1: function COMPUTE(vertex)
2:     if superstepNo = 1 then
3:         vertex.value ← vertex.id
4:         sendToNeighbors(vertex.value)
5:     else
6:         minValue ← vertex.value
7:         for msginvertex.messageQueue do
8:             if msg.value < minValue then
9:                 minValue ← msg.value
10:        if minValue < vertex.value then
11:            vertex.value ← minValue
12:            sendToNeighbors(vertex.value)
13:        else
14:            vertex.active ← False
```

Figure 2. Algoritmos PageRank e WCC

Table 1. Símbolos do modelo de previsão de desempenho

Símbolo	**Descrição**
V	Número de vértices no grafo de entrada
E	Número de arestas no grafo de entrada
N	Número de *workers* alocados para um *job*
D	Densidade de arestas do grafo ($\frac{E}{V}$)
S	Número de *supersteps* de uma computação
V_s	Número de vértices ativos no *superstep* s

também no HDFS. A equação 1 calcula o tempo total previsto para execução destes passos. Os símbolos usados nesta seção estão descritos na tabela 1.

$$T_{total} = T_{load} + T_{supersteps} + T_{save} \tag{1}$$

Para carregar o grafo, por padrão o GPS usa um esquema de *round-robin* na distribuição dos vértices para os *workers*. Neste esquema, o vértice v é atribuído ao *worker* $v \bmod N$. O carregamento é feito de maneira paralela entre os *workers*: cada *worker* obtém um trecho do arquivo de entrada (ou de um dos arquivos, se a entrada estiver particionada) no HDFS, carrega os dados do arquivo e distribui os vértices para os outros *workers* conforme o esquema *round-robin*. Ao final da computação, cada *worker* salva os dados dos vértices de sua partição também no HDFS. Dessa forma, os tempos de carga T_{load} e de armazenamento T_{save} do grafo só dependem do tamanho do grafo (em termos de vértices e arestas) e do número de *workers*. Por isso, são aproximados com boa precisão por regressões simples, que serão omitidas por uma questão de espaço.

Já o tempo para computação dos *supersteps* $T_{supersteps}$ demanda uma análise mais detalhada. Inicialmente pode-se considerar o tempo $T_{supersteps}$ como a soma do tempo de cada *superstep* individual $T_{superstep}$. Portanto, faz sentido estabelecer um modelo para estimar o tempo de cada *superstep* individual antes de se chegar ao tempo total da computação. Seguindo o modelo do Pregel, em cada *superstep* a função definida pelo usuário é executada em cada vértice ativo do grafo. Como os vértices estão distribuídos igualmente entre os *workers*, são $\frac{V_S}{N}$ execuções da função por *superstep*.

A cada vértice em que a função é executada, mensagens podem ser produzidas. Como os vértices estarão distribuídos entre os *workers*, uma parte das mensagens pro-

duzidas terá que ser transmitida pela rede, já que o vértice de origem estará num *worker* diferente do vértice de destino. Como os vértices são distribuídos de maneira *round-robin*, é seguro assumir que os vértices estão distribuídos uniformemente entre os *workers*. Considerando o número de vértices ativos no *superstep* V_S e a densidade de arestas D, haverá $V_S \times D$ arestas ativas no *superstep*. Portanto, cada *worker* deverá processar $\frac{V_S \times D}{N}$ arestas. Entretanto, quando se observa um único *worker*, nota-se que ele possui a proporção de $\frac{N-1}{N}$ destas arestas ativas. Multiplicando estes dois termos é possível estimar o número mensagens que cruzarão *workers*, conforme a equação 2.

$$\frac{V_S \times D \times (N-1)}{N^2} \tag{2}$$

Com base nestas informações é possível estabelecer um modelo que capture as variações que ocorrem devido à implementação do algoritmo GPS, às condições da rede e a diferenças entre os grafos de entrada. Entretanto, tentar capturar todas estas variações para todos os *supersteps* resulta em um modelo complexo, como o proposto em [Li et al. 2017], com muitas variáveis que precisam ser inicializadas por um especialista para alcançar uma boa precisão, o que limita sua aplicação prática.

