UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MESTRADO ACADÊMICO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS PINHEIRO DE QUEIROZ

UM MECANISMO DE PROVENIÊNCIA PARA UMA NUVEM DE SERVIÇOS DE COMPUTAÇÃO DE ALTO DESEMPENHO ORIENTADA A COMPONENTES

FORTALEZA

2018

LUCAS PINHEIRO DE QUEIROZ

UM MECANISMO DE PROVENIÊNCIA PARA UMA NUVEM DE SERVIÇOS DE COMPUTAÇÃO DE ALTO DESEMPENHO ORIENTADA A COMPONENTES

Dissertação apresentada ao Curso de Mestrado Acadêmico 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 mestre em Ciência da Computação. Área de Concentração: Computação de Alto Desempenho

Orientador: Prof. Dr. Francisco Heron de Carvalho Júnior

FORTALEZA

2018

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

Q45m Queiroz, Lucas Pinheiro de.
Um mecanismo de proveniência para uma nuvem de serviços de computação de alto desempenho orientada a componentes / Lucas Pinheiro de Queiroz. – 2018.
109 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2018.
Orientação: Prof. Dr. Francisco Heron de Carvalho Júnior.

1. Proveniência. 2. Workflows científicos. 3. Desenvolvimento baseado em componentes. 4. Nuvens computacionais. I. Título.

CDD 005

LUCAS PINHEIRO DE QUEIROZ

UM MECANISMO DE PROVENIÊNCIA PARA UMA NUVEM DE SERVIÇOS DE COMPUTAÇÃO DE ALTO DESEMPENHO ORIENTADA A COMPONENTES

Dissertação apresentada ao Curso de Mestrado Acadêmico 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 mestre em Ciência da Computação. Área de Concentração: Computação de Alto Desempenho

Aprovada em: 30 de Novembro de 2018

BANCA EXAMINADORA

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

Prof. Dr. Dalvan Jair Griebler
Sociedade Educacional Três de Maio (SETREM)

Prof. Dr. Javam de Castro Machado
Universidade Federal do Ceará (UFC)

Prof. Dr. Allberson Bruno de Oliveira Dantas
Universidade da Integração Internacional da Lusofonia Afro-Brasileira (UNILAB)

À minha Mãe Lídia, por acreditar e investir em mim.

AGRADECIMENTOS

RÀ minha mãe, Dona Lídia, que me ensinou o melhor aprendizado: a vontade de aprender.

Ao meu lar que renova minhas forças na luta de cada dia: esposa Luana Marcelino, enteado Ian Marcelino e filho Tiê Queiroz. Muito obrigado Mamor!

Ao meu pai Ataias pelo afeto e carinho.

Ao meu orientador Prof. Francisco Heron de Carvalho Júnior pelo incentivo, dedicação e paciência na condução atenciosa dessa dissertação de Mestrado. Não posso deixar de enfatizar a formação exemplar durante grande parte da vida acadêmica com seus direcionamentos.

Aos professores da Banca, Prof. Allberson, Prof. Javam e Prof. Dalvan Griebler, pelos conselhos, sugestões de melhoria e destreza no olhar para o aprimoramento deste trabalho.

Aos parentes (irmãs, ti@s, prim@s e afins) pelo apoio e encorajamento. Ao meu irmão Leo pelas palavras de ânimo.

Aos amigos pelo incentivo, espontaneidade e diversão em certos momentos.

Aos colegas e profissionais do Programa de Pós-Graduação do Departamento de Computação da Universidade Federal do Ceará.

Aos Laboratórios LIA e LSBD pela disposição de recursos materiais.

Finalmente, à Deus pelo dom da vida e pela realização desta importante etapa da minha vida.

"Quando a ciência entrar em teu coração e a sabedoria for doce em tua alma, pede e te será dado."

(Arcano 7 – O Triunfo)

RESUMO

Experimentos científicos fazem cada vez mais uso de computações intensivas sobre dados em larga escala, exigindo, muitas vezes, a disponibilidade de recursos de Computação de Alto Desempenho. À medida que os experimentos aumentam em escala e complexidade, a capacidade de reprodução e entendimento dos experimentos adquirem maior dificuldade. Dessa forma, os dados de proveniência (dados produzidos ao longo do ciclo de vida dos experimentos) cumpre um papel fundamental para a transparência de experimentos, avaliação da qualidade dos dados, suporte à reprodutibilidade computacional e melhoria nos processos de gerência das informações. Os experimentos, representados através de *workflows* científicos, beneficiam-se também no reuso, manutenção e evolução de seus elementos. Os dados de proveniência pode ser classificada como prospectiva e retrospectiva. A proveniência prospectiva representa a especificação de tarefas computacionais para se alcançar um resultado. A proveniência retrospectiva consiste em um histórico estruturado e detalhado da execução de tarefas computacionais. O objetivo dessa dissertação é apresentar uma abordagem para a proveniência prospectiva e retrospectiva no contexto da HPC Shelf visando o compartilhamento e reuso de *workflows* em projetos de aplicação e o suporte a reprodutibilidade de execução de *workflows*. A HPC Shelf é uma proposta de plataforma de serviços de HPC em nuvem voltados para a composição, implantação e execução de sistemas de computação paralela de larga escala baseados em componentes. Nesse sentido, a HPC Shelf com informações de proveniência habilitadas obtém um melhor entendimento do comportamento dos experimentos, maior facilidade no diagnóstico de problemas, interpretação de resultados, identificação de regiões críticas de execução, entre outros usos. Para validação de nossa abordagem, apresentamos um arcabouço para construção de sistemas Gust operando sobre um conjunto de problemas da teoria dos grafos.

Palavras-chave: proveniência; workflows científicos; desenvolvimento baseado em componentes; nuvens computacionais.

ABSTRACT

Scientific experiments are increasingly using large-scale data intensive computations, often requiring the availability of High Performance Computing resources. As experiments increase in scale and complexity, the ability to reproduce and understand the experiments becomes more difficult. Thus, the provenance data (data produced throughout the life cycle of the experiments) comply with a fundamental role for the transparency of experiments, evaluation of data quality, support to computational reproducibility and improvement in information management processes. The experiments, represented through scientific workflows, also benefit in the reuse, maintenance and evolution of its elements. The provenance data can be classified as prospective and retrospective. The prospective provenance represents the specification of computational tasks to achieve a result. The retrospective provenance consists of a structured and detailed history of the execution of computational tasks. The objective of this dissertation is to present an approach to the prospective and retrospective provenance in the context of HPC Shelf, aiming at the sharing and reuse of workflows in application projects and the support of the reproducibility of execution of workflows. HPC Shelf is a proposed HPC cloud service platform for the composition, deployment and execution of large-scale component-based parallel computing systems. In this way, HPC Shelf with registered provenance information obtains a better understanding of the behavior of the experiments, greater ease in problem diagnosis, interpretation of results, identification of critical regions of execution, among other uses. To validate our approach, we present a framework for building Gust systems operating on a set of graph theory problems.

Keywords: provenance; scientific workflows; component based development; computational clouds.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE CÓDIGOS-FONTE

1 INTRODUÇÃO

Experimentos científicos, bem como simulações computacionais em geral, também de interesse das engenharias, geralmente lidam com grandes quantidades de dados, exigindo, muitas vezes, a disponibilidade de recursos de Computação de Alto Desempenho (HPC[1]). Nas tarefas computacionais de tais aplicações oriundas das ciências computacionais e engenharia, essas computações são comumente de longa duração, fazendo com que um usuário especialista de domínio (cientista, engenheiro, etc.) tenha que dividir o experimento em múltiplas partes, estrategicamente, para que possa tomar decisões levando em conta os resultados intermediários e, assim, definir os próximos passos a serem executados. Esse processamento envolve a execução de um fluxo de atividades que podem ser modeladas através de *workflows* científicos.

A tecnologia de *workflows* permite guiar as etapas da computação em tempo de execução sem a intervenção manual do especialista. Dessa forma, experimentos científicos podem ser ajustados ou reprojetados para fazer uso de *workflows*, uma vez que fornecem todas as abstrações necessárias, propiciando o uso efetivo de recursos computacionais e o desenvolvimento de ambientes robustos de resolução de problemas (aplicações), que coordenam os recursos de HPC.

O uso de *workflows* emergiu oriundos de aplicações do meio comercial e corporativo, visando definir fluxos de tarefas que compõem processos de negócio de uma organização, automatizando processos industriais e gerenciais. Diversos modelos, técnicas e ferramentas foram desenvolvidos para apoiar o ciclo de vida dos *workflows*, fato que aproximou cada vez mais a indústria e a academia nesse sentido. A comunidade científica, por sua vez, adotou o uso de *workflows*, contudo, porém, devido ás suas particularidades, desenvolveu padrões e técnicas específicas para o gerenciamento de *workflows* ditos científicos (BARGA; GANNON, 2007).

A fim de reduzir o tempo de processamento em computações científicas de longa duração, os *workflows* podem ser executados fazendo uso de técnicas de processamento paralelo, em plataformas voltadas a aplicações de HPC. Porém, laboratórios científicos geralmente não possuem à sua disposição especialistas em computação para especificar *workflows* do seu interesse, bem como implementar as atividades de processamento envolvidas seguindo as melhores práticas de programação e sendo capazes de lidar com a heterogeneidade das plataformas paralelas. Dessa forma, em geral, os próprios cientistas implementam essas atividades através de *scripts* com linguagens de alto nível e softwares matemáticos ou estatísticos com funções

[1] Sigla em inglês para *High Performance Computing*.

pré-definidas e abstrações apropriadas para a sua composição.

Para possibilitar uma melhor gerência na concepção, execução e análise de *workflows*, são comumente empregados sistemas gerenciadores de *workflows* científicos (SGWfCs), visando auxiliar os cientistas a focar em suas pesquisas ao tornar a tarefa de projetar os *workflows* menos onerosa. Os SGWfCs devem permitir o gerenciamento e a exploração dos recursos de computação distribuídos. As tarefas executadas nesses sistemas devem lidar com diversos tipos de paralelismo e abordagens de escalonamento nos ambientes de execução.

Com o objetivo de suportar uma melhor análise de *workflows* científicos, os SGWfCs habilitam recursos de proveniência, que fornece informações relacionadas a diversas fases do ciclo de vida de um *workflow*. As informações úteis à proveniência são dados e metadados capturados durante todo o processo de derivação de um resultado, incluindo as fontes de dados originais, dados intermediários e as etapas computacionais do *workflow* que foram realmente executadas para produzir tal resultado.

Com relação a experimentos modelados como *workflows* científicos, a proveniência pode ser classificada como prospectiva e retrospectiva. A proveniência prospectiva representa a especificação de tarefas computacionais em si, ou seja, corresponde à estruturação dos passos computacionais a serem seguidos para se alcançar um resultado. A proveniência retrospectiva consiste em um histórico estruturado e detalhado da execução de tarefas computacionais (metadados associados à execução de atividades e características do ambiente).

Salientamos que o conhecimento *a priori* da estrutura do *workflow* e a possibilidade de análise dos resultados intermediários a partir da execução do *workflow* podem auxiliar bastante o cientista no momento da avaliação do método empregado no experimento, permitindo identificar gargalos na execução e até mesmo rejeição de sua hipótese científica antes mesmo de finalizar completamente o experimento. No caso de falha na execução de um *workflow*, uma possível reexecução do *workflow* pode também ser otimizada executando apenas as etapas ainda não realizadas e aproveitando os dados intermediários processados anteriormente.

Os dados de proveniência proporcionam também a viabilidade de reproduzir um resultado de experimento visto que todas as etapas e dados utilizados são armazenados. Isso aumenta a transparência na validação dos resultados, reduz a rejeição da comunidade científica, torna mais fácil a colaboração e constitui a base do método científico. O desafio é assegurar que os dados de proveniência são suficientes e completos para suportar a reprodutibilidade.

Neste trabalho de pesquisa, estudamos os SGWfCs mais populares e descritos na

literatura como estado-da-arte, a fim de investigar seus atributos de proveniência. São eles: Vistrails[2], Taverna[3] e o Kepler[4]. O propósito desse estudo é a necessidade de inserir recursos de proveniência sobre a plataforma HPC Shelf, apresentada a seguir.

1.1 HPC Shelf

A HPC Shelf é uma proposta de uma plataforma de serviços de HPC em nuvem voltados para a composição, implantação e execução de sistemas de computação paralela de larga escala baseados em componentes. Tais serviços são oferecidos para *aplicações*, os quais servem como pontos de entrada para usuários finais, chamados de *especialistas*, interessados em resolver problemas dentro de um determinado domínio que demandam computação intensiva. As aplicações fornecem abstrações de alto nível para especificação de problemas e sintetizam soluções baseadas em *sistemas de computação paralela* da HPC Shelf.

A HPC Shelf é compatível com o modelo Hash de componentes paralelos (CARVALHO-JUNIOR; LINS, 2008), suportando um conjunto de espécies de componentes que representam diferentes elementos de composição para a construção de sistemas de computação paralela de larga escala: **plataformas virtuais**, **computações**, **fontes de dados**, **conectores** e **ligações** (*bindings* de serviços e de ações).

Em torno da HPC Shelf, atuam intervenientes pertencentes a uma das classes a seguir:

- *especialistas*, interessados em fazer uso das aplicações para resolver problemas do seu interesse usando interfaces de alto nível e linguagens de domínio específico;
- *provedores de aplicações*, interessados no desenvolvimento e implantação de aplicações para determinados domínios de interesse de usuários especialistas;
- *desenvolvedores de componentes*, interessados em desenvolver componentes que serão usados pelos provedores para construção de aplicações;
- *mantenedores de plataformas*, interessados em fornecer infraestruturas de computação paralela para execução das aplicações.

A atuação harmoniosa dos intervenientes na nuvem HPC Shelf se dá por meio de uma arquitetura formada de três elementos: Frontend, Core e Backend. O SAFe representa o Frontend da HPC Shelf. Trata-se de um *framework* por meio do qual o provedor de aplicações

[2] <http://www.vistrails.org/>
[3] <https://taverna.incubator.apache.org>
[4] <http://kepler-project.org/>

constroem e executam sistemas de computação paralela, a partir de componentes disponíveis através do serviço Core. Portanto, o Core representa o catálogo de componentes disponível no SAFe, possuindo serviços que permitem gerenciar todo o ciclo de vida dos componentes empregados em sistemas de computação paralela. Implementa também um sistema de escolha de componentes, baseado na noção de *contrato contextual*. Finalmente, o Backend é um conjunto de serviços associados a infraestruturas de computação paralela, sobre as quais as plataformas virtuais que compõem os sistemas de computação paralela são instanciados e executados.

Em nosso contexto, o SAFe é visto um sistema gerenciador de *workflows* científicos, proposto na Tese de Doutorado de Jefferson Silva (SILVA, 2016; CARVALHO-JUNIOR *et al.*, 2018). Neste trabalho, o suporte a proveniência é motivado e indicado como um dos trabalhos futuros. Tendo em vista às peculiaridades do SAFe, a fim de atender aos requisitos da HPC Shelf, entendia-se que abordar o problema da proveniência merecia uma abordagem mais específica dentro do contexto de um projeto de pesquisa próprio, com potencial para contribuições no projeto de sistemas de proveniência para outras plataformas que incorporassem certas características ainda particulares da HPC Shelf, especialmente no nível das abstrações de desenvolvimento de sistemas baseados em *workflows*. Enfim, o problema proposto para essa Dissertação de Mestrado é introduzir a proveniência na HPC Shelf, do ponto de vista tanto prospectivo quanto retrospectivo, levando em consideração a integração de conceitos de desenvolvimento baseado em componentes, sistemas de computação de alto desempenho e plataformas de serviços baseadas na abstração de nuvem computacional.

Os estudos de caso utilizados nesta Dissertação são baseados nos estudos de caso para avaliar o Gust (*Graphs Upon Shelf archiTecture*), uma extensão de um arcabouço de componentes desenvolvido sobre a HPC Shelf para processamento de larga escala baseado em MapReduce. O propósito do Gust é atender aos requisitos específicos de processamento de larga escala de grafos grandes (REZENDE C. A.; CARVALHO-JUNIOR, 2018). Os recursos de proveniência propostos nesta dissertação foram usados para a construção de um arcabouço de sistemas de computação paralela, agora vistos como componentes, para processamento Gust.

1.2 Objetivo

Diante do contexto e motivações mencionadas acima, seguem os objetivos, geral e específicos, desta Dissertação de Mestrado.

1.2.1 Objetivo Geral

O objetivo dessa dissertação é apresentar uma abordagem para a proveniência prospectiva e retrospectiva no contexto da HPC Shelf, a fim de suportar a reprodutibilidade da execução de *workflows*, levando em consideração a integração de conceitos de desenvolvimento baseado em componentes, sistemas de computação de alto desempenho e plataformas de serviços baseadas na abstração de nuvem computacional.

1.2.2 Objetivos Específicos

Além do objetivo geral da Tese, os seguintes objetivos específicos são alcançados:

- O tratamento, como componentes, de sistemas de computação paralela, atualmente ditos *efêmeros* por serem gerados sob demanda pela aplicação, de modo a promover o reuso desses sistemas em uma mesma aplicação ou até mesmo entre aplicações parceiras, com aplicação em proveniência prospectiva;
- Desenvolvimento de um mecanismo de rastreio de execução para orquestrações SAFeSWL, com aplicação em proveniência retrospectiva;
- Projeto e desenvolvimento de um arcabouço de sistemas de computação paralela para facilitar o desenvolvimento de aplicações de processamento de grafos grandes através do arcabouço de componentes Gust.

1.3 Estrutura do Documento

Esta Dissertação de Mestrado é composta por outros 5 capítulos, além deste capítulo introdutório. No Capítulo 2, é apresentado o contexto geral e estado-da-arte referente a mecanismos de proveniência em sistemas gerenciadores de *workflows* científicos, buscando contextualizar o trabalho. No Capítulo 3, é apresentada a HPC Shelf, a plataforma de serviços HPC sobre a qual as ideias propostas são implementadas. O Capítulo 4 apresenta as contribuições específicas advindas da implementação das propostas desta dissertação. Nesse capítulo, a Seção 4.1 trata de contribuições em *proveniência prospectiva*, em que é proposta a inclusão de uma nova espécie de componentes na HPC Shelf, para representar *workflows* científicos, os quais podem ser reusados por uma mesma aplicação ou aplicações distintas. Por sua vez, a Seção 4.2 trata das contribuições em *proveniência retrospectiva*, o qual envolve a inclusão de um mecanismo para lidar com a reprodução da execução de *workflows*. A Seção 4.3 inclui um exemplo utilizando

o *framework* MapReduce com suporte a proveniência prospectiva e retrospectiva. Um estudo de caso para fins de prova de conceito, baseado no arcabouço Gust, é apresentado e discutido no Capítulo 5. Finalmente, o Capítulo 6 apresenta as considerações finais deste documento de dissertação, onde, em particular, são anunciadas as conclusões dos estudos de caso. Além disso, são apresentadas as dificuldades e limitações para execução final da proposta, bem como discutidos possíveis trabalhos futuros.

2 PROVENIÊNCIA EM *WORKFLOWS* CIENTÍFICOS

O *Oxford English Dictionary* (OED) define proveniência como "o lugar de origem ou a história mais antiga conhecida de algo; o início da existência de alguma coisa; A origem de alguma coisa". Nas artes plásticas, a importância desta noção de proveniência muitas vezes pode ser medida através do seu custo. Por exemplo, uma obra do artista Picasso com sua procedência devidamente comprovada tem um grande valor de mercado.

Em experimentos científicos, a proveniência ajuda a entender e interpretar seus resultados. O *Global Change Information System* (GCIS) (GCIS, 2016), desenvolvido pelo *U.S. Global Change Research Program* (USGCRP) (USGCRP, 2016), reúne informações sobre vários órgãos federais dos EUA e analisa os impactos das mudanças climáticas. Um dos objetivos do GCIS é suportar a reprodutibilidade completa dos seus resultados. Por exemplo, o gráfico da mudança de temperatura ao longo dos anos poderia ser facilmente reproduzido. O rastreamento e disponibilização de meios para que os usuários avaliem a qualidade dos dados e recursos é fundamental para a confiabilidade e análise reprodutível de quaisquer dados científicos.

Um *workflow* científico é a descrição de um processo para a realização de um objetivo científico, normalmente expressa em termos de tarefas e suas dependências. Um *workflow* científico pode ser executado várias vezes no contexto de um determinado projeto, gerando uma grande quantidade de dados finais e intermediários de interesse para o usuário. Nesse contexto, a proveniência de dados permite ao cientista armazenar esses dados e metadados relevantes relacionados ao *workflow*. A captura desses dados é complexa, pois envolve fatores específicos de cada área científica que utiliza os sistemas de computação para a realização de pesquisas. Os dados de proveniência importantes para pesquisas de bioinformática, por exemplo, poderão não ser importantes para outras áreas científicas. Cada domínio de conhecimento possui esquemas, metadados ou modelos próprios, ou seja, a relevância dos dados de proveniência depende de cada área científica pesquisada. (MATTOSO *et al.*, 2008).

A proveniência de dados é de fundamental importância para a transparência de experimentos, avaliar a qualidade dos dados, e dar suporte à reprodutibilidade computacional. A proveniência de dados em *workflows* é dada como um conjunto de dependências entre objetos de dados considerando o histórico de mudanças e avanços ao longo de um *workflow*. Geralmente, fornece uma representação gráfica onde os nós denotam objetos de dados e as arestas são anotadas com a etapa que produziu os dados. As dependências entre objetos de dados, processados ou criados durante a execução, podem ser inferidas a partir da ordem lógica dos eventos. A Figura

1 mostra um exemplo de grafo que ilustra essa dependência entre dados. O objeto *A* deriva os objetos *B*, *C* e *D*, a partir das etapas S_1, S_2 e S_3, respectivamente. O objeto *E* é composto do agrupamento dos objetos *B*, *C* e *D*, processados pelas etapas S_4, S_5 e S_6, respectivamente. Finalmente, o objeto *F* é constituído do objeto *E* processado pela etapa S_7.

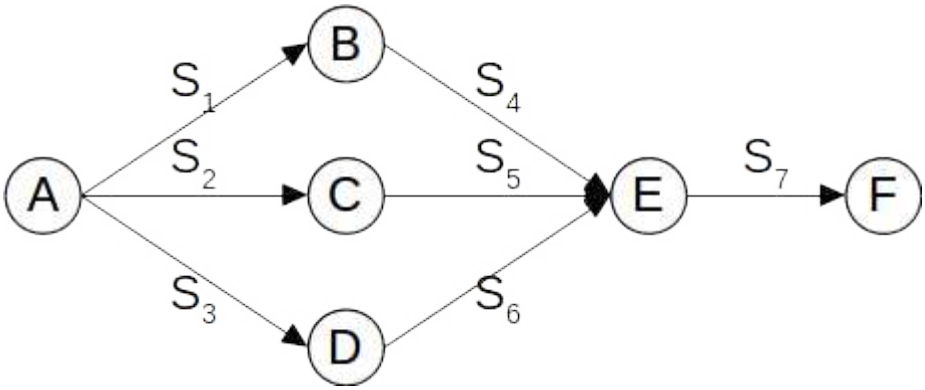

Figura 1 – Exemplo de grafo da dependência de dados.

A proveniência não é utilizada apenas para a interpretação de dados e fornecer resultados reproduzíveis, mas também para solução de problemas (depuração) e otimizar a eficiência durante o processo de refinamento de um experimento (*checkpoint*). Uma reexecução do *workflow* pode também ser otimizada, executando apenas as etapas ainda não realizadas e aproveitando os dados intermediários já disponíveis. Uma reexecução do *workflow* pode ocorrer por diversos motivos como: exceção ou falhas no processamento, problemas de comunicação em rede, limitações de armazenamento, disponibilidade de recursos de computação, etc.

Quando se trata de tarefas computacionais, Clifford *et al.* distinguem a proveniência em duas formas: prospectiva e retrospectiva (CLIFFORD *et al.*, 2008). Proveniência prospectiva inclui o registro das etapas de um *workflow* e sua dependência de dados, a serem seguidos para se alcançar um resultado. Proveniência retrospectiva inclui a captura de informações mais específicas sobre o ambiente de execução e recursos consumidos, bem como o comportamento efetivo do *workflow*, sem ter a necessidade de saber a sequência de atividades envolvidas. Os autores afirmam ainda ser uma boa prática combinar proveniência retrospectiva e prospectiva em uma representação única e uniforme.

Um SGWfC (Sistema de Gerenciamento de *Workflows* Científicos) auxilia os cientistas a projetar, executar e analisar *workflows* utilizando níveis de abstração que ocultam a complexidade do gerenciamento da computação envolvida nessas tarefas. SGWfCs vêm habilitando a captura, armazenamento, compartilhamento e consulta de informações de proveniência

tornando mais fácil a reprodutibilidade de seus resultados.

Módulos dentro de *workflows* científicos frequentemente operam em coleções de dados para produzir novas coleções de resultados. Quando realizada uma após a outra, essas operações podem produzir coleções de dados cada vez mais aninhadas, onde os diferentes módulos operam potencialmente em diferentes níveis de aninhamento. Assim, modelos de proveniência devem realizar o gerenciamento transparente de coleções de dados aninhadas permitindo maior independência e reusabilidade de *workflows* científicos sobre dados aninhados (MCPHILLIPS *et al.*, 2006).

Workflows científicos que tratam da exploração de dados e visualização são frequentemente de natureza exploratória e têm como consequência a investigação de espaços de parâmetros e técnicas alternativas. Um grande número de *workflows* relacionados são, por conseguinte, criados em uma sequência de refinamentos iterativos a partir da especificação inicial, como um cientista formula e testa hipóteses. A representação da evolução do *workflow* baseada no histórico de suas modificações é concisa e usa substancialmente menos espaço do que a alternativa de armazenar várias versões de um *workflow* (DAVIDSON *et al.*, 2007).

Um usuário pode ainda indicar quais módulos na especificação do *workflow* são relevantes para gravação das informações de proveniência. Para isso, os módulos compostos são usados como um mecanismo de abstração. Isso simplifica a visualização, bem como também melhora a compreensão das atividades envolvidas. A informação de proveniência é então vista por um usuário considerando o fluxo de dados entre os módulos conforme o seu entendimento (BITON *et al.*, 2008).

2.1 Ciclo de Vida do *Workflow*

O ciclo de vida de um *workflow* científico é uma descrição dos estados de transição desde a criação até a finalização do *workflow* científico. A gravação de metadados e informações de proveniência podem ser feitas durante as várias fases do ciclo de vida do *workflow*. A maioria das representações do fluxo de dados são muito simples por natureza. As construções de controle, tais como *loops*, geralmente não são incluídas. Um sistema de proveniência deve então ser concebido em termos dos modelos de computação que regem a execução de *workflows* científicos para garantir que todos os eventos relevantes sejam registrados.

O ciclo de vida do *workflow* científico ainda é um conceito mal compreendido e muitos sistemas gerenciadores de *workflow* ainda não abordam todas os aspectos da conexão entre

suas fases com mecanismos de coleta de proveniência (CRUZ *et al.*, 2009). Em (MATTOSO *et al.*, 2010), o ciclo de vida do *workflow* é dividido em 3 fases: composição, execução e análise. Além dessas 3 fases, (DEELMAN *et al.*, 2009) aborda a fase de mapeamento do *workflow*. Dessa forma, as fases do ciclo de vida de um *workflow* consideradas nesse trabalho são:

- Composição: cientistas registram aspectos do experimento, tais como definição da hipótese, recursos e dados. Em seguida, definem a concepção do fluxo do experimento (*workflow* abstrato), incluindo as entradas e saídas;
- Mapeamento: se refere à associação dos módulos do *workflow* aos recursos computacionais onde serão executados. Vale salientar que nesse momento ocorre o processo de instanciação do *workflow*, o que faz com que seu ambiente de execução seja preparado e ele se torne pronto para executar;
- Execução: o *workflow* é executado de acordo com as entradas e parâmetros estabelecidos, e novos dados intermediários e finais são criados. O monitoramento da execução é essencial para o cientista acompanhar o progresso e, se preciso, poder realizar alguma ação durante a execução. As execuções de um mesmo *workflow* podem gerar sequências de atividades diferentes, devido aos caminhos alternativos que determinados contextos podem demandar;
- Análise: cientistas podem consultar, visualizar e analisar resultados obtidos na execução do *workflow*. As informações de proveniência capturados pelo SGWfC nas fases anteriores podem também oferecer *insights* sobre o experimento.

2.2 Características de Proveniência

Uma taxonomia das características de um SGWfC é proposta em (CRUZ *et al.*, 2009) a partir da perspectiva do suporte à proveniência. Essa taxonomia pode ser útil na identificação de semelhanças e diferenças entre os SGWfCs que suportam a proveniência. Segue o conjunto de características que encapsula as funcionalidades expostas por vários SGWfCs: Captura de Proveniência, Assunto (*Subject*) de Proveniência, Acesso a Proveniência e Armazenamento da Proveniência.

A Captura de Proveniência aborda questões como mecanismos de captura usando estruturas internas ao sistema ou serviços externos; modos de rastreabilidade, que fornecem a proveniência dos produtos de dados enquanto ocorre a transformação dos dados ou apenas quando requerido; níveis de captura, podendo ser a nível do *workflow*, atividade (módulo) e sistema operacional; técnicas de captura, as quais podem recriar resultados através da inversão e

pela anotação de produtos de dados.

O Assunto de Proveniência trata das diferentes formas de categorizar os dados de proveniência. Uma maneira seria corresponder os dados a uma fase do ciclo de vida do *workflow*. Por exemplo, os dados capturados na fase de composição podem estar associados à proveniência prospectiva. Em outra forma, a granularidade da proveniência dos produtos de dados que podem variar de fina (*fine-grained*) para grossa (*coarse-grained*) granularidade de informações. Finalmente, a descrição das etapas do *workflow* junto com seus dados e parâmetros podem ser representados por grafos na visão da dependência de processos ou dependência de dados.

