UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

CENEZ ARAÚJO DE REZENDE

UM ARCABOUÇO BASEADO EM COMPONENTES PARA COMPUTAÇÃO PARALELA DE LARGA ESCALA SOBRE GRAFOS

FORTALEZA

2017

CENEZ ARAÚJO DE REZENDE

UM ARCABOUÇO BASEADO EM COMPONENTES PARA COMPUTAÇÃO PARALELA DE LARGA ESCALA SOBRE GRAFOS

Tese apresentada ao Curso de Doutorado em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de doutor em Ciência da Computação. Área de Concentração: Ciência da Computação

Orientador: Prof. Dr. Francisco Heron de Carvalho Junior

FORTALEZA

2017

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

R356a Rezende, Cenez Araújo de.
Um Arcabouço Baseado em Componentes para Computação Paralela de Larga Escala sobre Grafos / Cenez Araújo de Rezende. – 2017.
176 f. : il. color.

Tese (doutorado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação , Fortaleza, 2017.
Orientação: Prof. Dr. Francisco Heron de Carvalho Junior.

1. Computação de Alto Desempenho. 2. Programação Paralela. 3. Engenharia de Software baseada em Componentes. 4. Grafos. I. Título.

CDD 005

CENEZ ARAÚJO DE REZENDE

UM ARCABOUÇO BASEADO EM COMPONENTES PARA COMPUTAÇÃO PARALELA DE LARGA ESCALA SOBRE GRAFOS

Tese apresentada ao Curso de Doutorado em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de doutor em Ciência da Computação. Área de Concentração: Ciência da Computação

Aprovada em: 29 de Agosto de 2017

BANCA EXAMINADORA

Prof. Dr. Francisco Heron de Carvalho Junior (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. André Rauber Du Bois
Universidade Federal de Pelotas (UFPEL)

Prof. Dr. Cláudia Linhares Sales
Universidade Federal do Ceará (UFC)

Prof. Dr. José Antonio Fernandes de Macêdo
Universidade Federal do Ceará (UFC)

Prof. Dr. Rafael Dueire Lins
Universidade Federal de Pernambuco (UFPE)

A Deus e à minha família.

AGRADECIMENTOS

Primeiramente, agradeço ao meu orientador, Francisco Heron de Carvalho Junior, como orientador e pessoa, pois esteve comigo guiando com excelência todos os meus passos relacionados às pesquisas, desde o mestrado até o doutorado. Agradeço aos membros da banca André Rauber Du Bois, Cláudia Linhares Sales, José Antonio Fernandes de Macêdo e Rafael Dueire Lins, pela disposição em ler, comentar, discutir, sugerir e criticar a tese, bem como novamente ao meu orientador por presidir a banca. À UFC (Universidade Federal do Ceará) e ao Departamento de Computação, pelo comprometimento em oferecer todos os recursos necessários para a formação de pesquisadores. Aos professores, pois tive a oportunidade de assistir aulas com conteúdo de grande qualidade, impulsionando meus estudos e pesquisas. Aos demais funcionários da UFC, pois foram fundamentais em diversas situações, tais como manutenção e coordenação das salas de aulas e seminários, redes de computadores e laboratórios, bem como atendimento no setor administrativo e secretaria. À CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior), deixo meus agradecimentos por contribuir e investir nas pesquisas desenvolvidas. Devo também lembrar que antes do início da minha jornada acadêmica tive ainda incentivo de vários pessoas, destacando Roberto Benedito de Oliveira Pereira, Sildenir Alves Ribeiro, Diógenes Antonio Marques José e Daniel Teixeira. Aos meus pais, Antonio Pereira de Rezende Sobrinho e Teresa Maria de Araújo Rezende, obrigado pelo incentivo e apoio, sempre acreditando que era possível. Ao meu irmão, Marcos Araújo de Rezende, pelas palavras de força em várias situações. Além disso, no ano de 2016, fui também presenteado com dois filhos gêmeos, Juan e Christian. Agradeço à mãe deles, Cerise Silveira, pelo presente, bem como pelo companheirismo, compreensão e incentivo desde o mestrado. Finalmente, agradeço a Deus por guiar minha vida e dar o encaminhamento necessário para este momento.

RESUMO

Diante do progressivo crescimento da produção de dados a serem processados por sistemas de computação, produto do contexto tecnológico vigente e de aplicações emergentes tanto de interesse industrial quanto científico, têm-se buscado soluções para alavancar a capacidade de processamento e análise de dados em larga escala. Além de volumosos, estão propícios a serem processados por algoritmos de alta complexidade, destacando as dificuldades inerentes a problemas em grafos grandes (*BigGraph*), frequentemente usados para modelar informações de grandes bases de dados. O modelo MapReduce, embora com limitações nesse domínio, abriu o caminho para a construção de vários arcabouços de alto desempenho, buscando atender à demanda por eficiente processamento de larga escala com propósito geral. Isso motivou o surgimento de soluções mais especializadas, voltadas a grafos, tais como os modelos Pregel e GAS (Gather, Apply, Scatter), bem como extensões do próprio MapReduce. Contudo, arcabouços que implementam esses modelos possuem ainda limitações, como restrições a multiplataformas e modelos mais gerais de programação. Neste trabalho, mostramos como a programação paralela orientada a componentes pode lidar com as limitações MapReduce e de modelos convencionais Pregel. Isso é feito usando a HPC Shelf, uma plataforma de computação em nuvem baseada em componentes para serviços HPC. Visando essa plataforma, apresentamos o Gust, um arcabouço *BigGraph flexível*, *extensível* e *adaptável* baseado em MapReduce. Através de estudo experimental, os resultados têm sido competitivos com o estado da arte, tanto em desempenho com na engenharia de software paralelo, com base em interesses funcionais e não funcionais.

Palavras-chave: Computação de Alto Desempenho. Programação Paralela. Engenharia de Software baseada em Componentes. Grafos.

ABSTRACT

Faced with the increasing growth of data production to be processed by computer systems, a result of the current technological context and emerging applications of both industrial and scientific interest, researchers and companies have been looking for solutions to leverage large-scale data processing and analysis capacity. In addition to the large volume, many of these data must be processed by high-complexity algorithms, highlighting the inherent difficulties of problems in large graphs (*BigGraph*), often used to model information from large databases. Although with limitations in graph processing, the MapReduce model has motivated the construction of several high-performance frameworks, in order to meet the demand for efficient large-scale general purpose systems. Such a context has led to the proposal of more specialized solutions, such as Pregel and GAS (Gather, Apply, Scatter), as well as MapReduce extensions to deal with graph processing. However, frameworks that implement these models still have limitations, such as multi-platform constraints and general propose programming models for graphs. In this work, we show how component-oriented parallel programming can deal with MapReduce and conventional Pregel constraints. For that, we have employed HPC shelf, a component-based cloud computing platform for HPC services. On top of this platform, we introduce Gust, a *flexible*, *extensible* and *adaptable BigGraph* framework based on MapReduce. Besides the gains in software architecture, due to the use of a component-oriented approach, we have obtained competitive performance results compared to the state-of-the-art through an experimental study, using estatistical methods to increase confidence.

Keywords: High performance computing. Component-based software engineering. Component-based high performance computing. Parallel programming. BigGraph.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Ao longo do caminho percorrido pela Ciência da Computação desde a metade do século passado, as áreas de investigação interessadas no desenvolvimento das técnicas de desenvolvimento de software têm se confrontado com desafios para otimizar o seu desempenho frente à rápida evolução do hardware. Tais desafios tem crescido substancialmente nas últimas duas décadas com o aumento da conectividade e do desempenho das redes de computadores, incluindo a própria Internet, bem como a popularização de arquiteturas de computadores com múltiplos processadores, processadores com múltiplos núcleos, dispositivos de aceleração computacional e hierarquias de memória de múltiplos níveis, trazendo novas oportunidades para otimizar o desempenho do software, encontrando apoio em técnicas de computação paralela, concorrente e distribuída antes aplicadas em nichos muito específicos, notadamente de computação científica. Dentro desse contexto moderno, a computação paralela heterogênea e distribuída em torno de recursos computacionais espalhados na internet tornou-se uma tendência em Computação de Alto Desempenho, usando as abstrações de *grades computacionais* e *computação em nuvens*.

Nesse contexto onde tornou-se viável o emprego de múltiplas infraestruturas computacionais interligadas para oferta de serviços e solução de problemas, possivelmente separadas geograficamente, tanto a academia quanto a indústria, motivados e fomentados tanto por interesses estatais quanto corporativos, tem desenvolvido novas aplicações, antes inviáveis. Tais aplicações emergentes têm feito crescer em escala exponencial ao longo dos anos a produção de dados e a demanda pelo seu processamento eficiente. Para compreensão da natureza e relevância dessas aplicações, bem como o nível de quantidade de dados que precisam processar, podemos citar vários exemplos, dentre as quais destacamos:

- aplicações multiusuário distribuídas, tais como redes sociais (Facebook, Twitter, Google Search Engine, etc), as quais precisam processar imensas quantidades de dados gerados por usuários a fim de capturar informações relevantes sobre seus interesses, incluindo os interesses de consumo;
- aplicativos de georreferenciamento, incluindo sistemas de navegação em ruas e rodovias que captam dados produzidos por usuários a fim de determinar melhores rotas frente às condições dinâmicas de tráfego;
- sistemas de vigilância, tanto contra acidentes naturais (terremotos, tsunamis, tornados, ciclones, etc) quanto contra atos de terrorismo ou que ameaçam a ordem pública (processamento de dados públicos produzidos por usuários de redes sociais, processamento de

imagens geradas por sistemas de videomonitoramento para identificação de veículos e reconhecimento facial, etc);

- aceleradores de partículas, como o *Large Hadron Collider*, cuja computação sobre a imensa quantidade de dados gerada teve a colaboração de vários países, através de uma grade computacional integrando *clusters* (geograficamente) distribuídos.

De fato, o processamento sobre massas de dados geradas e transmitidas através de redes geograficamente dispersas por essas e outras aplicações de mesma natureza podem exceder a capacidade de armazenamento e processamento dos computadores tradicionais, mesmo em se tratando de supercomputadores como os que se encontram classificados no ranque Top500 (<http://www.top500.org>).

O impacto do desafio do processamento de grandes massas de dados torna-se mais claro à medida em obervamos a necessidade de aplicar algoritmos complexos de computação para solução de problemas, tais como os que levam à aplicação de determinados algoritmos em grafos sobre bases de dados de escala significativa e utilizando arquiteturas de processamento paralelo de larga escala, tais como *Clique*(SVENDSEN, 2012), *Menor Caminho*(PLIMPTON; DEVINE, 2011) (*Single Source Shortest Path*), *Enumeração de triângulos*(COHEN, 2009), *Conjunto independente*(PLIMPTON; DEVINE, 2011), *Page Rank*(BAHMANI *et al.*, 2011), dentre outros. De fato, grafos são as estruturas de dados genéricas mais comumente utilizadas para modelagem de relações entre objetos em bases de dados de grande escala. Ferramentas que dão suporte a algoritmos de processamento de larga escala sobre grafos de grande porte geralmente são embasadas em algum modelo bem definido de processamento paralelo, o qual estabelece padrões e restrições pertinentes à comunicação e processamento, tais como o MapReduce e o Pregel, pontos de partida para a pesquisa desenvolvida nesta Tese de Doutorado. Dentro desse contexto, investigações que buscam técnicas e ferramentas para implementar algoritmos em grafos sobre plataformas e padrões de computação paralela específicos têm sido realizadas.

Em arcabouços (*frameworks*) de processamento paralelo de larga escala, encapsulam-se as partes complexas relativas aos interesses de paralelismo e distribuição de tarefas, incluindo o particionamento de dados. Por exemplo, um algoritmo segundo o modelo MapReduce tradicional terá duas etapas bem definidas: *distribuição* ou *mapeamento* (etapa relacionada a *Map*) e *redução* (etapa *Reduce*). Cada etapa recebe como entrada e oferece como saída pares chave/valor que representam os dados que serão processados por conjuntos de agentes mapeadores e redutores. Fundamentalmente, a responsabilidade do programador restringe-se geralmente a escrever as

funções (sequenciais) referentes a mapeamento e redução que serão executadas por esses agentes, delegando todas as responsabilidades inerentes ao paralelismo para o *framework* que implementa o modelo.

Embora o MapReduce tenha se popularizado, trata-se de um modelo não específico para grafos, contendo limitações que motivaram aprimoramentos(BU *et al.*, 2010; ZHANG *et al.*, 2012; QIN *et al.*, 2014) e novos modelos(MALEWICZ *et al.*, 2010; LOW *et al.*, 2010; ISARD *et al.*, 2007). Proposto para operar em lote, uma das limitações é a parte iterativa, especialmente exigida no processamento dos vértices de um grafo, pois eles podem necessitar ser lidos mais de uma vez. Além disso, a expressividade deve ser satisfeita em funções de *mapeamento* e *redução*, prejudicando o desenvolvimento de algoritmos projetados para três ou mais etapas de computação distintas, possivelmente sem necessidade de uma operação de mapeamento ou redução. Por exemplo, o algoritmo de triangulação MapReduce(SURI; VASSILVITSKII, 2011) possui duas etapas de *mapeamento* seguido de *redução*, onde a função de mapeamento, por ser uma identidade, torna desnecessária a etapa de mapeamento, podendo ser eliminada caso o *framework* permita que não seja considerada.

Uma solução alternativa tem sido disponibilizada através dos *frameworks* que implementam a especificação Pregel(MALEWICZ *et al.*, 2010). Diferente do MapReduce, Pregel é voltado para grafos, permitindo uma programação direcionada a esse tipo de estrutura de dados. Em sua especificação, Pregel apresenta um modelo de programação que consiste no desenvolvedor programar uma única função que utiliza dados de um vértice, o que é visto como uma etapa orientada a vértices. Esse tipo de sistema é inspirado no modelo *Bulk Synchronous Parallel* (BSP), o qual é organizado através da execução de uma computação seguida de um sincronismo, correlacionado com comunicação. Nesse sentido, Pregel é composto por uma função denominada *compute*, onde ocorre a etapa de processamento do vértice. Após isso, há o sincronismo. Nele, o destino de cada conjunto de dados computados por cada vértice é definido. Esses dados servem de entrada para uma próxima iteração, onde a função *compute* é novamente executada, gerando novos dados para sincronização.

A partir da especificação do MapReduce e Pregel, surgiram soluções como Spark (ZAHARIA *et al.*, 2010), GraphLab(LOW *et al.*, 2010), MR-MPI(PLIMPTON; DEVINE, 2011), dentre outras. Essas soluções emergentes buscam oferecer alto nível de abstração na programação de aplicações paralelas e distribuídas, o que minimiza o trabalho de desenvolvimento desse tipo de software, pois a implementação da maior parte dos interesses relativos ao paralelismo não fica

sob a responsabilidade do programador, que pode concentrar maior atenção à lógica intrínseca de sua aplicação.

Os benefícios de interfaces com alto nível de abstração em relação aos interesses de paralelismo devem ser ponderados, pois isso também pode conduzir a restrições de expressividade, especialmente quando se olha o legado de aplicações com requisitos de CAD (Computação de Alto Desempenho). Por exemplo, as interfaces de programação que têm conseguido ser expressivas, bem como portáveis, assim como explorando com sucesso o potencial de desempenho de plataformas de computação paralela, possuem baixo nível de abstração, como é o caso das bibliotecas que implementam passagem de mensagens MPI (*Message Passing Interface*). Uma utilização recente dessa biblioteca em MapReduce ocorre no *framework* MR-MPI, que tem operado em alto nível encapsulando a expressividade do MPI, que está na sua base de desenvolvimento.

De fato, há um significativo legado de aplicações CAD que usam interfaces de baixo nível de abstração, pois obtiveram sucesso com linguagens como C e Fortran por se tratarem de projetos de software de pequena escala. Entretanto, novos cenários que exigem o desenvolvimento de larga escala em aplicações típicas de CAD, devido à fatores como a complexidade inerente do software emergente, dado o novo patamar tecnológico descrito no início desse capítulo, bem como a necessidade de reuso de suas partes e desenvolvimento por diferentes equipes, de forma interdisciplinar, tem exigido a aplicação de técnicas avançadas de engenharia de software, tais como a fatoração em componentes (CARVALHO JUNIOR; REZENDE, 2013), que permite ao programador definir e expressar o software utilizando abstrações de mais alto nível, baseadas na noção de componente de software.

Nos *frameworks* para processamento de grafos em larga escala, isso também é uma realidade, pois buscam oferecer alto nível de abstração em relação a mecanismos de paralelismo e tolerância a falhas, oferecendo serviços automatizados que realizam suposições sobre a arquitetura a fim de efetuar distribuição de dados e paralelização do processamento das computações. Isso é diferente da metodologia tradicional empregada na maioria das aplicações legadas e atuais(CHAVARRÍA-MIRANDA *et al.*, 2015; BAILEY, 1991), onde a preocupação com a arquitetura e os detalhes da paralelização é totalmente delegada ao desenvolvedor, conhecedor de técnicas particulares de como explorar de forma eficiente as características de plataformas de computação paralela, eventualmente heterogêneas e distribuídas.

Na prática, certos aspectos da metodologia tradicional ainda são necessários. Por

exemplo, busca-se extrair o melhor das plataformas de execução através da otimização de alguém que conheça o ambiente alvo. Por exemplo, como observado por *Zaharia et al*(ZAHARIA *et al.*, 2008), quando perceberam que os algoritmos de escalonamento do Hadoop eram ineficientes para máquinas heterogêneas e propuseram um novo escalonador de tarefas para esse sistema. Diante disso, vários *frameworks* (por exemplo, implementações de MapReduce) permitem ao desenvolvedor realizar a customização de certas atividades, como o particionamento de dados. Atividades como partição de dados podem também ser de interesse do código que representa o algoritmo, que potencialmente processa vértices, blocos ou subgrafos. Por exemplo, interesse pela proximidade das unidades que processam determinados conjuntos de vértices, para melhora da comunicação, o que reflete num relacionamento do elemento particionador com o contexto do algoritmo. Visando esse tipo de situação, onde se relacionam requisitos de alta expressividade, alto nível de abstração, consideração de detalhes arquiteturais e outros detalhes do contexto de ambiente de execução do software, a programação baseada em componentes tem sido objeto de estudos deste trabalho, especialmente através do modelo de componentes Hash e do projeto de nuvem computacional HPC Shelf, propostos pelo grupo de pesquisa do qual faz parte o autor desta Tese.

1.1 O Modelo Hash

Voltado à construção de componentes paralelos, que se propõem a serem utilizados no desenvolvimento de aplicações com requisitos de CAD, o modelo de componentes Hash apresenta uma forma expressiva para tratar requisitos de paralelismo em arcabouços de construção de software baseado em componentes, permitindo a decomposição do software com base nos interesses de software, como convém à arquitetura de software baseada em componentes, ao invés de processos, como é feito tradicionalmente com ferramentas de programação paralela (CARVALHO JUNIOR; LINS, 2005; CARVALHO JUNIOR *et al.*, 2007; CARVALHO JUNIOR; LINS, 2008).

A partir da perspectiva do modelo Hash, interesses de software são ortogonais aos processos, de modo que sua implementação encontra-se espalhada em um conjunto deles. Dessa forma, cada processo que forma um programa paralelo representa um agente cuja implementação pode ser fatiada em várias *unidades*, cada uma das quais com um papel específico na implementação de um interesse tratado por determinado componente que forma o programa. Assim, componentes do modelo Hash podem ser combinados entre si hierarquicamente para

formar outros componentes e programas, encapsulando quaisquer interesses de paralelismo, funcionais e não-funcionais, que poderiam ser programados e expressos através de bibliotecas de passagem de mensagens, bem como outras interfaces de programação paralelas gerais voltadas a arquiteturas distribuídas de computação paralela.

Outra característica de plataformas de componentes que aderem ao modelo Hash é a divisão dos componentes em espécies, onde cada espécie de componente pode representar abstrações particulares para blocos de construção de aplicações dentro de um determinado domínio de interesse ou propósito geral.

A implementação de referência do modelo Hash é a plataforma de componentes HPE (*Hash Programming Environment*), voltado a construção orientada a componentes de programas paralelos para plataformas de *cluster computing* (CARVALHO JUNIOR *et al.*, 2007; CARVALHO JUNIOR; REZENDE, 2013). Suas espécies de componentes são voltadas ao desenvolvimento de programas paralelos para propósitos gerais.

1.2 A Plataforma HPC Shelf

A plataforma HPC Shelf é produto de um projeto voltado à construção de uma plataforma de serviços de desenvolvimento, implantação e execução de aplicações com requisitos de CAD sob a abstração de nuvens computacionais e utilizando tecnologia de componentes paralelos do modelo Hash. Dessa forma, componentes são desenvolvidos a partir da composição de outros componentes que podem representar elementos tanto de hardware quanto de software, bem como expressando interesses não-funcionais, formando sistemas de computação paralela. Isso implica em componentes que tratam de interesses tais como características de plataformas de computação paralela, algoritmos, estruturas de dados e padrões de comunicação entre processos.

Na plataforma HPC Shelf, atuam quatro tipos de atores. Usuários *especialistas* são aqueles que demandam por soluções computacionais para problemas dentro de um certo domínio de interesse, os quais possuem requisitos inerentes de computação de alto desempenho. Tais usuários não possuem conhecimento sobre a natureza da infraestutura computacional da HPC Shelf, tampouco sobre a abstração de componentes sobre a qual esssas soluções computacionais são implementadas. Por sua vez, *provedores de aplicação* são aqueles reponsáveis pela construção dessas soluções computacionais, por intermédio de aplicações que disponbilizam a usuários especialistas. São portanto especialistas em técnicas computacionais aplicadas à solução de problemas dentro de um certo domínio de interesse, sendo capazes de discernir os

componentes necessários para compôr tais soluções sobre a HPC Shelf. Esses componentes são desenvolvidos pelos *desenvolvedores de componentes*, os quais possuem conhecimento de técnicas de programação paralela e arquiteturas de computação paralela, sendo responsáveis pelo desenvolvimento de componentes especializados para explorar as particularidades arquiteturais dessas plataformas. Finalmente, os *mantenedores de plataformas* são responsáveis pelo gerenciamento de componentes que representam plataformas (virtuais) de computação paralela, ou seja, são responsáveis, em seu conjunto, por oferecer a infraestrutura computacional da nuvem.

Na forma tradicional, para conseguir explorar satisfatoriamente o desempenho de uma plataforma de computação paralela, o desenvolvedor necessita possuir experiência com técnicas de paralelismo e ser conhecedor de detalhes da plataforma sobre a qual deseja executar os programas que desenvolve. Na HPC Shelf, as aplicações usufruem de plataformas virtuais, representadas por componentes oferecidos pelos *mantenedores*. Na construção de sistemas de computação paralela para resolver problemas especificados pelo especialista através de sua interface de alto nível de abstração, uma aplicação declara o interesse em um conjunto de componentes, dentre os quais aqueles que representam plataformas virtuais que oferecem o serviço de execução aos componentes que representam algoritmos e repositórios de dados a serem processados. Através de um sistema de *contratos contextuais* derivado do sistema de tipos de componentes do HPE (CARVALHO JUNIOR *et al.*, 2016), a aplicação é capaz de especificar requisitos que restringem quais as plataformas virtuais que podem ser empregadas pela nuvem para serem instanciadas na execução e receber componentes de software que executarão sobre ela, observando especialmente critérios de qualidade e de custo dos serviços de execução.

Neste trabalho de pesquisa, usam-se as funcionalidades e características da plataforma HPC Shelf de adaptação de componentes conforme a plataforma de execução para lidar com desafios na construção de plataformas de processamento paralelo de larga escala sobre grafos grandes. O produto desenvolvido é o Gust, uma aplicação através da qual usuários podem definir computações potencialmente complexas sobre grafos a partir de uma interface e modelo de programação de alto nível, de modo que computações são vistas como sistemas de computação paralela constituídos de componentes para essa finalidade. Sobre esse arcabouço, estão as principais contribuições desta Tese de Doutorado, suportando diferentes modelos de programação (centrado em vértice ou subgrafo) e adaptação às características de modelos de computação paralela, particularmente MapReduce e Pregel, permitindo customizações, através do sistema de contratos contextuais, que permitem usufruir de variações dos modelos de computação originais

implementados por diversos arcabouços baseados nesses modelos que tem sido propostos nos últimos anos, de maneira potencialmente extensível.

1.3 Objetivos

Com base nas motivações e problemas discutidos nos parágrafos anteriores, definem-se a seguir, de forma estruturada, os objetivos desta Tese de Doutorado.

1.3.1 Objetivo Geral

Utilizando uma abordagem orientada a componentes, construir um arcabouço para composição de sistemas de computação paralela destinados ao processamento paralelo de larga escala de grafos grandes que seja *flexível*, *extensível* e *adaptativo*.

- Por *flexível*, entende-se a capacidade de se adaptar a diferentes variações propostas por modelos alternativos de processamento existentes para a mesma finalidade, tais como MapReduce e Pregel.
- Por *extensível*, entende-se capacidade de incorporar a expressividade de novos modelos sem a necessidade de alterar o modelo geral proposto pelo arcabouço.
- Por *adaptativo*, entende-se a capacidade de escolha alternativa de componentes de acordo com as características arquiteturais das plataformas de computação paralela empregadas, visando utilizar o máximo de seu desempenho potencial.

1.3.2 Objetivos Específicos

1. Oferecer uma análise crítica sobre as características dos arcabouços existentes para processamento paralelo de larga escala de grafos grandes, com base na literatura, buscando identificar suas limitações expressivas e de desempenho;
2. Mapear as principais funcionalidades que diferenciam fundamentalmente as diversas alternativas de *frameworks* baseados em MapReduce e Pregel;
3. Propor um arcabouço orientado a componentes para processamento paralelo de larga escala baseado no modelo MapReduce;
4. Validar as características e funcionalidades da plataforma HPC Shelf através de estudos de caso envolvendo os arcabouços propostos neste trabalho;
5. Propor e validar técnicas baseadas em componentes para descrição de aplicações e compu-

tações envolvendo processamento paralelo de larga escala;

6. Demonstrar a expressividade de *contratos contextuais* para abstração de componentes em relação a propriedades alternativas de sua implementação, permitindo variar a escolha de implementações de um componente abstrato em uma aplicação pela simples variação da escolha de argumentos de contexto.

1.4 Contribuição ao Projeto HPC Shelf: MapReduce e Gust

O arcabouço MapReduce proposto para a HPC Shelf como uma das contribuições desta Tese é a primeira aplicação que executou com sucesso sistemas de computação paralela sobre essa plataforma, permitindo validar a expressividade do seu conjunto de espécies de componentes, o uso de múltiplas plataformas virtuais em um mesmo sistema e a orquestração de *workflows* usando o SAFe, o arcabouço de aplicações da HPC Shelf (SILVA; CARVALHO JUNIOR, 2016), dentre outras coisas.

Por sua vez, o arcabouço Gust introduziu a HPC Shelf dentro do contexto de processamento de grafos grandes, com uma abordagem naturalmente de larga escala (múltiplos *clusters*). Permitiu ainda avaliar características de extensibilidade de arcabouços de componentes sobre a plataforma, uma vez que foi definido por extensão do arcabouço MapReduce. O Gust ofereceu a oportunidade para a construção de sistemas de computação paralela mais complexos, para implementar algoritmos de grafos conhecidos, oferecendo a oportunidade para avaliação de desempenho da plataforma HPC Shelf.

1.5 Contribuições dos Resultados da Tese

Em relação aos tradicionais sistemas MapReduce e Pregel, os arcabouços MapReduce e Gust propostos por esta Tese envolvem técnicas de desenvolvimento orientado a componentes de software de larga-escala, onde esses componentes encapsulam os interesses de execução paralela *intra-cluster*. Isso permite a construção de sistemas naturalmente modulares, adaptáveis e extensíveis. Para isso, cada componente de software é primeiramente modelado de forma objetiva para atender à proposta da HPC Shelf, viabilizando um modelo arquitetural útil para a execução de workflows, tais como os suportados pelo SAFe. Como resultado, na sua implementação, Gust também atende aos requisitos para execução em múltiplos clusters ou plataformas virtuais da HPC Shelf (execução paralela *entre-clusters*), definidas via contratos contextuais, o que atualmente a

literatura não aborda para o processamento de grafos em larga escala. Para isso, plataformas de computação paralela também são representadas como componentes e suas características fazem parte da assinatura contextual dos componentes computacionais empregados no processamento de grafos. As contribuições do Gust dentro do contexto de processamento de grafos grandes são destacados em seguida:

- **assincronismo**: MapReduce e Gust suportam mensagens assíncronas de forma natural, permitindo sobreposição entre etapas de processamento dos algoritmos (e.g. mapeamento e redução). Para isso, usando um *iterador* de conjuntos de pares chave/valor (*chunks*), dados são enviados e recebidos sob demanda, sendo um importante recurso disponível na construção dos sistemas de computação paralela que exploram o paralelismo *pipeline*.
- **modelo de programação**: através do componente Graph e do contexto chave/valor suportado pelo arcabouço MapReduce, onde os tipos de dados são genéricos, Gust permite que a programação seja centrada em *vértices* (*vertex-centric*) ou *blocos de vértices* (*block-centric*). Esses são modelos já utilizados por *frameworks* Pregel. Contudo, de forma semelhante ao Modelo *GAS*(GONZALEZ *et al.*, 2012) (*Gather, Apply, Scatter*), que é centrado em vértices, Gust utiliza as definições dos métodos *unroll*, *compute* e *scatter* para descrição da programação voltada a *vértices* ou *blocos de vértices*;
- modos empurrar/puxar(BESTA *et al.*, 2017) (*push/pull*): no processamento de grafos usando o modelo de programação centrado em vértice (*vertex-centric*), onde o desenvolvedor foca no estado de um vértice em processamento para aplicar sua lógica de programação, o modo *empurrar* consiste em o vértice processar um valor e oferece-lo aos outros estados de vértices, tipicamente vizinhos. Por outro lado, o modo *puxar* consiste em adquirir valores (de outro estado) para um estado particular de vértice. Ambas essas técnicas são possíveis de serem utilizadas em sistemas Gust, desde que o desenvolvedor insira no iterador de saída os valores referentes a vértices processados nos super-passos (*Supersteps*). Nos estudos de caso apresentado nesta Tese, um dos *benchmarks* utilizados que explora explicitamente a possibilidade de usar os dois modos é o de ranqueamento de páginas (*PageRank*), onde *puxar* e *empurrar* são feitos respectivamente pelos métodos *getCurrentValue* e *setValue*;
- **desempenho**: foi realizada uma avaliação de desempenho comparativa entre o Gust e outras alternativas dentro do estado-da-arte: GraphX e Giraph. Para isso, foram usados três *benchmarks*: ranqueamento de páginas (*PageRank*), enumeração de triângulos e caminho

mínimo (*SSSP*), que são frequentemente utilizados para analisar desempenho de sistemas de processamento de grafos grandes. Para isso, as medidas utilizadas foram tempo gasto no processamento de uma tarefa e consumo de memória, ambas apresentando resultados promissores em relação ao Gust. Além disso, por ser otimizado para economizar memória, Gust conseguiu processar significativas cargas de grafos em máquinas virtuais com poucos recursos.

1.6 Estrutura do trabalho

A primeira parte desta Tese, referente ao Capítulo 2, apresenta uma revisão sobre o modelo de programação MapReduce, bem como descreve algumas de suas implementações mais conhecidas, tais como o *framework* de referência do Google e o Hadoop. No Capítulo 3, apresentam-se técnicas e ferramentas emergentes para computação paralela de larga escala sobre grafos, onde é analisado o estado-da-arte dentro desse contexto, buscando expôr as qualidades e limitações das alternativas existentes. A importância desses capítulos é oferecer subsídios para definição dos arcabouços MapReduce e Gust, as abordagens alternativas propostas para cumprir o objetivo principal desta Tese. No Capítulo 4, apresenta-se a plataforma HPC Shelf, que serviu de base para implementação das contribuições desta Tese. No Capítulo 5, são apresentados os arcabouços MapReduce e Gust, os quais são validados no Capítulo 6, através um estudo experimental de avaliação de desempenho. Finalmente, as conclusões e perspectivas de trabalhos futuros são discutidas no Capítulo 7.

2 MAPREDUCE

A busca por melhores práticas para o tratamento de dados tem sido, nas últimas décadas, uma prioridade para a indústria e academia, que tentam acompanhar as novas formas de interação e paralelismo do *software* em computadores contemporâneos, bem como constantes inovações no âmbito do *hardware*, tais como novas tecnologias em processadores, incluindo aceleradores computacionais, dentre os quais se destacam GPUs (*Graphic Processing Units*) (FAN *et al.*, 2004), MICs (*Many Integrated Cores*) (DURAN; KLEMM, 2012) e FPGAs (*Field-Programmable Gate Array*) (HERBORDT *et al.*, 2007).

Com o advento das redes e da *Internet* como plataformas do software, as barreiras geográficas para conexão de dispositivos computacionais foram reduzidas, o que permite uma crescente proliferação de dados que necessitam ser analisados e computados de forma eficiente. Informações são geradas por sistemas geograficamente espalhados. Não apenas em computadores e servidores *web*, mas também em dispositivos móveis e sensores em geral, tais como sistemas de posicionamento (e.g. *GPS*). Isso motivou o surgimento do conceito de *Big Data* (ORACLE, 2013), diretamente relacionado ao processamento de grandes volumes de dados e metodologias eficientes para analisá-los e transformá-los em informações úteis. Em se tratando de volume significativo, uma opção é a divisão para paralelização, distribuindo-se a carga de processamento entre unidades de processamento distribuídas que formam sistemas de computação paralela de larga escala. Visando encapsular esse paralelismo de larga escala, emergiu o paradigma de processamento paralelo que ficou conhecido como MapReduce, e que tornou-se popular industrialmente (Figura 1) a partir das publicações de *Dean* e *Sanjay*, pesquisadores do Google (DEAN; GHEMAWAT, 2004; DEAN; GHEMAWAT, 2008). Neste Capítulo, abordam-se os principais conceitos relativos ao modelo de processamento paralelo MapReduce, bem como algumas de suas implementações mais disseminadas, tais como o *framework* de referência do Google e o Hadoop, implementação de referência de código aberto.

2.1 Modelo de Programação MapReduce

A partir de meados do ano 2000, empresas como Google, Yahoo, Facebook e Amazon destacaram o uso de MapReduce como uma forma de efetuar computação paralela eficiente com dados em volumes na ordem de *terabytes* ou *petabytes*. A característica desse modelo tem permitido dividir computações e dados entre máquinas, bem como encapsular operações

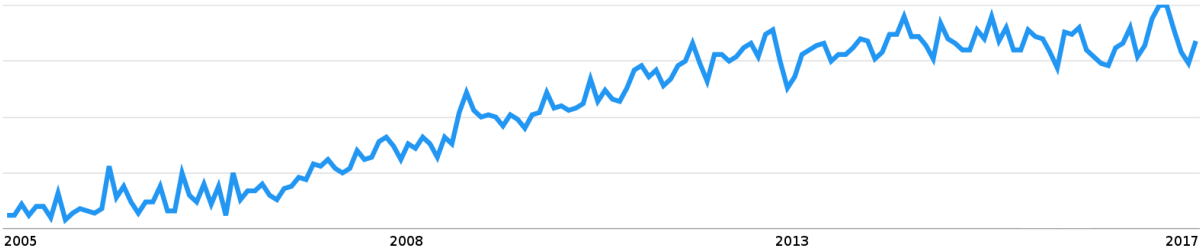


Figura 1 – Crescimento das buscas *Map Reduce* no *Google Trends* (GOOGLE, 2006).

complexas de paralelismo. Nesta seção, discutimos os principais detalhes desse modelo de programação, iniciando com seu histórico e fundamentos, para posteriormente discutir as funções *map* e *reduce*, e apresentar exemplos de algoritmos.

2.1.1 Histórico e Fundamentos do Modelo

O MapReduce é um modelo de programação patenteado (sob número *7650331* (GOVERNO USA, 1975)) em nome de *Dean* e *Sanjay*, com direito de exploração do Google. Por esse motivo, eventualmente ocorrem discussões contrárias a esse direito (DBMS2, 2010), afirmando ser um modelo semelhante a alguns produtos já existentes, ou que o modelo pode ser naturalmente implementado em linguagens de manipulação de dados, como *SQL*. Argumenta-se ainda que a popularização do MapReduce promovida pelo Google não poderia ser entendida como invento. Diante disso, o Google tem recentemente demonstrado não ter intenção de barrar a evolução de tecnologias (GOOGLE, 2013), uma vez que grande parte delas na computação se fez através de software livre, em detrimento da posse de soluções. O primeiro sinal do Google não coibindo a utilização do MapReduce foi na concessão de licença para o projeto Hadoop, da *Apache Software Foundation* (APACHE, 2005). Além disso, a empresa divulgou uma lista (GOOGLE, 2013) de patentes, que inclui MapReduce, liberadas de direitos, desde que para aplicação em iniciativas com software livre.

O modelo MapReduce tem origem essencialmente inspirada por linguagens funcionais, como afirmam os próprios autores (DEAN; GHEMAWAT, 2008). De fato, a inspiração pode ser mais bem entendida através de funções disponíveis em linguagens funcionais como *Lisp* ou Haskell. Uma característica notável dessas linguagens são as funções de alta ordem, o que significa suporte para que uma função aceite como argumento ou retorne como resultado outra função. Isso é especialmente útil se houver utilização de polimorfismo nessas funções, o qual é suportado nessas linguagens.

```
1. foldr1 :: (t→t→t)→[t]→t
2. foldr1 f [a] = a
3. foldr1 f (a : b : z) = f a (foldr1 f (b : z))
4. map :: (t→u)→[t]→[u]
5. map f [] = []
6. map f list = [f a | a ←list]
7. size :: [t]→Int
8. size [] = 0
9. size (a : z) = 1 + size z
10. sum :: Int→Int→Int
11. sum a b = a + b
12. r :: Int
13. r = (foldr1 (sum) (map size [“a”,“ab”,“abc”]))
```

Figura 2 – Fundamentos *MapReduce* em linguagem funcional (Haskell)

Adaptado de *Du Bois* (DU BOIS, n.d.)

Exemplos de funções de alta ordem úteis para entender o processamento MapReduce são `map` e `foldr1`, comuns em linguagens funcionais de alta ordem, como Haskell. Por exemplo, considere `size` e `sum` duas funções que respectivamente retornam o tamanho de uma lista e a soma de dois números. Uma amostra da utilização dessas funções com `map` e `foldr1` pode ser visto na Figura 2. Lembramos que, em Haskell, uma lista vazia é representada por `[]`, enquanto uma lista não vazia, composta de n elementos, é representada por $[v_1, v_2, \ldots, v_n]$, onde v_i, para $i \in \{1, 2, \ldots, n\}$, representa um valor de um certo tipo de dado da lista.

A implementação da função `map` possui dois parâmetros: uma lista de valores (*list*) e uma função que será aplicada a cada elemento da lista (f). A função `map` obtém como retorno outra lista cujos elementos são os resultados da aplicação da função f para cada elemento da lista *list*. Portanto, no exemplo, o resultado da aplicação de `map` é a lista `[1,2,3]`, ou seja, os tamanhos respectivos das cadeias de caracteres “a”,“ab”,“abc”, fornecidas na lista passada como argumento através do parâmetro *list*. O retorno da aplicação de `map` é repassado como argumento para uma aplicação de `foldr1`, na linha 13. Dessa forma, `foldr1` tem `sum` e `[1,2,3]` como parâmetros, retornando a soma dos valores da lista `[1,2,3]`, ou seja, o valor `6`, na variável `r`. A função `map`, análoga a etapa de *mapeamento* do processamento MapReduce, pode ser entendida como uma forma de transformação de dados baseada na função fornecida como parâmetro, enquanto que a função `foldr1`, análoga a etapa de *redução* do processamento MapReduce, pode ser entendida como uma função de agregação baseada na função fornecida como argumento.

2.1.2 Mapeamento e Redução

O objetivo do MapReduce é estabelecer uma forma genérica de se programar um sistema de processamento paralelo, composto por duas etapas bem distintas, análogas às funções *map* e *fold* das linguagens funcionais (LIN; DYER, 2010).

Na etapa de *mapeamento*, o programador deve definir a função específica que será aplicada a todos os registros de entrada de dados, o que ocorre de forma paralela, no âmbito de memória distribuída, sobre subconjuntos de dados. Na etapa de *redução*, o programador detalha a função para agregação de valores oriundos da etapa anterior. As tarefas complexas de paralelização, distribuição e segurança do sistema, incluindo a comunicação, disponibilidade, tolerância a falhas, concorrência, dependência, escalonamento e monitoramento são abstraídas do desenvolvedor, ficando sob a responsabilidade do *framework*, ou arcabouço, MapReduce.

Existe uma etapa de tratamento intermediário dos dados produzidos na etapa de mapeamento e que serão usados na etapa de redução. Essa fase é denominada em algumas implementações (exemplo Hadoop) como *embaralhamento* (do inglês, *shuffle*), sendo controlada pelo arcabouço MapReduce. Em ambiente paralelo, o *embaralhamento* deve agrupar os dados produzidos pelas instâncias paralelas da etapa de *mapeamento*. Nesse agrupamento, que é feito com dados de mesma *chave* (ver próxima Seção 2.1.3), há uma pré-ordenação (vinculando nós *trabalhadores* a determinados dados) e posterior troca de dados entre os nós *trabalhadores*, responsáveis por executar as funções de *mapeamento* e *redução*.

2.1.3 Assinatura Chave/Valor e Fase de Embaralhamento (Shuffle)

A expressividade do modelo MapReduce considera o uso de dados na forma *chave*/*valor*, juntamente com a descrição das etapas de *mapeamento* e *redução*. A entrada do *mapeamento* contém pares *chave*/*valor*, que são processados gerando como saída pares *chave*/*valor*, denominados pares intermediários, de modo que uma chave intermediária será associada a um ou mais valores intermediários. Essa associação é feita localmente na máquina que faz o *mapeamento*. Posteriormente, os dados intermediários de mesma *chave* são paralelamente reagrupados através da etapa de *embaralhamento*, e servem de entrada para a etapa posterior, a *redução*. Portanto, ocorrem trocas de mensagens, entre processos paralelos que fazem o *mapeamento* e que fazem a *redução*. De posse do agrupamento, cada instância de operação de *redução* finalmente processa e produz seus resultados, também na forma de pares *chave*/*valor*. Até aqui, os pares de entrada e

saída do *mapeamento* e *redução* podem ser compostos por tipos diversos. Por exemplo, caracteres, cadeias de caracteres (*strings*), números inteiros, números de ponto flutuante, *bytes*; ou ainda tipos mais complexos, tais como *listas*, *vetores*, *matrizes*, dentre outros. Essa definição de tipagem obedece aos requisitos de entrada e saída do próprio algoritmo desenvolvido para MapReduce.

$map[k_1, v_1] \ \rightarrow$ processar $\rightarrow list[k_2, v_2]$
$reduce[k_2, \ list(v_2)] \ \rightarrow$ processar $\rightarrow list(v_3)$

Tabela 1 – Chave/valor em MapReduce (DEAN; GHEMAWAT, 2004)

As assinaturas para o modelo são definidas pela Tabela 1. Nota-se que o par *chave/valor* de entrada do mapeamento (*[k_1,v_1]*) é processado, produzindo como saída uma lista de pares intermediários, representados por *list[k_2,v_2]*. Posteriormente é feita a etapa de embaralhamento, que agrupa paralelamente os dados intermediários pela mesma *chave*, produzindo um conjunto de registros, representados por *[k_2, $list(v_2)$]*. Mais adiante, a *redução* irá processar seu par *chave/valor* (*[k_2, $list(v_2)$]*), de modo que o valor $list(v_2)$ contém um ou mais valores intermediários para uma mesma chave k_2. Como resultado, a redução gera $list(v_3)$, que é usualmente apenas um valor calculado a partir da *chave* k_2.

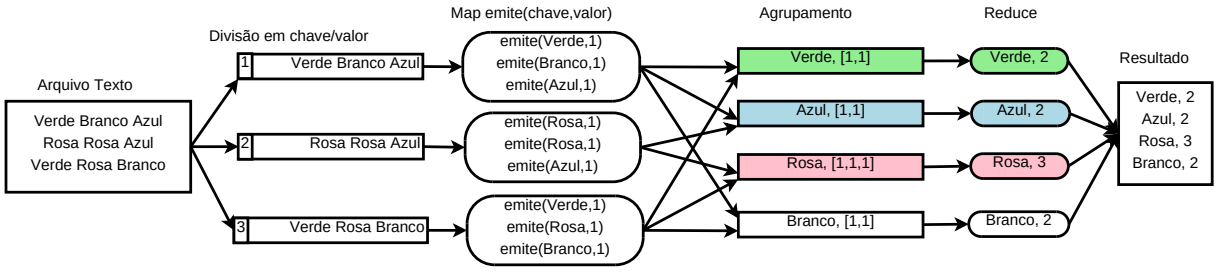


Figura 3 – Exemplo de fluxo de dados com MapReduce. Consiste na contagem de ocorrências de palavras em um arquivo texto.

2.1.4 Exemplo de Algoritmo MapReduce - Contador de Palavras

Um dos mais conhecidos exemplos de algoritmos para MapReduce é o contador de palavras repetidas dentro de um texto, onde o texto pode estar associado com arquivos, banco de dados ou *URLs* da *Internet*. Esse exemplo está disponível no Algoritmo 1. Nele, a função de *mapeamento* toma como parâmetros um identificador do texto de entrada (representado pelo inteiro *i*) e o texto (representado pela variável `texto`). A etapa de *mapeamento* faz a leitura de

cada palavra no texto e emite o registro `conta(‘‘1’’)` para cada ocorrência. Ou seja, o algoritmo não se preocupa com palavras repetidas, simplesmente as rotula com o valor 1. Através desse *mapeamento*, são geradas listas de valores 1 na variável `contagem`, que representa uma lista de ocorrências das palavras. Dessa forma, sob a representação *chave*/*valor*, cada palavra em `palavra_p` agora é *chave*, e a lista de valores 1 é repassada à etapa seguinte, de *redução*, através das variáveis `palavra_p` e `contagem`. Uma aplicação da função de *redução* soma cada 1 em `Lista contagem` e então emite o resultado associado àquela palavra, de origem em `palavra_p`.

Algoritmo 1: Contador de Palavras.

```
1 Map
    Entrada: ChaveValor<Inteiro i, Texto texto>
    para todo palavra_p ∈ texto hacer
      | Emitir(palavra_p, contar("1"))
    fin
2 Reduce
    Entrada: ChaveValor<String palavra_p, Lista contagem>
    int soma = 0
    para todo valor v ∈ contagem hacer
      | soma = soma + toInt(v)
    fin
    Emitir(palavra_p, toString(soma))
```

(Adaptado de: *Dean* (DEAN; GHEMAWAT, 2008))

Esse algoritmo pode também ser observado na Figura 3. Percebe-se que há um texto que é transformado em pares *chave*/*valor* através da divisão em linhas: 1 (“verde branco azul”), 2 (“rosa rosa azul”) e 3 (“verde rosa branco”). Em paralelo, cada etapa de *mapeamento* processa sua linha, emitindo a cor como *chave* e o inteiro 1 como *valor*. No *agrupamento*, o conjunto de *1’s* é relacionado com uma cor, que é a *chave*. Após o *agrupamento*, a etapa de *redução* realiza a soma dos *1’s*, que consiste na quantidade de ocorrências de palavras. No caso deste exemplo, cada palavra é uma cor (verde, azul, rosa e branco). Como resultado, todas essas cores ocorrem duas vezes, com exceção da cor rosa, que ocorre três vezes.

2.1.5 Frameworks MapReduce

Nas próximas seções, discute-se o estado da arte de *frameworks* MapReduce. Para isso, inicialmente, aborda-se o *framework* de referência do Google, objetivando caracterizar o ponto de partida para um conjunto de outros *frameworks* que o sucederam, como o Hadoop (APACHE, 2005), primeira *plataforma* de código livre.

2.2 O *Framework* Referência do Google

A especificação MapReduce do Google busca fornecer uma metodologia capaz de encapsular e minimizar a complexidade da implementação de interesses de computação paralela, tarefa normalmente recorrente para o desenvolvedor de aplicações com requisitos de Computação de Alto Desempenho. Dessa forma, o programador se concentra diretamente na lógica de negócio da sua aplicação em desenvolvimento, e não em detalhes a cerca do seu ambiente de execução. A estrutura do *framework* MapReduce do Google considera um conjunto de N máquinas, com $N-1$ máquinas *trabalhadoras* e uma única máquina *gerente*. Portanto, trata-se de uma instância do modelo de comunicação *mestre/escravo* (*master/slave*), voltado a *clusters*.

O ambiente de execução faz uso de um sistema de arquivos distribuído chamado GFS (*Google File System*) (GHEMAWAT *et al.*, 2003), que oferece suporte ao armazenamento de dados que serão processados no *cluster*. Esse sistema de arquivos também se organiza através da ideia de *gerente/trabalhador*, onde o *gerente* recebe o nome de *GFS master* e cada *trabalhador* recebe o nome de *GFS chunkserver*, sendo que cada *máquina trabalhadora* é um nó do sistema de arquivos, que colabora para servir arquivos. Isso permite reduzir transferências de dados na rede, em comparação, por exemplo, com o tradicional NFS (*Network File System*) do *Linux*, que utiliza normalmente um nó específico e único como servidor. Para garantir disponibilidade, são feitas replicações de dados. Mais detalhes sobre o GSF serão fornecidos na Seção 2.2.7.

As operações paralelas, encapsuladas pelo *framework*, utilizam os códigos das etapas de *mapeamento* e *redução*, formando o aplicativo MapReduce, tratado como *job*. Um usuário submete seu *job* para o *cluster*, a fim de que ele seja organizado em tarefas paralelas, e processado. As tarefas desse *job* são atribuídas pela máquina *gerente* para um conjunto de *máquinas trabalhadoras*, utilizando para isso um sistema de escalonamento. Caso o *cluster* não suporte uma determinada quantidade de tarefas paralelas[1], torna-se necessário que o escalonador acompanhe a necessidade de execução dessas tarefas através de uma fila, a fim de que elas possam ser atribuídas às máquinas que eventualmente ficarem disponíveis. Além das tarefas já escalonadas, o *gerente* pode também replicar tarefas para fins como tolerância a falhas (veja Seção 2.2.6). Durante esse tempo de processamento, o usuário pode acompanhar o ciclo de execução de seu *job*, tendo como facilidade o acesso às informações fundamentais do sistema, como previsão para finalização do seu *Job*.

[1] Por exemplo: sobrecarga de *Jobs*.

2.2.1 Execução do job

O processo de execução de um *job* no *cluster* começa a partir da divisão (*splitting*) dos dados de entrada do usuário (*input data*). Esses dados são divididos em M_i blocos (*input split*), sendo seu tamanho variável (*entre 16 e 64 MB*) e compatível com o tamanho dos blocos do sistema de arquivos distribuído (*chunk size*) (GHEMAWAT *et al.*, 2003).

Cada tarefa de *mapeamento* utiliza um *componente leitor* (*Reader*) para ler o conteúdo do bloco M_i[2] de sua responsabilidade e, em seguida, fatora pares *chave*/*valor* para serem utilizados nas chamadas da função de mapeamento. Nesse cenário, o *componente leitor* não é restrito a ler apenas arquivos. O *framework* disponibiliza uma *interface* para o desenvolvedor implementar seu próprio *Reader*, onde é possível obter dados de fontes diversas, como banco de dados ou *sockets* T*CP/IP*. Na prática, o *Reader* consiste em um iterador de dados.

Durante o *mapeamento*, os dados são gravados em memória, através de *buffer*. Periodicamente, esse *buffer* é particionado (*partition*), classificado pelas chaves (*ordering*) e enviado para *R* regiões/partições no disco local, onde se mesclam. Mais adiante, os dados contidos nessas regiões/partições são opcionalmente reclassificados pelas *chaves*, através da fase de combinação (Seção 2.2.4), produzindo *R* saídas ordenadas. O *framework* permite que o desenvolvedor implemente seu próprio combinador, sendo isso uma forma de reduzir tráfego na rede, uma vez que os dados repetidos são agrupadas em uma única *chave* na etapa de *mapeamento*, para serem utilizados na etapa de *redução*. De fato, na chamada da função de *redução* o propósito é ter uma *chave* única para um conjunto de *valores*, após processamento de vários *trabalhadores* distribuídos.

Nesse ciclo de execução, o *gerente* guarda o *status* de todas as tarefas de *mapeamento* e *redução* (*ociosa*, *processando* ou *completa*), além da localidade e tamanho dos dados de cada tarefa (Seção 2.2.5). Assim, periodicamente e ao final do *mapeamento*, a máquina *gerente* é avisada (*heartbeat*[3]), ficando ciente da localidade dos dados intermediários, tendo portanto a responsabilidade de transmitir essa informação à operação de *redução*.

Existem *R* máquinas que irão realizar a etapa de *redução*, e elas irão utilizar chamada de procedimento remoto (*RPC*) para fazer a leitura dos dados. Após a leitura, é feita uma nova classificação com base nas *chaves* intermediárias, agrupando *valores* intermediários a uma mesma *chave*. Nesse momento, ocorre a comunicação entre tarefas de máquinas diferentes, uma

[2] i corresponde ao identificador da tarefa paralela, enquanto M se refere ao tipo de tarefa, que é do tipo mapeamento.

[3] Mensagens do nó *trabalhador*, enviadas periodicamente ao nó *gerente*.

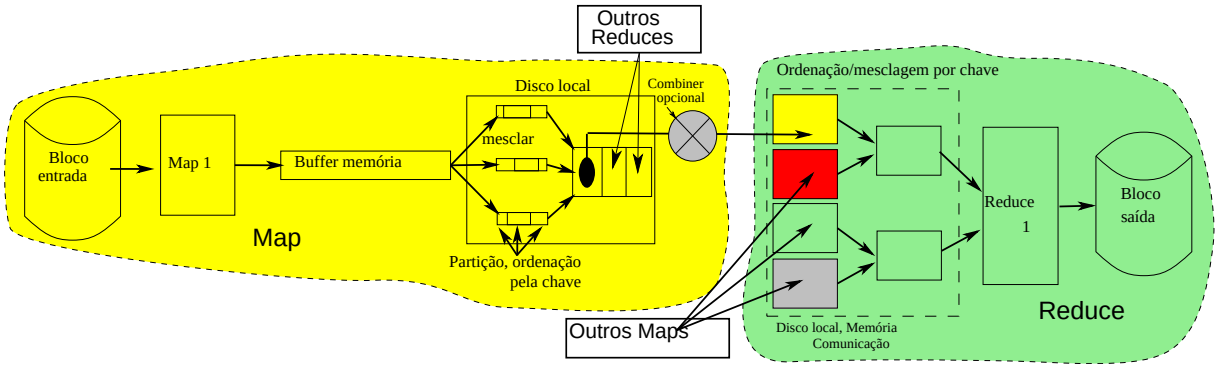


Figura 4 – Partição, Ordenação, Combinação do Hadoop (WHITE, 2009)

vez que *valores* de mesma *chave* estarão possivelmente espalhados no *cluster*.

A Figura 4 demonstra as etapas de uma tarefa de *mapeamento*, com envio de dados da memória ao disco local, classificação e combinação, gerando a saída intermediária que é lida e processada pela *redução*. É importante notar que essa figura não apresenta os detalhes do *embaralhamento*, concentrando-se apenas no armazenamento de dados ocorrido através das tarefas paralelas (`Map 1` e `Reduce 1`). Essa abordagem vislumbra detalhes incluídos na implementação do Hadoop (WHITE, 2009).

2.2.2 Tolerância a Falhas

O tratamento de falhas é fundamental para o bom funcionamento de sistemas potencialmente vulneráveis, como os que pertencem a ambientes distribuídos. O MapReduce é voltado para processamento de larga escala de dados em ambiente distribuído. Por isso, a implementação do Google prevê duas possibilidades: falhas na *tarefa mestre* e falhas nas *tarefas trabalhadoras*.