Em vez disso, o modelo proposto neste trabalho procura focar apenas no *superstep* em que o maior número de vértices está ativo. Com base nessa estimativa, calcula-se o tempo total usando um segundo modelo ajustado à variação do número de vértices ativos durante a computação para cada tipo de algoritmo. O modelo proposto para estimar o tempo necessário para completar um *superstep* T_{modelo} está descrito na equação 3.

$$T_{model} = \left(\frac{V_s}{N} \times v_1\right) + \left(\frac{V_S \times D \times (N-1)}{N^{v_3}} \times v_2\right) + (D \times v_4) \tag{3}$$

O treinamento do modelo consiste primeiramente em calibrar os parâmetros $[v_1, v_2, v_3, v_4]$ da equação 3. A equação é calculada para cada cenário usado no treinamento e, a partir do resultado, é calculado o erro entre o valor previsto pela equação e o real da execução, como $|\frac{T_{real}-T_{model}}{T_{real}}|$. É então aplicado um modelo não linear de mínimos quadrados[4] para estimar os parâmetros, com o objetivo de minimizar a média dos erros. Os parâmetros calibrados são armazenados para serem usados posteriormente na geração de previsões.

Diferentemente do modelo de previsão proposto em [Li et al. 2017], que necessita de um especialista para definir os valores iniciais dos parâmetros, no modelo apresentado aqui esta inicialização se dá de forma automática. Do GPS é obtido, por *superstep*, uma média do tempo de execução da função definida pelo usuário em todos os vértices do grafo e o tempo total do *superstep*. O primeiro valor é usado para iniciar v_1, que procura capturar o tempo necessário para executar a função do usuário em todos os vértices do grafo em cada *worker*. Já a diferença entre os dois corresponde ao tempo necessário para enviar as mensagens e sincronizar todos os *workers*. Este valor é usado para iniciar v_2, que procura capturar o tempo de envio das mensagens pela rede. Dado que o envio de mensagens não acontece apenas no final do *superstep*, o valor de v_1 estará superestimado e o de v_2 subestimado, mas são bons o suficiente para que o modelo convirja para uma

[4] `https://stat.ethz.ch/R-manual/R-devel/library/stats/html/nls.html`

previsão precisa. O parâmetro v_3 tem por objetivo capturar desbalanceamentos entre os *workers* e é iniciado com 2, seguindo o observado na equação 2. Já v_4 é iniciado com 1, pois é usado apenas para acomodar diferenças na topologia dos grafos (diferenças nas densidades de arestas) usados no treinamento.

Depois de preparar um modelo para estimar o tempo de um *superstep*, é necessário preparar também um modelo para o tempo total dos *supersteps* $T_{supersteps}$. Para isso, durante o treinamento calcula-se a proporção P_{model} do tempo total em relação ao tempo do *superstep* com o maior número de vértices ativos ($T_{max(v)}$), normalizado pelo número total de *supersteps* da computação, como descrito na equação 4. Este cálculo é feito para cada grafo que está sendo usado no treinamento e para cada nível de *workers* W, e fica armazenado para ser usado posteriormente na geração das previsões.

$$P_{model} = \frac{T_{supersteps}}{T_{max(v)} \times S} \tag{4}$$

Depois de realizado o treinamento, para estimar o tempo de execução de uma computação, inicialmente escolhe-se o modelo treinado com grafos mais parecidos com o grafo que se deseja estimar, considerando densidade de arestas e tamanho do grafo. Com base na equação 3, nos parâmetros do grafo de entrada, no número de *workers* que se deseja usar e nos parâmetros ajustados no treinamento ($[v_1, v_2, v_3, v_4]$), calcula-se uma previsão para o *superstep* T_{model} com o maior número de vértices ativos. Depois, este valor é multiplicado pela média das proporções calculadas durante o treinamento $\overline{P_{model}}$ e pelo número de *supersteps* previsto para o grafo em questão S_p, conforme equação 5.

$$T_{supersteps} = T_{model} \times \overline{P_{model}} \times S_p \tag{5}$$

O número de *supersteps* previsto pode ser fornecido pelo usuário ou calculado com base na média do número de *supersteps* das execuções usadas no treinamento do modelo. Com base nessa previsão e nos valores previstos de T_{load} e de T_{save}, chega-se ao tempo total T_{total} previsto, conforme equação 1.