O Acesso à Proveniência descreve como usuários realizam a consulta e análise dos repositórios de dados de proveniência. O acesso geralmente ocorre através da navegação pelo grafo de derivação dos produtos de dados, linguagens de consulta, APIs de serviço fornecidas pelo SGWfC e consulta visual (*query-by-example*), por exemplo, usando a mesma interface para construir *workflow*, como oferecido pelo Vistrails (Seção 2.4.1). As linguagens de consulta para dados de proveniência constituem uma importante área de pesquisa, visto que atuam sobre dados orientados a grafos.

O Armazenamento de Proveniência lida com o registro das informações de proveniência levando em consideração a escalabilidade do armazenamento (centralizado ou distribuído); estratégia de acoplamento entre os dados de proveniência e os produtos de dados utilizados no *workflow*; estratégia de arquivamento da proveniência, armazenando todas as versões com seus respectivos *timestamps* ou apenas uma versão de referência e as diferenças entre as versões; e a expressividade dos modelos conceituais usados pelo sistema para persistir os dados de proveniência.

O trabalho (RAGAN *et al.*, 2016) organiza os diferentes tipos de informação de proveniência em 5 categorias:

- *Data*: histórico de alterações dos dados pode incluir subconjuntos, fusão de dados, formatação, transformações, ou a execução de uma simulação para consumir ou gerar novos dados;
- *Visualization*: preocupado com o histórico de visualizações gráficas e estados de visualização;
- *Interaction*: foca no histórico de ações do usuário e os comandos com o sistema;
- *Insight*: inclui o histórico de hipóteses, percepções e outras formas de resultados analíticos, cognitivos, devido à exploração de dados e inferências;

- *Rationale*: para uma compreensão mais profunda de *insights* e interações, é necessário compreender o histórico das intenções do usuário e raciocínio por trás deles.

Diferentes projetos enfatizam propósitos diferentes para o uso da informação de proveniência. O trabalho de (RAGAN *et al.*, 2016) também organiza os propósitos para proveniência em:

- *Recall*: Manter ou recuperar a memória e consciência dos estados atuais e anteriores da análise;
- *Replication*: Reproduzir as etapas ou *workflow* de uma análise prévia;
- *Action recovery*: Manter o histórico de ações permitindo desfazer/refazer operações e ações durante a análise;
- *Collaborative communication*: Comunicação e compartilhamento de dados, informações e ideias com outras pessoas que estão realizando a mesma análise;
- *Presentation*: Envolve a comunicação com aqueles que não estão envolvidos diretamente com a análise em si. Exemplos: comunicação ao público em geral ou para níveis superiores de gestão, relatórios de auditoria, ou de ensino;
- *Meta-analyses*: Rever a própria análise de processos, a fim de compreender e melhorar aspectos da análise (tais como a eficiência do processo, a eficiência do treinamento, ou estratégias analíticas).

Muitos pesquisadores têm enfatizado o valor de registrar o histórico de interações do usuário com o sistema, o qual deve ser projetado para uma captura sistemática (por exemplo, as interações do usuário, *screenshots* de visualização) e anotações explícitas de usuários. O sistema precisa salvar tanto uma imagem do estado do sistema como as configurações necessárias. Esse tipo de registro pode conter informações sobre tomadas de decisão e possuir diferentes níveis de granulosidade.

A proveniência dos dados muitas vezes envolve alguns desafios como: controle de versão e *forking*, atualizações de *datasets*, o movimento e a duplicação de dados, e os níveis associados de incerteza. Cada versão dos dados deveria ser mantida, bem como os relacionamentos com suas respectivas versões, podendo sofrer modificações quanto ao conteúdo ou estrutura.

2.3 Modelos de Dados de Proveniência

Atualmente, existem 2 principais padrões principais destinados a gerenciar registros de proveniência, o *Open Provenance Model* (OPM) (MOREAU *et al.*, 2011) e o modelo PROV (PROV-OVERVIEW, 2016). Antes de abordá-los, é preciso contudo distinguir entre proveniência de processo (focado no *workflow* e ambientes de execução) e proveniência de dados (focado na criação e transformação dos dados) (SIMMHAN *et al.*, 2006). Dados participam de várias transformações no seu ciclo de vida desde a sua geração até a sua supressão.

2.3.1 OPM

O OPM representa os dados de proveniência usando um grafo causal que registra a trajetória de vida de um processo ou artefato. O OPM foi concebido como um modelo de dados de proveniência aberto com contribuições de diversos pesquisadores e iniciou através de uma série de encontros conhecidos como *Provenance Challenge* (CHALLENGER, 2017) durante o *International Provenance and Annotation Workshop* (IPAW). Diante disso, o OPM foi adotado em diversas abordagens de proveniência em *workflows* científicos como Taverna, Vistrails, Swift, eBioFlow, Karma, Kepler e outros, além de ter uma *toolkit* Java para manipulação de grafos OPM.

O OPM é baseado em três entidades principais, como descrito na Figura 2: Artefatos (*Círculo - A*), entidade imutável de estado que representa um objeto real ou digital; Processos (*Retângulo - P*), ação ou conjunto de ações realizadas ou causadas por artefatos resultando em novos artefatos; e Agentes (*Octógono - Ag*), entidade contextual que age como catalisador de um processo, possibilitando, facilitando, controlando ou afetando sua execução. O OPM procura demonstrar a relação causal entre eventos que afetam objetos e descreve essa relação através de um grafo acíclico direcionado. Existem cinco tipos de dependências causais entre as entidades definidas pelo modelo OPM: *Used* e *WasGeneratedBy* relacionam Artefato e Processo, *WasControlledBy* relaciona Processo e Agente, *WasTriggeredBy* relaciona Processos entre si e *WasDerivedFrom* relaciona Artefatos entre si.

O OPM não fornece nenhum mecanismo específico para fazer atribuição de um grafo de proveniência ou parte dele. Isto é geralmente feito pelo mecanismo de anotações.

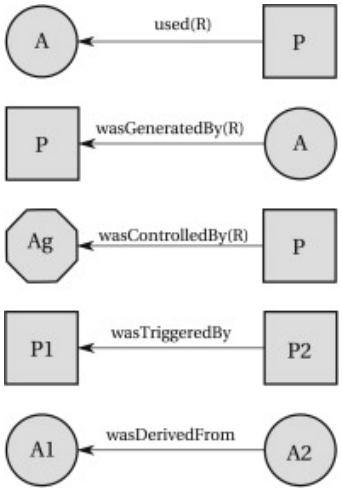

Figura 2 – Relações entre as entidades do modelo OPM (MOREAU *et al.*, 2011).

2.3.2 PROV

O modelo PROV consiste de 12 documentos que contêm sua especificação e definem vários aspectos necessários para a interoperabilidade de informações de proveniência em ambientes heterogêneos. A Figura 3 ilustra a organização dos documentos que são:

- PROV-OVERVIEW, uma visão geral da família de documentos PROV;
- PROV-PRIMER, uma introdução intuitiva para o modelo de dados PROV;
- PROV-O, a ontologia PROV, uma ontologia OWL2 que permite o mapeamento do modelo de dados PROV para RDF;
- PROV-DM, o modelo de dados do PROV para proveniência;
- PROV-N, uma notação de proveniência destinada ao consumo humano;
- PROV-CONSTRAINTS, um conjunto de restrições aplicáveis ao modelo de dados PROV;
- PROV-XML, um esquema XML para o modelo de dados PROV;
- PROV-AQ, mecanismos de acesso e consulta de proveniência;
- PROV-DICTIONARY introduz um tipo específico de coleção, consistindo em pares de entidades-chave;
- PROV-DC fornece um mapeamento entre PROV-O e *Dublin Core Terms*[1];
- PROV-SEM, uma especificação declarativa em termos de lógica de primeira ordem do modelo de dados PROV;
- PROV-LINKS introduz um mecanismo para ligar pacotes.

O *PROV Data Model* (PROV-DM) é um dos principais documentos que especifica o

[1] <http://dublincore.org/documents/2012/06/14/dcmi-terms/>
[2] PROV-Overview: <http://www.w3.org/TR/prov-overview/>

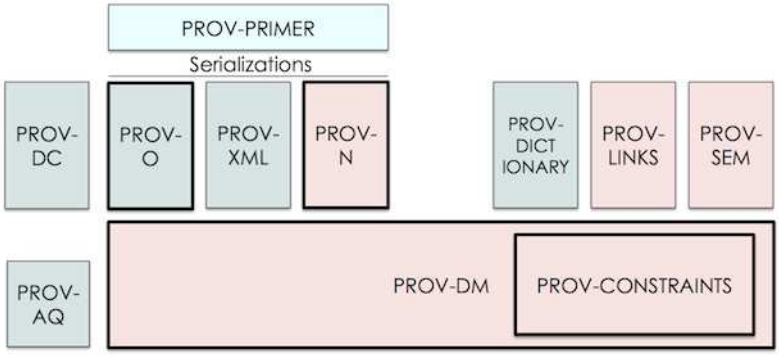

Figura 3 – Organização do modelo PROV[2].

modelo de dados de proveniência e possibilita a interoperabilidade entre sistemas. O PROV-DM distigue as estruturas centrais e estruturas estendidas. As estruturas centrais formam a essência das informações de proveniência comumente utilizadas. As estruturas estendidas aprimoram e refinam as estruturas centrais com maior expressividade para atender usos mais avançados de proveniência.

O PROV-DM é organizado em 6 componentes dentre seus elementos e relações: (1) Entidades e Atividades - Entidades representam objetos e Atividades representam os processos que atuam sobre as Entidades; (2) Derivações de Entidades - descreve a transformação de uma Entidade em outra e a derivação de subtipos; (3) Agentes, Responsabilidade, e Influência - um Agente possui certa Responsabilidade por uma Atividade e define uma noção geral de influência; (4) Pacote - especifica um conjunto de descrições de proveniência que possui um mecanismo para suportar proveniência da proveniência; (5) Alternativa - consiste de relações entre entidades que referenciam a mesma coisa; (6) Coleção - Entidade que compreende a noção de coleções e possui vários membros.

O modelo PROV-DM possui uma quantidade maior de relacionamentos, o que permite expressar a proveniência de forma mais precisa. As Entidades podem ser geradas por Atividades (*WasGeneratedBy*), atribuídas a Agentes (*WasAttributedTo*) ou derivar de outras Entidades (*WasDerivedFrom*. As Atividades podem usar as Entidades (*Used*), estar associadas a Agentes (*WasAssociatedWith*) e ainda ser dependentes de outras Atividades (*WasInformedBy*). Um Agente pode ter responsabilidade ou autoridade sobre outro Agente (*ActedOnBehalfOf*). A Figura 4 ilustra essas relações básicas entre os elementos do PROV-DM.

[3] PROV-DM: <http://www.w3.org/TR/prov-dm/>

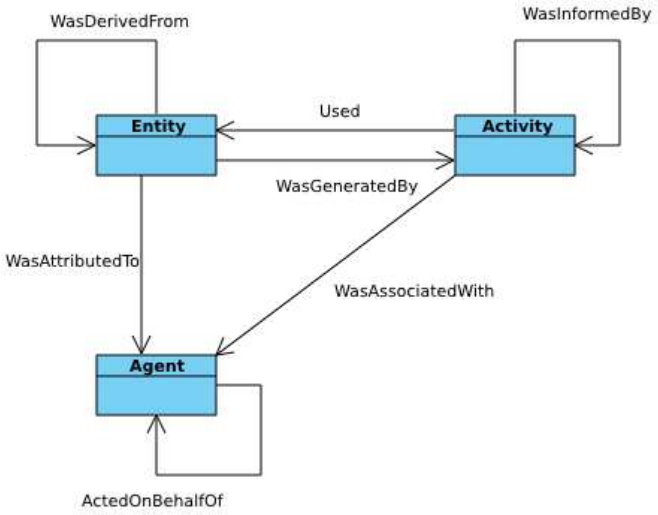

Figura 4 – Relações básicas do modelo PROV[3].

2.4 Exemplos de SGWfCs

Os SGWfCs possibilitam que processos científicos sejam mapeados para *workflows* com maior eficiência e precisão. Esses sistemas auxiliam os pesquisadores na modelagem dos *workflows* e orquestramento da sua execução, controlando a utilização de recursos. Um SGWfC permite também a gerência dos dados de um *workflow* científico envolvendo a captura e armazenamento dos dados de proveniência obtidos ao longo do ciclo de vida dos *workflows*. A seguir temos a descrição de alguns SGWfCs e seus mecanismos de proveniência.

2.4.1 Vistrails

Vistrails é um SGWfC desenvolvido na linguagem Python e oferece suporte para a exploração e visualização de dados a partir de sua interface gráfica (VISTRAILS, 2016). Um *workflow* é formado por módulos que podem ser tanto executáveis como tipos de dados. Os módulos se comunicam através de portas de entrada e saída, conectando-se a porta de saída de um módulo com a porta de entrada de um outro módulo.

Uma importante característica do Vistrails é sua infraestrutura de proveniência, que mantêm informações históricas detalhadas sobre as etapas seguidas no decorrer de uma tarefa exploratória, além de rastrear as informações sobre a execução de *workflows* (FREIRE *et al.*, 2006). O Vistrails também fornece uma estrutura flexível de anotação em que você pode especificar as informações de proveniência específica da aplicação (FREIRE *et al.*, 2008).

A informação de proveniência é armazenada de maneira estruturada nesse sistema.

A proveniência prospectiva é representada internamente como objetos Python que podem ser serializados em arquivos XML no sistema de arquivos. A proveniência retrospectiva utiliza banco de dados relacional ou arquivos XML. Dessa forma, a consulta de informações sobre as especificações ou instâncias de *workflows* no Vistrails pode ser realizada por linguagens de consulta em arquivos XML como, por exemplo, XPath e XQuery.

O sistema Vistrails pode ser configurado para centralizar informações em um banco de dados usado como um repositório compartilhado, oferecendo suporte para a exploração colaborativa entre múltiplos usuários. O sistema fornece uma interface de consulta baseada em texto e também visual, chamada de *Query-by-Example* (QBE), especificamente projetada para *workflows*, onde é possível explorar e reusar informações de proveniência de forma intuitiva e flexível.

O Vistrails possui um mecanismo de varredura de parâmetros que permite especificar conjunto de valores para parâmetros diferentes em um *workflow*, o qual gera múltiplos produtos de dados. Os resultados da exploração do parâmetro pode ser exibido lado a lado em uma planilha de visualização para facilitar a comparação.

A informação de proveniência do VisTrails é organizada em três níveis: a evolução do *workflow*, o *workflow* e a sua execução (SCHEIDEGGER *et al.*, 2008). No primeiro, são armazenadas todas as modificações que o usuário realiza no *workflow* (evolução) em uma árvore de versões. O segundo nível consiste na especificação das tarefas e suas conexões, o que constitui o *workflow*. Por fim, o nível de execução contém informações sobre a execução do *workflow* (por exemplo, quem executou um módulo, em qual máquina, o tempo decorrido, etc.).

2.4.2 Taverna

O SGWfC Taverna é resultado do projeto MyGrid[4] para fornecer suporte a experimentos no domínio da bioinformática, embora seja utilizado em diversas áreas de pesquisa como astronomia, pesquisa médica, música, entre outras (OINN *et al.*, 2007). Atualmente, o projeto Taverna está em transição para a incubadora da Apache[5]. Um *workflow* no Taverna é especificado como um grafo direcionado onde os nós são chamados de processadores, representando componentes de software, como um serviço, que pode ser local ou remoto via *webservices*. Um nó processador consome dados pela porta de entrada e produz dados na porta de saída. A ligação

[4] <http://www.mygrid.org.uk>

[5] The Apache Software Foundation (<http://apache.org/)>

entre dois nós representa a conexão entre a porta de saída de um nó com a porta de entrada do outro nó. Taverna possui vários tipos de serviços disponíveis para integração como: *Web services* WSDL e RESTful, BioMart, BioMoby, SoapLab, R, Beanshell, Excel e planilhas CSV.

O Taverna 3 possui uma linguagem para descrição de *workflows* chamada SCUFL2 (*Simple Conceptual Unified Flow Language*), que substituiu o antigo formato *t2flow* do Taverna 2. O SCUFL2 define um modelo de *workflow* usando ontologias (formato *.wfbundle*) e uma API Java para trabalhar com estruturas de *workflow*, permitindo inspecionar, modificar, criar e executar *workflows*, independente de plataformas. Informações sobre o *workflow*, como autoria e descrição, podem ser adicionadas usando anotações. É possível ainda buscar e importar *workflows* existentes a partir do projeto *MyExperiment* (ROURE *et al.*, 2009), um ambiente colaborativo onde cientistas publicam seus *workflows*.

A definição do *workflow* é armazenada em SCUFL2 no formato RDF/XML, permitindo que ferramentas que atuem em RDF possam ser vinculadas com as definições de *workflow*, como a consulta usando SPARQL[6]. A proveniência pode ainda ser salva em diversos formatos, incluindo o modelo padrão PROV-O[7] recomendado pela W3C[8]. Taverna não captura a proveniência da evolução do *workflow*, pois assume que o cientista gerencia a evolução do *workflow* através de algum sistema de controle de versão ou pelo projeto *MyExperiment*. No entanto, um identificador interno do *workflow* (UUID) é atualizado a cada mudança do *workflow* e todos esses identificadores anteriores são incluídos dentro do formato da definição do *workflow*, permitindo a detecção de *workflows* com ancestralidade comum (TAVERNA, 2017).

Taverna captura a proveniência de execução de *workflows* e mantêm esses dados em uma base de dados interna, utilizados na perspectiva *Results* do Taverna onde é possível visualizar lista de execuções anteriores, progresso da execução e resultados intermediários e finais. O registro da proveniência é desabilitado por padrão no Taverna, pois o tempo consumido para isso pode ser relevante para a execução do *workflow*. Quando habilitado, o Taverna mantém as informações sobre os valores de entrada e saída para cada serviço do *workflow*.

2.4.3 Kepler

O SGWfC Kepler permite a análise e modelagem de dados científicos, simplificando os esforços para criar modelos executáveis usando representações visuais desses processos

[6] <https://www.w3.org/TR/sparql11-overview/>
[7] <http://www.w3.org/TR/prov-o/>
[8] World Wide Web Consortium

(LUDÄSCHER *et al.*, 2006). Sua ideia principal consiste em usuários com pouco conhecimento em computação poderem criar *workflows* com componentes padrões ou modificar *workflows* existentes para atender às suas necessidades. Kepler incorpora tecnologias de computação distribuída permitindo cientistas compartilhar dados e *workflows* com outros cientistas.

Um *workflow* no Kepler consiste de componentes, tais como diretores, atores e parâmetros, assim como relações e portas que facilitam a comunicação entre os componentes (KEPLER, 2015c). Todo *workflow* deve ter um componente diretor que orquestra a sua execução usando um modelo de computação. Por exemplo, a sincronização das tarefas onde um componente é executado por vez é obtida pelo diretor SDF (*Sequence Data Flow*). Por outro lado, uma execução paralela onde componentes executam simultaneamente, pode ser obtida pelo diretor PN (*Process Network*). Cada etapa do *workflow* é representada por um ator, um componente de processamento. Um ator especifica o que processa enquanto o diretor especifica quando ocorre. Cada ator contém uma ou mais portas para comunicação com outros atores. As portas são categorizadas em 3 tipos: porta de entrada, para consumo de dados pelo ator; porta de saída, para produção de dados pelo ator; e porta de entrada e saída, a qual suporta ambos consumo e produção de dados pelo ator. Cada porta pode ser configurada como única ou múltipla, dependendo se pode ser conectada somente a um único canal ou a múltiplos canais. Existem ainda as portas externas, utilizadas para comunicação entre *subworkflow* e o *workflow* que o contém, e as portas *port-parameters*, que permitem a configuração de um valor padrão para um parâmetro pelo usuário.

Os parâmetros são valores configuráveis em tempo de execução que podem ser associados ao *workflow*, diretor ou ator. Assim, um valor inicial poderia ser configurado como parâmetro para um ator ou diretor. O número de iterações de um *workflow* pode ser tratado como um parâmetro. A *relação* é um outro elemento do Kepler que permite ramificar *workflows* para enviar dados para outros pontos do *workflow* ao mesmo tempo.

workflows aninhados ou *subworkflows* podem ser representados no Kepler por um ator composto, onde um conjunto de atores desempenham juntos uma tarefa com operações complexas. Um ator composto pode ser incluído como um componente dentro de um *workflow* e ainda possuir um diretor diferente. Os *workflows* científicos no Kepler oferecem acesso a diversos recursos distribuídos tais como repositórios de dados e serviços computacionais. A complexidade de tais tarefas de processamento é ocultada dos cientistas, que podem focar no seu interesse científico.

A proveniência no Kepler é habilitada através da instalação do módulo *Provenance Module*, o qual torna possível capturar, analisar e consultar o histórico da execução de *workflows* (KEPLER, 2015a). Por padrão, as informações de proveniência são registradas em um banco de dados HSQL mas pode mudar o banco para MySQL, Oracle ou PostgreSQL alterando o arquivo de configuração com dados para acesso. Esse módulo registra informações tanto da evolução da definição do *workflow* quanto da execução do *workflow*. O módulo de proveniência inclui uma API Java para acesso às informações de proveniência, além da possibilidade de consulta no próprio banco de dados usando SQL. Um programa chamado *Provenance Manager* exporta as execuções de *workflows* a partir do banco de dados para uma representação em JSON do modelo padrão PROV.

O *Workflow Run Manager* (KEPLER, 2015b) é um módulo do Kepler que fornece uma GUI capaz de gerenciar o histórico de execução de *workflows* armazenados localmente pelo módulo de proveniência (*Provenance Module*). As configurações de proveniência, como os dados de acesso ao banco de dados, podem ser distintas para cada *workflow* adicionando um ator *Provenance Recorder*. As anotações no *workflow* são adicionadas através do ator *Annotation* incluído na biblioteca padrão do Kepler.

3 HPC SHELF

A HPC Shelf é uma nuvem computacional que se propõe a oferecer um conjunto de serviços voltados para a composição, implantação e execução de *sistemas de computação paralela* de interesse de *aplicações* (CARVALHO-JUNIOR *et al.*, 2018). Através de aplicações da HPC Shelf, os usuários finais, especialistas de um dado domínio de conhecimento, podem especificar problemas que desejam resolver e obter soluções computacionais na forma de sistemas de computação paralela. Portanto, as aplicações podem ser enxergadas como as portas de entradas de usuários especialistas para os serviços da HPC Shelf.

Os sistemas de computação paralela da HPC Shelf são sistemas orientados a componentes. Componentes representam algoritmos, repositórios de dados, padrões de comunicação e sincronização, plataformas de computação paralela (e.g. *clusters*) e requisitos não-funcionais. Para isso, a HPC Shelf utiliza a noção de componentes inerentemente paralelos do modelo Hash (CARVALHO-JUNIOR; LINS, 2008), o qual apresenta uma forma expressiva para tratar requisitos de paralelismo no desenvolvimento de aplicações baseadas em componentes.

3.1 Sistema de Computação Paralela

Na plataforma HPC Shelf, um *sistema de computação paralela* é composto por um conjunto de instâncias de componentes envolvidos na realização de uma tarefa computacionalmente intensiva que demanda o emprego de uma ou mais plataformas de computação paralela, onde serão realizadas a computações que representam os interesses tratados por esses componentes. Tais plataformas de computação paralela, também são tratadas como componentes. Dessa forma, uma sistema de computação paralela representa tanto elementos de *software* quanto elementos de *hardware*, sendo composto na verdade por diversos componentes de sistema. Um componente de sistema é a composição de um componente de *software* com uma plataforma de execução em que ele executará (*hardware*). Essa abordagem de *software* e *hardware* integrada em um mesmo sistema permite que um software paralelo, por exemplo, possa explorar o máximo do potencial da plataforma de computação paralela, utilizando algoritmos que fazem suposições sobre características de sua arquitetura, tais como hierarquias de memória, conjuntos de instruções SIMD, presença de aceleradores computacionais, e assim por diante.

Em um sistema de computação paralela, além dos componentes ditos de solução, representando algoritmos e estruturas de dados, temos dois componentes especiais: o compo-

nente Application e o componente Workflow. O componente Application é responsável pela intermediação entre componentes de solução e a aplicação propriamente dita, através de portas de serviço, para fins, por exemplo, de monitoração e troca de dados de entrada e saída. Por sua vez, o componente Workflow atua como um regente, que guia o fluxo de execução dos componentes (orquestração) através sincronização de suas ações computacionais para a obtenção da solução do problema. Para isso, introduz-se a abstração de portas de ações, para ativar ações computacionais exportadas por componentes de solução.

3.2 Atores da HPC Shelf

Os interessados na utilização da HPC Shelf, tanto para prover quanto para consumir serviços, são chamados de atores e agrupados em quatro categorias:

- **Especialistas** são atores interessados em fazer uso das aplicações através de uma linguagem de alto nível de abstração para resolver problemas específicos do seu domínio, sendo considerados os usuários finais dos serviços de uma nuvem HPC Shelf. Em geral, não possuem conhecimento de técnicas de computação, tampouco sobre a arquitetura de plataformas de computação paralela.
- **Provedores de aplicações** são atores interessados no desenvolvimento e implantação de aplicações capazes de sintetizar sistemas de computação paralela para resolver problemas de um determinado domínio. Um provedor deve ser capaz realizar a composição de componentes registrados para a construção dos sistemas de computação paralela e definir todas as funcionalidades e e abstrações necessárias ao usuário especialista.
- **Desenvolvedores de componentes** são atores interessados em desenvolver componentes que serão usados pelos provedores na construção de sistemas de computação paralela. Um desenvolvedor possui amplo conhecimento em técnicas de computação paralela e habilidade de compor componentes a partir de outros já existentes, além de empreender as melhores técnicas e ferramentas dependendo do contexto arquitetural do sistema de computação (por exemplo, memória compartilhada ou distribuída e uso de aceleradores computacionais).
- **Mantenedores de plataformas** são interessados em fornecer e gerenciar infraestruturas de computação paralela para execução das aplicações, para as quais os componentes podem ser otimizados segundo suas características arquiteturais. Componentes da espécie plataforma são registrados no catálogo de componentes da HPC Shelf pelo mantenedor e

encapsulam todas as informações necessárias para a instanciação das plataformas virtuais.

Diante dos diferentes modelos de computação em nuvem, uma nuvem HPC Shelf pode ser vista segundo o modelo de serviços SaaS (*Software-as-a-Service*) na perspectiva do usuário especialista. Por sua vez, para o provedor de aplicações e desenvolvedor de componentes, pode ser vista segundo o modelo PaaS (*Platform-as-a-Service*). Finalmente, para mantenedores de plataformas, pode ser vista como uma nuvem IaaS (*Infrastructure-as-a-Service*).

3.3 Arquitetura da HPC Shelf

A atuação harmoniosa dos atores em uma nuvem HPC Shelf se dá por meio de uma arquitetura formada de três elementos: Frontend, Core e Backend. O Frontend corresponde ao SAFe (*Shelf Application Framework*), o *framework* utilizado para a construção de sistemas de computação paralela da HPC Shelf. O provedor de aplicações acessa serviços do SAFe e deriva sistemas de computação paralela para resolver problemas especificados pelo especialista através da interface de alto nível da aplicação. O SAFe pode ser visto como um sistema gerenciador de *workflows* (SWfMS) para orquestração de componentes paralelos (CARVALHO-JUNIOR *et al.*, 2018). A especificação de como os componentes são orquestrados e como eles interagem na aplicação também é feita por meio do SAFe, através de uma linguagem específica chamada de SAFeSWL. A linguagem SAFeSWL serve tanto para especificar *workflows* científicos quanto para descrever a arquitetura de um sistema de computação paralela.

O Core representa o catálogo de componentes disponíveis para usuários do SAFe, responsável por gerenciar todo o ciclo de vida dos componentes. Desenvolvedores de componentes e mantenedores de plataformas registram seus componentes e respectivos contratos através dos serviços do Core. Dessa forma, podemos enxergar o Core como uma nuvem CaaS (*Component-as-a-Service*). O Core implementa um sistema de contratos contextuais, no qual uma aplicação acessa seus serviços para descobrir e selecionar componentes de acordo uma abstração de contexto de execução, que leva em conta os requisitos da aplicação em relação ao componente e as características da plataforma alvo sobre o qual irá executar (CARVALHO-JUNIOR *et al.*, 2016). Depois de instanciados, os componentes são orquestrados diretamente pelas aplicações, sem intermediação do Core.

Finalmente, o conjunto de infraestruturas de computação paralela, onde são executadas as aplicações, é representado pelo Backend. O Backend é capaz de gerenciar e monitorar o uso dos recursos físicos na instanciação de plataformas virtuais de computação paralela. É

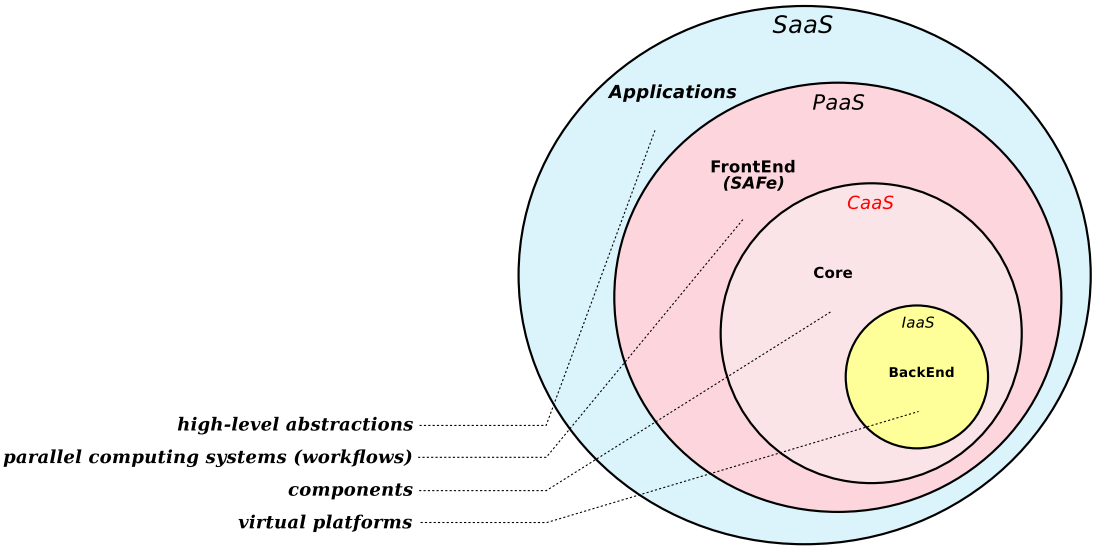

Figura 5 – A Arquitetura de Nuvem da HPC Shelf.

importante notar que as plataformas virtuais também são registradas no Core pelo mantenedor de plataformas para composição do sistema de computação paralela. O Core utiliza os serviços do Backend para a implantação de plataformas virtuais. Depois de implantadas, as plataformas virtuais podem se comunicar diretamente com o Core para instanciar componentes. Após instanciados, os componentes ficam prontos para comunicação direta com aplicações por meio de ligações de serviço e ação.