2.2.2.0.1 Falhas de Processamento em Unidades Trabalhadoras

A identificação de falha em *tarefas trabalhadoras* é obtida através de mensagens periódicas entre elas e a tarefa mestre, de maneira que é estabelecido um tempo limite para atendimento a requisições. Caso alguma *tarefa trabalhadora* não responda nesse tempo limite, a tarefa mestre vai considerá-la como falha. Nesse momento, toda tarefa de mapeamento de posse dessa *tarefa trabalhadora* é considerada perdida, inclusive as completas, pois os dados intermediários estariam no disco local do *trabalhador* inacessível. Dessa forma, a tarefa mestre vai fazer novo escalonamento dessa tarefa de mapeamento em outra *tarefa trabalhadora* disponível, e as tarefas de redução que não leram os dados do mapeamento falho serão informadas

pela tarefa mestre acerca da existência de um novo *trabalhador*. No caso de falhas em tarefas de redução, só é necessário novo escalonamento se a tarefa não foi completada, pois a redução irá gravar seus dados de saída de forma global no sistema de arquivos.

2.2.2.0.2 Falhas de Processamento na Unidade Mestre

A prevenção é feita a partir de cópias periódicas das informações de controle (*metadados*) do *cluster*. No caso de falha da tarefa mestre, o sistema irá utilizar a última cópia, lançando uma nova tarefa mestre, que assumirá o controle a partir daí. Por outro lado, como só existe uma máquina que hospeda a tarefa mestre, caso essa falhe, devem-se abortar as computações em progresso, executando-as novamente para correção. Para corrigir isso, uma possibilidade é através de redundância de *metadado* em outro *host*, embora a falha da máquina onde executa a tarefa mestre seja considerada improvável pelo autor do *framework* (DEAN; GHEMAWAT, 2004), especialmente por se tratar de apenas uma máquina, a qual possivelmente recebe atenção especial. No Hadoop, o sistema de arquivos (HDFS) possui um nó que faz cópias periódicas do *metadado*, chamado *nó de nome secundário* (`Secondary NameNode`), descrito na Seção 2.3.1.

2.2.3 Partição

O usuário do sistema indica o número *R* de dados finais desejados para um *trabalho* MapReduce. A partir disso, as *R* saídas geradas pelas tarefas de *mapeamento* são utilizadas pelas *R* tarefas de *redução*, que produzem *R* dados finais. Nesse cenário, os dados são vistos como um particionamento horizontal (*sharding*), de tal forma que a indexação de uma tarefa de *redução* é definida conforme uma função de particionamento das *R* regiões. Essa função de particionamento visa balanceamento de carga, estabelecendo distribuição dos dados entre as tarefas da forma mais uniforme possível. Por padrão, uma função *hash* é utilizada, indexando o *redutor* conforme o resto da divisão entre a chave e *R*.

2.2.4 Etapa de Combinação

Algumas aplicações MapReduce podem ser implementados com o auxílio de uma rotina opcional ainda na fase de *mapeamento*, chamada de *combinação*, constituindo um pré-processamento para a fase de *redução*. Se considerarmos o impacto da combinação em um *framework*, o objetivo é evitar tráfego na rede, combinando chaves antes no mapeamento para

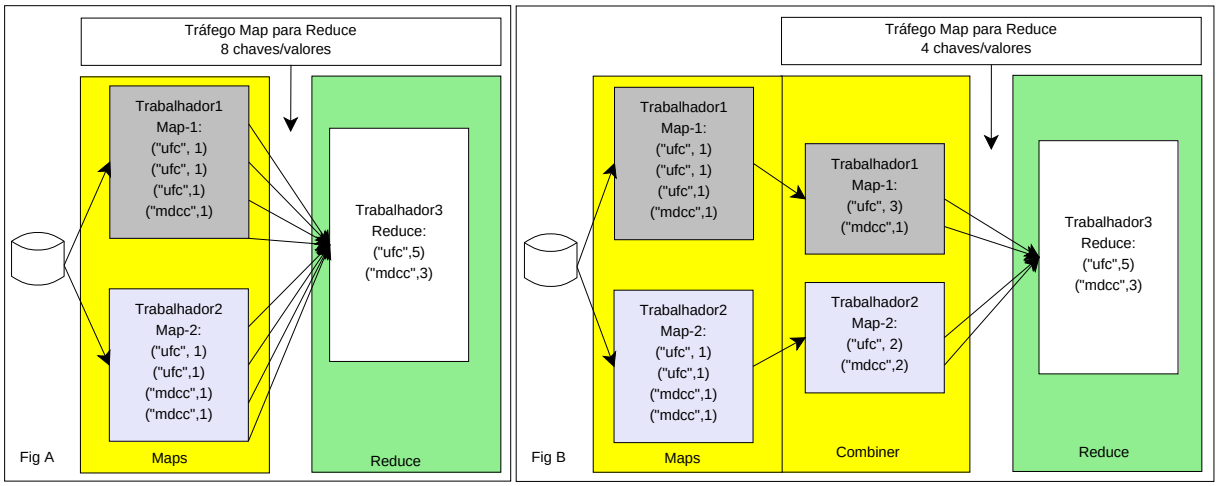


Figura 5 – Execução sem/com Combiner.

diminuir a quantidade de dados que serão utilizados e trafegados na etapa de redução. Ainda, a separação de uma rotina específica para combinar resultados permite divisão de responsabilidades no processador local, útil, especialmente, na paralelização por memória compartilhada, com uso de *threads*, permitindo ao *framework* fazer uso eficiente de computadores com processadores de múltiplos núcleos, bem como com tecnologia HT (*Hyper Threading*^{TM}) da *Intel*.

Por exemplo, um aplicativo que faz uso eficiente da combinação é o de contagem de ocorrências de palavras em um texto. Considere o cenário em que o Algoritmo 1 recebe como entrada o texto manipulado na Figura 5. Na Figura 5.*A*, há uma excessiva transmissão de dados na rede, pois existem *chaves* repetidas na saída intermediária do mapeamento. Por outro lado, com a utilização da combinação, o tráfego de dados intermediário é diminuído em 50%, uma vez que as *chaves* foram convertidas em únicas, reduzindo consideravelmente a quantidade de palavras (de 8 para 4). O Google fez experimentos com a etapa de combinação e obteve redução no tempo de processamento de algumas aplicações (DEAN; GHEMAWAT, 2004).

2.2.5 Escalonador

A responsabilidade de atribuir tarefas é exclusiva do nó *gerente* do *cluster*, onde são tomadas as decisões acerca de qual *trabalhador* executa qual tarefa, utilizando, para isso, um sistema escalonador. O *gerente* guarda as principais informações sobre cada nó e tarefas MapReduce. Essas informações compreendem estado da tarefa, localização e tamanho de dados. Uma tarefa pode estar em três estados possíveis: *ociosa*, *em progresso* e *concluída*. A localização diz respeito a qual nó *trabalhador* e onde estão os dados da tarefa no sistema de arquivos. O

tamanho dos dados diz respeito ao tamanho das *R* regiões produzidas pelo mapeamento. A comunicação entre *gerente* e *trabalhador* no *cluster* é feita por *pulsação* ("*heartbeat network*"). Isso significa que existem mensagens periódicas entre *trabalhador* e *gerente*, de modo que o estado de execução da tarefa no *trabalhador* é monitorado, para controle do *cluster*.

O escalonador tem papel fundamental na obtenção de desempenho a partir da orquestração das tarefas. A estratégia primária dele é guiar as tarefas aos dados, e não o contrário. Considerando a estrutura do sistema de arquivos distribuído, uma tarefa terá prioridade de escalonamento em uma máquina que possa conter dados locais úteis a essa tarefa. Nesse contexto, o escalonador pode se fundamentar em políticas para escolher o melhor destino da tarefa. Por exemplo, uma vez que o sistema de arquivos distribuído é representado pela junção dos dados locais de cada nó do *cluster*, objetivando reduzir tráfego, a política de escalonamento segue a seguinte estratégia:

- A execução da tarefa deve ser feita em um *trabalhador* que contém dados locais para a tarefa;
- Caso o *trabalhador* esteja sobrecarregado, a execução da tarefa deve ser feita em um *trabalhador* que contenha maior largura de banda, pois dados serão transferidos;
- Caso falhe a segunda opção, procura-se outro *trabalhador* disponível;
- Por último, sem *trabalhadores* disponíveis, a tarefa entra em uma fila.

O escalonador aguarda a execução de todas as tarefas de *mapeamento* para iniciar o escalonamento das tarefas de *redução*. Ao final do *mapeamento*, são feitas atualizações de localidade e tamanho das *R* regiões processadas por ele. Caso restem algumas tarefas de *mapeamento* retardatárias, e existirem *trabalhadores* que sinalizaram disponibilidade, o escalonador pode utilizar ainda tarefas de *backups* (Seção 2.2.6) para especular falhas.

2.2.6 Tarefas de Backup

Uma tarefa de *backup* é uma solução de replicação utilizada pelo *framework* MapReduce do Google para melhorar o desempenho do *cluster*, uma vez que algumas tarefas escalonadas podem ter problemas com recursos disponíveis, desempenho ou erros de *hardware*. Como discutido anteriormente (Seção 2.2.5), o nó *gerente* do *cluster* guarda informações a respeito das execuções de tarefas de *mapeamento* e *redução*, de modo que através de *pulsação*, o *gerente* tem ciência sobre o que acontece com o *trabalhador* e o estado de execução de suas tarefas de *mapeamento* e *redução* (ociosa, em progresso ou finalizada), além da localidade e ta-

manho dos dados. Essas informações, entre outras utilidades para o escalonador, são pertinentes para tomada de decisões num eventual problema com o *trabalhador* ou tarefa.

A solução com tarefa de *backup* consiste em descobrir tarefas duvidosas em seu curso normal de execução. Essas possíveis tarefas duvidosas são denominadas *tarefas retardatárias* ("*straggler tasks*"), e geralmente são as últimas a serem finalizadas em rotinas de *mapeamento* e de *redução*. Por exemplo, pode ocorrer de parte das tarefas de *mapeamento* escalonadas não concluírem no tempo tido como usual, segundo os critérios do escalonador, impedindo a finalização global do *mapeamento*. Nesse caso, as não finalizadas são replicadas, geralmente em outras máquinas, em uma espécie de especulação. Inicia-se então um monitoramento dessas tarefas (primária e secundária), de modo que a primeira a finalizar põe fim àquela computação.

Um exemplo de sucesso no uso de replicação é quando existe uma determinada máquina com baixa taxa de leitura/escrita de disco, característica comum em ambiente heterogêneo. Suponha 8% de leitura/escrita (I/O), comparado-se com as demais máquinas de um *cluster*, e que a divisão de trabalho é uniforme. Nesse cenário, todas as demais máquinas finalizam suas tarefas, enquanto, a máquina em questão, não. Nesse caso, replica-se outra tarefa em uma segunda máquina. Assim, existe a possibilidade da tarefa secundária finalizar primeiro que a tarefa primária. Dessa forma, a replicação pode descongestionar trabalhos em execução, melhorando o tempo global do aplicativo, uma vez que esse tempo é dado em função da última tarefa finalizada, bem como o tempo das etapas de *mapeamento* e *redução* é dado em função da última tarefa finalizada. A estratégia de tarefa de *backup* também é utilizada pelo escalonador implementado no Hadoop, mas com a nomenclatura diferente, chamada de execução especulativa de tarefas (*speculative task execution*). Matei Zaharia *et al* (ZAHARIA *et al.*, 2008) fizeram algumas considerações sobre o escalonador utilizado no Hadoop, e afirmaram poder melhorá-lo em máquinas virtuais. Para isso, fizeram experimentos na nuvem da Amazon EC2 (AMAZON, 2013), comparando o escalonador original do Hadoop com o escalonador otimizado por eles, denominado LATE (*Longest Approximate Time to End*).

Um outro exemplo no uso de replicação, agora de fracasso, é quando a tarefa de *mapeamento* realmente está processando um grupo grande de dados para uma mesma *chave*, não liberando a *redução* para a *chave*. O *mapeamento* pode estar, por exemplo, processando ocorrências de muitas palavras iguais em um dado texto, ou fazendo uma combinação com grande quantidade de dados. Enquanto isso, outras tarefas de *mapeamento* teriam finalizado primeiro, devido ao processamento de partes menores. Nesse caso, o lançamento de réplica

apenas consumiria recursos de alguma outra máquina do *cluster*. Entretanto, para algumas aplicações, o Google tem afirmado que diminuiu o tempo de trabalhos MapReduce (*jobs*) em até 44% (DEAN; GHEMAWAT, 2004).

2.2.7 Google File System - GFS

O GFS (*Google File System*) (GHEMAWAT *et al.*, 2003) é um sistema de arquivos distribuído que foi projetado para comportar larga escala de dados, buscando garantir as tradicionais necessidades de ambientes distribuídos, tais como disponibilidade, segurança, desempenho e escalabilidade. Além disso, o GFS objetiva suportar centenas ou milhares de máquinas heterogêneas, entre as quais podem haver componentes de baixo custo que podem falhar repentinamente. Isso levou os projetistas a desenvolver soluções de tolerância a falhas, com monitoramento, detecção de erros e recuperação automática. No ambiente de operação realístico, os desenvolvedores destacam que o sistema tem sido utilizado e testado na crescente e larga base de dados do Google, especialmente com MapReduce. O modelo de comunicação do GFS é semelhante ao *mestre/escravo* ou *gerente/trabalhador*, onde o *gerente* é identificado como *GFS master* e cada *trabalhador* como GFS *chunkserver*, o que é naturalmente compatível com a implementação MapReduce do Google. Dessa maneira, um sistema completa o outro.

O GFS utiliza a hierarquia de diretórios e caminhos, oferecendo ao usuário uma *interface* simples para operações essenciais, como leitura, escrita, abrir, fechar, criar e apagar arquivos. Os arquivos do sistema podem ainda ser lidos ou gravados por vários usuários simultâneos, de modo que o GFS busca garantir a atomicidade através de capturas instantâneas (*snapshot*) das informações (*metadados*) das partes de arquivos. Essas partes são da seguinte forma: no ambiente, os arquivos são divididos em pedaços/blocos de tamanho fixo (considere 64 MB), denominados blocos (*chunks*), o que é semelhante aos setores dos tradicionais sistemas de arquivos locais. Cada *chunkserver* representa uma parte do sistema de arquivos distribuído, armazenando e controlando blocos. No momento da criação do bloco, o nó *mestre* o relaciona a um *metadado* através de um identificador único de 64 *bits*.

Para viabilizar disponibilidade e confiabilidade, por padrão existem três réplicas de blocos na rede, mas é possível ao usuário definir níveis de replicação. Contudo, o nó *mestre* não armazena blocos. Ele apenas guarda os *metadados* associados aos blocos distribuídos. *Metadados* podem conter, por exemplo, identificação do bloco, espaço de nomes de arquivos (*namespace*), locais de réplicas, tarefas em progresso que estão lendo/escrevendo no bloco, além

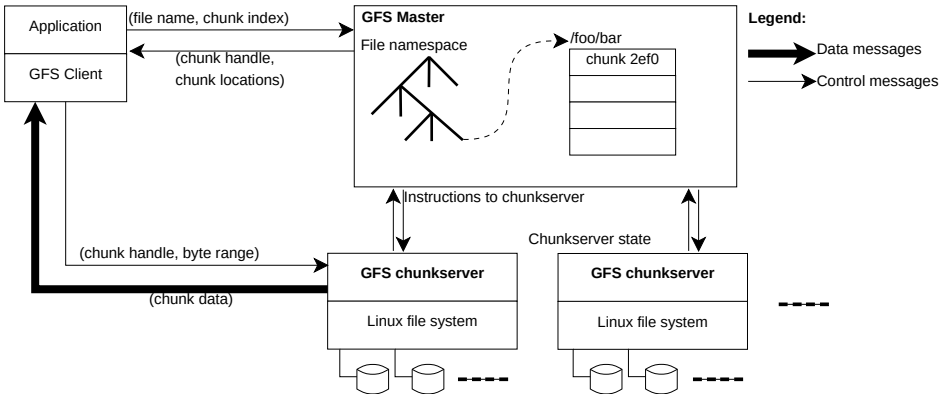


Figura 6 – Arquitetura GFS (GHEMAWAT *et al.*, 2003).

de ser o referencial para garantir atomicidade (*snapshot* do metadado do bloco) para usuários. Nesse cenário, as atualizações de *metadados* são feitas através de pulsações periódicas enviadas pelos *chunkservers* ao nó *mestre*. Dessa forma, a comunicação com o nó *mestre* não requer tráfego intenso (pois não grava blocos), o que é importante especialmente para minimizar gargalo, pois o *mestre* irá atender vários *chunkservers*. Além disso, a comunicação entre nó *mestre* e o cliente usuário [4] também ocorre orientada aos *metadados*, onde o *mestre* libera referências e controles abstratos (*chunk controle, chunk localização*) aos clientes, que podem finalmente ter acesso aos dados dos blocos diretamente nos *chunkservers*. De fato, há uma centralização de responsabilidades na máquina *mestre*, que é ainda responsável pelas estratégias de controle de balanceamento de carga entre os *chunkservers*, e por efetuar coleta de lixo ("*garbage collection*").

No contexto de modelo MapReduce, o GFS é fundamental, pois permite que a implementação MapReduce alcance um de seus principais objetivos, que é o desempenho. Um GFS *master* está para o *gerente* do MapReduce assim como os GFS *chunkservers* estão para os *trabalhadores* MapReduce. Dessa forma, o desempenho é obtido porque o sistema escalonador procura relacionar *máquina trabalhadora* com a tarefa que vai manipular alguma réplica de bloco local na máquina. Portanto, evita-se a transferência de dados, uma vez que as tarefas são levadas aos dados. Dessa forma, a comunicação se faz minimamente em casos específicos, como na transferência de dados pela rede quando não há máquina disponível com dados locais para uma tarefa. A arquitetura do GFS pode ser vista na Figura 6, que é original dos artigos *Google File System*.

[4] Nesse caso, o cliente não é nó *gerente* ou *trabalhador*. Trata-se do responsável pela submissão do *Job*.

2.3 Hadoop MapReduce

O projeto Hadoop tem início em 2006 por iniciativa de *Doug Cutting* e *Mike Cafarella*, tendo como objetivo suportar o projeto Nutch (APACHE, 2003), um motor de busca de código aberto, relacionado ao projeto *Lucene* (CUTTING, 1999), desenvolvido por *Doug Cutting*. Nutch começa em 2002 e, a partir de 2003, com os artigos do *Google File System* (GHEMAWAT *et al.*, 2003), incorpora a ideia do sistema de arquivos distribuído do Google, sendo implementado o NDFS (*Nutch Distributed File System*). Posteriormente, o NDFS é renomeado para HDFS (SHVACHKO *et al.*, 2010) (*Hadoop Distributed File System*) dando início ao projeto independente Apache Hadoop, composto por um *framework* MapReduce e o sistema de arquivos HDFS. Trata-se de uma implementação predominantemente em Java, com algumas bibliotecas nativas em C, de código aberto, sob licença Apache, versão 2.0, e que segue o modelo MapReduce de acordo com as definições fundamentais do *framework* referência do Google.

Existe uma distribuição base da Apache que dá origem a outras versões alternativas. Algumas dessas são DataStax (DATASTAX INC, 2013) e Cloudera (CLOUDERA, 2013), as quais oferecem serviços agregados como suporte e ferramentas utilitárias. Dessa forma, Hadoop tem sido apoiado globalmente por muitos usuários e colaboradores, tais como Amazon, IBM, Yahoo, LinkedIn, EBay, Twitter, Facebook dentre outros. Devido ao significativo número de colaboradores em torno do projeto Hadoop, surgem ainda outros projetos, tais como:

- Apache HBase, um banco de dados não relacional;
- Yahoo Pig, uma Linguagem *Data Flow* para executar e programar aplicações MapReduce em alto nível;
- Hive, um *data warehouse* desenvolvido pelo Facebook;
- Commons, para criação e manutenção de componentes/bibliotecas Java reusáveis no Hadoop, permitindo a integração do Hadoop com outros projetos.

O desenvolvimento desses e outros projetos, que eventualmente surgem para trabalhar juntamente com o Hadoop, forma o *ecossistema* Hadoop. A maioria desses *softwares* tem como colaboradores os primeiros desenvolvedores e utilizadores do Hadoop, ganhando uma página no portal da Apache com ampla documentação, onde é adotado o seguinte endereço: *projeto*`.apache.org`, tal que *projeto* é o nome do projeto desenvolvido. Portanto, o Hadoop é composto de um núcleo (*framework* MapReduce e seu sistema de arquivos HDFS) que incorpora outros projetos e ferramentas, tornado o *framework* cada vez mais completo. A Figura 7 oferece uma visão geral de alguns projetos apoiadores do Hadoop. Nas seções seguintes, abordam-se

elementos fundamentais da arquitetura do Hadoop, tais como: componentes de um *cluster* Hadoop, Hadoop YARN, escalonamento de tarefas, sistema de arquivos distribuído (HDFS).

2.3.1 Componentes de um Cluster Hadoop

Os artigos de *Jeffrey Dean et al* (DEAN; GHEMAWAT, 2004; DEAN; GHEMAWAT, 2008) introduziram o esboço do funcionamento de um *framework* MapReduce, porém não apresentavam muitos detalhes a respeito da implementação. O Hadoop, por ser de código aberto, permitiu uma melhor compreensão nesse sentido, apresentando uma implementação com características semelhantes ao *framework* MapReduce do Google, apesar das terminologias serem diferentes.

O Hadoop é composto de duas principais funcionalidades: armazenar/manipular grande quantidade de dados e executar computações paralelas fazendo uso desses dados. Para isso, o sistema é organizado em um *cluster* que adota o modelo de comunicação *gerente/trabalhador*, onde são trocadas mensagens por *pulsação*. Nesse sistema, são utilizados seis serviços principais, descritos a seguir:

- `JobTracker` (nó *gerente*), responsável por supervisionar e coordenar o processamento paralelo de dados MapReduce, sendo responsável por atividades como distribuição, controle e escalonamento das tarefas entre os *nós trabalhadores*. É ainda responsável por corrigir eventuais falhas, bem como utilizar replicação de tarefas especulativas.
- `TaskTracker` (*nó trabalhador*), responsável por processar as tarefas de mapeamento e redução atribuídas pelo `JobTracker`. Dados produzidos pelo mapeamento são particionados em *R* regiões, ordenados, mesclados localmente/paralelamente pela *chave* para as *R* tarefas de redução. O `TaskTracker` contém um número estabelecido de *slots*, representando a quantidade de tarefas que podem ser executadas no *nó trabalhador*, de maneira que *slots* livres são usados para novas tarefas dadas pelo `JobTracker`. Por isso, `TaskTracker` envia, por *pulsação*, mensagens periódicas ao `JobTracker`, para informar que está operante e que possui ou não *slots* livres. Quando as tarefas do *job* são escalonadas, elas são executadas em processos separados das classes `TaskTracker` no *trabalhador*, para evitar eventuais falhas e queda do processo principal `TaskTracker`. Dessa forma, `TaskTracker` monitora cada tarefa e captura saídas, avisando eventuais ocorrências ao `JobTracker`.
- `NameNode` (*nó de nome*, gerente do HDFS), responsável por supervisionar e coordenar os nós componentes do sistema de arquivos distribuído.

- `DataNode` (*nó de dados*, trabalhador do HDFS). A maioria das máquinas de um *cluster* Hadoop é do tipo *trabalhador* e executa o rastreador de tarefas (`TaskTracker`) e o nó de dados (`DataNode`). Porém, o rastreador de tarefas recebe instruções do rastreador de trabalhos e se preocupa com processamento, enquanto que a parte nó de dados recebe instruções do nó de nome (`NameNode`) e se preocupa com o armazenamento de blocos de dados.
- `Secondary NameNode` (*nó de nome secundário*), responsável pela recuperação e otimização do arquivo de estado do *cluster*. O objetivo não é alta disponibilidade, tal como a entrada do secundário em operação no caso de falha do principal. O objetivo é copiar e manter os registros do *cluster* periodicamente (pontos de verificação de *metadados*), fazendo uma combinação com registros locais, e devolvendo ao nó de nome o último ponto de verificação (*checkpoint*). Por padrão, um ponto de verificação pode ser feito a cada hora. No caso de falhas no nó de nome principal, pode-se reiniciá-lo considerando o último ponto de verificação. Porém, dados em memória no nó de nome que não puderam ser combinados em arquivo se perdem;
- **Cliente**, o qual se trata do nó usuário do *cluster* Hadoop, não sendo nem *gerente* nem *trabalhador*. Contém bibliotecas que permitem a utilização do Hadoop.

2.3.2 Hadoop YARN

O YARN (*Yet Another Resource Negotiator*) foi introduzido no Hadoop a partir da sua versão 0.23. Esse novo recurso trouxe para o Hadoop o conceito de segunda geração, MapReduce 2 ou MRv2, e teve como motivação a expansão dos projetos em volta do *framework* (ecossistema Hadoop), pois o Hadoop foi sendo reconhecido como uma plataforma, passando a ser o centro desses novos projetos. Portanto, o objetivo era escalabilidade. Além disso, implementações anteriores ao YARN apresentavam dificuldades para trabalhar em ambientes acima de quatro mil nós (WHITE, 2009). Essas implementações anteriores ficaram conhecidas como primeira geração, MapReduce 1 ou MRv1.

O Hadoop MRv1 envolve os componentes da Figura 7(*A*). Nessa arquitetura, uma aplicação MapReduce segue os seguintes passos para execução. Inicialmente, o cliente submete um trabalho (*job*). Posteriormente, o objeto *rastreador de trabalhos*, da classe Java `JobTracker`, coordena a execução do trabalho. Por sua vez, o objeto *rastreador de tarefas* executa tarefas indicadas pelo rastreador de trabalhos, implementando a classe `TaskTracker`. Finalmente, o

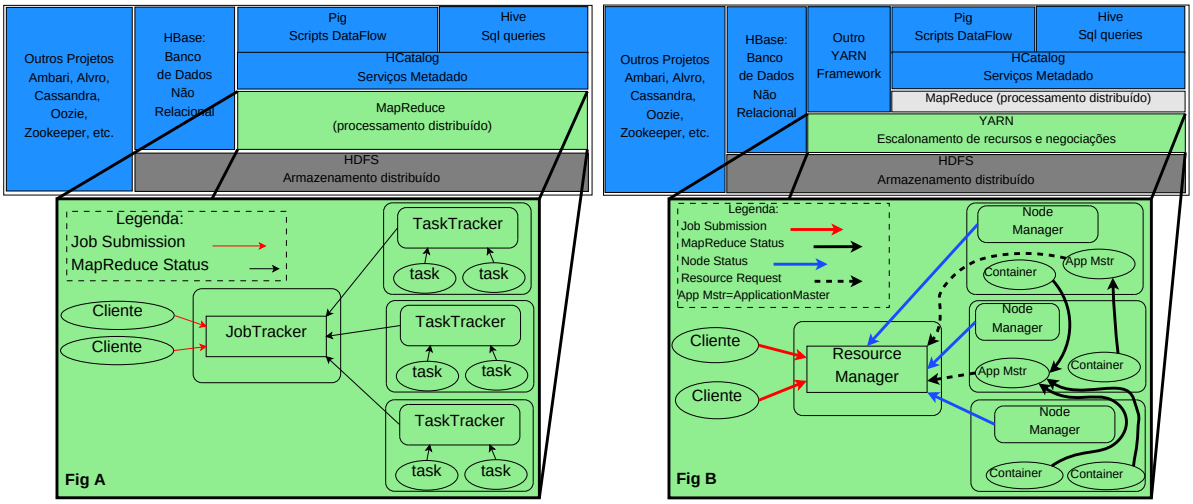


Figura 7 – Visão em Alto Nível do Hadoop sem e com YARN (MURTHY *et al.*, 2014; APACHE, 2005)

HDFS (*NameNode/DataNode*) faz o armazenamento e controle dos arquivos no *cluster*. Os componentes utilizados para executar essa aplicação formam, em alto nível, o núcleo do Hadoop, dividido em *framework* MapReduce e HDFS. Nesse cenário, a ideia do YARN é dividir as duas principais responsabilidades do rastreador de trabalhos (APACHE, 2005), que são o gerenciamento de recursos e o escalonamento/monitoramento dos trabalhos. Para isso, são incluídos, além de recipientes (*containers*), os seguintes *daemons*: Resource Manager, Node Manager e Application Master.

- O recipiente (Container) pode ser visto como o direito de utilizar determinada quantidade de recursos em uma máquina (Node Manager). Os recursos são *CPU*, memória, etc;
- O *gerente de recursos* (Resource Manager) é o *gerente* global escalonador, comunicando-se com *gerente de nó* (Node Manager) e o *mestre de aplicação* (Application Master) para arbitrar e dividir os recursos disponíveis do *cluster* entre as aplicações. Ele busca otimizar a utilização e, para isso, considera atributos como capacidade dos nós e justiça na distribuição de tarefas, bem como garantias de *SLAs* (MURTHY *et al.*, 2014). O gerente de recursos pode trabalhar com várias implementações de escalonadores. Porém, atualmente, as disponíveis são: *fair scheduler*, que funciona em ambiente multiusuário; *capacity scheduler*, também multiusuário; *HOD scheduler - Hadoop on Demand Scheduler*; e o utilizado como padrão, que é *FIFO*, ou fila de processos;
- O mestre de aplicação (Application Master) tem a tarefa de negociar recursos com o *gerente* de recursos (Resource Manager). Para isso, trabalha em conjunto com o *gerente* de nó (Node Manager) no monitoramento das tarefas e do consumo de recursos em recipientes

(*CPU* e memória). O mestre de aplicação se comunica com o *gerente* de recursos através de mensagens do tipo *requisição de recursos* (ResourceRequest), contendo o seguinte cabeçalho: [*resource-name, priority, resource-requirement, number-of-containers*]. O item *resource-name* pode ser uma máquina, um armário de telecomunicação, algum elemento específico na configuração da rede, ou ainda um asterisco ("*") para deixar a negociação em aberto. O item *priority* é a prioridade de uma aplicação. O item *resource-requirement* é o que se requer, como *CPU* e memória. Finalmente, o item *number-of-containers* se refere aos recipientes;

- Node manager está incluso em cada *nó trabalhador*, sendo responsável por iniciar os recipientes das aplicações e monitorar os recursos utilizados por eles, informando os resultados ao gerenciador de recursos.

2.3.3 Sistema de Escalonamento

O Hadoop começou utilizando uma forma básica de escalonamento: a fila (FIFO). As tarefas eram agendadas e esperavam por sua vez pela ordem de chegada. Porém, a estratégia não era muito eficiente, pois uma tarefa poderia utilizar apenas parte dos recursos disponíveis para ela, deixando outras em espera, apesar da disponibilidade de recursos. Para melhorar, posteriormente introduziram o conceito de prioridades [5], o qual coloca à frente da fila as tarefas com as maiores prioridades. No entanto, essa abordagem também pode ser ineficiente se não houver algo mais sofisticado, como *preemptividade*, pois uma tarefa de baixa prioridade pode ser demasiadamente longa e deixar outra de maior prioridade na fila. Ou ainda, em ambiente multi-usuário, tarefas de baixa prioridade sofrer com inanição. Apesar disso, as versões contemporâneas do Hadoop ainda fazem uso do escalonamento *FIFO*, mas permitem a utilização de outros escalonadores. Além do *FIFO*, que é o padrão, estão disponíveis escalonador justo (*fair scheduler*) e escalonador por capacidade (*capacity scheduler*).

O escalonador justo suporta múltiplos usuários[6], preempção e prioridades. Ele busca distribuir os recursos de forma uniforme, permitindo uma parcela média de utilização entre usuários. Logo, um usuário que submete mais trabalhos que outro não irá receber mais recursos por isso. No entanto, caso exista apenas um usuário, esse utilizará todo o *cluster*. Na versão MRv1 do Hadoop, o escalonador justo utiliza o conceito de *pools* e *slots* para escalonar tarefas.

[5] Método `setJobPriority()` na classe `JobClient`.
[6] Considera-se usuário uma aplicação MapReduce

Cada usuário tem um *pool*, e cada *pool* tem uma quantidade de *slots*, de modo que cada *slot* é relacionado a uma tarefa de mapeamento ou redução. Portanto, os trabalhos novos com suas tarefas vão ocupando *slots* livres. Ainda, o escalonador oferece a possibilidade de personalizar a capacidade mínima/máxima de *slots* para o *pool* em arquivo *XML*. Entretanto, *slots* não têm restrições à utilização de recursos (*CPU* e memória), adotando capacidade máxima. Dessa forma, uma tarefa com prioridade baixa e execução longa vai deixar na fila outra de prioridade alta, exceto se for pausada e entregar o *slot*. Numa outra situação, uma tarefa de mapeamento pode utilizar todo o recurso enquanto que uma outra de redução estaria sem recurso. Nesse sentido, houve uma melhora significativa com o YARN MRv2, pois o escalonamento do escalonador justo ocorre com base na utilização quantificada de recursos, e não com *slots* com capacidade ilimitada, como no escalonador justo da versão MRv1. Por exemplo, em uma negociação entre os *daemons gerente de recursos* e *mestre de aplicação*, a aplicação ganha o direito de utilizar determinada quantidade de recursos (*CPU* e memória) em uma máquina (*gerente de nó*), e não o todo da máquina, como ocorre em *slots*.

O escalonador de capacidade foi desenvolvido para suportar o compartilhamento de recursos entre vários clientes, os quais são normalmente empresas. Essas empresas podem fazer uso de equipamentos comuns no *cluster*, mas estão logicamente separadas. Dessa forma, o escalonador de capacidade visa escalonar trabalhos para múltiplos inquilinos. O escalonador deve oferecer ao inquilino garantias de que suas aplicações terão recursos mínimos. Nesse modelo, a vantagem é que a união das empresas financia a manutenção do sistema. Em contrapartida, as necessidades desses inquilinos no *cluster* são variáveis, pois ora sub-utilizam, ora super-utilizam recursos. Assim, um inquilino pode utilizar em algum momento os recursos que não estão sendo utilizados por outros inquilinos. Portanto, a elasticidade é uma característica fundamental para o escalonador, valorizando a relação melhor custo benefício.

2.3.4 Sistema de Arquivos Distribuído do Hadoop - HDFS

O HDFS - *Hadoop Distributed File System* foi projetado tendo como referência o *Google File System* (GFS), para ser o sistema de arquivos distribuído do Hadoop. Possui código aberto implementado em Java, e sua proposta é trabalhar com grande volume de dados em máquinas de alto ou baixo custo, conectadas em rede local. O volume de dados pode atingir a ordem de *petabytes*, o que significa uma potencial divisão entre *hosts*, compatível com a capacidade de cada um deles. Logo, a tolerância a falhas é essencial no HDFS, devido,

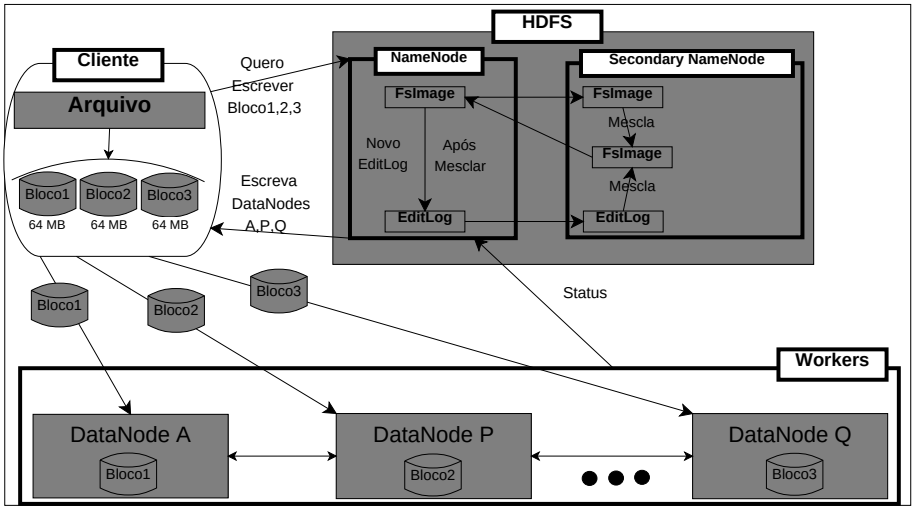


Figura 8 – Armazenamento de blocos de arquivos no HDFS

Adaptado de (HEDLUND, 2011)

especialmente, ao número de máquinas que o sistema pode envolver. Em caso de falha em alguma máquina do HDFS, o sistema deve, de forma transparente, detectar e recuperar, reiniciando ou transferindo o trabalho para outra máquina apta. A arquitetura utilizada é a *gerente/trabalhador*, composta pelo gerente do tipo *nó de nome* (`NameNode`) e os trabalhadores do tipo *nó de dados* (`DataNode`), contendo ainda o recuperador de estado, do tipo *nó de nome* secundário, como descrito na Seção 2.3.1.

O HDFS trabalha com blocos de dados, de forma semelhante ao que foi especificado para o GFS. Dessa forma, antes do armazenamento físico no HDFS, ocorre a divisão dos dados em blocos de tamanhos fixos, que por padrão é *64 MB*. Isso também é feito em sistemas de arquivos tradicionais, que normalmente possuem blocos pequenos, na ordem de *bytes*. Entretanto, como o HDFS trabalha com arquivos grandes, um arquivo pode possuir vários blocos e estar em um ou vários *hosts*. Além disso, o HDFS replica blocos em *hosts* diferentes, sendo uma forma de aumentar o nível de segurança. Assim como no GFS, por padrão, o HDFS replica três cópias de blocos na rede, sendo esse parâmetro configurável. Na distribuição, *nós de dados* podem estar próximos fisicamente, como conectados no mesmo *switch* de rede (mesmo armário de telecomunicação), ou podem estar distantes fisicamente, como conectados através de *backbones* de rede (dois ou mais armários de telecomunicação). No segundo caso, por questão de desempenho e confiabilidade, para recuperação de falha e escalonamento, duas das réplicas devem estar próximas fisicamente (mesmo armário). Além disso, o *nó de nomes* não armazena blocos, ficando essa tarefa para os *nós de dados*.

Para gerenciar o sistema, existem dois arquivos de controle utilizados pelo *nó de*

nome: `FsImage` e `EditLog`. O primeiro é onde estão as informações sobre blocos de arquivos, como localização de réplicas e espaço de nomes (*namespaces*) de diretórios e arquivos. O segundo é semelhante, porém é mais dinâmico, pois armazena o registro de todas as alterações que ocorrem nos metadados dos arquivos. Portanto, a tendência é que o `EditLog` vá crescendo infinitamente. Porém, para evitar esse aumento excessivo, o *nó de nomes* secundário faz uma cópia do `EditLog` e `FsImage`, mesclando-os e enviando cópia para ao *nó de nomes* primário. Então, o *nó de nomes* cria outro arquivo `EditLog` vazio para iniciar novo ciclo. Esse procedimento de segurança é um ponto de verificação (*checkpoint*). Em caso de falha no *nó de nomes*, faz-se uma reinicialização do servidor *gerente* e, após reinício, o *gerente* faz a mesclagem do `EditLog` com o `FsImage`, gerando novo `FsImage` para leitura e configuração do *cluster*. Em caso de perda do arquivo `FsImage`, o *nó de nomes* secundário contém a cópia, embora alterações que não foram mescladas estarão perdidas. O ponto de verificação é feito mediante um tempo configurável no Hadoop. Portanto, as funções do *nó de nomes* secundário são limpar periodicamente o arquivo *log* `EditLog` e manter uma cópia de segurança do arquivo `FsImage`. A Figura 8 mostra uma visão de alto nível das transações com blocos de arquivos, `FsImage` e `EditLog`.

2.4 Limitações MapReduce e Novos Frameworks Emergentes

Juntamente com a ascensão do Hadoop, o interesse por pesquisas com MapReduce ganhou evidência, inclusive com motivação decorrente do sucesso do Google. Nesse período, o Hadoop foi sendo aperfeiçoado. Mas, ao seguir à especificação original do MapReduce, destacaram-se também as limitações do próprio modelo, tais como: expressividade, capacidade de iteração e dificuldade de balanceamento da carga. Isso motivou o desenvolvimento de novos *frameworks* MapReduce, os quais tem aplicado técnicas significativas para melhoria das suas limitações inerentes. Contudo, apesar da oferta de novas soluções, o processamento de problemas em grafos tem sido um dos principais desafios para esses *frameworks*, o que justificou o surgimento de novas plataformas baseadas em um novo modelo chamado Pregel (MALEWICZ *et al.*, 2010), que são, de fato, projetadas para trabalhar especificamente com grafos, sendo vistas com detalhes no Capítulo 3. Na seção seguinte, abordam-se duas soluções emergentes que contribuíram para o processamento MapReduce iterativo, o qual é especialmente útil na computação de problemas em grafos de larga escala, assunto de interesse deste trabalho.

2.4.1 HaLoop e iHadoop

O HaLoop MapReduce consiste em uma extensão do Hadoop com reformulação de alguns módulos, e a inclusão de novas funcionalidades. Um dos objetivos desse *framework* é oferecer suporte eficiente para algoritmos iterativos. Por isso, inclui as seguintes estratégias:

- Oferece uma *interface* de programação com o objetivo de simplificar a expressividade de programas MapReduce iterativos.
- Disponibiliza no *gerente* um módulo de controle de laço que, de fato, permite iterações para algoritmos, onde a condição de parada pode ser definida pelo usuário desenvolvedor. Diferente de uma operação tradicional Hadoop MapReduce, onde cada rodada de mapeamento e redução conta como um *job*, o laço no HaLoop permite a execução do *job* através de uma ou mais etapas de mapeamento e redução, verificando sempre a condição de parada em cada passo.
- Considera um novo escalonador para aplicações iterativas, observando a localização dos dados oriundos dessas iterações;
- Guarda cache de dados invariantes na primeira iteração, o que é especialmente útil para processamento de grafos com estrutura constante no decorrer da execução. O método `AddInvariantTable` é usado pelo programador para informar dados invariantes.
- Guarda *cache* e índice de dados produzidos nos mapeadores/redutores. Isto é, nas unidades *trabalhadores*.

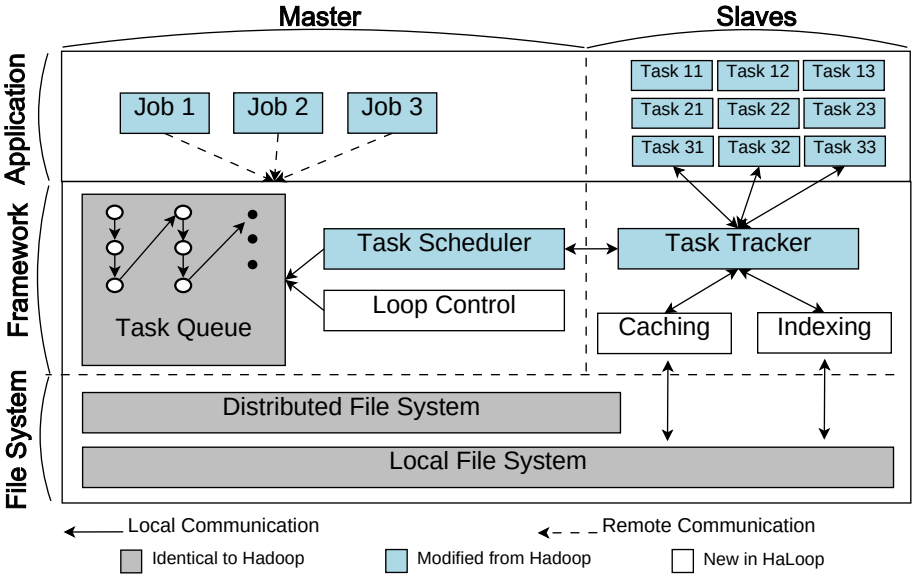


Figura 9 – Framework HaLoop, uma extensão do Hadoop (BU *et al.*, 2010)

A Figura 9 apresenta a extensão do Hadoop. Note que o HaLoop não faz modificações na parte de sistema de arquivos, nem na fila de tarefas. Entretanto, modifica o algoritmos de escalonamento, o qual considera a nova característica iterativa da plataforma, que é apoiada pelo controle de laço (`Loop Control`). Além disso, o rastreador de tarefas (`TaskTracker`) agora faz uso do cache e da indexação de dados. O novo escalonador irá se beneficiar de uma processo chamado *Inter-Iteraction Locality*, entendendo as tarefas de diferentes iterações, mas que acessam os mesmos dados locais. A indexação de dados auxilia na aceleração de leitura feita por tarefas futuras, em uma iteração seguinte.

O iHadoop (ELNIKETY *et al.*, 2011) consiste em uma extensão do HaLoop, descrito anteriormente. De forma semelhante, busca suportar algoritmos iterativos. Porém, explorando operações assíncronas. Para isso, busca adiantar o mapeamento usando um fluxo de dados que sai das reduções. Ou seja, a saída de uma redução imediatamente alimenta a entrada do mapeamento na iteração seguinte.

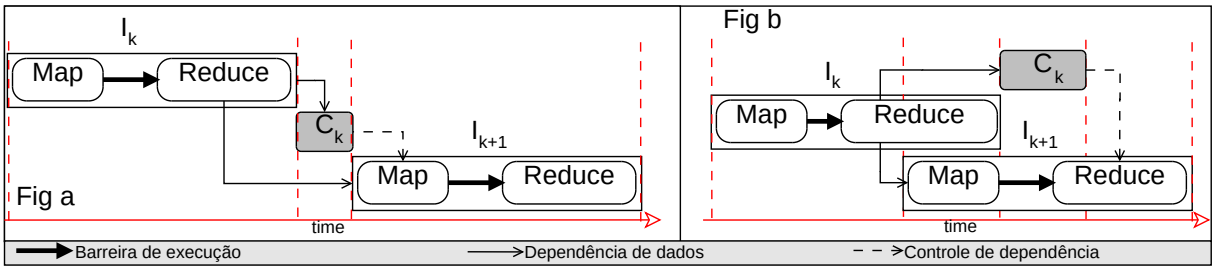


Figura 10 – Dependência e iterações no iHadoop (ELNIKETY *et al.*, 2011)

Na Figura 10b, considere uma iteração I_k e uma condição de parada C_k. A saída da redução na iteração I_k é um fluxo de entrada do mapeamento na iteração I_{k+1}. Dada a condição C_k de finalização, as operações assíncronas vão sendo realizadas, considerando que C_k não irá finalizar na iteração I_k. Portanto, o sistema especula que não haverá finalização, pois C_k será verdade apenas na *n-ésima* iteração. O assincronismo e controle de iteração pode ser visto na Figura 10b, enquanto o sincronismo é visto na Figura 10a, uma vez que é necessária a execução da condição C_k para início do próximo mapeamento/redução. A comunicação assíncrona é feita na forma "*1:L*", onde *L* é o conjunto de tarefas de mapeamento consumidoras e *1* é a tarefa de redução produtora de dados. O escalonador deve garantir que nenhuma tarefa de mapeamento pertencente a *L* esteja na mesma máquina da tarefa de redução produtora.

2.5 Considerações

Neste Capítulo, buscou-se o detalhamento dos conceitos e técnicas por trás do MapReduce, incluindo aspectos técnicos da plataforma de código aberto que tem sido utilizada significativamente por empresas e pesquisadores, o Hadoop. Nas mais recentes pesquisas, essa plataforma tem se destacado como motivador e colaborador de projetos sofisticados com suporte a grafos, tais como Apache Spark GraphX e Apache Giraph, plataformas que estão presentes nos experimentos do estudo de caso deste trabalho de doutorado, representando o estado-da-arte no processamento de grados. No próximo capítulo, é apresentado um mapeamento das soluções emergentes voltadas ao processamento de grafos em larga escala, onde são classificados características comuns entre *frameworks* segundo as suas características.

3 PROCESSAMENTO PARALELO DE LARGA ESCALA DE GRAFOS

Diante das limitações que se destacaram no modelo MapReduce, pesquisadores têm proposto extensões ou definido modelos mais eficientes para determinados domínios de aplicações, tal como processamento de grafos em larga escala, tema de interesse deste trabalho. Dentre as novas alternativas, podem-se destacar aquelas que suportam computação distribuída síncrona, como Pregel (MALEWICZ *et al.*, 2010) (memória distribuída), e assíncrona, como GraphLab (LOW *et al.*, 2010) (memória compartilhada). Essas soluções têm motivado o desenvolvimento de diversos *frameworks* destinados ao processamento de grafos grandes. Este capítulo faz uma revisão dos recentes modelos de *frameworks* e suas técnicas, que visam processamento de grafos grandes em larga escala.

3.1 Definições para Grafos

Nesta seção, são apresentados conceitos fundamentais relacionados a grafos. Essas definições são adotadas ao longo deste trabalho, e estão presentes nas implementações de algoritmos dos *frameworks* apresentados neste capítulo, tais como Spark, Giraph e MR-MPI.

3.1.1 Grafo

Um grafo $G = (V, E)$ é definido por um conjunto V não vazio de vértices, também denotado por $V(G)$, um conjunto de arestas E, também denotado por $E(G)$, e uma função $\psi_G(e)$ que atribui um par não ordenado de vértices para cada aresta e, de modo que esse par pertence a $V(G) \times V(G)$ (BIGGS *et al.*, 1986; BONDY; MURTY, 1976). A exemplo, se e_1 é uma aresta de $E(G)$ e $\psi_G(e_1) = (u, v)$, então u e v são extremidades de e_1 e vértices de $V(G)$. Muitos problemas do mundo real podem ser representados e computados através dessa abstração. Por exemplo, aplicativos de rede social, onde uma pessoa é vista como vértice e seu vínculo com um amigo é visto como aresta.

Graficamente, a representação pode ser feita por pontos ou círculos que representam vértices, unidos por linhas que representam arestas, como exemplificado na Figura 11. O grafo H possui seu conjunto de vértices dados por $V(H) = \{v_1, v_2, v_3, v_4, v_5\}$, e seu conjunto de arestas dadas por $E(H) = \{e_1, e_2, e_3, e_4, e_5\}$. Além disso, a função $\psi_G(e)$ aplicada às arestas e_i retorna o par de extremidades de e_i, como por exemplo: $\psi_G(e_1) = (v_1, v_2)$. Caso essas extremidades sejam idênticas, portanto mesmo vértice, a aresta é chamada de *laço*, como ocorre com a aresta e_5 do

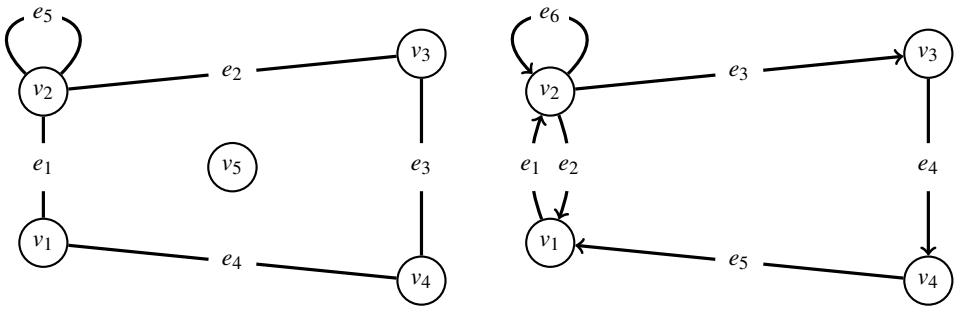


Fig $_1$: Grafo H

Fig $_2$: Grafo D

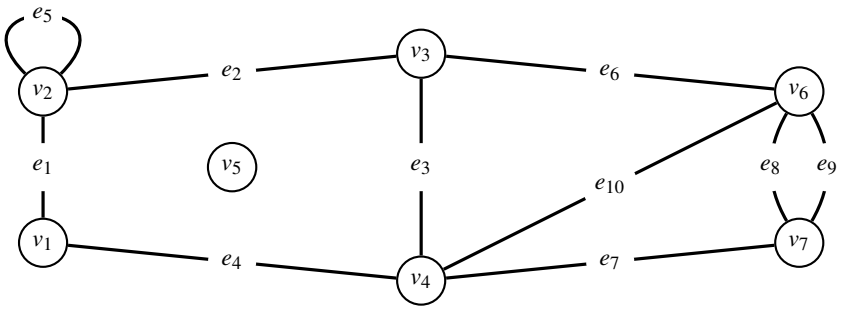


Fig $_3$: Grafo G

Figura 11 – Exemplos de grafos

grafo H. É possível também a ocorrência de *arestas paralelas*, quando duas ou mais arestas partilham o mesmo par de extremidades, como ocorre com e_8 e e_9 no grafo G. Nesse contexto, grafos podem ser ainda subgrafos de outros grafos. Por exemplo, o grafo H será subgrafo de G se $V(H)$ for subconjunto de $V(G)$, $E(H)$ subconjunto de $E(G)$, e ψ_H é uma restrição de ψ_G para arestas de $E(H)$. Partindo dessas definições, um grafo poderá ainda ser classificado de duas formas: um *grafo simples*, quando não existem *laços* ou *arestas paralelas*; um *multigrafo*, caso contrário.

3.1.1.0.3 Grau do vértice

O grau de um vértice v de G, dado pela função $d_G(v)$, é o número *inteiro* de arestas de G que incidem em v. No caso de arestas do tipo laço, por possuirem duas extremidades ligadas a v, deve-se contar duas vezes. Por exemplo, o grafo H na Figura 11 contém o vértice v_2 com grau 4, pois $d_G(v_2) = 4$.

3.1.1.0.4 Representação

Grafos podem ser representados por *matrizes de incidência* ou *matrizes de adjacência*. Em um grafo G com vértices v_i e arestas e_j, uma matriz de incidência $M_{(G)} = [m_{ij}]$ possui

tamanho $|V| \times |E|$, onde $|V|$ e $|E|$ são respectivamente o número de vértices e arestas em G, e m_{ij} é um valor inteiro entre 0, 1 e 2, que define o número de vezes que um vértice v_i e uma aresta e_j são incidentes. Além disso, $1 \leq i \leq |V|$ e $1 \leq j \leq |E|$, tal que i e j são os índices, respectivamente, de uma linha e de uma coluna da matriz $M_{(G)}$. Veja a representação desses detalhes na Figura 12 (**Fig**$_1$). De forma análoga, uma matriz de adjacência $A_{(G)} = [a_{ij}]$ possui tamanho $|V|^2$, onde a_{ij} é o número de arestas com extremidades v_i e v_j. Assim, a matriz de adjacência é menor para grafos que possuem mais arestas que vértices. Na análise dessas representações, especialmente quando a matriz é esparsa, utiliza-se muito espaço ($|V|^2$) para armazenar poucas arestas. Para esse tipo de matriz, há a alternativa da *lista de adjacência*, denotada por $L_{(v)}$, que utiliza, para cada vértice v, um vetor de adjacência acessível em $O(1)$ através de estruturas de dados rápidas, como tabela *hash*. Na Figura 12 (**Fig**$_2$), pode ser observada a representação dessa estrutura de dados.

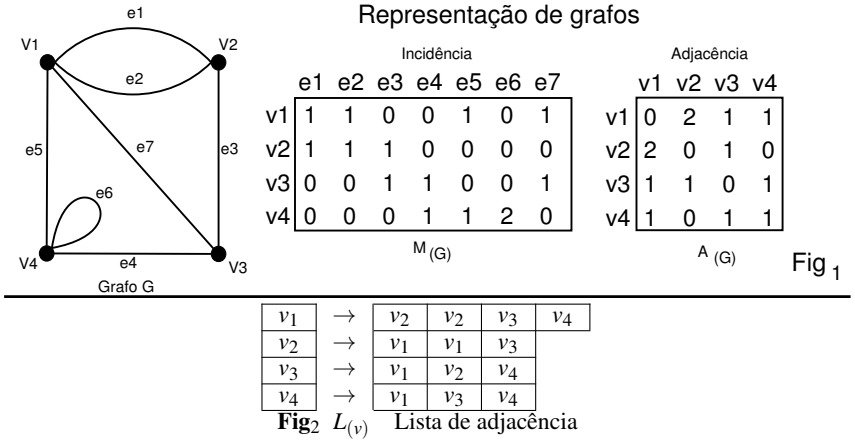


Figura 12 – Lista de adjacência, Matrizes de incidência e adjacência (BONDY; MURTY, 1976)

3.1.2 Digrafo

Alguns problemas representados por grafos utilizam a noção de fluxo. Por exemplo, o tráfego de dados em uma rede de computadores, onde há o início e o fim de um determinado pacote de dados. Para esse tipo de problema, são utilizados grafos direcionados ou simplesmente digrafos. Um *digrafo* $D = (V,A)$ contém um conjunto V não vazio de vértices ($V(D)$), um conjunto A de arestas ($A(D)$), e a função $\psi_D(e)$, que atribui agora um par ordenado de vértices de $V(D) \times V(D)$, para cada aresta e. Ou seja, $(v_1, v_2) \neq (v_2, v_1)$, como pode ser visto no grafo D da Figura 11 (**Fig**$_2$), com a desigualdade $e_1 \neq e_2$. Duas ou mais arestas serão paralelas se

possuírem as mesmas extremidades e orientação.