4. Escalonamento de Tarefas

Quando se trata de escalonamento de computação paralela e distribuída, as tarefas costumam ser classificadas em três tipos: rígidas, moldáveis e maleáveis. Tarefas rígidas estabelecem um número específico de processadores necessários para que sejam executadas. Em tarefas moldáveis, o número de processadores pode ser ajustado conforme a necessidade e há uma função de *speedup* associada à tarefa, que varia de acordo com o número de processadores. As tarefas maleáveis seguem o mesmo modelo das moldáveis, mas com a possibilidade de variar o número de processadores durante a computação [Leung 2004].

É fácil perceber que o modelo de computação do Pregel é bem adequado à categoria de tarefas moldáveis: é possível variar o número de processadores alocados a uma tarefa distribuindo mais ou menos vértices a cada *worker* no particionamento inicial do grafo de entrada. Também é possível perceber que a função de *speedup* neste tipo de tarefa não é linear devido à comunicação entre os vértices (ver equação 2). Já a categoria de tarefas maleáveis não é adequada ao modelo de grafos, pois variar o número de *work-*

ers durante a computação implicaria em redistribuir os vértices entre eles. Nesse caso, o *overhead* dificilmente compensaria a operação.

O problema de se alocar tarefas moldáveis independentes para otimizar o *makespan* é considerado NP-*Hard*, portanto, a literatura existente procura heurísticas que permitam se chegar a uma alocação eficiente em um tempo polinomial. De maneira geral, as soluções da literatura são baseadas em duas fases: primeiro é feita uma alocação de processadores para cada tarefa, e depois as tarefas são escalonadas como se fossem rígidas. Neste trabalho é usado um algoritmo que dá uma garantia de aproximação de 2 vezes da alocação ótima [Turek et al. 1992]. Detalhes podem ser observados no artigo original, mas resumidamente, assume-se um conjunto de n tarefas $\{T_1, T_2, ..., T_n\}$ a serem alocadas em um conjunto de m processadores $\{P_1, P_2, ..., P_m\}$. Cada tarefa T_j está associada a uma função $t_j(\beta_j)$ que determina o tempo de processamento da tarefa dada a alocação $\beta_j \in \{1, ..., m\}$ de processadores. Além disso, também existe uma função que determina o *trabalho* realizado por uma tarefa T_j com i processadores alocados, como $w_j(i) = t_j(i) \times i$ (ou seja, tempo multiplicado pelo número de processadores).

O objetivo do algoritmo é alcançar um equilíbrio entre o trabalho total realizado pelas tarefas (expresso pela função w_j) e o tempo da tarefa mais longa. Para isso, são geradas diversas alocações até que se alcance esse equilíbrio. A primeira alocação é obtida minimizando o trabalho realizado por cada tarefa: $\beta_j^1 = \arg\min_i w(i)$. A k-ésima β_j^k alocação é obtida encontrando-se a tarefa com o maior tempo de execução na alocação anterior $k-1$ e aumentando o número de processadores associados a ela, até encontrar uma alocação com um tempo de execução menor e o menor trabalho possível. Após encontrar a melhor alocação possível, o escalonamento pode ser resolvido como um problema de empacotamento de retângulos rígidos, mantendo a garantia de aproximação de 2 vezes o *makespan* ótimo. Neste caso, é usado um algoritmo de escalonamento de listas tipo *First Fit*. Os algoritmos base da literatura foram alterados incluindo restrições ao número de processadores que podem ser alocados a cada tarefa, levando em consideração limitações de memória nos *workers* e o número *workers* usados no treinamento.

5. Avaliação Experimental

A avaliação experimental tem por objetivo verificar se o modelo de predição é capaz de produzir previsões precisas sobre o tempo de execução de diversos grafos em variadas configurações de *workers*. Nos experimentos foram usados 5 grafos reais, descritos na tabela 2. Os grafos foram obtidos das bibliotecas WebGraph[5] [Boldi and Vigna 2004] e SNAP [Leskovec and Krevl 2014], e foram escolhidos por representarem uma boa variação, tanto em tamanho quanto na densidade média de arestas.