As implementações de serviços SAFe, Core e Backend interagem através de interfaces fracamente acopladas baseadas em *Web Services*. Assim, cada um poderia ser implementado em uma linguagem de programação e ambiente de execução distintos. A Figura 5 ilustra a arquitetura de nuvem multicamada da HPC Shelf.

A HPC Shelf não se preocupa com o tipo de interface de alto nível que cada aplicação fornece aos seus especialistas. Essa interface pode ser tão rudimentar quanto um simples formulário em que o especialista insere parâmetros de entrada, clica em um botão para executar uma solução e recebe um resultado textual; ou tão sofisticado quanto uma DSL ou ambiente de programação visual, usando o padrão de projeto MVC (*Model* / *View* / *Controller*), destinado a descrever problemas compondo blocos de construção e monitorando interativamente o progresso de soluções, possivelmente usando recursos avançados de visualização. Tal separação de interesses é uma consequência de ter o SAFe como o Frontend da HPC Shelf, com a responsabilidade de ocultar as abstrações da HPC Shelf atrás das aplicações.

3.4 Espécies de Componentes

Como dito anteriormente, a HPC Shelf é compatível com o modelo Hash de componentes paralelos, o qual classifica os componentes em diferentes espécies, de forma que componentes de uma mesma espécie possuem um comportamento similar em todas as etapas do seu ciclo de vida. Dessa maneira, a HPC Shelf deve suportar um conjunto de espécies de componentes que representam diferentes elementos de composição para a construção de sistemas de computação paralela. As espécies de componentes suportadas pela HPC Shelf são:

- Plataformas Virtuais;
- Computações;
- Fontes de Dados;
- Qualificadores;
- Conectores;
- Ligações (*Bindings*).

Plataformas virtuais representam plataformas de computação paralela de memória distribuída, tais como *clusters*, onde são executadas as aplicações.

Computações representam algoritmos de computação paralela capazes de explorar as características de uma classe de plataformas virtuais. Vale salientar que, em um sistema de computação paralela, todo componente de computação é associado a uma plataforma virtual, onde será implantado e instanciado para execução.

Fontes de dados oferecem o acesso a infraestruturas de onde podem ser obtidos dados de interesse de componentes de computação, bem como podem ser armazenados grandes massas de dados por eles produzidos, inclusive intermediários.

Qualificadores são utilizados em contratos contextuais para representar interesses não-funcionais de componentes, aqueles que não correspondem a elementos concretos de software, tais como, por exemplo, características da plataforma virtual sobre o qual um componente de computação vai executar ou os requisitos/restrições de uma aplicação em relação ao componente que deseja utilizar.

Conectores são mediadores de sincronização e comunicação entre computações e fontes de dados, os quais podem residir em plataformas virtuais distintas. Podem atuar no papel de regente na orquestração de componentes de computação ou de agente intermediador da coreografia entre componentes de computação e fontes de dados. Para isso, conectores são compostos de um conjunto de facetas, cada uma das quais associada à plataforma virtual onde um

componente ligado ao conector encontra-se alocado, visando ligação direta através de *bindings*.

Finalmente, *ligações* (*bindings*) podem ser enxergados como casos especiais de conectores, com papéis específicos. Há três tipos de ligações:

- *ligações de serviços*;
- *ligações de alocação*;
- *ligações de ações*.

Ligações de serviços conectam pares de portas oferecidas por componentes de computação, fontes de dados e conectores. Portas podem ser de um dentre dois papéis: *usuárias* ou *provedoras*. De fato, uma ligação de serviço liga uma porta usuária de um componente a uma porta provedora de outro, o qual oferece um serviço que deve ser consumido pelo componente que possui a porta usuária. Entretanto, o serviço consumido através da porta usuária pode ser diferente do serviço provido através da porta provedora, sendo responsabilidade da implementação do *binding* tratar das questões de adaptação.

Uma ligação de serviços pode ser *direta* ou *indireta*. Uma ligação direta acontece quando as portas ligadas pertencem a componentes, ou facetas de conectores, que residem em uma mesma plataforma virtual. Caso contrário, trata-se de uma ligação indireta. No projeto de um sistema de computação paralela, é preferível priorizar as ligações diretas, para efeito de desempenho, deixando os conectores a cargo da comunicação entre componentes conectados em plataformas virtuais distintas.

Ligações de alocação conectam um componente computação a uma plataforma virtual onde será implantado para execução. Atualmente, são implementadas como ligações de serviços, aonde a porta de alocação da plataforma virtual é *provedora* e a porta de alocação do componente de computação é *usuária*.

Ligações de ações são usadas para orquestração de componentes computação e conectores. Uma ligação de ações liga um conjunto de portas de ações de componentes distintos, as quais devem possuir o mesmo conjunto de *nomes de ações*. Normalmente, cada nome de ação é um identificador para uma ação computacional oferecida por um componente de computação ou conector, o qual reage a sua *ativação*. Uma ação se completa em uma ligação de ações quando todos os componentes associados à ligação de ação através de uma porta possuem uma ativação pendente para a ação, referenciada através do seu nome, como ilustrado na Figura 6.

O principal uso de ligações de ações diz respeito ao componente Workflow em sistemas de computação paralela. Através dessas ligações, o componente Workflow realiza a

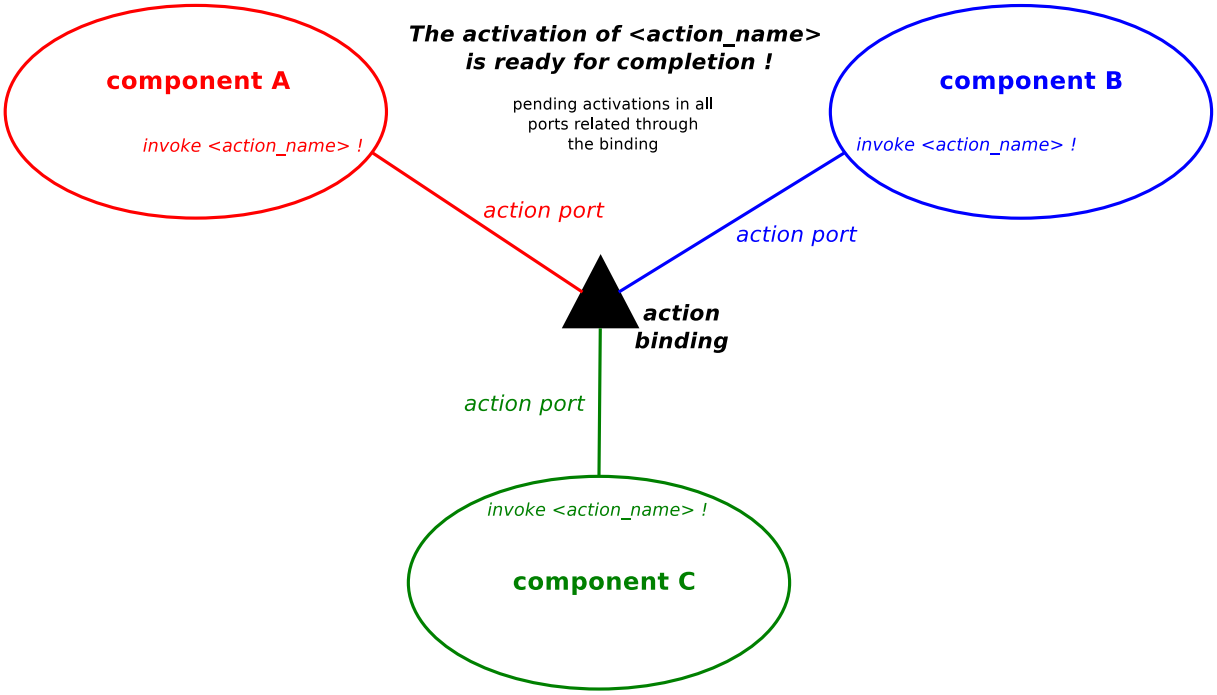

Figura 6 – Semântica de Ativação em *Bindings* de Ações

orquestração computacional dos componentes de solução, notadamente computações e conectores, guiando a execução do sistema. Além disso, através desse tipo de ligação, o componente Workflow controla o ciclo de vida dos componentes do sistema de computação paralela. Cada componente possui uma porta de ações pré-definida, chamada de *LifeCycle*, com as ações **resolve**, **deploy**, **instantiate**, **run** e **release**, as quais são conectadas individualmente a uma porta correspondente do componente Workflow, de forma que o ciclo de vida do componente possa ser controlado. O significado das ações de ciclo de vida é apresentado a seguir:

- **resolve**, para seleção de uma implementação do componente que satisfaça os requisitos descritos no seu contrato contextual (Seção 3.6).
- **deploy**, para a implantação do componente selecionado sobre a plataforma de computação paralela para ele escolhida pelo procedimento de resolução. Esse processo envolve as etapas de compilação e configuração de bibliotecas. Caso o componente seja da espécie plataforma, uma plataforma virtual é criada na infraestrutura do mantenedor com as características especificadas no seu contrato contextual.
- **instantiate**, para instanciar o componente na plataforma alvo, tornando-o apto para a execução de suas ações computacionais.
- **run**, para começar a execução do procedimento que implementa a lógica de orquestração de ações computacionais do componente.

- release, para liberação do componente a partir da plataforma em que está instalado. Caso seja um componente plataforma, os recursos da máquina virtual são virtualmente desalocados, de acordo com a política de uso dos recursos da infraestrutura do mantenedor.

3.5 SAFeSWL (SAFe *Scientific Workflow Language*)

SAFeSWL é uma linguagem de especificação de *workflows* científicos proposta pela HPC Shelf, que se destina a descrever tanto a arquitetura de um sistema de computação paralela quanto a orquestração dos componentes que deles fazem parte, o *workflow* propriamente dito (CARVALHO-JUNIOR *et al.*, 2018). Para isso, é composta por dois subconjuntos ortogonais. O subconjunto arquitetural especifica quais os componentes que serão instanciados e como serão ligados entre si através de portas de serviços e de ações. Por sua vez, o subconjunto de orquestração fornece construtores para a especificação do fluxo de ativação das ações das portas do componente Workflow. Ambos, o código arquitetural e o código de orquestração, são definidos através de um formato baseado em XML, implantados por aplicativos da HPC Shelf através do SAFe. Baseado na proposta de tratar sistemas de computação paralela em componentes, os códigos arquitetural e de orquestração também podem ser recuperados do Core.

O subconjunto arquitetural fornece aos provedores de aplicação a capacidade de especificar os componentes e ligações (*bindings*) que constituem a arquitetura de um sistema de computação paralela. Cada componente de solução possui um contrato contextual agregado, o qual guia a seleção da implementação apropriada do componente de acordo com os requisitos definidos pelo contrato.

Os construtores sintáticos do subconjunto arquitetural de SAFeSWL são delimitados pelo seguintes elementos:

- `<application>`: representa o componente Application, declarando suas portas usuárias e provedoras;
- `<workflow>`: representa o componente Workflow, declarando suas portas de ações;
- `<body>`: representa as declarações dos componentes de solução, onde cada espécie de componente possui seu elemento próprio `<computation>`, `<connector>`, `<repository>` ou `<platform>`;
- `<service_binding>`: representa uma ligação de serviço que conecta uma porta usuária a uma porta provedora;
- `<action_binding>`: representa uma ligação de ação que conecta portas de ações pos-

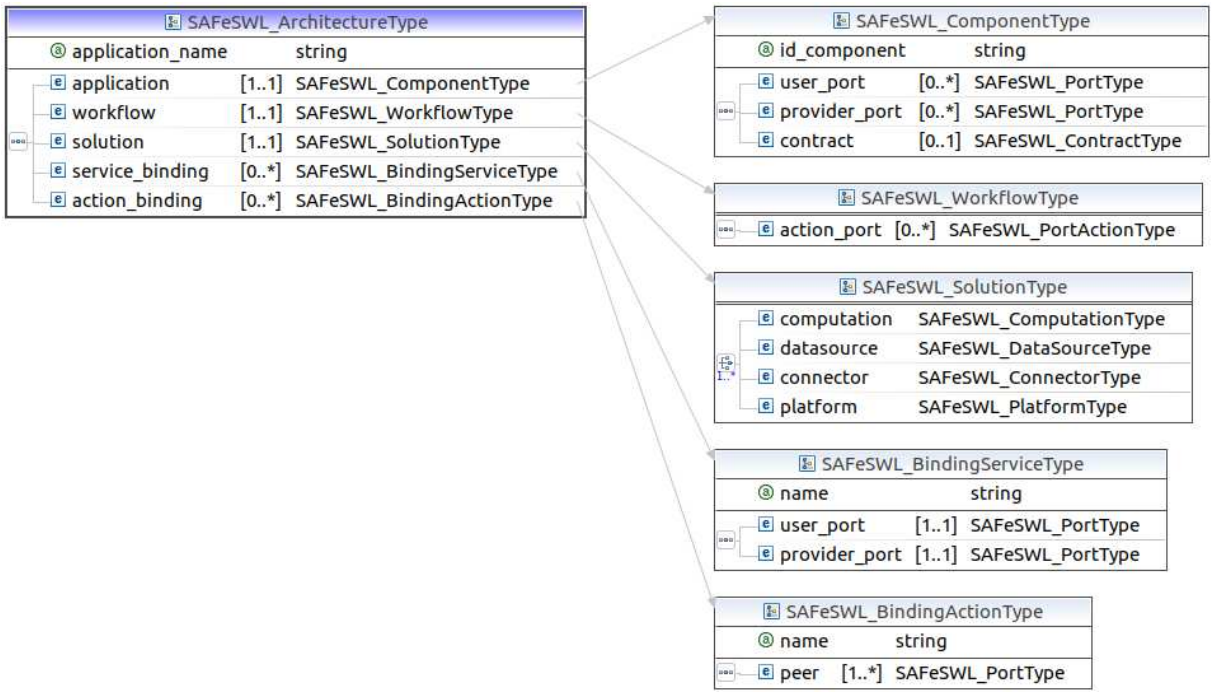

Figura 7 – A Gramática XSD para a Linguagem Arquitetural.

suindo o mesmo conjunto de nomes de ações.

A estrutura arquitetural de um sistema de computação paralela é formado a partir dos elementos apresentados acima. A descrição de uma estrutura arquitetural possui um único elemento `<application>`, um único elemento `<workflow>`, um ou mais componentes de solução no `<body>`, zero ou mais `<service_binding>` e zero ou mais `<action_binding>`. A gramática XSD para especificar a sintaxe concreta do subconjunto arquitetural de SAFeSWL é representada na Figura 7.

O provedor de aplicação gera código de *workflows* usando o subconjunto de orquestração do SAFeSWL. O código de orquestração é interpretado pelo componente Workflow, que é conectado às portas de ações de computações e conectores por meio de ligações de ações, para orientar o progresso da computação do sistema de computação paralela.

A lógica de ativação de ações é controlada pelo componente Workflow utilizando operadores primitivos e diversos combinadores, os quais permitem especificar sequenciamento, alternativas, iterações, concorrência e chamadas assíncronas de ações. A gramática XSD para especificar a sintaxe concreta do subconjunto de orquestração de SAFeSWL é representada na Figura 8.

Uma tarefa é definida pela orquestração de um conjunto de ações. Existem sete tipos primitivos de tarefas e quatro combinadores de tarefas. As tarefas primitivas são:

- **skip** denota uma ação vazia;
- **break** termina a iteração na qual se encontra aninhada;
- **continue** volta ao início da iteração;
- **start** denota a ativação assíncrona de uma ação, opcionalmente tratada através de um identificador (*handle_id*);
- **wait** expressa a espera da conclusão de uma ação anteriormente ativada de forma assíncrona, possivelmente bloqueando a *thread* do *workflow* em que foi executada;
- **cancel** representa o cancelamento da ativação de uma ação previamente ativada de forma assíncrona;
- **invoke** denota a ativação síncrona de uma ação, equivalente a uma invocação assíncrona (**start**) seguida diretamente por uma espera (**wait**).

Os combinadores de tarefas são:

- **sequence** denota a execução sequencial de uma lista de tarefas, na ordem em que são declaradas;
- **parallel** expressa a execução concorrente de um conjunto de tarefas, em que a tarefa combinadora termina depois de todas as tarefas internas terem sido concluídas (paradigma *fork-join*);
- **choice** indica a execução de uma dentre um conjunto de tarefas. A tarefa escolhida é a primeira na sequência cuja ação de guarda **select** está ativada no componente;
- **iterate** representa a execução iterativa de uma tarefa. As cláusulas **loop** e **until** controlam a terminação da iteração, a qual também pode ser encerrada quando um **break** é executado dentro do seu escopo. O combinador também pode ser usado para guiar uma invocação iterativa em um conjunto de ações, usando a cláusula **branch**.

3.6 Contratos Contextuais

A HPC Shelf atualmente emprega um sistema de contratos contextuais (CARVALHO-JUNIOR *et al.*, 2016) herdado do HPE (*Hash Programming Environment*) (CARVALHO-JUNIOR; REZENDE, 2013), que separa interface e implementação de componentes (componentes abstratos e concretos, respectivamente), de modo que um ou mais componentes concretos possam coexistir no catálogo do Core para um determinado componente abstrato. Assim, diferentes componentes concretos atenderão a diferentes suposições sobre os requisitos da aplicação alvo e os recursos das plataformas de computação paralela onde eles podem ser instanciados

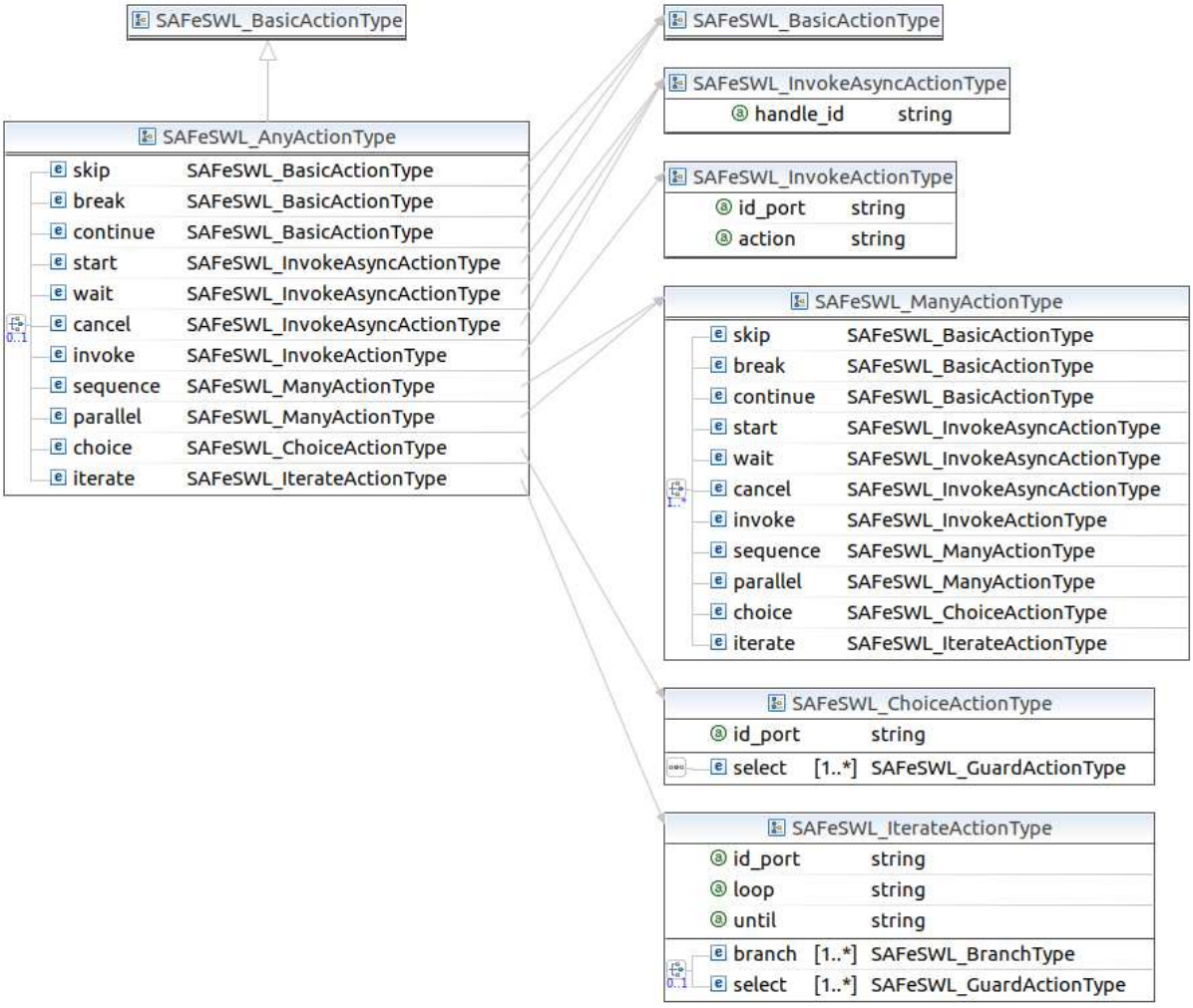

Figura 8 – A Gramática XSD para a Linguagem de Orquestração.

(*contexto de execução*). Para isso, um componente abstrato possui uma *assinatura contextual*, composta por um conjunto de *parâmetros de contexto*. Por sua vez, um componente concreto deve ser associado a um tipo, ou seja, um *contrato contextual*, definido por um componente abstrato e um conjunto de *argumentos de contexto* que valoram seus parâmetros de contexto. Um componente abstrato representa um conjunto de componentes que implementam o mesmo interesse para diferentes contextos de execução. Portanto, apresentam a mesma interface.

Durante a orquestração, quando a ação **resolve** de um componente (abstrato) é ativada, o procedimento de resolução é acionado para escolher um componente concreto que melhor corresponda ao contexto (contrato contextual) fornecido. A resolução dos componentes ocorre dinamicamente durante o processo de execução do *workflow*. Exemplos de assinaturas e contratos contextuais são apresentados na próxima seção, dentro do contexto de uma aplicação capaz de gerar sistemas de computação paralela para processamento Map/Reduce.

As aplicações científicas normalmente se deparam com cenários onde podem neces-

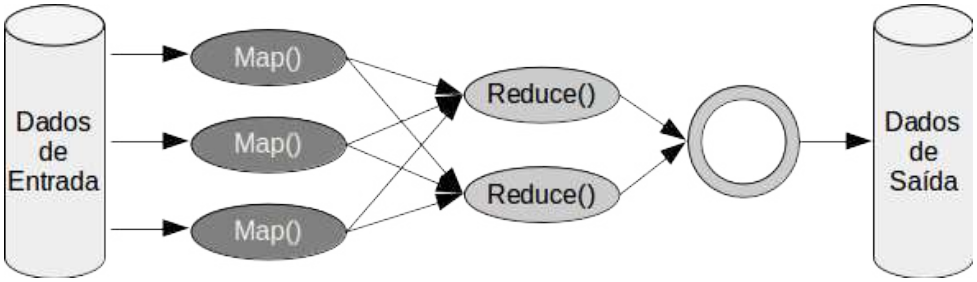

Figura 9 – Processamento básico do Map/Reduce.

sitar de bastante tempo para executar, devido a diversos fatores como o tempo de computação em si, a troca de dados entre componentes que estão distribuídos, a fila de espera na submissão de plataformas de computação paralela, entre outros. Assim, a dinamicidade no processo de execução, obtida através do sistema de contratos contextuais, é essencial para execuções de longa duração pois proporciona uma flexibilidade para a aplicação possibilitando se adequar a mudanças de contextos ou requisitos, que possam ocorrer durante a execução.

3.7 Aplicação de Referência: Processamento Map/Reduce

A fim de oferecer exemplos de sistemas de computação paralela da HPC Shelf, será utilizado nessa dissertação um arcabouço para processamento Map/Reduce recentemente proposto. O intuito principal desse arcabouço é permitir a utilização de um sistema paralelo em larga escala, envolvendo vários *clusters* possivelmente dispersos geograficamente, por usuários que não necessariamente possuam experiência em programação paralela. Dessa forma, os detalhes de particionamento de dados, comunicação entre nós e escalonamento da execução são realizados pela plataforma que implementa tal modelo de programação, deixando os usuários isentos desses detalhes particulares da programação de sistemas paralelos.

O Map/Reduce teve sua arquitetura originalmente proposta pela empresa Google Inc. (DEAN; GHEMAWAT, 2008), a partir de conceitos e abstrações provenientes da programação funcional. Segundo o conceito original, um processamento Map/Reduce em particular é definido por duas funções: *mapeamento* (*map*) e *redução* (*reduce*). A função *map* processa a entrada em um conjunto de pares *chave/valor* e a função *reduce* opera sobre os dados intermediários associados a uma mesma chave produzidos na fase de mapeamento anterior. A Figura 9 ilustra o processamento básico do Map/Reduce.

Devido a aplicação do modelo a um grande número de problemas, surgiram várias implementações introduzindo extensões à ideia inicial do modelo. Além da implementação de

referência feita pela Google em C++ e RPC, temos o Hadoop[1], projeto da *Apache Foundation* reconhecido por diversos projetos de pesquisa e possui uma grande base de usuários. Hadoop é implementado em Java mas as funções de mapeamento e redução podem utilizar outras linguagens de programação. Outra implementação do Map/Reduce, o MR-MPI, possui funcionalidades básicas feitas em C++ e MPI e voltada especificamente para computações científicas.

3.7.1 Problema de Contagem de Palavras

O exemplo de contador de palavras repetidas é bastante utilizado para ilustrar o funcionamento do modelo Map/Reduce. O algoritmo consiste em contar as ocorrências de cada palavra em um texto. Por exemplo, considerando um texto que contenha apenas as palavras verde, branco e azul, o resultado final do algoritmo será a quantidade de palavras existentes para cada cor. Esse exemplo é ilustrado na Figura 10, utilizando as funções básicas *map* e *reduce*, além das fases intermediárias *splitting* e *shuffling* de cada função, respectivamente.

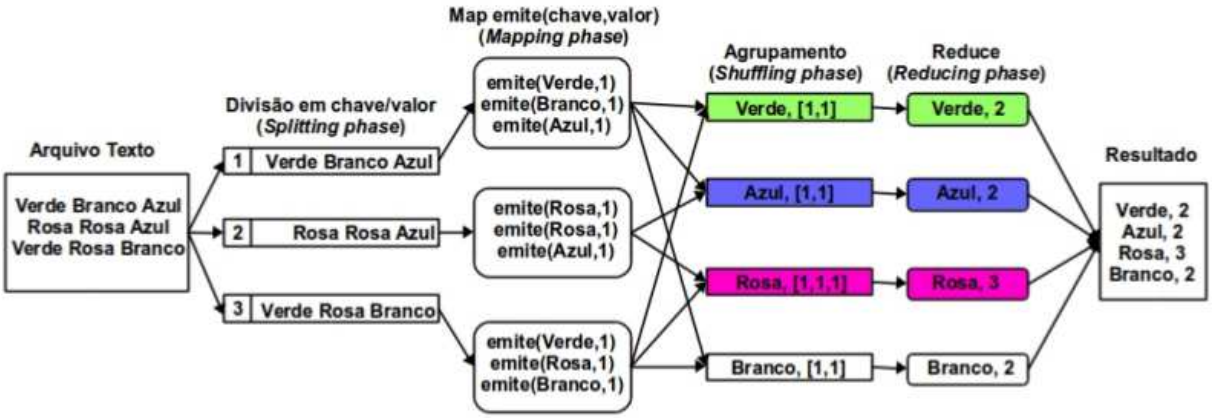

Figura 10 – Exemplo clássico de contagem de palavras com Map/Reduce (DANTAS, 2017).

O algoritmo inicia na fase de *splitting*, agrupando uma parte do texto de entrada (no caso, uma linha) em tuplas de *chave/valor* onde a chave representa o número da linha e o valor é a própria frase daquela linha. Então, esse conjunto de tuplas é distribuído, dependendo da chave, para um conjunto de agentes de mapeamento paralelos, os quais realizam a fase de mapeamento para cada par *chave/valor*. A função de *map* lê cada palavra da linha que recebeu e emite um par *cor/1* rotulando cada ocorrência com o valor 1.

Finalizada a fase de mapeamento, os pares referentes à mesma cor devem ser agrupados para posteriormente aplicar a função de redução. Essa fase intermediária é chamada de

[1] <http://hadoop.apache.org>

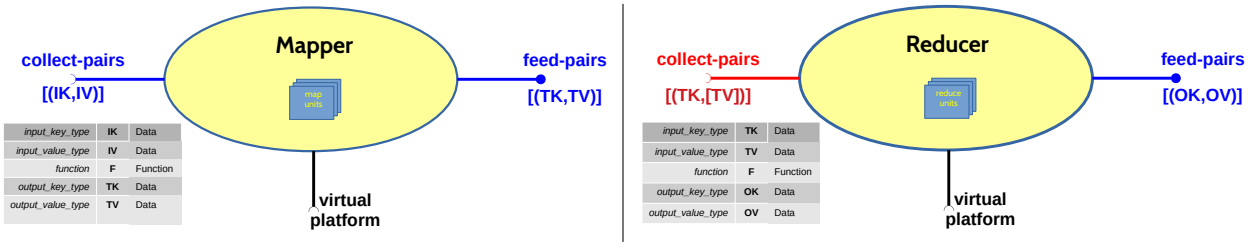

Figura 11 – Blocos de Construção MapReduce: Computações MAPPER and REDUCER

shuffling e realiza a distribuição dos pares *cor/[1..1]*, conforme suas chaves, para os agentes de redução paralelos. A função *reduce* recebe os pares agrupados e realiza a computação para obter a soma dos valores $1's$ da chave correspondente, que consiste na quantidade de ocorrências da palavra. O resultado final é composto pelos pares calculados por cada agente de redução. A Figura 10 ilustra o exemplo do Map/Reduce detalhado acima com os respectivos resultados intermediários de cada fase.