3.1.2.0.5 Grau (de entrada e saída)

O grau de um vértice em um digrafo é distinguido em *grau de entrada* e o *grau de saída*. O primeiro é dado pelas arestas orientadas que chegam ao vértice, enquanto o segundo é dado pelas arestas que saem do vértice.

3.1.2.0.6 Representação

A representação é um pouco diferente daquela apresentada na Figura 12, pois é necessário incluir a orientação. Para isso, a parte acima e abaixo da diagonal da matriz é usada para definir a orientação. Por exemplo, para linha e coluna em A_G, v_1v_3 poderia ser marcado com 1, enquanto v_3v_1 poderia ser marcado com 0. Isso definiria uma aresta com cauda v_1 e cabeça v_3 ($v_1 \rightarrow v_3$). Para laços, usa-se a própria diagonal, como já é praticado nos grafos não orientados. No caso da lista de adjacência, o desenvolvedor pode utilizar apenas um vetor de saída, se isso for suficiente para oferecer informações ao seu algoritmo. Por exemplo, o Apache Giraph (APACHE, 2011; SHAPOSHNIK *et al.*, 2015) trabalha apenas com grafos orientados e possui informações unicamente das arestas de saída, ficando na responsabilidade do algoritmo descobrir arestas de entrada, se necessário (isso é visto na Seção 3.2). A segunda opção é definir um vetor de entrada e outro de saída, fixando a orientação das arestas com extremidades no vértice.

3.1.3 Algoritmos Comuns em Frameworks Emergentes

Existem diversos problemas em grafos que são de interesse científico, em diferentes aplicações. No processamento de grandes grafos, os *frameworks* existentes tem compartilhado o uso de alguns desses problemas para fins de avaliação de desempenho de suas implementações. Dentre essas problemas, são descritos nas próximas seções aqueles que são considerados mais disseminados e cujos algoritmos potencialmente consomem mais recursos computacionais de processamento, comunicação e memória. São eles: *ranqueamento de páginas* (*PageRank*), *SSSP* (*Single Source Shortest Path*) e *contagem de triângulos*.

3.1.3.1 PageRank

O *PageRank* (PAGE *et al.*, 1999) é um algoritmo que explora a associação entre vértices em um grafo direcionado. Seu objetivo é atribuir importância, denominada *rank*, aos vértices, especialmente através do *grau de entrada*. Um vértice que possui alto *grau de entrada* tende a possuir alto *rank*. Além disso, se um vértice v com alto *rank* possuir uma aresta direcionada para um vértice w, então w também tende a ter *rank* significativo. O algoritmo foi desenvolvido pelos fundadores do Google, *Larry Page* e *Sergey Brin*, para classificar *páginas web* de acordo com a sua popularidade. Os *vértices* representam *páginas web*, e as *arestas* direcionais representam *links* de uma *página* para outra.

Equação 3.1 $rank(j) = c\sum_{i \in L(j)} \frac{rank(i)}{S_{aida}(i)}$

Equação 3.2 $P(j) = \frac{\alpha}{|V(D)|} + (1-\alpha)\sum_{i \in L(j)} \frac{P(i)}{C(i)}$

Considere um digrafo $D(V,E)$ cujas arestas direcionadas são $i \rightarrow j$, onde i e j representam, respectivamente, cauda e cabeça. A Equação 3.1 calcula $rank(j)$, ou seja, o *rank* de um vértice $j \in V$, onde c é um fator constante de normalização[1] da distribuição, compreendido entre 0 e 1 ($0 < c \leq 1$), *L(j)* é o conjunto de vértices que apontam para j, e $S_{aida}(i)$ é o grau de saída do vértice i.

Algoritmo 2: Algoritmo *PageRank* em MapReduce

```
1  inicio()
     Dado um digrafo D=(V,E), todo vértice v ∈ V(D) possui inicialmente rank=1
   Iterate
2    Map(Integer id, Vertex v):
       if(outDegreeOf(v)>0)
         p_i = v.rank/outDegreeOf(v)
         Foreach Vertex w ∈ outgoingVertexOf(v)
           emite(w.id, p_i)
3    Reduce(Integer id, List p_i)
       Vertex V = recoveryVertexById(id)
       Float sum = 0.0
       For each float p ∈ p_i
         sum += p
       V.rank = 0.15/|V(D)| + 0.85*sum
       emite(id, V)
```

Em uma variação dessa equação, *Lin* (LIN; DYER, 2010), bem como os exemplos do Giraph (SHAPOSHNIK *et al.*, 2015), têm apresentado um algoritmo *PageRank* em MapReduce

[1] Leva em conta o fluxo de *navegação* de usuários, que seguirão algum *link* da página.

com base na Equação 3.2, descrita como: $P(j)$ é o *rank* de um vértice j; $|V(D)|$ é o total de vértices no grafo D; α é o fator probabilístico de um usuário ir (sem usar *links*) para uma página aleatória; $L(j)$ é o conjunto de páginas que possuem *links* para j; e $C(i)$ é o grau de saída do vértice i. Cada *rank* $P(i)$ é a probabilidade de se chegar ao vértice i. Nesse contexto, a probabilidade de se chegar a j por i é $\frac{P(i)}{C(i)}$. Ou seja, a chance de estar em i é dividida pelo número de possibilidades, pois um de seus *links* vai a j[2]. Assim, para calcular o *rank* de j a partir de i, deve-se somar todos os termos $\frac{P(i)}{C(i)}$. Essa soma é multiplicada pela chance do usuário seguir algum *link* da página i, que é $(1-\alpha)$. Ou seja, por se tratar de usuários seguindo *links* em *páginas*, é possível que ele não os siga, e simplesmente digite outro endereço no navegador, o que ocorre com probabilidade α.

O Algoritmo 2 implementa a Equação 3.2 em `MapReduce`. Pode-se iterar um número fixo de vezes, ou adicionar um limite de precisão (*erro*) para os vértices. Por exemplo, *rank* novo do vértice subtraído do seu *rank* atual, o que tenderá a zero com o aumento das iterações, deixando cada *rank* estável.

3.1.3.2 Single Source Shortest Path - SSSP

Na teoria dos grafos, a identificação de um *caminho mínimo* entre dois vértices é um problema cuja solução satisfaz outros problemas práticos, como aqueles envolvendo roteamento em redes de computadores e logística de transporte terrestre. Para definição de *caminho mínimo*, considere inicialmente um grafo $G = (V,E)$, cujas arestas $e \in E$ possuem peso $w \geq 0$ associado. O caminho mínimo será a menor soma de pesos w para um conjunto sucessivo de arestas e_k que conecta um vértice *fonte* e um *destino*. Nesse contexto, o *SSSP* consiste em encontrar o menor caminho entre um vértice *fonte* $i \in V$ e todos os outros vértices $j \in V$, onde o objetivo é minimizar a soma dos pesos de i até j. Existem algoritmos tradicionais que resolvem esse problema, tais como o algoritmo de *Dijkstra*, que considera arestas com peso maior ou igual a zero, e o algoritmo *Bellman-Ford*, que considera arestas com peso tanto positivo como negativo, porém sem ciclos negativos.

Em *MapReduce*, as implementações de validação de alguns *frameworks* citados (ZAHARIA *et al.*, 2010; SHAPOSHNIK *et al.*, 2015; PLIMPTON; DEVINE, 2011) neste capítulo têm usado a lógica do Algoritmo 3. Esse código trabalha com grafo com peso em arestas. Todo vértice é inicialmente ativo e passa pela etapa de *mapeamento*, possuindo distância infinita,

[2] São excluídas redundâncias de *links*.

Algoritmo 3: Algoritmo *SSSP* em *MapReduce*

```
1 inicio()
    Dado G = (V,E) com peso, vértice fonte possui distância=0, e os demais distancia=∞
  Iterate
2   Map(Integer id, Vertex v)
      if(v.dmin!=∞)
        float d = v.dmin
        Foreach Vertex w ∈ neighborsOf(v)
          emite(w.id, d + edgeWeight(v,w))
3   Reduce(Integer id, List d_i)
      Vertex v = recoveryVertexById(id)
      float dmin = v.dmin
      Foreach d ∈ d_i
        if d < dmin
          dmin = d
      if(dmin!=v.dmin)
        v.dmin = dmin
        emite(v.id, v)
```

exceto o *fonte*. Na primeira iteração, apenas o vértice *fonte* emite mensagens no *mapeamento*, avisando seus vizinhos sobre sua distância. Na redução, esses vizinhos estão ativos, e devem buscar distâncias menores que `v.dmin`, que é a variável de armazenamento. Ao constatar uma opção menor, o vértice é atualizado e se ativa para o próximo mapeamento da próxima iteração, através da função *emite*. Esse processo acaba quando não há mais emissores na *redução*.

3.1.3.3 Enumeração de Triângulos

Um triângulo T, subgrafo de $G = (V,E)$, é composto por três vértices v_1, v_2, v_3 $\in$ *V(T)* e um conjunto de arestas $E(T)$ não direcionadas, e que formam um *ciclo* de tamanho três: $e_1(v_1,v_2), e_2(v_2,v_3), e_3(v_1,v_3)$. Para enumerar todos os triângulos de um dado grafo G, uma estratégia possível é a utilização de duas etapas (COHEN, 2009). Na primeira, faz-se a varredura do grafo para encontrar pares de arestas adjacentes *candidatas* à formação de triângulos: $e_1(v_1,v_2)$ e $e_2(v_2,v_3)$. Na segunda, faz-se a verificação da existência da terceira aresta $e_3(v_1,v_3)$, uma vez que ela fecha o *ciclo*, formando um triângulo válido.

Para efetivar essas etapas, pode-se atribuir números identificadores inteiros aos vértices, e ordenar arestas da baixa para alta ordem, tal como $e_k(i,j)$ para $i < j$, de modo que em uma matriz M_{ij} será usada apenas sua parte superior. Por exemplo, com o triângulo *candidato* formado pelas arestas $e_1(1,2)$ e $e_2(1,5)$, é necessária a aresta $e_3(2,5)$ para o triângulo existir, e o algoritmo deve identificá-la para validar o triângulo.

Considerando o custo dessa estratégia, podem existir vértices de alto grau, como

Algoritmo 4: Algoritmo de Enumeração de Triângulos

```
1  inicio()
     Um grafo G = (V,E) particionado em subgrafos para processamento distribuído
     Cada unidade distribuída map/reduce processa um conjunto de vértices
   MapReduce1
2    Map1(Vertex v, Messages null)
       Foreach Vertex w in neighborsOf(v)
         if (v.id < w.id)
           emite (w, v)
3    Reduce1(Vertex w, Messages v_i)
       Foreach Vertex z in neighborsOf(w)
         if (w.id < z.id)
           Foreach Vertex v in v_i
             emite (v, z)
   MapReduce2
4    Map2(Vertex v, Messages z)
       emite (v, z)
5    Reduce2(Vertex v, Messages z_i)
       Integer count = 0
       Foreach Vertex z in z_i
         if(neighborsOf(v).Contains(z))
           count = count + 1
       emite (v, count)
```

$e_1(1,2)$, $e_2(1,3)$, $e_3(1,4)$, $e_4(1,5)$, ..., $e_n(1,k)$, onde 1 é o vértice de grau n. Quanto maior o grau, maior o número de combinações de triângulos candidatos, que podem ser definidos com suas arestas adjacentes. Por exemplo, o vértice 1 de grau $n > 1$ gerará $\frac{n(n-1)}{2}$ triângulos candidatos para serem testados.

Para minimizar isso, alguns algoritmos consideram o menor grau do vértice para gerar triângulos *candidatos*. Caso existam dois vértices de mesmo grau, prioriza-se aquele com menor *id*. O *framework* MR-MPI (PLIMPTON; DEVINE, 2011) implementa essa técnica. Contudo, o mais comum são versões que comparam apenas *ids* dos vértices para lançar um triângulo candidato. Por exemplo, o Apache Giraph, o qual tem feito parte do nosso estudo de caso, utiliza a comparação de *ids*.

3.1.3.3.1 Algoritmo:

o Algoritmo 4 considera duas rodadas MapReduce, e busca encontrar triângulos com vértices crescentemente ordenados v_1, v_2, v_3. Para isso, cada unidade paralela de *mapeamento* e *redução* possui a estrutura do grafo, que foi previamente particionado em subgrafos, definindo um conjunto de vértices para cada unidade computacional. No **Map1**, a entrada é um vértice com mensagens nulas, pois é o primeiro passo do processamento. Apenas vértices vizinhos maiores que o *v.id* são emitidos para o **Reduce1**. Por sua vez, essa primeira redução deve emitir, para cada

mensagem recebida, os vizinhos com *id* maior que o *id* do vértice em processamento, lançando um candidato a triângulo para ser verificado no próximo etapa MapReduce. **Map2** simplesmente repassa *chave/valor* para o próximo *Redutor*. **Reduce2** deve avaliar se cada mensagem recebida é um vizinho do vértice em processamento, contando um triângulo no caso afirmativo. Por fim, emite-se como *valor* a soma parcial de triângulos, tendo como *chave* o vértice de menor *id*.

3.2 Pregel e o Framework *Apache* Giraph

A partir do MapReduce, emergiram novas soluções para o processamento de dados em larga escala, resultado de pesquisas que identificaram novos requisitos não previstos ou especificados em *frameworks* MapReduce. Nesse contexto, os desenvolvedores do Google perceberam que era necessária uma metodologia específica de programação e comunicação de dados que satisfizesse os interesses da computação de grafos. Com essa motivação, introduziram a especificação Pregel (MALEWICZ *et al.*, 2010), de código fechado, sendo suas principais características incluídas recentemente nos *frameworks* Spark (XIN *et al.*, 2013), Giraph, Pregelix (BU, 2013) e ExPregel (SAGHARICHIAN *et al.*, 2015).

No modelo Pregel, a entrada de uma computação é um grafo direcionado, dividido entre um conjunto de máquinas na forma de subgrafos. Essa divisão é realizada através de uma função *hash*, pela qual é calculada uma máquina de destino, ou partição, com base no identificador *id* de cada vértice.

O modelo adota a estratégia *gerente/trabalhador*. *Nós trabalhadores* realizam o processamento de vértices numa etapa denominada *superstep*, inspirada no modelo BSP *(Bulk Synchronous Parallel)* (VALIANT, 1990), como demonstrado na Figura 13a. Um *superstep* é um passo iterativo na execução de vários vértices, usando uma função programada pelo usuário, denominada *compute*, que representa um único vértice em processamento. O *gerente* coordena cada passo iterativo, que é constituído de processamento e sincronismo. Em cada uma dessas iterações, os *trabalhadores* retornam informações para o *gerente*, no processo de agregação (Figura 13b). Os dados de cada vértice incluem: *valor*, lido e escrito respectivamente pelos métodos `getValue` e `mutableValue`; *buffer* de *entrada*, incluindo mensagens enviadas por vizinhos de entrada no *superstep* S e recebidas via *iterator* no *superstep* S+1, sem garantia de ordem; *buffer* de *saída*, incluindo mensagens enviadas para vizinhos de saída; *estado*, podendo assumir os valores *ativo* ou *inativo*.

Em cada *superstep*, o nó *gerente* deve decidir se a execução deve ou não ser finalizada.

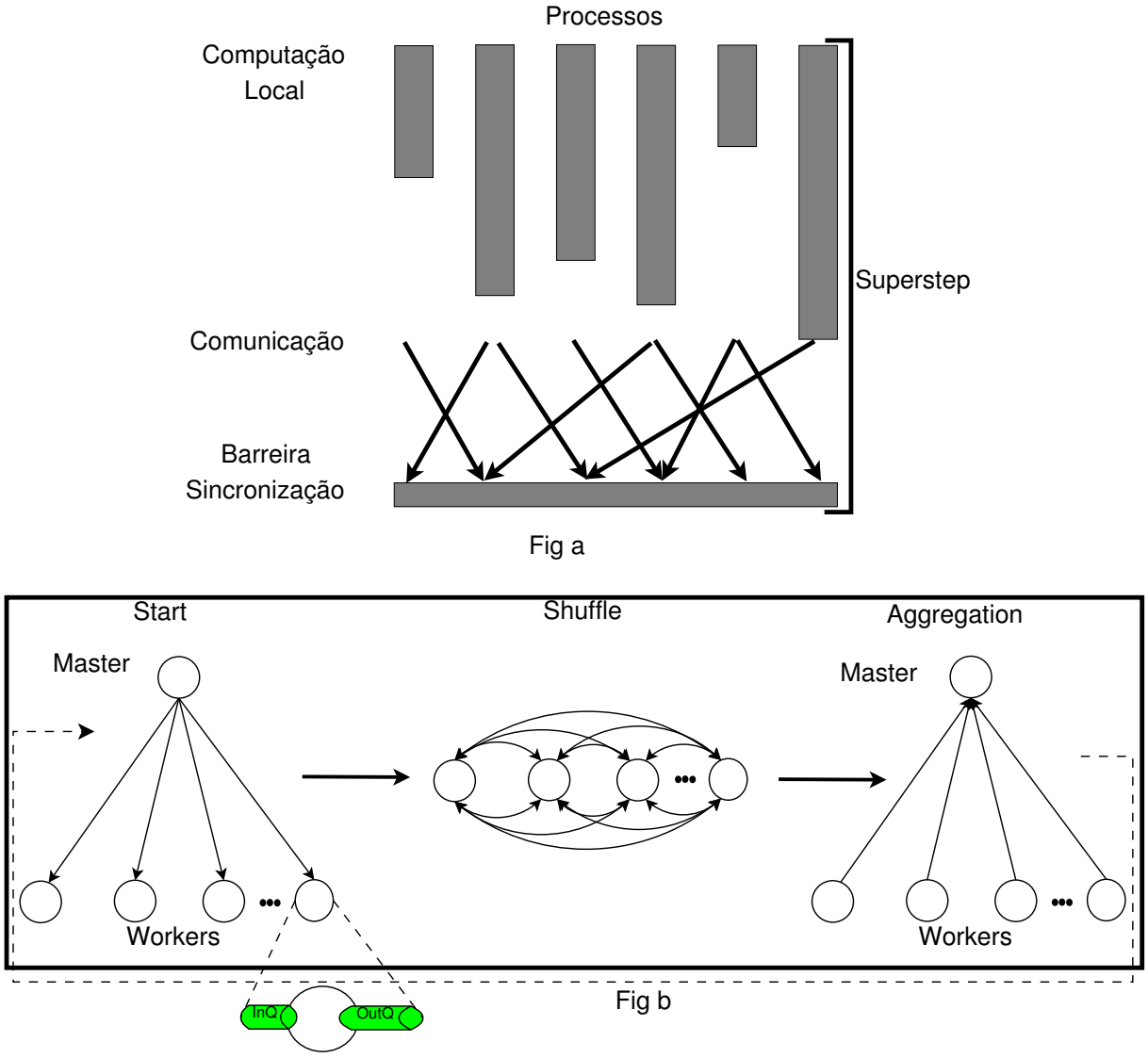


Figura 13 – Modelo de programação BSP (Fig a) e Pregel (Fig b) (SAKR, 2014; XIA *et al.*, 2015)

Para isso, o atributo de *estado* é observado. Enquanto existir *estado ativo*, a execução deve prosseguir. Durante essa execução, se o atributo *valor* não sofrer mudança, o *estado* do vértice é marcado como *inativo*. Porém, se o vértice processado receber alguma mensagem de outro vértice, esse *estado* é novamente *ativado*. Portanto, vértices poderão enviar mensagens aos seus vizinhos sempre que seus *valores* forem alterados. Não havendo essa alteração, são desativados, permitindo ao gerente finalizar a aplicação.

3.2.1 Apache Giraph - Uma Implementação Pregel

O Giraph é implementação do modelo Pregel em Java, que executa sobre o Hadoop e HDFS, suportando um modelo de programação centrado em vértices. O desenvolvedor

implementa seu algoritmo dentro da função *compute*, e o Giraph oferece um objeto do grafo denominado *vertex*, que é um vértice do grafo, e um objeto iterador chamado *messages*, que contém as mensagem destinadas ao vértice em processamento. Esse *framework* suporta apenas *digrafos*, e o objeto *vertex* não possui informações sobre vértices de entrada. Na documentação, argumenta-se que esse tipo de situação pode ser contornado tornando um digrafo em grafo não orientado, a partir da duplicação de arestas. Por exemplo, se existe uma aresta $i \rightarrow j$, cadastra-se também uma aresta $j \rightarrow i$. Outra alternativa é utilizar um *superstep* exclusivo para enviar mensagens do tipo *id* de vértice, com destino aos vizinhos de saída, pois cada um desses vizinhos destinatários *no superstep S* saberá seus *ids* de entrada *no superstep S+1*.

No Algoritmo 5, tem-se um exemplo do objeto *vertex* e a metodologia de envio/-recebimento de *mensagens*. Segundo esse código, *vertex* terá *id*, *valor* e *mensagens* do tipo `IntWritable`, com peso de aresta nulo (`NullWritable`); o método `vertex.getEdges` retorna arestas de saída; o método `sendMessage` permite enviar *mensagens* para serem recebidas no próximo *superstep*; `vertex.setValue` define o valor de um vértice; e `vertex.voteToHalt` afirma que o vértice não quer ser processado no próximo *superstep*. O *framework* possui uma variável inteira que representa um *superstep*. No exemplo, descreve-se a computação que será realizada no *superstep* 1. Porém, a função *compute* será iterada muitas vezes, sendo que em cada iteração a variável `superstep` é incrementada, podendo seu valor ser obtido pelo método `getSuperstep`. A instrução `vertex.voteToHalt` no final da função `compute` marca todo vértice como candidato a não ser processado no próximo *superstep*, mas vértices que recebem mensagens automaticamente são reativados para processamento.

Para que seja possível processar um algoritmo no Giraph existem ainda dois aspectos a considerar: o formato de entrada do grafo, e o formato de saída dos resultados processados. Para o primeiro aspecto, existem as classes do tipo `InputFormat`, que são programadas para esperar dados de um determinado tipo, como *id* de vértices do tipo *integer*, *long*, *text* etc, que serão lidos de arquivos no HDFS. Essas classes instruem o *framework* sobre o formato de entrada do grafo disposto nos arquivos em disco. De forma análoga, existem as classes do tipo `OutputFormat`, que são programadas para interpretar os dados de saída em um dado formato, e enviá-los ao disco, no sistema distribuído de arquivos.

Algoritmo 5: Exemplo da funcão *compute* em Giraph

```
1 Compute Function em Giraph
    public void compute(
    Vertex<IntWritable, IntWritable, NullWritable> vertex,
    Iterable<IntWritable> messages) throws IOException {
     if (getSuperstep() == 1) {
       for (Edge<IntWritable, NullWritable> edge: vertex.getEdges()) {
         if (edge.getTargetVertexId().get() > vertex.getId().get()) {
           for (IntWritable message: messages) {
             sendMessage(message, edge.getTargetVertexId());
           }
         }
       }
     }
     vertex.setValue(new IntWritable(0));
     .......
     vertex.voteToHalt();
    }
```

3.3 GraphLab e seu Ambiente de Comunicação

Os *frameworks* para processamento de grafos grandes podem ser subdivididos pela utilização de memória distribuída e compartilhada. Dentre aqueles de memória compartilhada, GraphLab têm sido considerado o estado-da-arte, sendo uma alternativa para processamento de grafos em computadores com escalabilidade vertical. Sua característica principal consiste em explorar comunicação assíncrona entre vértices, reduzindo tempo de execução.

O desenvolvimento desse *framework* inspirou novas abstrações, como PowerGraph (GONZALEZ *et al.*, 2012), e implementações de *frameworks* de memória distribuída, como PowerSwitch (XIE *et al.*, 2015), e de memória compartilhada, como GasCL (CHE, 2014). PowerGraph introduziu o modelo GAS, implementando em ambiente distribuído o modelo de consistência de dados do GraphLab, o qual define o escopo da consistência de dados do grafo, como mostrado na Figura 14: consistência de vértices, consistência de arestas e consistência total. O PowerSwitch é uma implementação baseada em PowerGraph, e busca explorar o modo híbrido (sincronismo e assincronismo) na comunicação em ambiente distribuído. Por sua vez, GasCL é de memória compartilhada, e possui as três fases de execução (*scatter*, *gather*, *apply*) centradas em vértices. Entretanto, enquanto GraphLab trabalha com algoritmos processados em CPU, GasCL é voltado para processar grafos em GPUs.

Os *frameworks* com memória distribuída são os mais comuns, dada a vocação a que foram propostos, onde dados muitas vezes excedem a capacidade de armazenamento e processamento em um único local. Além disso, permitem escalabilidade horizontal. Para comunicação, podem ser usados recursos como bibliotecas de troca de mensagens (PLIMPTON; DEVINE, 2011; KHAYYAT *et al.*, 2013). Nesse cenário, surgiram *frameworks* com controlador

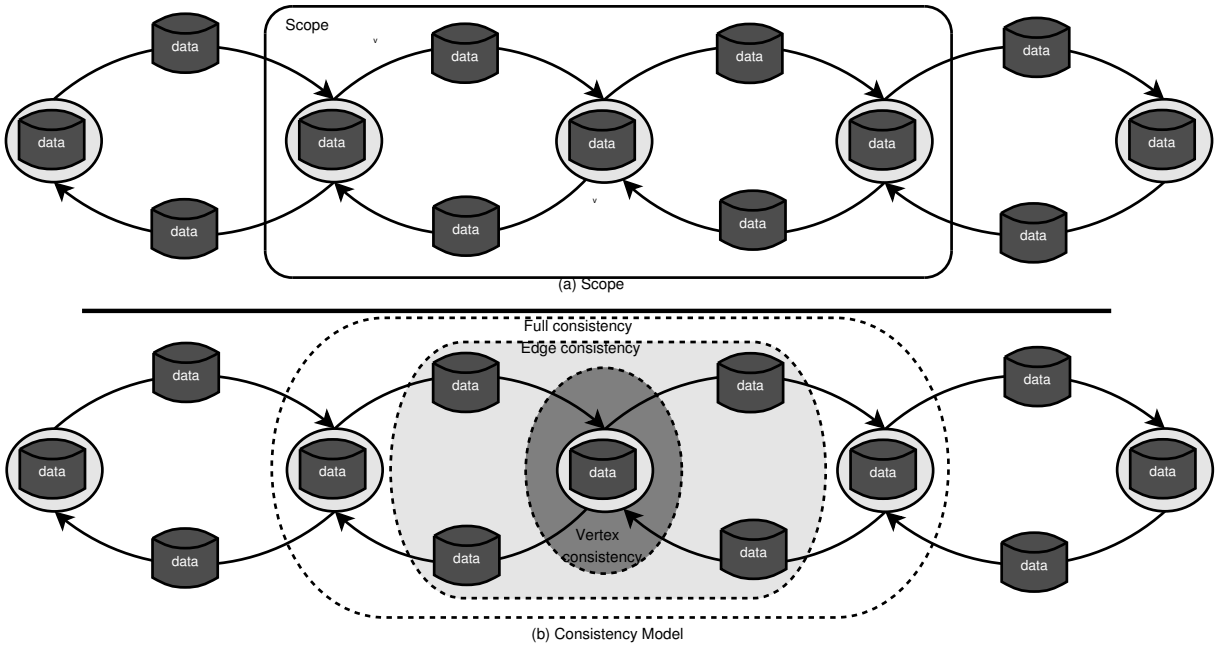


Figura 14 – Modelo de consistência do GraphLab (LOW *et al.*, 2010).

centralizado (*Master/Slave*), mas também surgem opções **descentralizadas** (KHAYYAT *et al.*, 2013; MAROZZO *et al.*, 2012), onde o controle é todo realizado pelos *nós trabalhadores*.

No GraphLab, seu modelo de consistência passa pelo controle de operações paralelas em memória compartilhada, o que envolve semáforos, monitores e técnicas de programação concorrente. Por outro lado, como explorado no PowerGraph, essa consistência deve passar pelo controle da distribuição e persistência de vértices e arestas distribuídas, pois o grafo deve ser particionado. Por exemplo, em ambiente distribuído, a *Full Consistency* do GraphLab deve considerar uma partição que ocorre em arestas ou vértices. Sobre isso, a próxima seção apresenta algumas soluções que têm sido usadas no particionamento de grafos grandes.

3.4 Particionamento e Distribuição de Grafos de Larga Escala

3.4.1 GraphBuilder

De iniciativa da *Intel*, o *framework* GraphBuilder (JAIN *et al.*, 2013) pode ser integrado ao Hadoop, e oferece ferramentas para facilitar a construção de grafos, bem como para balancear a carga de processamento em MapReduce. Oferece uma abordagem similar às séries de *extração*, *transformação* e *carga* usadas com *Data Warehouse*, ou *ETL* (*Extract Transform Load*), visando garantir os requisitos de representação e catalogação de grafos.

Na *extração*, os dados são obtidos a partir de diversas fontes ou sistemas. O formato heterogêneo desses dados é considerado, convertendo-os em representação específica da construção de grafos (tabela), permitindo o processamento da próxima etapa, que é a *transformação*. Na *transformação*, ocorre a aplicação das regras que visam limpar informações indevidas (como dados redundantes), unificando e padronizando. A parte da *carga* consiste em atribuir dados aos sistemas nas unidades de processamento, particionando, serializando e comprimindo grafos. Aqui, o aspecto crítico é o particionamento[3], pois trata-se de um problema *NP-difícil* (XU *et al.*, 2015; JAIN *et al.*, 2013). Um método comum para partição do grafo consiste em dividir um conjunto V de vértices em k subconjuntos disjuntos balanceados, minimizando o total de arestas com extremidades em subconjuntos distintos (partições).

Nesse contexto, surgem duas abordagens: partição de grafos em modo *offline* (KARYPIS; KUMAR, 1996) e partição de grafos em modo *online* (STANTON; KLIOT, 2012; TSOURAKAKIS *et al.*, 2014). O primeiro requer como entrada todo o grafo para fornecer uma solução. O segundo assume que os vértices e arestas do grafo são carregados continuamente em um fluxo (FILIPPIDOU; KOTIDIS, 2015). Ao processar um vértice ou aresta, decisões sobre as partições de destino devem ser tanto quanto possível rápidas, minimizando computações. Particionamento *online* é necessário especialmente quando um grafo é grande o suficiente para não se encaixar em uma máquina, seja por armazenamento, seja por processamento. A ideia é utilizar um *agente particionador* que lê vértices e arestas em série, devendo imediatamente decidir sua partição destino. Essa alternativa não encontra uma solução ótima, mas é uma estratégia realística para conjuntos de dados grandes, que potencialmente atingem os limites computacionais da plataforma de execução alvo.

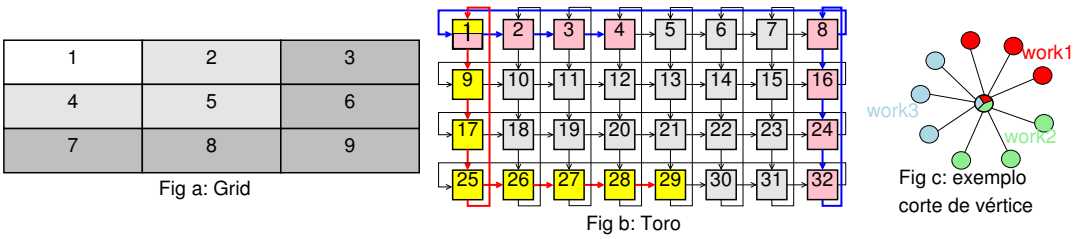


Figura 15 – Esquema de particionamento em *grade* (Fig a) e em *tórus* (Fig b). Exemplo de corte de vértice na Fig c. Fonte (JAIN *et al.*, 2013).

O GraphBuilder adota o modo *online*, com cortes de vértices ao invés de arestas,

[3]Possibilita balanceamento de carga e minimização de comunicação.

justificado pelos bons resultados obtidos em grafos cujos graus de vértices seguem a *lei de potência* (*power-law function*) (ABOU-RJEILI; KARYPIS, 2006), comum em grafos de larga escala, onde há muitos vértices com baixo grau e poucos vértices com alto grau. Apesar do corte de arestas apresentar balanceamento de carga satisfatório, também alcança pior caso em comunicação (GONZALEZ *et al.*, 2012). Por exemplo, a divisão de vértices por função *hash*, usada na especificação do modelo Pregel, é vista como corte de arestas, pois não atribui um mesmo vértice para duas ou mais máquinas. O contrário ocorre com corte de vértices. É assinado um mesmo vértice para uma ou mais máquinas, sendo um deles o vértice real, e os demais são réplicas. De fato, o corte de vértices busca atribuir arestas às máquinas, enquanto o corte de arestas busca atribuir vértices (DOEKEMEIJER; VARBANESCU, 2014).

A estratégia usada no GraphBuilder é atribuir uma aresta **e={u,v}** a uma máquina **A(e)** $\in \{$**1, 2, 3, ..., p**$\}$, onde **p** é o número total de partições. Com o corte de vértices, há uma potencial replicação de vértices em um conjunto de máquinas, pois o vértice (representando cópia local) deve pertencer às arestas localizadas em várias partições, como observado no exemplo da Figura 15c, onde as cores vermelha, verde e azul representam a acomodação do vértice central em três máquinas, balanceando três arestas por máquina. Uma mudança em alguma aresta não necessita comunicação. Porém, alterações no vértice replicado envolve comunicação e atualização, mantendo a integridade. Visando essas replicações, considere o vértice **v** pertencente ao conjunto de máquinas **A(v)**, onde **A(v)** $\subseteq \{$**1, 2, 3, ..., p**$\}$. O tamanho de |**A(v)**| é o número de replicação do vértice **v** no sistema distribuído, onde $\sum_{v \in V} |A(v)|$ deve ser minimizado (JAIN *et al.*, 2013; GONZALEZ *et al.*, 2012). Para isso, o *framework* conta com um esquema em *grade* (Figura 15a) ou em *tórus* (Figura 15b).

Veja o exemplo em grade para uma aresta **e={u,v}**:

- Se **v** é mapeado por uma função *hash* para o espaço **5**, **v** pertence ao conjunto **{2,4,5,6,8}**, que é a representação **linha/coluna**;
- Se **u** é mapeado por uma função *hash* para **9**, **u** pertence ao conjunto **{3,6,7,8,9}**;
- Então, **e={u,v}** pertencerá a alguma máquina de intersecção **6** ou **8**.

Exemplo do esquema em tórus para uma aresta **e={u,v}**:

- Se **v** é mapeado pela função *hash* para **25**, **v** pertence ao conjunto **{1, 9, 17, 25, 26, 27, 28, 29}**;
- Se **u** é mapeado pela função *hash* para **8**, **u** pertence ao conjunto **{1, 2, 3, 4, 8, 16, 24, 32}**;
- Então, **e={u,v}** pertencerá à intersecção representativa da máquina **1**.

3.4.2 PowerGraph

PowerGraph introduz uma abstração capaz de suportar características do modelo *Pregel* e a consistência do GraphLab em sistemas distribuídos, visando uma nova forma de particionar e processar grafos grandes. As definições do PowerGraph são usadas para persistir e manter a consistência de subgrafos que são particionados através do corte de vértices, ao invés de arestas como é indicado em *Pregel*. Um vértice que recebe um corte possui *espelhos* somente leitura espalhados no *cluster*. O *vértice principal*, também chamado de *master*, fica na máquina que contém os dados. Por exemplo, o dado *rank* de um vértice em uma computação *PageRank*. Veja um esboço do processo na Figura 16. Trata-se de um modelo de computação e assincronismo denominado GAS, baseado nas seguintes operações:

- *reunir* (*gather* ou *pull*), que é o resultado de uma iteração *mapreduce*;
- *soma* (*sum*), uma operação realizada juntamente com *reunir*, visando agrupar mensagens acumuladas oriundas de vértices vizinhos[4] do *vértice principal*;
- *aplicar* (*apply*), que realiza a gravação de dados no *vértice principal*, e replica imediatamente nos *espelhos*;
- *espalhar* (*scatter*), que envia mensagens para outros vértices se necessário, para nova iteração.

Por exemplo, para o algoritmo do *SSSP* (*Single Source Shortest Path*), dado um vértice *v* e uma distância *dmin*, os passos são: *reunir* e *soma* fazem respectivamente a obtenção e acumulação das distâncias mínimas que foram sugeridas pelos vizinhos de entrada de *v*; *aplicar* verifica se há alguma distância sugerida que seja menor que *dmin*, e em caso afirmativo, *dmin* local e de todos os vértices *espelho* são imediatamente atualizados; *espalhar*, se houve mudança em *dmin*, efetua-se o envio de mensagens para vizinhos de saída de *v*, sugerindo uma opção melhor de distância.

3.5 Spark

O MapReduce foi projetado para aplicações com fluxo acíclico de dados, o que significa executar o *Job* através de uma rodada de mapeamento e redução. Ou seja, sem reusar, no mesmo *Job*, os dados produzidos pela redução. O *framework* Spark (ZAHARIA *et al.*, 2010) visa suportar esse reuso, importante para algoritmos iterativos, como aqueles envolvendo

[4] Remoto e local.

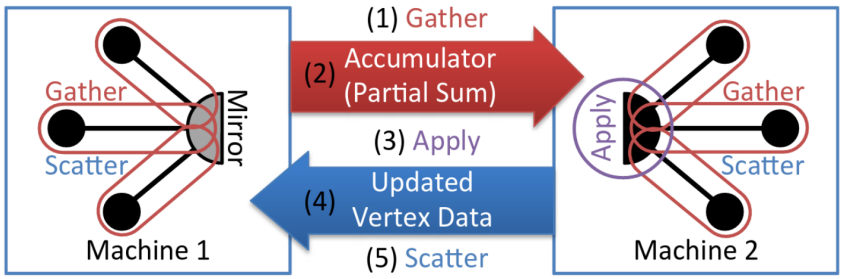


Figura 16 – GAS PowerGraph (GONZALEZ *et al.*, 2012)

grafos, aprendizagem de máquina, dentre outros. Além disso, oferece recursos para mineração de dados, com o auxílio de ferramentas, como as linguagens de programação R e Python. Foi majoritariamente desenvolvido na linguagem Scala, que incorpora recursos de linguagens orientadas a objetos e funcionais, sendo interoperável com Java.

O Spark busca equilíbrio entre expressividade, escalabilidade e confiabilidade. Para isso, envolve uma estrutura de alto nível de abstração para memória distribuída, através do conceito de conjuntos de dados distribuídos resilientes, ou *RDD* (*Resilient Distributed Datasets*). Essa solução explora o potencial da memória para manipular de forma eficiente os dados, permitindo que os processos façam reuso de dados diretamente na memória, em vez do sistema de arquivos. Juntamente com a abstração *RDD*, o *framework* estabelece um conjunto de operações[5] que serão usadas para manipulação dos dados, formando o modelo de programação do Spark.

Para construir uma aplicação, os desenvolvedores devem escrever uma parte *gestora*, denominada Driver. O Driver controla o fluxo da aplicação, lançando tarefas paralelas para um conjunto de nós processadores, que posteriormente devolverão ao *gestor* as informações de processamento. A Figura 17 apresenta o *gestor* e seus *trabalhadores*, os quais usam os RDDs através de blocos de dados em disco e cache de memória.

*RDD*s consistem em uma coleção de objetos somente leitura, particionados entre um conjunto de máquinas. Cada *RDD* é um objeto Scala que pode ser recuperado caso a partição que ele pertença falhe, usando linhagem de dados (BOSE; FREW, 2005), solução que possibilita analisar o ciclo de vida do objeto para recuperação. A ideia é o *RDD* possuir um identificador que é usado para reconstruí-lo a partir de informações armazenadas. Parte-se do princípio de que, no conjunto de objetos distribuídos, os objetos filhos possuem dependência com objetos pais, o que permite realizar um rastreamento até as informações armazenadas. Por exemplo, no código Spark da Tabela 2, as variáveis são instanciadas de forma dependente: **ones** depende de **cachedErrs**;

[5] Por exemplo: *ObjetoRDD.(reduce | save | collect | count), dentre outras.*

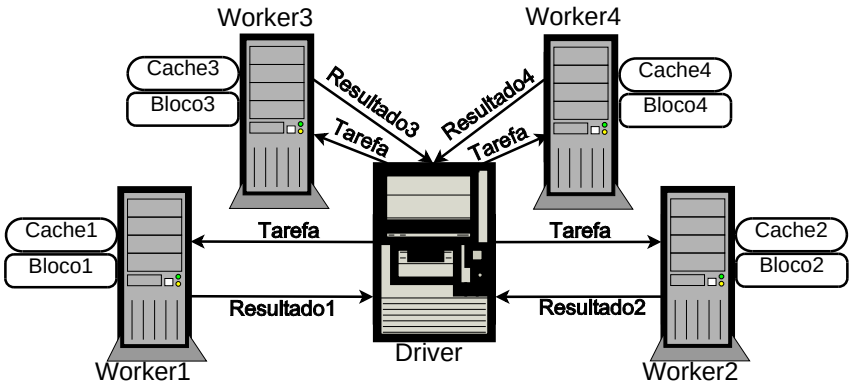


Figura 17 – Driver e `workers` no Spark.

cachedErrs depende de **errs**; **errs** depende de **file**. Portanto, há uma hierarquia entre RDDs pais e filhos. Nota-se ainda que esse código preza pelo aumento de desempenho quando o desenvolvedor explicitamente faz *cache* (*método cache do objeto*) do *RDD* na memória principal, o que acelera o reuso dos dados. Isso adquire (*pulling*) um conjunto de dados do *cluster*, sendo útil quando o acesso é repetitivo, como em algoritmos iterativos, como o *PageRank*.

Como uma chamada de método, operações são aplicadas diretamente no *RDD* e podem devolver informações para o gestor. As principais operações paralelas que retornam essas informações para o gestor são:

- *reduce*, que combina dados através de uma dada função e disponibiliza seus resultados; e
- *collect*, que é uma operação paralela para reunir os elementos de um conjunto de dados.

Além disso, há as operações que geram uma nova instância, construindo um novo *RDD*:

- *map*, que aplica uma determinada função sobre os dados, produzindo RDDs que podem ser usados por operações como *collect* e *reduce*;
- *cache*, que constrói um novo *RDD* na memória.

$$
\begin{array}{l}
var\ \textbf{file} = spark.textFile(\text{“}hdfs://...\text{”}) \\
var\ \textbf{errs} = file.filter(\_.contains(\text{“}ERROR\text{”})) \\
var\ \textbf{cachedErrs} = errs.cache() \\
var\ \textbf{ones} = cachedErrs.map(\_ => 1) \\
var\ \textbf{count} = ones.reduce(\_+\_)
\end{array}
$$

Tabela 2 – Exemplo de código Spark (ZAHARIA *et al.*, 2010).

No código da Tabela 2 é apresentado um exemplo de contagem de erros em um arquivo. Os passos dessa codificação são: a variável *file* está vinculada aos dados armazenados

no sistema de arquivos distribuído; a variável *errs* contém linhas do arquivo, geradas através da aplicação do filtro ("*ERROR*"); *cachedErrs* guarda *cache* em memória para futuro reuso; *ones* contém o mapeamento do número *1* às linhas filtradas; *count* é o resultado da soma dos *1's* efetuada pele redução.

3.5.1 GraphX

GraphX (XIN *et al.*, 2013) é a parte do Spark responsável por oferecer funcionalidades voltadas ao processamento de grafos em larga escala. Ela estende a abstração *RDD*, introduzindo o *RDG* (*Resilient Distributed Graph*), que associa registros de dados com vértices e arestas em grafo, e oferece significativa expressividade através de uma coleção de primitivas computacionais nesses *RDGs*. As principais funcionalidade incluem: visão de vértices e arestas (métodos `vertices` e `edges`); filtragem através dos métodos `filterVertices(`*pred*`)` e `filterEdges(`*pred*`)`, para construção de *subgrafos* que satisfazem o predicado *pred*; os métodos `mapVertices(`f`)` e `mapEdges(`f`)` utilizam a função f do usuário, processando e retornando novo grafo. É importante destacar que os dados são somente leitura, para garantir recuperação de falha, rastreando o objeto *pai*. Portanto, toda operação que venha transformar um grafo ou subgrafo retorna outro objeto *RDGs*. Parte da *interface* `Graph` do GraphX pode ser vista no Algoritmo 6, juntamente com um exemplo de cadastro e filtragem de dados para o grafo na Figura 18, detalhada no item *modelo e controle de dados* desta seção.

Algoritmo 6: Em Scala - parte da *interface* GRAPH e exemplo de filtro

```
1 Graph[V, E]
    def vertices(): RDD[(Id, V)]
    def edges(): RDD[(Id, Id, E)]
    def filterVertices(f: (Id, V)=>Bool): Graph[V, E]
    def filterEdges(f: Edge[V, E]=>Bool): Graph[V, E]
    def mapVertices(f: (Id,V)=>(Id,V2)): Graph[V2, E]
    def mapEdges(f: (Id, Id, E)=>(Id, Id, E2)): Graph[V, E2]
    ......
2 Exemplo de construção do grafo e filtragem de dados dos vértices
    val vertexArray = Array(
        (1L, ("Julia", 10)), (2L, ("Luiz", 20)), (3L, ("Pedro", 30)),
        (4L, ("Joel", 25)), (5L, ("Paula", 10)), (6L, ("Maria", 21))
    )
    val edgeArray = Array(
        Edge(1L, 2L, 3), Edge(1L, 5L, 2), Edge(2L, 5L, 8), Edge(4L, 3L, 2),
        Edge(5L, 3L, 7), Edge(5L, 4L, 10), Edge(6L, 1L, 1), Edge(6L, 5L, 1)
    )
    val vertexRDD: RDD[(Long, (String, Int))] = sc.parallelize(vertexArray)
    val edgeRDD: RDD[Edge[Int]] = sc.parallelize(edgeArray)
    val graph: Graph[(String, Int), Int] = Graph(vertexRDD, edgeRDD)
    graph.vertices.filter { case (id, (name, age)) => age > 20 }.collect.foreach {
        case (id, (name, age)) => println(s"$id, $name , $age")
    }
```

3.5.1.0.2 Particionamento

A partir dessas primitivas, os desenvolvedores do GraphX implementaram abstrações como PowerGraph GAS e Pregel, validando sua expressividade. O padrão de comunicação GAS aborda programação centrada em vértice e particionamento do grafo pelo corte de vértices, que resulta em vértices espelhados em uma ou mais unidades de processamento. Através da extensão de um particionamento bidimensional (ÇATALYÜREK *et al.*, 2010), a atribuição de arestas $i \longrightarrow j$ às máquinas do *cluster* de tamanho M segue a seguinte expressão: $h(i \longrightarrow j) = \sqrt{M} \times (h(i) mod \sqrt{M}) + (h(j) mod \sqrt{M})$, onde $h(i)$ é a aplicação de uma função *hash* nos *ids* dos vértices do grafo. Essa expressão garante que cada vértice se aloje em no máximo $2\sqrt{M}$ máquinas de um *cluster* de tamanho M (MALEWICZ *et al.*, 2010).

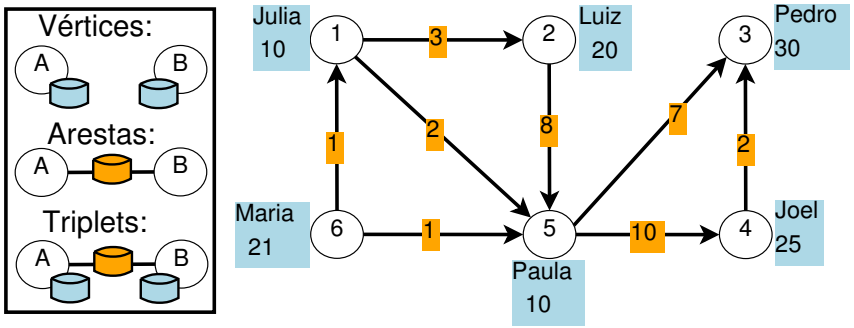


Figura 18 – Exemplo de grafo no Spark

3.5.1.0.3 Modelo e controle de dados

Grafos no GraphX podem ser entendidos de duas formas: *imutáveis*, quando somente acessíveis para leitura; e *mutáveis*, no sentido de que a aplicação de uma primitiva vai potencialmente gerar novo grafo ou subgrafo, dessa forma suportando alterações para processamento de novas computações. Na estruturação dos dados, há uma separação entre identificadores (*Ids*) e dados de vértices e arestas, como podemos verificar na Figura 18 e na implementação exemplo no Algoritmo 6. Esse é um grafo de rede social, contendo: o *id* do vértice (*Long*); o nome do vértice (*String*); a idade do vértice (*Int*); e um inteiro como dado da aresta, que representa "*curtidas*" ("*likes*") oriundos do vizinho. A persistência que envolve dados da aresta e dados dos seus vértices extremos é chamada de *Triplets*, contendo acesso através de objetos *RDG*. No exemplo de filtragem do Algoritmo 6, `vertexArray` e `edgeArray` são os objetos que respectivamente

contém vértice e arestas, sendo o parâmetro para instância dos objetos paralelos `vertexRDD` e `edgeRDD`, definido pelo *SparkContext* (objeto `sc`). De posse desses objetos, cria-se o objeto *graph*, que permite filtrar vértices com dados de idade acima de 20 anos. Nota-se que a tipagem de dados *[Long, (String, Int)]* equivale a *[Id, V]*, enquanto *Edge[Int]* equivale a *[Id, Id, E]*.

3.5.1.0.4 Utilização de Recursos

Um *cluster* que roda Spark com GraphX é entendível pela aplicação através de um objeto do tipo `SparkContext`. Esse objeto é usado pela aplicação e permite a paralelização de *RDD's*. De fato, são instruções sobre o *cluster*, que podem ser obtidas em arquivos XML compreensíveis pelo Spark, por linha de comando no momento de rodar a aplicação, ou ainda de forma padrão ou personalizada quando se usa o terminal de iteração Spark, no qual o usuário insere códigos Scala em tempo real. Dado esse objeto de contexto, recursos do *cluster* são consumidos por processos distribuídos denominados *executores*. Um *executor* explora a concorrência de processadores através de tarefas (*Tasks*), lançadas para trabalhar sobre os *núcleos* de processamento disponíveis. Dentre as várias formas de rodar uma aplicação, o usuário pode optar por parametrizar recursos e definir o `SparkContext` através de linha de comando. Por exemplo, os parâmetros "*–num-executors 6 –executor-cores 5 –executor-memory 14G*" definem que haverá 6 *executores*, cada um com 5 *núcleos* disponíveis e 14 *gigabyte* de memória. Isso diz que, no *cluster* como um todo, a aplicação trabalhará dividindo 84 *gigabyte* de memória e 30 *núcleos* de processamento. A literatura recomenda a maximização de *executores* ao invés de *núcleos* (GANELIN *et al.*, 2016). Um limite sugerido é que o número de *núcleos* não venha exceder cinco, pois muitos *núcleos* por *executor* potencialmente podem causar *sobrecarga* de *entrada/saída* no sistema de arquivos distribuído. Por exemplo, no Hadoop HDFS, geralmente usado juntamente com o Spark. Considerando a Figura 17, cada *executor* rodaria nos *workers* e seria controlado pelo *driver*, o qual também pode ser parametrizado através de linha de comando.

3.6 Outros *Frameworks* Inspirados em Pregel e MapReduce

Pregel e MapReduce inspiraram o surgimento de vários *frameworks*, os quais podem ser classificados pelos seguintes critérios: modelo de computação, ambiente de comunicação, controle de dados, plataforma e abstração de particionamento. Uma taxonomia baseada nesses critérios (MCCUNE *et al.*, 2015; DOEKEMEIJER; VARBANESCU, 2014) é discutida nas

próximas seções.

3.6.1 Modelo de Computação

O modelo de computação pode ser avaliado quanto a dois aspectos detalhados nas seções que se seguem: o *modelo de programação* e o *modelo de execução*.

3.6.1.1 Modelo de Programação

Os *frameworks* para processamento de dados em larga escala coordenam a parte computacional da aplicação através de um modelo bem definido de programação, que deve ser utilizado pelo desenvolvedor para sustentar sua lógica na aplicação. No caso de MapReduce, o modelo é de propósito geral, onde a implementação deve ser feita através da programação de uma função de *mapeamento*, vista como vértice *map*, e outra de *redução*, vista como vértice *reduce*, formando um grafo acíclico *map* $\rightarrow$ *reduce*. Entretanto, para o domínio de grafos e seus requisitos em grandes instâncias, o *framework* deve levar o desenvolvedor a pensar na programação a partir de características fundamentais abstraídas dos grafos. Essa foi a conclusão que chegaram *Grzegorz Malewicz et al* (MALEWICZ *et al.*, 2010), iniciando o modelo de programação *pense como um vértice* (TLAV[6]), ou *modelo centrado em vértices*. Trata-se de uma metodologia para programação onde o desenvolvedor programa uma ou mais funções que tem o vértice de um grafo como entrada. Normalmente essas funções são automaticamente paralelizadas pelo *framework*, assim como ocorre em relação às funções de mapeamento e redução do MapReduce.

O modelo TLAV pode ser naturalmente estendido. Por exemplo, pensar em vértice pode ser estendido para pensar em arestas, como ocorre com o *framework* X-Stream (ROY *et al.*, 2013), que é baseado no conceito de arestas, pois utiliza duas fases centradas em vértices (fases de computação para vértices fonte e destino). A metodologia *pensar em vértice* também pode ser estendida em *pensar em subgrafos*, como acontece com o *framework* GoFFish (SIMMHAN *et al.*, 2014), que é centrado em subgrafos, bem como com o *framework* Giraph++ (TIAN *et al.*, 2013), uma extensão centrada em subgrafos do *framework* Giraph (APACHE, 2011), o qual é baseado em vértices e implementa a especificação Pregel. A extensão ocorre também com matrizes, como visto no *framework* Presto (VENKATARAMAN *et al.*, 2013), que efetua operações com *matrizes* ou *arrays* distribuídos. Esse tipo de programação trabalha em conjunto

[6] *Think Like a Vertex*

com alguns interesses funcionais, descritos em seguida:

- **Fases de execução**: além da metodologia de programação centrada, *frameworks* têm sido classificados por uma ou mais fases de execução, como no caso do *mapeamento* e *redução*. Os *frameworks* baseados em Pregel utilizam apenas uma fase, que é a de executar a função *compute*, que processa o vértice. Já o *framework* Signal/Collect (STUTZ *et al.*, 2010) utiliza duas fases. Sua estratégia parte do princípio de que muitas computações com semântica *Web* (*página/link*) envolvem a passagem de informação entre recursos, vistos como dois vértices ligados. O vértice fonte envia um sinal através da aresta, efetuando a primeira fase. Em seguida, um coletor reúne os sinais de entrada nos vértices destinos, os quais vão executar alguma computação. Em uma proposta alternativa, PowerGraph (GONZALEZ *et al.*, 2012) fatorou o *framework* GraphLab nas três fases *scatter*, *apply*, *gather*, dando origem ao GAS.
- **Modo empurrar (*push*) versus modo puxar (*pull*)**: diz respeito às duas possíveis direções das informações, definindo o fluxo dos dados entre os vértices. Alguns *frameworks* utilizam o modo de fluxo *empurrar* (*push*), como Mizan (KHAYYAT *et al.*, 2013), Giraph, GPS (SALIHOGLU; WIDOM, 2013) e Signal/Collect. Nesse modo, o vértice atualiza sua informação e a envia aos seus vizinhos de saída. Por sua vez, no modo de fluxo *puxar* (*pull*), o vértice atualiza seu valor após ler informações dos seus vizinhos de entrada, destacando os *frameworks* Piccolo (POWER; LI, 2010), LFGraph (HOQUE; GUPTA, 2013), GraphLab, GoFFish2014. Dessa forma, trata-se da capacidade do *framework* em obter dados adjacentes. Com essa capacidade *puxar*, o vértice interessado na informação pode atuar de forma seletiva, considerando alguma suposição para obter ou não determinada informação, reduzindo comunicação de dados distribuídos. Finalmente, há também os *frameworks* que suportam ambos os modos, *empurrar* e *puxar*, como PowerSwitch, Kineograph (CHENG *et al.*, 2012), PowerGraph, Giraph++ (TIAN *et al.*, 2013) e Trinity (SHAO *et al.*, 2013);

3.6.1.2 Modelo de Execução

Definidos os aspectos da programação, os *frameworks* têm considerado um modelo de execução da computação com as seguintes características: *síncrona*, *assíncrona* ou *híbrida*:

- **Modo síncrono:** a execução síncrona é característica dos *frameworks* que implementam a especificação Pregel. Esse tipo de execução é mais previsível se comparado ao assín-

crono, pois há uma barreira bem definida entre computações realizadas pelos algoritmos. Entretanto, a sincronização possui um custo, que é a espera computacional nessa barreira. Aplicações iterativas devem considerar a maximização da computação entre as sincronizações (XIE *et al.*, 2015);

- **Modo assíncrono:** esse modelo de execução geralmente não possui um custo significativo de espera, sendo viável especialmente quando as computações são menores entre iterações (XIE *et al.*, 2015). Todavia, o sincronismo pode ser exigido em um dado momento. É o tipo mais comum em *frameworks* de memória compartilhada. Porém, alguns trabalhos têm sido desenvolvidos para modo híbrido em sistemas distribuídos;
- **Modo híbrido:** os *frameworks* Grace (WANG *et al.*, 2013), GraphHP (CHEN *et al.*, 2017) e PowerSwitch (XIE *et al.*, 2015) são híbridos, e buscam minimizar as barreiras do sincronismo global. Grace separa vértices que possuem apenas vizinhos locais (*partição local*) daqueles com vizinhos remotos (*entre partições remota*). Os vértices com vizinhos locais realizam em cada iteração *uma computação* e sua respectiva comunicação assíncrona. Quando há vizinhos remotos, ocorre o sincronismo, realizado no final da iteração. GraphHP segue o mesmo princípio. Porém, ao invés de realizar apenas *uma computação* por vértice na mesma iteração, os vértices locais podem realizar várias computações com operações assíncronas antes de um sincronismo global, o qual envolve vértices remotos. No modelo GAS do PowerGraph, quando um vértice é alterado, e possui espelho remoto, imediatamente cada espelho é atualizado, deixando a operação assíncrona, pois a computação prossegue com outros vértices na mesma iteração. O sincronismo é exigido apenas na fase *espalhar* (*scatter*), no final da iteração. PowerSwitch é baseado no modelo GAS do PowerGraph. Ele utiliza heurísticas para dinamicamente alternar entre sincronismo e assincronismo. Sua estratégia consiste em recolher estatísticas de execução em tempo real, aplicando suposições para prever o desempenho futuro e determinar quando um modo híbrido é rentável. O *framework* busca maximizar a computações antes da realização de um sincronismo.