O grafo *lj* representa as relações de amizade entre os usuários da rede social LiveJournal. O grafo *wiki* representa as ligações entre páginas da WikiPedia em inglês em 2013. Os grafos *indochina*, *uk* e *arabic* são resultado de *crawlers* que mapearam as páginas e links em domínios da Indochina (em 2004), Reino Unido (em 2002) e países árabes (em 2005), respectivamente.

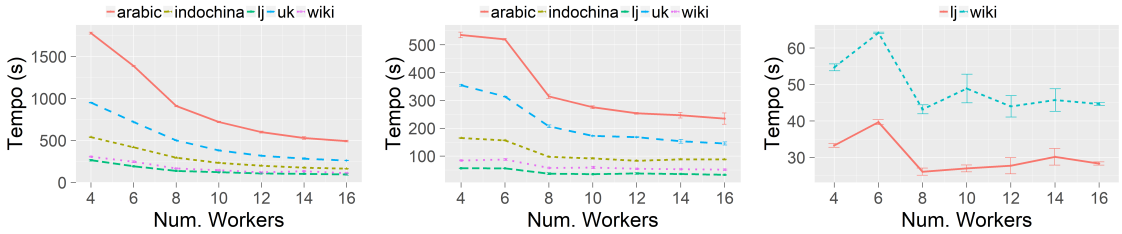
Para os experimentos foi escrito um protótipo[6] de um sistema de gerenciamento para o GPS. Este sistema é responsável por rodar as execuções de treinamento, coletar os

[5] `http://law.di.unimi.it/datasets.php`

[6] `https://github.com/presser/gps_scheduler`

Table 2. Grafos usados nos experimentos

Nome	Vértices	Arestas	V/A	Tipo
lj	4,8M	68,9M	14,23	Rede Social
wiki	4,2M	101,3M	24,09	Rede Social
indochina	7,4M	194,1M	26,18	Páginas Web
uk	18,5M	298M	16,10	Páginas Web
arabic	22,7M	639,9M	28,14	Páginas Web



(a) PageRank (b) WCC (c) SSSP

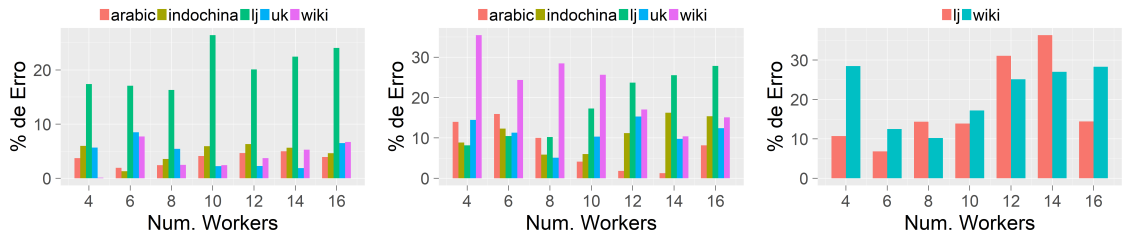
Figure 3. Execuções de Treinamento

resultados e produzir os modelos de predição descritos na seção 3. Ele também realiza o escalonamento usando os algoritmos descritos na seção 4 quando os usuários submetem um conjunto de *jobs* para serem executados.

Os experimentos foram realizados na *Amazon Elastic Cloud Compute* (ECC), usando de 2 a 16 instâncias tipo *r3.large*. Como as aplicações de processamento distribuído de grafos necessitam dos dados na memória, este tipo de instância otimizada para memória foi escolhida pois apresenta uma boa relação de custo/benefício entre capacidade de computação e memória disponível. Cada instância conta com 2 CPUs virtuais de processadores *E5-2670 v2 (Ivy Bridge)*, 15.25 GiB de memória e 32 GiB de armazenamento em SSD.

O primeiro experimento tem por objetivo avaliar a precisão das previsões geradas pelo modelo. Para isso, todos os grafos foram executados em configurações variando de 2 a 16 instâncias para os algoritmos PageRank e WCC. O SSSP é mais complexo, pois depende de se encontrar um vértice de origem que percorra uma parte considerável do diâmetro do grafo. Só foi possível encontrar um vértice adequado nos grafos *wiki* e *lj*, por isso os experimentos ficaram limitados a esses dois grafos.