3.7.2 *Um Arcabouço Map/Reduce para a HPC Shelf*

Na Tese de Doutorado de Jefferson Silva (SILVA, 2016) é apresentada uma primeira versão de um arcabouço Map/Reduce para construção de sistemas de computação paralela voltados a processamento Map/Reduce sobre a HPC Shelf, aprimorada na Tese de Doutorado de Cenez Rezende (REZENDE, 2017). Nesse arcabouço, um sistema de computação paralela é concebido por uma composição de componentes de quatro tipos, ilustrados nas figuras 11 e 12: MAPPER, REDUCER, SPLITTER e SHUFFLER. Deve-se notar que todos esses componentes possuem duas portas de serviço: uma porta usuária chamada `collect_pairs`, através da qual recebe uma *stream* de pares chave/valor de entrada, e uma porta provedora chamada `feed_pairs`, através da qual entrega uma *stream* de pares *chave/valor* de saída, como ilustrado na Figura 13. Na notação utilizada nas figuras 11 e 12, a fim de especificar os tipos das portas de serviços, $\langle K,V \rangle$ denota um par chave-valor com chave do tipo K e valor do tipo V, enquanto $[T]$ denota uma lista de elementos do tipo T. Finalmente, são também listados os parâmetros de contexto

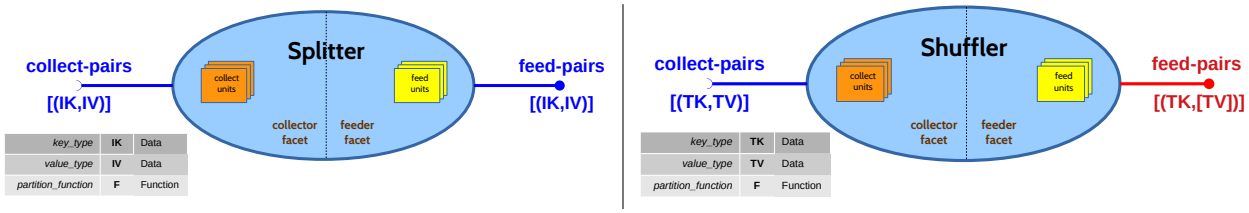

Figura 12 – Blocos de Construção MapReduce: Conectores SPLITTER and SHUFFLER

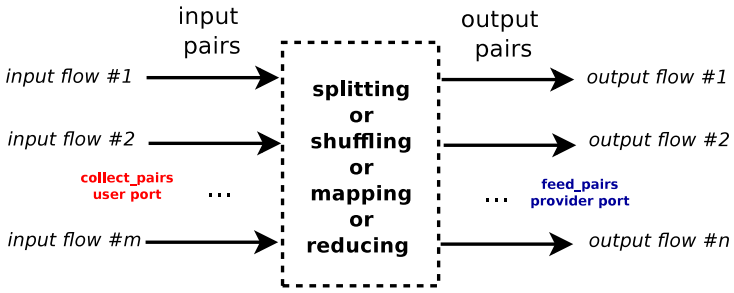

Figura 13 – Fluxos de Entrada e Saída de Componentes Map/Reduce

de cada componente, notadamente os tipos de chaves e valores, bem como funções para a sua distribuição e processamento.

MAPPER é um componente de computação que representa agentes de mapeamento capazes de executar uma função de mapeamento sobre a *stream* de pares chave/valor de entrada. Assim, para cada par, são produzidos pares *chave/valor* emitidos na *stream* de saída. Por sua vez, REDUCER é também um componente de computação, que representa um agente de redução. Para isso, recebe através da *stream* de entrada um conjunto de pares *chave/multi-valor* e aplica uma função de redução, a qual produz, para cada par, um par *chave/valor* que deve ser emitido pela *stream* de saída. Como ilustrado na Figura 11, ambos, MAPPER e REDUCER, são compostos de uma única unidade paralela, instanciada sobre os núcleos da plataforma virtual onde encontram-se alocados.

SPLITTER e SHUFFLER são conectores, que podem ser usados para ligar agentes de mapeamento e de redução, bem como fontes de dados de entrada e saída, através de *streams* de pares *chave/valor*. Para isso, possuem múltiplas facetas dos tipos **collector** e **feeder**, respectivamente cada uma com uma porta `collect_pairs` ou `feed_pairs`, as quais podem ser ligados a agentes de mapeamento e redução, bem como fontes de dados. Enquanto um conector SPLITTER apenas distribui pares recebidos através de uma ou mais *streams* de entrada em torno de uma ou mais *streams* de saída, usando uma função de particionamento aplicado a cada par

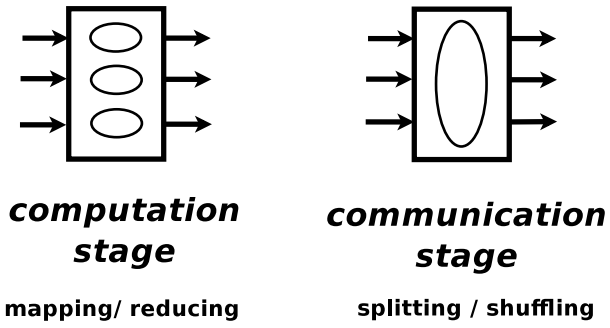

Figura 14 – Estágios MapReduce - Computação e Comunicação

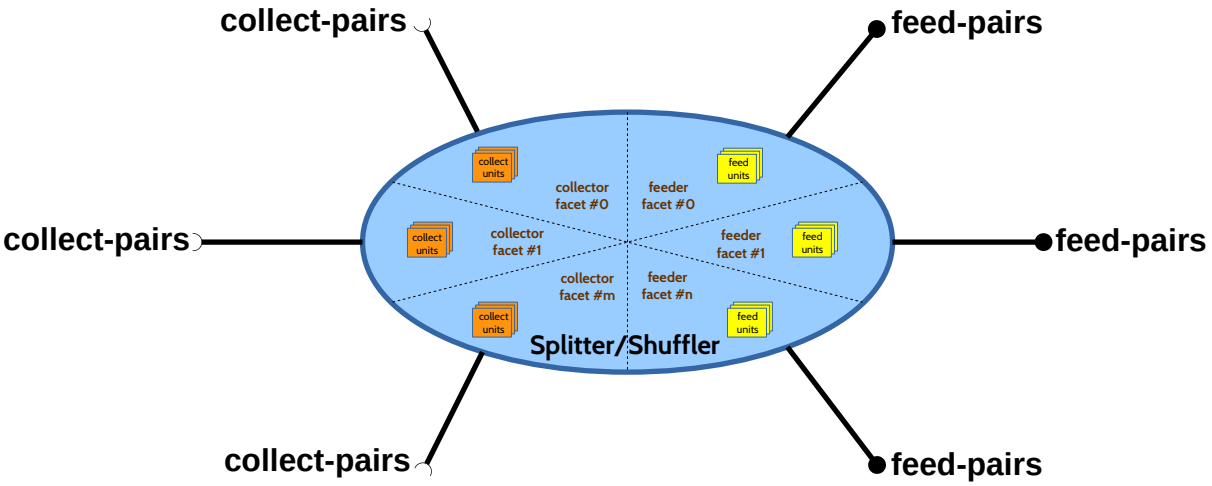

Figura 15 – Multiplicidade das Facetas de SPLITTER e SHUFFLER.

chave/valor, um conector SHUFFLER agrupa pares que tem a mesma chave, emitindo-as em uma mesma *stream* de saída escolhida de acordo com a chave. Como ilustrado na Figura 12, ambos, SPLITTER e SHUFFLER, possuem um par de unidades paralelas, cada uma associada a uma das facetas e, portanto, instanciadas sobre cada plataforma virtual onde encontram-se os agentes de mapeamento e redução acoplados por seu intermédio. Por sua vez, a Figura 15 ilustra o caráter múltiplo das facetas **collector** e **feeder** nos conectores MapReduce, de modo a que múltiplos agentes de mapeamento, redução e fontes de dados podem ser acoplados por seu intermédio.

Os *bindings* de serviço que ligam conectores SPLITTER e SHUFFLER a fontes de dados podem fazer o papel de adaptação entre a estrutura de dados mantida na fonte de dados e pares *chave/valor*. Podem portanto serem dependentes da aplicação.

Um sistema de computação paralela Map/Reduce pode ligar agentes de mapeamento e redução, utilizando conectores SPLITTER e SHUFFLER como intermediários, de acordo com a necessidade de um processamento em particular, representando rodadas de mapeamento e redução, envolvendo um ou mais agentes de mapeamento/redução em cada rodada. Conforme ilustrado na Figura 14, rodadas de mapeamento/redução, cada uma envolvendo múltiplos agentes, correspondem a estágios de computação de um algoritmo baseado em Map/Reduce, enquanto rodadas de intermediação através de instâncias singulares de conectores SPLITTER e SHUFFLER podem ser vistos como estágios de comunicação desses algoritmos.

A Figura 16 ilustra um sistema de computação paralela Map/Reduce comum, usado para contagem de palavras, formado por uma rodada de mapeamento seguida de uma rodada de redução. Entre essas rodadas de computação, é possível ver rodadas de comunicação. Um conector SPLITTER distribui pares de entradas (linhas dos arquivos) lidos de uma fonte de dados

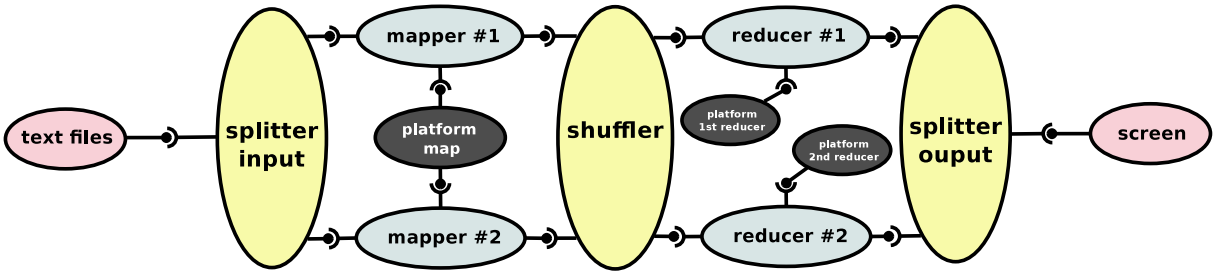

Figura 16 – Esquema Arquitetural para um Sistema Map/Reduce de Contagem de Palavras

de entrada (***text files***) entre os agentes de mapeamento. Por sua vez, um conector SHUFFLER agrupa os pares resultantes dos mapeamentos (contagem de palavras em cada linha) e entrega os pares resultantes aos agentes de redução, que realizam a contabilização final dos resultados. Finalmente, um segundo conector SPLITTER recebe os resultados dos agentes de redução e os deixa disponível para leitura por parte da fonte de dados de saída (componente ***screen***). Deve-se notar que os agentes de mapeamento estão alocados na mesma plataforma virtual (componente **platform_map**), enquanto cada agente de redução está implantada em uma plataforma virtual distinta (componentes **platform 1st reducer** e **platform 2nd reducer**).

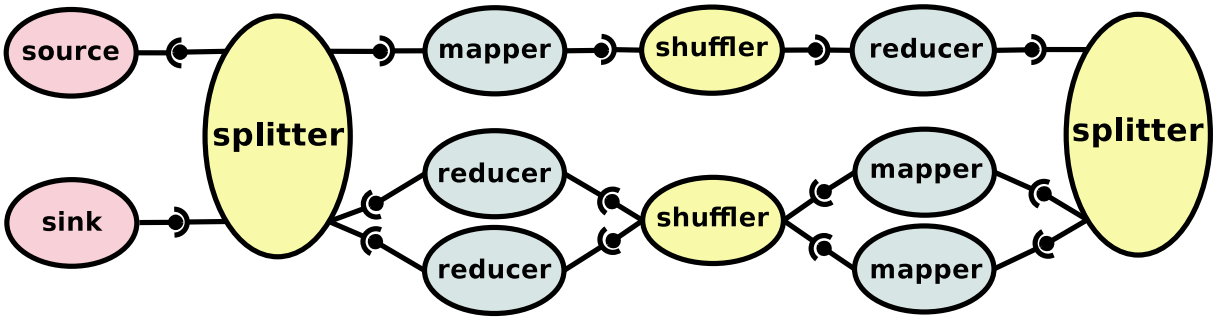

Figura 17 – Esquema Arquitetural de um Sistema MapReduce Iterativo

Na Figura 17, ilustramos um sistema de computação paralela Map/Reduce menos convencional, iterativo e envolvendo múltiplas rodadas de mapeamento e redução.

Os componentes Map/Reduce fazem também uso ainda da abstração de contratos contextuais, descrita na seção 3.6. Os componentes utilizam a assinatura contextual para que o sistema escolha a implementação mais apropriada dependendo, por exemplo, dos tipos dos pares chave/valor de entrada, intermediários e de saída, bem como das funções de mapeamento e redução. Parâmetros não-funcionais relacionados às características da plataforma, bem como parâmetros de contexto relacionados a qualidade e custo também podem fazer parte dos aspectos a serem avaliados. Esse mecanismo de resolução dos componentes é realizado pelo sistema de contrato contextual disponível na plataforma HPC Shelf.

name	bound	description
input_key_type	DATA	the type of keys in input pairs
input_value_type	DATA	the type of values in input pairs
function	FUNCTION	the custom function (map or reduce)
output_key_type	DATA	the type of keys in output pairs
output_key_value	DATA	the type of values in output pairs

Tabela 1 – Assinatura Contextual de MRCOMPUTATION (MAPPER e REDUCER)

name	bound	description
key_type	DATA	The type of keys in pairs
value_type	DATA	The type of values in pairs
partition_function	PARTITION	The custom function for distributing keys across mappers or reducers

Tabela 2 – Assinatura Contextual de MRCONNECTOR (SPLITTER e SHUFFLER)

As tabelas 1 e 2 apresentam detalhes das assinaturas contextuais de componentes MRCOMPUTATION e MRCONNECTOR, respectivamente. A partir desses componentes abstratos são derivados os componentes abstratos de computação (MAPPER e REDUCER) e de comunicação (SPLITTER e SHUFFLER) do sistema de computação paralela Map/Reduce.

Parameter Name	Component	Contextual Bound
input_key_type	MAPPER REDUCER	INTEGER STRING
input_value_type	MAPPER REDUCER	STRING INTEGER
function	MAPPER REDUCER	COUNTWORDS[...] SUMVALUES[...]
output_key_type	MAPPER REDUCER	STRING
output_value_type	MAPPER REDUCER	INTEGER

Tabela 3 – Contratos de agentes MAPPER e REDUCER na aplicação de contagem de palavras

Por sua vez, a Tabela 3 apresenta os argumentos de contexto aplicados aos parâmetros de contexto na formação dos contratos contextuais dos agentes de mapeamento e redução do sistema de computação paralela de contagem de palavras.

3.8 Gust: Um Arcabouço de Processamento de Grafos Baseado em MapReduce

Como principal produto da Tese de Doutorado de Cenez Rezende (REZENDE, 2017), foi apresentado o Gust (*Graphs Upon Shelf archiTecture*), uma extensão do *framework* MapReduce, também aprimorado nesse mesmo trabalho, para atender aos requisitos específicos

Tabela 4 – Assinatura Contextual do Componente Gusty

Parameter Name	Var.	Contextual Bound
intermediary_key_type	TK	DATA
intermediary_value_type	TV	DATA
reduce_function	Rf	GUSTYFUNCTION[...]
output_key_type	OK	DATA
output_value_type	OV	DATA
graph_type	G	DATA
input_bin_type	IB	INPUTBIN

de processamento de larga escala de grafos grandes (REZENDE C. A.; CARVALHO-JUNIOR, 2018). Como subproduto desta dissertação, visando um estudo de caso que será apresentado no Capítulo 5, são introduzidos recursos de proveniência ao Gust. Por esse motivo, apresentamos adiante alguns detalhes mais relevantes sobre o Gust.

Gust acrescenta ao arcabouço MapReduce os componentes abstratos GUSTY, derivado de REDUCER, e GUSTYFUNCTION, derivado de REDUCEFUNCTION. A assinatura contextual do componente GUSTY encontra-se apresentada na Tabela 5, onde nota-se a presença de dois parâmetros de contexto adicionais: ***graph_type*** e ***input_bin_type***.

GUSTYFUNCTION permite um modelo de programação onde o programador pode usar tanto o paradigma *centrado-em-vértices* quanto o paradigma *centrado-em-grafos*. Para isso, possui um componente aninhado do tipo GRAPH, delimitado pelo parâmetro de contexto ***graph_type***, o qual implementa algoritmos de grafos de pequena escala baseado em vértices e arcos, podendo oferecer suporte tanto a grafos direcionados quanto a grafos não-direcionados, dependendo do contexto delimitado pelos argumentos de contexto. Além disso, é distribuído, assumindo um subgrafo em cada unidade de processamento. A implementação de cada subgrafo estende uma interface de operações sobre grafos baseada em JGraphT (NAVEH *et al.*, 2003). Porém, GRAPH utiliza tipos primitivos de C# e programação genérica para reduzir o consumo de memória, comparado com a implementaçção original de JGraphT, em Java.

INPUTBIN é um componente usado por GUSTY e GUSTYFUNCTION para extrair a estrutura de grafos e criar instâncias do componente GRAPH. No início da computação, INPUTBIN atua no particionamento do grafo através do corte de vértices, a fim de criar um vértice principal e seus espelhamentos distribuídos. Durante o particionamento, subgrafos são transferidos às unidades de processamento, alimentando as instâncias de GRAPH.

Tabela 5 – Argumentos de Contexto para as instâncias de Gusty para *TC*, *SSSP* and *PageRank*

Parameter Name	*TriangleEnumeration*	*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	TC0[...] / TC1[...] / TC2[...]	SSSP[...]	PAGERANK[...]
output_key_type	VERTEXBLOCK	VERTEXBLOCK	VERTEXBLOCK
output_value_type	DATATRIANGLE	DATASSSP	DATAPGRANK

Em implementações de GUSTYFUNCTION, o usuário deve prover código para as operações ***unroll***, ***compute*** e ***scatter***. ***Unroll*** coleta novos pares chave-valor, produzidos assincronamente, do iterador de entrada, a fim de incrementar valores que pertencem à mesma chave. Ou seja, quando o programador insere um par no *buffer* de saída (disponível à operação *scatter*) esse par torna-se disponível para ser consumido na próxima iteração de uma computação iterativa, sendo recuperado e incrementada no *unroll* subsequente. Por sua vez, ***compute*** é chamado após *unroll* completar sua execução. É onde o desenvolvedor insere a lógica da computação. Finalmente, ***scatter***, habilitado após o passo de computação, é usado para enviar dados para o iterador de saída.

Figura 18 – Sistema de Computação Paralela Gust para *TC* (Enumeração de Triângulos)

Para fins de prova de conceito e estudos de caso, Cenez Rezende, em sua Tese de Doutorado, desenvolveu implementações de GUSTYFUNCTION para sistemas de computação paralela que resolvem três problemas de grafos frequentemente usados para avaliar a expressividade e desempenho de arcabouços de processamento de grafos: *enumeração de triângulos* (*TC*), *menor caminho de única fonte* (*SSSP*) e *ranqueamento de páginas* (*PageRank*).

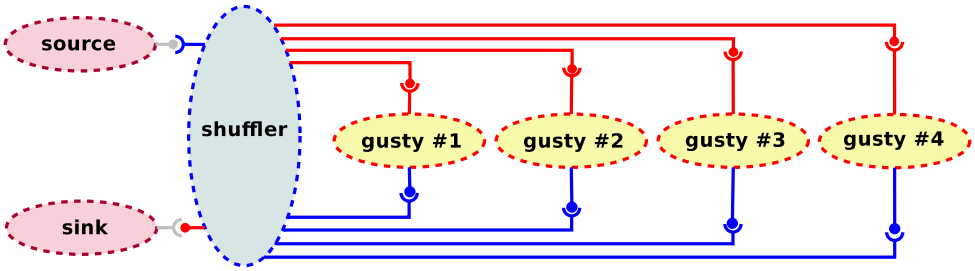

Figura 19 – Sistema Gust para *SSSP* and *PageRank*, distribuído em 4 Plataformas Virtuais

A Figura 18 apresenta a arquitetura de um sistema Gust para a enumeração de triângulos, empregando três agentes GUSTY sequenciais, alocados em plataformas virtuais distintas. Cada agente executa uma fase do algoritmo MapReduce para enumeração de triângulos, sendo portanto configurados, através do parâmetro de contexto ***reduce_function*** com diferentes funções de redução: TC0, TC1 e TC2. Por usa vez, a Figura 19 apresenta a arquitetura de um sistema MapReduce iterativo, que serve tanto para textitSSSP (número dinâmico de iterações) quanto para *PageRank* (número fixo de iterações). Finalmente, a Tabela 5 resume os argumentos contextuais para os componentes GUSTY empregados nas arquiteturas dos sistemas que implementam os três algoritmos.

4 PROVENIÊNCIA PROSPECTIVA E RETROSPECTIVA NA HPC SHELF

Com o intuito de estender características da HPC Shelf relacionadas a construção e execução de *workflows* científicos, propomos um mecanismo para habilitar recursos de proveniência prospectiva e retrospectiva, incluindo:

- o tratamento de *workflows* SAFeSWL como componentes da plataforma, visando habilitar a proveniência prospectiva (Seção 4.1);
- o registro das ações realizadas pelos componentes, a fim de habilitar recursos de proveniência retrospectiva (Seção 4.2);

4.1 *Workflows* como Componentes (Proveniência Prospectiva)

Um especialista, tido como usuário final de uma plataforma HPC Shelf, descreve o problema que deseja resolver utilizando uma aplicação, desenvolvida sobre o SAFe, o sistema gerenciador de *workflows* científicos da HPC Shelf. Essa aplicação, fornecida por um provedor de aplicações, atende a um domínio específico e utiliza recursos de fácil manipulação para um usuário especialista, como uma linguagem de propósito especial, possivelmente utilizando elementos de abstração visual, voltado ao seu domínio de interesse. Dessa forma, a aplicação isenta o especialista de conhecer a arquitetura da HPC Shelf, abstraindo-se da complexidade da composição e execução de seus sistemas de computação paralela.

Portanto, a aplicação, por intermédio do SAFe, constitui-se responsável pela escolha dos componentes apropriados, bem como a especificação de como devem ser orquestrados, para resolver problemas descritos pelo especialista. Para isso, a aplicação traduz o problema em uma solução especificada através de um sistema de computação paralela, descrito por meio dos subconjuntos arquitetural e de orquestração da linguagem SAFeSWL. O programa de orquestração é chamado de *workflow*. Atualmente, esses *workflows* são *efêmeros*, uma vez que são gerados dinamicamente e então submetidos ao SAFe, sem possibilidade de serem reusados.

Em SGWfCs tradicionais, alternativos ao SAFe, o especialista constrói diretamente o *workflow*, possivelmente usando uma linguagem visual, lidando diretamente com elementos de composição. A noção arquitetural encontra-se implícita junto com o *workflow*, não havendo tal separação entre interesses arquiteturais e de orquestração. Esses *workflows* em geral podem ser armazenados para uso posterior, de forma a permitir a reprodução de experimentos e simulações. O registro dos *workflows* pode também prestar-se a agir como uma *cache*, evitando a geração de

um mesmo *workflow* para uma mesma solução. O fato de que *workflows* gerados (estaticamente ou dinamicamente) possam ser salvos de alguma forma (persistência), e depois reusados, pode ser encarado como o registro de *proveniência prospectiva*, não suportado pelos *workflows*, efêmeros, da HPC Shelf.

Uma abordagem para a proveniência prospectiva na HPC Shelf é o tratamento de sistemas de computação paralela, envolvendo tanto sua arquitetura quanto seu *workflow* de orquestração, como uma nova espécie de componentes. Dessa forma, tanto o código de orquestração (*workflow*) quanto o código arquitetural de um sistema de computação paralela, gerados pela aplicação, pode ser registrados no catálogo de componentes mantido pelo Core. Desse modo, outras aplicações, ou a própria aplicação, podem reusá-los. Além disso, a integração dos sistemas de computação paralela com o sistema de contratos contextuais pode oferecer oportunidades para escolha entre sistemas alternativos de acordo com um contexto de execução, assim como é feito com componentes das demais espécies, levando à proposta de um sistemas abstratos, cuja parametrização possa ser fornecida em tempo de execução através da comunicação entre a aplicação e os componentes de solução nos sistemas de computação paralela.

Para que componentes que representam sistemas de computação paralela possam ser compatibilizados com o mecanismo de contratos contextuais da HPC Shelf, faz-se necessário definir suas noções abstrata e concreta de *workflows*, descritas nas duas seções a seguir.

4.1.1 Sistemas Abstratos

Sistemas de computação paralela típicos do domínio da aplicação podem ser tratados através da noção de componentes abstratos de uma nova espécie de componentes, assim chamados de *sistemas*, os quais são desenvolvidos por provedores de aplicações e registrados no catálogo de componentes mantido pelo Core para usufruto e reuso tanto da própria aplicação dos provedores que os desenvolveram quanto de aplicações de provedores terceiros.

A noção de sistema abstrato fornece uma abstração para sistemas dotados de pontos de variabilidade, através dos parâmetros de contexto de suas assinaturas contextuais. Dessa forma, além de os parâmetros de contexto de cada sistema abstrato poderem ser usados para guiar a escolha dos componentes que representam sistemas concretos, bem como serem repassados aos contratos contextuais de seus componentes de solução a fim de guiar sua seleção e especificar seus requisitos funcionais e não-funcionais, podem ser usados pelos componentes sistema concretos para guiar a construção de sistemas de computação paralela finais, como descrito a seguir.

4.1.2 Sistemas Concretos

Um sistema concreto é capaz de sintetizar dinamicamente um sistema de computação paralela a partir dos argumentos de contexto repassados por meio de seu contrato contextual, definido por uma arquitetura de componentes de solução e uma orquestração para tais componentes. Por exemplo, parâmetros com valores numéricos poderiam especificar a cardinalidade de regimentos de componentes de computação distribuídos em múltiplas plataformas virtuais para solução de um problema, seja organizados segundo o padrão *pipeline* ou segundo o padrão *farm*. Devido a esse caráter dinâmico, com pontos de variabilidade especificados por parâmetros de contexto, sistemas concretos são ditos representar *templates* de sistemas de computação paralela, e não sistemas de computação paralela finais prontos para execução.

Código-fonte 1 – `ITaskPort` Interface (C#)

```
1  public interface ITaskBindingKind : IActivateKind, GoPort
2  {
3    //synchronous, simple, no reaction code
4    void invoke(object action);
5
6    //synchronous, multiple, no reaction code
7    object invoke(object[] action);
8
9    //asynchronous, simple, no reaction code
10   void invoke(object action, out IActionFuture f);
11
12   //asynchronous, multiple, no reaction code
13   void invoke(object[] action, out IActionFuture f);
14
15   //asynchronous, simple, reaction code
16   void invoke(object action, Action reaction, out IActionFuture f);
17
18   //asynchronous, multiple, reaction code
19   void invoke(object[] action, Action[] reaction, out IActionFuture f);
20 }
21
22 public interface IActionFuture
23 {
24   void wait();                  // waits for completion of the action
25   bool test();                  // tests the completion of an action
26   IActionFutureSet createSet(); // creates a set of pending actions
27   object Action { get; }        // retrieves the reaction code to be executed
28   void registerWaitingSet (AutoResetEvent waiting_set);
```

```
29    void unregisterWaitingSet (AutoResetEvent waiting_set);
30  }
31
32  public interface IActionFutureSet: IEnumerable<IActionFuture>
33  {
34    void addAction(IActionFuture f); // add a new activation action
35    void waitAll();                  // waits for completion of all pending actions
36    IActionFuture waitAny();         // waits for completion of any pending action
37    bool testAll();                  // tests the completion of all pending actions
38    IActionFuture testAny();         // tests the completion of any pending action
39    IActionFuture[] Pending { get; } // get the list of pending actions (not completed)
40  }
```

Como contribuição acessória desta dissertação, é implementada a extensão da ideia original da HPC Shelf no projeto do SAFe para que a especificação do sistema de computação paralela não precise necessariamente ser realizada através da geração de código SAFeSWL. A alternativa dinâmica a ser suportada será a adoção de APIs tanto para a criação de componentes e suas ligações quanto para sua orquestração.

A interface de orquestração, acessada na implementação do componente *Workflow*, será a mesma utilizada por componentes de solução, apresentada no Código-fonte 1, em C#, linguagem hospedeira da implementação atual do SAFe. Ela inclui algumas variações do procedimento `invoke`, nas seguintes dimensões:

- **síncrona *versus* assíncrona**;
- **simples *versus* múltipla**;
- **com operação de reação *versus* sem operação de reação**.

Para melhor compreensão, o leitor deve entender a semântica da ativação de ações em *bindings* de ação na HPC Shelf, explicada na Seção 3.4 e ilustrada através da Figura 6. A ativação de uma ação, referenciada pelo seu *nome de ação* ($\langle action\_name \rangle$) em uma das portas de cada componente ligadas através de um *binding* de ações, só se completa quando há uma operação de ativação pendente em cada porta ligada por intermédio do *binding*.