3.6.2 Controle de Dados

Frameworks contemporâneos têm chamado a atenção pela forma como organizam seus dados para processamento, bem como para tráfego em rede. Para problemas iterativos em grafos, algumas plataformas de processamento, tais como Twister (EKANAYAKE *et al.*,

2010), iMapReduce (ZHANG *et al.*, 2012), I2MapReduce (ZHANG; CHEN, 2013), MR-MPI, consideram que a estrutura fundamental do grafo, com conexões estáticas entre vértices, não necessita ser trafegada em cada iteração. Isso significa que há uma separação entre **dados dinâmicos e estáticos**, onde apenas os dinâmicos são passíveis de serem transportados pela rede. Entretanto, existem problemas onde o grafo pode sofrer alterações em sua estrutura durante o processamento. Muitos *frameworks*, como GPS, Surfer (CHEN *et al.*, 2010), Giraph++, Giraph e Mizan consideram essa alteração estrutural no grafo. A **mutação do grafo** já era suportada naturalmente por alguns *frameworks* MapReduce, quando para cada iteração lançava-se uma nova tarefa *map-reduce*, reusando saídas da redução e a estrutura do grafo alterada.

Outra característica no controle de dados é o suporte ao atributo *out-of-core* de alguns *frameworks*, como MR-MPI, Hadoop, Giraph, HaLoop, GraphChi (KYROLA *et al.*, 2012), X-Stream, BPP (NAJEEBULLAH *et al.*, 2014) e Pregelix. Isso significa que há um limite no uso da memória principal. O que excede o limite deve ser gravado em disco. Alguns *frameworks*, como o Hadoop e Pregelix, contam com estruturas de armazenamento através de sistema de arquivos distribuído. Diferente disso, o MR-MPI faz *out-of-core* sem sistema de arquivos distribuídos. Porém, seus autores sugerem o uso do HDFS (SHVACHKO *et al.*, 2010).

3.6.3 Particionamento

Em um ambiente distribuído, a divisão eficiente do grafo entre unidades de processamento permite a minimização de comunicação e promove o equilíbrio da carga de trabalho. Entretanto, uma divisão ótima é um problema difícil (XU *et al.*, 2015; JAIN *et al.*, 2013; DOEKEMEIJER; VARBANESCU, 2014). Pregel e GraphLab têm usado uma forma básica de particionamento do grafo, através de uma função *hash*, semelhante ao MapReduce. Esse tipo de técnica apresenta um balanceamento de carga básico para processamento, pois divide uniformemente um conjunto de vértices entre máquinas: *hash(key) mod p*, onde *p* é a quantidade de partições. Entretanto, isso não releva a necessidade de minimizar a comunicação.

Usando os algoritmos de particionamento do Metis (KARYPIS; KUMAR, 1995), GPS adota um reparticionamento dinâmico em cada iteração. Para isso, tenta realocar vértices que fazem muitas trocas de mensagens. Usando um plano de migração, Mizan (KHAYYAT *et al.*, 2013) também utiliza reparticionamento dinâmico, buscando balanceamento de carga para computação e comunicação. Mizan é uma implementação Pregel (sem nó *gerente*) que monitora seus vértices em cada *superstep*, recolhendo informações sobre tempo de execução

e quantidade de mensagens de entrada/saída para vértices. Em cada *superstep*, há a barreira de sincronização Pregel e a consideração do plano de migração. Dessa forma, avalia-se a necessidade do reparticionamento, ou realocação de determinado vértice.

ExPregel (SAGHARICHIAN *et al.*, 2015) é uma extensão Pregel que faz o particionamento normalmente usando uma função *hash*. Porém, esse *framework* minimiza a comunicação entre vértices que estão em partições diferentes, pois utiliza uma distinção entre grupos locais e grupos remotos de vértices. Em cada *superstep*, o grupo local de vértices explora o problema até encontrar a melhor solução, para só então prosseguir para novo *superstep*. Ou seja, um vértice ativo pode ser executado várias vezes no atual *superstep*, antes da sincronização global.

Como discutido na Seção 3.4, o particionamento feito pelos *frameworks* pode ser abstraído através de corte de vértices (atribuir arestas às partições) ou por corte de arestas (atribuir vértices às partições). PowerGraph e GraphBuilder utilizam corte de vértices. Outros *frameworks* que seguem essa abordagem são X-Stream, Gbase (KANG *et al.*, 2012), PowerSwitch (XIE *et al.*, 2015) e GraphX (XIN *et al.*, 2013). Entretanto, a maioria dos *frameworks* observados neste trabalho realizam corte de arestas, muitos deles baseados no modelo Pregel, o que enfatiza o pioneirismo desse modelo. Suportando as duas abordagem, *vértice* e *aresta*, destacam-se as plataformas Naiad (MURRAY *et al.*, 2013) e Pegasus (KANG *et al.*, 2011).

3.7 Considerações Finais

A Tabela 3 apresenta os *frameworks* discutidos neste capítulo e suas características. A metodologia para identificação dessas características se baseia em uma revisão sistemática, após aplicação de comandos (*strings*) de buscas nos meios digitais *ACM, IEEE, Scopus, Springer, Science Direct*. Dentre os artigos lidos, destacam os *Surveys* (DOEKEMEIJER; VARBANESCU, 2014; MCCUNE *et al.*, 2015), que apresentam resultados compatíveis. A tabela possui a seguinte organização:

- Computação: divide-se em V/G/M, respectivamente referente ao modelo de programação com abstração em vértices, grafos/subgrafos, matrizes, bem como de propósito geral com (*PG*); pulL/pusH (L/H) referente à direção das informações; Fases, refere-se à quantidade de etapas adotadas para iniciar um novo ciclo de iteração;
- Execução: síncrona (S), assíncrona (A) e híbrida (H);
- Ambiente: distribuído (D) e centralizado (C);
- Out-of-core: envio de dados às unidades de armazenamento (Sim ou Não), especialmente

quando excedem a capacidade de memória;

- Partição: a partição dos dados é estática, dinâmica ou não se aplica (Não), devido ao uso de memória compartilhada pelo *framework*;
- Mutável: quando a plataforma permite mutação do grafo durante o processamento (Sim ou Não).

	Computação							
	V/G/M/PG	pulL/pusH	Fases	Execução	Ambiente	OutOfCore	Partição	Mutável
GUST**	G/V	L/H	1 ou 2	H	D	Não	Estática	Não
BPP	V	L	1	A	C	Sim	Não	sim
ExPregel	V	H	1	S	D	Sim	Estática	sim
Giraph	V	H	1	S	D	Sim	Estática	sim
Spark GraphX	G	L/H	2	H	D	Sim	Estática	sim
Giraph++	G/V	L/H	1	H	D	Não	Estática	sim
Giraphx	V	L	1	A	C	Não	Estática	sim
GoFFish	G	L	1	A	D	Não	Estática	não
GPS	V	H	1	S	D	Não	Dinâmica	sim
Grace	V	L	1	H	C	Não	Não	não
GraphChi	V	L	1	A	C	Sim	Não	sim
GraphHP	V	H	2	H	D	Não	Estática	não
GraphLab	V	L	3	A	C	Não	Não	não
GasCL	V	L	3	A	C	Não	Não	não
Kineograph	V	L/H	1	H	D	Não	Estática	não
LFGraph	V	L	1	S	D	Não	Estática	não
Mizan	V	H	1	S	D	Não	Dinâmica	sim
PowerGraph	V	L/H	3	H	D	Não	Estática	não
PowerSwitch	V	L/H	3	H	D	Não	Estática	não
Pregel	V	H	1	S	D	Não	Estática	sim
Pregelix	V	H	1	S	D	Sim	Estática	sim
Presto	M	H	1	S	D	Não	Dinâmica	não
Signal/collect	V	H	2	H	C	Não	Não	não
Trinity	V	L/H	1	H	D	Não	Estática	não
Twister*	PG	H	2	S	D	Não	Não	não
X-Stream	V	H	2	S	C	Sim	Não	sim
MR-MPI	PG	H	2	S	D	Sim	Estática	sim

*MapReduce iterativo usando *publish/subscribe*.
**Arcabouço de componentes (deste trabalho) que utiliza fluxo computacional (`Workflow`).

Tabela 3 – Tabela de características.

Dos modelos apresentados neste trabalho, destacam-se MapReduce, Pregel e GAS, a partir dos quais *frameworks* foram desenvolvidos e ofereceram algum tipo de contribuição relevante envolvendo computação em grafos de larga escala. Existem grupos de características que são compartilhadas nessas plataformas, como o tipo de programação, comunicação, particionamento, ambiente, uso de memória e estratégia de cada iteração organizada em fases. Cada característica oferece algum tipo de benefício para o algoritmo, mas eventualmente são excludentes. Por exemplo, o *framework* PowerGraph, que implementa o modelo GAS, permite definir se haverá ou não sincronismo na atualização de vértices. Sua versão híbrida, o PowerSwitch, faz experimentos com os algoritmos *PageRank, SSSP, Coloração de grafos e LBP-Loopy Belief Propagation*, chegando-se à conclusão de que o assincronismo nem sempre é rentável para

atingir melhor tempo de execução. PowerSwitch se apoia em heurísticas para decidir quando um modo síncrono é viável. Entretanto, ele ainda não suporta computação centrada em subgrafos, característica útil quando existe computação intensa de muitos vértices, resolvendo um problema local para posterior troca de mensagens e resolução do problema global. Suportando características do PowerGraph e Pregel, o Spark GraphX tem representado o estado-da-arte para processamento de grafos, especialmente no aspecto arquitetura, uma vez que em desempenho não superou seu precursor PowerGraph. Contudo, suporta subgrafos e tem oferecido aspectos técnicos importantes, como iteratividade com o usuário, através de um terminal de comandos em linguagem funcional Scala. Em um *cluster* que roda Spark, deve-se ajustar os XML's Spark com a configuração do *cluster*, ou parametrizar a execução de aplicações por linha de comando[7], para instanciar um objeto do tipo `SparkContext`, que permite a paralelização de *RDD*s nos termos da arquitetura do *cluster*. Um *cluster* Spark deve ter seus parâmetros rigorosamente configurados, especialmente para processar grandes grafos. Nesses problemas, apesar do sistema de arquivos distribuído, a memória é exigida, e seguem os limites das máquinas virtuais, que devem controlar muitos objetos.

Neste trabalho, introduz-se um arcabouço de componentes paralelos para suportar grafos. Nele, o componente controlador da estrutura de dados utiliza tipos primitivos ao invés de objetos, reduzindo a exigência de memória. Esse arcabouço contribui com o estado-da-arte em alguns aspectos, tais omo: utilização de arquitetura baseada em componentes paralelos; suporte a requisitos da HPC Shelf, uma nuvem de componentes destinada à execução de aplicações em um ou mais *clusters*, os quais são plataforma virtuais vinculadas ao contexto da aplicação. Nos próximos capítulos, introduzem-se os conceitos para atingir essas contribuições, iniciando com definições e atributos da nuvem de componente HPC Shelf.

[7] Tipicamente: `spark_submit binárioDaAplicação parâmetrosDeConfiguraçãoDoCluster`

4 HPC SHELF

HPC Shelf é um conceito de plataforma orientada a componentes para provimento de serviços de computação de alto desempenho, sob a abstração de nuvens computacionais. Através de uma *aplicação* da HPC Shelf, usuários ditos especialistas podem descrever problemas, possivelmente usando abstrações de alto nível, dentro de um determinado domínio de interesse, cuja solução demanda inerentemente por grande capacidade computacional, possivelmente extrapolando a capacidade oferecida por plataformas de computação paralela as quais o usuário tem acesso. Para solução desses problemas, a HPC Shelf oferece *sistemas de computação paralela* construídos através da composição e orquestração de componentes paralelos do modelo Hash, através da tecnologia de workflows científicos.

Sistemas de computação paralela executam sobre infraestruturas de computação paralela potencialmente oferecidas por diferentes centros de computação. Para isso, utilizam a abstração de *plataformas virtuais*, as quais representam plataformas de computação paralela instanciadas sobre tais infraestruturas, possivelmente usando mecanismos de virtualização já difundidos em nuvens computacionais. Plataformas virtuais são tratadas sob a visão de componentes da espécie plataforma, do modelo Hash. O conjunto de plataformas virtuais, disponibilizado através da HPC Shelf para atender as aplicações representa a infraestrutura computacional da HPC Shelf.

Como ilustrado na Figura 19, uma aplicação da HPC Shelf pode ser enxergada como uma nuvem to tipo SaaS (*Software-as-a-Service*), implementada sobre um *framework* de aplicações, conhecido SAFe (*Shelf Application Framework*), o qual, por sua vez, constitui uma nuvem do tipo PaaS (*Platform-as-a-Service*). Finalmente, a infraestrutura de computação sobre a qual as plataformas virtuais são instanciadas forma uma nuvem IaaS (*Infrastructure-as-a-Service*) sob a perpectiva do SAFe.

As seções a seguir apresentam maiores detalhes sobre a plataforma HPC Shelf. Porém, inicialmente, são apresentados alguns fundamentos necessários sobre as tecnologias de componentes de software aplicada no contexto de Computação de Alto Desempenho.

4.1 Computação de Alto Desempenho Baseada em Componentes

Um componente apresenta como característica o encapsulamento de funcionalidades de software, atuando como unidade independente que se relaciona com outros componentes

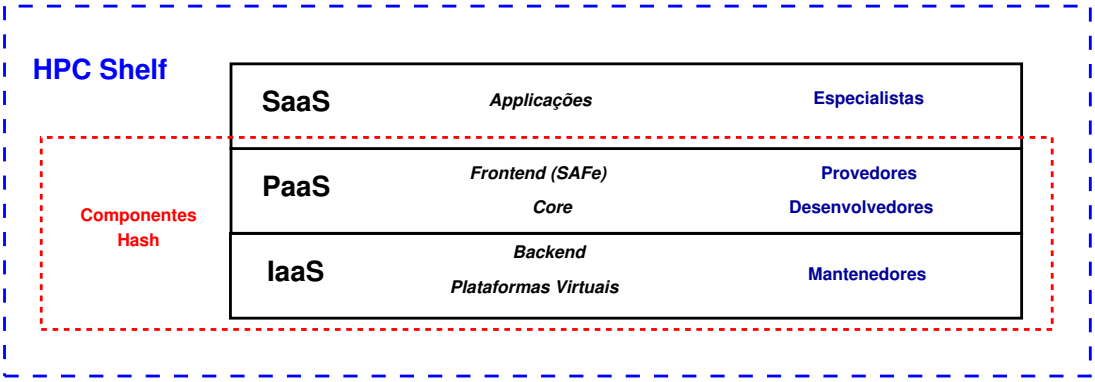


Figura 19 – HPC Shelf: Tipo de Nuvem Computacional

para compor sistemas maiores. Essa abordagem tem feito parte das técnicas modernas da engenhara de software, com significativa presença em aplicações corporativas. Para suportar requisitos de computação de alto desempenho, *modelos* e plataformas que suportam esse tipo de desenvolvimento têm surgido. Nesse contexto, apresenta-se na próxima seção como essas soluções têm surgido, bem como seus desafios que se estendem para HPC Shelf.

Explorar o potencial de desempenho de arquiteturas paralelas tem sido uma das principais motivações da área de Computação de Alto Desempenho (CAD), que fundamentalmente busca melhorar o desempenho computacional de aplicações de computação intensiva, em especial aquelas de interesse das ciências e das engenharias. Para essa finalidade, soluções recentemente propostas, consolidadas e disseminadas, como os *clusters* e as grades computacionais, tem melhorado a relação *custo/benefício* para potenciais usuários com interesse em CAD, no sentido de apoiar a execução das aplicações de processamento paralelo e distribuído.

Juntamente com a evolução dos *clusters* e grades computacionais, a complexidade do *software* nos domínios das ciências e das engenharias tem aumentado, bem como novos desafios tem surgido, como novas aplicações corporativas com requisitos de CAD, ou computação em nuvem. Isso tem exigido melhores práticas de desenvolvimento, com a finalidade de melhorar a produtividade e manutenção dessas aplicações (REZENDE, 2011; CARVALHO JUNIOR; REZENDE, 2011). Como resultado, gradualmente ocorre uma transição de metodologias de desenvolvimento, o que significa deixar as práticas apropriadas para programação de *software* de pequena escala (*programming-in-the-small*) e adotar novas técnicas para desenvolvimento de *software* de larga escala (*programming-in-the-large*).

Porém, na prática, ainda predominam nas aplicações CAD as técnicas de programação em pequena escala, diferente em relação ao que ocorre com domínios de aplicações

corporativas e de negócios. Isso porque enquanto esforços para aumentar a escala de desenvolvimento para aplicações CAD começaram somente nos anos 1990, já ocorria uma rápida proliferação de metodologias de desenvolvimento em larga escala nos domínios corporativos e de negócios desde o início dos anos 1970, com o objetivo de satisfazer a crescente complexidade do *software* nesses domínios, contexto que ficou conhecido como "Crise do Software" (DIJKSTRA, 1972), a partir do final dos anos 1960 (Figura 20). Essa "crise" motivou a produção de diversos artefatos de desenvolvimento, tais como linguagens e métodos formais (BACKUS, 1978), modelos e técnicas de programação, com a finalidade de aumentar a produtividade no desenvolvimento de *software*. Porém, o *software* de domínio das ciências e das engenharias esteve satisfeito por várias décadas com as técnicas de programação voltadas à pequena escala, característica que pode ser observada pelo grande legado de *software* desenvolvido e ainda em operação especialmente em linguagens como Fortran e C.

Com objetivos diferentes, o legado tecnológico posterior à *Crise do Software*, validado, testado e destinado ao desenvolvimento de software de larga escala, para aplicações corporativas e de negócios, não teve como foco principal os requisitos de desempenho necessários para aplicações de domínio CAD. Isso ocorre porque eram requisitos considerados pouco relevantes, salvo raras exceções. Considerava-se a existência de uma uniformidade nas plataformas de execução, não levando em conta os detalhes intrínsecos das arquiteturas. Para as aplicações nos domínios de interesse de CAD, esse contexto histórico aponta para a necessidade de novas técnicas de desenvolvimento que satisfaçam os requisitos de programação de software de larga escala nesse domínio, conclusão compartilhada por alguns grupos de pesquisa, levando ao surgimento do conceito de CBHPC (*Component-Based High Performance Computing*), em torno de iniciativas de consórcios de pesquisa que desenvolveram o *modelo* de componentes CCA (*Common Component Architecture*) (ARMSTRONG *et al.*, 2006), bem como Fractal/ProActive (BRUNETON *et al.*, 2002) e GCM (*Grid Component Model*) (BAUDE *et al.*, 2009).

Apesar disso, existiam ainda algumas dificuldades para serem superadas, especialmente porque esses *modelos* necessitam suportar noções gerais para construção de componentes paralelos, com conectores apropriados para promover sincronização paralela eficiente e de forma expressiva. Ou seja, permitindo o suporte a quaisquer padrões de paralelismo. Isso tem motivado novas pesquisas (CARVALHO JUNIOR *et al.*, 2007b), ocorrendo extensão de *modelos*. Uma justificativa para as dificuldades existentes está na herança dos paradigmas tradicionais de programação paralela, onde os processos são tratados como unidades primárias de composição do

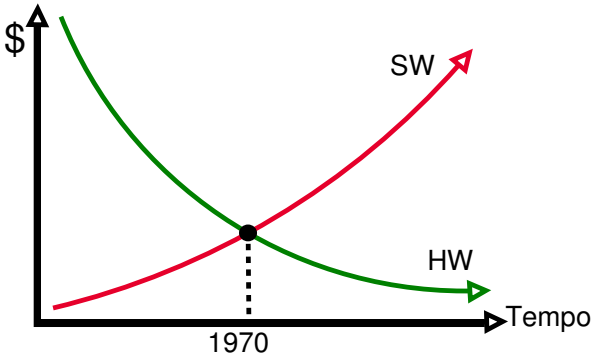


Figura 20 – Custo do software em alta

software, ao invés de serem baseados em interesses, como sugerido pelas técnicas modernas de engenharia de *software*. Consequentemente, ocorrem problemas de modularidade, dificultando o desenvolvimento de software com requisitos de processsamento paralelo, especialmente em plataformas de memória distribuída, fracamente acopladas. Como alternativa, uma outra solução tem sido proposta: o *modelo* de componentes Hash (CARVALHO JUNIOR; LINS, 2005). Ao invés de processos, componentes Hash têm a preocupação de considerar os interesses de *software*. Dessa forma, processos são tratados como unidades de decomposição transversais aos interesses, e são vistos de forma ortogonal. Como resultado, ocorre uma maior abrangência e suporte ao paralelismo em *modelos* de componentes paralelos. Na próxima seção, é apresentado o *modelo* de componentes paralelos Hash.

4.2 O Modelo Hash (Componentes Paralelos)

O modelo de componentes Hash tem suas motivações a partir da necessidade, discutida na Seção 4.1, por abordar a construção de aplicações de computação paralela a partir da composição de componentes, onde tais componentes são capazes de encapsular interesses de paralelismo, em particular quaisquer padrões de paralelismo que possam ser programados com bibliotecas de passagem de mensagens voltadas a plataformas de computação paralela de memória distribuída. Por esse motivo, são tratados, de forma genérica, como uma abstração que representa uma noção geral de *componente paralelo*, orientado a requisitos de CAD.

Para que seja dita compatível com o modelo Hash, uma plataforma de componentes paralelos precisa suportar as seguintes abstrações:

- unidades;
- composição hierárquica por sobreposição;

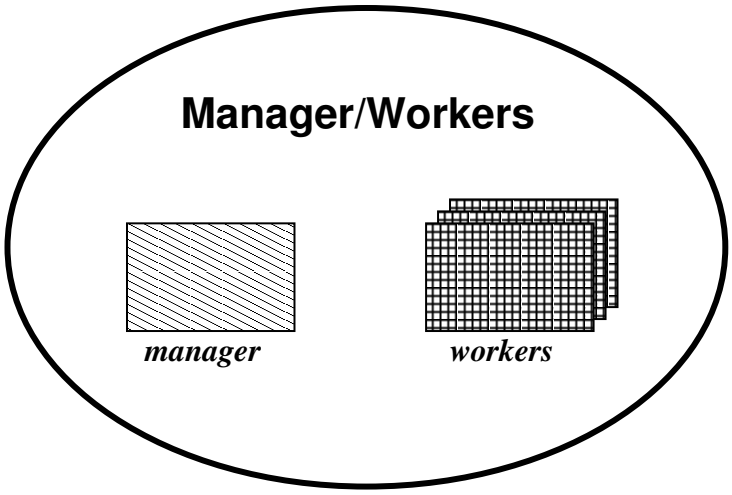


Figura 21 – Componente Hash para o Padrão Gerente/Trabalhadores (Unidades)

- espécies de componentes.

4.2.1 Unidades

Um componente Hash é constituído por um conjunto de *unidades*, as quais constituem os agentes de execução de tarefas associados a cada unidade de processamento de uma plataforma de computação paralela de memória distribuída. Uma unidade pode ser *singular* ou *paralela*.

Uma *unidade singular* representa um agente único, distinguido de outras unidades do componente paralelo, disposto em uma única unidade de processamento, enquanto uma *unidade paralela* representa um regimento de unidades idênticas engajadas na implementação do interesse do componente. De fato, unidades paralelas podem ser vistas como capazes de expressar computações paralelas do padrão SPMD (*Single Program Multiple Data*).

Por exemplo, como ilustrado na Figura 21, uma computação paralela implementada segundo o padrão *gerente/trabalhadores* (*manager/workers*), onde um processo *gerente* distribui um conjunto de tarefas para processamento entre um conjunto de processos ditos *trabalhadores*, pode ser representado por um componente Hash constituído de uma unidade singular, chamada ***manager***, e uma unidade paralela, chamada ***workers***, as quais cooperam entre si, dentro do escopo do componente, para realizar a computação, comunicando-se por meio de uma interface de troca de mensagens (e.g. MPI).

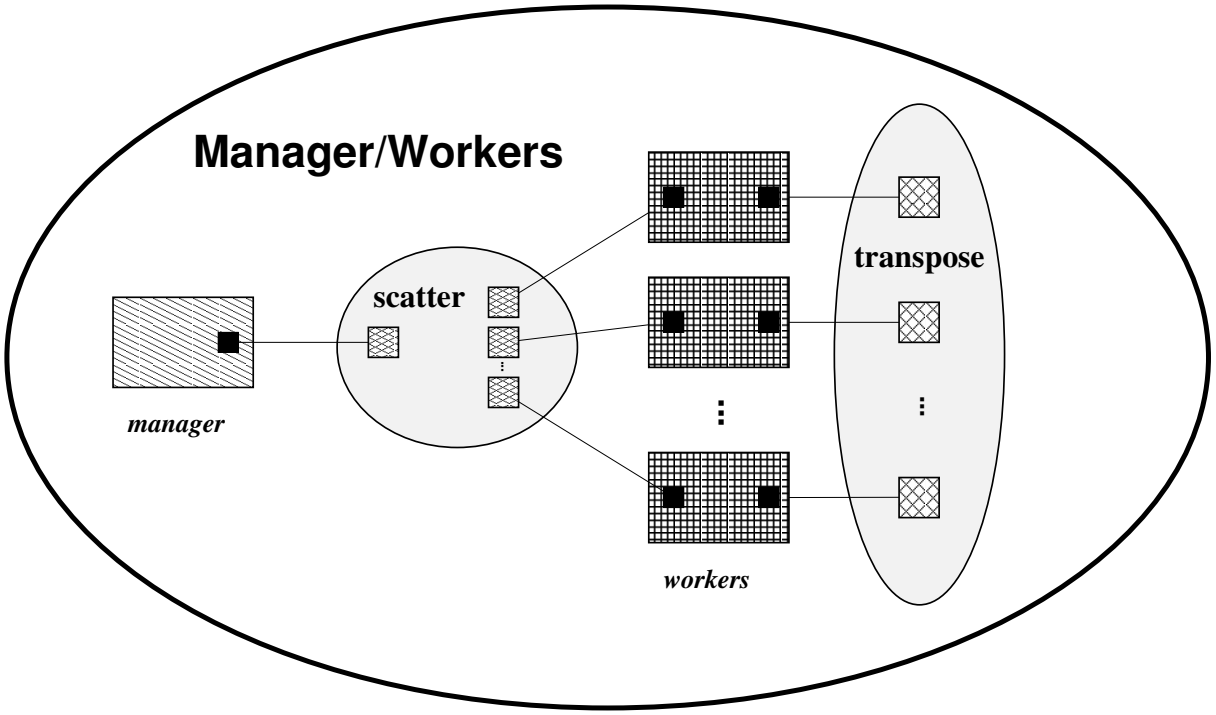


Figura 22 – Componente Hash para o Padrão Gerente/Trabalhadores (Sobreposição)

4.2.2 Sobreposição de Componentes

Sob uma perspectiva *bottom-up*, componentes Hash podem ser compostos hierarquicamente para definir novos componentes Hash. Por outro lado, vistos sob uma perspectiva *top-down*, componentes Hash, ditos *componentes hospedeiros*, precisam das funcionalidades oferecidas pelos componentes que fazem parte da sua composição, ditos *componentes aninhados*, para implementar o seu interesse.

A composição é especificada através de uma *função de sobreposição* que mapeia as unidades dos componentes aninhados às unidades do componente hospedeiro. Essa função deve ser total, de modo que todas as unidades dos componentes aninhados devem estar mapeadas a alguma unidade do componente hospedeiro. Além disso, cada unidade do componente hospedeiro é imagem de um conjunto de unidades de componentes aninhados distintos, as quais são chamadas de suas *fatias* (*slices*). Portanto, uma fatia de uma unidade *u* é definida como uma unidade de um componente aninhado mapeada a *u* através da função de sobreposição.

A Figura 22 ilustra a configuração do componente que o representa padrão *gerente/trabalhadores* introduzido na Figura 21. Nessa configuração, há dois componentes aninhados sendo compostos por sobreposição, os quais representam operações de comunicação coletiva utilizadas. No caso, o componente **scatter** é usado para distribuir dados do processo *gerente* para

os processos *trabalhadores*, enquanto o *componente aninhado* **transpose** representa uma troca total de dados entre os processos trabalhadores, onde cada processo possui um item de dados a ser enviado a cada um dos outros trabalhadores. Comparando com a *interface* de passagem de mensagens MPI, essas duas operações correspondem às suas subrotinas `MPI_Scatter` e `MPI_AlltoAll`, respectivamente.

4.2.3 Espécies de Componentes

Espécies de componentes agrupam componentes Hash que compartilham de um mesma representação concreta, isto é, como são definidos em termos de unidades de *software* usuais. Assim, componentes de uma mesma espécie diferenciam-se de componentes de outras espécies pelo seu *modelo de implantação*, *modelo de conexão* e *modelo de ciclo de vida*. Tendo em vista as diferenças entre componentes de espécies diferentes, que dificultam a sua interoperabilidade, com relação a esses quesitos, faz-se necessário separá-los em espécies distintas e tratá-los de maneira especial na plataforma. Assim, um sistema de *espécie de componentes* de uma certa plataforma de componentes compatível com o modelo Hash determina como os componentes evoluem, como são implantados, como são representados, quais são as regras de composição entre eles de acordo com suas espécies, e assim por diante.

O conceito de *espécie de componente* pode ser utilizado para diferentes finalidades, de acordo com os interesses e domínios de aplicação da plataforma de componentes. Por exemplo, um sistema de *espécies de componentes* poderia ser usado para definir uma plataforma de componentes capaz de permitir a composição de componentes de diferentes *modelos* e plataformas de componentes. Nesse caso, espécies de componentes fariam papel similar aos chamados *meta-componentes* suportados pelo *framework* SCIJump (PARASHAR *et al.*, 2009), do padrão CCA, ou seja, a *interoperabilidade* do *framework* com outras plataformas de componentes.

Outro uso possível é a definição de abstrações de composição, ou seja, *espécies de componentes* podem representar elementos de abstração dentro de um domínio de aplicação de interesse da plataforma de componentes, de maneira a tornar a composição de aplicações, por sobreposição de componentes, uma atividade mais simples para especialistas desse domínio.

Para o caso específico de computação paralela, *espécies de componentes* podem ser úteis para especificar diferentes tipos de conectores entre componentes, os quais representam padrões de paralelismo pré-determinados suportados pela plataforma. Isso remete a outra característica importante de plataformas de componentes do *modelo* Hash, que é a capacidade de

representar *conectores* como componentes, segregados em *espécies de componentes* distintas (CARVALHO JUNIOR *et al.*, 2007a).

Outros *modelos* de componentes costumam introduzir novas abstrações ao modelo de componentes para suportar certos conceitos não previstos por esse modelo. Por exemplo, podemos citar as *membranas*, *controladores* e *portas coletivas* de Fractal/GCM, bem como *ligações indiretas* (*indirect bindings*) e *regimentos* (*cohorts*) em *frameworks* CCA. No caso de plataformas compatíveis com o modelo Hash, tais abstrações poderiam ser introduzidas como espécies de componentes, sendo possível a evolução do conjunto de espécies.

4.2.4 Hash Programming Environment (HPE)

O HPE (*Hash Programming Environment*) é uma plataforma de referência do modelo de componentes Hash, tendo sido desenvolvida para apoiar o desenvolvimento orientado a componentes de programas paralelos para execução em plataformas de *cluster computing* (CARVALHO JUNIOR *et al.*, 2007; CARVALHO JUNIOR; REZENDE, 2013). Para isso, suporta as seguintes espécies de componentes: *plataformas*, *ambientes* (de suporte ao paralelismo), *estruturas de dados*, *computações*, *sincronizadores*, *qualificadores* e *aplicações*.

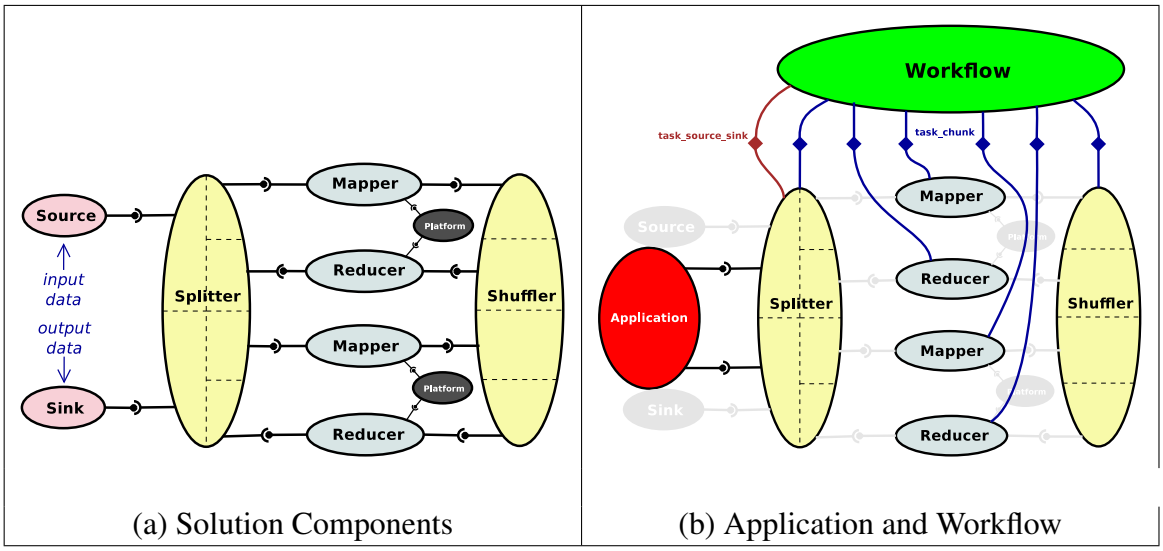
O HPE também pode ser visto como um *framework* CCA de propósito geral (CARVALHO JUNIOR; CORRÊA, 2010). De fato, introduz um conceito geral de componente paralelo para o modelo CCA, baseado no modelo Hash (CARVALHO JUNIOR; REZENDE, 2013).

Dentre as principais características introduzidas pelo HPE, destaca-se o HTS (*Hash Type System*), um sistema de tipos para componentes paralelos associado a um sistema de seleção dinâmica de componentes de acordo com os requisitos da aplicação hospedeira e as características da plataforma de computação paralela onde irá executar (*contexto*) (CARVALHO JUNIOR *et al.*, 2013). De fato, o HTS é o precussor do sistema de contratos contextuais da HPC Shelf, discutindo na Seção 4.7.

4.3 Espécies de Componentes da HPC Shelf

Por ser uma plataforma compatível com o modelo Hash, a HPC Shelf define um conjunto de espécies de componentes apropriadas para construção de sistemas de computação paralela de interesse de suas aplicações. São eles:

- Plataformas Virtuais: representam plataformas de computação paralela de memória distri-



(a) Solution Components (b) Application and Workflow

Figura 23 – Arquitetura de um Sistema de Computação Paralela MapReduce

buída e homogêneas, ou seja, constituídas de um conjunto de unidades de processamento de mesmas características;

- Computações: representam algoritmos de computação paralela capazes de explorar o desempenho de plataformas virtuais, ou seja, possivelmente tomando vantagem de suas características particulares, tais como a presença de aceleradores, múltiplos núcleos de processamento, hierarquias de memória, etc;
- Fontes de dados: representam grandes bases de dados, estruturadas ou não estruturadas, de interesse de componentes de computação;
- Conectores: conectam um conjunto de componentes de computação e fontes de dados, podendo atuar com diferentes finalidades, tais como a orquestração de componentes de computação, meio para coreografia entre esses componentes, acesso eficiente a dados, etc;
- *Bindings* de Serviços: Conectam componentes das espécies anteriores entre si através de interfaces de serviços, no modelo cliente-servidor, geralmente associados a interesses não funcionais;
- *Bindings* de Ações: Conectam componentes das espécies anteriores entre si por meio de interfaces par-a-par, através das quais sincronizam ações de computação, governando o fluxo de execução dos sistemas de computação paralela (interesses funcionais).

A Figura 23(a) apresenta o esquema arquitetural de um sistema de computação paralela destinado a executar uma computação MapReduce, composto de duas fontes de dados, de entrada e de saída, chamadas **Source** e **Sink**; quatro componentes de computação, sendo dois

empregados para executar operações de mapeamento (**Mapper**) e outros dois responsáveis por operações de redução (**Reducer**); dois conectores, chamados **Splitter** e **Shuffler**; e duas plataformas virtuais (anônimas na figura), de modo que cada uma hospedará um par de componentes, um de mapeamento e outro de redução. Ligando esses componentes, há um conjunto de *bindings* de serviços.

Na Figura 23(b), são incluídos dois componentes especiais que devem aparecer em sistemas de computação paralela, chamados **Application** e **Workflow**. O primeiro serve para comunicação da aplicação, executando sobre o SAFe, com os demais componentes, ditos *componentes de solução*, enquanto o último executará a orquestração do sistema de computação paralela, através do conjunto de *bindings* de ações que o liga aos componentes de computação.

Conectores são constituídos de um conjunto de *facetas*, instanciadas, cada qual, sobre as plataformas virtuais onde encontram-se os componentes de computação e fontes de dados que conectam. Na Figura 23, as facetas dos componentes **Shuffler** e **Splitter** são delimitadas pelas linhas tracejadas. Cada *faceta* instancia um certo subconjunto de unidades do conector. Dessa forma, os *bindings* de serviços e de ações entre computações/fontes de dados e conectores são diretos, não exigindo comunicação entre plataformas virtuais dispostas em domínios diferentes. Isso torna possível que conectores encapsulem formas mais eficientes de sincronização e comunicação entre computações e fontes de dados geograficamente dispersos, evitando tais preocupações na tarefa de desenvolvimento desses componentes.

Bindings, por também serem distribuídos entre as plataformas virtuais onde encontram-se os componentes que conectam, podem ser vistos como formas especiais de conectores, muito embora apresentando características distintas. Para o suporte a *bindings*, os componentes conectados através deles devem possuir portas compatíveis. No caso de *bindings* de serviços, há um componente dito provedor, o qual deve possuir uma porta provedora, e um componente dito usuário, o qual deve possuir uma porta usuária. Se um par de portas, usuária e provedora, possuem o mesmo serviço (interface de operações) podem ser conectadas, formando um *binding* de serviço. Dessa forma, chamadas realizadas pelo componente usuário através de sua porta usuária são atendidas pelo componente provedor através de sua porta provedora, de forma que o componente *binding* é responsável por intermediar a comunicação, seja ela direta ou indireta. Por sua vez, no caso de *bindings* de ações, um conjunto de componentes parceiros possuem portas que exportam o mesmo conjunto de nomes de ações podem ligá-las a fim de formar um *binding* de ações. Diz-se que uma ação é ativada quando todos os componentes parceiros

ativam um mesmo nome de ação na porta de ações. A ativação de uma ação em um determinado componente parceiro pode disparar uma computação ou comunicação dentro dele. Dessa forma, componentes podem sincronizar operações de computação, definindo um *workflow*.

É importante salientar que tais *bindings* são do tipo $M \times N$ (paralelos) (JACOB *et al.*, 2005; ZHANG; PARASHAR, 2006; DAMEVSKI *et al.*, 2007), como consequência natural do próprio caráter paralelo dos componentes Hash. Tanto do lado provedor quando do lado usuário de um *binding* de serviço, assim como nos diversos lados parceiros de um *binding* de ações, pode haver um conjunto de unidades paralelas, as quais realizam invocações de serviços, atendimentos a serviços e ativação de ações de forma paralela e cooperativa. Porém, uma vez que os *bindings* encapsulam a lógica de troca de mensagens entre os lados envolvidos na comunicação, formas otimizadas de comunicação, no caso de *bindings* indiretos, podem ser aplicadas em casos mais simples, quando, por exemplo, há um provedor sequencial (única unidade), um usuário sequencial, apenas um par de componentes parceiros, um dos quais sequencial, etc.

4.4 Intervenientes da HPC Shelf

Uma plataforma HPC Shelf orquestra a atuação de um conjunto de *atores*, ou *intervenientes*, que a veem sob diferentes perspectivas. Por exemplo, como ilustrado na Figura 19, para a perspectiva dos usuários *especialistas*, a *nuvem* oferece *software* como serviço (SaaS), enquanto para *provedores* e *desenvolvedores* consiste em plataforma como serviço (PaaS). Já para o *mantenedor*, a visão é de infraestrutura como serviço (IaaS).

4.4.1 Especialistas

Especialistas de domínio são os *atores* que fazem uso das funcionalidades de aplicações da HPC Shelf para resolver problemas de seu interesse. Esses *atores* são especialistas em um determinado domínio, notadamente nas áreas de ciências e engenharias. Espera-se que aplicações disponibilizem para usuários *especialistas* uma *interface* de alto nível, com abstrações adequadas que facilitem a especificação de problemas, ao modo, por exemplo, de *linguagens específicas de domínio* (*DSLs - Domain Specific Languages*). Desse modo, especialistas abstraem-se totalmente das infraestruturas computacionais sobre as quais os problemas que descreve são solucionados, bem como da própria natureza orientada a componentes dessas soluções.

Porém, a flexibilidade oferecida pela plataforma de componentes da HPC Shelf torna

possível que outros usos se façam de suas aplicações. Por exemplo, uma aplicação poderia oferecer uma interface para artefatos de propósito geral para desenvolvimento de programas paralelos, voltados a especialistas em programação paralela, como é realizado atualmente no HPE (CARVALHO JUNIOR; REZENDE, 2013).

4.4.2 Provedores

Provedores de aplicações são os atores responsáveis pela construção e disponibilização de aplicações para os usuários *especialistas*. Provedores possuem competência na composição de soluções computacionais para problemas dentro do domínio de interesse de uma determinada classe de especialistas, a partir de componentes que implementam funcionalidades necessárias. A abstração de componentes do modelo Hash torna possível que não seja necessário que provedores sejam especialistas em computação paralela. De fato, do ponto de vista do provedor, sistemas de computação paralela são enxergados como *workflows* científicos baseados em componentes, onde as preocupações de paralelismo encontram-se encapsuladas nos componentes. Entretanto, a flexibilidade da plataforma ainda permite que interesses de paralelismo sejam expostos aos provedores em contratos contextuais dos componentes usados na composição, incluindo plataformas virtuais, quando necessário e os provedores possuam competência para tal.

4.4.3 Desenvolvedores

Desenvolvedores de componentes são os atores responsáveis pela construção de componentes paralelos que representam software especializado para fazer uso das características, possivelmente particulares, de plataformas de computação paralela oferecidas pela HPC Shelf, chamadas de *plataformas virtuais* e também tratadas como componentes. São esses componentes que serão usados para formar novos componentes, e poderão ser consumidos pelos *provedores de aplicações* para montar os sistemas de computação paralela que atendem às necessidades de aplicações. Para isso, supõe-se que os *desenvolvedores* são especialistas em programação paralela e plataformas de computação paralela, pois necessitarão disso para otimizar cada componente segundo as características arquiteturais do perfil de plataforma virtual sobre a qual o componente executará.

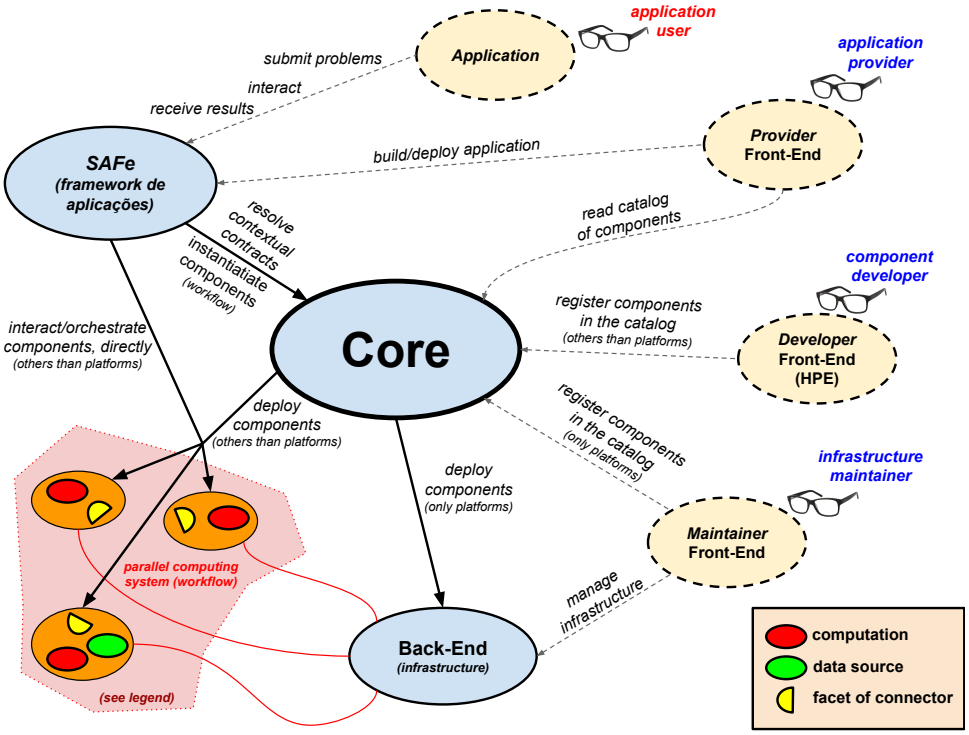


Figura 24 – Arquitetura da HPC Shelf

4.4.4 *Mantenedores*

Mantenedores de plataformas são os *atores* mais próximos das plataformas de computação paralela, ou seja, da infraestrutura de computação paralela da HPC Shelf. São responsáveis pela criação e manutenção dos componentes que representam *plataformas virtuais*, que representam plataformas de computação paralela instanciadas sobre a infraestrutura física dentro do domínio de sua responsabilidade, possivelmente utilizando mecanismos de virtualização. A representação de uma *plataforma virtual* na HPC Shelf se dará através de um componente da *espécie plataforma*, o qual deve encapsular informações para instanciá-la sobre a infraestrutura do *mantenedor*. Como já foi dito, essas instâncias podem ser suportadas por meio de alguma tecnologia de virtualização, capaz de prover infraestrutura como serviço (IaaS), embora seja também possível o uso de plataformas de computação paralela sem virtualização. Nesse último caso, torna-se inviável explorar funcionalidades de elasticidade de computação paralela.

4.5 Arquitetura da HPC Shelf

Os elementos arquiteturais da HPC Shelf, descritos nas seções que se seguem, podem ser vistos na Figura 24. São eles: Frontend, Core e Backend.

4.5.1 Frontend (SAFe)

O SAFe (*Shelf Application Framework*) é o *framework* utilizado para construção de aplicações da HPC Shelf. Portanto, é o artefato através do qual provedores acessam os seus serviços. Para isso, o SAFe permite o desenvolvimento de soluções computacionais, na forma de *workflows científicos* baseados em componentes, para os problemas descritos pelo especialista através da interface de alto nível da aplicação, os que também são chamados de *sistemas de computação paralelos*.

O SAFe é implementado por um conjunto de *classes* e *interfaces* Java, bem como um conjunto de diretrizes para construção das aplicações. Por exemplo, uma aplicação deve ser obtida por herança a partir da classe *Application*, que implementa os serviços básicos para a comunicação com o ambiente HPC Shelf. Essa classe requer a implementação do método *setServices*, o qual recebe como argumento um objeto que deve implementar a *interface Services*. Com o serviço recebido, a aplicação pode se conectar às *portas* dos componentes que serão orquestrados para computar a solução de problemas através de um *workflow* descrito com a linguagem SAFeSWL (*SAFe Scientific Workflow Language*). A linguagem SAFeSWL está descrita na Seção 4.6.

4.5.2 Core

O Core é o elemento arquitetural central da plataforma HPC Shelf, visto que por seu intermédio é controlado o ciclo de vida dos seus componentes. Tais componentes são expostos através do chamado *catálogo de componentes*, o serviço através do qual o Core expõe ao SAFe os componentes que podem ser usados pelos provedores para composição de sistemas de computação paralela.

Uma das principais características do Core é o suporte a um *sistema de contratos contextuais* para abstração de componentes em relação a suposições que determinam como são implementados, incluindo linguagens de programação, algoritmos e estruturas de dados, bem como a capacidade de tomar vantagem de características particulares de uma determinada arquitetura de plataforma de computação paralela. Para isso, utiliza a abstração de *contexto*, determinado pelos requisitos da aplicação e características da plataforma virtual no contexto do qual um componente deverá executar. Para um determinado contexto, um procedimento de *resolução* de contratos contextuais é capaz de encontrar o componente mais apropriado.

Contratos contextuais são detalhados, inclusive com apresentação de exemplos, na Seção 4.7. Outros exemplos de contratos contextuais são apresentados no Capítulo 5, referentes a estudos de caso deste trabalho.

Os serviços do Core são utilizados por *provedores*, *desenvolvedores* e *mantenedores*, os quais têm interesse em acessar o catálogo de componentes. No caso das aplicações, é através dos serviços do Core que é possível ao SAFe orquestrar a resolução, implementação e instanciação dos componentes necessários a um sistema de computação paralela, em seus *workflows*.

4.5.3 Backend

O Backend é o elemento responsável pelo gerenciamento da infraestrutura computacional da HPC Shelf. Através dos serviços de um Backend, o Core pode instanciar plataformas virtuais sobre a infraestrutura de um mantenedor. De fato, cada mantenedor pode suportar a sua própria implementação de um serviço Backend apropriado para instanciar plataformas virtuais sobre a sua infraestrutura física de computadores paralelos. A composição de tais serviços oferecidos pelo conjunto de mantenedores é o que de fato define o Backend da HPC Shelf.

Porém, do ponto de vista do SAFe, recomenda-se abstrair-se da identidade dos mantenedores das plataformas virtuais utilizadas em um sistema de computação paralela, o que é possível graças ao sistema de contratos contextuais. Em relação a componentes da espécie plataforma, contratos contextuais são chamados de *perfis* de plataformas virtuais, descrevendo características contextuais que guiarão a escolha de um mantenedor apropriado para instanciá-la.

4.6 SAFeSWL (*SAFe Scientific Workflow Language*)

SAFeSWL é uma linguagem de *workflows* composta por dois subconjuntos, sendo um para descrição arquitetural e outro para orquestração de *workflows* científicos (SILVA; CARVALHO JUNIOR, 2016). Seu objetivo é descrever sistemas de computação paralelos, potencialmente de larga escala e heterogêneos, implantados na HPC Shelf. Contudo, embora tenha sido originalmente projetado para as necessidades da HPC Shelf, suas ideias e conceitos podem ser adotados por outros SWfMS (*Scientific Workflow Management Systems*) com semelhantes requisitos. Nas próximas seções, apresentam-se detalhes dos dois subconjuntos que constituem a linguagem SAFeSWL.

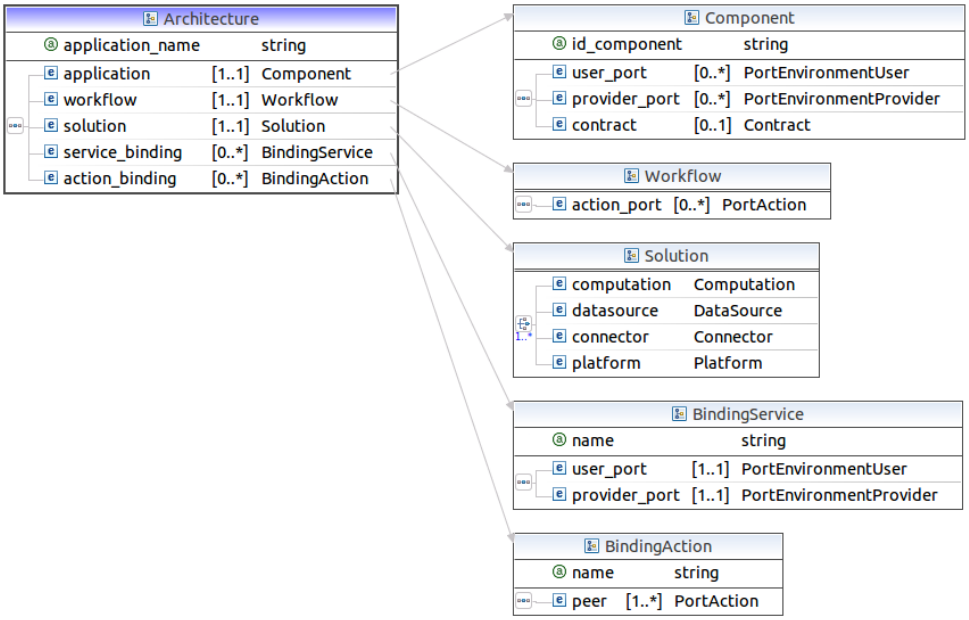


Figura 25 – A gramática *XSD* para a linguagem arquitetural

4.6.0.1 Subconjunto Arquitetural

Através do subconjunto de linguagem arquitetural de SAFeSWL, os provedores podem especificar os componentes e *bindings* que formam a arquitetura de um sistema de computação paralela, como aquele apresentado na Figura 23. Cada componente de solução possui um contrato contextual a ele anexado, que orientará a seleção de uma implementação apropriada, de acordo com os requisitos especificados.

Os elementos do arquivo arquitetural são baseados em XML, sendo bastante intuitivos. O arquivo é organizado em seções, delimitadas pelos seguintes elementos em forma de *tags*:

- A *tag* `<application>` é usada para definir o componente `Application`, declarando suas portas usuárias e provedoras;
- A *tag* `<workflow>` é usada para definir o componente `Workflow`, declarando suas portas de ações;
- A *tag* `<solution>` é usada para definir o corpo do sistema de computação paralela, onde os componentes de solução são especificados por meio das *tags* `<computation>`, `<connector>`, `<repository>`, e `<platform>`, sendo uma para cada espécie de componentes da HPC Shelf;
- A *tag* `<env_binding>` é usada para declarar um *binding* de serviço, que conecta portas usuárias às portas provedoras;
- A *tag* `<task_binding>` é usada para declarar um *binding* de ações, que sincroniza um

conjunto de portas de ações, contendo um mesmo conjunto de nomes de ações.