Cada instância executa apenas um *worker*, exceto por uma que também executa o processo *master*. No GPS, o processo *master* é bastante leve e não houve diferença perceptível no desempenho da instância em ele foi executado. Cada execução foi replicada 3 vezes e os resultados foram coletados para treinamento do modelo. A figura 3 apresenta os resultados dessa rodada de treinamento. Em todos os casos é possível observar a característica não linear da função de *speedup* em relação ao número de *workers*. Observa-se também que, no caso do SSSP, a variação nos resultados é grande. Isso é característica desse algoritmo, pois o tempo de execução depende não só da topologia do grafo de entrada, mas também do vértice escolhido como origem.



(a) PageRank (b) WCC (c) SSSP

Figure 4. Percentuais de erro entre valores previstos e reais

Para realizar as predições vários modelos foram treinados agrupando grafos com características parecidas, como tamanho e densidade de arestas. Por exemplo, para gerar as previsões para o grafo *enwiki-2013*, foi usado um modelo treinado com os grafos *arabic-2005* e *indochina-2004*. A partir dos modelos, foram geradas previsões para todos os grafos, variando o número de *workers* de 4 a 16. A previsão foi comparada com o resultado real da execução, calculando-se o percentual do erro em relação ao tempo real. Os resultados obtidos são ilustrados na figura 4.

No geral, a média dos erros ficou próxima de 10%. Observa-se que no caso do PageRank, as previsões foram bastante precisas, com média de 7% de erro, exceto pelo grafo *lj*. Atribui-se essa diferença ao fato do grafo ser muito diferente em relação aos demais: o grafo com tamanho mais próximo, que foi usado no treinamento, é o *enwiki-2013*, que tem uma densidade de arestas muito diferente. Já no caso do WCC, o erro das previsões ficou em 13% em média, com alguma variação que é esperada dada a complexidade maior deste algoritmo. No caso do SSSP, as previsões ficaram com um erro médio de 20%, o que é esperado dada a complexidade deste algoritmo. Entretanto, é importante ressaltar que o SSSP costuma ser um algoritmo rápido, portanto, erros percentuais maiores tem efeito reduzido no tempo total da computação.

As predições são usadas pelo algoritmo de escalonamento, descrito na seção 4. O usuário submete uma lista de *jobs* para o sistema, que produz uma alocação e então um escalonamento das tarefas. Um exemplo de escalonamento pode ser visto na figura 5. Cada bloco corresponde a uma tarefa, e cada tarefa é um *job* de execução dos algoritmos PageRank e WCC em cada o grafo, e do SSSP nos grafos *wiki* e *lj*. Neste experimento, o *makespan* previsto pelo sistema foi de 17 minutos e 12 segundos. Já o *makespan* da execução real dos *jobs* foi de 15 minutos e 5 segundos. Ou seja, a diferença ficou em 11,3%, o que é um erro aceitável nestas circunstâncias.

6. Trabalhos Relacionados

A área de processamento distribuído de grafos é bastante ativa tanto na academia quanto na indústria, com diversos modelos sendo propostos nos últimos anos, como Pregel [Malewicz et al. 2010], GraphX [Xin et al. 2013] e PowerGraph [Gonzalez et al. 2012]. Pregel é um dos mais usados, com diversas implementações de código aberto, como Apache Giraph [Avery 2011], Apache Hama, [Seo et al. 2010] e GPS [Salihoglu and Widom 2013], que é usado neste trabalho. Existem diversos trabalhos na literatura sobre desempenho de modelos de processamento de grafos [Guo et al. 2014] e

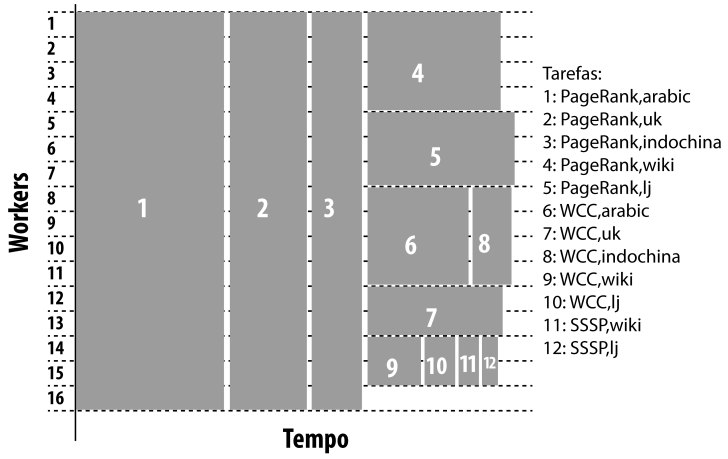


Figure 5. Um exemplo de escalonamento de *jobs* de processamento de grafos

[Han et al. 2014]. No entanto, estes trabalhos estão voltados para comparação de desempenho entre os diversos modelos. De maneira geral, o GPS apresenta um bom desempenho se comparado às outras implementações baseadas no Pregel, por isso foi escolhido como base neste trabalho.