Na invocação síncrona, o controle só retorna da chamada ao `invoke` depois que a ativação da ação se completa. Por sua vez, na invocação assíncrona, o controle retorna imediatamente, junto com um objeto do tipo `IActionFuture`, que pode ser usado para aguardar que a ativação da ação se complete ou testar se já se completou. A invocação simples é usada em ações que não são alternativas, representada por um único nome de ação, enquanto a invocação múltipla é usada para ações alternativas, aquelas que possuem vários nomes de ações associadas

e são utilizadas em escolhas e decisões sobre terminação de iterações no código SAFeSWL. O retorno da operação invoke, no caso de invocação múltipla, permite identificar qual das ações alternativas foi ativada nas portas de ação envolvidas. Finalmente, pode ser associada à chamada de `invoke` um método de reação, do tipo `Action`, que é chamado automaticamente quando a ativação da ação se completa. Note que, no caso de chamada múltipla, há um vetor de operações de reação, para cada nome de ação alternativo. Em C# a operação de reação é implementada por meio do recurso de *delegados*.

Código-fonte 2 – `BuilderService` Interface (C#)

```
1  public interface BuilderService : cca.Port
2  {
3      cca.ComponentID createInstance(string instanceName, string className,
4                                     cca.TypeMap properties);
5      cca.ComponentID[] getComponentIDs();
6      cca.TypeMap getComponentProperties(cca.ComponentID cid);
7      void setComponentProperties(cca.ComponentID cid, cca.TypeMap map);
8      cca.ComponentID getDeserialization(string s);
9      cca.ComponentID getComponentID(string componentInstanceName);
10     void destroyInstance(cca.ComponentID toDie, float timeout);
11     string[] getProvidedPortNames(cca.ComponentID cid);
12     string[] getUsedPortNames(cca.ComponentID cid);
13     cca.TypeMap getPortProperties(cca.ComponentID cid, string portName);
14     void setPortProperties(cca.ComponentID cid, string portName, cca.TypeMap map);
15     cca.ConnectionID connect(cca.ComponentID user, string usingPortName,
16                              cca.ComponentID provider, string providingPortName);
17     cca.ConnectionID[] getConnectionIDs(cca.ComponentID[] componentList);
18     cca.TypeMap getConnectionProperties(cca.ConnectionID connID);
19     void setConnectionProperties(cca.ConnectionID connID, cca.TypeMap map);
20     void disconnect(cca.ConnectionID connID, float timeout);
21     void disconnectAll(cca.ComponentID id1, cca.ComponentID id2, float timeout);
22 }
```

A interface de criação e ligação de componentes (descrição arquitetural) é a interface `BuilderService` do modelo de componentes do CCA, apresentado no Código-fonte 2. Para isso, o sistema concreto não deve possuir um arquivo `architecture.safeswl` associado às suas unidades. Em vez disso, o código de sua unidade deve conectar-se à porta de serviço *builder services*, como previsto no padrão CCA, e usá-la para criar componentes e suas ligações.

Muito embora tenha sido proposta para criação dinâmica de instâncias de compo-

nentes CCA, e suas conexões, a interface `BuilderServices` possui a expressividade necessária para a descrição arquitetural suportada por SAFeSWL, especialmente devido a flexibilidade oferecida pelo parâmetro `properties` na criação de instâncias de componentes. Como o tipo desse parâmetro é um mapeamento de chaves a valores, podemos utilizá-lo para informar as portas usuárias, provedoras e de ações do componente, bem como as informações sobre suas facetas. Além disso, deve-se notar que as ligações entre portas de ações também serão realizadas através da operação `connect`, a qual foi projetada para ligações entre portas usuárias e provedoras no CCA. Nesse caso, a ordem das portas não é relevante. Caso seja necessário conectar mais de duas portas de ações, essas podem ser referenciada em múltiplas chamadas à operação `connect`.

Vale ressaltar, finalmente, que o uso de SAFeSWL faz-se útil, além da descrição arquitetural por seu intermédio, como formato de armazenamento de sistemas de computação paralela gerados dinamicamente através da API, para fins de proveniência retrospectiva, como descrito na próxima seção.

4.2 Suporte a Proveniência Retrospectiva

O suporte a reprodução de experimentos é uma ferramenta importante para a pesquisa científica, notadamente no contexto da publicação de seus resultados. Algumas situações, como a modificação de um gráfico, requer identificar qual *script*/programa, versão, conjunto de dados (*dataset*) e outros recursos utilizados para obter exatamente aquele mesmo gráfico. Geralmente, os pesquisadores não fazem um controle tão elaborado dos recursos utilizados e acabam se perdendo diante de tantas execuções de teste, mudanças de parâmetros, pre-processamentos de *datasets*, etc. O suporte à reprodução de experimentos torna mais fácil, para o pesquisador, a tarefa de realizar pesquisas baseadas nos resultados de suas próprias pesquisas anteriores, ou de outros pesquisadores, trazendo, consequentemente, mais confiabilidade e visibilidade para os resultados de uma pesquisa. A possibilidade de um terceiro reproduzir um experimento reforça a tarefa de revisar um trabalho trazendo maior confiabilidade aos seus apontamentos pois raramente o revisor tem disponibilidade para implementar o experimento descrito e analisar de forma mais minuciosa, nos casos, também raros, onde isso é possível.

Os dados de proveniência retrospectiva são utilizados com o intuito de habilitar a reprodução de simulações e experimentos científicos *in-silico*. No entanto, existem desafios a serem enfrentados para viabilizar tal possibilidade, os quais valem para qualquer sistema que seja apropriado a isso, tais como:

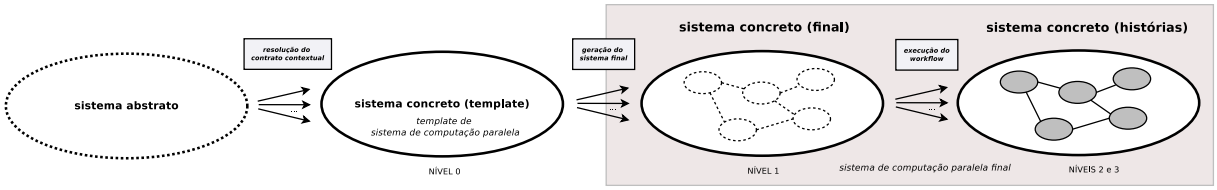

Figura 20 – Componentes Sistema e Níveis de Proveniência Retrospectiva

- realizar o rastreamento da proveniência de forma automática e simples;
- assegurar que os dados de proveniência sejam suficientes e completos;
- garantir o armazenamento de longo prazo, confiável e sustentável sobre infraestrutura estável e baseada em padrões abertos;
- manter consistente as referências entre artigos/publicações, resultados, códigos e dados.

Através da nova espécie de componentes sistema, propõe-se um mecanismo para registro das informações sobre a execução de sistemas de computação paralela da HPC Shelf. As informações de proveniência retrospectiva na HPC Shelf podem ser classificadas em 4 níveis:

- **Nível 0**: a seleção do *sistema concreto* a partir de um contrato contextual para um *sistema abstrato*;
- **Nível 1**: o sistema de computação paralela final gerado pelo sistema concreto para o contexto delimitado pelo contrato do sistema abstrato;
- **Nível 2**: a seleção dos componentes de solução a partir dos seus contratos contextuais especificados no sistema de computação paralela gerado;
- **Nível 3**: a sequência de ativação de ações durante a orquestração, tanto do ciclo de vida dos componentes quanto computacionais, de maneira total ou parcial.

A noção de sistema concreto, apresentada na Seção 4.1.2, define uma base para o nível 0 de proveniência retrospectiva. Para o suporte aos demais níveis de proveniência retrospectiva, é necessária uma noção mais geral de sistema concreto que representa uma *história parcial* referente a execução de um sistema concreto do nível 0, desde a criação do sistema de computação paralela final até a conclusão de sua orquestração, como ilustrado na Figura 20. Uma história (parcial) é dita *total* quando suporta até o nível 3 de proveniência retrospectiva e tem, portanto, todo o fluxo de ativação de ações de uma execução de um sistema de computação paralela registrado. Por exemplo, um sistema concreto que representa um fluxo total pode representar um determinado experimento científico que precisa ser reproduzido. Em um fluxo parcial, suporta-se um certo nível de variabilidade, incluindo no nível 3, onde certas partes do fluxo de orquestração podem executar segundo a lógica determinada pelo sistema de nível 0.

Portanto, um sistema concreto de nível 0 pode resultar na criação de vários outros sistemas concretos, para diferentes execuções, com diferentes pontos de variabilidade relativos à sua história de execução, de acordo com os níveis de proveniência suportados.

Vale ressaltar que, no caso de um sistema de computação paralela final ter sido criado através da interface `BuilderService`, dinamicamente, o subconjunto arquitetural SAFeSWL é o formato usado para armazenar o sistema geral em um sistema concreto de nível 1.

4.2.1 Parâmetro de Contexto para Delimitar Experimento

Com a finalidade de suportar a escolha de sistemas concretos que representam histórias parciais de execução de sistemas de computação paralela previamente executados, sistemas abstratos incluem obrigatoriamente em sua assinatura um parâmetro de contexto qualificador chamado ***experiment_label***. Após a execução de um sistema de computação paralela, notadamente gerado a partir de um sistema concreto, um rótulo pode ser associado para identificar a história de execução associada de forma única no catálogo de componentes. Esse rótulo, dinamicamente gerado, é representado como um componente qualificador da HPC Shelf.

Através do parâmetro de contexto ***experiment_label***, uma aplicação precisa apenas conhecer o rótulo do sistema concreto associado ao experimento, que representa a sua história de execução, para reexecutá-lo. A aplicação poderia portanto fornecer ao cientista o próprio rótulo de experimento, o qual poderia ser executado usando a própria aplicação ou por meio de alguma aplicação especificamente desenvolvida, independente da aplicação original, para reexecução de experimentos. Isso é possível devido a natureza baseada em componentes dos sistemas de computação paralela, os quais são registrados no catálogo de componentes do Core de forma independente em relação à aplicação.

A relação de subtipos entre rótulos de experimentos é definida de tal modo que: se uma execução de rótulo I de um determinado sistema é criada a partir de uma execução de rótulo J do mesmo sistema, então I é subtipo de J. Isso significa que a execução I é segura de ser executada quando é solicitada uma reexecução de J, uma vez que executa todos os passos previstos em J sobre restrições possivelmente mais fortes. Portanto, o sistema concreto de nível 0 possui como rótulo um qualificador que é supertipo do rótulo de qualquer sistema concreto gerado a partir da sua execução.

4.2.2 Repositórios de Dados

Neste trabalho, assume-se que os repositórios de dados utilizados pelos *workflows* não são passíveis de modificação, ou seja, o acesso aos dados é do tipo somente-leitura. Portanto, na reprodução de um experimento, através da execução de um *workflow* que o representa, não trataremos a evolução desses repositórios, que poderiam ser alimentados dinamicamente pela execução de um novo experimento. Para isso, seria necessário desenvolver um mecanismo de proveniência para os repositórios de dados, o que deixamos para eventuais trabalhos futuros.

Muito embora possa ser enxergado como uma limitação do trabalho proposto, ao ponto de considerarmos algo a ser tratado como extensão em trabalhos futuros, é de fato realístico assumir, na maioria dos casos, que repositórios de dados são somente para leitura, ou possam fornecer versões de seu conteúdo que foram utilizadas no tempo da execução de um determinado experimento, de modo a permitir a sua reprodução fiel. Além disso, é possível que, pela própria semântica dos dados do repositório, o resultado da execução de um experimento não seja sensível a mudanças, ou haja tolerância em relação a diferenças nos resultados.

4.2.3 Porta de Proveniência

Com o propósito de habilitar os recursos de proveniência em uma execução, o componente Application do sistema de computação paralela possui, por *default*, uma porta usuária de proveniência, a qual deve ser conectada à porta de serviço `ProvenanceService` anteriormente ao início da execução da lógica de orquestração do componente Workflow.

Código-fonte 3 – `IProvenance` Interface (C#)

```
1 public interface IProvenance
2 {
3     void start (int level);
4     string confirm ();
5     void cancel ();
6 }
```

A interface da porta `ProvenanceService` encontra-se apresentada no Código-fonte 3. A operação `start` habilita os recursos de proveniência no nível informado através do parâmetro `level`. Por sua vez, a operação `confirm` aguarda até o final da execução do sistema de

computação paralela, retornando uma cadeia de caracteres que representa o rótulo de experimento associado ao sistema concreto gerado. Normalmente, espera-se que essa operação seja invocada ao final da execução. Finalmente, a operação `cancel` desabilita os recursos de proveniência durante a execução, não sendo possível realizar uma nova chamada a `start` para reinicializá-la.

4.3 Exemplo com o framework MapReduce

A seguir, é apresentado um exemplo, baseado no framework MapReduce apresentado no Capítulo 3, que visa descrever o processo de resolução de um problema utilizando sistemas de computação paralela da HPC Shelf com suporte a proveniência, prospectiva e retrospectiva. O exemplo possui algumas simplificações, de modo a tornar mais fácil o entendimento do leitor. No Capítulo 5, um exemplo mais detalhado, bem como realístico, é apresentado.

4.3.1 Sistema Abstrato

Com base em entradas fornecidas pelo especialista, interessado em resolver um problema, a aplicação solicita a construção de um sistema de computação paralela que representa uma solução. Neste exemplo, assume-se a existência de um *framework* de sistemas de computação paralela que representam computações MapReduce, com base em instâncias dos componentes MAPPER, REDUCER, SPLITTER e SHUFFLER. O componente base desse *framework* se chama mapreduce.system.SYSTEM. A partir de seus subtipos, componentes abstratos derivados a partir dele, são instanciados componentes concretos capazes de gerar arquiteturas particulares de sistemas MapReduce. Para diferenciar cada sistema, o componente mapreduce.system.SYSTEM possui um parâmetro de contexto de nome ***system_type***, da espécie qualificador, com um valor base mapreduce.systemtype.base.SYSTEMTYPE, cujos subtipos rotulam arquiteturas particulares de sistemas MapReduce. Por exemplo, assumimos a existência de um tipo mapreduce.systemtype.simple.SYSTEMTYPE, que determina a escolha de um sistema MapReduce comum, chamado mapreduce.system.simple.SYSTEM, composto de uma única etapa de mapeamento seguida de uma única etapa de redução, como o da Figura 21. Logo, mapreduce.system.simple.SYSTEM é um subtipo de mapreduce.system.SYSTEM para o qual ***system_type*** = mapreduce.systemtype.simple.SYSTEMTYPE. Para esse sistema, poderão haver múltiplos agentes de mapeamento e redução (pontos de variabilidade), executando em plataformas virtuais distintas. O número de agentes de mapeamento e redução é delimitado pelos

Nome	Limite	Descrição
input_key_type	DATA	tipo das chaves dos pares de entrada
input_value_type	DATA	tipo dos valores nos pares de entrada
intermediate_key_type	DATA	tipo das chaves dos pares intermediários
intermediate_value_type	DATA	tipo dos valores nos pares intermediários
output_key_type	DATA	tipo das chaves dos pares de saída
output_key_value	DATA	tipo dos valores nos pares de saída
map_function	MAPFUNCTION	the função de mapeamento
reduce_function	REDUCEFUNCTION	função de redução
split_function	PARTITIONFUNCTION	função de particionamento (fase *split*)
shuffle_function	PARTITIONFUNCTION	função de particionamento (fase *shuffle*)
size_mappers	INTEGER	quantidade de agentes de mapeamento
size_reducers	INTEGER	quantidade de agentes de redução

Tabela 6 – Assinatura Contextual do Sistema mapreduce.system.simple.SYSTEM

parâmetros de contexto ***size_mappers*** e ***size_reducers***, que aparecem na assinatura contextual do componente mapreduce.system.simple.SYSTEM, do qual é derivado diretamente o componente concreto que representa esses sistemas MapReduce simples. A Tabela 6 apresenta os parâmetros de contexto presente na assinatura contextual de mapreduce.system.simple.SYSTEM.

Portanto, se um provedor deseja executar uma contagem de palavras empregando quatro plataformas virtuais para executar agentes de redução, deve construir um contrato contextual como se segue:

mapreduce.system.SYSTEM [***system_type*** =mapreduce.system_type.simple.SYSTEMTYPE,
size_mappers = 1,
size_reducers = 4,
map_function = 1,
reduce_function = 1,
input_key_type = INTEGER
input_value_type = STRING,
intermediate_key_type = STRING,
intermediate_value_type = INTEGER,
output_key_type = STRING,
output_value_type = INTEGER]

4.3.2 Sistema Concreto (nível 0)

Tendo em vista que há um único sistema concreto (*template* de sistema de computação paralela) cujo valor do parâmetro ***system_type*** é mapreduce.systemtype.simple.SYSTEMTYPE,

este é selecionado pelo sistema de contratos contextuais, constituindo o sistema de nível 0 para o mecanismo de proveniência. Ou seja, em uma reexecução de nível 0 de proveniência, a resolução do componente sistema não seria realizada, sendo escolhido o mesmo componente concreto da execução original. Para o exemplo em questão, isso é irrelevante, uma vez que, por definição, há um único componente concreto para cada valoração de ***system_type***. Entretanto, em uma implementação completa da HPC Shelf vários sistemas concretos poderiam existir para diferentes valorações de parâmetros de contexto que representam critérios de QoS (Qualidade de Serviço).

4.3.3 Sistema Concreto (nível 1)

Os valores dos parâmetros de contexto podem ser acessados pelo sistema concreto a fim de construir o sistema de computação paralela final, o qual constitui o sistema concreto de nível 1 para o sistema de proveniência. No caso do sistema mapreduce.system.simple.SYSTEM, corresponde a configuração de um sistema com um certo número de agentes de mapeamento e redução, único ponto de variabilidade presente na configuração do sistema. Para o sistema em questão, esses valores são 1 e 4, ou seja, serão instanciados um único agente de mapeamento e 4 agentes de redução a fim de executar a contagem de palavras.

Os demais parâmetros de contexto são usados para configurar as instâncias de componentes presentes no sistema MapReduce. A Figura 22 mostra um trecho de código que define a instância do componente MAPPER presente na arquitetura.

4.3.4 Sistema Concreto (nível 2)

Havendo um sistema de computação paralela final gerado a partir do sistema concreto, a execução propriamente dita do *workflow* pode iniciar. A execução consiste em uma sequência de ativação de ações nas portas de ações de componentes de computação e conectores, guiada pelo componente *Workflow* do sistema de computação paralela. Dentre essas ações, estão ações

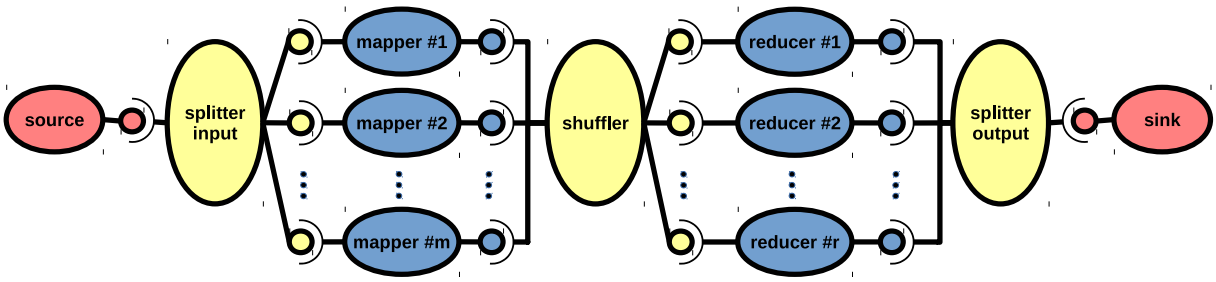

Figura 21 – Esquema Arquitetural do Sistema mapreduce.system.simple.SYSTEM

Parâmetro	Argumento	Description
input_key_type	INTEGER	número da linha/parágrafo
input_value_type	STRING	conteúdo da linha/parágrafo
intermediate_key_type	STRING	palavra-chave
intermediate_value_type	INTEGER	constante inteira 1, para cada palavra-chave
output_key_type	STRING	palavra-chave
output_key_value	INTEGER	número de ocorrências no texto
map_function	COUNTWORDS[...]	identificação de palavras-chave
reduce_function	SUMVALUES[...]	contagem das palavras-chave
splitter_function	*não informado*	particionamento *default*
shuffler_function	*não informado*	particionamento *default*
size_mappers	2	único agente de mapeamento
size_reducers	4	único agente de redução

Tabela 7 – Contrato Contextual de mapreduce.system.simple.SYSTEM para Contagem de Palavras

Figura 22 – Exemplo da definição do componente Mapper em SAFeSWL

```
0  <computation id_component="mapper">
1      <user_port id_port="collect_pairs"/>
2      <user_port id_port="platform_map"/>
3      <provider_port id_port="feed_pairs"/>
4      <action_port id_port="task_map">
5          <action name="READ_CHUNK"/>
6          <action name="PERFORM"/>
7          <action name="CHUNK_READY"/>
8      </action_port>
9      <action_port id_port="life_cycle">
10         <action name="RESOLVE"/>
11         <action name="DEPLOY"/>
12         <action name="INSTANTIATE"/>
13         <action name="RUN"/>
14         <action name="RELEASE"/>
15     </action_port>
16 </computation>
```

Fonte: Elaborado pelo autor.

das portas de controle do ciclo de vida desses componentes, onde a ação `RESOLVE` corresponde a resolução do contrato contextual do próprio componente. Logo, deve ser ativada antes de qualquer outra ação relativa ao componente.

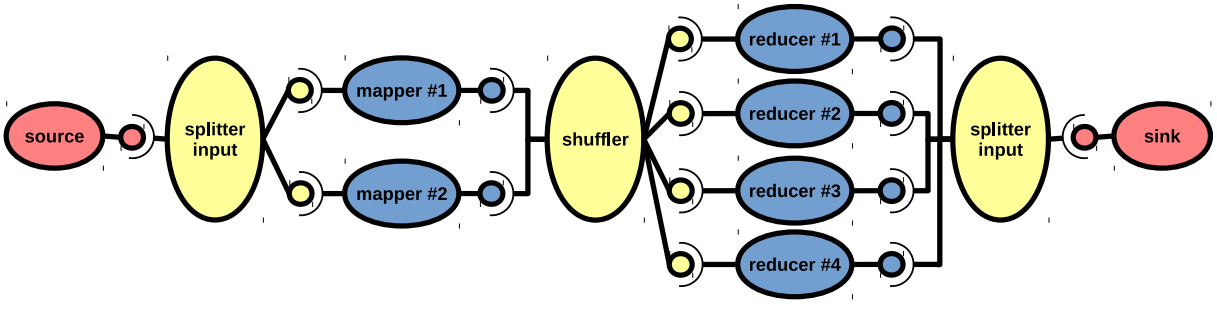

Figura 23 – Arquitetura mapreduce.system.simple.SYSTEM para Contagem de Palavras

O sistema concreto de nível 2 corresponde ao sistema de computação paralela final considerando todas as resoluções de componentes para uma dada execução. Portanto, em uma reexecução de um sistema concreto de nível 2, tanto o sistema de computação paralela final quanto as escolhas dos seus componentes de solução (e.g. agentes de mapeamento e redução) são os mesmos em relação à execução original.

4.3.5 Sistema Concreto (nível 3)

Conforme a orquestração vai ocorrendo, caso o mecanismo de proveniência esteja habilitado, a sequência de ativações de ações nas portas dos componentes são registradas. Ao final da execução, são consolidadas em um sistema concreto de nível 3.

É importante frisar que as informações de proveniência de nível 2 e 3 são geradas concomitantemente, pois a resolução é uma operação habilitada pela ativação de uma ação (`RESOLVE`), como explicado na seção anterior. Na reexecução de um sistema concreto de nível 3, a ativação da ação `RESOLVE` simplesmente recupera a escolha realizada na execução original do procedimento de resolução.

A Figura 24 ilustra um pequeno trecho de uma sequência de ações ativadas em um determinada execução do problema de contagem de palavras.

Uma reexecução utilizando dados de proveniência de nível 3 indica executar exatamente a sequência de ações registradas em uma execução anterior. Na implementação atual, são gravadas todas as ações da execução do componente. Entretanto, seria possível restringir o mecanismo de modo a termos uma gravação parcial, somente em alguns trechos da execução. Pretende-se oferecer esse suporte em futuras versões do mecanismo.

Figura 24 – Exemplo de ações registradas em XML

```
0  <provenance>
1      <action action_name="resolve"        port_name="life-cycle-splitter" />
2      <action action_name="deploy"         port_name="life-cycle-splitter" />
3      <action action_name="instantiate"    port_name="life-cycle-splitter" />
4      <action action_name="resolve"        port_name="life-cycle-mapper" />
5      <action action_name="deploy"         port_name="life-cycle-mapper" />
6      <action action_name="instantiate"    port_name="life-cycle-mapper" />
7      ...
8      <action action_name="collector-read-chunk"  port_name="split-feeder-chunk" />
9      <action action_name="feeder-read-ready"     port_name="split-feeder-chunk" />
10     <action action_name="read-chunk"     port_name="task-map" />
11     <action action_name="perform"        port_name="task-map" />
12     ...
13     <action action_name="release"        port_name="split-feeder-chunk" />
14     <action action_name="release"        port_name="task-map" />
15 </provenance>
```

Fonte: Elaborado pelo autor.

5 ESTUDO DE CASO E PROVA DE CONCEITO: SISTEMAS GUST

Neste capítulo, é apresentado um estudo de caso para fins de prova de conceito com o mecanismo de proveniência apresentado no Capítulo 4 para a HPC Shelf. Para essa finalidade, é utilizado o arcabouço Gust (REZENDE C. A.; CARVALHO-JUNIOR, 2018), apresentado na Seção 3.8, o qual propõe fornecer uma interface e modelo de programação de alto nível para processamento paralelo heterogêneo e de larga escala sobre grafos grandes. O estudo de caso é baseado na construção de um arcabouço Gust para uma aplicação destinada a resolver um certo conjunto fixo de problemas de processamento de grafos. Para isso, é projetada uma arquitetura de sistemas abstratos e concretos a fim de atender aos requisitos dessa aplicação.

Como descrito na Seção 3.8, o Gust estende o MapReduce introduzindo os componente abstratos Gusty, derivado do Reducer, e GustyFunction, derivado de ReduceFunction. Basicamente, a fim de customizar um sistema Gust para resolver um determinado problema, o provedor de aplicações deve fornecer um argumento para satisfazer o parâmetro ***reduce_function*** de agentes Gusty presentes no sistema, notadamente com contratos contextuais de componentes abstratos derivado de GustyFunction, os quais descrevem o papel de cada agente.

Através do Gust, sistemas de computação paralela formados por agentes GUSTY, bem como agentes de mapeamento e redução de MapReduce, podem ser construídos para implementar diferentes arquiteturas de processamento em grafos. Atualmente, os sistemas são compostos dinamicamente usando SAFeSWL. Não há nenhuma preocupação, a nível dos serviços da HPC Shelf, com o registro dessa composição para reuso dentro de uma mesma aplicação ou entre aplicações distintas. A cada nova solução que precisa ser desenvolvida, um novo sistema (código SAFeSWL, arquitetural e de orquestração) precisa ser sintetizado, mesmo que sua arquitetura tenha pequenas diferenças em relação a sistemas anteriormente empregados. Entretanto, com o mecanismo de proveniência proposto neste trabalho, sistemas Gust podem ser tratados como componentes. Assim, podem ser reutilizados dentro de uma mesma aplicação ou entre aplicações diferentes. A definição de um arcabouço básico de sistemas de computação paralela para determinada finalidade torna-se então possível.

Portanto, a extensão ao Gust proposta neste trabalho permite que sistemas de computação paralela sejam registrados no catálogo de componentes. Assim, sistemas que apresentem certos padrões arquiteturais, com elementos de composição similares, bem como as relações entre tais elementos, podem se beneficiar dessa característica, utilizando sistemas já registrados ou estendendo sistemas pré-existentes para atender a requisitos mais específicos.

Na seções que se seguem, faremos uso dos sistemas de computação paralela originalmente propostos na Tese de Doutorado de Cenez Rezende neste estudo de caso. Tais sistemas constituem soluções para algums problemas de processamento de grafos: *Clique*, Enumeração de Triângulos (*TC*), Caminho mais Curto de Fonte Única (*SSPS*) e Ranqueamento de Páginas (*PageRank*). Na próxima seção, apresentamos uma explicação de cada um dos problemas, e suas soluções que têm sido utilizadas neste trabalho.

5.1 Algoritmos de Processamento de Grafos

Os algoritmos descritos nessa seção são utilizados como estudo de caso para validar nossa proposta. Esse algoritmos são considerados os mais disseminados na avaliação de processamento de grafos de larga escala, e potencialmente consomem mais recursos computacionais de processamento, comunicação e memória.

5.1.1 Clique

O Clique se refere ao tradicional problema da teoria dos grafos que busca analisar vértices totalmente conectados (subgrafos completos) em um conjunto de vértices, de modo que exista uma aresta para todo par de vértices do clique. Problemas relacionados podem ser: encontrar o maior clique, encontrar os cliques maximais, verificar a existência de um clique de determinado tamanho, dentre outros.

O problema em questão neste trabalho consiste em encontrar todos os cliques maximais. Alguns algoritmos paralelos são desenvolvidos em três etapas: dividir um grafo em subgrafos distribuídos; computar o subgrafo localmente; e agrupar os resultados. Nós utilizamos a solução proposta por (SVENDSEN *et al.*, 2015) com o algoritmo de (BRON; KERBOSCH, 1973) para encontrar cliques maximais. O algoritmo 1 recebe como *chave* de entrada (da função de *mapeamento*) o identificador do vértice e como *valor* de entrada, a lista de vizinhos do vértice. Os pares *chave/valor* da saída intermediária da fase de mapeamento são gerados segundo essa estratégia: seja I e N respectivamente o identificador do vértice e o conjunto de vizinhos, então emite-se para todo k em N o par $[k,(I,N)]$. Dessa forma, quando ocorre a fase de redução, aplica-se o algoritmo de (BRON; KERBOSCH, 1973) para encontrar cliques maximais no subgrafo.