A gramática XSD (*XML Scheme Definition*) foi especificada para documentar a sintaxe do arquivo arquitetural, bem como para facilitar a construção de um analisador sintático. A Figura 25 descreve sua estrutura geral, através de um diagrama fornecido por um *plugin* da plataforma Eclipse, usado para manipular arquivos XML. Nessa Figura, pode-se ver o tipo do elemento de nível superior, chamado *Architecture*, representado pela *tag* `<architecture>`, bem como os tipos de seu elementos internos: o componente de aplicação (`Component`), o componente *workflow* (`Workflow`), os componentes de solução (`Solution`), os *bindings* de serviços (`BindingEnvironment`) e os *bindings* de ações (`BindingTask`).

```
<application id_component="application">
   <user_port id_port="state-L"/>
   <user_port id_port="state-R"/>
   <provider_port id_port="event-L"/>
   <provider_port id_port="event-R"/>
</application>
```

Código 1 – Exemplo de XML Arquitetural: O Componente Application

```
<service_binding>
   <user_port     id_component="application" id_port="state-L"/>
   <provider_port id_component="component-L" id_port="state"/>
</service_binding>
```

Código 2 – Exemplo de XML Arquitetural: Um Binding de Serviço

O Código 1 contém a declaração de um componente `Application` na sintaxe arquitetural do SAFeSWL. Basicamente, apresentam-se as portas usuárias e provedoras de um sistema de computação paralela hipotético, cuja arquitetura é descrita na Figura 26. Por exemplo, `<user_port id_port = ‘‘state-L’’>` declara que uma porta usuária está conectada, através da declaração `<service_binding>` mostrada no Código 2, à porta provedora de um componente `component-L`. Uma vez que é uma porta de serviço, `state-L` lida com um interesse não funcional. Por exemplo, no sistema de computação paralela visto na Figura 26, `application` monitora o estado de `component-L` e `component-R` durante a execução, através das respectivas portas usuárias `state-L` e `state-R`. Além disso, `application` é notificado sobre eventos gerados por esses componentes através das portas provedoras `event-L` e `event-R`, respectivamente.

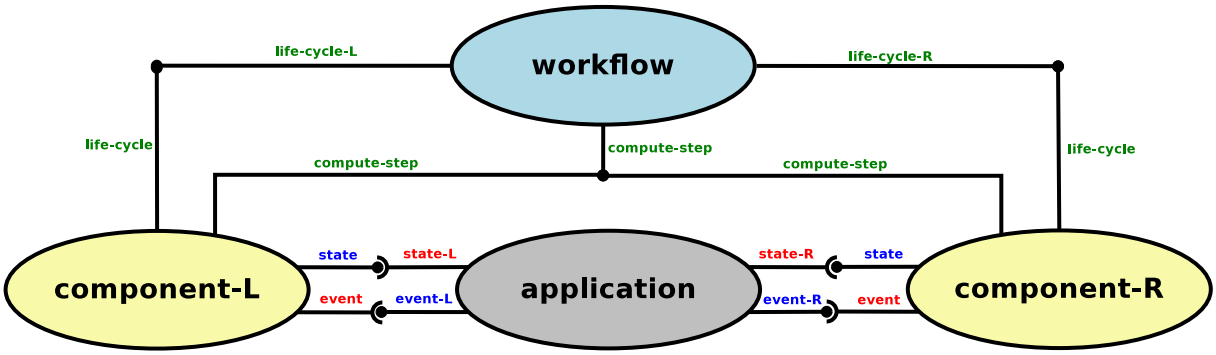


Figura 26 – Application

```
<action_binding>
   <peer id_component="workflow"     id_port="compute-step"/>
   <peer id_component="component-L" id_port="compute-step"/>
   <peer id_component="component-R" id_port="compute-step"/>
</action_binding>
```

Código 3 – Exemplo de XML Arquitetural: Um Binding de Ações

O Código 3 exemplifica a declaração de um *binding* de ação envolvendo três portas de ações, denominadas compute-step, dos componentes `workflow`, `component-L` e `component-R`. A porta compute-step tem quatro nomes de ação: `INITIALIZE`, `COMPUTE_STEP`, `TERMINATE_FLAG` e `FINALIZE`, como ilustrado no Código 4, onde o componente `workflow` é declarado.

```
<workflow id_component="workflow">
   <action_port id_port="compute-step">
      <action name="INITIALIZE"/>
      <action name="COMPUTE_STEP"/>
      <action name="TERMINATE_FLAG"/>
      <action name="FINALIZE"/>
   </action_port>
</workflow>
```

Código 4 – Exemplo de XML Arquitetural: O Componente Workflow

As portas de ações lidam com interesses funcionais, desencadeando as operações computacionais que definem o progresso da execução do *workflow*. Dessa forma, no exemplo, os nomes de ações de `component-step` representam operações funcionais executadas pelos

TASK ::= **Skip** | **Perform** *action* | **Start** *handle action* | **Wait** *handle* | **Cancel** *handle* |
Sequence TASK+ | **Parallel** TASK+ | **Repeat** TASK | **Break** | **Continue** |
Select {*component action*: TASK}+

Skip representa uma ação vazia. **Perform** representa uma invocação de ação síncrona, que se completará após todos os parceiros de componente ativar a mesma ação, através de um *binding* de ação. Por sua vez, **Start**, **Wait** e **Cancel** são usados para invocação de ações assíncronas, onde uma invocação de ação é iniciada por **Start**. Em seguida, a tarefa de orquestração pode aguardar a conclusão da ação, executando **Wait** para o mesmo *handle*, ou cancelá-la, por meio da execução de **Cancel**. **Sequence** e **Parallel** representam respectivamente a invocação serial e paralela de um conjunto de tarefas. **Repeat** representa a execução iterativa de uma tarefa até que **Break** seja executado. **Continue** força o início de uma nova iteração. Finalmente, **Select** representa uma escolha entre um conjunto de tarefas, de acordo com a invocação de um conjunto de ações em sentinela.

Figura 27 – Sintaxe Abstrata do Subconjunto de Orquestração de SAFeSWL

componentes `component-L` e `component-R`, em uma ordem prescrita. O componente `workflow` é responsável por orquestrar essas ações de acordo com os requisitos da aplicação.

O arquivo de descrição arquitetural é processado pelo SAFe para declarar automaticamente as portas de componentes, bem como para conectá-las através de *bindings*, preparando o sistema de computação paralela para a execução do *workflow*, mesmo antes dos componentes de solução terem sido implantados e instanciados.

4.6.0.2 Subconjunto de Orquestração

O interveniente provedor gera o código de orquestração do seu *workflow* usando o subconjunto de orquestração de SAFeSWL, cuja sintaxe abstrata é apresentada na Figura 27. Isso é interpretado pelo componente Workflow, que é conectado às portas de ações das computações e dos conectores através de *bindings* de ações, para impulsionar o progresso computacional do sistema de computação paralela.

Quando há ativações pendentes de uma determinada ação comum a todos os componentes que são parceiros através de um *binding* de ação, cada componente de computação ou faceta de conector, escritos atualmente na linguagem C#, pode reagir executando uma computação associada. Para isso, o componente comunica-se com objetos que implementam a interface `ITaskPort`. Como pode ser visto no Código 5, a interface `ITaskPort` contém algumas versões síncronas e assíncronas para um método chamado `invoke`, destinado a invocar ações nas portas do componente. Por sua vez, ao invés de uma linguagem de programação completa, o componente Workflow usa o conjunto de primitivas de ação e combinadores do subconjunto de orquestração de SAFeSWL. Isso ocorre porque ele só está interessado em controlar o fluxo de ativação das ações, e não em reagir a elas executando computações, como acontece com

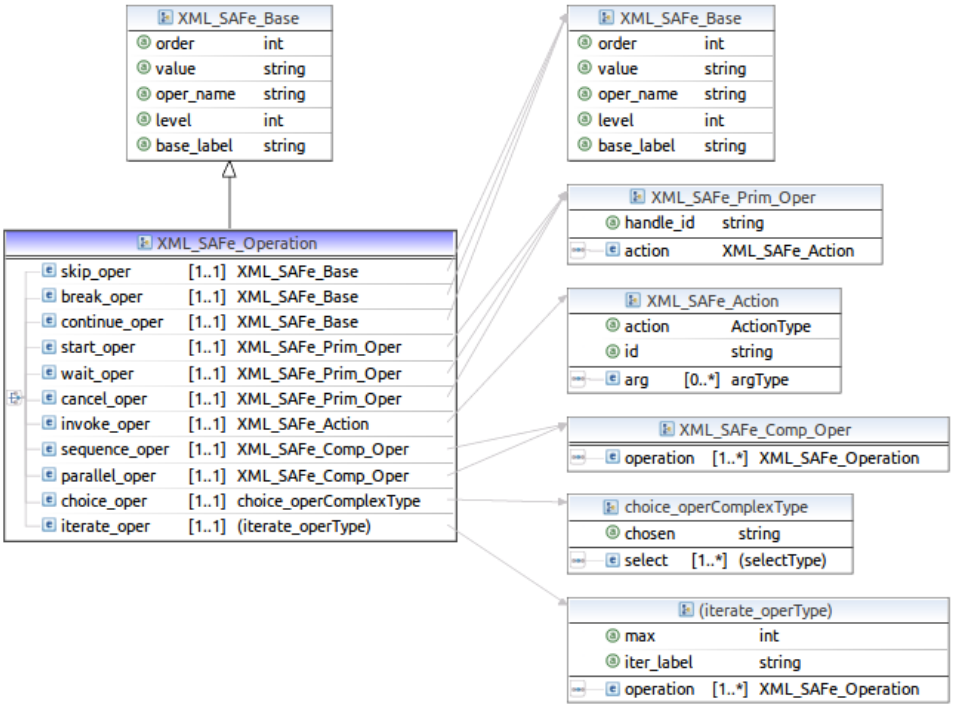


Figura 28 – A Gramática XSD para a Linguagem de Orquestração

componentes de *computação* e *conectores*.

```
1  public interface ITaskPort<T> : BaseITaskPort<T> where T:ITaskPortType {
2   // Synchronous activation of action action_id.
3   void invoke(object action_id);
4
5   /* Asynchronous activation of action action_id, where
6      a "future" object is returned for delayed synchronization. */
7   void invoke(object action_id, out IActionFuture future);
8
9   /* Asynchrouns activation of action action_id, including
10     a "reaction" method that is called when activation is completed. */
11  void invoke(object action_id, Action reaction, out IActionFuture future);
12 }
```

Código 5 – `ITaskPort` Interface (C#)

O subconjunto de linguagem de orquestração é responsável pela execução da lógica do *workflow*. Para isso, além do subconjunto de linguagem arquitetural, existe também uma gramática `XSD` para especificar a sintaxe concreta do subconjunto de orquestração de SAFeSWL, cuja estrutura é representada na Figura 28. As operações primitivas da linguagem de orquestração são apresentadas no lado esquerdo. Por exemplo, **Perform**, na sintaxe abstrata, é representada

por uma *tag* chamada `<invoke>`, do tipo `XML_SAFe_Action`.

```
<!-- creating components fragment -->
<parallel>
  <sequence>
    <invoke id_port="life-cycle-L" action="RESOLVE"     />
    <invoke id_port="life-cycle-L" action="DEPLOY"      />
    <invoke id_port="life-cycle-L" action="INSTANTIATE"/>
  </sequence>
  <sequence>
     <invoke id_port="life-cycle-R" action="RESOLVE"     />
     <invoke id_port="life-cycle-R" action="DEPLOY"      />
     <invoke id_port="life-cycle-R" action="INSTANTIATE"/>
  </sequence>
</parallel>
```

Código 6 – Exemplo de Orquestração de Workflow em XML: a Invocação de Ações

O exemplo de orquestração apresentado no Código 6 demonstra a invocação de ações através do uso da *tag* `invoke`, que realiza chamadas para *resolver*, *implantar* e *instanciar* os componentes `component-L` e `component-R`, tornando-os preparados para execução.

O *framework* `SAFe` implementa uma classe para cada *tag* XML. Assim, em cada classe é definido o código que irá executar a semântica da *tag* XML.

```
<sequence>
  <!-- creating components fragment -->
  <parallel>
  ...
  </parallel>
  <invoke id_port="compute-step" action="INITIALIZE"/>
  <iterate>
     <choice>
       <select id_port="compute-step" action="COMPUTE_STEP">     <skip/>  </select>
       <select id_port="compute-step" action="TERMINATION_FLAG"> <break/> </select>
     </choice>
  </iterate>
  <invoke id_port="compute-step" action="FINALIZE"/>
  <parallel>
     <invoke id_port="life-cycle-L" action="RELEASE"/>
     <invoke id_port="life-cycle-R" action="RELEASE"/>
  </parallel>
</sequence>
```

Código 7 – Chamando uma Ação em um Componente de Computação

Por sua vez, no exemplo de orquestração apresentado no Código 7, depois de instanciar os componentes `component-L` e `component-R`, o componente `Workflow` executa o protocolo de ativação dos nomes das ações, primeiramente invocando `INITIALIZE`. Em seguida, invoca `COMPUTE_STEP` várias vezes, até a ativação de `TERMINATION_FLAG`. Finalmente, a ação `FINALIZE` é invocada, seguida por `RELEASE` de ambos os componentes, executada concorrentemente.

4.7 Contratos Contextuais

HTS (*Hash Type System*) (CARVALHO JUNIOR *et al.*, 2016) é um sistema de tipos introduzido inicialmente pelo HPE, para os seguintes propósitos:

- Separação entre especificação (interface) e implementação de componentes, para alcançar modularidade e segurança;
- Suporte para implementações alternativas, mediante especificação de componente para diferentes contextos de execução, onde um contexto é definido pelos requisitos da aplicação e das características arquiteturais da plataforma de computação paralela alvo;
- Seleção dinâmica entre um conjunto de implementações alternativas de componentes, de acordo com o contexto de execução.

As especificações dos componentes são denominadas *componentes abstratos*, enquanto suas implementações são *componentes concretos*, ou simplesmente *componentes*. Os *componentes abstratos* representam um conjunto de componentes com a mesma interface, implementando o mesmo interesse para diferentes suposições sobre os requisitos da aplicação e características da plataforma de computação paralela de destino. Essas premissas definem o contexto de execução.

Para abstrair das suposições de contexto, um componente abstrato declara uma *assinatura contextual*, composta por um conjunto de *parâmetros de contexto*, cada um representando uma suposição distinta sobre o contexto de execução.

Um *contrato contextual* é um componente abstrato cujos parâmetros de contexto estão supridos com *argumentos de contexto*, cujo conjunto determina um contexto de execução específico.

Um parâmetro de contexto possui um tipo, determinado por um *contrato contextual* (*limitante*[1]). Um argumento de contexto que pode o suprir é determinado por um *contrato*

[1] *bound*

Parameter Name	Var.	components	Contextual Bound
bin_function	Bf	SPLITTER	PARTITIONFUNCTION[...]
input_key_type	IK	MAPPER SPLITTER	DATA
input_value_type	IV	MAPPER SPLITTER	DATA
map_function	Mf	MAPPER	MAPFUNCTION[...]
intermediary_key_type	TK	MAPPER SPLITTER SHUFFLER REDUCER	DATA
intermediary_value_type	TV	MAPPER SPLITTER SHUFFLER REDUCER	DATA
partition_function	Pf	SHUFFLER	PARTITIONFUNCTION[...]
reduce_function	Rf	REDUCER	REDUCEFUNCTION[...]
output_key_type	OK	REDUCER SPLITTER	DATA
output_value_type	OV	REDUCER SPLITTER	DATA

Tabela 4 – Assinatura Contextual dos Componentes MapReduce na HPE Shelf.

contextual compatível com o tipo do parâmetro. Dessa forma, como os contratos contextuais podem ser interpretados como tipos de componentes, os chamados *tipos de instanciação*, a relação de compatibilidade entre contratos contextuais é definida como uma relação entre subtipos.

Um componente concreto declara o contrato contextual que implementa. Além disso, em sistemas de computação paralela, os componentes são especificados por contratos contextuais onde alguns parâmetros de contexto podem ser mantidos *livres*, isto é, sem nenhum argumento de contexto associado. Quando uma ação *resolve* é ativada para um desses contratos contextuais, o Core desencadeia um procedimento de resolução para selecionar um componente concreto cujo contrato contextual é um subtipo dele, levando em consideração apenas os parâmetros de contexto não livres.

A Tabela 4 lista os parâmetros de contexto das assinaturas contextuais dos componentes abstratos MAPPER, REDUCE, SPLITTER e SHUFFLER, propostos como partes do sistema

Parameter Name	Contextual Bound
input_key_type	INTEGER
input_value_type	STRING
map_function	COUNTWORDS[...]
intermediary_key_type	STRING
intermediary_value_type	INTEGER
reduce_function	SUMVALUES[...]
output_key_type	STRING
output_value_type	INTEGER

Tabela 5 – Contrato Contextual para Computação do Contador de Palavras

paralelo de computação MapReduce anteriormente apresentado na Figura 23. Por sua vez, a Tabela 5 apresenta o conjunto de argumentos contextuais aplicados a eles para especificar os contratos contextuais de um determinado sistema de computação paralela MapReduce destinado a contar palavras em um conjunto de arquivos de texto disponíveis em um diretório. As elipses entre colchetes significam que os componentes abstratos, que definem o tipo de parâmetros de contexto e argumentos de contexto, também possuem *parâmetros/argumentos* de contexto. Observe que INTEGER e STRING devem ser subtipos de DATA, assim como COUNTWORDS e SUMVALUES devem ser subtipos de MAPFUNCTION e REDUCEFUNCTION, respectivamente. O leitor também pode notar que existem alguns parâmetros de contexto *livres*: ***bin_function*** e ***partition_function***. Nesses casos, funções *default* são aplicadas.

Na implementação atual, o mecanismo de resolução do Core selecionará as implementações genéricas existentes de MAPPER e REDUCER, que são indiferentes com relação aos tipos de chaves e valores (pares *key/value*), bem como para as funções *map* e *reduce*. No entanto, elas podem coexistir com versões especializadas, que podem fornecer implementações otimizadas de acordo com as valorações dos parâmetros de contexto.

4.8 Considerações

As tendências no uso e desenvolvimento de aplicações têm sido relacionadas recentemente ao contexto de nuvens computacionais, dadas as possibilidades de organização e controle no uso do *hardware* e do *software*. De forma semelhante, o desenvolvimento de *software* tem obtido sucesso aplicado ao paradigma de componentes, embora em menor número de propostas para o contexto de computação de alto desempenho. HPC Shelf vai de encontro a esses dois aspectos, propondo uma série de elementos arquiteturais de alto nível, suportados por componentes em nuvem através do modelo Hash. Esse modelo é de propósito geral, permitindo descrever qualquer padrão de paralelismo que poderia ser programado através de bibliotecas tradicionais de passagem de mensagens, como o MPI.

Como principais características da HPC Shelf, são destacadas a separação entre Core, onde há a resolução de contratos contextuais; o Backend, onde ocorre a instanciação e monitoramento das plataformas virtuais, destinadas às computações paralelas; e o SAFe, compondo o Front-End necessário para a construção de aplicações. Para provedores de aplicações, o SAFeSWL, linguagem introduzida pelo SAFe, é o elemento responsável pelos aspectos arquiteturais e de controle de fluxo de trabalho, onde o relacionamento entre componentes é suportado por

conectores especiais, os *bindings*, de ações e serviços. Contudo, HPC Shelf ainda necessita de estudos de caso que possam explorar de forma abrangente sua expressividade para construção de aplicações, bem como sua eficiência computacional, ponderando a relação entre os benefícios do ambiente de nuvem e o alto nível de abstração para desenvolvimento e execução de software com requisitos de Computação de Alto Desempenho. Essa tem sido uma das motivações relacionadas a este trabalho.

Outra motivação consiste na engenharia de software paralelo e técnicas para suportar eficientemente o processamento em larga escala de grafos. Isso tem se mostrado um desafio, especialmente pelo crescimento dos dados em diferentes segmentos, como redes sociais, georreferenciamento e ciências contemporâneas, justificando o surgimento de uma série de trabalhos com MapReduce, Pregel e outras soluções apresentadas no Capítulo 3. Problemas em grafos estão entre os importantes segmentos da ciência da computação que necessitam do paralelismo. Por esse motivo, tem-se trabalhado a proposta de construção de um *arcabouço* de componentes para grafos, projetado para operar efetivamente sobre uma plataforma HPC Shelf. O próximo capítulo apresenta detalhadamente a arquitetura desse arcabouço, denominada Gust, desenvolvido sobre um um outro arcabouço de componentes, de propósito geral, destinado a computações MapReduce, também introduzido por esta Tese.

5 GUST - ARCABOUÇO DE COMPONENTES EM NUVEM PARA O PROCESSAMENTO DE GRAFOS EM LARGA ESCALA

Gust é um arcabouço de componentes paralelos para construção de sistemas de computação paralela voltados ao processamento em larga escala de grafos grandes. Tem por objetivo, portanto, o suporte à implementação dos requisitos computacionais de algoritmos paralelos sobre grafos, bem como a definição das plataformas de execução adequadas para esses algoritmos, através da HPC Shelf. Dessa forma, é uma aplicação concebida para essa plataforma, a qual encapsula um conjunto de recursos e técnicas para paralelismo que permitem que desenvolvedores tirem proveito da capacidade de processamento de plataformas de computação paralelas distribuídas geograficamente, em diferentes domínios, constituindo o que chamamos de infraestruturas de computação paralela de larga escala.

Este capítulo se divide em três partes. Primeiramente, destacam-se a problemática e as motivações para realização deste trabalho, tendo como base a revisão bibliográfica realizada nos capítulos 2 e 3. Após isso, introduzem-se os estudos preliminares realizados com a concepção, projeto e implementação de componentes para descrição de sistemas de computação paralela para processamento MapReduce. De fato, essa constitui a primeira aplicação da HPC Shelf, a partir da qual o Gust é desenvolvido, porém também constituindo, em si, uma das contribuições desta Tese. Finalmente, destacam-se as características, bem como os principais componentes e algoritmos (também vistos como *benchmarks*) implementados como estudo de caso para o arcabouço.

5.1 Problemática e Motivações

Aplicações com requisitos de Computação de Alto Desempenho (CAD) são frequentemente desenvolvidas visando explorar detalhes das arquiteturas de computação paralela. Para isso, com base no conhecimento arquitetural, os desenvolvedores adotam técnicas eficientes que otimizam a utilização dos recursos, tais como unidades de processamento, memória (local/distribuída), aceleradores computacionais (e.g. *GPUs* (FAN *et al.*, 2004), *MICs* (DURAN; KLEMM, 2012) e *FPGAs* (HERBORDT *et al.*, 2007)), bem como interação através de redes de comunicação. No entanto, a atual complexidade do *software* tem exigido um aprimoramento dessa abordagem, necessitando de ferramentas que propiciem maior produtividade no desenvolvimento de programas paralelos.

Atualmente, com o grande volume de dados originários das pesquisas científicas

(NIELSEN, 2005), aplicações móveis e na *Internet* (FALOUTSOS *et al.*, 1999) (e.g. motores de busca, redes sociais, aplicações georreferenciadas - GPS, etc), emergiram plataformas de computação para suprir a demanda por processamento eficiente de intensas massas de dados, o que resultou no sucesso de *frameworks* que implementam modelos como MapReduce e Pregel, tais como Spark, Giraph, GraphX[1], dentre outros. Trata-se de um cenário em que boa parte desses *frameworks* buscam oferecer alto nível de abstração para o desenvolvimento das aplicações, eliminando a necessidade do desenvolvedor programar o paralelismo inerente. Vê-se que isso é diferente das práticas utilizadas nas aplicações legadas de CAD, onde predomina a programação estruturada, em linguagens como Fortran ou C, que possuem significativa expressividade de paralelismo, tendo apoio de ferramentas como as bibliotecas de passagem de mensagens (MPI), mas ainda possuindo baixo nível de abstração. Nos *frameworks* emergentes, significa que a preocupação com a arquitetura e com os detalhes da paralelização é potencialmente delegada aos próprios *frameworks*. Com isso, a parte complexa de paralelismo pode ser automaticamente realizada com a aplicação de suposições sobre a distribuição e processamento dos dados e das computações, tarefa comumente programada de forma manual em aplicações científicas legadas e atuais (CHAVARRÍA-MIRANDA *et al.*, 2015; BAILEY, 1991).

O alto nível surgente nos *frameworks* modernos está mais próximo da engenharia de *software* contemporânea, pois possibilita também o incremento da escala de desenvolvimento, requisito especialmente útil para a crescente complexidade do *software* de CAD. Porém, o encapsulamento das técnicas de paralelismo nem sempre permite explorar o melhor das plataformas de execução, como ocorreu no caso dos algoritmos de escalonamento ineficientes para máquinas heterogêneas no Hadoop, onde *Zaharia et al* (ZAHARIA *et al.*, 2008) propuseram um novo escalonador de tarefas. Além disso, existem algoritmos, por exemplo de particionamento e balanceamento de carga de grafos, sensíveis aos detalhes dos ambientes de execução, bem como sensíveis às técnicas de sincronização de mensagens (HARSHVARDHAN *et al.*, 2014). Por exemplo, Xu *et al*, sobre partição e balanceamento de carga, demonstra a perda de desempenho quando há máquinas com diferentes capacidades, além de canais para operações de comunicação diferentes (XU *et al.*, 2015). Nesse trabalho, foram experimentadas 26 máquinas com 3 delas diferentes das demais (menor capacidade), onde a remoção dessas máquinas diferentes resulta em aumento do desempenho, ou ainda, alternativamente, podem receber menos carga que as demais, mantendo o desempenho. De fato, algoritmos possuem perfis diversificados quanto aos requisitos

[1] Responsável por processamento de grafos no Spark.

de computação distribuída, de forma que o ambiente de execução deve ser levado em conta para minimizar degenerações. Em grafos, por exemplo, alguns algoritmos (HARSHVARDHAN *et al.*, 2014; XIE *et al.*, 2015) possuem desempenho melhor ou pior de acordo com o caráter síncrono ou assíncrono do mecanismo de troca de mensagens, como discutido na Seção 3.7.

Identificar a melhor forma de execução em uma plataforma, bem como os requisitos de paralelismo para algoritmos particulares, não é uma tarefa simples se for feita de forma automatizada, conclusão já observada no lançamento do *framework* Giraph++ (TIAN *et al.*, 2013), quando ofereceram ao utilizador a possibilidade de especificar a estratégia de particionamento. Nesse trabalho, o utilizador, ao definir sua estratégia, deve conhecer também a arquitetura, pois considerará ou não a existência de unidades de processamento heterogêneas, delegando mais ou menos carga segundo uma certa capacidade do *hardware*, caso assim considere vantajoso. Porém, há *frameworks* recentes (CHEN *et al.*, 2017; XIE *et al.*, 2015) que buscam suportar determinadas características ou técnicas automatizadas para paralelismo, onde seus resultados apontam sucesso para alguns algoritmos. Por exemplo, eles apostam em heurísticas para atingir modos híbridos em sincronismo/assincronismo, supondo quando um é melhor que o outro, bem como modos empurrar/puxar (*push/pull*) dos dados. Todavia, a ciência deve ainda examinar abordagens com propósito geral no processamento de grafos, onde há a possibilidade de equilibrar instruções automáticas encapsuladas no *framework* e a expressividade do desenvolvedor, o qual informa suas necessidades para o algoritmo em uma dada plataforma, com uma certa técnica de paralelismo. Com isso, características de computação (Seção 3.6.1), tais como modelo de programação, fases de execução, modo empurrar/puxar, modo de execução com sincronismo/assincronismo/híbrido, também podem ser generalizados, ou seja, personalizáveis. Por exemplo, se um algoritmo particionador A é eficiente para balancear carga de grafos em um conjunto de máquinas homogêneas H, esse mesmo algoritmo tem chances de não ser eficiente se o conjunto de máquinas for alterado para um conjunto de máquinas heterogêneas H'. Porém, se A receber uma instrução dizendo qual o contexto do conjunto de máquinas alvo, A tem pelo menos a chance de recalcular suas suposições, podendo ainda ser eficiente. Em *frameworks* que adotam *pipe-lines* de MapReduce, sugerido por *Condie et al* (CONDIE *et al.*, 2009), esse cenário de particionamento é crítico, especialmente para multitarefas escalonadas em um mesmo *cluster* multi-inquilino, ocupando concorrentemente recursos das máquinas, caracterizando a necessidade de balanceamento dinâmico de carga entre fases de processamento ou *pipe-lines*.

Um caminho que pode atingir esse equilíbrio é observado através das práticas

propostas para a plataforma HPC Shelf, discutida no Capítulo 4. Isso porque utiliza como pilar o desenvolvimento baseado em componentes Hash, no qual os componentes são desenvolvidos e selecionados por uma certa aplicação segundo suposições a respeito de um contexto de execução. Esse contexto, potencialmente genérico, é naturalmente expresso pelo desenvolvedor na criação das aplicações, o que é uma característica de alto nível que instrui a execução, onde características de plataformas e requisitos de aplicações são resolvidos por meio do elemento arquitetural Core. Um arcabouço para grafos grandes, em conformidade com o modelo de componentes Hash e a nuvem HPC Shelf, permite potencializar o propósito geral, pois possibilita que a contextualização defina a melhor estratégia para execução, que poderá ficar sujeita às instruções do especialista no domínio de grafos e da generalidade dos componentes disponibilizados pelo desenvolvedor especialista em computações paralelas.

Dessa forma, será introduzido neste capítulo o arcabouço Gust, que visa suportar o processamento de grafos em larga escala, aplicando a engenharia de software baseada em componentes às técnicas discutidas nos *frameworks* do Capítulo 3. Na Seção 5.3, discute-se como a estrutura de componentes é organizada para suprir os requisitos de grafos. Para isso, apresentam-se primeiramente os estudos envolvendo a implementação de um arcabouço MapReduce baseado em componentes paralelos do modelo Hash, que é voltado à plataforma HPC Shelf para computação paralela em larga escala. Alguns componentes Gust estendem componentes desse arcabouço, para permitir o funcionamento em ambiente distribuído. Portanto, após as definições do MapReduce, destacam-se arquitetura e propriedades do Gust.

5.2 MapReduce Sobre a HPC Shelf

A implementação de um arcabouço MapReduce sobre a HPC Shelf constitui a primeira aplicação para essa plataforma. Nessa aplicação, o especialista especifica um ou mais fluxos de operações de mapeamento e redução, possivelmente paralelas e iterativas, constituindo um sistema de computação paralela que será traduzido para a linguagem SAFeSWL. Como discutido no Capítulo 4, dentro do contexto da plataforma HPC Shelf, SAFeSWL é a linguagem de descrição de sistemas de computação paralela do arcabouço de aplicações SAFe. Nesta seção, serão descritos os componentes necessários para constituição da arquitetura e do fluxo de orquestração de ações computacionais (*workflow*) para sistemas de computação paralela MapReduce.

Os agentes de mapeamento e redução são representados por componentes dos tipos

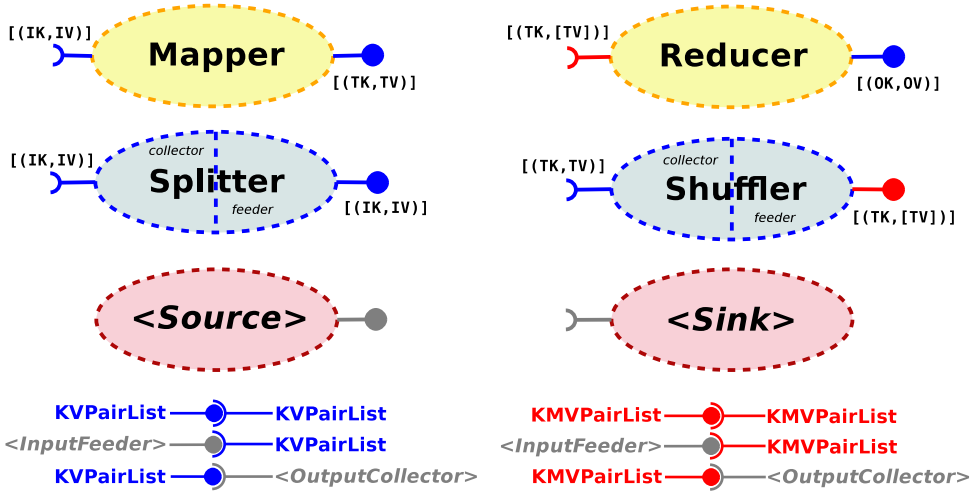


Figura 29 – Blocos de Construção de Sistemas MapReduce na HPC Shelf

MAPPER e REDUCER, respectivamente, os quais são conectados entre si através dos conectores SPLITTER e SHUFFLER. Relembrando o que foi apresentado na Seção 4.3, conectores se destacam pela sua capacidade de ligar componentes de computação e fontes de dados dispostos em plataformas virtuais distintas, a fim de realizar a orquestração de suas ações ou servir de meio para sua coreografia. Para isso, é constituído de facetas, cada uma das quais implantada sobre a plataforma virtual onde um dos componentes de computação ou fontes de dados que liga encontram-se implantados. Dados de entrada e saída encontram-se armazenados em componentes da espécie fontes de dados, conectados a instâncias dos componente SPLITTER e SHUFFLER.

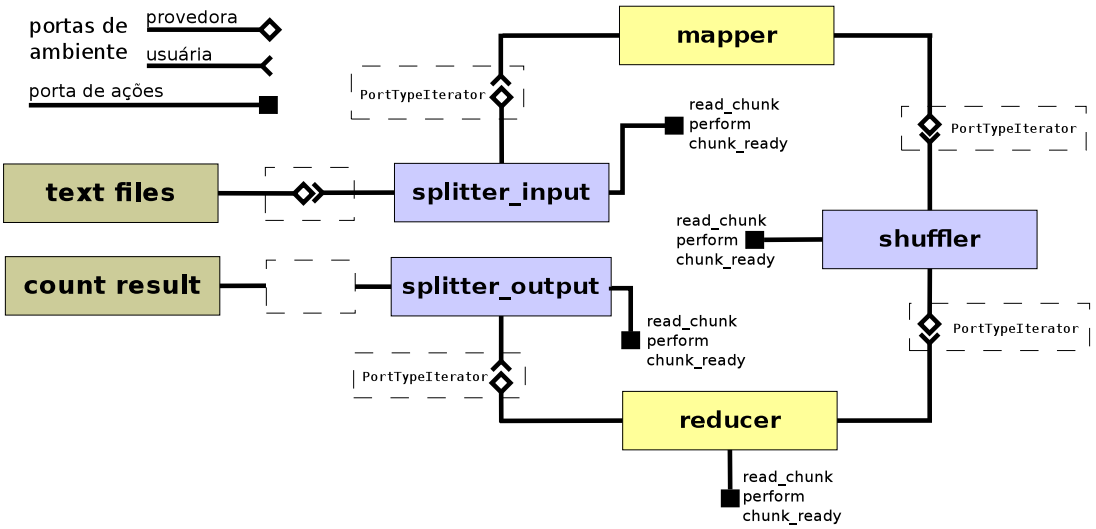


Figura 30 – Arquitetura de um Sistema MapReduce para Contagem de Palavras.

A Figura 30 apresenta um arranjo típico dos componentes de um sistema de computação paralela MapReduce simples, utilizado em um estudo de caso com contagem de palavras

em arquivos textos contidos em um diretório, explicado adiante.

De forma geral, esses sistemas de computação paralela MapReduce que propomos para a HPC Shelf exploram o paralelismo de três formas: em pequena escala, em cada agente de mapeamento e redução, os quais são componentes paralelos do modelo Hash executando em *clusters* separados (plataformas virtuais); em larga escala, entre múltiplos agentes de mapeamento ou de redução atuando de forma paralela em uma fase da computação; e, também em larga escala, explorando o padrão *pipeline* de processamento paralelo, entre fases da computação dentro de uma mesma iteração. Para que esse *pipeline* seja possível, utiliza comunicação incremental por meio *bindings* de ambiente do tipo PORTTYPEITERATOR, os quais suportam comunicação de pares chave/valor em "*chunks*". Portanto, um "*chunk*" é definido como um agrupamento de pares chave/valor que pode ser transmitido entre os componentes do sistema MapReduce. O tamanho dos "*chunks*" controla a granularidade da comunicação, a fim de minimizar sua sobrecarga.

Devemos ressaltar os componentes de ligação (*bindings*), tanto de ambiente quanto de tarefas, empregados na arquitetura de sistemas de computação paralela MapReduce, exemplificados na Figura 30. Lembrando que os componentes *bindings*, genéricos, empregados são ENVIRONMENTBINDING e TASKBINDING, temos as seguintes valorações para os argumentos de contexto aplicados aos seus parâmetros (definição dos tipos das portas) para o sistema MapReduce de contagem de palavras:

- ENVIRONMENTBINDING [***client_port_type*** = PORTTYPEITERATOR,
  server_port_type = PORTTYPEITERATOR],
  para comunicação de pares chave/valor de forma incremental de um componente para outro;
- ENVIRONMENTBINDING [***client_port_type*** = PORTTYPEITERATOR,
  server_port_type = PORTTYPETEXTFILESREADER],
  para comunicação entre o componente **splitter_input** e a fonte de dados de entrada de onde são lidos os arquivos-texto a serem processados (**text files**);
- ENVIRONMENTBINDING [***client_port_type*** = PORTTYPEDATASINKINTERFACE,
  server_port_type = PORTTYPEITERATOR],
  para comunicação entre a fonte de dados para onde serão copiados os pares que representam a saída do processamento (**count result**) e o conector **splitter_output**, que recebe o resultado do agente de redução (**reducer**); spitter_output
- TASKBINDING [***port_type*** = TASKPORTTYPEADVANCEREADCHUNK],

tal que o tipo TASKPORTTYPEADVANCEREADCHUNK inclui uma única ação, chamada `READ_CHUNK`, cuja ativação, quando completada, faz o componente ler o próximo "*chunk*" da porta usuária do tipo PORTTYPEITERATOR;

- TASKBINDING [***port_type*** = TASPORTTYPEADVANCEPERFORM],
  tal que o tipo TASKPORTTYPEADVANCEPERFORM inclui uma única ação, chamada `PERFORM`, cuja ativação, quando completada, faz o componente realizar o processamento sobre o "*chunk*" atual, mais recentemente obtido;
- TASKBINDING [***port_type*** = TASKPORTTYPEADVANCECHUNKREADY],
  tal que o tipo TASKPORTTYPEADVANCECHUNKREADY inclui uma única ação, chamada `CHUNK_READY`, cuja ativação, quando completada, indica que um novo "*chunk*" foi gerado pelo componente em sua porta provedora do tipo PORTTYPEITERATOR, podendo ser lido por outro componente através da porta usuária ligada a ela através de um *binding*;
- TASKBINDING [***port_type*** = TASKPORTTYPEADVANCE],
  tal que o tipo TASPORTTYPEADVANCE inclui todas as três ações: `READ_CHUNK`, `PERFORM` e `CHUNK_READY`.

Uma importante observação é que os *bindings* de ambiente acima descritos são diretos, isto é, suas unidades *cliente* e *servidora* residem no mesmo espaço de endereçamento. Toda a comunicação entre plataformas virtuais onde estão os componentes envolvidos na computação MapReduce é realizada por intermédio de componentes SPLITTER e SHUFFLER. Entretanto, também é possível conectar agentes de mapeamento, agentes de redução e fontes de dados de forma indireta, ou seja, através de *bindings* de ambiente indiretos, sem intermediação dos conectores, dependendo dos interesses de uma aplicação. Para isso, é necessário haver uma implementação indireta do componente abstrato ENVIRONMENTBINDING para os argumentos de contexto apropriados. Porém, em nossos estudos de caso, não houve necessidade de explorar essa possibilidade.

Em relação às portas de ações, ao serem conectadas a portas de ações de mesmo tipo, incluindo a portas do componente Workflow do sistema de computação paralela, a orquestração do sistema MapReduce pode ser controlada por esse último.

As seções seguintes apresentam mais detalhes gerais sobre cada um dos tipos de componentes empregados em sistemas MapReduce sobre a HPC Shelf, destacando suas arquiteturas de sobreposição e assinaturas contextuais.

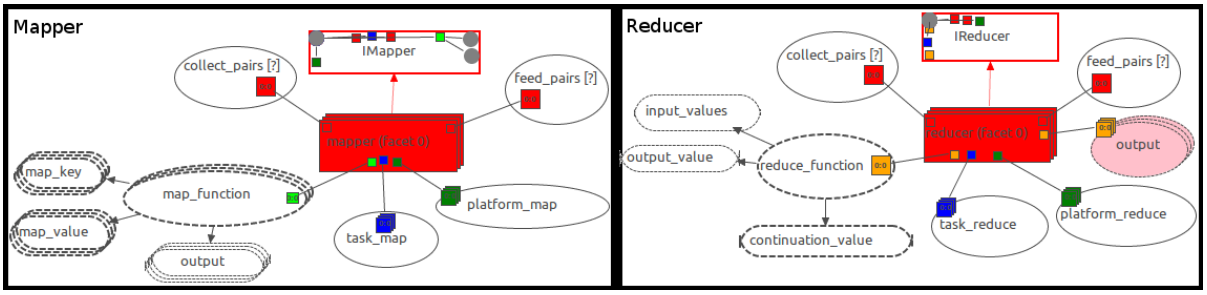


Figura 31 – Arquitetura por Sobreposição de MAPPER e REDUCER

input_key_type	*IKey*	DATA	tipo da chave de entrada
input_key_value	*IValue*	DATA	tipo do valor de entrada
map_function	Mf	MAPFUNCTION	tipo da função de mapeamento
intermediary_key_type	*TKey*	DATA	tipo da chave intermediária
intermediary_key_value	*TValue*	DATA	tipo do valor intermediário

Tabela 6 – Parâmetros de Contexto do Contrato de MAPPER

5.2.1 Agente de Mapeamento (MAPPER)

A Figura 31 (à esquerda) ilustra a arquitetura de sobreposição do componente abstrato MAPPER, usando uma captura de tela do FrontEnd do HPE, adaptada para a necessidade de escrever componentes para o protótipo de referência da HPC Shelf. Ele possui uma única unidade paralela chamada `mapper`, responsável pela execução de uma função de mapeamento sobre pares chave/valor de entrada recebidos através da porta usuária `collect_pairs` para geração de pares chave/valor intermediários, retornados através da porta provedora `feed_pairs`. Ambas são do tipo PORTTYPEITERATOR. Finalmente, possui uma única porta de ações, chamada `task_map`, do tipo TASKPORTTYPEADVANCE.

Na Tabela 6, são apresentados os parâmetros de contexto da assinatura contextual do componente abstrato MAPPER, destacando-se os tipos da chaves e valores dos pares de entrada e intermediários, bem como o tipo da função de mapeamento, especializada para processar dados desses tipos.

É importante relembrar que diferentes implementações de MAPPER podem existir de maneira transparente, especializadas de acordo com os tipos das chaves de entrada e intermediárias, bem como o tipo da função de mapeamento empregada. Esse é um dos pontos fortes da orientação a componentes proposta pela HPC Shelf para sistemas MapReduce, permitindo contornar, usando técnicas adequadas de acordo com a aplicação, problemas de expressividade comuns de outros *frameworks*, aonde o agente de mapeamento é genérico.

intermediary_key_type	*TKey*	DATA	tipo da chave intermediária
intermediary_key_value	*TValue*	DATA	tipo do valor intermediário
reduce_function	*Rf*	REDUCEFUNCTION	tipo da função de redução
`output_key_type`	*OKey*	DATA	tipo da chave de saída
output_key_value	*OValue*	DATA	tipo do valor de saída

Tabela 7 – Parâmetros de Contexto do Contrato de REDUCER

Ao serem conectadas a portas de ações de mesmo tipo, incluindo a portas do componente Workflow do sistema de computação paralela, a orquestração do sistema MapReduce pode ser controlada por esse último.

5.2.2 *Agentes de Redução (*REDUCER*)*

A Figura 31 (à direita) ilustra a arquitetura de sobreposição para o componente abstrato REDUCER. Sua descrição é análoga a do componente MAPPER. Assim, possui uma única unidade paralela chamada `reducer`, responsável pela execução de uma função de redução sobre pares chave/valor intermediários recebidos através da porta usuária `collect_pairs` para geração de pares chave/valor de saída, retornados através da porta provedora `feed_pairs`. Também possui uma única porta de ações (`task_reduce`), do tipo TASKPORTTYPEADVANCE.

Na Tabela 7, são apresentados os parâmetros de contexto da assinatura contextual do componente abstrato REDUCER, onde também destacam-se os tipos da chaves e valores dos pares de intermediários e de saída, bem como o tipo da função de redução especializada para processar dados desses tipos.

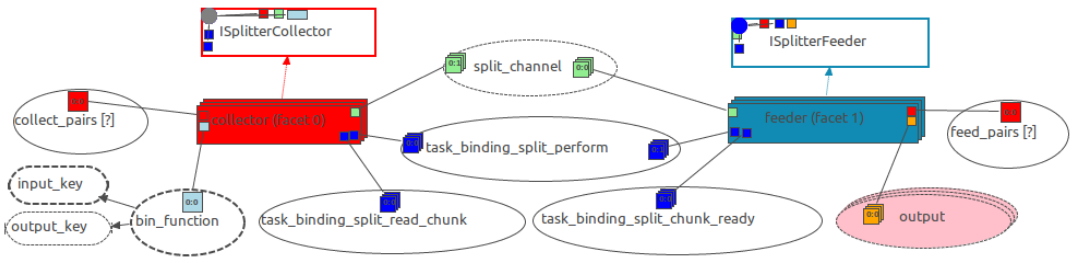


Figura 32 – Arquitetura por Sobreposição do Componente Abstrato SPLITTER

5.2.3 *O Conector* SPLITTER

Um conector do tipo SPLITTER possui arquitetura de sobreposição apresentada na Figura 32. A Tabela 8, por sua vez, apresenta a sua assinatura contextual, incluindo parâmetros

input_key_type	*IKey*	DATA	Tipo da chave de entrada
input_key_value	*IValue*	DATA	Tipo do valor de entrada
bin_function	*Bf*	PARTITIONFUNCTION	Tipo da função de distribuição das chaves de entrada entre os agentes de mapeamento

Tabela 8 – Parâmetros de Contexto do Contrato de SPLITTER

de contexto que definem os tipos dos pares chave/valor que ele lida, bem como da *função de particionamento*.

Um conector do tipo SPLITTER é constituído de duas facetas múltiplas, ou seja, que podem ser replicadas em várias instâncias, as quais chamamos *coletora* e *fornecedora*. Cada faceta coletora possui uma única unidade paralela chamada `collector`, capaz de coletar pares chave/valor de diferentes fontes através de uma porta usuária do tipo PORTTYPEITERATOR. Por sua vez, cada faceta fornecedora possui uma única unidade paralela chamada `feeder`, que fornece pares chave/valor a diferentes destinos através de uma porta provedora também do tipo PORTTYPEITERATOR. De fato, um connector SPLITTER distribui um conjunto de pares chave/valor recebidos através de suas facetas coletoras, fornecidas por uma ou mais várias fontes de entrada, entre uma ou mais várias fontes de saída, através de sua faceta fornecedora, usando a função de particionamento definida pelo parâmetro de contexto `bin_function`.

Fontes de entrada e saída podem ser quaisquer componentes que possam ser ligados através através das respectivas portas supracitadas, das facetas coletoras e fornecedoras, as quais encontram-se associadas, incluindo, indistintamente, agentes de mapeamento (MAPPER), agentes de redução (REDUCER), fontes de dados de entrada e fontes de dados de saída. Note que essa é uma visão mais geral de um sistema de computação paralela MapReduce, visto que em um sistema convencional (não-iterativo) existe uma única fonte de dados de entrada do lado coletor e um único agente de mapeamento do lado fornecedor.

Vale ressaltar que comunicação entre as facetas colectoras e fornecedoras, assim como para qualquer conector da HPC Shelf é realizada por meio de um interface de passagem de mensagens implementada por um componente chamado `split_channel`, referenciado na arquitetura de sobreposição da Figura 32.

O componente SPLITTER possui três portas de ações, chamadas `task_port_read_chunk`, `task_port_chunk_ready` e `task_port_perform`, respectivamente dos tipos TASKPORTTYPEADVANCEREADCHUNK, TASKPORTTYPEADVANCEPERFORM e TASKPORTTYPEADVANCECHUNKREADY, explicadas anteriormente.

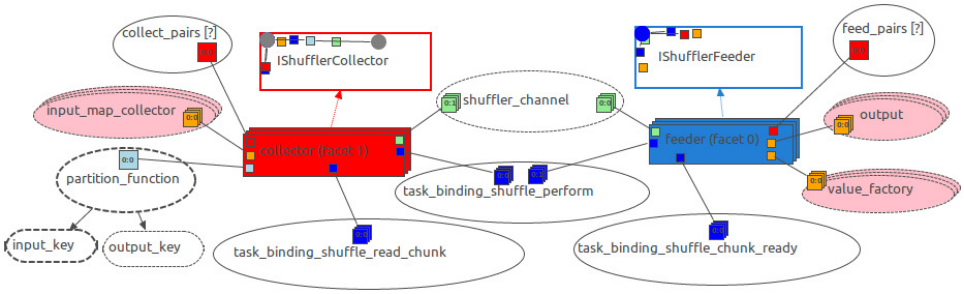


Figura 33 – Arquitetura por Sobreposição do Componente Abstrato SHUFFLER

intermediary_key_type	*TKey*	DATA	Tipo da chave de entrada
intermediary_key_value	*TValue*	DATA	Tipo do valor de entrada
partition_function	*Pf*	PARTITIONFUNCTION	Tipo da função de distribuição das chaves de entrada entre os agentes de redução

Tabela 9 – Parâmetros de Contexto do Contrato de SHUFFLER

5.2.4 *O Conector* SHUFFLER

Um conector do tipo SHUFFLER também faz a intermediação entre uma ou mais fontes de entrada e uma ou mais fontes de saída, possuindo também, para essa finalidade, facetas coletoras e fornecedoras. A arquitetura de sobreposição do componente abstrato SHUFFLER encontra-se na Figura 33, enquanto sua assinatura contextual encontra-se apresentada na Tabela 9.

A diferença de um conector SHUFFLER para um conector SPLITTER está no tipo de dado fornecido para as fontes de saída. Enquanto o último fornece pares chave/valor, o primeiro fornece pares chave/multi-valor, pois agrupa os valores associados a mesma chave que são recebidos das fontes de entrada. Os pares chave/multi-valor são também distribuídos entre as facetas fornecedoras através de uma função de particionamento aplicada às chaves, definida pelo parâmetro de contexto `partition_function`.

As portas de ações de componentes SHUFFLER correspondem às mesmas portas descritas para componentes SPLITTER.

5.2.5 *Orquestração em Sistemas MapReduce*

A fim de explicar o fluxo de orquestração de sistemas de computação paralela MapReduce, usaremos o exemplo de arquitetura da Figura 30, o qual foi empregado para a contagem de ocorrências de palavras em arquivos de texto contidos em um diretório acessível por um

componente fonte de dados de entrada. Vamos lembrar, inicialmente, que os componentes de computação e conectores envolvidos contém ações que sincronizam o fluxo de execução e troca de dados, suportados pelas definições da linguagem SAFeSWL, usando os *bindings* de tarefa. O componente *binding*, genérico, é do tipo TASKBINDINGBASE, possuindo um parâmetro de contexto denominado ***task_port_type***, do tipo TASKPORTTYPE, que delimita o tipo do *binding*, ou seja, o conjunto de nomes de ações que suporta. Os argumentos usados nos conectores (SPLITTER e SHUFFLER) de sistemas MapReduce são TASKPORTTYPEADVANCEREADCHUNK, TASKPORTTYPEADVANCEPERFORM e TASKPORTTYPEADVANCECHUNKREADY, respectivamente contendo uma única ação: `READ_CHUNK`, `PERFORM` e `CHUNK_READY`. Há ainda o tipo TASKPORTTYPEADVANCE, usado em MAPPER e REDUCER, que inclui todas as três ações.

No caso do sistema de contagem de palavras, que não é iterativo, somente não é explorado o paralelismo entre múltiplos agentes de mapeamento e redução, ou seja, um único agente de mapeamento é responsável pela fase de mapeamento e um único agente de redução é responsável pela fase de redução. O paralelismo ocorre entre cada agente de mapeamento e de redução, bem como no *pipeline* formado entre esse dois.

A seguir, descrevemos os passos da execução do sistema MapReduce de contagem de palavras, de acordo com a nomenclatura apresentada na Figura 30:

- A fonte de dados de entrada (**text files**) oferece o acesso a um conjunto de arquivos de texto ao conector **spitter_input**, através de um *binding* cujo tipo da porta provedora, ligada a **data_source**, é PORTTYPETEXTFILESREADER e cujo tipo da porta usuária, ligada a **spitter_input**, é PORTTYPEITERATOR. Portanto, esse *binding* é capaz de ler o conteúdo dos arquivos de texto e emitir uma sequência de "*chunks*", que são sequências de pares chave/valor de entrada, representando o conteúdo das linhas desses arquivos, onde a chave representa um inteiro sequencial e o valor representa o conteúdo da linha em si. O conector **spitter_input** lê os "*chunks*" incrementalmente, operação completada cada vez que se completa uma ativação da ação `READ_CHUNK`, na sua porta `task_port_read_chunk`.
- Cada vez que se completa uma ativação da ação `PERFORM` na porta `task_port_perform`, **spitter_input** usa a função de particionamento *default* para distribuir os pares de entrada do "*chunk*" atual, mais recentemente obtido, entre as facetas fornecedoras, com base na chave inteira. A cada "*chunk*" distribuído, a ação `CHUNK_READY` da porta `task_port_chunk_ready` é ativada, a partir de quando uma ativação à ação `READ_CHUNK` da porta `task_map` pode ser ativada, para que **mapper** consuma o "*chunk*" produzido.

- O agente de mapeamento **mapper**, após a leitura de um *chunk*, aplica a função de mapeamento a cada par de entrada quando uma ativação da ação `PERFORM` é completada em sua porta `task_map`. Essa função de mapeamento emite, para cada linha, um conjunto de pares chave/valor intermediários, associando a uma palavra contida na linha o número de ocorrências nessa linha. Quando uma certa quantidade de pares é emitida, um "*chunk*" é formado e a ação `CHUNK_READY` é ativada. Quando essa ação se completa, uma ativação da ação `READ_CHUNK` de **shuffler** pode ser ativada, para consumir o "*chunk*" de pares intermediários.
- O conector **shuffler**, a cada vez que uma ativação da ação `PERFORM` é completada em sua porta `task_shuffle_perform`, distribui os pares intermediários do "*chunk*" atual entre suas facetas fornecedoras, de acordo com a função de particionamento *default*, onde são agrupadas em pares chave/multi-valor que serão consumidas de forma incremental pelo agente de redução **reducer**. Nesse caso, a ação `CHUNK_READY` é ativada cada vez que um "*chunk*" é completamente lido. Uma vez completada essa ativação, a ativação da ação `READ_CHUNK` do **reducer** pode ser completada.

  Nesse ponto, valem algumas observações importantes. Os pares chave/multi-valor emitidos pelo shuffler só estão completamente definidos após o processamento de todos os "*chunks*" intermediários, quando todos os valores associados a uma chave são recebidos. No caso do contador de palavras, é preciso processar todas as linhas de entrada a fim de que sejam agrupadas todas as ocorrências de palavras em todas essas linhas. Logo, a fim de manter o paralelismo *pipeline* entre os agentes de mapeamento e redução, é necessário um mecanismo segundo o qual **reducer** possa processar os valores de cada par chave/multi-valor emitido pelo **shuffler** de forma incremental, explicado adiante.
- O agente de redução **reducer**, a cada vez que uma ativação da ação `PERFORM` é completada em sua porta `task_reduce`, processa os valores emitidos pelo **shuffler** e consumidos após a última ativação da ação `READ_CHUNK` ter se completado. Além disso, novos pares chave/multi-valor podem surgir. Para que isso seja possível, a definição da função de redução inclui como parâmetro um acumulador que é alimentado com a última saída produzida pela função de redução para uma certa chave de um par chave/multi-valor.

  Além disso, o agente de redução mantém um dicionário armazenando pares chave/valor de saída parciais, o qual é atualizado cada vez que a função de redução é aplicada a um novo valor emitido associado à chave. Note que os pares chave/valor de saída parcial somente

podem ser considerados pares de saída definitivos após o processamento de todos os pares chave/multi-valor intermediários. Quando isso, acontece, são emitidos pelo agente de redução, em "*chunks*", a cada ativação de `CHUNK_READY`, quando esses pares de saída podem ser lidos pelo conector **splitter_output** quando a ação `READ_CHUNK` de sua porta `task_port_read_chunk` se completa. Em uma aplicação onde seria possível decidir pela finalização de um par de saída antes do final da computação, uma implementação específica do componente REDUCER poderia existir para garantir paralelismo *pipeline* entre a fase de redução e fases posteriores, inclusive no caso de computações iterativas.