Em se tratando de modelagem de desempenho, há apenas um trabalho na literatura [Li et al. 2017], onde é apresentado um modelo de predição semelhante ao descrito neste artigo. Entretanto, o modelo é bastante complexo, com 9 variáveis que precisam ser inicializadas manualmente por um especialista, o que limita seu uso na prática. Além disso, trata-se de um modelo para predição de desempenho voltado para estabelecer a melhor relação custo/benefício na escolha de instâncias de máquinas virtuais em ambiente de nuvem. O modelo leva em consideração, além do número de *workers*, limitações de banda de rede (de 20 MB/s a 100 MB/s), que são incomuns em ambiente de nuvem. O modelo apresentado aqui, por ser mais simples, pode ser inicializado automaticamente com os dados das execuções e produzir previsões quase tão precisas quanto as deste trabalho relacionado.

Já no caso de escalonamento de tarefas, existe uma vasta literatura tanto em algoritmos para escalonamento, quanto para aplicação em cenários de nuvem e computação distribuída de larga escala. Há diversos trabalhos que tratam de escalonamento em modelos de processamento distribuído de larga escala de propósito geral, como o Hadoop [Wolf et al. 2010, Kc and Anyanwu 2010], que é usado pelo Apache Giraph, uma implementação de código aberto do Pregel. Entretanto, estes modelos de escalonamento são específicos para as abstrações do Hadoop (ou seja, as funções *Map* e *Reduce*), não sendo adequados para uso com o Apache Giraph.

O trabalho [Dutot et al. 2005] trata de escalonamento para tarefas de uma aplicação BSP clássica, em que a variação do número de processadores alocados para uma tarefa é mais restrita. Por exemplo, uma tarefa que leva t para completar com n processadores levará aproximadamente $2t$ para completar com $n - 1$ processadores, já que um deles precisará processar 2 processos e os restantes precisarão ficar esperando. Embora proponham um algoritmo específico para o BSP, os autores notam que o algoritmo proposto em [Turek et al. 1992] é o que apresenta o melhor desempenho, o que motivou

seu uso neste trabalho.

Para modelos de processamento de grafos, é importante esclarecer que existem diversos trabalhos que tratam de escalonamento de *grafos de tarefas*, ou seja, conjuntos de tarefas cujas dependências são modeladas como um grafo. Estes modelos diferem substancialmente deste trabalho, que trata de processamento de *dados* modelados como grafos. Existem na literatura dois trabalhos que tratam especificamente de modelos de processamento de grafos. Canary [Qu et al. 2016] é um escalonador para computação de alto desempenho, tratando de casos em que o próprio escalonador pode se tornar um gargalo. Já Octopus [Padala et al. 2015] apresenta um escalonador para o caso de múltiplos usuários competindo por um cluster, usando um modelo de compartilhamento de tempo entre os usuários. Entretanto, ambos trabalhos tratam os *jobs* de processamento de grafos como tarefas rígidas, cuja alocação é predeterminada pelo usuário.