Uma organização é feita previamente nos dados na etapa de redução. Considerando

k um pivô, todo vértice I em Valores $V[I,N]$ maior que k é inserido em uma lista chamada de *Superior*, e todo vértice I menor que k é inserido em uma lista chamada de *Inferior* e na lista *Candidato* é inserido o pivô k. Dessa forma, a lista *Superior* contém os vértices que serão analisados, a lista *Inferior* contém todos os vértices que já foram analisados e na lista *Candidato* contém os vértices candidatos a possíveis cliques maximais. Após tal particionamento, são encontradas recursivamente intersecções entra a lista *Superior* e N, bem como *Inferior* e N, onde $N = V[I] = V[v]$, sendo I e N elementos (*chave/valor* de V) emitidos na etapa de mapeamento.

Algoritmo 1: Algoritmo de *Clique* em MapReduce

```
1 inicio()
    Um grafo G = (V,E) particionado em subgrafos para processamento distribuído
    Cada unidade distribuída map/reduce processa um conjunto de vértices
2 Map(Integer id, List N):
    Foreach Vertex k ∈ N
      emite(k.id, [id,N])
3 Reduce(Integer k, List V)
    List Superior, Inferior, Candidato = null
    Foreach Vertex w ∈ V
       if w.chave > k
          Superior.add(w)
       else
          Inferior.add(w)
    Candidato.add(k)
    BronKerbosch(V, Superior, Candidato, Inferior)
4 BronKerbosch(V, List Superior, List Candidato, List Inferior)
    if (Superior.length == 0 and Inferior.length == 0)
       Clique.add(Candidato)
    while Superior.length != 0
       Integer v = Superior[0]
       List s = interseção(V[v], Superior)
       List i = interseção(V[v], Inferior)
       List c = Candidato.Clone
       c.add(v)
       BronKerbosch(V,s,c,i)
       Superior.remove(v)
       Inferior.add(v)
```

5.1.2 PageRank

O PageRank (PAGE *et al.*, 1999) é um algoritmo que mede a importância relativa de cada página *web*, de acordo com os sites que as referenciam. Esse algoritmo foi proposto pelos fundadores do Google para auxiliar no aprimoramento das pesquisas retornadas pela ferramenta de busca. Na modelagem sobre grafos, as *páginas web* são representadas por *vértices*, e as *arestas* direcionais representam *links* de uma *página* para outra.

A importância de um vértice é denominada de *rank* e está relacionada ao *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 valor do *rank* (LIN; DYER, 2010) é definido pela seguinte equação:

$$P(j) = \frac{\alpha}{|V(D)|} + (1-\alpha) \sum_{i \in L(j)} \frac{P(i)}{C(i)} \tag{5.1}$$

onde $P(j)$ 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[1]. 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 apresentado por (LIN; DYER, 2010) implementa a Equação 5.1 seguindo o modelo 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.

5.1.3 SSSP (Single Shortest Source Path)

O problema *SSSP* constitui em encontrar o menor caminho entre um vértice *fonte* e todos os outros vértices do grafo, cujo a soma dos pesos das arestas que constituem o caminho seja minimizado. Esse problema tem muitas aplicações práticas, como aqueles envolvendo o roteamento em redes de computadores e logística de transporte terrestre.

Vários algoritmos tradicionais resolvem esse problema tais como o algoritmo de *Dijkstra* e o algoritmo *Bellman-Ford*. Nesse trabalho utilizamos o Algoritmo 3 que segue o modelo MapReduce e sua implementação faz parte da validação de alguns *frameworks* aplicados a grafos (ZAHARIA *et al.*, 2010; MARTELLA *et al.*, 2015; PLIMPTON; DEVINE, 2011). Esse

[1] São excluídas redundâncias de *links*.

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)
```

algoritmo trabalha com grafo com peso em arestas. Todo vértice é inicialmente ativo e passa pela etapa de *mapeamento*, possuindo distância infinita, exceto o *fonte*. Na primeira iteração, apenas o vértice *fonte* emite mensagens no *mapeamento*, avisando seus vizinhos sobre sua distância. Na etapa de *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*.

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)
```

5.1.4 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.

Algoritmo 4: Algoritmo de Enumeração de Triângulos em MapReduce

```
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)
```

Para efetivar essas etapas, atribui-se números identificadores inteiros aos vértices e ordena 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 $e_1(1,2)$, $e_2(1,3)$, $e_3(1,4)$, $e_4(1,5)$, ..., $e_n(1,k)$, 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*. Assim, caso existam dois vértices de mesmo grau, prioriza-se aquele com menor *id*.

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 na 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*.

O framework Gust utiliza uma alteração no algoritmo de enumeração de triângulo onde a implementação consiste de três etapas. Na primeira etapa é possível contar todos os triângulos da partição local, restando a contagem global. Na segunda etapa, cada vértice procura por vizinhos e envia uma mensagem para cada mensagem recebida anteriormente. Finalmente, na terceira etapa contabiliza-se os triângulos checando as mensagens de seus vizinhos.

- **mapreduce.system.SYSTEM**
  - **mapreduce.system.linear.SYSTEM**
    - **mapreduce.system.linear.singlestage.SYSTEM**
      - mapreduce.system.linear.singlestage.iterative.SYSTEM
      - mapreduce.system.linear.singlestage.noniterative.SYSTEM
    - **mapreduce.system.linear.threestages.SYSTEM**
      - mapreduce.system.linear.threestages.noniterative.SYSTEM

Figura 25 – Hierarquia de Sistemas Abstratos

5.2 Um Arcabouço para Construção de Sistemas Gust

O arcabouço doravante proposto auxilia uma aplicação destinada a especialistas interessados em realizar determinados processamentos, suportados pela aplicação, sobre grafos grandes armazenados em certos repositórios de seu interesse. Os processamentos suportados atualmente são os que estão apresentados na Seção 5.1 e seguem o modelo MapReduce, porém

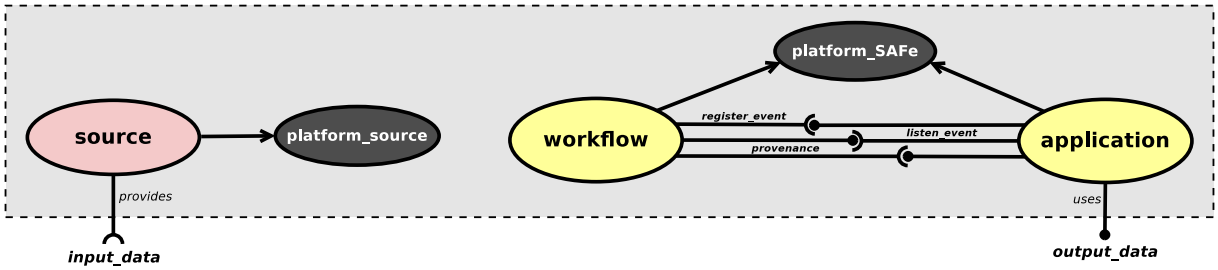

Figura 26 – Esquema Arquitetural de mapreduce.system.SYSTEM

apresentando particularidades que justificam o uso de agentes GUSTY para maior eficiência.

Usando a idéia de tratar sistemas de computação paralela como componentes, uma alternativa de projeto, mais simples, seria criar sistemas abstratos para cada problema específico, sendo o provedor responsável por escolher um sistema ou outro para resolver determinado problema, passando os argumentos de contexto necessários. Entretanto, para um mesmo problema, podem haver mais de uma arquitetura de sistema Gust para sua solução, que podem ser escolhidas alternativamente, de acordo com o contexto, pelo próprio sistema de contratos contextuais. Além disso, essa solução simples não captura similaridades arquiteturais que podem existir (e existem, de fato) entre arquiteturas de diferentes sistemas.

A alternativa de projeto proposta inicia com uma hierarquia de sistemas abstratos que capturam similaridades entre os sistemas suportados para resolver o conjunto de problemas suportados pela aplicação, apresentada na Figura 25. Os componentes na base da hierarquia correspondem a sistemas para os quais existem os sistemas concretos implementados, representando os sistemas que de fato implementam soluções para um ou mais problemas de grafos. Esses sistemas são descritos com maiores detalhes nas seções 5.3, 5.4 e 5.5.

A Figura 26 ilustra a arquitetura de mapreduce.system.SYSTEM, o sistema Gust básico. O componente **source** representa o componente de acesso repositório de dados, na plataforma virtual **platform_source**, que é escolhida de acordo com o contexto. Por sua vez, APPLICATION e WORKFLOW são aqueles componentes presentes em qualquer sistema de compu-

mapreduce.workflow.WORKFLOW
- **mapreduce.workflow.linear.WORKFLOW**
  - **mapreduce.workflow.linear.singlestage.WORKFLOW**
    - mapreduce.workflow.linear.singlestage.iterative.static.WORKFLOW
    - mapreduce.workflow.linear.singlestage.noniterative.WORKFLOW
  - **mapreduce.workflow.linear.threestages.WORKFLOW**
    - mapreduce.workflow.linear.threestages.noniterative.WORKFLOW

Figura 27 – Árvore de herança entre os componentes APPLICATION'S.

mapreduce.application.**APPLICATION**
- mapreduce.application.linear.**APPLICATION**
  - mapreduce.application.linear.singlestage.**APPLICATION**
    - mapreduce.application.linear.singlestage.iterative.APPLICATION
    - mapreduce.application.linear.singlestage.noniterative.APPLICATION
  - mapreduce.application.linear.threestages.**APPLICATION**
    - mapreduce.application.linear.threestages.noniterative.APPLICATION

Figura 28 – Árvore de herança entre os componentes WORKFLOW's.

tação paralela, como descrito na Seção 3.1, instanciados na mesma plataforma (**platform_SAFe**) onde executa a aplicação, sobre o SAFe. O grafo de entrada é recuperado do repositório *source* através do *binding* **input_data**, enquanto o resultado do processamento é entregue ao componente APPLICATION através do *binding* **output_data**.

A fim de satisfazer as particularidades de cada arquitetura, os componentes WORKFLOW e APPLICATION também encontram-se organizados segundo uma hierarquia, que espelha a hierarquia dos componentes SYSTEM, como mostrado nas Figuras 27 e 28.

Parâmetros de Contexto de mapreduce.system.SYSTEM		
Identificador	**Variável**	**Limite**
experiment_label	*L*	EXPERIMENTLABEL
system_type	*SYS*	mapreduce.systemtype.BASICSYSTEM
problem_type	*P*	PROBLEMTYPE
input_data_host	*M*	DATAHOST
input_key_type	*IK*	VERTEXBASIC
input_value_type	*IV*	DATAOBJECT
output_key_type	*OK*	VERTEXBASIC
output_value_type	*OV*	DATAOBJECT
workflow_type	*W*	mapreduce.workflow.WORKFLOW ***system_type*** = *SYS*
application_type	*A*	mapreduce.application.APPLICATION ***system_type*** = *SYS* ***output_key_type*** = *OK* ***output_value_type*** = *OV*
problem_config	*C*	mapreduce.problem_config.PROBLEMCONFIG ***system_type*** = *SYS* ***problem_type*** = *P* ***input_key_type*** = *IK* ***input_value_type*** = *IV* ***output_key_type*** = *OK* ***output_value_type*** = *OV*

Tabela 8 – Assinatura Contextual de mapreduce.system.SYSTEM

mapreduce.systemtype.**BASICSYSTEM**
- mapreduce.systemtype.**LINEARSYSTEM**
  - mapreduce.systemtype.linear.**SINGLESTAGELINEARSYSTEM**
    - mapreduce.system.linear.singlestage.ITERATIVESINGLESTAGELINEARSYSTEM
    - mapreduce.system.linear.singlestage.NONITERATIVESINGLESTAGELIEARSYSTEM
  - mapreduce.system.linear.**THREESTAGESLINEARSYSTEM**
    - mapreduce.system.linear.threestages.NONITERATIVETHREESTAGESLINEARSYSTEM

Figura 29 – Hierarquia de Qualificadores para Rotular Arquiteturas de Sistemas Gust

A Tabela 8 detalha a assinatura contextual do componente mapreduce.system.SYSTEM. A seguir, uma descrição de cada parâmetro:

- ***experiment_label***: rótulo de experimento, conforme explicado na Seção 4.2.1;
- ***system_type***: qualificador que delimita a arquitetura Gust utilizada na solução, conforme hierarquia de qualificadores apresentada na Figura 29;
- ***problem_type***: qualificador que determina o problema a ser solucionado, conforme a seguinte hierarquia de qualificadores:

PROBLEMTYPE : CLIQUE | PAGERANK | SSSP | TRIANGLEENUMERATION

- ***input_data_host***: qualificador que determina a localização do repositório **source**;
- ***input_key_type***: determina o tipo da chave de entrada;
- ***input_value_type***: determina o tipo do valor de entrada;
- ***output_key_type***: determina o tipo da chave de saída;
- ***output_value_type***: determina o tipo do valor de saída;
- ***workflow_type***: determina o tipo do componente WORKFLOW da solução;
- ***application_type***: determina o tipo do componente APPLICATION da solução;
- ***problem_config***: qualificador que determina a configuração de uma solução para o problema, ou seja, é responsável pela associação entre um problema e uma arquitetura de sistema Gust que irá solucioná-lo.

No modo mais simples de utilização por parte do provedor de aplicações, seria suficiente associar um argumento ao parâmetro de contexto ***problem_type*** a fim de que uma arquitetura Gusty seja configurada para resolvê-lo. Ou seja, seria suficiente apenas determinar o problema a ser resolvido e deixar a cargo do sistema de contratos contextuais a escolha do sistema concreto adequado. Para isso, o componente PROBLEMCONFIG, cujos subtipos apresentados na Figura 30 podem ser associados ao parâmetro ***problem_config***, desempenha um importante papel, como explicado a seguir.

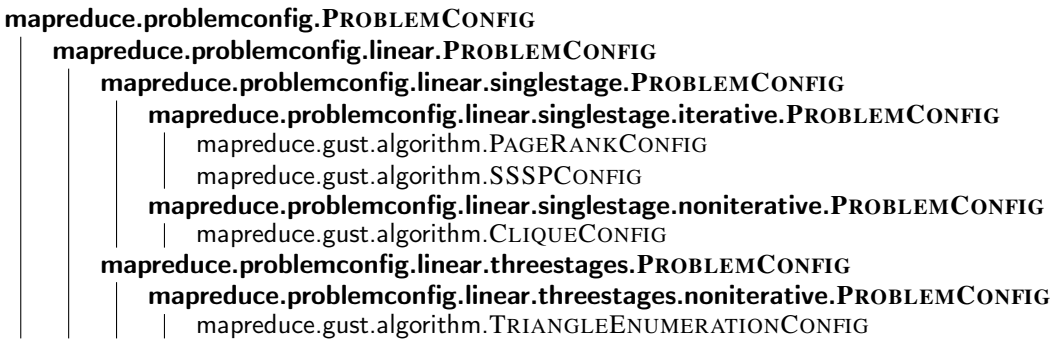

Figura 30 – Hierarquia de Qualificadores para Configurar Problemas em Sistemas Gust

Na Tabela 8, note que o tipo mapreduce.problem_config.PROBLEMCONFIG possui como parâmetros de contexto todos os parâmetros de contexto do sistema abstrato que servem para configurar o problema para ser solucionado por seu intermédio, incluindo o parâmetro ***problem_type***, ou seja, excetuando apenas ***experiment_label***, ***application_type***, ***workflow_type*** e, obviamente, o próprio ***problem_config***. Isso vale para todos os subtipos de mapreduce.system.SYSTEM, os quais introduzem seus próprios parâmetros de contexto para configurar seus agentes de computação e conectores. Em qualquer sistema abstrato, há um componente aninhado chamado **problem_config**, cujo tipo é delimitado pelo parâmetro ***problem_config***. Ao determinar um valor para o parâmetro ***problem_type***, o sistema de contratos contextuais escolhe um componente para **problem_config**, notadamente aquele associado ao problema em questão. Por exemplo, caso ***problem_type*** = PAGERANK é selecionado, para **problem_config**, o componente que implementa o tipo mapreduce.gust.algorithm.PAGERANKCONFIG. Como ilustrado pela hierarquia da Tabela 30, o valor para o seu parâmetro ***system_type*** deve ser ITERATIVESINGLESTAGELINEARSYSTEM, uma vez que é subtipo de

mapreduce.problem_config.linear.singlestage.iterative.PROBLEMCONFIG

. Portanto, PAGERANKCONFIG configura todos os parâmetros de contexto de um sistema Gusty de único estágio e iterativo, incluindo a configuração dos agentes Gusty envolvidos (parâmetros de contexto relativos a tipos de chaves e valores, funções de particionamento e de redução, etc).

Na versão atual do arcabouço Gust, assume-se o sistema de contrato contextual em sua versão mais simples (CARVALHO-JUNIOR *et al.*, 2016), onde parte-se do pressuposto que existe no máximo uma única escolha de componente para cada contrato contextual (combinação de argumentos de contexto). Logo, para cada tipo de problema atualmente suportado pelo arcabouço, existe uma única arquitetura de sistema Gust para sua solução. Entretanto, em versões estendidas do sistema de contratos contextuais que tem sido desenvolvidas pelos proponentes da

HPC Shelf, parâmetros que configuram requisitos de qualidade e de custo podem ser usados para delimitar outros tipos de requisitos. Nesse caso, para o arcabouço Gust, seria possível o usuário informar requisitos de qualidade e custo além do tipo de problema a ser solucionado, e deixar a cargo do sistema a escolha dentre várias alternativas de sistemas Gust para resolver o mesmo problema.

Outra possibilidade seria estender a assinatura contextual dos sistemas Gust para especificar algo a respeito dos grafos a serem processados, de modo a escolher dentre alternativas de arquiteturas com base no tipo de grafo a ser processado. Por exemplo, diversos autores classificam grafos quanto ao tamanho de acordo com suas próprias métricas, como (MCCOLL *et al.*, 2013), o qual definiu grafos contendo 1K (*tiny*), 32K (*small*), 1M (*medium*), e 16M (*large*) vértices com 8K, 256K, 8M, e 256M arestas não-direcionadas, respectivamente. Por sua vez, o projeto Graph500 (MURPHY *et al.*, 2010), que tem como objetivo avaliar a capacidade dos supercomputadores sob processamento intensivo de dados, ao invés de considerar apenas a velocidade de processamento, define seis escalas de tamanhos de grafos: *toy* (226 vértices; 17 GB de memória RAM), *mini* (229; 137 GB), *small* (232; 1.1 TB), *medium* (236; 17.6 TB), *large* (239; 140 TB), e *huge* (242; 1.1 PB). Diferentes caracterizações do(s) grafo(s) a serem processados poderiam portanto influenciar na escolha de uma ou outra arquitetura, ou pelo menos na quantidade de agentes Gusty empregados em uma computação.

Nas seções que se seguem, apresentamos detalhes sobre as arquiteturas de sistemas Gust concretos, os quais foram escolhidos em função dos problemas suportados pelo arcabouço Gust proposto neste trabalho.

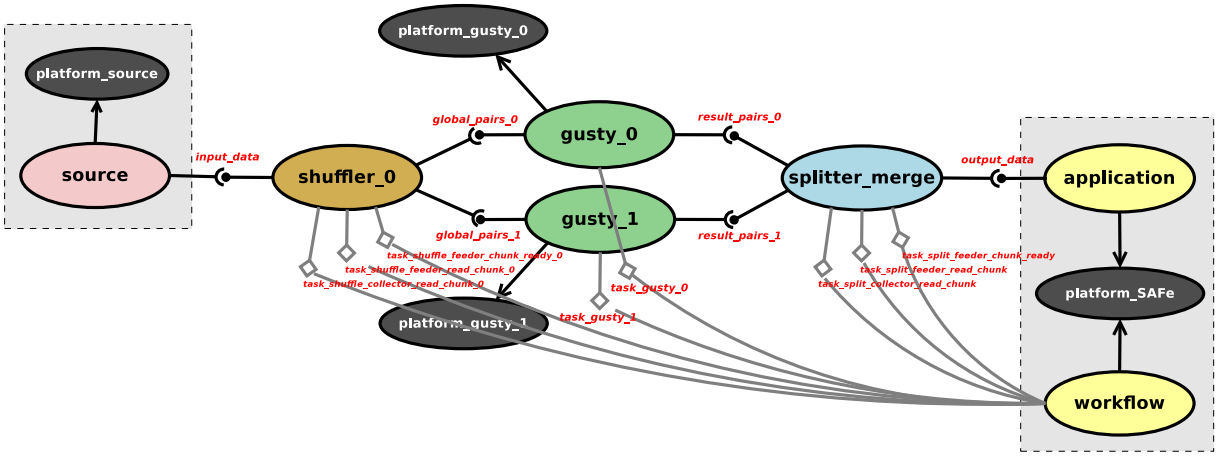

Figura 31 – Esquema Arquitetural de um *workflow* de fase única para um Sistema Map/Reduce.

5.3 *Clique*: Sistema Não-Iterativo de Único Estágio

A arquitetura mapreduce.system.linear.singlestage.noniterative.SYSTEM estende os elementos básicos encontrados na arquitetura mapreduce.system.SYSTEM, adicionando novos elementos específicos conforme podemos ver na Figura 31. A parte delimitada pelos retângulos cinza indica os elementos estendidos e que todo sistema Gust apresenta, herdado do sistema básico. Os demais elementos foram acrescentados na arquitetura.

Parâmetros de Contexto de mapreduce.system.⟨*qualifier*⟩.SYSTEM
⟨*qualifier*⟩ = linear.singlestage.noniterative

Identifier	Variable	Bound
experiment_label	*L*	EXPERIMENTLABEL
system_type	*SYS*	⟨*qualifier*⟩.systemtype.SYSTEMTYPE
problem_type	*P*	PROBLEMTYPE
input_data_host	*M*	DATAHOST
key_type	*TK*	VERTEXBASIC
value_type	*TV*	DATAOBJECT
graph_type	*G*	GRAPH
graph_input_format	*GIF*	INPUTFORMAT
gusty_function	*RF*	GUSTYFUNCTION: ***input_key_type*** *TK*; ***input_value_type*** *TV*; ***output_key_type*** *TK*; ***output_value_type*** *TV*; ***graph_type*** *G*; ***graph_input_format*** *GIF*
partition_function	*PF*	PARTITIONFUNCTION [***input_key_type*** = *TK*]
number_of_gusty_agents	*NG*	INTEGER
workflow_type	*W*	mapreduce.workflow.⟨*qualifier*⟩.WORKFLOW: ***system_type*** *SYS*
application_type	*A*	mapreduce.application.⟨*qualifier*⟩.APPLICATION: ***system_type*** *SYS*; ***output_key_type*** *TK3*; ***output_value_type*** *TV3*
problem_config	*C*	mapreduce.problemconfig⟨*qualifier*⟩.PROBLEMCONFIG: ***system_type*** *SYS*; ***problem_type*** *P*; ***key_type*** *TK*; ***value_type*** *TV*; ***graph_type*** *G*; ***graph_input_format*** *GIF*; ***gusty_function*** *RF*; ***partition_function*** *PF*

Tabela 9 – Assinatura Contextual de mapreduce.system.linear.singlestage.noniterative. SYSTEM

A porta **collect_pairs** do conector **shuffler** é ligada à porta **input_data** do componente **source**, para que o grafo de entrada seja lido e distribuído a dois agentes GUSTY (**gusty_0** e **gusty_1**) paralelos, alocados a plataformas virtuais distintas, através das ligações **global_pairs_0** e **global_pairs_1**, respectivamente. Essas ligações ligam a porta **feed_pairs** de uma das facetas *feeder* do conector **shuffler** à porta **collect_pairs** de cada agente GUSTY. Os dados produzidos por **gusty_0** e **gusty_1** são intercalados pelo conector **splitter_merge** e

mapreduce.gust.algorithm.CLIQUECONFIG	
Parâmetros de Contexto de mapreduce.linear.singlestage.noniterative.problem_config.PROBLEMCONFIG	
system_type	mapreduce.systemtype.linear.singlestage.NONITERATIVE
problem_type	mapreduce.gust.problem_type.CLIQUE
key_type	VERTEXBASIC
value_type	DATATRIANGLE
graph_type	UNDIRECTEDGRAPHV ***container*** = DATACONTAINERV ***vertex_type*** = VERTEXBASIC ***edge_type*** = EDGEBASIC [***vertex_type*** = VERTEXBASIC] ***vertex_type*** = VERTEXBASIC ***edge_type*** = EDGEBASIC [***vertex_type*** = VERTEXBASIC]
graph_input_format	INPUTFORMAT
gusty_function	br.ufc.mdcc.hpcshelf.gust.example.clique.CLIQUE
partition_function	PARTITIONFUNCTION [***input_key_type*** = VERTEXBASIC]

Tabela 10 – Contrato Contextual para uma Solução MapReduce ao Problema do Clique

entregues à aplicação através da ligação **output_data**.

O sistema mapreduce.system.linear.singlestage.noniterative.SYSTEM acrescenta os seguintes parâmetros na sua assinatura contextual, além daqueles herdados do sistema básico (mapreduce.system.SYSTEM), conforme Tabela 9:

- ***graph_type*** determina o tipo de grafo;
- ***graph_input_format*** determina o formato do grafo de entrada;
- ***gusty_function*** determina a função *gusty*, a qual opera uma única vez;
- ***partition_function*** determina a função de particionamento, o qual opera uma única vez;
- ***number_of_gusty_agents*** determina a quantidade de agentes GUST empregados na computação, para fins de controle de escalabilidade.

Note que, para geração do sistema de computação paralela ilustrado na Figura 31, o valor associado ao parâmetro ***number_of_gusty_agents*** do sistema concreto é 2.

Uma vez que o algoritmo de solução do problema *Clique* se encaixa em um sistema de único estágio e não-iterativo, uma configuração de problema (componente de um subtipo de PROBLEMCONFIG) foi desenvolvida para esse problema e chamada de mapreduce.gust.algorithm.CLIQUECONFIG, já mencionado na Figura 30 e cujo contrato contextual encontra-se especificado na Tabela 10. Portanto, para um problema de *Clique*, o provedor de aplicações deve solicitar a resolução do contrato mapreduce.system.SYSTEM[***problem_type*** = CLIQUE]. Nesse caso, o argumento para o parâmetro de contexto ***problem_config*** é valorado como PROBLEMCONFIG[***problem_type*** = CLIQUE]. Dessa forma, o único componente concreto para ***problem_config*** é aquele cujo contrato contextual está especificado na Tabela 10, onde estão definidos os demais parâmetros que alimentarão, por sua vez, os parâmetros do mapreduce.system.SYSTEM. Como o parâmetro ***system_type*** é igual a mapre-

duce.systemtype.linear.singlestage.NONITERATIVE, então o sistema Gust escolhido será de fato aquele que implementa mapreduce.system.linear.singlestage.noniterative.SYSTEM.

A orquestração dos componentes envolvidos no sistema Gust não-iterativo de único estágio é coordenada pelo componente Workflow através de um conjunto de ligações de ações, que podem ser observadas na Figura 31. No apêndice desta dissertação, estão listados os códigos de orquestração (SAFeSWL) dos estudos de caso apresentados nesse capítulo. Em particular, o Código-fonte 4 apresenta o código de orquestração para o sistema tratado nesta seção. As ativações das ações de resolução, implantação (*deployment*), instanciação e execução (*run*) dos componentes são realizadas antes do início da orquestração das ações computacionais. Obedecendo as restrições do protocolo que deve ser seguido para o ciclo de vida de cada componente, inicialmente os contratos de todos componentes são resolvidos concorrentemente, seguindo-se sua implantação, sua instanciação e sua execução. Tanto na implantação quanto na instanciação, nota-se a preocupação de garantir que as plataformas virtuais sejam implantadas/instanciadas antes dos outros componentes, os quais serão implantados/instanciados sobre elas. Ao final, após a orquestração das ações computacionais, os componentes são destruídos (*release*). Esse mesmo protocolo de controle do ciclo de vida será utilizados nos próximos estudos de caso (seções 5.4 e 5.5).

A orquestração das ações computacionais controla o processamento *pipeline* de agrupamentos de pares chave/valor chamados de *chunk*. Na iteração principal, os *chunks* (de entrada) são lidos da fonte de dados **source** e entregues à faceta *collector* do componente **shuffler** a cada ativação da ação `READ_CHUNK`, até que a ação `FINISH_CHUNK` seja ativada no componente **shuffler**. De acordo com a função de particionamento, os *chunks* são particionados (no exemplo, em duas partições) a fim de serem processados pelo time de agentes GUSTY (**gusty_0** e **gusty_1**), sempre que a ação `CHUNK_READY` de **shuffler** é ativada, quando os agentes podem ler o próximo *chunk* (ação `READ_CHUNK`) e processá-lo (ação `PERFORM`). Em uma iteração paralela, o componente **splitter_output** recebe os resultados do processamento dos agentes GUSTY sempre que estão disponíveis, também formando *chunks* (de saída), a cada ativação da ação `READ_CHUNK`, até a ativação da ação `FINISH_CHUNK`. A cada iteração, portanto, um *chunk* (de pares chave/valor) é disponibilizado para saída.

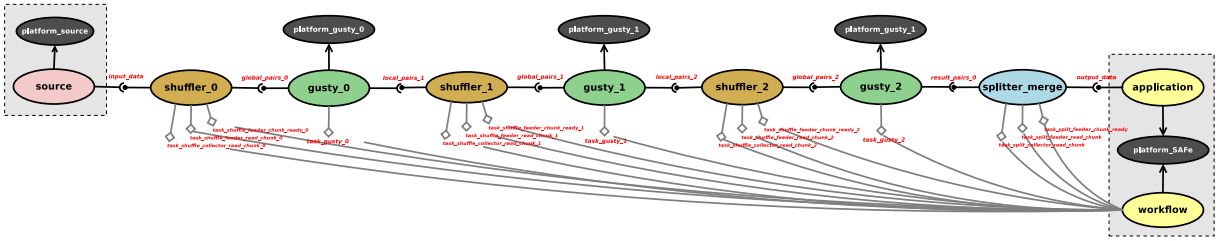

Figura 32 – Esquema Arquitetural de um *workflow* fixo de três fases para um Sistema Map/Reduce.