- Finalmente, o conector **splitter_output** é responsável por alimentar, ao final da computação, a fonte de dados de saída (**count result**) com as chaves de saída emitidas pelo agente de redução, cada vez que se completam as respectivas ativações das ações `PERFORM` e `CHUNK_READY`.

```
<parallel>
   <iterate>
    <sequence>
       <invoke id_port="task_split_input_read_chunk" action="READ_CHUNK" />
       <invoke id_port="task_split_input_perform" action="PERFORM" />
    </sequence>
   </iterate>
   <iterate>
    <sequence>
       <invoke id_port="task_split_input_chunk_ready" action="CHUNK_READY" />
       <invoke id_port="task_map" action="READ_CHUNK" />
       <start id_port="task_map" action="PERFORM" handle_id="h1" />
    </sequence>
   </iterate>
   <iterate>
    <sequence>
       <invoke id_port="task_map" action="CHUNK_READY" />
       <invoke id_port="task_shuffle_read_chunk" action="READ_CHUNK" ></invoke>
       <start id_port="task_shuffle_perform" action="PERFORM" handle_id="h2" />
    </sequence>
   </iterate>
   <iterate>
    <sequence>
       <invoke id_port="task_shuffle_chunk_ready" action="CHUNK_READY" />
       <invoke id_port="task_reduce" action="READ_CHUNK" />
       <start id_port="task_reduce" action="PERFORM" handle_id="h3" />
    </sequence>
   </iterate>
   <iterate>
    <sequence>
       <invoke id_port="task_reduce" action="CHUNK_READY" />
```

```
32          <invoke id_port="task_split_output_read_chunk" action="READ_CHUNK" />
33          <start id_port="task_split_output_perform" action="PERFORM" handle_id="h4" />
34       </sequence>
35      </iterate>
36      <iterate>
37       <invoke id_port="task_split_output_chunk_ready" action="CHUNK_READY" />
38      </iterate>
39  </parallel>
```

Código 8 – SAFeSWL Fragmento de orquestração MapReduce.

Na Listagem 8, apresentamos um fragmento do fluxo de orquestração MapReduce para os componentes na arquitetura apresentada na Figura 30, ignorando a ativação das ações de controle do ciclo de vida dos componentes de computação e conectores envolvidos (**splitter_input**, **mapper**, **shuffler**, reducer e **splitter_output**).

A Tabela 10 apresenta os argumentos de contexto aplicados aos parâmetros de contexto dos componentes do sistema MapReduce para contagem de palavras.

Nome do Parâmetro	Limite Contextual
input_key_type	INTEGER
input_value_type	STRING
map_function	COUNTWORDS[...]
intermediary_key_type	STRING
intermediary_value_type	INTEGER
reduce_function	SUMVALUES[...]
output_key_type	STRING
output_value_type	INTEGER

Tabela 10 – Argumentos de contexto dos componentes para contagem de palavras

Nas seções seguintes, onde apresentamos o arcabouço Gust, teremos oportunidade de mostrar diferentes formas, não-triviais, de combinar os componentes de sistemas MapReduce, incluindo algoritmos iterativos e com múltiplas rodadas de mapeamento e redução. Essa característica de composicionabilidade é uma das principais características do arcabouço para sistemas MapReduce proposto nesta Tese de Doutorado, que oferece oportunidades para contornar dificuldades expressivas relatadas na literatura em relação a outros arcabouços dentro do estado-da-arte.

5.3 O Arcabouço Gust

Na revisão apresentada no Capítulo 3, foram levantados um conjunto de arcabouços para processamento paralelo de dados em larga escala, com ênfase em grafos grandes. Muitas dessas soluções tem demonstrado seu potencial de escalabilidade e desempenho, ao mesmo tempo em que introduzem modelos de programação para permitir a escrita de algoritmos em grafos, de modo a reduzir ou tornar mais simples as preocupações com requisitos de computação paralela por parte do desenvolvedor, onde, cada solução, tais como os arcabouços MapReduce e suas variantes, parte de seu conjunto particular de requisitos. Dentro desse contexto, emergiram as motivações para este trabalho, quando há potencial para uma solução alternativa baseada em componentes paralelos e sob a abstração de nuvem computacional, que ofereça uma estrutura de componentes genérica e que possa comportar características encontradas nos arcabouços atualmente existentes. Além disso, que ofereça alternativas para contornar as limitações de arcabouços MapReduce e Pregel, lidando tanto com requisitos de *hardware* quanto de *software*.

5.3.1 Componentes de Sistemas Gust: Visão Geral

A denominação Gust é um acrônimo para ***G**raph **U**pon **S**helf archi**T**ecture*, bem como uma referência aos fenômenos naturais conhecidos como *gust front* (KNUPP, 2006)[2], fortes fluxos de ar associados à formação de nuvens prateleira (*shelf clouds*).

Na plataforma HPC Shelf, Gust representa um arcabouço de componentes para construção de sistemas de computação paralela para o processamento paralelo em larga escala de grafos grandes, oferecendo componentes que permitem controlar grafos, seus vértices e arestas, bem como criar fluxos de computação que satisfaçam os requisitos de algoritmos escritos para esse tipo de processamento. Por exemplo, pode suportar algoritmos escritos sobre os paradigmas MapReduce ou Pregel, ou ainda encadeando duas ou mais sequências de execução nesses modelos. Ou ainda definindo etapas computacionais adicionais (ou *fases*), que satisfaçam requisitos específicos para um dado algoritmo, como a aplicação da função `compute` do modelo Pregel seguida por uma *combinação* (*combine*) com o propósito de diminuir volume de dados nas trocas de mensagens subsequentes.

Por exemplo, o Algoritmo 4 do Capítulo 3, referente a uma enumeração de triângulos, apresenta um cenário onde duas etapas de mapeamento e outras duas etapas de redução,

[2] Frente de rajada.

alternadas, são necessárias. No *workflow* de um sistema de computação paralela Gust que implemente esse algoritmo sobre a HPC Shelf, essas quatro etapas podem ser representadas através do encadeamento de regimentos de componentes MAPPER e GUSTY, onde esse último representa uma especialização do componente REDUCER especializado para as necessidades de algoritmos em grafos. As entradas e saídas de dados desses times de componentes são interligadas por conectores SPLITTER e SHUFFLER.

O diferencial de GUSTY em relação ao REDUCER está nas suas ferramentas adicionais incluídas em GUSTYFUNCTION, uma especialização de REDUCEFUNCTION para atender requisitos de processamento de grafos, e que será vista com maiores detalhes nas próximas seções. Entretanto, por ser também uma função de redução, ao desenvolvedor é opcional o uso das ferramentas inclusas nessa função. Ou seja, o fluxo no sistema pode também se comportar como um MapReduce tradicional, usando GUSTYFUNCTION como se fosse uma REDUCEFUNCTION.

Todavia, quando não há necessidade de um agente de mapeamento em um fluxo de processamento de grafos, escrever funções identidade de *mapeamento* frequentemente consiste em um trabalho desnecessário, o que justifica no modelo Pregel existir apenas a função `compute`, onde a entrada é um vértice com suas mensagens recebidas, e a saída é um conjunto de mensagens com destino aos vizinhos. No GUST, essa lógica também é possível, sendo GUSTYFUNCTION capaz de comportar-se de forma equivalente a `compute`. Assim, caso o desenvolvedor não tenha necessidade de escrever uma função de mapeamento (MAPFUNCTION), o sistema fornece uma implementação genérica (padrão), que alimenta o SHUFFLER subsequente, simplesmente repassando pares *chave/valor* recebidos do SPLITTER. Isso faz com que o SHUFFLER agrupe *valores* em *chaves* únicas, para entregar ao GUSTY um conjunto de *valores* para uma mesma *chave*.

De fato, no Gust, a estrutura de programação de um algoritmo consiste nos códigos realizados dentro de um componente GUSTY, em GUSTYFUNCTION, que recebe pares *chave/valor* e emite um ou mais pares *chave/valor*, podendo ser iterativo caso exista um fluxo repetitivo. Para focalizar nessa parte computacional, à interligação ordenada dos componentes MAPPER[3], SHUFFLER[4] e GUSTY[5] demos o nome de `fase` MAPGUSTY, como destacado na Figura 34 no círculo pontilhado *Instância Exemplo*. Vê-se que existe um controle por parte do componente Workflow, através de *bindings* de ações, a fim de sincronizar as ações `READ_CHUNK`, `PERFORM` e

[3]Computação padrão no arcabouço.
[4]Computação padrão no arcabouço.
[5]Computação do algoritmo ou parte dele.

CHUNK_READY dos componentes SPLITTER, MAPPER, SHUFFER e GUSTY.

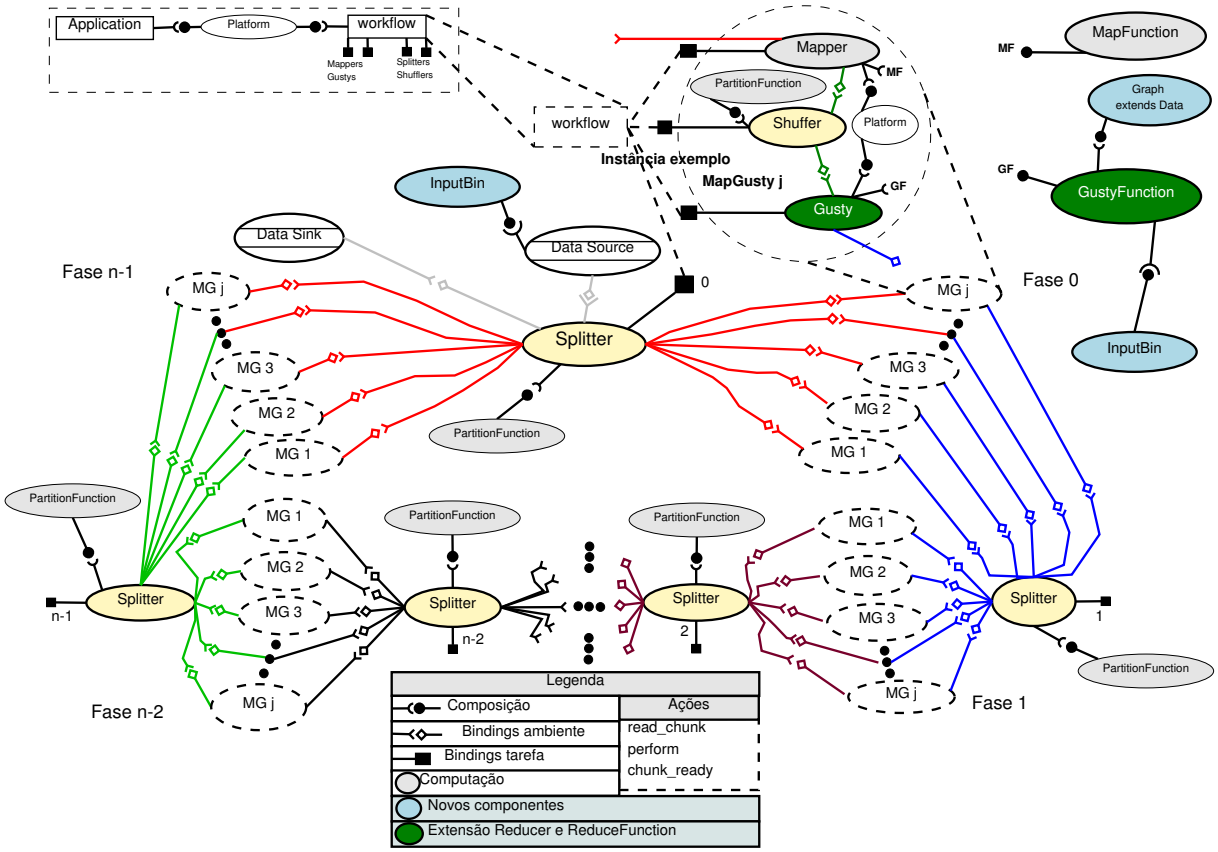


Figura 34 – Arquitetura Modelo: Componentes Gust, para *n* fases de processamento.

Em um fluxo de fases MAPGUSTY, cada fase deve conter um tipo de saída compatível com o tipo de entrada da fase subsequente, o que pode ser definido no contrato contextual, compatibilizando *bindings* de ambiente que ligam computações através de conectores. Na Figura 34, as cores de *bindings* indicam compatibilidade. Isso permite estabelecer um número finito de $n-1$ fases, que potencialmente não compartilham a mesma plataforma. A arquitetura descreve ainda todos os componentes que estão disponíveis para um processamento Gust. Entretanto, desde que existam *bindings* compatíveis, na maioria dos casos poucos *blocos de construção* são necessários. Além dos componentes GUSTY e GUSTYFUNCTION, são destacados os novos componentes, INPUTBIN e GRAPH. Nas próximas seções focalizamos nas descrições dos novos componentes e demais extensões ao arcabouço MapReduce.

5.3.2 INPUTBIN*: Particionamento inicial do grafo*

INPUTBIN é um componente abstrato do arcabouço GUST utilizado em dois momentos: *no início da computação*, envolvido na extração da estrutura do grafo, e *na construção de uma instância* para GRAPH, usada em GUSTYFUNCTION.

No início da computação, INPUTBIN atua no particionamento do grafo por corte de vértices. Para isso, recebe como entrada o endereço de um arquivo texto que representa um grafo. Atualmente, seu único componente concreto está programado para receber arquivos contendo linhas na forma i j w, onde i e j são identificadores numéricos de vértices maiores que zero, formando uma aresta $e[i, j]$ com peso facultativo w. Como exemplo, considere o arquivo *grafo.e*. Ao receber a localização deste arquivo, INPUTBIN verifica se existe seu cabeçalho, denominado *grafo.e.head*, o qual contém uma única linha com os valores *vsize*, *esize* e P, respectivamente representando a quantidade de vértices, arestas e partições. Se o cabeçalho não existir, haverá uma sobrecarga durante a leitura do arquivo *graph.e*, criando-se ao final o cabeçalho *graph.e.head* com as devidas instruções, usando um valor padrão para P. Além disso, INPUTBIN verifica se existe uma estratégia de particionamento pronta, buscando o arquivo *grafo.e.part.P*, onde P é o número de partições em *grafo.e.head*. Uma estratégia pronta consiste em cada linha do arquivo *grafo.e.part.P* ser um vértice, e o conteúdo da linha, o número da partição que ele pertence. Por exemplo, se a linha 7 contém o valor 10, significa que o vértice 7 está na partição 10. Caso não exista uma estratégia pronta, INPUTBIN utiliza seu algoritmo padrão, baseado em *hash*. Nos parágrafos a seguir, é descrito como são alocadas as arestas.

Como mencionado anteriormente, usa-se o corte de vértices, o que significa atribuir arestas às partições, ao invés de vértices. Para isso, se $i \bmod P = X$ e $j \bmod P = Y$, as partições X e Y receberão $e[i, j]$. Porém, o j é fantasma na partição X, enquanto o i é fantasma na partição Y. Um vértice é dito fantasma quando a partição não é responsável por ele, ou seja, X não é responsável por j. Por exemplo, uma máquina que receba arestas da partição X é responsável apenas pela integridade de vértices dessa partição. Portanto, vértices fantasmas estão ali apenas como espelho de um vértice principal, localizado em outra partição. Por outro lado, a responsabilidade pela integridade do peso da aresta (w) fica com o vértice de menor identificador numérico.

Ao ler e processar o *graph.e*, INPUTBIN vai criando tabelas de partição e instâncias denominada `InputBinInstance`, as quais representam *chunks* no sistema MapReduce. Essas tabelas também poderiam ser feitas pelo componente PARTITIONFUNCTION. Entretanto, optamos

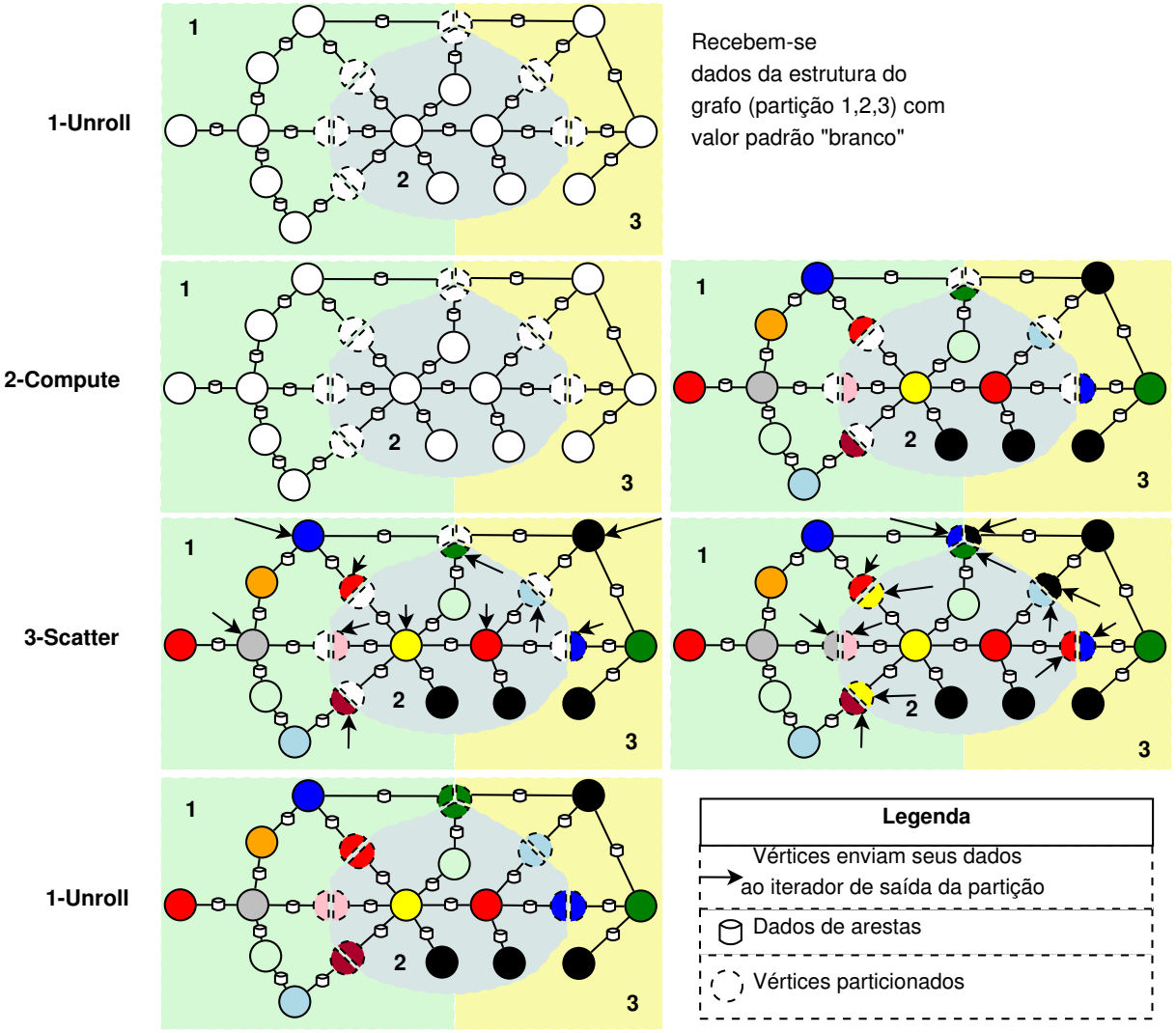


Figura 35 – Esquema Gust de partição (através de INPUTBIN), bem como computação e distribuição de dados (através de GUSTY,GUSTYFUNCTION,GRAPH)

por não usar nesse momento o PARTITIONFUNCTION, com a finalidade de explorar o particionamento feito pela ferramenta Metis (KARYPIS; KUMAR, 1995) em nosso estudo de caso (para balanceamento de carga), pois Metis fornece um arquivo *graph.e.part.P* compatível com INPUTBIN, que pode ser comparado com o algoritmo padrão, baseado em *hash*. Além disso, para eventuais trabalhos futuros, pode-se também comparar outras estratégias complexas de particionamento, apenas informando o arquivo *graph.e.part.P*. Assim, cada *chunk* `InputBinInstance` contém os seguintes vetores: `source` (armazena i), `target` (armazena j), `weight` (armazena w) e `partition` (armazena a tabela de partição, sendo uma cópia em memória do arquivo *graph.e.part.P*).

A Figura 35 apresenta o exemplo de como cada vértice recebe o corte. Vê-se que há três partições, com as cores de fundo definindo a localização de cada vértice, bem como dos

dados das arestas. Normalmente, cada partição é destinada para uma unidade de processamento, computando GUSTYFUNCTION, onde está o método `unroll`, o qual deve desempacotar todos os *chunks* `InputBinInstance` para construção da estrutura de cada subgrafo GRAPH (vinculado à partição), usado pelo programador para escrever seu algoritmo. Portanto, um vértice que recebe o corte está em mais de uma partição, existindo *espelhos* e um vértice *principal* responsável pela consistência. Qualquer alteração no vértice *principal* deve ser replicada aos demais. Além disso, vizinhos do vértice *principal*, que estão em outras partições, devem também replicar suas alterações para o vértice *principal*. Na Figura 35, no item `2-Compute`, os vértices *espelhos* são aqueles que não receberam cores na segunda coluna, onde a cor representa consistência. Contudo, a distinção entre quem é responsável pela consistência dos dados fica mais clara com a explicação dos métodos `unroll`, `compute` e `scatter`, os quais serão abordados dentro da seção que descreve GUSTYFUNCTION. Antes de tratarmos a respeito desse componente, descrevemos o componente GRAPH, aninhado ao GUSTYFUNCTION.

```
interface IGraphInstance<V, E, TV, TE>
  where V:IVertex where E:IEdge<V> where TE: IEdgeInstance<V, TV>
  bool addVertex (TV vertex);
  bool addEdge(TE edge);
  TE addEdge(TV source, TV target);
  ICollection<TE> getAllEdges(TV source, TV target);
  TE getEdge(TV source, TV target);
  bool containsEdge(TV source, TV target);
  ICollection<TV> neighborsOf (TV vertex);
  ICollection<TE> edgesOf (TV vertex);
  TE removeEdge(TV source, TV target);
  bool removeVertex (TV vertex);
  bool containsEdge(TE edge);
  bool containsVertex (TV vertex);
  int degreeOf(TV vertex);
  ICollection<TV> vertexSet ();
  IEnumerator<TE> edgeSet();
  void noSafeAdd (TE edge);
  void noSafeAdd (TV source, TV target);
  void noSafeAdd (TV source, TV target, float weight);
  int countE();
  int countV();
  IDataContainerInstance<V, E> DataContainer { get; set; }
```

```
24 interface IDirectedGraphInstance<V, E, TV, TE>: IGraphInstance<..>
25   int inDegreeOf(TV vertex);
26   int outDegreeOf(TV vertex);
27   ICollection<TE> incomingEdgesOf(TV vertex);
28   ICollection<TV> incomingVertexOf(TV vertex);
29   ICollection<TE> outgoingEdgesOf(TV vertex);
30   ICollection<TV> outgoingVertexOf(TV vertex);
```

Código 9 – Principais métodos de `IGraphInstance` e `IDirectedGraphInstance` (C#)

5.3.3 GRAPH*: Controlador de Subgrafos*

GRAPH, atualmente escrito em C#, originou-se como uma extensão de parte do projeto JGraphT (NAVEH; SICHI, 2003) (escrito em Java), voltado para algoritmos seriais em grafos. Foram necessárias otimizações para suportar grafos grandes, bem como ambiente paralelo e o próprio modelo Hash. Do JGraphT, restaram alguns métodos da sua interface `Graph`, especialmente os que estão acima da linha 18 no Código 9.

Ao lidar com grafos grandes, para cada método com retorno `ICollection<T>`, há também um método que retorna um iterador `IEnumerator<T>`, sendo opcional ao desenvolvedor. Isso ocorre porque as coleções (de vértices vizinhos ou arestas adjacentes) são criadas no momento que o método é chamado, diferente do JGraphT, o qual retorna uma referência à coleção já armazenada em memória. Em contrapartida, GRAPH ganha espaço em memória, de modo que o programador possa ainda fazer cache de uma coleção retornada, ou usar um iterador para ler apenas uma vez os dados. Por exemplo, uma aresta é um objeto que implementa a interface `IEdgeInstance`, a qual possui além do peso dois vértices: `source` e `target`. Diante disso, não são armazenadas instâncias de arestas, pois com base nos vértices e peso, os objetos arestas são criados quando necessários, usando o padrão de projeto *Factory*.

Para usar GRAPH, o desenvolvedor recebe uma instância de objeto do tipo `IGraphInstance`, a qual permite controlar vértices e arestas particionados através de INPUTBIN. Essa instância oferece métodos matemáticos que buscam suportar a teoria dos grafos, incluindo suporte a *grafos simples* e *multigrafos*, conforme vistos nos códigos 9 e 10. Para isso, um componente aninhado do tipo DATACONTAINER é responsável por toda a concentração de dados, sendo um componente serializável, ou seja, que pode ser transmitido através da rede. Para

aceitar *multigrafos*, DATACONTAINER pode ser configurado com as propriedades booleanas `AllowingMultipleEdges` e `AllowingLoops`. Além disso, na criação de uma instância GRAPH com o método `Graph.newInstanceT<T>`, onde `T` pode assumir qualquer valor, de tipos primitivos ou objetos, o tipo de identificador de vértice é genérico. De fato, `T` pode ser inclusive um *subgrafo*, embora os tipos primitivos como `int`, `uint`, `long` tenham reduzido significativamente o consumo de memória.

Nome do Parâmetro	Var.	componentes	Limite Contextual
input_key_type	*TKey*	GUSTY GUSTY_FUNCTION	DATA
input_value_type	*TValue*	GUSTY GUSTY_FUNCTION	DATA
output_key_type	*OKey*	GUSTY GUSTY_FUNCTION	DATA
output_value_type	*OValue*	GUSTY GUSTY_FUNCTION	DATA
reduce_function	*RF*	GUSTY	GUSTYFUNCTION
graph_type	*G*	GUSTY GUSTY_FUNCTION	DATA
graph_input_format	*GIF*	GUSTY GUSTY_FUNCTION	INPUTBIN
vertex_type	*V*	EDGE DATA_CONTAINER GRAPH	VERTEX
edge_type	*E*	DATA_CONTAINER GRAPH	EDGE
container	*CTN*	GRAPH	DATACONTAINER

Tabela 11 – Parâmetros de contexto dos componentes GUSTY, GUSTYFUNCTION, GRAPH, DATACONTAINER e EDGE

A Tabela 11 apresenta todos os parâmetros de contexto usados em DATACONTAINER, GRAPH, GUSTY e GUSTYFUNCTION, onde existem componentes como VERTEX e EDGE. A principal ideia com o uso do componente GRAPH é justamente que ele trabalhe com os tipos primitivos identificadores de objetos, mas relacionando-se com o contexto. Por exemplo, o componente VERTEX, cuja instância é `IVertexInstance`, contém um identificador `ID`, do tipo inteiro. VERTEX pode portanto ser uma *chave* `TKey` no sistema MapReduce, cujo `ID` está ligado ao tipo `T` estabelecido na instância de GRAPH. VERTEX também pode representar um subgrafo, onde seu identificador define o número da partição. Nesse caso, `TValue` representa um conjunto de dados, como um vetor de inteiros, os quais estão relacionados com o tipo `T` definido na instância do grafo. Essa opção consiste em uma *granularidade* mais grossa, onde a comunicação pode ser realizada com blocos grandes de dados, após o algoritmo computar

seus devidos cálculos. Por exemplo, considere `TValue` como um componente C que possui dois vetores: `vertexs` e `ranks`. No primeiro, estão os inteiros que identificam vértices. No segundo, estão os valores de ponto flutuante que representam *ranks* para os vértices em um algoritmo *page_rank*. Isso consiste em apenas dois objetos internos ao `TValue`, potencialmente reduzindo consumo de memória. O Código 10 é um exemplo de como é criada uma instância com vértices do tipo `uint`. Além disso, apresenta o uso dos métodos de adição de vértices e arestas, especialmente quando o *chunk* do tipo `InputBinInstance` é recebido, oferecendo os vetores `source` e `target`.

```
1 void graph_creator(InputBinInstance chunk)
2    if (graph == null)
3       graph = Graph.newInstanceT<uint>();
4    graph.DataContainer.AllowingLoops = false;
5    graph.DataContainer.AllowingMultipleEdges = false;
6    for (int i = 0; i < chunk.size; i++)
7       graph.addVertex(chunk.source [i]);
8       graph.addVertex(chunk.target [i]);
9       graph.addEdge(chunk.source [i], chunk.target [i]);
```

Código 10 – Inserção de vértices e arestas no `Graph`

5.3.4 GUSTY *e* GUSTYFUNCTION

GUSTY e GUSTYFUNCTION são componentes intimamente relacionados, de modo que o primeiro possui um componente aninhado do tipo do segundo em sua arquitetura de sobreposição. Os parâmetros de sua assinatura contextual podem ser vistos na Tabela 11, onde DATA é o tipo base de componente da espécie estrutura de dados na HPC Shelf. É importante destacar que o parâmetro ***graph_type*** não é limitado a GRAPH, o que permite a GUSTY e GUSTYFUNCTION aceitar outras versões, ou mesmo ignorar esse parâmetro, permitindo trabalhar com *chaves/valores* do MapReduce tradicional. Isso deixa o desenvolvedor livre para realizar suas otimizações, manipulando os tipos de chave e valores, bem como o tipo do grafo, conforme suas necessidades. A Figura 36 apresenta GUSTYFUNCTION e GRAPH modelados no HPE. Todos os componentes aninhados a GUSTYFUNCTION, definidos a seguir, são acessíveis na configuração de GUSTY:

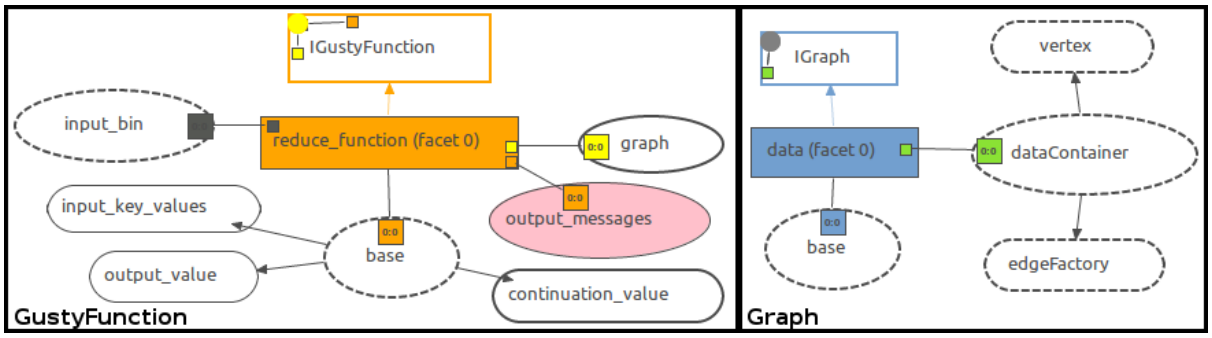


Figura 36 – Componentes GUSTYFUNCTION e GRAPH modelados no HPE

- `input_bin` é do tipo INPUTBIN, com instâncias `InputBinInstance` recebidas através de pares *chave/valor*, onde a chave é a partição da instância;
- `input_key_values` é um iterador de entrada, composto por uma *chave* e vários *valores*;
- `output_value` produz instâncias de pares *chave/valor* como resultado de um determinado processamento;
- `output_messages` é um iterador ou *buffer* de saída para um conjunto de pares *chave/valor*, os quais são distribuídos através de conectores, para novos processamentos;
- `continuation_value` representa um *valor* de saída incremental. Ou seja, no recebimento de uma *chave* vinculada ao *valor* ocorre sua recuperação para incremento, tal como uma soma cumulativa;
- `graph` é o componente GRAPH, funcionando como um *subgrafo* paralelo, o qual é alimentado pelo particionamento em `InputBinInstance`.

Dadas as definições dos componente aninhados, no modelo de programação aplicado ao Gust, todo algoritmo escrito em GUSTYFUNCTION deve conter no mínimo três métodos: `unroll`, `compute` e `scatter`, descritos em seguida.

- `unroll`: método destinado a coletar dados em GUSTYFUNCTION, chamado pelo agente GUSTY sempre que há um novo par *chave/valor* no iterador de entrada. Permite o incremento de *valores* que pertencem a uma mesma *chave*, uma vez que pares são produzidos de forma assíncrona. Ou seja, quando o programador insere um par no *buffer* de saída (disponibilizado no `scatter`) esse par já está disponível para consumo na próxima iteração ou fase GUSTY, sendo recuperado e incrementado nesse `unroll` subsequente. No modelo GAS (*Gather*, *Apply*, and *Scatter*) do POWERGRAPH (GONZALEZ *et al.*, 2012), `unroll` pode ser visto como um `gather` que usa o método cumulativo denominado `sum`.
- `compute`: é o método chamado após todo o `unroll` ser concluído, sendo onde o desenvol-

vedor insere a lógica do seu algoritmo. Com chamada controlada por GUSTY, é importante destacar duas possibilidades de implementação concreta para GUSTY. A primeira forma consiste em `compute` ser chamado apenas uma vez na iteração. Isso faz com que o desenvolvedor deva iterar nos vértices do grafo através do retorno do método `graph.vertexSet`. O programador possui acesso irrestrito e pode focar tanto no *subgrafo* como nos vértices individuais iterados. A segunda forma consiste no parâmetro de contexto ***graph_type*** ser limitado a GRAPH, bem como ***input_key_type*** e ***output_key_type*** serem limitados a VERTEX, sendo uma forma mais restritiva. Isso faz com que GUSTY tenha acesso direto aos componentes aninhados `vertex` e `edgeFactory`. Nesse caso, GUSTY deve chamar, para cada vértice, a função `compute`. Além disso, deve fornecer dados e vizinhança, para que o programador tenha foco exclusivo naquele vértice em processamento, embora ele ainda tenha acesso a `graph`. Os autores chamam esse modelo de "vertex-centric", realizando o mesmo que a função `Apply` do modelo `GAS`, ou o mesmo que a função `compute` do modelo Pregel, que é ainda mais restritivo, pois apenas fornece dois *buffers* de mensagens (limitados a *digrafo*), sendo *entrada* e *saída*. Em nosso estudo de caso, usamos apenas a primeira forma, pois nos permitiu atingir resultados de desempenho competitivos com o estado da arte.

- `scatter`: método usado para enviar dados ao iterador de saída (`output_messages`), sendo chamado uma única vez pelo GUSTY. Ativado após a etapa `compute`, supõe-se que o desenvolvedor já tenha realizado suas computações otimizadas, possivelmente com tabelas `hash` e tenha explorado todo o poder de processamento do *hardware*, para finalmente liberar seus dados para partições vizinhas.

Embora com suas devidas finalidades, os três métodos discutidos podem ser usados conforme requisito de algoritmo ou mesmo técnica do desenvolvedor. Por exemplo, o iterador de saída `output_messages` está sempre disponível, podendo ser usado inclusive dentro de `unroll`, caso essa lógica seja útil. A Figura 35 apresenta um típico caso de como ocorre uma iteração GUSTY usando esses métodos. Vê-se que, no primeiro `unroll`, `graph` está vazio e deve ser criado através de instâncias do tipo `InputBinInstance`, recebidas com o valor do vértice na cor branca (padrão). Com o GRAPH devidamente instanciado e `unroll` finalizado, inicia-se `compute`, que deve construir a cor de cada vértice a partir do branco. Vértices cuja consistência não é de responsabilidade da partição permanecem brancos. Após isso, o `scatter` é ativado. Como o corte, vértices *principais* devem replicar suas cores aos *espelhos*. Além disso, vértices

adjacentes ao *principal* e que estejam em outras partições devem inserir suas cores no iterador de saída da partição. Dessa forma, no próximo `unroll`, é possível recolher a cor quando ela chegar ao iterador de entrada, na forma de par *chave/valor*. A partir disso, todos conhecem as cores de todos, possibilitando cálculos de novas cores a partir de cores atuais. Nesse contexto abstrato, cores podem ser qualquer tipo de dados, como um *rank* (*PageRank*), um caminho de um vértice fonte até algum vértice destino (*SSSP*), o identificador de um triângulo (*TriangleCount*). Esses referidos algoritmos são significativamente explorados por plataformas que processam grafos grandes. Portanto, fazem parte do estudo de caso deste trabalho, sendo explicados nas seções seguintes.

5.4 Estudos de Caso

As seções que se seguem apresentam como três conhecidos algoritmos, discutidos no Capítulo 3, tem sido implementados sobre o Gust como estudos de caso com a finalidade de avaliá-lo. São eles: enumeração de triângulos (Seção 3.1.3.3), ranqueamento de páginas (Seção 3.1.3.1) e SSSP (Seção 3.1.3.2).

5.4.1 Enumeração de Triângulos

Um triângulo T é um subgrafo de $G = (V, E)$ composto por três vértices: $v_i, v_j, v_k \in V$, formando um *ciclo*, com $[v_i, v_j], [v_j, v_k], [v_i, v_k] \in E$, onde $v_i < v_j < v_k$. A implementação deste algoritmo possui três etapas, também referidas como *super-passos* ou interações.

No Código 11, o super-passo atual é definido pela variável `this.Superstep`. No primeiro passo, um vértice v_i procura vizinhos v_j e envia uma mensagem v_i para cada v_j. Para visualizar isso, acompanhe a função `compute`, no super-passo 0. A entrada é um grafo simples não direcionado. Porém, a instância usada é um grafo direcionado, pois faz parte da lógica, onde queremos encontrar vértices na forma $i < j$, buscando vizinhos de saída. Para isso, cria-se o grafo na forma $i < j$[6], de modo que a chamada `graph.outgoingVertexOf(i)` retorne sempre j maior que i. Entretanto, para processar vértices i, eles devem ser de responsabilidade da partição local. Verifica-se isso através da função `!isGhost(i)`, que retorna verdade caso o vértice seja espelho (ou fantasma) do principal. Além disso, no caso de j ser espelho, uma operação *push* deve ser realizada, passando a mensagem i para j, com destino ao super-passo 1. Porém, caso j não seja espelho, é possível rastrear seus vizinhos em busca de k, para finalmente checar

[6]Com a chamada `graph.addEdge(i<j?i:j, i>j?i:j)`

se *k* também é vizinho de *i*. Portanto, no super-passo 0, é possível contar todos os triângulos da partição local, restando a contagem global. No super-passo 1, *j* procura por vizinhos *k* e envia uma mensagem *k* para cada *i* recebido anteriormente, através de `pushValue(i,k)`, função preparada para que *i* possa receber múltiplos *k*, uma vez que *i* pode ser o menor vértices de vários triângulos. É importante notar que isso é feito dentro de `unroll`, o que permite operações assíncronas, embora neste algoritmo aguardou-se o `scatter` para liberação de blocos de dados. Finalmente, no super-passo 2, é possível contabilizar triângulos checando se *k* é vizinho de *i*. O `scatter` é chamado nos três passos para enviar blocos de dados adquiridos através de `getOutputBlockArray<int,int>(pid)`, os quais são alimentados por `pushValue`. Além disso, destaca-se que `pid` é o número da partição destino, `item` é o objeto *chave/valor*, `output_value` é o iterador distribuído de saída que recebe `item` através de `put(item)`.

```
1  void unroll() {
2    while (input_values.fetch_next (out o))
3      if (this.Superstep == 0)
4        this.graph_creator(o);
5      else {
6        BlockArray<int,int> block = (BlockArray<int,int>) ((IDataTriangleInstance)o).Value;
7        if (this.Superstep == 1)
8          for(int b = 0; b < block.SIZE ; b++) {
9            int j = block.Keys[b];
10           foreach (int k in graph.outgoingVertexOf (j))
11             foreach (int i in block.Values[b])
12               pushValue (i, k);
13         }
14       else if (this.Superstep == 2)
15         for(int b = 0; b < block.SIZE ; b++) {
16           int i = block.Keys[b];
17           foreach (int k in block.Values[b])
18             if (graph.outgoingVertexOf (i).Contains (k))
19               triangleCount++;
20         }
21     }
22 }
23 void compute() {
24   if (this.Superstep == 0)
25     foreach (int i in graph.vertexSet ())
26       if (!isGhost (i)) {
27         ICollection<int> outgoing_i = graph.outgoingVertexOf (i);
28         foreach (int j in outgoing_i)
29           if (isGhost (j))
30             pushValue (j, i);
31           else
32             foreach (int k in graph.outgoingVertexOf (j))
```

```
33                if (outgoing_i.Contains (k))
34                  triangleCount++;
35          }
36  }
37  void scatter() {
38    for (int pid = 0; pid < partitions; pid++)
39      if (pushs [pid].Count > 0 || (this.Superstep == 2)) {
40        if ((this.Superstep == 2)) pid = partitions = partid;
41        IKVPairInstance<IVertexBlock,IDataTriangle> item = Output_messages.createItem ();
42        ((IVertexBlockInstance)item.Key).PId = (byte)pid;
43        ((IDataTriangleInstance)item.Value).Value = getOutputBlockArray<int,int>(pid);
44        ((IDataTriangleInstance)item.Value).Count = triangleCount;
45        output_value.put (item);
46      }
47  }
```

Código 11 – Métodos unroll, compute, scatter para TriangleCount

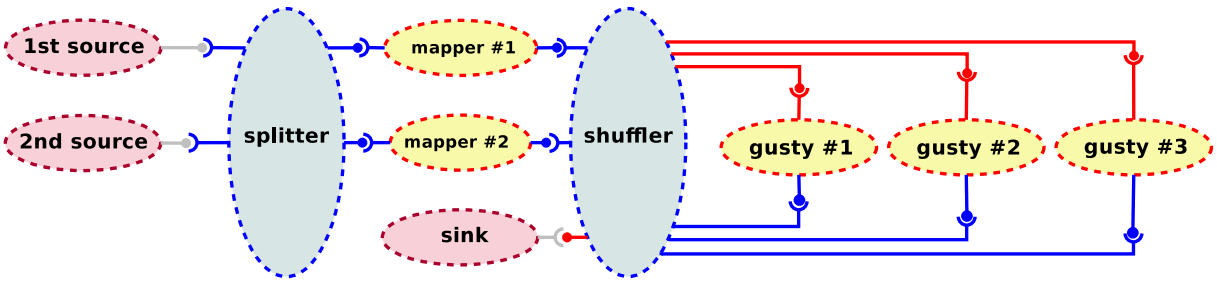


Figura 37 – *TriangleCount* nas plataformas virtuais #1,#2,#3

A Figura 37 mostra a arquitetura de um sistema de computação paralela para *contagem de triângulos*, empregando três agentes GUSTY colocados em plataformas virtuais distintas. As facetas coletoras do componente **splitter** leem os dados de entrada em duas fontes (**1st source** e **2nd source**). Os pares chave/valor são distribuídos entre as facetas fornecedoras, que irão alimentar dois *mappers* localizados em plataformas distintas (**#1** e **#2**). Os agentes de mapeamento, entretanto, aplicam apenas uma *função identidade*, ou seja, apenas repassarão os pares para as facetas coletoras do **shuffler**, as quais agruparão os valores por suas chaves e as distribuirão para as facetas fornecedoras, que irão alimentar três agentes GUSTY (**gusty #1**, **gusty #2** e **gusty #3**) localizados nas plataformas **#1**, **#2** e **#3**. Os pares de saída produzidos pelos componentes GUSTY alimentam as outras três facetas coletoras do **shuffler**, tornando as disponível para começar uma nova iteração. Após a terceira iteração, o **shuffler** repassará os pares de saída para **sink**.

Figura 38 – *TriangleCount (sem iteração, sem mappers)*

Uma vez que o algoritmo realiza exatamente três iterações, e as funções do *map* são identidades, a Figura 38 propõe uma arquitetura alternativa mais apropriada, não-iterativa, a qual conecta ambas as fontes de dados de entrada diretamente ao primeiro `shuffler`. Neste caso, haverá apenas o $Superstep_0$, de modo que cada agente processará um código compatível com sua fase. Isso demonstra a flexibilidade do arcabouço proposto para definição de arranjos entre os componentes para implementar o mesmo algoritmo, que satisfaçam requisitos de aplicações.

Parameter Name	***Triangle Enumeration***	***SSSP***	***PageRank***
input_key_type	VERTEXBLOCK	VERTEXBLOCK	VERTEXBLOCK
input_value_type	DATATRIANGLE	DATASSSP	DATAPGRANK
intermediary_key_type	VERTEXBLOCK	VERTEXBLOCK	VERTEXBLOCK
intermediary_value_type	DATATRIANGLE	DATASSSP	DATAPGRANK
reduce_function	TRIANGLECOUNT[...]	SSSP[...]	PAGERANK[...]
output_key_type	VERTEXBLOCK	VERTEXBLOCK	VERTEXBLOCK
output_value_type	DATATRIANGLE	DATASSSP	DATAPGRANK

Tabela 12 – Argumentos de contexto para os componentes dos sistemas de computação paralela dos estudos de caso

A Tabela 12 apresenta os argumentos de contexto dos componentes do sistemas de computação paralela de enumeração de triângulos, bem como dos demais sistemas de computação paralela das seções seguintes. DATATRIANGLE armazena triângulos de um subgrafo particionado e VERTEXBLOCK (Herdado de VERTEX) é a chave. Assim, a troca de mensagens envolve blocos de dados para o subgrafo, que é visto como um bloco de vértices.

```
1  void unroll() {
2    while (ivalues.fetch_next (out o)) {
3      IDataPGRANKInstance VALUE = (IDataPGRANKInstance)o;
4      if (this.Superstep == 0) //cria grafo com valor padrao 1
5        this.graph_creator (o);//ou seja, getCurrentValue(v)=1
6      else {
7        Block<int,double> block = (Block<int,double>)VALUE.Value;
8        for (int b = 0; b < block.SIZE; b++)
9          setValueGhost (block.Keys[b], block.Values[b]);//atualiza ranks dos espelhos
10     }
11   }
12 }
13 void compute() {
```

```
14    if (this.Superstep < MAX_ITERATION)
15      foreach (int v in graph.vertexSet ())
16        if (!isGhost (v)) {
17          double sum = 0;
18          foreach (int w in graph.incomingVertexOf (v))
19            sum += getCurrentValue (w) / graph.outDegreeOf (w);//pulling vertex w
20          setValue (v, 0.15 + sum * 0.85);//define proximo rank e faz push nos vizinhos
21        }
22  }
23  void scatter() {
24    for (int pid = 0; pid < partitions; pid++) {
25      IKVPairInstance<IVertexBlock,IDataPGRANK> item = Output_messages.createItem ();
26      if (pushed (pid).Count > 0) {
27        ((IVertexBlockInstance)item.Key).PId = (byte)pid;
28        ((IDataPGRANKInstance)item.Value).Value = getOutputBlock<int,double> (pid);
29        output_value_instance.put (item);
30      }
31      else if (this.Superstep == MAX_ITERATION) {
32        ((IVertexBlockInstance)item.Key).PId = (byte)partid;
33        ((IDataPGRANKInstance)item.Value).Ranks = currentData (partid);
34        output_value_instance.put (item);
35        break;
36      }
37    }
38  }
```

Código 12 – Métodos unroll, compute, scatter para PageRank

5.4.2 Ranqueamento de Páginas (PageRank)

O sistema de computação paralela proposto para implementação do algoritmo *PageRank* utiliza grafo direcionado. Sua implementação é apresentada no Código 12, e os argumento de contexto de seus contratos contextuais encontram-se na Tabela 12.

O *rank* dos vértices são definidos segundo a equação $r(v) = 0,15 + 0,85 \sum_{w \in B_v} \frac{r(w)}{|N_w|}$, onde $r(v)$ e $r(w)$ são respectivamente os *ranks* de v e w, B_v o conjunto de vértices que apontam para v, e $|N_w|$ é o grau de saída de w. Na primeira iteração, o grafo é inicializado com valor padrão 1, para cada *rank* de vértice. O número de iterações é configurável por uma variável, chamada `MAX_ITERATION`. Geralmente, seu valor é 30. Enquanto não é atingido o limite de iteração, a função `compute` é executada, fazendo com que cada *rank* convirja para seu devido valor. Essa implementação reflete o que ocorre na Figura 35, onde, em cada execução de `unroll`, a cor de

cada espelho é atualizada. No caso de *ranks*, espelhos se atualizam através de `setValueGhost`, enquanto seu principal é atualizado com `setValue`. Além disso, `getCurrentValue` retorna o *rank* do início da iteração, ou seja, o valor 1 é retornado no super-passo 0. No método `scatter`, `getOutputBlock<int,double>(pid)` retorna o bloco de dados de vértices espelhos, os quais receberam mensagens através do valor previamente passado para `setValue`. Esses blocos são constituídos por vetores, parametrizados de forma genérica (parâmetros `IK` e `IV`, no Código 13), especialmente para diminuir o tamanho dos dados a serem transferidos quando seus tipos são primitivos. *PageRank* tem sido implementado em conformidade com a arquitetura iterativa da enumeração de triângulos, conforme Figura 37. O componente DATAPGRANK (argumento ***value_type***) é usado para levar dados de uma partição para outra.

```
[Serializable]
class Block<IK,IV> {
  int SIZE = 0;
  IK[] Keys;
  IV[] Values;
  Block(int size) {
    this.SIZE = size;
    this.Keys = new IK[size];
    this.Values = new IV[size];
  }
}
```

Código 13 – Classe `Block`

```
void unroll() {
  while (ivalues.fetch_next (out o)) {
    IDataSSSPInstance VALUE = (IDataSSSPInstance)o;
    if (this.Superstep == 0) {
      this.graph_creator (o);
      setValue (SOURCE, 0f);
    }
    else {
      halt_sum += VALUE.Halt;
      Block<int,float> bin = (Block<int,float>)VALUE.Value;
      for (int k = 0; k < bin.SIZE; k++) {
        float v_dmin = bin.Values [k];
        if (v_dmin > 0) {
          int v = bin.Keys[k];
          Queue<int> queue = new Queue<int> ();
          if (getValue(v)>v_dmin) {
            setValue(v,v_dmin);
            queue.Enqueue (v);
```

```
19              while (queue.Count > 0) {
20                v = queue.Dequeue ();
21                v_dmin = getValue (v);
22                foreach(IEdgeWeightedInstance<IVertex,int> e in g.edgesOf(v)) {
23                  int n = e.Source==v?e.Target:e.Source;
24                  if (getValue(n)>(e.Weight + v_dmin)) {
25                    setValue(n,(e.Weight + v_dmin));
26                    queue.Enqueue (n);
27                  }
28                }
29              }
30            }
31          }
32        }
33      }
34    }
35  }
36  void compute() {
37    if (this.Superstep == 0) {
38      int v = SOURCE;
39      if (graph.containsVertex (v)) {
40        Queue<int> queue = new Queue<int> ();
41        queue.Enqueue (v);
42        while (queue.Count > 0) {
43          v = queue.Dequeue ();
44          float v_dmin = getValue (v);
45          foreach(IEdgeWeightedInstance<IVertex,int> e in g.edgesOf(v)) {
46            int n = e.Source==v?e.Target:e.Source;
47            if (getValue(n)>(e.Weight + v_dmin)) {
48              setValue (n, (e.Weight + v_dmin));
49              queue.Enqueue (n);
50            }
51          }
52        }
53      }
54    }
55  }
56  void scatter() {
57    if (this.HALT) {
58      IKVPairInstance<IVertexBlock,IDataSSSP> ITEM = Output_messages.createItem ();
59      ((IVertexBlockInstance)ITEM.Key).PId = (byte) this.partid;
60      ((IDataSSSPInstance)ITEM.Value).Path_size = currentData(this.partid);
61      output_value.put (ITEM);
62    }
63    else
64      for (int pid = 0; pid < partitions; pid++) {
65          IKVPairInstance<IVertexBlock,IDataSSSP> ITEM = Output_messages.createItem ();
66          ((IVertexBlockInstance)ITEM.Key).PId = (byte) pid;
67          ((IDataSSSPInstance)ITEM.Value).Value = getOutputBlock<int, float> (pid);
```

```
68              ((IDataSSSPInstance)ITEM.Value).Halt = EMITTED ? 1 : 0;
69              output_value.put (ITEM);
70          }
71      }
```

Código 14 – Métodos unroll, compute, scatter para SSSP

5.4.3 SSSP (Single Source Shortest Path)

No sistema de computação paralela para implementar o algoritmo *SSSP*, o método `compute` contém código apenas para o super-passo 0, conforme implementação apresentada no Código 14. Usa-se a variável *SOURCE* como parametrização do *vértice fonte*, que tem valor fixo em 0. Todos os demais, inicialmente, possuem valor infinito. A partição que possua *SOURCE* dá o início à computação, de modo que *SOURCE* realiza uma iteração sobre seus vizinhos para que eles busquem atualizações de seus valores. Sempre que um vizinho atualizar seu valor, é colocado em fila. Além disso, sempre que ele sai da fila, deve realizar uma iteração para que seus vizinhos candidatos entrem na fila. Isso ocorre de forma encadeada, em todos os vértices do grafo.

As instruções `e.Weight` e `v_dmin` consistem respectivamente no peso da aresta adjacente e no valor do vértice desenfileirado *v*, onde `v_dmin` busca ser a menor distância de *v* até *SOURCE*. Quando há uma melhor opção para vizinhos de *v*, definem-se valores através do método `setValue`. Isso faz com que o `scatter` envie dados para todas as partições pertencentes ao grafo, especialmente porque não há um número fixo de iterações. Por esse motivo, no critério de finalização, a propriedade `Halt` de `IDataSSSPInstance` recebe 1 se a partição emissora estiver ativa, ou 0 caso contrário. De fato, partições ficam ativas se houver algum valor alterado pelo método `setValue`. Com isso, os *Halts* são somado no `unroll`, ativando a propriedade `this.HALT` se o somatório de *Halts* for 0. Ou seja, quando todos estiverem desativados, o iterador de saída é fechado e a computação finalizada.

Apesar desse código ser maior que os demais, suas instruções são semelhantes em `unroll` e `compute`, podendo inclusive serem reusadas. Deixa-se redundante apenas para destacar as computações isoladas de *SOURCE*, o qual é executado apenas no super-passo 0. Assim como na enumeração de triângulos, o *SSSP* está preparado para rodar na arquitetura iterativa expressa na Figura 37.

5.5 Considerações

A engenharia de software baseada em componentes de computação de alto desempenho permite que os arcabouços MapReduce e Gust descritos neste capítulo lidem com algumas limitações de tradicionais modelos, como MapReduce e Pregel, ou contribua com eles da seguinte forma:

- *adaptabilidade*: quando permitem computações iterativas ou o encadeamento de blocos que realizam sequências de MAPPER e REDUCER. Além disso, desde que existam conectores compatíveis, computações podem formar arranjos arbitrários, possibilitando representar situações adversas que possam vir a existir;
- *modularidade*: pela própria natureza da computação baseada em componentes, aplicações que se formam são modulares, adaptáveis e podem servir de base para formar novas aplicações. Além disso, usando o modelo Hash voltado à plataforma HPC Shelf, componentes possuem espécies e interesses bem definidos, em especial o paralelismo;
- *abstração*: o software é construído em alto nível, através de um modelo de conexão bem estabelecido, permitindo que o desenvolvedor junte componentes e construa novas aplicações;
- *plataformas*: o componente pode representar vários interesses, o que inclui a plataforma virtual onde o software é executado. Essa característica é inerente à HPC Shelf;
- *Graph*: separação clara da finalidade deste componente voltado para aplicações com interesse em grafos. GRAPH suporta propriedades matemáticas da teoria dos grafos, podendo ainda ser estendido ou possuir novas versões concretas que atendam mais requisitos além dos que atualmente foram testados. Por exemplo, versões concretas para desempenho em processamento, uso de memória, ou inclusão de *out-of-core*. Na HPC Shelf, essas alternativas são adaptáveis à plataforma de execução, por exemplo usando a nova versão *out-of-core* em plataformas com limitada memória;
- *Gusty*: define três métodos usados no processamento de grafos, dos quais `unroll` destaca-se por permitir operações assíncronas, onde dados podem ser recebidos, pré-processados e imediatamente liberados.