7. Conclusões e Trabalhos Futuros

Neste trabalho foi apresentada uma proposta de modelagem de desempenho para processamento distribuído de grafos, acompanhada de um escalonador para este tipo de tarefa. O modelo de previsão de desempenho é mais simples que o existente na literatura, o que permite sua inicialização automática, sem que haja prejuízo relevante na precisão dos resultados. O escalonador trata os *jobs* de processamento de grafos como tarefas moldáveis, alocando o melhor número de *workers* para cada tarefa de forma a minimizar o tempo total da computação.

Como trabalhos futuros está sendo considerada a modelagem de desempenho de novos algoritmos de análises em grafos, principalmente aqueles que modificam a topologia do grafo durante a computação. Além disso, planeja-se ampliar o escalonador incluindo outras funções objetivo, além de minimizar o *makespan*, como cumprimento de *deadlines* e acordos de nível de serviço (SLAs).

References

Avery, C. (2011). Giraph: Large-scale graph processing infrastructure on Hadoop. In *Proceedings of Hadoop Summit*.

Boldi, P. and Vigna, S. (2004). The WebGraph framework I: Compression techniques. In *Proc. of the 13th International World Wide Web Conference*, pages 595–601, Manhattan, USA.

Dutot, P.-F., Netto, M. A., Goldman, A., and Kon, F. (2005). Scheduling moldable BSP tasks. In *Proc. of the 11th International Workshops on Job Scheduling Strategies for Parallel Processing*, pages 157–172. Springer.

Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. (2012). PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. of the 10th Symposium on Operating System Design and Implementation*.

Guo, Y., Biczak, M., Varbanescu, A. L., Iosup, A., Martella, C., and Willke, T. L. (2014). How well do graph-processing platforms perform? An empirical performance evaluation and analysis. *Proc. of the International Parallel and Distributed Processing Symposium*, pages 395–404.

Han, M., Daudjee, K., Ammar, K., Özsu, M. T., Wang, X., and Jin, T. (2014). An experimental comparison of pregel-like graph processing systems. *Proceedings of the VLDB Endowment*.

Kc, K. and Anyanwu, K. (2010). Scheduling hadoop jobs to meet deadlines. In *Proc. of the 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*.

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

Leung, J. Y. (2004). *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press.

Li, Z., Zhang, B., Ren, S., Liu, Y., Qin, Z., Goh, R. S. M., and Gurusamy, M. (2017). Performance modelling and cost effective execution for distributed graph processing on configurable VMs. *Proc. of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 74–83.

Lumsdaine, A., Gregor, D., Hendrickson, B., and Berry, J. (2007). Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20.

Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In *Proc. of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 135–146.

Padala, S., Kumar, D., Raj, A., and Dharanipragada, J. (2015). Octopus: A multi-job scheduler for Graphlab. In *Proc. of the 2015 IEEE International Conference on Big Data*, pages 293–298.

Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab.

Qu, H., Mashayekhi, O., Terei, D., and Levis, P. (2016). Canary: A scheduling architecture for high performance cloud computing. *arXiv preprint arXiv:1602.01412*.

Salihoglu, S. and Widom, J. (2013). GPS: A graph processing system. In *Proc. of the 25th International Conference on Scientific and Statistical Database Management*.

Seo, S., Yoon, E. J., Kim, J., Jin, S., Kim, J.-S., and Maeng, S. (2010). Hama: An efficient matrix computation with the mapreduce framework. In *Proc. of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 721–726.

Turek, J., Wolf, J. L., and Yu, P. S. (1992). Approximate algorithms scheduling parallelizable tasks. In *Proceedings of the 4th ACM Symposium on Parallel Algorithms and Architectures*.

White, T. (2012). *Hadoop: The definitive guide*. O'Reilly Media, Inc.

Wolf, J., Rajan, D., Hildrum, K., Khandekar, R., Kumar, V., Parekh, S., Wu, K.-L., and balmin, A. (2010). Flex: A slot allocation scheduling optimizer for mapreduce workloads. In *Proc. of the 11th ACM/IFIP/USENIX International Conference on Middleware*.

Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I. (2013). GraphX: A resilient distributed graph system on spark. In *Proc. of the 1st International Workshop on Graph Data Management Experiences and Systems*.

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., et al. (2016). Apache spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65.