5.4 Enumeração de Triângulos: Sistema Não-Iterativo de Três Estágios

O sistema mapreduce.system.linear.threestages.noniterative.SYSTEM estende os elementos básicos do sistema básico com os novos elementos ilustrados na Figura 32.

De forma análoga ao explicado para o sistema descrito na seção anterior, os dados de entrada são lidos do componente **source** através de um conector SHUFFLER, no caso chamado **shuffler_0**, o qual alimenta o agente GUSTY do primeiro estágio, chamado **reducer_0**. A partir daí os pares intermediários trafegam do primeiro estágio até o terceiro estágio através de um pipeline de conectores SHUFFLER (**shuffler_1** e **shuffler_2**) e agentes Gusty (**reducer_1** e **reducer_2**), interligados por suas portas **collect_pairs** e **feed_pairs**. Finalmente, **reducer_2** entrega os dados ao conector SPLITTER, chamado **splitter_output**, o qual entrega os pares de saída à ligação que entrega os dados à aplicação. O processo é orquestrado pelo componente Workflow, por meio das portas de ações indicadas na figura.

O sistema não-iterativo de três estágios acrescenta na sua assinatura contextual, em relação ao sistema básico, os mesmos parâmetros do sistema não-iterativo de estágio único, com a diferença que cada parâmetro é acrescentado três vezes, pelo fato de existirem três estágios distintos. A Tabela 11 mostra a assinatura contextual completa. Note que, para geração do sistema de computação paralela da Figura 32, o valor unitário é associado aos parâmetros ***number_of_gusty_agents_0***, ***number_of_gusty_agents_1*** e ***number_of_gusty_agents_2*** do sistema concreto.

Uma vez que o algoritmo da solução do problema de enumeração de triângulos se encaixa no sistema de três estágios e não-iterativo, uma configuração de problema, chamada de mapreduce.gust.algorithm.TRIANGLEENUMERATIONCONFIG, foi desenvolvida, mencionada na Figura 30 e cujo contrato contextual encontra-se especificado na Tabela 12. Os parâmetros de contexto ***key_type_i*** e ***value_type_i***, para $i \in \{0,1,2,3\}$ são referentes a entrada e saída de cada etapa. Nas etapas intermediárias os tipos de saída são os tipos de entrada da próxima etapa.

Parâmetros de Contexto de ⟨*qualifier*⟩.system.SYSTEM ⟨*qualifier*⟩ = mapreduce.linear.threestages.noniterative		
Identifier	**Variable**	**Bound**
experiment_label	*L*	EXPERIMENTLABEL
system_type	*SYS*	⟨*qualifier*⟩.system_type.SYSTEMTYPE
problem_type	*P*	PROBLEMTYPE
input_data_host	*M*	DATAHOST
key_type_i, $i \in \{0,1,2,3\}$	*TKi*	VERTEXBASIC
value_type_i, $i \in \{0,1,2,3\}$	*TVi*	DATAOBJECT
graph_type_i, $i \in \{0,1,2\}$	*Gi*	GRAPH
graph_input_format_i, $i \in \{0,1,2\}$	*GIFi*	INPUTFORMAT
gusty_function_i, $i \in \{0,1,2\}$	*RFi*	GUSTYFUNCTION ***input_key_type*** = *TKi* ***input_value_type*** = *TVi* ***output_key_type*** = *TKi*+1 ***output_value_type*** = *TVi*+1 ***graph_type*** = *Gi* ***graph_input_format*** = *GIFi*
partition_function_i, $i \in \{0,1,2\}$	*PFi*	PARTITIONFUNCTION [***input_key_type*** = *TKi*]
number_of_gusty_agents_i, $i \in \{0,1,2\}$	*NG*	INTEGER
workflow_type	*W*	⟨*qualifier*⟩.workflow.WORKFLOW ***system_type*** = *SYS*
application_type	*A*	⟨*qualifier*⟩.application.APPLICATION ***system_type*** = *SYS* ***output_key_type*** = *TK3* ***output_value_type*** = *TV3*
problem_config	*C*	⟨*qualifier*⟩.problem_config.PROBLEMCONFIG ***system_type*** = *SYS* ***problem_type*** = *P* ***key_type_i***, $i \in \{0,1,2,3\}$ = *TKi* ***value_type_i***, $i \in \{0,1,2,3\}$ = *TVi* ***graph_type_i***, $i \in \{0,1,2\}$ = *Gi* ***graph_input_format_i***, $i \in \{0,1,2\}$ = *GIFi* ***gusty_function_i***, $i \in \{0,1,2\}$ = *RFi* ***partition_function_i***, $i \in \{0,1,2\}$ = *PFi*

Tabela 11 – Assinatura Contextual de mapreduce.system.linear.threestages.noniterative. SYSTEM

Por isso também, os tipos de saída da última etapa ***key_type_3*** e ***value_type_3*** são os mesmos aplicados no parâmetro ***application_type*** da assinatura contextual.

O código de orquestração para o sistema não-iterativo de três estágios está apresentado no Código-fonte 5 do Apêndice A. Como dito no estudo de caso anterior, o protocolo de ciclo de vida segue os mesmos padrões descritos anteriormente, para o sistema não-iterativo de único estágio. Com relação à orquestração das ações computacionais, trata-se de uma generalização em três estágios da orquestração do sistema não-iterativo de único estágio, onde cada um dos três estágios é tratado em uma iteração independente, alimentando os *chunks* para o próximo estágio. O último estágio (terceiro) alimenta os *chunks* para o componente **splitter_output**. Note que cada estágio possui apenas um único agente GUSTY, enquanto o único estágio do estudo de caso anterior possuía um par de agentes cujas ações computacionais precisavam ser ativadas em paralelo. Caso houvessem vários agentes em algum estágio, o mesmo protocolo

<table>
<tr><td colspan="2">mapreduce.gust.algorithm.TRIANGLEENUMERATIONCONFIG</td></tr>
<tr><td colspan="2">Parâmetros de Contexto de mapreduce.linear.threestages.noniterative.problem_config.PROBLEMCONFIG,</td></tr>
<tr><td>system_type
problem_type
key_type_i, $i \in \{0,1,2,3\}$
value_type_i, $i \in \{0,1,2,3\}$</td><td>mapreduce.systemtype.linear.threestages.NONITERATIVE
TRIANGLEENUMERATION
VERTEXBASIC
DATATRIANGLE</td></tr>
<tr><td>graph_type_i, $i \in \{0,1,2\}$</td><td>
<table>
<tr><td colspan="2">UNDIRECTEDGRAPHV</td></tr>
<tr><td>container</td><td>
<table>
<tr><td colspan="2">DATACONTAINERV</td></tr>
<tr><td>vertex_type</td><td>VERTEXBASIC</td></tr>
<tr><td>edge_type</td><td>EDGEBASIC [vertex_type = VERTEXBASIC]</td></tr>
</table>
</td></tr>
<tr><td>vertex_type</td><td>VERTEXBASIC</td></tr>
<tr><td>edge_type</td><td>EDGEBASIC [vertex_type = VERTEXBASIC]</td></tr>
</table>
</td></tr>
<tr><td>graph_input_format_i, $i \in \{0,1,2\}$
gusty_function_0
gusty_function_1
gusty_function_2
partition_function_i, $i \in \{0,1,2\}$</td><td>INPUTFORMAT
TC0
TC1
TC2
PARTITIONFUNCTION [input_key_type = VERTEXBASIC]</td></tr>
</table>

Tabela 12 – Contrato Contextual para uma Solução MapReduce ao Problema de Enumeração de Triângulos

seria usado.

5.5 *SSSP* e *PageRank*: Sistema Iterativo de Único Estágio

O sistema mapreduce.system.linear.singlestage.iterative.SYSTEM apresenta suporte a algoritmos que operam iterativamente, repetindo um único estágio de computação GUSTY. A Figura 33 mostra o processo de iteração entre dois agentes GUSTY paralelos (***gusty_0*** e ***gusty_1***) e um conector SHUFFLER (***shuffler***), responsável por governar as iterações.

A assinatura contextual do sistema iterativo de estágio único, apresentada na Tabela 13, inclui os mesmos parâmetros da assinatura da versão não-iterativa, acrescentando-se o parâmetro ***action_port_type_reduce***, usado, como dito anteriormente, para sinalizar a condição de terminação. Deve-se notar que o valor 2 é associado ao argumento ***number_of_gusty_agents***

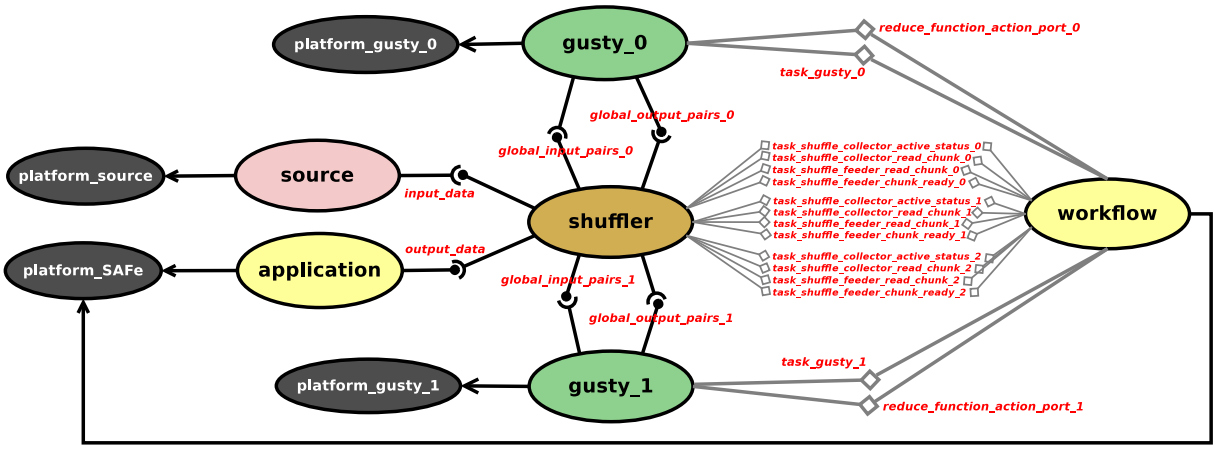

Figura 33 – Esquema Arquitetural de um *workflow* iterativo para um Sistema Map/Reduce.

Parâmetros de Contexto de ⟨*qualifier*⟩.system.SYSTEM ⟨*qualifier*⟩ = mapreduce.linear.singlestage.iterative		
Identifier	**Variable**	**Bound**
experiment_label	*L*	EXPERIMENTLABEL
system_type	*SYS*	⟨*qualifier*⟩.system_type.SYSTEMTYPE
problem_type	*P*	PROBLEMTYPE
input_data_host	*M*	DATAHOST
key_type	*TK*	VERTEXBASIC
value_type	*TV*	DATAOBJECT
graph_type	*G*	GRAPH
graph_input_format	*GIF*	INPUTFORMAT
action_port_type_reduce	*AT*	TERMINATIONFLAGTASKPORTTYPE
gusty_function	*RF*	GUSTYFUNCTION ***input_key_type*** = *TK* ***input_value_type*** = *TV* ***output_key_type*** = *TK* ***output_value_type*** = *TV* ***graph_type*** = *G* ***graph_input_format*** = *GIF* ***action_port_type*** = *AT*
partition_function	*PF*	PARTITIONFUNCTION [***input_key_type*** = *TK*]
number_of_gusty_agents	*NG*	INTEGER
workflow_type	*W*	⟨*qualifier*⟩.workflow.WORKFLOW ***system_type*** = *SYS*
application_type	*A*	⟨*qualifier*⟩.application.APPLICATION ***system_type*** = *SYS* ***output_key_type*** = *TK3* ***output_value_type*** = *TV3*
problem_config	*C*	⟨*qualifier*⟩.problem_config.PROBLEMCONFIG ***system_type*** = *SYS* ***problem_type*** = *P* ***key_type*** = *TK* ***value_type*** = *TV* ***graph_type*** = *G* ***graph_input_format*** = *GIF* ***action_port_type_reduce*** = *AT* ***gusty_function*** = *RF* ***partition_function*** = *PF*

Tabela 13 – Assinatura Contextual de mapreduce.system.linear.singlestage.iterative.SYSTEM

do sistema concreto para geração do sistema de computação paralela da Figura 33.

A arquitetura iterativa proposta nessa seção fornece suporte aos algoritmos de solução dos problemas SSSP e PAGERANK. Sobre a mesma arquitetura foi desenvolvido uma configuração para cada problema, respectivamente mapreduce.gust.algorithm.SSSPCONFIG e mapreduce.gust.algorithm.PAGERANKCONFIG, definidos nos contratos contextuais 14 e 15.

O código de orquestração para o sistema não-iterativo de três estágios está apresentado no Código-fonte 6 do Apêndice A. O código de orquestração das ações para esse estudo computacional merece ser olhada com mais cuidado, por implementar uma computação iterativa, ao contrário dos estudos de caso anteriores.

Na *primeira iteração*, os *chunks* de entrada são lidos apenas a partir da fonte de dados **source**. Para isso, as facetas *collector* de retorno dos componentes **gusty_0** e **gusty_1** são

mapreduce.gust.algorithm.SSSPCONFIG	
Parâmetros de Contexto de mapreduce.linear.singlestage.iterative.problem_config.PROBLEMCONFIG	
system_type	mapreduce.systemtype.linear.singlestage.ITERATIVE
problem_type	mapreduce.gust.problem_type.SSSP
key_type	VERTEXBASIC
value_type	DATATRIANGLE
graph_type	UNDIRECTEDGRAPHV ***container*** → DATACONTAINERV ***vertex_type*** = VERTEXBASIC ***edge_type*** = EDGEBASIC [***vertex_type*** = VERTEXBASIC] ***vertex_type*** = VERTEXBASIC ***edge_type*** = EDGEBASIC [***vertex_type*** = VERTEXBASIC]
graph_input_format	INPUTFORMAT
action_port_type_reduce	TERMINATIONFLAGPORTTYPE
gusty_function	br.ufc.mdcc.hpcshelf.gust.example.sssp.SSSP
partition_function	PARTITIONFUNCTION [***input_key_type*** = VERTEXBASIC]

Tabela 14 – Contrato Contextual para uma Solução MapReduce ao Problema do Caminho mais Curto de Fonte Única (SSSP)

desabilitadas, através da ativação das ações nas portas **task_shuffle_collector_active_status**_*i*, para $i \in \{0,1,2\}$, as quais possuem as ações `CHANGE_STATUS_BEGIN`, `CHANGE_STATUS_END`, `ACTIVE` e `INACTIVE` para esse propósito. Portanto, nessa etapa, o componente **shuffler** apenas espera receber *chunks* a partir da faceta *collector* conectada a **source** pela ligação **input_data**, particionando-os e distribuindo-os entre os agentes **gusty_0** e **gusty_1** conectados em suas facetas *feeder* por meio das ligações **global_input_pairs_0** e **global_input_pairs_1**. Após a primeira iteração, as facetas *collector* de retorno de **shuffler** são habilitadas, enquanto a faceta *collector* ligada ao componente **source** é desabilitada, quando as próximas iterações podem iniciar. Nessas iterações, até o final da execução, os *chunks* de entrada são recebidos a partir das ligações **global_output_pairs_0** e **global_output_pairs_1**, ou seja, respectivamente produzidas por **gusty_0** e **gusty_1**. A terminação é sinalizada através a ativação sincronizada,

mapreduce.gust.algorithm.PAGERANKCONFIG	
Parâmetros de Contexto de mapreduce.linear.singlestage.iterative.problem_config.PROBLEMCONFIG	
system_type	mapreduce.systemtype.linear.singlestage.ITERATIVE
problem_type	mapreduce.gust.problem_type.PAGERANK
key_type	VERTEXBASIC
value_type	DATATRIANGLE
graph_type	UNDIRECTEDGRAPHV ***container*** → DATACONTAINERV ***vertex_type*** = VERTEXBASIC ***edge_type*** = EDGEBASIC [***vertex_type*** = VERTEXBASIC] ***vertex_type*** = VERTEXBASIC ***edge_type*** = EDGEBASIC [***vertex_type*** = VERTEXBASIC]
graph_input_format	INPUTFORMAT
action_port_type_reduce	TERMINATIONFLAGPORTTYPE
gusty_function	br.ufc.mdcc.hpcshelf.gust.example.pgr.PGRANK
partition_function	PARTITIONFUNCTION [***input_key_type*** = VERTEXBASIC]

Tabela 15 – Contrato Contextual para uma Solução MapReduce ao Problema de Ranqueamento de Páginas *Web*

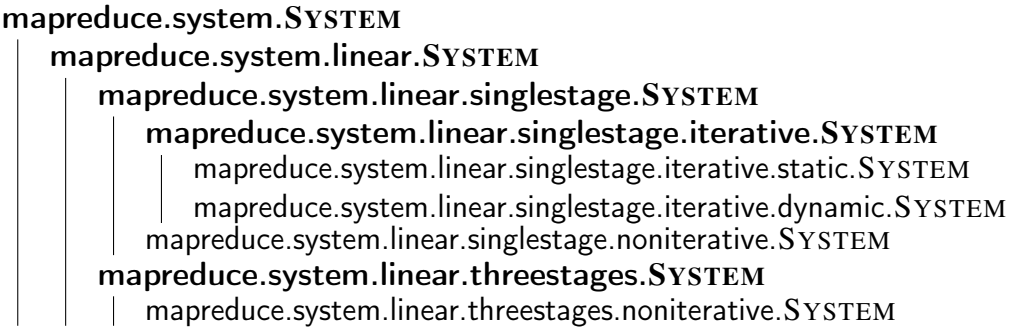

Figura 34 – Hierarquia de Sistemas Abstratos (Projeto Alternativo)

por parte de **gusty_0** e **gusty_1**, em suas funções *gusty*, das ações `TERMINATE` nas ligações **reduce_function_action_port_0** e **reduce_function_action_port_1**. Alternativamente, isto é, no caso de ser necessária uma nova iteração, a ação ativada nessas ligações é `CONTINUE`.

5.5.1 Sistemas Iterativos Dinâmicos e Estáticos

As soluções dos problemas SSSP e PAGERANK, embora utilizem o mesmo sistema Gust iterativo de estágio único, possuem características diferentes. Enquanto a primeira possui quantidade fixa de iterações, independente do grafo, a última tem a quantidade de iterações dependente do grafo de entrada. Um projeto alternativo para os sistemas iterativos de estágio único seria a diferenciação entre sistemas estáticos e dinâmicos, como apresentado na Figura 34.

No sistema estático, algumas simplificações/otimizações seriam possíveis. Por exemplo, poderia ser incluído, em seu contrato contextual, um parâmetro que determina o número fixo de iterações, de modo que seria possível gerar um sistema Gust linear de *N* estágios para implementar uma execução de *N* iterações, especialmente quando o *N* é pequeno, ou simplesmente gerar um código de orquestração linear para executar o sistemas iterativo. Em ambos os casos, não haveria necessidade de a função *gusty* fazer uso da porta **reduce_function_action_port** para determinar o término da iteração, o que ficaria a cargo do código de orquestração.

5.6 Extensão do Framework

A possibilidade de extensão de um *framework* é uma de suas principais propriedades, de forma que possa suportar possíveis customizações para resolver os mais variados problemas, cada vez mais específicos. A seguir é discutido como *framework* proposto pode ser estendido, tanto para resolver outros problemas de processamento em grafos quanto para suportar outras arquiteturas de sistemas Gust.

Um cenário mais simples é a inclusão do suporte à solução de um novo problema de processamento em grafos que utilize um dentre os sistemas concretos já suportados. Com isso, são aproveitados todos os componentes abstratos e concretos já suportados pelo *framework*, sendo necessário apenas acrescentar acrescentar:

- um componente qualificador para identificar o novo problema, herdado de PROBLEMTYPE;
- um componente (concreto) de configuração do problema (PROBLEMCONFIG), correspondente ao sistema a ser utilizado (Figura 30), associando-se o rótulo do tipo de sistema (Figura 29) ao parâmetro de contexto ***system_type***;
- os componentes de customização necessários para configurar o sistema, tais como a função Gust de cada agente GUSTY empregado no sistema, funções de particionamento específicas de cada conector e estruturas de dados para chaves e valores.

A configuração do sistema se dá de acordo com os parâmetros do contrato contextual do novo componente de configuração de problema.

Caso um sistema abstrato que possa ser configurado para implementar a solução do problema não exista, é preciso inseri-lo na hierarquia de sistemas (Figura 25) e implementá-lo, o que envolve também a implementação dos componentes Workflow e Application e de configuração de problema associados (figuras 27, 28 e 30), além dos sistemas abstrato e concreto necessários.

5.7 Avaliação

Como resultado desses estudos de caso, sistemas Gust simples e iterativos foram propostos e organizados de modo a favorecer o reuso em nível de proveniência prospectiva. Como exemplo de reuso, apesar da diferença da natureza dos problemas, *SSSP* e *PageRank* tiveram soluções baseadas no mesmo sistema Gust (iterativo de estágio único). De outra maneira, cada solução, independentemente, deveria especificar seus componentes e ligações, a fim de compor sistemas distintos. Dessa forma, o estudo de caso mostra que o uso de sistemas de computação paralela como componentes (proveniência prospectiva) contribui para a reusabilidade e manutenibilidade de soluções. O projeto alternativo levando em consideração a diferenciação entre sistemas iterativos estáticos e dinâmicas (Seção 5.5.1), além da possibilidades da inserção de novos problemas e sistemas, demonstram também a extensibilidade do *framework* proposto.

As informações de proveniência prospectiva e retrospectiva em diferentes níveis, são tratados sob a mesma perspectiva de componente sistema, fazendo com que essas informações

sejam acopladas ao modelo de componentes. No momento em que o provedor de aplicações adota o sistema abstrato básico mapreduce.system.SYSTEM para criar um sistema de computação paralela que solucione um problema de processamento em grafos, faz uso de recursos de proveniência prospectiva. A proveniência retrospectiva surge desde sistemas concretos escolhidos para sintetizar dinamicamente um sistema de computação paralela até a orquestração propriamente dita. Os dados de proveniência retrospectiva são ainda categorizados em 4 níveis, conforme especificado na Seção 4.2.

Também é importante notar a associação das categorias propostas com as fases do ciclo de vida de *workflows* em SGWfSs (*Composição*, *Mapeamento*, *Execução* e *Análise*), conforme discutido na Seção 2.1. A fase de *Composição* está relacionada ao sistema abstrato e aos sistemas concretos de níveis 0 e 1, onde a estrutura do fluxo do experimento ainda está sendo concebido. A fase de *Mapeamento* é realizado durante a execução, visto que os componentes são escolhidos dinamicamente durante o processamento. Dessa forma, essa fase está relacionada com o nível 2 do sistema concreto, atuando tanto na ativação da ação `resolve` como na ação `instantiate`. Finalmente, a fase de *Execução* está relacionada com o nível 3. Entretanto, a fase de *Análise* não possui uma correspondência direta na proposta, mas o acesso a *workflows* ocorre através dos provedores de aplicação, os quais podem definir aplicações específicas para elaboração de análise e consultas sobre os *workflows* registrados, podendo prover diferentes formas de acesso, seja através de API, recurso gráfico com GUI, linguagem de consulta, etc.

6 CONCLUSÕES E TRABALHOS FUTUROS

Problemas científicos fazem cada vez mais uso de computações intensivas sobre dados em larga escala, empregando plataformas de computação distribuídas. À medida que os experimentos aumentam em escala e complexidade, a capacidade de reprodução e entendimento dos experimentos adquirem maior dificuldade. Nesse sentido, a proveniência de dados cumpre um papel fundamental, buscando melhorias nos processos de gerência das informações.

A intenção é que os cientistas se beneficiem dos dados produzidos ao longo do ciclo de vida dos experimentos, auxiliando-os no reuso, manutenção e evolução destes, representados através de *workflows* científicos. Com o acesso às informações de proveniência, é possível obter um melhor entendimento do comportamento dos experimentos, bem como facilitar o diagnóstico de problemas, interpretar resultados e identificar regiões críticas na execução.

No contexto do projeto de desenvolvimento da plataforma HPC Shelf, o presente trabalho propõe formas de obter maior controle e gerência na execução dos *workflows*, dispondo de indicadores sobre os níveis de informação. Os dados de proveniência prospectiva contribuem para usufruto e reuso de sistemas de computação paralela. Os dados de proveniência retrospectiva da execução de *workflows*, por sua vez, são registrados para viabilizar a reprodução de experimentos. O modelo de proveniência proposto neste trabalho para a HPC Shelf é inovador no sentido que utiliza um modelo de dados comum para representar tanto o *workflow*, quanto sua execução em si, além de fazendo uso da expressividade do sistema de tipos da HPC Shelf (HTS) para exprimir naturalmente a noção de tipos e subtipos de *workflows*.

No mecanismo de proveniência proposto neste trabalho, sistemas de computação paralela oriundos do *framework* Gust podem ser tratados como componentes registrados no catálogo do Core, através da da linguagem SAFeSWL, para descrever as soluções. Os sistemas registrados podem ser então reutilizados dentro de uma mesma aplicação ou entre aplicações diferentes.

Uma variedade de soluções que possuam os mesmos requisitos arquiteturais podem ser organizadas utilizando sistemas abstratos e concretos. Soluções de problemas que possuem sistemas que apresentam certos padrões arquiteturais podem se beneficiar do armazenamento de dados de proveniência prospectiva, utilizando sistemas já registrados ou estendendo sistemas pré-existentes para atender a requisitos mais específicos.

Através do recurso de proveniência retrospectiva, as ações realizadas pelos componentes são registradas e torna-se possível tratar a reprodutibilidade de execuções de *workflows*.

No entanto, outras propósitos podem ser explorados por intermédio do uso da informação de proveniência.

Nessa dissertação também apresentamos um arcabouço para construção de sistemas Gust operando sobre um conjunto de problemas da teoria dos grafos. Uma prova de conceito é desenvolvida visando mostrar a estratégia de tratar os sistemas de computação paralela em componentes de sistema. O estudo de caso demonstra os benefícios que o suporte ao registro de dados de proveniência dos *workflows* contribuem para o reuso, manutenibilidade e facilidade na construção de soluções.

O acesso aos registros de proveniência na HPC Shelf ocorre por meio dos provedores de aplicação, os quais podem construir aplicações com requisitos específicos de análise, levando em conta as tecnologias utilizadas no armazenamento e modelo de dados adotado. O *workflow* no formato SAFeSWL que utilizamos poderia ser, portanto, apresentado em diferentes formatos de *workflows*, como os utilizados pelos modelos de proveniência OPM ou PROV, verificando as correspondências entre os elementos adotados. Dessa forma, podemos obter o suporte a compartilhamento de *workflows* entre usuários de diferentes SGWfCs.

Neste trabalho, assumimos algumas características que podem supor limitações na proposta mas que poderiam ser trabalhadas em pesquisas futuras. Os repositórios de dados utilizados pelos *workflows* não permitem modificações (somente-leitura) e os algoritmos das soluções devem ser determinísticos, produzindo assim uma mesma saída para uma mesma entrada. No processo de registro dos dados de proveniência são gravadas todas as ações da execução do componente. Entretanto, seria possível restringir o mecanismo de modo a termos uma gravação parcial, somente em alguns trechos da execução de interesse do usuário. Pretende-se oferecer esse suporte em futuras versões do mecanismo.

Finalmente, trabalhos subsequentes devem fornecer mais estudos de caso de, maior escala e sistematização, a fim de validação do mecanismo de proveniência básico proposto neste trabalho, que se somem aos estudos de prova de conceito apresentados nesta dissertação.

6.1 Trabalhos Futuros

A análise de dados de proveniência constitui um desafio diante dos diferentes contextos envolvidos e integra um importante passo na análise de *workflows* e resultados de uma execução. Ferramentas e serviços de consultas devem ser bem avaliadas para manipulação simples e eficaz de quem possui conhecimento especialista. Assim, constitui como estudo

posterior, um *framework*/API de consulta às informações de proveniência tanto prospectiva quanto retrospectiva em seus variados níveis. Tal proposta de estudo auxiliaria a construção de aplicações específicas para elaboração de análise dos dados de proveniência.

Trabalhos futuros podem considerar que as bases de dados são passíveis de evolução nos dados ou modificações na estrutura. Dessa forma, é necessário o registro de informações que possam identificar as bases de dados no momento da execução, tendo em vista que para viabilizar uma reexecução de um *workflow* é preciso que a base de dados esteja consistente com seu estado anterior.

Dados úteis às aplicações da HPC Shelf são persistidos atualmente em componentes da espécie *fonte de dados*. Os dados intermediários produzidos pelos componentes durante o processamento do *workflow*, entretanto, não são armazenados. Por exemplo, os pares *chave/valor* processados nas aplicações MapReduce e resultados parciais de componentes Gusty. Dessa forma, seria interessante o registro de dados intermediários para, por exemplo, otimizar uma execução ou implementar um mecanismo de segurança à falhas. Os próprios repositórios poderiam também registrar informações dos componentes que à acessam e operam sobre suas bases de dados. Esse tipo de informação pode ser útil no avanço do desenvolvimento da HPC Shelf. Com essa estratégia. um grande volume de dados seria produzido e seria relevante incluir nas partes interessadas da HPC Shelf um *gerente de dados*, encarregado de manter e gerenciar esses dados.

REFERÊNCIAS

BARGA, Roger; GANNON, Dennis. Scientific versus Business Workflows. In: **Workflows for e-Science: Scientific Workflows for Grids**. Edição: Ian J. Taylor. London: Springer London, 2007. P. 9–16. ISBN 978-1-84628-757-2. DOI: 10.1007/978-1-84628-757-2_2. Disponível em: https://doi.org/10.1007/978-1-84628-757-2_2.

BITON, Olivier et al. Querying and managing provenance through user views in scientific workflows. In: IEEE. **2008 IEEE 24th International Conference on Data Engineering**. [S.l.: s.n.], 2008. P. 1072–1081.