6 AVALIAÇÃO EXPERIMENTAL

Este capítulo tem o propósito de apresentar uma avaliação de desempenho do arcabouço de componentes Gust, comparando-o com outros dois sistemas alternativos voltados para o processamento de grafos grandes. Nessas comparações, usam-se técnicas de medição e aplicam-se conceitos estatísticos a fim de oferecer garantias quanto a confiabilidade dos resultados. Além disso, com base nos resultados da análise quantitativa, são comparadas, qualitativamente, algumas características de programação comuns entre as alternativas.

Este capítulo está dividido em quatro seções. Na Seção 6.1, são apresentados os sistemas que serão comparados ao Gust, as cargas utilizadas no experimento, quais são as medidas a serem observadas nos sistemas e como serão aplicados os recursos estatísticos a fim de garantir maior objetividade nas conclusões. Na Seção 6.2, detalham-se os recursos utilizados para realização de experimentos, definindo as características e propriedades principais da plataforma experimental. Na Seção 6.3.1, comparam-se algumas formas de manipulação de dados características de cada sistema, que são importantes para embasar algumas conclusões. Na Seção 6.3.2, apresentam-se e discutem-se os dados obtidos a partir dos experimentos.

6.1 Metodologia

Os sistemas GraphX e Giraph, representantes do estado da arte em processamento de grafos, são aqueles que serão usados para avaliação do Gust.

O GraphX caracteriza-se por ser centrado em bloco ou grafo, implementando modelos como Pregel e GAS/PowerGraph. Sua escolha se deve ao fato de ser parte do Spark, uma evolução de plataformas tradicionais MapReduce, tais como Hadoop, oferecendo alto nível de abstração através de uma linguagem funcional, bem como permitindo interatividade com o usuário. Por outro lado, o Giraph foi escolhido por ser um sistema Apache centrado em vértices, implementando o modelo Pregel e utilizando a plataforma Hadoop como base de processamento MapReduce. O Giraph é referência para outros arcabouços de processamento e projetos de pesquisadores (HAN; DAUDJEE, 2015; TIAN *et al.*, 2013), tendo sido aplicados a bases de dados bastante significativos de grandes empresas, tais como Facebook (CHING *et al.*, 2015). Portanto, tais atributos sugerem que ambos os sistemas são eficientes e usados com sucesso em ambientes reais.

As cargas utilizadas são selecionadas com base no *tamanho*, na *relevância* e na

origem dos dados. Tamanho, porque devem ser compatíveis com o ambiente de execução, contemplando carga baixa, média e alta, com o propósito de atingir os limites do hardware disponível. Relevância, porque devem ter sido previamente usadas para propósitos de avaliação de desempenho nos sistemas escolhidos, conforme relatado na literatura. Origem, porque devem ter origem a partir de fontes pertinentes, notadamente disponibilizadas por laboratórios de pesquisas que lidam com processamento de grafos em larga escala. Por esses motivos, foram escolhidas as cargas *amazon0302*, *skitter*, *pokec* e *LiveJournal*, disponibilizadas pela universidade *Stanford* em *SNAP Datasets* (LESKOVEC; KREVL, 2014). Dentre essas, *LiveJournal* é a maior carga, sendo especialmente de interesse deste trabalho por ser adotada em outros trabalhos (GONZALEZ *et al.*, 2012; XIN *et al.*, 2013; ARIDHI *et al.*, 2016; CHING *et al.*, 2015) que avaliaram sistemas de processamento de grafos, incluindo GraphX e Giraph. A Tabela 13 descreve detalhes das cargas, bem como suas abreviações que serão utilizadas ao longo deste capítulo.

Cargas	**Vértices**	**Arestas**	**Triângulos**	**Diâmetro**
amazon0302 (AMA)	262.111	1.234.877	717.719	32
skitter (SKI)	1.696.415	11.095.298	28.769.868	25
pokec (POK)	1.632.803	30.622.564	32.557.458	11
LiveJournal (LIV)	4.847.571	68.993.773	285.730.264	16

Tabela 13 – Propriedades das cargas

Os sistemas de computação paralela usados são vistos como *benchmarks*. De forma semelhante à estratégia de escolha das cargas, devem ter sido usados previamente para o propósito de avaliação de *frameworks* de processamento de grafos grandes. Devem ainda oferecer uma amostra de diferentes padrões computacionais, contemplando baixa, média e alta taxa de computação/comunicação. Por esses motivos, foram utilizados os estudos de caso descritos na Seção 5.4, frequentemente adotados para fins de avaliação: ranqueamento de páginas (*PageRank*), enumeração de triângulos e menor caminho (*SSSP*). Além disso, na maioria dos *frameworks* são oferecidos como amostra representativa de funcionalidades, possuindo implementações claras e objetivas no Giraph e GraphX, com resultados que permitem comparações em desempenho e corretude.

A Tabela 14 apresenta as características das implementações dos algoritmos realizadas neste trabalho. Buscou-se utilizar os mesmos tipos de dados das implementações que rodam no Giraph e GraphX, especificamente valor de vértice e aresta, bem como identificadores

de vértices. Além disso, com relação à configuração dos *benchmarks*, o *PageRank* é o único parametrizável, de modo que definimos seu número de iterações para 30.

Benchmarks	Número de Iterações	Comunicação
TRIANGLECOUNT	Exatamente três iterações	média
SSSP	Número dinâmico de iterações	baixa
PAGERANK	Número de iterações atribuído por variável	alta

Tabela 14 – Características dos Benchmarks

Para aferir e comparar resultados, utiliza-se medição e métodos estatísticos, visando quantificar o desempenho de cada sistema e compará-los objetivamente. As medidas usadas são do tipo "quanto menor, melhor", sendo elas o tempo de processamento gasto para processar determinada tarefa e o consumo de memória durante o processamento. Para comparar tais aspectos, busca-se uma medição justa tanto quanto possível. Por exemplo, Giraph e GraphX são tolerantes a falhas, mas nativamente não suportam múltiplos *clusters*. Por sua vez, Gust não é tolerante a falhas e é voltado para HPC Shelf, a qual é capaz de explorar o desempenho de múltiplas plataforma virtuais (*clusters*). Convergindo para um ambiente comum, descartam-se eventuais medições com recuperação de falhas por parte dos *frameworks* medidos, bem como os experimentos são realizados em apenas um *cluster*, o que não resulta em perda de generalidade para os objetivos desse estudo experimental, visto que o paralelismo inter-cluster é de grossa granularidade e não envolve comunicação entre os agentes paralelos executados em plataformas virtuais diferentes. Para evitar impacto com *out-of-core*, as aplicações executadas sobre GraphX utilizam a opção `cache` ou `persist`, com nível `MEMORY_ONLY`, que visam minimizar envio de dados ao disco. De forma semelhante, os arquivos de formato XML usados no Giraph são configurados para disponibilizar o máximo possível de memória, permitindo que o sistema conserve todo o grafo na memória. Por outro lado, para execução em um único *cluster*, isolamos os códigos Gust de forma monolítica, uma vez que isso não causa impacto significativo para melhor ou pior desempenho (CARVALHO JUNIOR; REZENDE, 2013). Com isso, busca-se equiparar a comunicação e as computações que devem ser realizadas pelos três sistemas.

6.1.1 Fatores Explorados e Métodos Estatísticos

O processo de avaliação de desempenho é realizado pela análise de um conjunto de quatro fatores, identificados como *P*, *C*, *S* e *B*, referidos adiante como fatores *PCSB*. O fator

P consiste em um valor numérico inteiro correspondente ao número de unidades distribuídas disponíveis para processamento, sendo associado com os seguintes níveis: 1, 2, 4, 6, 8, 10, 20 e 30. O fator C consiste nas cargas de trabalho a serem processadas, cujos níveis encontram-se definidos na Tabela 13: AMA, SKI, POK, LIV. O fator S consiste nos sistemas que serão comparados, sendo os níveis Gust, Gust-Metis, GraphX e Giraph respectivamente abreviados da seguinte forma, respectivamente: GUS, MET, GRA e GIR. Gust-Metis representa o próprio Gust, porém com balanceamento de carga realizado pela ferramenta Metis (KARYPIS; KUMAR, 1995). O fator B consiste nos tipos de *benchmarks* usados, os quais implementam os algoritmos de ranqueamento de páginas, enumeração de triângulos e caminho mínimo, respectivamente representados pelos níveis PR, TRI, $SSSP$.

Configuração	Total
$P[20,30] \times C[LIV] \times S[GUS,GRA] \times B[PR,TRI,SSSP]$	12
$P[6,8,10] \times C[AMA,SKI,POK,LIV] \times S[GUS,MET,GRA,GIR] \times B[PR,TRI,SSSP]$	144
$P[4] \times C[AMA,SKI,POK,LIV] \times S[GUS,MET,GRA] \times B[PR,TRI,SSSP]$	36
$P[4] \times C[AMA,SKI,POK] \times S[GIR] \times B[PR,TRI,SSSP]$	9
$P[4] \times C[LIV] \times S[GIR] \times B[TRI,SSSP]$	2
$P[2] \times C[AMA,SKI,POK,LIV] \times S[GUS,MET] \times B[PR,TRI,SSSP]$	24
$P[1] \times C[AMA,SKI,POK,LIV] \times S[GUS] \times B[PR,TRI,SSSP]$	12
Total	239

Tabela 15 – Configuração de experimentos

No processo experimental, definem-se os fatores e seus conjuntos de níveis com expressões do tipo $X[x_1,...,x_n]$, onde X é um dos fatores $PCSB$, e $x_1,...,x_n$ o conjunto de níveis utilizado por X. A Tabela 15 apresenta todas as configurações de experimentos a serem realizados. Por exemplo, a expressão $P[6,8,10] \times C[AMA,SKI,POK,LIV] \times S[GUS,MET,GRA,GIR] \times B[PR,TRI,SSSP]$ refere-se a um total de 144 experimentos, combinando o número de níveis. A remoção de níveis significa que o nível que deveria ser avaliado não conseguiu ser executado com a devida configuração. Por exemplo, Giraph não conseguiu executar em 4 unidades distribuídas com a carga LIV no PR, em decorrência de falta de memória. Isso implica na remoção de GIR para PR, pois atingiu seu limite para o número de unidades distribuídas/carga. De fato, o critério de eliminação consiste em o *framework* não conseguir executar seu trabalho em pelo menos 50% das tentativas, ou seja, quando o sistema começa falhar repetidas vezes torna-se inviável coletar suas medidas. Além disso, realizam-se testes com 20 e 30 unidades de processamento apenas com o Gust e o GraphX, uma vez que Giraph excedeu o limite de memória com processadores com múltiplos núcleos, usados a partir do nível 10. Dessa forma, 10, 20 e 30 consistem respectivamente na alocação de 1, 2 e 3 processos por processo trabalhador.

O ponto de corte é em 30, uma vez que, acima disso, GraphX também excedeu o limite de memória. Aos experimentos que se aplicam apenas ao Gust, como é o caso de $P[1]/P[2]$, cada experimento é realizado através da repetição de 10 execuções, uma vez que o desvio padrão é mínimo e as medidas servem apenas como parâmetros do próprio Gust. Para as demais experiências, conforme sugerido na literatura, cada experimento é realizado através da repetição de 31 execuções, gerando 31 medições de tempo. Isso caracteriza uma variável aleatória cuja média define a tendência central do tempo de execução.

Optamos pela eliminação de valores discrepantes (*outliers*), permitindo que os dados amostrais sejam aproximados a uma distribuição normal. Um *outlier* consiste em um valor com tempo de resposta baixo ou elevado demais em comparação às respostas de outras medições. Suas ocorrências são provavelmente originárias de variações inesperadas em um ou mais nós do *cluster* durante uma determinada execução. Para as características destes experimentos, devem ser eliminados *outliers* relativos a valores elevados. Porém, todo tempo medido só é confirmado após validação dos dados de saída da execução, feita por soma de verificação (*checksum*), bem como análise manual em busca de eventuais falhas. Com essa confirmação, inserem-se dados em uma planilha eletrônica, para que ao término dos experimentos sejam identificados e eliminados os *outliers*. A taxa de remoção foi abaixo dos 20%.

Além dos fatores diretamente relacionados com métodos estatísticos, inclui-se também uma seção para discussão sobre as características de programação oferecidas pelos sistemas, resultado da adaptabilidade, modularidade e abstração nesses sistemas. Em seguida, destacam-se os métodos estatísticos aplicados aos dados medidos.

6.1.1.1 Métodos

Fatores *PCSB* e seus níveis são organizados para realização de avaliação de desempenho, que consiste em cálculo de *intervalos de confiança* (*IC*) a 95%, casos de testes com *Test-t*, tabelas de experimentos fatoriais (JAIN, 1991) e gráficos. Além disso, define-se o *IC* também para a sobrecarga do uso do Gust frente aos outros sistemas. Para isso, na comparação entre dois sistemas, um *IC* de sobrecarga (em valor percentual) é calculado da seguinte forma: dados os intervalos $[g_0, g_1]$ para Gust e $[x_0, x_1]$ para algum *framework GRA* ou *GIR*, os valores mínimo e máximo da sobrecarga de uma execução Gust são dados respectivamente por $(g_0 - x_1)/\bar{x}$ e $(g_1 - x_0)/\bar{x}$, tal que $\bar{x} = (x_0 + x_1)/2$. Sobrecargas negativas indicam melhor desempenho na execução Gust.

Fatores	Níveis/efeitos	Definições
P	$[6,8,10]/P_k$	Número de unidades de processamento
C	LIV	Carga LiveJournal
S	$[MET, GUS, GRA, GIR]/S_j$	Sistemas comparáveis
B	$[PR, SSSP, TRI]/B_i$	Benchmarks

$\overline{y}_{ijk}$ = média de um conjunto de medições y_{ijkn}, onde n é o índice da medição válida

μ = média geral

$\overline{\mu}_{i..}$ = média do i

$\overline{\mu}_{.j.}$ = média do j

$\overline{\mu}_{..k}$ = média do k

$S_j = \overline{\mu}_{.j.} - \mu$, é o efeito do fator S no nível j

$B_i = \overline{\mu}_{i..} - \mu$, é o efeito do fator B no nível i

$P_k = \overline{\mu}_{..k} - \mu$, é o efeito do fator P no nível k

Por exemplo, P_8 é a média das linhas cinza - μ

	S	P	B			média	efeito
			PR	SSSP	TRI		
LIV	MET	6	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$	$\overline{\mu}_{.j.}$	S_j
		8	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$		
		10	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$		
	GUS	6	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$	$\overline{\mu}_{.j.}$	S_j
		8	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$		
		10	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$		
	GRA	6	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$	$\overline{\mu}_{.j.}$	S_j
		8	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$		
		10	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$		
	GIR	6	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$	$\overline{\mu}_{.j.}$	S_j
		8	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$		
		10	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$	$\overline{y}_{ijk}$		
média			$\overline{\mu}_{i..}$	$\overline{\mu}_{i..}$	$\overline{\mu}_{i..}$	μ	
efeito			B_i	B_i	B_i		

P	média	efeito
6	$\overline{\mu}_{..k}$	P_k
8	$\overline{\mu}_{..k}$	P_k
10	$\overline{\mu}_{..k}$	P_k

Tabela 16 – Tabela de experimentos fatoriais com a carga fixa LIV e processadores limitados a 6, 8, 10

Tabelas de experimentos fatoriais são destinadas a calcular efeitos de nível de fator, especificamente P_k, S_j e B_i, conforme descrito nos exemplos da Tabela 16. Um efeito apenas diz o quanto a média de execuções do nível está afastada da média geral, de modo que a soma dos efeitos é zero. Quando a média geral é subtraída da média do nível, efeitos com melhores resultados ficam abaixo de zero, enquanto os piores ficam acima. Portanto, zero ou próximo de zero representa igualdade. Neste trabalho, as tabelas de experimentos fatoriais serão apresentadas com $P[6,8,10]$ unidades de processamento, quando todos os sistemas estão presentes e conseguem executar a carga LIV, bem como quando a distribuição dos dados tende a atingir todo o *cluster*. Através disso, busca-se verificar numericamente quais são os níveis com desempenho próximo ou divergente.

6.2 Ambiente Experimental

Os experimentos são realizados em um ambiente de nuvem computacional, com infraestrutura localizada no Departamento de Computação da Universidade Federal do Ceará. Os recursos são 10 máquinas virtuais (*VM*) controladas pelo software OpenStack (JACKSON, 2012), onde cada *VM* possui *16GB* de *RAM*, *8 vCPU*, *20GB* de armazenamento. O conjunto de *VMs* organiza uma plataforma virtual ou *cluster*, conectado por memória distribuída. As configurações das *VMs* representam um ambiente limitado de recursos, mas compatível com as cargas de trabalho escolhidas, pois há o objetivo de observar a capacidade de cada sistema em lidar com pouco recurso. De fato, as cargas também são variadas em tamanho, o que proporciona experimentos com baixo, médio ou alto índice de utilização de processador, memória e rede. Cada máquina virtual recebe o código fonte e a compilação do Giraph, Spark e Hadoop. Para compilação, as versões estáveis que estiveram disponíveis no início da montagem do ambiente de execução e da coleta de dados para este trabalho são: Spark 2.0.2, Giraph 1.1.0, Hadoop 2.7.2. Para execução das versões Gust (*GUS* e *MET*), usa-se o Mono 4.2.2 (Xamarin, 2004) e MPI.NET 1.0, os quais suportam C#. Finalmente, o sistema operacional utilizado é o Ubuntu 16.04.

6.3 Análise dos Sistemas

Nesta seção, são apresentados os resultados e discussões a respeito da avaliação de desempenho cuja metodologia foi explicada nas seções anteriores. Antes disso, discutimos na próxima seção detalhes sobre as interfaces de manipulação de dados dos sistemas, relevantes posteriormente para embasar discussões.

6.3.1 Manipulação de dados em Gust, GraphX e Giraph

Os principais artefatos de software que lidam diretamente com as abstrações de vértices e arestas nos sistemas Gust, GraphX e Giraph são, respectivamente, o componente GRAPH, a classe GraphX.`Graph` e a classe Giraph.`Vertex`.

No Giraph, onde o programador lida com um modelo de programação centrado nos vértices ("*vertex-centric*"), o desenvolvedor possui uma referência a objetos do tipo `Vertex<`*I*`,`*V*`,`*E*`>`, onde: *I* é o tipo do identificador de vértice; *V* é o tipo do vértice; e *E* é o tipo da aresta. `Vertex` possui os métodos descritos na Tabela 17, onde os métodos correspondentes do componente GRAPH, do Gust, são apresentados (coluna à direita). É importante destacar que GRAPH é

usado pelo programador dentro de GUSTYFUNCTION, deixando todos os vértices acessíveis pelo método `vertexSet`. De fato, Gust implementa o modelo Pregel sob uma visão centrada em grafo, enquanto Giraph adota o modelo clássico centrado em vértice. Entretanto, como estamos lidando com componentes e um contexto *chave/valor* MapReduce, cada chave `TKey` pode representar um vértice, sugerindo ao desenvolvedor a possibilidade de trabalhar com vértices individuais do grafo, ou seja, a forma clássica centrada em vértices suportada por Giraph.

Giraph.Vertex<I,V,E>:	Gust.IGraphInstance<V, E, TV, TE>:
I=tipo de id do vértice V=tipo de dado do vértice E=tipo de dado de aresta	TV=tipo id do vértice V=componente Vertex para dados E=componente Edge para dados TE=instâncias de arestas geradas pelo padrão *Factory*
I getId();	foreach (TV v in vertexSet ())
V getValue();	getValue(v)
void setValue(V value);	setValue(value)
int getNumEdges();	degreeOf(TV vertex), inDegreeOf(TV vertex), outDegreeOf(TV vertex)
Iterable<Edge<I, E>> getEdges();	ICollection<TE> edgesOf(TV vertex)
void setEdges(Iterable<Edge<I, E>> edges);	alterações individuais em arestas com setEdgeWeight(TE e)
E getEdgeValue(I targetVertexId);	getEdge(TV source, TV target).Weight
void setEdgeValue(I targetVertexId, E edgeValue);	setEdgeWeight(TE edge)
Iterable<E> getAllEdgeValues(final I targetVertexId);	foreach(TE e in edgesOf(TV vertex)) e.Weight
void addEdge(Edge<I, E> edge);	addEdge(TE e) e addEdge(TV source, TV target)
void removeEdges(I targetVertexId);	removeAllEdges(TV source, TV target)

Tabela 17 – Tabela comparativa de funcionalidades dos artefatos que manipulam grafos nos sistemas Gust e Giraph

var g = GraphX.Graph[V, E]	var g = Gust.IGraphInstance<V, E, TV, TE>
g.inDegrees.filter(v => v._1 == 10L).first	g.inDegreeOf(10)
g.inDegrees.filter(v => v._2 < 2).foreach(println)	foreach(int v in g.vertexSet().Where(v=>g.inDegreeOf(v)<2)) Console.WriteLine(v+","+g.inDegreeOf(v));
g.outDegrees.filter(v => v._1 == 10L).first	g.outDegreeOf(10)
g.outDegrees.filter(v => v._2 > 20).foreach(println)	foreach(int v in g.vertexSet().Where(v=>g.outDegreeOf(v)>20)) Console.WriteLine(v+","+g.outDegreeOf(v));
g.edges.filter(e => e.srcId > e.dstId).count	g.edgeSet().Where (e => e.Source > e.Target).Count ();
g.vertices.filter(v => v._1 > 262000L).count	g.vertexSet().Where (v => v > 262000).Count ();

Tabela 18 – Tabela comparativa de algumas operações com artefatos que manipulam vértices e arestas nos sistemas Gust e GraphX

Em uma estratégia usando expressões de consultas, a Tabela 18 apresenta algumas pesquisas realizadas com a classe `Graph` do GraphX, bem como operações similares realizadas com o componente GRAPH do Gust, na segunda coluna. Nota-se a facilidade em realizar filtragens em coleções de dados com o GraphX. Contudo, a linguagem C#, usada no Gust, também contempla operações similares, o que é possível através do componente .NET chamado System.Linq, que suporta a linguagem de consultas LINQ (*Language Integrated Query*), inspirada em SQL (*Structured Query Language*). Todavia, algumas diferenças estão na forma de lidar com a coleção ou tipo primitivo, como visto em seguida:

- no GraphX, as propriedades destacadas são: `inDegrees`, coleção de vértices e seus graus de entrada; `outDegrees`, coleção de vértices e seus graus de saída; `vertices`, coleções de vértices e seus valores; `edges`, coleções de arestas e seus valores;
- no Gust, os métodos `inDegreeOf` e `outDegreeOf` simplesmente retornam inteiros, enquanto `vertexSet` e `edgeSet` respectivamente retornam as coleções de vértices e arestas da partição;
- Na primeira operação de GraphX, a coleção `g.inDegrees.filter(v => v._1 == 10L).first`, destaca um operador *v* com os atributos *v.*_1 e *v.*_2, os quais representam respectivamente o identificador do vértice iterado na coleção e o seu grau de entrada. No exemplo, a busca é feita pelo vértice 10. No Gust, simplesmente é usado `g.inDegreeOf(10);`
- Na segunda operação, GraphX reduz bastante o código em relação ao Gust, justamente pelo acesso direto ao iterador `foreach` e porque o grau de entrada em GraphX é acessado por *v.*_2, enquanto em Gust deve-se usar `g.inDegreeOf(v)` e um `Console.WriteLine` dentro do `foreach`. A operação imprime todos os vértices com grau de entrada menor que 2;
- Nas demais operações que não usam o iterador `foreach(`*count*`)`, as formas de acesso são semelhantes, variando apenas a nomenclatura.

O estilo funcional, herdado da linguagem Scala, torna o GraphX, bem como o próprio Spark, uma ferramenta relativamente simples de se usar, especialmente para consultas de dados em um ambiente que pode ou não ser interativo com o usuário. No caso de um ambiente interativo, isso é feito através de um terminal, onde o operador insere códigos e imediatamente a resposta da execução ou consulta é obtida. No caso de um ambiente não interativo, o operador pode compilar o código e lançar um trabalho (*job*) que será paralelizado em um *cluster*. Em ambos os casos, a exigência de memória é crítica para cargas grandes. Neste aspecto, obtivemos resultados competitivos usando Gust, como serão vistos nas tabelas e gráficos de desempenho apresentados nas próximas seções deste capítulo. Avaliamos que essa vantagem está no uso de coleções otimizadas internamente aos componentes GRAPH e DATACONTAINER, que armazenam apenas dados essenciais, que podem ser tipos primitivos. Além disso, para redução do consumo de memória, tem-se considerado vantajoso também o uso do próprio C#, que aceita o uso de tipos primitivos em variáveis de tipo na definição dos tipos genéricos de coleções, uma vez que preliminarmente comparamos implementações de GRAPH análogas em Java, manipulando grandes grafos. Por outro lado, a parte interativa com Gust é vista como um trabalho futuro.

Além disso, as consultas através da sintaxe LINQ podem ser melhor exploradas, com um maior número de coleções retornadas pelo componente Graph, através de novos métodos, também em trabalhos futuros. Na próxima seção, discutem-se os resultados experimentais de desempenho.

C	S	P	B			média	efeito Sj
			PR	SSSP	TRI		
LIV	MET	6	243.192	84.907	120.539	143.900	-125.851
		8	229.253	83.798	121.978		
		10	208.436	83.136	119.862		
	GUS	6	278.530	119.926	237.401	183.482	-86.269
		8	251.870	110.456	181.465		
		10	213.270	108.098	150.322		
	GRA	6	428.173	197.057	424.319	319.871	50.121
		8	380.927	209.790	360.767		
		10	353.116	196.852	327.843		
	GIR	6	438.393	65.188	1012.727	431.749	161.998
		8	373.029	69.543	797.744		
		10	330.605	63.822	734.691		
		média	310.733	116.048	382.471	269.751	
		efeito Bi	40.982	-153.703	112.721		

P	média	efeito P_k
6	304.196	34.445
8	264.218	-5.532
10	240.838	-28.913

Tabela 19 – Experimento fatorial LIV

6.3.2 Experimentos e Discussões

Na comparação entre dois sistemas, tais como $GUS \times GRA$ (Gust compardo com GraphX) ou $GUS \times GIR$ (Gust comparado com Giraph), os resultados podem apontar igualdade ou superioridade de um sobre o outro. Por esse motivo, o analista pode realizar um esboço relacional dos principais dados medidos, de forma preliminar, para avaliar quais opções tendem a ser diferentes ou iguais. Tabelas de experimentos fatoriais são ferramentas que auxiliam na obtenção dessas informações, pois apontam os fatores e a distância entre níveis, considerados efeitos. Dessa forma, observaram-se três fatores e efeitos principais: *benchmark* B_i, sistema S_j e processadores P_k. Em seguida, discutem-se esses resultados.

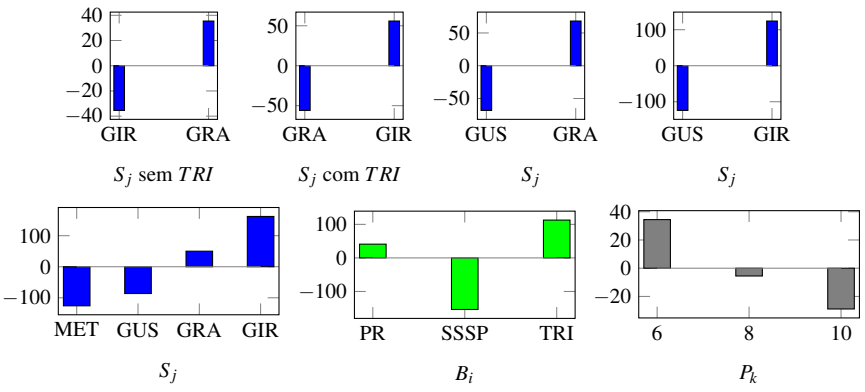


Figura 39 – Comparação de Efeitos Principais S_j, B_i, P_k, para LIV e $P_{k>4}$

6.3.2.1 *Experimentos Fatoriais*

Para os *processadores*, destacam-se experimentos fatoriais com $P[6,8,10] \times C[LIV]$, pois é onde inicia-se a convergência para o uso máximo dos sistemas e do *cluster*. Como resultado, os efeitos P_k não divergiram do esperado para sistemas distribuídos escaláveis, uma vez que, aumentando o número de processadores, o desempenho melhorou. Isso pode ser visto na Tabela 19, ou graficamente na Figura 39, onde valores abaixo de zero são melhores. Ao observar esses efeitos, nota-se que a distância entre 6 e 8 VMs é maior que a distância entre 8 e 10 VMs, apontando que o *cluster* começa a estabilização no tempo de execução, que será percebida com o uso de 20 e 30 processadores (Figura 43). Nas cargas menores, esse quadro de estabilização é visto já no início do incremento *P*, como apontam os gráficos de escalabilidade apresentados nas figuras 45 e 44, onde há memória disponível mas o custo de comunicação e particionamento prejudica o desempenho em *GRA* e *GIR*. Por outro lado, em um hipotético aumento de carga, o incremento de máquinas virtuais ou memória deve ser considerado, especialmente porque os recursos foram delimitados. Contudo, para Gust, sua estrutura de armazenamento de vértices e arestas é compacta, o que permitiu executar a carga *LiveJournal* em todos os *benchmarks*, até mesmo em 1 processador, enquanto que GraphX e Giraph receberam ponto inferior de corte em 4. Além disso, a curva de escalabilidade para Gust segue em queda ou em reta nas menores cargas.

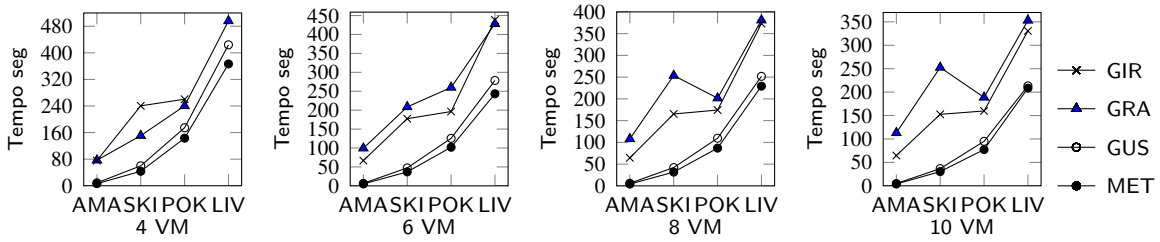


Figura 40 – Perfil de execução do Page Rank - 4 a 10 VM

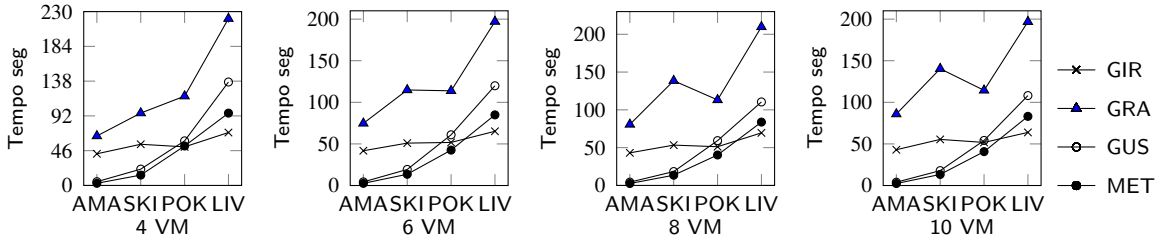


Figura 41 – Perfil de execução do SSSP - 4 a 10 VM

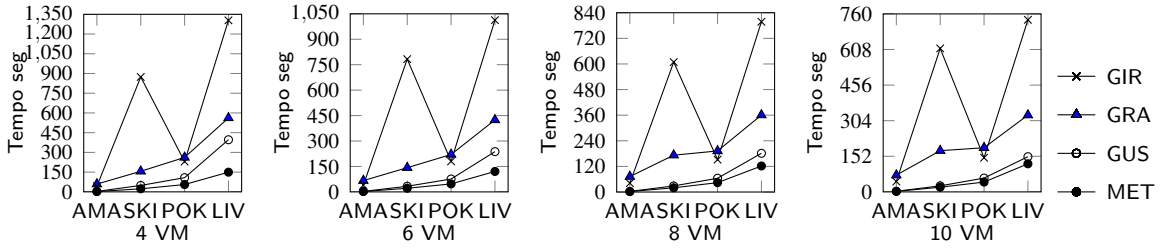


Figura 42 – Perfil de execução do Triangle Count - 4 a 10 VM

Para os *benchmarks*, a enumeração de triângulos (*TRI*) é a que demandou mais tempo, quando envolvido com todas as medições experimentais. Porém, esse resultado não significa que *TRI* requer processamento e comunicação mais intensa em relação aos demais, pois alguns dos sistemas podem apresentar desempenho insuficiente para *TRI*, de forma isolada. Na prática, isso é visto na Figura 42, para as cargas *SKI* e *LIV*, no Giraph. Tem-se considerado que isso ocorre devido à estratégia de particionamento do Giraph, que realiza corte de arestas, uma vez que Spark e Gust fazem corte de vértices. De fato, enquanto Giraph reporta *TRI* como ponto crítico, para Spark e Gust o *benchmark* com maior custo é *PR*, seguido por *TRI*, como apontam os gráficos nas figuras 42 e 40. Em comum, *SSSP* é considerado o mais leve.

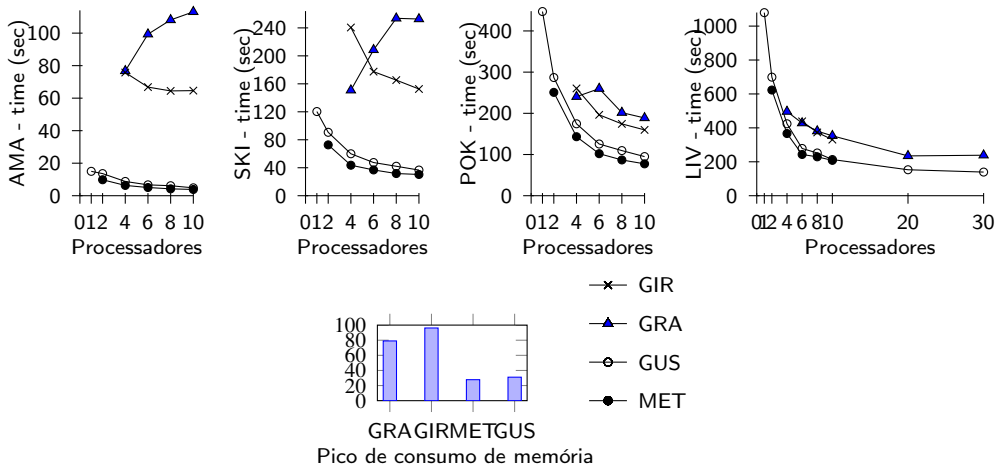


Figura 43 – PageRank - Escalabilidade Horizontal

Para os efeitos S_k, de forma ordenada, os sistemas com maior desempenho foram, nessa ordem, Gusti-Metis, Gust, GraphX e Giraph. Para o primeiro, por representar o Gust com melhor balanceamento de carga proporcionado pela ferramenta Metis, era esperado melhor desempenho em relação ao Gust tradicional, com particionamento padrão *hash*. Entre GraphX e Giraph, considerando todos os *benchmarks*, o primeiro obteve melhor desempenho. Porém,

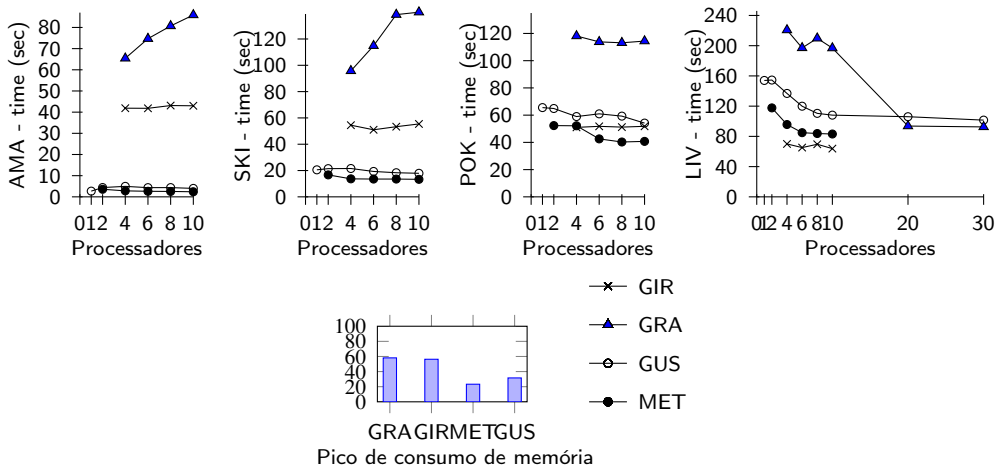


Figura 44 – SSSP - Escalabilidade Horizontal

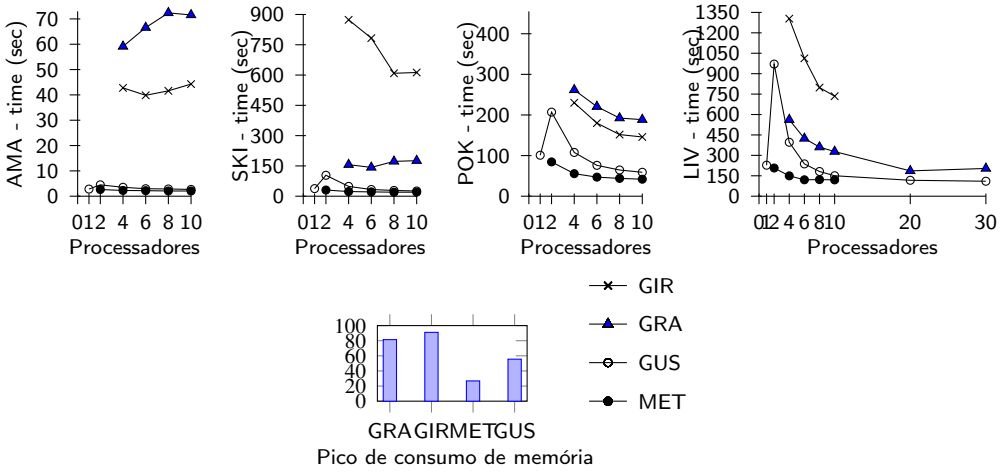


Figura 45 – Triangle Count - Escalabilidade Horizontal

ao remover o caso crítico do Giraph (enumeração de triângulos), Giraph obteve melhor desempenho que GraphX, como destacado na Figura 39 (S_j sem TRI). Por outro lado, para o Gust, todas as comparações S_j apontam melhor desempenho deste sistema frente aos demais. Além disso, conforme as medições de pico de memória, destacadas nas figuras 43, 45 e 44, Gust foi significativamente mais econômico.

Portanto, os efeitos S_j apresentaram resultados competitivos para os sistemas, onde MET e GUS superaram GRA e GIR em desempenho com tempo de resposta e memória, para todas as cargas e *benchmarks*. Esta é uma observação preliminar com experimentos fatoriais, de modo que o próximo passo consiste na observação estatística de intervalos de confiança e *Test-t*, bem como os gráficos de escalabilidade.

6.3.2.2 *Resultados para Intervalo de Confiança (IC) e Teste-t*

Um intervalo de confiança (*IC*) permite definir os limites para uma média amostral com algum grau de confiança, tipicamente 90% ou 95%. Nesta seção, utiliza-se 95%, para comparar dois sistemas. A técnica *IC* usada nos experimentos consiste em subtrair medições pareadas de dois sistemas, para gerar uma amostra da diferença, também vista como *overhead*. Por exemplo, sejam x_i e y_i as medições i coletadas para os sistemas X e Y, a subtração $x_i - y_i$ produz uma amostra da diferença entre os sistemas X e Y. Quando é calculado o *IC* com os limites $[l_0, l_1]$, há três possibilidades: l_0 e l_1 menor que zero, o sistema X possui melhor desempenho que Y; $l_0 > 0$ e $l_1 > 0$, Y supera X; $l_0 =< 0$ e $l_1 >= 0$, os sistemas são iguais estatisticamente.

Nas tabelas 20, 21, 22 e 23, inserem-se os dados comparativos entre dois sistemas pareados: $GUS \times GRA$ e $GUS \times GIR$. As linhas são respectivamente referentes à média amostral, mínimo e máximo valor medido, desvio padrão, intervalo de confiança da amostra, diferença das medições $(x_i - y_i)$ juntamente com seu *IC*, *Teste-t* para casos de teste.

Pelas regras de intervalos de confiança, *GUS* obteve desempenho superior aos demais sistemas em quase todos os casos, com exceção do experimento com *benchmark SSSP* processado pelo Giraph nas cargas *LIV* e *POK*. Os resultados são compatíveis com os percentuais de *overhead* definidos pelas linhas *min-overhead %* e *max-overhead %*, que estabelecem o intervalo de confiança de *overhead* do Gust em relação aos sistemas *GRA* e *GIR*, o que tem gerado limites negativos que favorecem *GUS*. Destaca-se que o desempenho Gust para *SSSP* também foi inferior ao GraphX com 20 e 30 unidades de processamento. Porém, o *overhead* positivo em favor do GraphX é relativamente menor que o *overhead* negativo em favor do Gust com *PR* e *TRI*. Além disso, GraphX inverte sua curva de desempenho para *PR* e *TRI*, nos casos de 20 para 30 processadores, indicando que o aumento de unidades de processamento é desnecessário para o tamanho da carga. Numericamente, isso é melhor visto na Tabela 20, no campo média *GRA*, para 20 e 30 processadores. Ou seja, percebe-se que não há ganho de desempenho com o incremento de 20 para 30 processadores no GraphX, especificamente com os *benchmarks PR* e *TRI*.

Para confirmar os resultados dos intervalos de confiança, os casos de testes (*Test-t*) são empregados e disponibilizados nas tabelas. Essa técnica diz se um sistema é diferente do outro através de rejeição ou aceitação, com $(1-\alpha) \times 100\%$ de confiança. No caso de aceitação, deve-se olhar para a subtração $x_i - y_i$, ou para *min-overhead %* e *max-overhead %*, onde valores

negativos definem o desempenho favorável ao Gust. Nas tabelas de *Teste-t* (20, 21, 22 e 23), os destaques em negrito referem-se aos casos de teste onde melhor desempenho foi obtido com *GUS*.

6.4 Considerações

Neste capítulo, foram apresentados os experimentos comparativos entre Gust e os *frameworks* estado da arte GraphX e Giraph. Os resultados demonstraram aspectos promissores em relação ao Gust, especialmente nas medidas de desempenho e economia no uso de memória. Para isso, a metodologia empregada abordou a representação de vários cenários, explorando baixa, média ou alta taxa de utilização de recursos. Por sua vez, os recursos foram compatíveis com as cargas e o paralelismo, que foi limitado a 30 unidades de processamento, quando o *framework* GraphX tem desempenho estabilizado (sem melhoramento no tempo de execução) para a carga *LiveJournal*, bem como exige mais memória da máquina virtual multi-núcleo, condicionando execuções no limite superior de 30 processadores.

Com esses resultados, abre-se a possibilidade de inserção de novas funcionalidades ao Gust, uma vez que os *benchmarks* estiveram livres para serem executados em todos os níveis fatoriais, não atingindo os limites de recursos do *cluster*. Por exemplo, inserção de novas formas de consultas nas coleções de dados, o que já é feito com sucesso no GraphX. Ou ainda, adição de instruções para *out-of-core* na interface Graph, visando tolerância a falhas. Ainda, ambiente de interação com o usuário, também já feito com sucesso pelo GraphX. Em contribuição com esses sistemas, Gust apresenta uma moderna engenharia de software paralelo, onde MapReduce, estendido por Gust, oferece um conjunto de *blocos de construção*, possibilitando adaptabilidade e modularidade para fluxos computacionais. Adicionalmente, sendo orientado a componentes e interesses de software, Gust consiste em uma aplicação HPC Shelf, voltada a paralelismo de larga escala, que suporta múltiplos *clusters*.

No modelo de programação, todos os sistemas ofereceram interfaces de alto nível para os desenvolvedores, encapsulando o paralelismo necessário ao processamento de grandes grafos. A interface `Graph` do GraphX é rica em recursos voltados ao controle de vértices e arestas, sendo "graph-centric", enquanto a interface `Vertex` em Giraph é relativamente restritiva, mas oferece operações fundamentais voltadas a vértices em grafos direcionados. No caso do Gust, a interface GRAPH oferece opções genéricas, pois seu uso deve ser compatível com o contexto *chave/valor* do MapReduce, onde as *chaves* podem representar subgrafos ou vértices.

Com nomenclaturas sugestivas, os métodos dessa interface procuram atender aos requisitos da teoria dos grafos, de modo que o programador possa entender os propósitos de cada método. Além disso, tem-se uma separação entre operações com grafos direcionados e não direcionados, sendo possível também determinar o uso de grafos simples e multigrafos, o que não é visto claramente nas interfaces do GraphX e Giraph que lidam com vértices e arestas. No próximo capítulo, discorrem-se as considerações finais e a perspectiva para os trabalhos futuros.

Gust/GraphX - LiveJournal																			
$\alpha = 0.05$		PR						SSSP						TRI					
		4	6	8	10	20	30	4	6	8	10	20	30	4	6	8	10	20	30
média	GUS	424.1	278.5	251.9	213.3	153.1	139.4	136.8	119.9	110.5	108.1	106.0	101.5	396.2	237.4	181.5	150.3	117.0	110.0
	GRA	496.0	428.2	380.9	353.1	234.7	238.7	220.7	197.1	209.8	196.9	93.7	92.5	562.0	424.3	360.8	327.8	186.1	203.8
mínimo	GUS	383.1	259.2	235.9	204.7	150.2	136.9	128.7	114.6	106.0	102.1	104.0	100.7	362.7	220.0	168.5	144.5	115.1	108.6
	GRA	346.9	358.5	265.1	336.8	221.5	228.2	167.7	173.7	125.4	164.5	89.8	90.3	386.8	336.6	235.3	313.4	173.5	185.2
máximo	GUS	447.5	310.0	264.8	222.3	155.9	140.3	144.1	127.7	116.7	115.1	107.5	102.2	414.0	253.5	192.8	157.6	118.7	111.5
	GRA	551.0	515.4	507.4	369.0	249.1	246.2	240.1	237.6	267.8	208.4	98.5	94.7	616.6	469.0	414.4	337.4	197.4	217.2
desvio	GUS	21.2	15.2	8.5	5.5	1.5	0.8	4.3	4.2	3.2	3.8	1.0	0.5	14.0	10.9	7.4	3.9	1.1	0.9
padrão	GRA	58.5	36.2	57.3	8.9	7.3	4.4	21.3	20.3	38.1	11.4	2.4	1.4	74.9	38.9	55.6	6.3	5.7	7.7
intervalo	GUS	15.7	11.3	6.3	4.1	1.1	0.6	3.2	3.1	2.4	2.8	0.7	0.4	10.4	8.1	5.5	2.9	0.8	0.6
confiança	GRA	43.3	26.8	42.4	6.6	5.4	3.2	15.8	15.1	28.2	8.5	1.8	1.0	55.5	28.8	41.2	4.7	4.2	5.7
GUS vs	média xi-yi	-71.9	-149.6	-129.1	-139.8	-81.7	-99.3	-83.9	-77.1	-99.3	-88.8	12.4	9.0	-165.8	-186.9	-179.3	-177.5	-69.1	-93.8
GRA	IC	15.8	8.9	19.9	1.5	2.3	1.4	7.1	6.4	14.0	3.2	0.6	0.4	24.7	11.8	19.5	1.1	1.8	2.7
Test-t	rejeita(Test-t<α)?	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	sim	sim	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	min-overhead %	**-0.3**	**-0.4**	**-0.5**	**-0.4**	**-0.4**	**-0.4**	**-0.5**	**-0.5**	**-0.6**	**-0.5**	0.11	0.08	**-0.4**	**-0.5**	**-0.6**	**-0.6**	**-0.4**	**-0.5**
	max-overhead %	**0.0**	**-0.3**	**-0.2**	**-0.4**	**-0.3**	**-0.4**	**-0.3**	**-0.3**	**-0.3**	**-0.4**	0.16	0.11	**-0.2**	**-0.4**	**-0.4**	**-0.5**	**-0.3**	**-0.4**

Gust/Giraph - LiveJournal											
$\alpha = 0.05$		PR			SSSP				TRI			
		6	8	10	4	6	8	10	4	6	8	10
média	GUS	278.53	251.87	213.27	136.81	119.93	110.46	108.10	396.23	237.40	181.46	150.32
	GIR	438.39	373.03	330.61	69.99	65.19	69.54	63.82	1304.18	1012.73	797.74	734.69
mínimo	GUS	259.20	235.89	204.67	128.74	114.59	105.96	102.08	362.68	219.97	168.52	144.50
	GIR	427.14	349.26	313.17	59.24	57.85	48.12	51.18	900.14	708.91	566.79	640.57
máximo	GUS	310.03	264.80	222.35	144.08	127.68	116.74	115.12	414.02	253.54	192.77	157.60
	GIR	447.65	392.17	344.84	84.46	73.96	81.64	75.75	1623.31	1280.23	976.92	875.87
desvio	GUS	15.22	8.52	5.48	4.29	4.22	3.24	3.83	14.04	10.90	7.40	3.88
padrão	GIR	5.53	12.90	7.57	9.05	4.34	8.72	7.15	248.31	179.87	129.28	62.08
intervalo	GUS	11.28	6.31	4.06	3.18	3.13	2.40	2.84	10.40	8.08	5.48	2.88
confiança	GIR	4.09	9.55	5.61	6.70	3.22	6.46	5.30	183.95	133.25	95.77	45.99
GUS vs GIR	média xi-yi	-159.86	-121.16	-117.33	66.82	54.74	40.91	44.28	-907.96	-775.33	-616.28	-584.37
	IC	3.91	1.79	0.98	2.00	0.53	2.27	1.35	92.25	66.37	47.86	22.86
Test-t	rejeita(Test-t<α)?	**sim**	**sim**	**sim**	sim	sim	sim	sim	**sim**	**sim**	**sim**	**sim**
	min-overhead %	**-0.40**	**-0.37**	**-0.38**	0.81	0.74	0.46	0.57	**-0.85**	**-0.91**	**-0.90**	**-0.86**
	max-overhead %	**-0.33**	**-0.28**	**-0.33**	1.10	0.94	0.72	0.82	**-0.55**	**-0.63**	**-0.65**	**-0.73**

Tabela 20 – Resultados LIV - GUS/GRA e GUS/GIR

Gust/GraphX - Pokec													
$\alpha = 0.05$		PR				SSSP				TRI			
		4	6	8	10	4	6	8	10	4	6	8	10
média	GUS	174.61	125.52	109.53	94.95	58.96	60.96	59.28	54.16	107.76	75.99	64.52	58.92
	GRA	240.37	259.95	201.68	188.76	118.13	113.82	113.13	114.40	261.78	220.53	192.41	188.49
mínimo	GUS	151.98	116.94	104.42	90.07	55.04	58.10	56.70	52.25	98.48	72.10	61.90	56.23
	GRA	174.29	225.39	150.39	173.39	96.10	103.61	88.87	86.56	182.70	182.75	131.14	159.85
máximo	GUS	183.68	134.69	114.81	99.09	62.17	65.75	62.20	56.58	111.97	80.43	67.10	61.38
	GRA	272.84	293.17	312.76	199.87	125.60	123.54	127.29	127.02	292.18	245.92	219.01	200.09
desvio	GUS	8.32	5.83	3.09	2.36	2.25	2.12	1.75	1.16	3.38	2.50	1.35	1.30
padrão	GRA	26.37	22.49	42.01	6.37	9.18	6.36	10.68	11.67	33.89	17.76	27.13	9.65
intervalo	GUS	6.16	4.32	2.29	1.75	1.66	1.57	1.30	0.86	2.50	1.85	1.00	0.96
confiança	GRA	19.54	16.66	31.12	4.72	6.80	4.72	7.91	8.64	25.10	13.16	20.10	7.15
GUS vs GRA	média xi-yi	-65.77	-134.43	-92.15	-93.81	-59.17	-52.86	-53.85	-60.24	-154.02	-144.54	-127.89	-129.57
	IC	7.44	6.61	15.50	1.60	2.91	1.80	3.61	4.17	12.10	6.06	10.19	3.30
Test-t	rejeita(Test-t<α)?	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	min-overhead %	**-0.38**	**-0.60**	**-0.62**	**-0.53**	**-0.57**	**-0.52**	**-0.56**	**-0.61**	**-0.69**	**-0.72**	**-0.77**	**-0.73**
	max-overhead %	**-0.17**	**-0.44**	**-0.29**	**-0.46**	**-0.43**	**-0.41**	**-0.39**	**-0.44**	**-0.48**	**-0.59**	**-0.56**	**-0.64**

Gust/Giraph - Pokec													
$\alpha = 0.05$		PR				SSSP				TRI			
		4	6	8	10	4	6	8	10	4	6	8	10
média	GUS	174.61	125.52	109.53	94.95	58.96	60.96	59.28	54.16	107.76	75.99	64.52	58.92
	GIR	260.45	196.35	174.35	159.77	51.02	51.78	51.22	51.79	230.08	180.01	151.73	145.86
mínimo	GUS	151.98	116.94	104.42	90.07	55.04	58.10	56.70	52.25	98.48	72.10	61.90	56.23
	GIR	249.33	186.69	156.52	143.74	44.04	43.92	42.67	42.22	207.38	153.00	138.13	122.60
máximo	GUS	183.68	134.69	114.81	99.09	62.17	65.75	62.20	56.58	111.97	80.43	67.10	61.38
	GIR	268.50	207.18	187.46	174.19	59.62	63.58	64.85	65.18	255.79	209.82	178.98	176.10
desvio	GUS	8.32	5.83	3.09	2.36	2.25	2.12	1.75	1.16	3.38	2.50	1.35	1.30
padrão	GIR	5.27	6.73	9.55	7.24	4.68	6.10	6.17	6.11	13.57	16.00	9.46	13.11
intervalo	GUS	6.16	4.32	2.29	1.75	1.66	1.57	1.30	0.86	2.50	1.85	1.00	0.96
confiança	GIR	3.90	4.98	7.08	5.36	3.47	4.52	4.57	4.53	10.05	11.85	7.01	9.71
GUS vs GIR	média xi-yi	-85.85	-70.83	-64.82	-64.82	7.95	9.18	8.06	2.36	-122.32	-104.02	-87.21	-86.94
	IC	1.35	0.47	2.56	1.94	0.99	1.62	1.77	1.99	4.13	5.30	3.20	4.65
Test-t	rejeita(Test-t<α)?	**sim**	**sim**	**sim**	**sim**	sim	sim	sim	**não**	**sim**	**sim**	**sim**	**sim**
	min-overhead %	**-0.37**	**-0.41**	**-0.43**	**-0.45**	0.06	0.06	0.04	-0.06	**-0.59**	**-0.65**	**-0.63**	**-0.67**
	max-overhead %	**-0.29**	**-0.31**	**-0.32**	**-0.36**	0.26	0.29	0.27	0.15	**-0.48**	**-0.50**	**-0.52**	**-0.52**

Tabela 21 – Resultados POK - GUS/GRA e GUS/GIR

Gust/GraphX - Skitter													
$\alpha = 0.05$		PR				SSSP				TRI			
		4	6	8	10	4	6	8	10	4	6	8	10
média	GUS	59.98	47.43	42.11	36.68	21.54	19.30	18.33	17.89	48.65	33.17	28.50	25.80
	GRA	150.83	208.57	253.69	252.69	95.76	114.85	138.53	140.37	156.22	142.65	172.54	175.94
mínimo	GUS	54.92	43.79	40.33	34.25	20.06	18.43	17.32	17.26	43.46	31.45	27.16	24.72
	GRA	132.49	167.08	113.48	120.15	89.52	99.19	73.74	113.91	129.72	126.50	138.69	130.09
máximo	GUS	63.44	50.48	44.18	37.88	23.20	20.23	19.18	18.86	51.90	34.39	29.58	26.82
	GRA	165.76	252.45	310.13	306.51	104.59	141.30	162.69	161.17	167.98	158.30	192.71	199.24
desvio	GUS	2.54	1.94	1.16	0.98	0.87	0.49	0.50	0.42	2.53	0.77	0.61	0.52
padrão	GRA	8.53	27.62	56.19	57.44	3.37	12.33	21.73	17.24	11.75	8.97	14.42	24.66
intervalo	GUS	1.88	1.43	0.86	0.73	0.65	0.36	0.37	0.31	1.87	0.57	0.45	0.39
confiança	GRA	6.32	20.46	41.63	42.55	2.50	9.14	16.10	12.77	8.70	6.65	10.69	18.27
GUS vs GRA	média xi-yi	-90.86	-161.14	-211.58	-216.00	-74.22	-95.55	-120.19	-122.47	-107.57	-109.49	-144.04	-150.13
	IC	2.48	10.10	21.66	22.15	1.00	4.65	8.36	6.61	3.74	3.22	5.43	9.50
Test-t	rejeita(Test-t<α)?	**sim**	sim	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	min-overhead %	**-0.66**	**-0.88**	**-1.00**	**-1.03**	**-0.81**	**-0.91**	**-0.99**	**-0.97**	**-0.76**	**-0.82**	**-0.90**	**-0.96**
	max-overhead %	**-0.55**	**-0.67**	**-0.67**	**-0.68**	**-0.74**	**-0.75**	**-0.75**	**-0.78**	**-0.62**	**-0.72**	**-0.77**	**-0.75**

Gust/Giraph - Skitter													
$\alpha = 0.05$		PR				SSSP				TRI			
		4	6	8	10	4	6	8	10	4	6	8	10
média	GUS	59.98	47.43	42.11	36.68	21.54	19.30	18.33	17.89	48.65	33.17	28.50	25.80
	GIR	240.56	177.43	165.32	152.59	54.46	50.98	53.33	55.41	873.53	782.88	608.76	612.67
mínimo	GUS	54.92	43.79	40.33	34.25	20.06	18.43	17.32	17.26	43.46	31.45	27.16	24.72
	GIR	228.34	166.81	147.03	136.17	47.05	43.81	46.68	43.99	693.85	549.94	453.43	455.70
máximo	GUS	63.44	50.48	44.18	37.88	23.20	20.23	19.18	18.86	51.90	34.39	29.58	26.82
	GIR	246.48	189.55	179.89	166.43	68.51	58.97	65.14	65.61	1047.26	1068.52	738.35	807.86
desvio	GUS	2.54	1.94	1.16	0.98	0.87	0.49	0.50	0.42	2.53	0.77	0.61	0.52
padrão	GIR	4.86	6.23	10.13	9.08	6.72	5.10	5.84	7.04	110.20	173.45	110.85	127.70
intervalo	GUS	1.88	1.43	0.86	0.73	0.65	0.36	0.37	0.31	1.87	0.57	0.45	0.39
confiança	GIR	3.60	4.62	7.50	6.73	4.98	3.78	4.33	5.22	81.63	128.49	82.12	94.60
GUS vs GIR	média xi-yi	-180.58	-129.99	-123.21	-115.90	-32.92	-31.68	-35.00	-37.52	-824.89	-749.71	-580.26	-586.87
	IC	1.02	1.72	3.53	3.19	2.32	1.82	2.11	2.60	42.23	67.70	43.23	49.86
Test-t	rejeita(Test-t<α)?	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	min-overhead %	**-0.77**	**-0.77**	**-0.80**	**-0.81**	**-0.71**	**-0.70**	**-0.74**	**-0.78**	**-1.04**	**-1.12**	**-1.09**	**-1.11**
	max-overhead %	**-0.73**	**-0.70**	**-0.69**	**-0.71**	**-0.50**	**-0.54**	**-0.57**	**-0.58**	**-0.85**	**-0.79**	**-0.82**	**-0.80**

Tabela 22 – Resultados SKI - GUS/GRA e GUS/GIR

Gust/GraphX - Amazon0302													
$\alpha = 0.05$		PR				SSSP				TRI			
		4	6	8	10	4	6	8	10	4	6	8	10
média	GUS	8.76	6.68	6.14	4.85	4.95	4.34	4.37	3.99	3.52	2.96	2.85	2.68
	GRA	76.76	99.34	108.11	113.09	65.42	74.69	80.71	85.89	59.13	66.57	72.38	71.57
mínimo	GUS	7.97	6.15	5.67	4.61	4.71	4.14	4.25	3.60	3.31	2.85	2.73	2.59
	GRA	73.01	94.01	100.92	90.27	61.61	69.03	75.24	77.40	56.61	57.69	65.23	55.61
máximo	GUS	9.44	7.05	6.47	5.07	5.17	4.47	4.46	4.20	3.71	3.07	2.96	2.76
	GRA	94.30	104.37	114.92	120.11	71.32	85.19	92.03	88.89	65.32	75.81	83.80	76.41
desvio	GUS	0.48	0.27	0.22	0.13	0.14	0.09	0.07	0.16	0.11	0.05	0.06	0.04
padrão	GRA	4.07	3.10	4.19	6.48	2.47	3.74	4.01	2.82	2.05	4.35	4.48	4.33
intervalo	GUS	0.35	0.20	0.16	0.10	0.10	0.06	0.05	0.12	0.08	0.04	0.04	0.03
confiança	GRA	3.02	2.30	3.10	4.80	1.83	2.77	2.97	2.09	1.52	3.22	3.32	3.20
GUS vs GRA	média xi-yi	-68.00	-92.66	-101.97	-108.24	-60.47	-70.35	-76.34	-81.91	-55.60	-63.62	-69.52	-68.89
	IC	1.48	1.11	1.56	2.50	0.92	1.44	1.55	1.05	0.76	1.69	1.73	1.68
Test-t	rejeita(Test-t<α)?	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	min-overhead %	**-0.93**	**-0.96**	**-0.97**	**-1.00**	**-0.95**	**-0.98**	**-0.98**	**-0.98**	**-0.97**	**-1.00**	**-1.01**	**-1.01**
	max-overhead %	**-0.84**	**-0.91**	**-0.91**	**-0.91**	**-0.89**	**-0.90**	**-0.91**	**-0.93**	**-0.91**	**-0.91**	**-0.91**	**-0.92**

Gust/Giraph - Amazon0302													
$\alpha = 0.05$		PR				SSSP				TRI			
		4	6	8	10	4	6	8	10	4	6	8	10
média	GUS	8.76	6.68	6.14	4.85	4.95	4.34	4.37	3.99	3.52	2.96	2.85	2.68
	GIR	75.76	66.80	64.46	64.64	41.87	41.81	43.12	43.00	42.77	39.84	41.61	44.26
mínimo	GUS	7.97	6.15	5.67	4.61	4.71	4.14	4.25	3.60	3.31	2.85	2.73	2.59
	GIR	69.17	58.80	58.62	58.21	36.02	37.20	38.81	38.61	35.34	35.64	35.08	38.23
máximo	GUS	9.44	7.05	6.47	5.07	5.17	4.47	4.46	4.20	3.71	3.07	2.96	2.76
	GIR	82.55	78.73	69.37	72.59	51.98	47.17	47.05	51.68	58.46	44.15	48.33	54.74
desvio	GUS	0.48	0.27	0.22	0.13	0.14	0.09	0.07	0.16	0.11	0.05	0.06	0.04
padrão	GIR	3.80	4.69	2.90	3.76	4.66	2.71	2.73	2.96	7.15	2.58	2.66	4.02
intervalo	GUS	0.35	0.20	0.16	0.10	0.10	0.06	0.05	0.12	0.08	0.04	0.04	0.03
confiança	GIR	2.82	3.47	2.15	2.78	3.45	2.01	2.02	2.20	5.30	1.91	1.97	2.98
GUS vs GIR	média xi-yi	-67.00	-60.12	-58.32	-59.79	-36.92	-37.47	-38.75	-39.02	-39.25	-36.88	-38.75	-41.59
	IC	1.31	1.74	1.05	1.42	1.78	1.03	1.04	1.11	2.76	0.99	1.02	1.56
Test-t	rejeita(Test-t<α)?	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	min-overhead %	**-0.93**	**-0.95**	**-0.94**	**-0.97**	**-0.97**	**-0.95**	**-0.95**	**-0.96**	**-1.04**	**-0.97**	**-0.98**	**-1.01**
	max-overhead %	**-0.84**	**-0.85**	**-0.87**	**-0.88**	**-0.80**	**-0.85**	**-0.85**	**-0.85**	**-0.79**	**-0.88**	**-0.88**	**-0.87**

Tabela 23 – Resultados AMA - GUS/GRA e GUS/GIR

7 CONCLUSÃO E PERSPECTIVAS DE TRABALHOS FUTUROS

O processamento e análise de grafos em larga escala ganhou nova perspectiva após o sucesso do MapReduce, embora esse modelo não tenha sido originalmente projetado para atender aos requisitos de algoritmos de processamento de grafos. Por isso, pesquisadores se esforçaram no desenvolvimento de soluções alternativas e especializadas, o que resultou em novos modelos e frameworks voltados ao processamento de grafos grandes (BigGraph). Esses sistemas têm sido significativamente influenciados pelo paradigma centrado em vértices (*vertex-centric*), ou "pensar como um vértice", introduzido pelo modelo Pregel, com implementação disseminada através do *framework* Giraph, o qual fez parte da análise experimental deste trabalho. Por sua vez, Pregel foi inspirado no modelo BSP, que prevê uma sequência de passos computacionais intercalados, compostos por processamento paralelo e sincronismo. Eventualmente, esse sincronismo é visto como passivo de otimização, o que motivou soluções como o modelo GAS (*Gather, Apply, Scatter*), implementado por sistemas como PowerGraph e GraphX, sendo esse último também utilizado na análise experimental deste trabalho. Apesar das inovações, atualmente a literatura não explora o uso do desenvolvimento baseado em componentes para atender aos requisitos de arcabouços de processamento paralelo de grafos.

Nesta Tese, verificou-se a hipótese de que sistemas de computação de grafos grandes, especialmente MapReduce e Pregel, podem ser beneficiados pela programação orientada a componentes paralelos. Considera-se que os benefícios não são apenas em desempenho, mas também na arquitetura, reusabilidade, modelagem, divisão em módulos ou partes bem definidas, como também adaptabilidade. Para validar essas afirmações, o autor desta Tese de Doutorado introduziu o Gust, um arcabouço BigGraph desenvolvido para a plataforma HPC Shelf, um conceito de plataforma orientada a componentes para provimento de serviços de computação de alto desempenho sob a abstração de nuvens computacionais.

Uma aplicação da HPC Shelf, sua porta de entrada para os seus usuários finais, ditos *usuários especialistas*, é vista sob a perspectiva *SaaS* (*Software-as-a-Service*), onde usuários especialistas descrevem problemas usando interfaces de alto nível. O suporte para execução dessas aplicações é oferecido através do arcabouço de aplicações SAFe, a perspectiva *PaaS* (*Platform-as-a-Service*) da HPC Shelf, através do qual *workflows* de ações computacionais oferecidas por componentes de sistemas de computação paralela podem ser orquestrados. Finalmente, é merecido destacar que a plataforma HPC Shelf suporta plataformas virtuais, uma abstração orientada a componentes (modelo Hash) para plataformas de computação paralela, as quais

visam representar a infraestrutura computacional da HPC Shelf, ou seja, sua perspectiva *IaaS* (*Infrastructure-as-a-Service*).

No desenvolvimento do Gust, exploram-se os benefícios advindos da programação orientada a componentes, definindo inicialmente um arcabouço de componentes para sistemas de computação paralela que implementam processamento MapReduce. Por sua vez, esse arcabouço foi desenvolvido a partir de componentes que representam agentes de mapeamento e redução, que se unem para formar *workflows* visando paralelismo de larga escala sobre múltiplas plataformas virtuais da HPC Shelf. Utilizando componentes do modelo Hash para HPC Shelf, a modularidade é atingida em ambiente distribuído através de conectores, os quais representam padrões de paralelismo pré-determinados suportados pela plataforma. Os componentes Hash são ainda segregados por *espécies* distintas, diferenciadas pelo seu *modelo de implantação*, *conexão* e *ciclo de vida*, onde se destacam, na HPC Shelf, as espécies *plataforma*, *computação*, *repositório de dados*, *conectores* e *ligações* (*bindings*) de ações e de serviços. Para serem adaptáveis, os componentes abstratos MapReduce declaram uma *assinatura contextual*, composta por um conjunto de *parâmetros de contexto*, que representam suposições sobre o contexto de execução. Dessa forma, a partir dos *argumentos de contexto*, determina-se um contexto de execução específico para uma computação MapReduce ou Gust.

Favorecido pela reusabilidade, Gust estende MapReduce lhe adicionando elementos fundamentais que suprem requisitos de sistemas de computação paralela voltados a grafos grandes. Essa extensão é inicialmente feita a partir dos componentes REDUCER e REDUCEFUNCTION, respectivamente constituindo os componentes GUSTY e GUSTYFUNCTION. Devido à extensão, além de se comportarem como redutores tradicionais do MapReduce, GUSTYFUNCTION e GUSTY têm como diferencial suas ferramentas voltadas a grafos, opcionalmente usadas pelo desenvolvedor. No caso de utilizadas, destacam-se os componentes GRAPH e INPUTBIN, que suportam a implementação dos algoritmos paralelos em grafos. Como benefício arquitetural, na modelagem e implementação do Gust, esses componentes permitem destacar a divisão das seguintes responsabilidades:

- INPUTBIN é o componente responsável pela estratégia de particionamento inicial do grafo, gerando subgrafos para serem processados posteriormente de forma distribuída. Sua implementação concreta atual efetua corte de vértices, técnica também usado no GraphX, ao invés de arestas, técnica usada em implementações Pregel, como Giraph. Dado o particionamento, instâncias INPUTBIN são entregues aos agentes GUSTY e suas funções

GUSTYFUNCTION, que alimentam o componente GRAPH;

- GRAPH é responsável por oferecer ao desenvolvedor os elementos de controle de vértices e arestas, atendendo aos princípios da teoria dos grafos. Sua construção envolve um componente aninhado chamado DATACONTAINER, visto como destinatário de dados referentes às arestas e vértices controlados por GRAPH. Além disso, a interface definida por GRAPH é genérica, aceitando tipos diversos, incluindo tipos primitivos como `int` e `long` vinculados aos vértices e arestas. A interface ainda pode ser de grafos orientados ou não orientados, suportando grafos simples e multigrafos, característica não trabalhada em *frameworks* como Giraph e GraphX, que veem os grafos apenas como orientados, não oferecendo ainda controles claros quanto aos multigrafos;
- GUSTYFUNCTION é responsável por suportar a codificação do algoritmo desenvolvido pelo programador. Para isso, aninha INPUTBIN e GRAPH, bem como oferece uma assinatura contextual que inclui chave/valor, usada para envio e recebimento de dados;
- GUSTY é responsável por controlar a entrada e saída de dados chave/valor do GUSTYFUNCTION. Além disso, controla as chamadas aos métodos `unroll`, `compute` e `scatter`, que definem as regras de programação aplicadas dentro da função de *redução* GUSTYFUNCTION.

A partir da divisão dessas responsabilidades e dos componentes envolvidos no MapReduce, tornou-se possível atingir os objetivos desta Teste, em relação à proposta do arcabouço de componentes paralelos denominado Gust:

- A **flexibilidade** do Gust pode ser atestada pela sua capacidade de atender aos modelos MapReduce e Pregel, bem como extensões existentes, tais como o GAS. Para isso, caso a codificação em GUSTYFUNCTION não utilize uma instância do componente GRAPH, Gust atende aos requisitos tradicionais de mapeamento e redução, pois estende o arcabouço MapReduce que implementa esse modelo. Por outro lado, para suportar Pregel, há duas possibilidades: o desenvolvedor pode trabalhar em um modelo MapReduce focado na redução, sem se preocupar com a implementação de MAPFUNCTION, que é oferecida de forma genérica; ou pode descartar agentes de mapeamento, usando apenas redutores, alterando o modelo MapReduce. Em ambos os casos, o componente GRAPH é fundamental para atender à especificação Pregel, que necessita de um modelo de programação voltado a vértice, padrão original de Pregel, ou de um modelo alterado voltado a bloco ou subgrafo, caso estendido do modelo centrado em vértices suportado por sistemas como GraphX.

Por Gust ser genérico para tipos de dados e computações, o foco em algum modelo de programação é diretamente relacionado com o tipo do par *chave/valor* utilizado no processamento MapReduce, atribuído via contrato contextual. Por exemplo, para atender ao modelo centrado em vértices, a chave deve ser um vértice. Por sua vez, para atender ao modelo centrado em blocos, a chave deve representar um bloco de vértices. Além disso, o componente GRAPH pode ser instanciado conforme o tipo do identificador de vértice. Por exemplo, tipos primitivos `int`, `uint`, `long`, ou tipos de objetos voltados às instâncias de componentes, como VERTEX, EDGE ou o próprio GRAPH. Para isso, usa-se o método construtor `Graph.newInstanceT<T>`, onde `T` define o tipo através do qual pode ainda ser informada a quantidade de vértices no subgrafo;

- A **extensibilidade** do Gust pode ser vista como a capacidade de incorporar novos modelos sem alterar o seu modelo original, que já suporta MapReduce, Pregel e, potencialmente, suas extensões conhecidas, conforme discutido no item anterior. Um sistema Gust consiste no uso de *workflows* científicos baseados na ligação entre agentes representados pelos componentes MAPPER, REDUCER e GUSTY, *ligadas* através de conectores SPLITTER e SHUFFLER, os quais possibilitam também comunicação com repositórios de dados arbitrários da HPC Shelf. Desde que se tenham portas adequadas, essas computações e repositórios de dados podem ser conectadas arbitrariamente pelos conectores, o que remete novamente à flexibilidade. A extensibilidade se dá pela capacidade de criar novos agentes de computação, estendendo REDUCER ou mesmo GUSTY. Além disso, pode-se incrementar o tradicional MapReduce ou Pregel. Por exemplo, com a utilização de múltiplos agentes REDUCER ou GUSTY. Nesse caso, é constituído um fluxo MapReduce ou Pregel de múltiplas etapas, possivelmente executados em plataformas virtuais diferentes e explorando o paralelismo *pipeline*;
- A **adaptabilidade**, em relação aos ambientes de execução, é uma consequência natural do sistema de contratos contextuais da HPC Shelf para escolha de componentes do *framework* de acordo com restrições impostas pelo provedor de aplicações ou mesmo pelos usuários especialistas, caso a aplicação permita. A situação mais simples é a coexistência de várias versões do componente GUSTY, bem como de um componente GUSTYFUNCTION, para diferentes plataformas de computação paralela, buscando explorar suas características arquiteturais a fim de alcançar o máximo de seu desempenho potencial. O mesmo é válido para a implementação do componente GRAPH, usado em implementações do

GUSTYFUNCTION. Por exemplo, tais componentes podem tentar fazer uso de aceleradores computacionais (GPUs, FPGAs ou MICs), quando esse tipo de recurso estiver presente na plataforma.

Para validar a parte de desempenho do arcabouço de componentes Gust, um estudo experimental aferiu resultados de medições usando métodos estatísticos para garantir confiabilidade nas conclusões a respeito dos dados gerados. Esse estudo envolveu dois *frameworks* voltados ao processamento de grafos dentro do estado-da-arte: GraphX e Giraph. Além disso, o Gust foi avaliado quanto à forma de particionamento, incluindo uma versão usando método aleatório, através de função *hash*, e outra versão usando a ferramenta de otimização Metis, que proporciona um balanceamento de carga mais sofisticado. As cargas e *benchmarks* foram selecionados de acordo com aquilo que foi extraído da literatura, incluindo os trabalhos que descrevem os próprios *frameworks* GraphX e Giraph. De uma forma geral, os resultados foram competitivos, tanto para métricas de desempenho de processamento quanto de uso da memória, apresentando respostas favoráveis ao Gust, tanto com carga balanceada quanto com carga aleatória. Destaca-se que as medições foram realizadas em um *cluster* único, modelo suportado por GraphX e Giraph, sem perda de generalidade para os objetivos do estudo experimental visto que o arcabouço não prevê a comunicação entre agentes de processamento (componentes MAPPER, REDUCER e GUSTY) entre cluster distintos, não sendo esperadas sobrecargas por esse motivo. Portanto, um dos diferenciais da HPC Shelf, que suporta plataformas virtuais ou multi-cluster, está aberto ainda para novos experimentos em grafos. Contudo, GraphX também oferece funcionalidades não suportadas pelo Gust, que podem ser vistas como trabalhos futuros a serem inseridos ao Gust, como tolerância a falhas e ambiente interativo com o usuário.

7.1 Trabalhos Futuros

Em seguida, descrevemos alguns trabalhos futuros que consideramos importantes:

- Tolerância a falhas: característica útil em sistemas distribuídos que processam dados. No caso do emprego de múltiplos clusters em um sistema de computação paralela, o requisito de tolerância a falhas é especialmente importante porque os pontos de falha podem ser significativos. Atualmente existem trabalhos no campo de linhagem de dados (*data lineage*), utilizadas no Spark. No caso da HPC Shelf, esse tipo de técnica pode atuar no salvamento e recuperação de informações sobre a execução de *workflows*, bem como de componentes de solução. Por exemplo, no caso do Gust, isso seria especialmente aplicado

aos dados do DATACONTAINER e chaves/valores do MapReduce;

- Ambiente de interatividade com o usuário: útil para a análise interativa das informações, permitindo ao usuário realizar consultas a dados do grafo, bem como executar algoritmos de forma interativa. Para isso, tem-se considerado o ambiente de avaliação de códigos C# do Mono, ou REPL (*Read-eval-print loop*), no qual é possível ao usuário entrar em um terminal de comandos e inserir códigos C# linha por linha, obtendo respostas em tempo de execução. De fato, os componentes implantados já ficam acessíveis por padrão em um ambiente REPL. Porém, seu comportamento interativo ainda não foi explorado e validado, especialmente em ambiente distribuído;
- Particionamento: atualmente Gust e MapReduce possuem técnicas aleatórias para particionar e balancear carga de dados e processamento. Um estudo nesse contexto poderia envolver a criação de novos algoritmos, bem como a avaliação de algoritmos e heurísticas já existentes, assim como foi feito com o Metis neste trabalho. Além disso, um particionamento dinâmico poderia ser incluído em cada iteração MapReduce, avaliando passo a passo onde se localizam gargalos, para tomada de decisão sobre reparticionamento.

REFERÊNCIAS

ABOU-RJEILI, A.; KARYPIS, G. Multilevel algorithms for partitioning power-law graphs. In: **Proceedings of the 20th International Conference on Parallel and Distributed Processing**. Washington, DC, USA: IEEE Computer Society, 2006. (IPDPS'06), p. 124–124. ISBN 1-4244-0054-6. Disponível em: <http://dl.acm.org/citation.cfm?id=1898953.1899055>.

AMAZON. **Amazon Elastic Compute Cloud (Amazon EC2)**. 2013. Disponível em: <http://aws.amazon.com/pt/ec2>. Acesso em: 06-08-2017.

APACHE. **Nutch Project**. 2003. Disponível em: <http://nutch.apache.org>. Acesso em: 06-08-2017.

APACHE. **Apache Hadoop**. 2005. Disponível em: <http://hadoop.apache.org>. Acesso em: 06-08-2017.

APACHE. **Apache Giraph**. USA: [s.n.], 2011. Disponível em: <http://giraph.apache.org>. Acesso em: 06-08-2017.

ARIDHI, S.; MONTRESOR, A.; VELEGRAKIS, Y. BLADYG: A Novel Block-Centric Framework for the Analysis of Large Dynamic Graphs. In: **Proceedings of the ACM Workshop on High Performance Graph Processing**. New York, NY, USA: ACM, 2016. (HPGP'16), p. 39–42. ISBN 978-1-4503-4350-3. Disponível em: <http://doi.acm.org/10.1145/2915516.2915525>.

ARMSTRONG, R.; KUMFERT, G.; MCINNES, L. C.; PARKER, S.; ALLAN, B.; SOTTILE, M.; EPPERLY, T.; DAHLGREN, T. The cca component model for high-performance scientific computing. **Concurr. Comput. : Pract. Exper.**, John Wiley and Sons Ltd., Chichester, UK, v. 18, n. 2, p. 215–229, 2006. ISSN 1532-0626. Disponível em: <http://dl.acm.org/citation.cfm?id=1107430.1107433>.

BACKUS, J. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. **Commun. ACM**, ACM, New York, NY, USA, v. 21, n. 8, p. 613–641, ago. 1978. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/359576.359579>.

BAHMANI, B.; CHAKRABARTI, K.; XIN, D. Fast personalized pagerank on mapreduce. In: **Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: ACM, 2011. (SIGMOD '11), p. 973–984. ISBN 978-1-4503-0661-4. Disponível em: <http://doi.acm.org/10.1145/1989323.1989425>.

BAILEY, D. H. **The NAS Parallel Benchmark CFD CODES**. 1991. Disponível em: <http://www.netlib.org/parkbench/html/npb.html>. Acesso em: 06-08-2017.

BAUDE, F.; CAROMEL, D.; DALMASSO, C.; DANELUTTO, M.; GETOV, W.; HENRIO, L.; PREZ, C. GCM: A Grid Extension to Fractal for Autonomous Distributed Components. **Annals of Telecommunications**, v. 64, n. 1, p. 5–24, 2009.

BESTA, M.; PODSTAWSKI, M.; GRONER, L.; SOLOMONIK, E.; HOEFLER, T. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In: **Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC'17)**. ACM, 2017. Disponível em: <http://www.unixer.de/~htor/publications/>.

BIGGS, N.; LLOYD, E. K.; WILSON, R. J. **Graph Theory 1736-1936**. New York, NY, USA: Clarendon Press, 1986. ISBN 0-198-53916-9.

BONDY, J. A.; MURTY, U. S. R. **Graph Theory With Applications**. [S.l.]: Elsevier Science Publishing Co., 1976. ISBN 0-444-19451-7.

BOSE, R.; FREW, J. Lineage retrieval for scientific data processing: A survey. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 37, n. 1, p. 1–28, mar. 2005. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/1057977.1057978>.

BRUNETON, E.; COUPAYE, T.; STEFANI, J. B. Recursive and dynamic software composition with sharing. In: **European Conference on Object Oriented Programming (ECOOP'2002)**. [S.l.: s.n.], 2002.

BU, Y. Pregelix: Dataflow-based big graph analytics. In: **Proceedings of the 4th Annual Symposium on Cloud Computing**. New York, NY, USA: ACM, 2013. (SOCC '13), p. 54:1–54:2. ISBN 978-1-4503-2428-1. Disponível em: <http://doi.acm.org/10.1145/2523616.2525962>.

BU, Y.; HOWE, B.; BALAZINSKA, M.; ERNST, M. D. Haloop: Efficient iterative data processing on large clusters. **Proc. VLDB Endow.**, VLDB Endowment, v. 3, n. 1-2, p. 285–296, set. 2010. ISSN 2150-8097. Disponível em: <http://dx.doi.org/10.14778/1920841.1920881>.

CARVALHO JUNIOR, F. H. de; CORRÊA, R. C. The Design of a CCA Framework with Distribution, Parallelism, and Recursive Composition. In: **CBHPC**. [S.l.: s.n.], 2010.

CARVALHO JUNIOR, F. H. de; CORRÊA, R. C.; LINS, R.; SILVA, J. C.; ARAÚJO, G. A. High Level Service Connectors for Components-Based High Performance Computing. In: **Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing**. [S.l.]: IEEE, 2007. p. 237–244.

CARVALHO JUNIOR, F. H. de; CORRÊA, R. C.; LINS, R. D.; SILVA, J. C.; ARAÚJO, G. A. On the Design of Abstract Binding Connectors for High Performance Computing Component Models. In: **Joint Conference on HPC Grid programming Environments and Components (HPC-GECO), and on Components and Frameworks for High Performance Computing (4^{th}CompFrame)**. [S.l.: s.n.], 2007.

CARVALHO JUNIOR, F. H. de; LINS, R. D. Separation of Concerns for Improving Practice of Parallel Programming. **INFORMATION, An International Journal**, International Information Institute, v. 8, n. 5, set. 2005. ISSN 1343-4500.

CARVALHO JUNIOR, F. H. de; LINS, R. D. An institutional theory for #-components. **Electronic Notes in Theoretical Computer Science**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 195, p. 113–132, jan. 2008. ISSN 1571-0661. Disponível em: <http://dx.doi.org/10.1016/j.entcs.2007.08.029>.

CARVALHO JUNIOR, F. H. de; LINS, R. D.; CORRÊA, R. C.; ARAÚJO, G. A. Towards an architecture for component-oriented parallel programming: Research articles. **Concurrency and Computation : Practice and Experience**, John Wiley and Sons Ltd., Chichester, UK, v. 19, n. 5, p. 697–719, abr. 2007. ISSN 1532-0626. Disponível em: <http://dx.doi.org/10.1002/cpe.v19:5>.

CARVALHO JUNIOR, F. H. de; REZENDE, C. A. Component-based refactoring of parallel numerical simulation programs: A case study on component-based parallel programming. In: **Proceedings of the 2011 23rd International Symposium on Computer Architecture and High Performance Computing**. Washington, DC, USA: IEEE Computer Society, 2011. (SBAC-PAD '11), p. 199–206. ISBN 978-0-7695-4573-8. Disponível em: <http://dx.doi.org/10.1109/SBAC-PAD.2011.28>.

CARVALHO JUNIOR, F. H. de; REZENDE, C. A. A Case Study on Expressiveness and Performance of Component-Oriented Parallel Programming. **Journal of Parallel and Distributed Computing**, v. 73, n. 5, p. 557–569, 2013. ISSN 0743-7315. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0743731512002882>.

CARVALHO JUNIOR, F. H. de; REZENDE, C. A.; SILVA, J. C.; ALAM, W. G. Al. Contextual Abstraction in a Type System for Component-Based High Performance Computing Platforms. In: **Lecture Notes in Computer Science**: Proceedings of the the Proceedings of the XVII Brazilian Symposium on Programming Languages (SBLP'2013). [S.l.]: Springer, 2013. v. 8129, p. 90–104.

CARVALHO JUNIOR, F. H. de; REZENDE, C. A.; SILVA, J. C.; ALAM, W. G. Al; ALENCAR, J. M. U. de. Contextual Abstraction in a Type System for Component-Based High Performance Computing Platforms. **Science of Computer Programming**, v. 132, p. 96–128, 2016. ISSN 0167-6423.

ÇATALYÜREK, Ü. V.; AYKANAT, C.; UÇAR, B. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. **SIAM J. Sci. Comput.**, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, v. 32, n. 2, p. 656–683, fev. 2010. ISSN 1064-8275. Disponível em: <http://dx.doi.org/10.1137/080737770>.

CHAVARRÍA-MIRANDA, D.; PANYALA, A.; MA, W.; PRANTL, A.; KRISHNAMOORTHY, S. Global transformations for legacy parallel applications via structural analysis and rewriting. **Parallel Computing**, v. 43, p. 1 – 26, 2015. ISSN 0167-8191. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0167819115000083>.

CHE, S. Gascl: A vertex-centric graph model for gpus. In: **High Performance Extreme Computing Conference (HPEC), 2014 IEEE**. [S.l.: s.n.], 2014. p. 1–6.

CHEN, Q.; BAI, S.; LI, Z.; GOU, Z.; SUO, B.; PAN, W. Graphhp: A hybrid platform for iterative graph processing. **CoRR**, abs/1706.07221, 2017. Disponível em: <http://arxiv.org/abs/1706.07221>.

CHEN, R.; WENG, X.; HE, B.; YANG, M. Large graph processing in the cloud. In: **Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: ACM, 2010. (SIGMOD '10), p. 1123–1126. ISBN 978-1-4503-0032-2. Disponível em: <http://doi.acm.org/10.1145/1807167.1807297>.

CHENG, R.; HONG, J.; KYROLA, A.; MIAO, Y.; WENG, X.; WU, M.; YANG, F.; ZHOU, L.; ZHAO, F.; CHEN, E. Kineograph: Taking the pulse of a fast-changing and connected world. In: **Proceedings of the 7th ACM European Conference on Computer Systems**. New York, NY, USA: ACM, 2012. (EuroSys '12), p. 85–98. ISBN 978-1-4503-1223-3. Disponível em: <http://doi.acm.org/10.1145/2168836.2168846>.

CHING, A.; EDUNOV, S.; KABILJO, M.; LOGOTHETIS, D.; MUTHUKRISHNAN, S. One trillion edges: Graph processing at facebook-scale. **Proc. VLDB Endow.**, VLDB Endowment, v. 8, n. 12, p. 1804–1815, ago. 2015. ISSN 2150-8097. Disponível em: <http://dx.doi.org/10.14778/2824032.2824077>.

CLOUDERA. **Cloudera**. 2013. Disponível em: <http://www.cloudera.com>. Acesso em: 06-08-2017.

COHEN, J. Graph twiddling in a mapreduce world. **Computing in Science and Engg.**, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 11, n. 4, p. 29–41, jul. 2009. ISSN 1521-9615. Disponível em: <http://dx.doi.org/10.1109/MCSE.2009.120>.

CONDIE, T.; CONWAY, N.; ALVARO, P.; HELLERSTEIN, J. M.; ELMELEEGY, K.; SEARS, R. **MapReduce Online**. [S.l.], 2009. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-136.html>.

CUTTING, D. **Apache Lucene**. 1999. Disponível em: <http://lucene.apache.org>. Acesso em: 06-08-2017.

DAMEVSKI, K.; ZHANG, K.; PARKER, S. Practical parallel remote method invocation for the babel compiler. In: **Proceedings of the 2007 symposium on Component and framework technology in high-performance and scientific computing**. New York, NY, USA: ACM, 2007. p. 131–140. ISBN 978-1-59593-867-1.

DATASTAX INC. **DataStax**. 2013. Disponível em: <http://www.datastax.com>. Acesso em: 06-08-2017.

DBMS2. **More patent nonsense - Google MapReduce**. 2010. Disponível em: <http://www.dbms2.com/2010/02/11/google-mapreduce-patent>. Acesso em: 06-08-2017.

DEAN, J.; GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In: **Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6**. Berkeley, CA, USA: USENIX Association, 2004. (OSDI'04), p. 10–10. Disponível em: <http://dl.acm.org/citation.cfm?id=1251254.1251264>.

DEAN, J.; GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. **Commun. ACM**, ACM, New York, NY, USA, v. 51, n. 1, p. 107–113, jan. 2008. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1327452.1327492>.

DIJKSTRA, E. W. The humble programmer. **Commun. ACM**, ACM, New York, NY, USA, v. 15, n. 10, p. 859–866, out. 1972. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/355604.361591>.

DOEKEMEIJER, N.; VARBANESCU, A. L. **A Survey of Parallel Graph Processing Frameworks**. [S.l.], 2014. Report number PDS-2014-003. Disponível em: <http://www.pds.ewi.tudelft.nl/fileadmin/pds/reports/2014/PDS-2014-003.pdf>.

DU BOIS, A. R. **Programação Funcional com a Linguagem Haskell**. n.d. Disponível em: <http://www.inf.ufpr.br/andrey/ci062/ProgramacaoHaskell.pdf>. Acesso em: 06-08-2017.

DURAN, A.; KLEMM, M. The Intel Many Integrated Core Architecture. In: **2012 International Conference on High Performance Computing and Simulation (HPCS)**. [S.l.]: IEEE Computer Society, 2012. p. 365–366. ISBN 978-1-4673-2359-8.

EKANAYAKE, J.; LI, H.; ZHANG, B.; GUNARATHNE, T.; BAE, S.-H.; QIU, J.; FOX, G. Twister: A runtime for iterative mapreduce. In: **Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing**. New York, NY, USA: ACM, 2010. (HPDC '10), p. 810–818. ISBN 978-1-60558-942-8. Disponível em: <http://doi.acm.org/10.1145/1851476.1851593>.

ELNIKETY, E.; ELSAYED, T.; RAMADAN, H. E. ihadoop: Asynchronous iterations for mapreduce. In: **Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science**. Washington, DC, USA: IEEE Computer Society, 2011. (CLOUDCOM '11), p. 81–90. ISBN 978-0-7695-4622-3. Disponível em: <http://dx.doi.org/10.1109/CloudCom.2011.21>.

FALOUTSOS, M.; FALOUTSOS, P.; FALOUTSOS, C. On power-law relationships of the internet topology. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 29, n. 4, p. 251–262, ago. 1999. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/316194.316229>.

FAN, Z.; QIU, F.; KAUFMAN, A.; STOVER, S. Yoakum. GPU Cluster for High Performance Computing. In: **Proceedings of the 2004 ACM/IEEE conference on Supercomputing (SC'04)**. IEEE Computer Society, 2004. p. 47–47. ISBN 0-7695-2153-3. Disponível em: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1392977>.

FILIPPIDOU, I.; KOTIDIS, Y. Online and on-demand partitioning of streaming graphs. In: **Proceedings of the 2015 IEEE International Conference on Big Data (Big Data)**. Washington, DC, USA: IEEE Computer Society, 2015. (BIG DATA '15), p. 4–13. ISBN 978-1-4799-9926-2. Disponível em: <https://doi.org/10.1109/BigData.2015.7363735>.

GANELIN, I.; ORHIAN, E.; SASAKI, K.; YORK, B. **Spark: Big Data Cluster Computing in Production**. [S.l.]: Wiley, 2016. ISBN 9781119254010.

GHEMAWAT, S.; GOBIOFF, H.; LEUNG, S.-T. The google file system. **SIGOPS Oper. Syst. Rev.**, ACM, New York, NY, USA, v. 37, n. 5, p. 29–43, out. 2003. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/1165389.945450>.

GONZALEZ, J. E.; LOW, Y.; GU, H.; BICKSON, D.; GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In: **Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2012. (OSDI'12), p. 17–30. ISBN 978-1-931971-96-6. Disponível em: <http://dl.acm.org/citation.cfm?id=2387880.2387883>.

GOOGLE. **Google Trends**. 2006. Disponível em: <http://www.google.com/trends>. Acesso em: 06-08-2017.

GOOGLE. **Open Patent Non-Assertion Pledge**. 2013. Disponível em: <http://www.google.com/patents/opnpledge/patents>. Acesso em: 06-08-2017.

GOVERNO USA. **United States Patent and Trademark Office**. 1975. Disponível em: <http://patft.uspto.gov/netahtml/PTO/srchnum.htm>. Acesso em: 06-08-2017.

HAN, M.; DAUDJEE, K. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 8, n. 9, p. 950–961, maio 2015. ISSN 2150-8097. Disponível em: <https://doi.org/10.14778/2777598.2777604>.

HARSHVARDHAN; FIDEL, A.; AMATO, N. M.; RAUCHWERGER, L. Kla: A new algorithmic paradigm for parallel graph computations. In: **Proceedings of the 23rd International Conference on Parallel Architectures and Compilation**. New York, NY, USA: ACM, 2014. (PACT '14), p. 27–38. ISBN 978-1-4503-2809-8. Disponível em: <http://doi.acm.org/10.1145/2628071.2628091>.

HEDLUND, B. **Understanding Hadoop Clusters and the Network**. 2011. Disponível em: <http://bradhedlund.com/?p=3108>. Acesso em: 06-08-2017.

HERBORDT, M. C.; VANCOURT, T.; GU, Y.; SUKHWANI, B.; CONTI, A.; MODEL, J.; DISABELLO, D. Achieving High Performance with FPGA-Based Computing. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 40, p. 50–57, March 2007. ISSN 0018-9162. Disponível em: <http://dl.acm.org/citation.cfm?id=1251558.1251716>.

HOQUE, I.; GUPTA, I. Lfgraph: Simple and fast distributed graph analytics. In: **Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems**. New York, NY, USA: ACM, 2013. (TRIOS '13), p. 9:1–9:17. ISBN 978-1-4503-2463-2. Disponível em: <http://doi.acm.org/10.1145/2524211.2524218>.

ISARD, M.; BUDIU, M.; YU, Y.; BIRRELL, A.; FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. **SIGOPS Oper. Syst. Rev.**, ACM, New York, NY, USA, v. 41, n. 3, p. 59–72, mar. 2007. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/1272998.1273005>.

JACKSON, K. **OpenStack Cloud Computing Cookbook**. [S.l.]: Packt Publishing, 2012. ISBN 1849517320, 9781849517324.

JACOB, B.; LARSON, J.; ONG, E. M×N Communication and Parallel Interpolation in Community Climate System Model Version 3 Using the Model Coupling Toolkit. **The International Journal of High Performance Computing Applications**, v. 19, n. 3, p. 293–307, 2005.

JAIN, N.; LIAO, G.; WILLKE, T. L. Graphbuilder: Scalable graph etl framework. In: **First International Workshop on Graph Data Management Experiences and Systems**. New York, NY, USA: ACM, 2013. (GRADES '13), p. 4:1–4:6. ISBN 978-1-4503-2188-4. Disponível em: <http://doi.acm.org/10.1145/2484425.2484429>.

JAIN, R. **The Art Of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling**. [S.l.]: John Wiley & Sons, INC, 1991.

KANG, U.; TONG, H.; SUN, J.; LIN, C.-Y.; FALOUTSOS, C. Gbase: An efficient analysis platform for large graphs. **The VLDB Journal**, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 21, n. 5, p. 637–650, out. 2012. ISSN 1066-8888. Disponível em: <http://dx.doi.org/10.1007/s00778-012-0283-9>.

KANG, U.; TSOURAKAKIS, C. E.; FALOUTSOS, C. Pegasus: Mining peta-scale graphs. **Knowl. Inf. Syst.**, Springer-Verlag New York, Inc., New York, NY, USA, v. 27, n. 2, p. 303–325, maio 2011. ISSN 0219-1377. Disponível em: <http://dx.doi.org/10.1007/s10115-010-0305-0>.

KARYPIS, G.; KUMAR, V. **METIS - Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0**. [S.l.], 1995.

KARYPIS, G.; KUMAR, V. Parallel multilevel k-way partitioning scheme for irregular graphs. In: **Proceedings of the 1996 ACM/IEEE Conference on Supercomputing**. Washington, DC, USA: IEEE Computer Society, 1996. (Supercomputing '96). ISBN 0-89791-854-1. Disponível em: <http://dx.doi.org/10.1145/369028.369103>.

KHAYYAT, Z.; AWARA, K.; ALONAZI, A.; JAMJOOM, H.; WILLIAMS, D.; KALNIS, P. Mizan: A system for dynamic load balancing in large-scale graph processing. In: **Proceedings of the 8th ACM European Conference on Computer Systems**. New York, NY, USA: ACM, 2013. (EuroSys '13), p. 169–182. ISBN 978-1-4503-1994-2. Disponível em: <http://doi.acm.org/10.1145/2465351.2465369>.

KNUPP, K. Observational analysis of a gust front to bore to solitary wave transition within an evolving nocturnal boundary layer. **Journal of the Atmospheric Sciences**, v. 63, n. 8, ago. 2006. ISSN 0022-4928. Disponível em: <http://dx.doi.org/10.1175/JAS3731.1>.

KYROLA, A.; BLELLOCH, G.; GUESTRIN, C. Graphchi: Large-scale graph computation on just a pc. In: **Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2012. (OSDI'12), p. 31–46. ISBN 978-1-931971-96-6. Disponível em: <http://dl.acm.org/citation.cfm?id=2387880.2387884>.

LESKOVEC, J.; KREVL, A. **SNAP Datasets: Stanford Large Network Dataset Collection**. 2014. Disponível em: <http://snap.stanford.edu/data>. Acesso em: 06-08-2017.

LIN, J.; DYER, C. **Data-Intensive Text Processing with MapReduce**. [S.l.]: Morgan and Claypool Publishers, 2010. ISBN 1608453421, 9781608453429.

LOW, Y.; GONZALEZ, J.; KYROLA, A.; BICKSON, D.; GUESTRIN, C.; HELLERSTEIN, J. M. Graphlab: A new parallel framework for machine learning. In: **Conference on Uncertainty in Artificial Intelligence (UAI)**. Catalina Island, California: [s.n.], 2010.

MALEWICZ, G.; AUSTERN, M. H.; BIK, A. J.; DEHNERT, J. C.; HORN, I.; LEISER, N.; CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In: **Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: ACM, 2010. (SIGMOD '10), p. 135–146. ISBN 978-1-4503-0032-2. Disponível em: <http://doi.acm.org/10.1145/1807167.1807184>.

MAROZZO, F.; TALIA, D.; TRUNFIO, P. P2p-mapreduce: Parallel data processing in dynamic cloud environments. **J. Comput. Syst. Sci.**, Academic Press, Inc., Orlando, FL, USA, v. 78, n. 5, p. 1382–1402, set. 2012. ISSN 0022-0000. Disponível em: <http://dx.doi.org/10.1016/j.jcss.2011.12.021>.

MCCUNE, R. R.; WENINGER, T.; MADEY, G. Thinking Like a Vertex: a Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing. **ACM Computing Surveys**, ACM, n. 2, nov. 2015.

MURRAY, D. G.; MCSHERRY, F.; ISAACS, R.; ISARD, M.; BARHAM, P.; ABADI, M. Naiad: A timely dataflow system. In: **Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles**. New York, NY, USA: ACM, 2013. (SOSP '13), p. 439–455. ISBN 978-1-4503-2388-8. Disponível em: <http://doi.acm.org/10.1145/2517349.2522738>.

MURTHY, A.; MARKHAM, J.; VAVILAPALLI, V. K.; EADLINE, D. **Apache Hadoop YARN: Moving beyond MapReduce and Batch Processing with Apache Hadoop 2**. [S.l.]: Addison-Wesley Professional, 2014. ISBN 0321934504, 978-0321934505.

NAJEEBULLAH, K.; KHAN, K.; NAWAZ, W.; LEE, Y. BPP: large graph storage for efficient disk based processing. **CoRR**, abs/1401.2327, 2014. Disponível em: <http://arxiv.org/abs/1401.2327>.

NAVEH, B.; SICHI, J. **Projeto JGraphT**. 2003. Disponível em: <http://jgrapht.org>. Acesso em: 06-08-2017.

NIELSEN, R. **Statistical Methods in Molecular Evolution**. Springer New York, 2005. (Statistics for Biology and Health). ISBN 9780387223339. Disponível em: <https://books.google.com.br/books?id=nJipT3toWFAC>.

ORACLE. **Oracle: Big Data for the Enterprise**. 2013. Disponível em: <http://www.oracle.com/us/products/database/big-data-for-enterprise-519135.pdf>. Acesso em: 06-08-2017.

PAGE, L.; BRIN, S.; MOTWANI, R.; WINOGRAD, T. **The PageRank Citation Ranking: Bringing Order to the Web.** [S.l.], 1999. Previous number = SIDL-WP-1999-0120.

PARASHAR, M.; LI, X.; PARKER, S. G.; DAMEVSKI, K.; KHAN, A.; SWAMINATHAN, A.; JOHNSON, C. R. Advanced Computational Infrastructures for Parallel/Distributed Adaptive Applications. In: ____. [S.l.]: Wiley Press, 2009. cap. The SCIJump Framework for Parallel and Distributed Scientific Computing.

PLIMPTON, S. J.; DEVINE, K. D. Mapreduce in mpi for large-scale graph algorithms. **Parallel Comput.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 37, n. 9, p. 610–632, set. 2011. ISSN 0167-8191. Disponível em: <http://dx.doi.org/10.1016/j.parco.2011.02.004>.

POWER, R.; LI, J. Piccolo: Building fast, distributed programs with partitioned tables. In: **Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2010. (OSDI'10), p. 1–14. Disponível em: <http://dl.acm.org/citation.cfm?id=1924943.1924964>.

QIN, L.; YU, J. X.; CHANG, L.; CHENG, H.; ZHANG, C.; LIN, X. Scalable big graph processing in mapreduce. In: **Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: ACM, 2014. (SIGMOD '14), p. 827–838. ISBN 978-1-4503-2376-5. Disponível em: <http://doi.acm.org/10.1145/2588555.2593661>.

REZENDE, C. A. **Avaliação de Desempenho de uma Plataforma de Componentes Paralelos**. Dissertação (Dissertação de mestrado) — Universidade Federal do Ceará, Fortaleza-CE, sep 2011.

ROY, A.; MIHAILOVIC, I.; ZWAENEPOEL, W. X-stream: Edge-centric graph processing using streaming partitions. In: **Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles**. New York, NY, USA: ACM, 2013. (SOSP '13), p. 472–488. ISBN 978-1-4503-2388-8. Disponível em: <http://doi.acm.org/10.1145/2517349.2522740>.

SAGHARICHIAN, M.; NADERI, H.; HAGHJOO, M. Expregel: a new computational model for large-scale graph processing. **Concurrency and Computation: Practice and Experience**, 2015. ISSN 1532-0634. Disponível em: <http://dx.doi.org/10.1002/cpe.3482>.

SAKR, M. G. Sherif. **Large Scale and Big Data: Processing and Management**. USA: Auerbach Publications, 2014. ISBN 9781466581500.

SALIHOGLU, S.; WIDOM, J. Gps: A graph processing system. In: **Proceedings of the 25th International Conference on Scientific and Statistical Database Management**. New York, NY, USA: ACM, 2013. (SSDBM), p. 22:1–22:12. ISBN 978-1-4503-1921-8. Disponível em: <http://doi.acm.org/10.1145/2484838.2484843>.

SHAO, B.; WANG, H.; LI, Y. Trinity: A distributed graph engine on a memory cloud. In: **Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: ACM, 2013. (SIGMOD '13), p. 505–516. ISBN 978-1-4503-2037-5. Disponível em: <http://doi.acm.org/10.1145/2463676.2467799>.

SHAPOSHNIK, R.; MARTELLA, C.; LOGOTHETIS, D. **Practical Graph Analytics with Apache Giraph**. 1st. ed. Berkely, CA, USA: Apress, 2015. ISBN 1484212525, 9781484212523.

SHVACHKO, K.; KUANG, H.; RADIA, S.; CHANSLER, R. The hadoop distributed file system. In: **Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on**. [S.l.: s.n.], 2010. p. 1–10.

SILVA, J. C.; CARVALHO JUNIOR, F. H. de. A Platform of Scientific Workflows for Orchestration of Parallel Components in a Cloud of High Performance Computing Applications. In: **Lecture Notes in Computer Science**: Proceedings of the XX Brazilian Symposium on Programming Languages (SBLP'2016). [S.l.]: Springer, 2016. v. 9889, p. 156–170.

SIMMHAN, Y.; KUMBHARE, A.; WICKRAMAARACHCHI, C.; NAGARKAR, S.; RAVI, S.; RAGHAVENDRA, C.; PRASANNA, V. GoFFish: A sub-graph centric framework for large-scale graph analytics. In: **Euro-Par 2014 Parallel Processing**. Springer International Publishing, 2014. p. 451–462. Disponível em: <http://dx.doi.org/10.1007/978-3-319-09873-9_38>.

STANTON, I.; KLIOT, G. Streaming graph partitioning for large distributed graphs. In: **Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining**. New York, NY, USA: ACM, 2012. (KDD '12), p. 1222–1230. ISBN 978-1-4503-1462-6. Disponível em: <http://doi.acm.org/10.1145/2339530.2339722>.

STUTZ, P.; BERNSTEIN, A.; COHEN, W. Signal/collect: Graph algorithms for the (semantic) web. In: **Proceedings of the 9th International Semantic Web Conference on The Semantic Web - Volume Part I**. Berlin, Heidelberg: Springer-Verlag, 2010. (ISWC'10), p. 764–780. ISBN 3-642-17745-X, 978-3-642-17745-3. Disponível em: <http://dl.acm.org/citation.cfm?id=1940281.1940330>.

SURI, S.; VASSILVITSKII, S. Counting triangles and the curse of the last reducer. In: **Proceedings of the 20th International Conference on World Wide Web**. New York, NY, USA: ACM, 2011. (WWW '11), p. 607–614. ISBN 978-1-4503-0632-4. Disponível em: <http://doi.acm.org/10.1145/1963405.1963491>.

SVENDSEN, M. S. **Mining maximal cliques from large graphs using MapReduce**. Dissertação (Dissertação de mestrado) — Iowa State University, Ames, Iowa, USA, 2012.

TIAN, Y.; BALMIN, A.; CORSTEN, S. A.; TATIKONDA, S.; MCPHERSON, J. From "think like a vertex"to "think like a graph". **Proc. VLDB Endow.**, VLDB Endowment, v. 7, n. 3, p. 193–204, nov. 2013. ISSN 2150-8097. Disponível em: <http://dx.doi.org/10.14778/2732232.2732238>.

TSOURAKAKIS, C.; GKANTSIDIS, C.; RADUNOVIC, B.; VOJNOVIC, M. Fennel: Streaming graph partitioning for massive scale graphs. In: **Proceedings of the 7th ACM International Conference on Web Search and Data Mining**. New York, NY, USA: ACM, 2014. (WSDM '14), p. 333–342. ISBN 978-1-4503-2351-2. Disponível em: <http://doi.acm.org/10.1145/2556195.2556213>.

VALIANT, L. G. A bridging model for parallel computation. **Commun. ACM**, ACM, New York, NY, USA, v. 33, n. 8, p. 103–111, aug 1990. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/79173.79181>.

VENKATARAMAN, S.; BODZSAR, E.; ROY, I.; AUYOUNG, A.; SCHREIBER, R. S. Presto: Distributed machine learning and graph processing with sparse matrices. In: **Proceedings of the 8th ACM European Conference on Computer Systems**. New York, NY, USA: ACM, 2013. (EuroSys '13), p. 197–210. ISBN 978-1-4503-1994-2. Disponível em: <http://doi.acm.org/10.1145/2465351.2465371>.

WANG, G.; XIE, W.; DEMERS, A. J.; GEHRKE, J. Asynchronous large-scale graph processing made easy. In: **CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings**. www.cidrdb.org, 2013. Disponível em: <http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper58.pdf>.

WHITE, T. **Hadoop: The Definitive Guide**. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2009. ISBN 0596521979, 9780596521974.

Xamarin. **Projeto Mono**. 2004. Disponível em: <http://mono-project.com>. Acesso em: 06-08-2017.

XIA, Y.; NG, T. E.; SUN, X. S. Blast: Accelerating high-performance data analytics applications by optical multicast. Rice University, China, 2015. Disponível em: <http://www.cs.rice.edu/~eugeneng/papers/INFOCOM15-Blast.pdf>.

XIE, C.; CHEN, R.; GUAN, H.; ZANG, B.; CHEN, H. Sync or async: Time to fuse for distributed graph-parallel computation. In: **Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**. New York, NY, USA: ACM, 2015. (PPoPP 2015), p. 194–204. ISBN 978-1-4503-3205-7. Disponível em: <http://doi.acm.org/10.1145/2688500.2688508>.

XIN, R. S.; GONZALEZ, J. E.; FRANKLIN, M. J.; STOICA, I. Graphx: A resilient distributed graph system on spark. In: **First International Workshop on Graph Data Management Experiences and Systems**. New York, NY, USA: ACM, 2013. (GRADES '13), p. 2:1–2:6. ISBN 978-1-4503-2188-4. Disponível em: <http://doi.acm.org/10.1145/2484425.2484427>.

XU, N.; CUI, B.; CHEN, L.; HUANG, Z.; SHAO, Y. Heterogeneous environment aware streaming graph partitioning. **Knowledge and Data Engineering, IEEE Transactions on**, v. 27, n. 6, p. 1560–1572, June 2015. ISSN 1041-4347.

ZAHARIA, M.; CHOWDHURY, M.; FRANKLIN, M. J.; SHENKER, S.; STOICA, I. Spark: Cluster computing with working sets. In: **Proceedings of the 2Nd USENIX Conference on Hot**

Topics in Cloud Computing. Berkeley, CA, USA: USENIX Association, 2010. (HotCloud'10), p. 10–10. Disponível em: <http://dl.acm.org/citation.cfm?id=1863103.1863113>.

ZAHARIA, M.; KONWINSKI, A.; JOSEPH, A. D.; KATZ, R.; STOICA, I. Improving mapreduce performance in heterogeneous environments. In: **Proceedings of the 8th USENIX conference on Operating systems design and implementation**. Berkeley, CA, USA: USENIX Association, 2008. (OSDI'08), p. 29–42. Disponível em: <http://dl.acm.org/citation.cfm?id=1855741.1855744>.

ZHANG, L.; PARASHAR, M. Enabling Efficient and Flexible Coupling of Parallel Scientific Applications. In: **20th International of Parallel and Distributed Processing Symposium (IPDPS'2006)**. [S.l.]: IEEE Computer Society, 2006. p. 117–127.

ZHANG, Y.; CHEN, S. I2mapreduce: Incremental iterative mapreduce. In: **Proceedings of the 2Nd International Workshop on Cloud Intelligence**. New York, NY, USA: ACM, 2013. (Cloud-I '13), p. 3:1–3:4. ISBN 978-1-4503-2108-2. Disponível em: <http://doi.acm.org/10.1145/2501928.2501930>.

ZHANG, Y.; GAO, Q.; GAO, L.; WANG, C. imapreduce: A distributed computing framework for iterative computation. **Journal of Grid Computing**, Springer Netherlands, v. 10, n. 1, p. 47–68, 2012. ISSN 1570-7873. Disponível em: <http://dx.doi.org/10.1007/s10723-012-9204-9>.