BRON, Coen; KERBOSCH, Joep. Algorithm 457: finding all cliques of an undirected graph. **Communications of the ACM**, ACM, v. 16, n. 9, p. 575–577, 1973.

CARVALHO-JUNIOR, F. H.; 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. DOI: 10.1016/j.jpdc.2012.12.007. Disponível em: http://www.sciencedirect.com/science/article/pii/S0743731512002882.

CARVALHO-JUNIOR, F. H.; REZENDE, C. A. et al. 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. DOI: http://dx.doi.org/10.1016/j.scico.2016.07.005.

CARVALHO-JUNIOR, F. H.; SILVA, J. C.; DANTAS, A. B. O. A Scientific Workflow Management System for Orchestration of Parallel Components in a Cloud of Large-Scale Parallel Processing Services. **Science of Computer Programming**, To appear in printed version (on-line), p. 60, 2018. ISSN 0167-6423. DOI: http://dx.doi.org/10.1016/j.scico.2018.04.004.

CARVALHO-JUNIOR, Francisco Heron; LINS, Rafael Dueire. An Institutional Theory for #-Components. **Electronic Notes in Theoretical Computer Science**, v. 195, p. 113–132, 2008. Proceedings of the Brazilian Symposium on Formal Methods (SBMF 2006). ISSN 1571-0661. DOI: http://dx.doi.org/10.1016/j.entcs.2007.08.029. Disponível em: http://www.sciencedirect.com/science/article/pii/S1571066108000157.

CHALLENGER, Provenance. **Provenance Challenger Wiki**. [S.l.: s.n.], 2017. Disponível em: http://twiki.ipaw.info/bin/view/Challenge/.

CLIFFORD, Ben et al. Tracking provenance in a virtual data grid. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 20, n. 5, p. 565–575, 2008.

COHEN, Jonathan. Graph twiddling in a mapreduce world. **Computing in Science & Engineering**, IEEE, v. 11, n. 4, p. 29–41, 2009.

CRUZ, Sérgio Manuel Serra da; CAMPOS, Maria Luiza M; MATTOSO, Marta. Towards a taxonomy of provenance in scientific workflow management systems. In: IEEE. **2009 Congress on Services-I**. [S.l.: s.n.], 2009. P. 259–266.

DANTAS, A. B. O. **Certificação de componentes em uma plataforma de nuvens computacionais para serviços de computação de alto desempenho**. 2017. 185 f. Tese (Doutorado em Ciência da Computação) - Universidade Federal do Ceará, Fortaleza, 2017.

DAVIDSON, Susan B; BOULAKIA, Sarah Cohen et al. Provenance in Scientific Workflow Systems. **IEEE Data Eng. Bull**., v. 30, n. 4, p. 44–50, 2007.

DEAN, Jeffrey; GHEMAWAT, Sanjay. 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. DOI: 10.1145/1327452.1327492. Disponível em: http://doi.acm.org/10.1145/1327452.1327492.

DEELMAN, Ewa; GANNON, Dennis et al. Workflows and e-Science: An overview of workflow system features and capabilities. **Future Generation Computer Systems**, Elsevier, v. 25, n. 5, p. 528–540, 2009.

FREIRE, Juliana; KOOP, David et al. Provenance for Computational Tasks: A Survey. Computing in Science and Engg., **IEEE Educational Activities Department**, Piscataway, NJ, USA, v. 10, n. 3, p. 11–21, mai. 2008. ISSN 1521-9615. DOI: 10.1109/MCSE.2008.79. Disponível em: http://dx.doi.org/10.1109/MCSE.2008.79.

FREIRE, Juliana; SILVA, Cláudio T. et al. Managing Rapidly-Evolving Scientific Workflows. In: **Provenance and Annotation of Data: International Provenance and Annotation Workshop**, IPAW 2006, Chicago, IL, USA, May 3-5, 2006, Revised Selected Papers. Edição: Luc Moreau e Ian Foster. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. P. 10–18. ISBN 978-3-540-46303-0. DOI: 10.1007/11890850_2. Disponível em: http://dx.doi.org/10.1007/11890850_2.

GCIS. **Global Change Information System**. [S.l.: s.n.], 2016. Disponível em: http://data.globalchange.gov/.

KEPLER, Project. **Getting Started with Kepler Provenance 2.5**. [S.l.: s.n.], 2015. Disponível em: https://code.kepler-project.org/code/kepler/trunk/modules/provenance/docs/provenance.pdf.

KEPLER, Project. **Getting Started with Kepler Workflow Run Manager 2.5**. [S.l.: s.n.], 2015. Disponível em: https://code.kepler-project.org/code/kepler/trunk/modules/workflow-run-manager/docs/workflow-run-manager.pdf.

KEPLER, Project. **Kepler User Manual 2.5**. [S.l.: s.n.], 2015. Disponível em: https://code.kepler-project.org/code/kepler-docs/trunk/outreach/documentation/shipping/2.5/UserManual.pdf.

LIN, Jimmy; DYER, Chris. Data-intensive text processing with MapReduce. **Synthesis Lectures on Human Language Technologies**, Morgan & Claypool Publishers, v. 3, n. 1, p. 1–177, 2010.

LUDÄSCHER, Bertram et al. Scientific workflow management and the Kepler system. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 18, n. 10, p. 1039–1065, 2006.

MARTELLA, Claudio et al. **Practical graph analytics with apache giraph**. [S.l.: s.n.]: Springer, 2015.

MATTOSO, Marta; WERNER, Claudia; TRAVASSOS, Guilherme Horta; BRAGANHOLO, Vanessa; OGASAWARA, Eduardo et al. Towards supporting the life cycle of large scale scientific experiments. **International Journal of Business Process Integration and Management**, Inderscience Publishers, v. 5, n. 1, p. 79–92, 2010.

MATTOSO, Marta; WERNER, Cláudia; TRAVASSOS, G; BRAGANHOLO, Vanessa; MURTA, Leonardo. Gerenciando experimentos científicos em larga escala. **SBC-SEMISH**, v. 8, p. 121–135, 2008.

MCCOLL, Robert et al. A Brief Study of Open Source Graph Databases. **CoRR**, abs/1309.2675, 2013. arXiv: 1309.2675. Disponível em: http://arxiv.org/abs/1309.2675.

MCPHILLIPS, Timothy; BOWERS, Shawn; LUDÄSCHER, Bertram. Collection-Oriented Scientific Workflows for Integrating and Analyzing Biological Data. In: **Data Integration in the Life Sciences: Third International Workshop**, DILS 2006, Hinxton, UK, July 20-22, 2006. Proceedings. Edição: Ulf Leser, Felix Naumann e Barbara Eckman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. P. 248–263. ISBN 22978-3-540-36595-2. DOI: 10.1007/11799511_23. Disponível em: http://dx.doi.org/10.1007/11799511_23.

MOREAU, Luc et al. The open provenance model core specification (v1. 1). **Future generation computer systems**, Elsevier, v. 27, n. 6, p. 743–756, 2011.

MURPHY, Richard C et al. Introducing the Graph 500, 2010.

NAVEH, B. et al. **JGraphT**. [S.l.: s.n.], 2003. Disponível em: http://jgrapht.org. Acesso em: 17 mai. 2017.

OINN, Tom et al. Taverna/myGrid: Aligning a Workflow System with the Life Sciences Community. In: **Workflows for e-Science: Scientific Workflows for Grids**. Edição: Ian J. Taylor. London: Springer London, 2007. P. 300–319. ISBN 978-1-84628-757-2. DOI: 10.1007/978-1-84628-757-2_19. Disponível em: http://dx.doi.org/10.1007/978-1-84628-757-2_19.

PAGE, Lawrence et al. **The PageRank citation ranking: Bringing order to the web**. [S.l.: s.n.], 1999.

PLIMPTON, Steven J; DEVINE, Karen D. MapReduce in MPI for large-scale graph algorithms. **Parallel Computing**, Elsevier, v. 37, n. 9, p. 610–632, 2011.

PROV-OVERVIEW. **An Overview of the PROV Family of Documents**. [S.l.: s.n.], 2016. Disponível em: http://www.w3.org/TR/prov-overview/. Acesso em: 27 set. 2016.

RAGAN, Eric D et al. Characterizing provenance in visualization and data analysis: an organizational framework of provenance types and purposes. **IEEE transactions on visualization and computer graphics**, IEEE, v. 22, n. 1, p. 31–40, 2016.

REZENDE C. A. **Um arcabouço baseado em componentes para computação paralela de larga escala sobre grafos**. 2017. 175 f. Tese (Doutorado em Ciência da Computação) - Universidade Federal do Ceará, Fortaleza, 2017.

REZENDE C. A.; CARVALHO-JUNIOR, Francisco Heron. MapReduce with Components for Processing Big Graphs. **Anais do XIX Simpósio de Sistemas Computacionais de Alto Desempenho (WSCAD'2018)**, XIX Simpósio de Sistemas Computacionais de Alto Desempenho (WSCAD'2018), São Paulo, 2018.

ROURE, David De; GOBLE, Carole; STEVENS, Robert. The design and realisation of the Virtual Research Environment for social sharing of workflows. **Future Generation Computer Systems**, v. 25, n. 5, p. 561–567, 2009. ISSN 0167-739X. DOI: http://dx.doi.org/10.1016/j.future.2008.06.010. Disponível em: http: //www.sciencedirect.com/science/article/pii/S0167739X08000939.

SCHEIDEGGER, Carlos et al. Tackling the Provenance Challenge One Layer at a Time. **Concurr. Comput. : Pract. Exper.**, John Wiley e Sons Ltd., Chichester, UK, v. 20, n. 5, p. 473–483, abr. 2008. ISSN 1532-0626. DOI: 10.1002/cpe.v20:5. Disponível em: http://dx.doi.org/10.1002/cpe.v20:5.

SILVA, J. C. **Um arcabouço para a construção de aplicações baseadas em componentes sobre uma plataforma de nuvem computacional para serviços de computação de alto desempenho**. 2016. 189 f. Tese (Doutorado em ciência da computação)- Universidade Federal do Ceará, Fortaleza-CE, 2016.

SIMMHAN, Yogesh L et al. Performance evaluation of the karma provenance framework for scientific workflows. In: SPRINGER. **INTERNATIONAL Provenance and Annotation Workshop**. [S.l.: s.n.], 2006. P. 222–236.

SVENDSEN, Michael; MUKHERJEE, Arko Provo; TIRTHAPURA, Srikanta. Mining Maximal Cliques from a Large Graph Using MapReduce. **J. Parallel Distrib. Comput.**, Academic Press, Inc., Orlando, FL, USA, v. 79, n. 100, p. 104–114, mai. 2015. ISSN 0743-7315. DOI: 10.1016/j.jpdc.2014.08.011. Disponível em: http://dx.doi.org/10.1016/j.jpdc.2014.08.011.

TAVERNA, Project. **Provenance management**. [S.l.: s.n.], 2017. Disponível em: https://taverna.incubator.apache.org/documentation/provenance/.

USGCRP. **U.S. Global Change Research Program**. [S.l.: s.n.], 2016. Disponível em: http://www.globalchange.gov/.

VISTRAILS. **VisTrails Documentation**. [S.l.: s.n.], 2016. Disponível em: https://www.vistrails.org/usersguide/dev/html/VisTrails.pdf.

ZAHARIA, Matei et al. Spark: Cluster computing with working sets. **HotCloud**, v. 10, n. 10-10, p. 95, 2010.

APÊNDICE A – CÓDIGO DE ORQUESTRAÇÃO SAFESWL PARA OS SISTEMAS GUST DOS ESTUDOS DE CASO

Este apêndice apresenta os códigos SAFeSWL usados para a orquestração dos sistemas Gust utilizados no estudo de caso do Capítulo 5. Notadamente, esse código encontra-se associado ao componente Workflow associado a cada sistema concreto (nível 0).

O Código-fonte 4 apresenta a orquestração do sistema não-iterativo de estágio único, usado na implementação do problema *Clique* e explicado na Seção 5.3. Por sua vez, o Código-fonte 5 apresenta a orquestração do sistema não-iterativo de três estágios, usado na implementação da enumeração de triângulos e explicado na Seção 5.4. Finalmente, o Código-fonte 6 apresenta a orquestração do sistema iterativo de estágio único, usado na implementação dos problemas *SSSP* e *PageRank* e explicado na Seção 5.5.

Código-fonte 4 – Orquestração para mapreduce.workflow.linear.singlestage.noniterative. WORKFLOW

```
<workflow>

  <sequence>

      <parallel>

          <!-- RESOLVE PLATFORMS -->
          <invoke id_port="platform_source"       action="resolve"/>
          <invoke id_port="platform_gusty_0"      action="resolve"/>
          <invoke id_port="platform_gusty_1"      action="resolve"/>

          <!-- RESOLVE SERVICE BINDINGS -->
          <invoke id_port="input_data"      action="resolve"/>
          <invoke id_port="global_pairs_0" action="resolve"/>
          <invoke id_port="global_pairs_1" action="resolve"/>
          <invoke id_port="result_pairs_0" action="resolve"/>
          <invoke id_port="result_pairs_1" action="resolve"/>
          <invoke id_port="output_data"     action="resolve"/>

          <!-- RESOLVE ACTION BINDINGS -->
          <invoke id_port="task_shuffle_collector_read_chunk" action="resolve"/>
          <invoke id_port="task_shuffle_feeder_read_chunk_0"  action="resolve"/>
          <invoke id_port="task_shuffle_feeder_chunk_ready_0" action="resolve"/>
          <invoke id_port="task_shuffle_feeder_read_chunk_1"  action="resolve"/>
          <invoke id_port="task_shuffle_feeder_chunk_ready_1" action="resolve"/>
          <invoke id_port="task_gusty"                        action="resolve"/>
          <invoke id_port="task_split_collector_read_chunk_0"   action="resolve"/>
          <invoke id_port="task_split_collector_read_chunk_1"   action="resolve"/>
          <invoke id_port="task_split_feeder_read_chunk"      action="resolve"/>
          <invoke id_port="task_split_feeder_chunk_ready"     action="resolve"/>

          <!-- RESOLVE DATA SOURCES, COMPUTATIONS AND CONNECTORS CONTRACTS -->
          <invoke id_port="source"           action="resolve"/>
          <invoke id_port="shuffler"         action="resolve"/>
          <invoke id_port="gusty_0"          action="resolve"/>
          <invoke id_port="gusty_1"          action="resolve"/>
          <invoke id_port="splitter_output" action="resolve"/>

      </parallel>

      <sequence>

          <parallel>

              <!-- DEPLOY PLATFORMS -->
              <invoke id_port="platform_source"       action="deploy"/>
              <invoke id_port="platform_gusty_0"      action="deploy"/>
              <invoke id_port="platform_gusty_1"      action="deploy"/>

          </parallel>

          <parallel>

```

```
54                <!-- DEPLOY SERVICE BINDINGS -->
55                <invoke id_port="input_data"      action="deploy"/>
56                <invoke id_port="global_pairs_0" action="deploy"/>
57                <invoke id_port="global_pairs_1" action="deploy"/>
58                <invoke id_port="result_pairs_0" action="deploy"/>
59                <invoke id_port="result_pairs_1" action="deploy"/>
60                <invoke id_port="output_data"     action="deploy"/>
61
62                <!-- DEPLOY ACTION BINDINGS -->
63                <invoke id_port="task_shuffle_collector_read_chunk" action="deploy"/>
64                <invoke id_port="task_shuffle_feeder_read_chunk_0"  action="deploy"/>
65                <invoke id_port="task_shuffle_feeder_chunk_ready_1" action="deploy"/>
66                <invoke id_port="task_shuffle_feeder_read_chunk_0"  action="deploy"/>
67                <invoke id_port="task_shuffle_feeder_chunk_ready_1" action="deploy"/>
68                <invoke id_port="task_gusty"                        action="deploy"/>
69                <invoke id_port="task_split_collector_read_chunk_0"   action="deploy"/>
70                <invoke id_port="task_split_collector_read_chunk_1"   action="deploy"/>
71                <invoke id_port="task_split_feeder_read_chunk"       action="deploy"/>
72                <invoke id_port="task_split_feeder_chunk_ready"      action="deploy"/>
73
74                <!-- DEPLOY DATA SOURCES, COMPUTATIONS AND CONNECTORS -->
75                <invoke id_port="source"           action="deploy"/>
76                <invoke id_port="shuffler"         action="deploy"/>
77                <invoke id_port="gusty_0"          action="deploy"/>
78                <invoke id_port="gusty_1"          action="deploy"/>
79                <invoke id_port="splitter_output" action="deploy"/>
80
81            </parallel>
82
83        </sequence>
84
85        <sequence>
86
87            <parallel>
88
89                <!-- INSTANTIATE PLATFORMS -->
90                <invoke id_port="platform_source"       action="instantiate"/>
91                <invoke id_port="platform_gusty_0"      action="instantiate"/>
92                <invoke id_port="platform_gusty_1"      action="instantiate"/>
93
94            </parallel>
95
96            <parallel>
97
98                <!-- INSTANTIATE SERVICE BINDINGS -->
99                <invoke id_port="input_data"      action="instantiate"/>
100               <invoke id_port="global_pairs_0" action="instantiate"/>
101               <invoke id_port="global_pairs_1" action="instantiate"/>
102               <invoke id_port="result_pairs_0" action="instantiate"/>
103               <invoke id_port="result_pairs_1" action="instantiate"/>
104               <invoke id_port="output_data"     action="instantiate"/>
105
106               <!-- INSTANTIATE ACTION BINDINGS -->
107               <invoke id_port="task_shuffle_collector_read_chunk" action="instantiate"/>
108               <invoke id_port="task_shuffle_feeder_read_chunk_0"  action="instantiate"/>
109               <invoke id_port="task_shuffle_feeder_chunk_ready_1" action="instantiate"/>
110               <invoke id_port="task_shuffle_feeder_read_chunk_0"  action="instantiate"/>
111               <invoke id_port="task_shuffle_feeder_chunk_ready_1" action="instantiate"/>
112               <invoke id_port="task_gusty"                        action="instantiate"/>
113               <invoke id_port="task_split_collector_read_chunk_0" action="instantiate"/>
114               <invoke id_port="task_split_collector_read_chunk_1" action="instantiate"/>
115               <invoke id_port="task_split_feeder_read_chunk"      action="instantiate"/>
116               <invoke id_port="task_split_feeder_chunk_ready"     action="instantiate"/>
117
118               <!-- INSTSANTIATE DATA SOURCES, COMPUTATIONS AND CONNECTORS -->
119               <invoke id_port="source"           action="instantiate"/>
120               <invoke id_port="shuffler"         action="instantiate"/>
121               <invoke id_port="gusty_0"          action="instantiate"/>
122               <invoke id_port="gusty_1"          action="instantiate"/>
123               <invoke id_port="splitter_output" action="instantiate"/>
124
125           </parallel>
126
127       </sequence>
128
129       <!-- RUN -->
130       <parallel>
131           <invoke id_port="source"           action="run"/>
132           <invoke id_port="shuffler"         action="run"/>
133           <invoke id_port="gusty_0"          action="run"/>
134           <invoke id_port="gusty_1"          action="run"/>
135           <invoke id_port="splitter_output" action="run"/>
136       </parallel>
137
138       <!-- ORCHESTRATION OF COMPUTATIONS AND CONNECTORS -->
139       <parallel>
140
141           <sequence>
142             <iterate id_port="task_shuffle_collector_read_chunk" until="FINISH_CHUNK" loop="READ_CHUNK">
143                 <parallel>
144                     <sequence>
145                         <invoke id_port="task_shuffle_feeder_read_chunk_0" action="READ_CHUNK" />
146                         <invoke id_port="task_shuffle_feeder_chunk_ready_0" action="CHUNK_READY" />
147                         <invoke id_port="task_gusty_0" action="READ_CHUNK" />
148                         <invoke id_port="task_gusty_0" action="PERFORM" />
149                     </sequence>
150                     <sequence>
151                         <invoke id_port="task_shuffle_feeder_read_chunk_1" action="READ_CHUNK" />
152                         <invoke id_port="task_shuffle_feeder_chunk_ready_1" action="CHUNK_READY" />
153                         <invoke id_port="task_gusty_1" action="READ_CHUNK" />
154                         <invoke id_port="task_gusty_1" action="PERFORM" />
155                     </sequence>
```

```
156                 </parallel>
157             </iterate>
158             <parallel>
159                 <sequence>
160                     <invoke id_port="task_shuffle_feeder_read_chunk_0" action="FINISH_CHUNK" />
161                     <invoke id_port="task_gusty_0" action="FINISH_CHUNK" />
162                     <invoke id_port="task_gusty_0" action="CHUNK_READY" />
163                 </sequence>
164                 <sequence>
165                     <invoke id_port="task_shuffle_feeder_read_chunk_1" action="FINISH_CHUNK" />
166                     <invoke id_port="task_gusty_1" action="FINISH_CHUNK" />
167                     <invoke id_port="task_gusty_1" action="CHUNK_READY" />
168                 </sequence>
169             </parallel>
170           </sequence>
171
172
173           <!-- MERGING AND SENDING OUTPUT -->
174           <sequence>
175             <iterate id_port="task_split_collector_read_chunk_0" until="FINISH_CHUNK" loop="READ_CHUNK">
176                 <sequence>
177                     <invoke id_port="task_split_collector_read_chunk_1"  action="READ_CHUNK" />
178                     <invoke id_port="task_split_feeder_read_chunk"  action="READ_CHUNK" />
179                     <invoke id_port="task_split_feeder_chunk_ready" action="CHUNK_READY" />
180                 </sequence>
181             </iterate>
182             <parallel>
183                 <invoke id_port="task_split_feeder_read_chunk_0" action="FINISH_CHUNK" />
184                 <invoke id_port="task_split_feeder_read_chunk_1" action="FINISH_CHUNK" />
185             </parallel>
186           </sequence>
187
188     </parallel>
189
190       <sequence>
191
192         <parallel>
193
194             <!-- RELEASE SERVICE BINDINGS -->
195             <invoke id_port="input_data"      action="release"/>
196             <invoke id_port="global_pairs_0" action="release"/>
197             <invoke id_port="global_pairs_1" action="release"/>
198             <invoke id_port="result_pairs_0" action="release"/>
199             <invoke id_port="result_pairs_1" action="release"/>
200             <invoke id_port="output_data"    action="release"/>
201
202             <!-- RELEASE ACTION BINDINGS -->
203             <invoke id_port="task_shuffle_collector_read_chunk" action="release"/>
204             <invoke id_port="task_shuffle_feeder_read_chunk"    action="release"/>
205             <invoke id_port="task_shuffle_feeder_chunk_ready"   action="release"/>
206             <invoke id_port="task_gusty"                        action="release"/>
207             <invoke id_port="task_split_collector_read_chunk"   action="release"/>
208             <invoke id_port="task_split_feeder_read_chunk"      action="release"/>
209             <invoke id_port="task_split_feeder_chunk_ready"     action="release"/>
210
211             <!-- RELEASE DATA SOURCES, COMPUTATIONS AND CONNECTORS CONTRACTS -->
212             <invoke id_port="source"          action="release"/>
213             <invoke id_port="shuffler"        action="release"/>
214             <invoke id_port="gusty_0"         action="release"/>
215             <invoke id_port="gusty_1"         action="release"/>
216             <invoke id_port="splitter_output" action="release"/>
217
218         <t/ns:parallel>
219
220         <parallel>
221
222             <!-- RELEASE PLATFORMS -->
223             <invoke id_port="platform_source"      action="release"/>
224             <invoke id_port="platform_gusty_0"     action="release"/>
225             <invoke id_port="platform_gusty_1"     action="release"/>
226
227         <t/ns:parallel>
228
229       </sequence>
230
231     </sequence>
232
233 </workflow>
```

Código-fonte 5 – Orquestração para mapreduce.workflow.linear.threestages.noniterative.WORKFLOW

```
1  <workflow>
2
3    <sequence>
4
5        <parallel>
6
7            <!-- RESOLVE PLATFORMS -->
8            <invoke id_port="platform_data_source" action="resolve"/>
9            <invoke id_port="platform_gusty_0"     action="resolve"/>
10           <invoke id_port="platform_gusty_1"     action="resolve"/>
11           <invoke id_port="platform_gusty_2"     action="resolve"/>
```

```
12
13          <!-- RESOLVE SERVICE BINDINGS -->
14          <invoke id_port="input_data"      action="resolve"/>
15          <invoke id_port="global_pairs_0" action="resolve"/>
16          <invoke id_port="local_pairs_1"  action="resolve"/>
17          <invoke id_port="global_pairs_1" action="resolve"/>
18          <invoke id_port="local_pairs_2"  action="resolve"/>
19          <invoke id_port="global_pairs_2" action="resolve"/>
20          <invoke id_port="result_pairs"   action="resolve"/>
21          <invoke id_port="output_data"    action="resolve"/>
22
23          <!-- RESOLVE ACTION BINDINGS -->
24          <invoke id_port="task_shuffle_collector_read_chunk_0" action="resolve"/>
25          <invoke id_port="task_shuffle_feeder_read_chunk_0"    action="resolve"/>
26          <invoke id_port="task_shuffle_feeder_chunk_ready_0"   action="resolve"/>
27          <invoke id_port="task_gusty_0"                        action="resolve"/>
28          <invoke id_port="task_shuffle_collector_read_chunk_1" action="resolve"/>
29          <invoke id_port="task_shuffle_feeder_read_chunk_1"    action="resolve"/>
30          <invoke id_port="task_shuffle_feeder_chunk_ready_1"   action="resolve"/>
31          <invoke id_port="task_gusty_1"                        action="resolve"/>
32          <invoke id_port="task_shuffle_collector_read_chunk_2" action="resolve"/>
33          <invoke id_port="task_shuffle_feeder_read_chunk_2"    action="resolve"/>
34          <invoke id_port="task_shuffle_feeder_chunk_ready_2"   action="resolve"/>
35          <invoke id_port="task_gusty_2"                        action="resolve"/>
36          <invoke id_port="task_split_collector_read_chunk"     action="resolve"/>
37          <invoke id_port="task_split_feeder_read_chunk"        action="resolve"/>
38          <invoke id_port="task_split_feeder_chunk_ready"       action="resolve"/>
39
40          <!-- RESOLVE DATA SOURCES, COMPUTATIONS AND CONNECTORS CONTRACTS -->
41          <invoke id_port="source"          action="resolve"/>
42          <invoke id_port="shuffler_0"      action="resolve"/>
43          <invoke id_port="shuffler_1"      action="resolve"/>
44          <invoke id_port="shuffler_2"      action="resolve"/>
45          <invoke id_port="gusty_0"         action="resolve"/>
46          <invoke id_port="gusty_1"         action="resolve"/>
47          <invoke id_port="gusty_2"         action="resolve"/>
48          <invoke id_port="splitter_output" action="resolve"/>
49
50      </parallel>
51
52      <sequence>
53
54          <parallel>
55
56              <!-- DEPLOY PLATFORMS -->
57              <invoke id_port="platform_data_source" action="deploy"/>
58              <invoke id_port="platform_gusty_0"     action="deploy"/>
59              <invoke id_port="platform_gusty_1"     action="deploy"/>
60              <invoke id_port="platform_gusty_2"     action="deploy"/>
61
62          </parallel>
63
64          <parallel>
65
66              <!-- DEPLOY SERVICE BINDINGS -->
67              <invoke id_port="input_data"      action="deploy"/>
68              <invoke id_port="global_pairs_0" action="deploy"/>
69              <invoke id_port="local_pairs_1"  action="deploy"/>
70              <invoke id_port="global_pairs_1" action="deploy"/>
71              <invoke id_port="local_pairs_2"  action="deploy"/>
72              <invoke id_port="global_pairs_2" action="deploy"/>
73              <invoke id_port="result_pairs"   action="deploy"/>
74              <invoke id_port="output_data"    action="deploy"/>
75
76              <!-- DEPLOY ACTION BINDINGS -->
77              <invoke id_port="task_shuffle_collector_read_chunk_0" action="deploy"/>
78              <invoke id_port="task_shuffle_feeder_read_chunk_0"    action="deploy"/>
79              <invoke id_port="task_shuffle_feeder_chunk_ready_0"   action="deploy"/>
80              <invoke id_port="task_gusty_0"                        action="deploy"/>
81              <invoke id_port="task_shuffle_collector_read_chunk_1" action="deploy"/>
82              <invoke id_port="task_shuffle_feeder_read_chunk_1"    action="deploy"/>
83              <invoke id_port="task_shuffle_feeder_chunk_ready_1"   action="deploy"/>
84              <invoke id_port="task_gusty_1"                        action="deploy"/>
85              <invoke id_port="task_shuffle_collector_read_chunk_2" action="deploy"/>
86              <invoke id_port="task_shuffle_feeder_read_chunk_2"    action="deploy"/>
87              <invoke id_port="task_shuffle_feeder_chunk_ready_2"   action="deploy"/>
88              <invoke id_port="task_gusty_2"                        action="deploy"/>
89              <invoke id_port="task_split_collector_read_chunk"     action="deploy"/>
90              <invoke id_port="task_split_feeder_read_chunk"        action="deploy"/>
91              <invoke id_port="task_split_feeder_chunk_ready"       action="deploy"/>
92
93              <!-- DEPLOY DATA SOURCES, COMPUTATIONS AND CONNECTORS -->
94              <invoke id_port="source"          action="deploy"/>
95              <invoke id_port="shuffler_0"      action="deploy"/>
96              <invoke id_port="gusty_0"         action="deploy"/>
97              <invoke id_port="shuffler_1"      action="deploy"/>
98              <invoke id_port="gusty_1"         action="deploy"/>
99              <invoke id_port="shuffler_2"      action="deploy"/>
100             <invoke id_port="gusty_2"         action="deploy"/>
101             <invoke id_port="splitter_output" action="deploy"/>
102
103         </parallel>
104
105     </sequence>
106
107     <sequence>
108
109         <parallel>
110
111             <!-- INSTANTIATE PLATFORMS -->
112             <invoke id_port="platform_data_source" action="instantiate"/>
113             <invoke id_port="platform_gusty_0"     action="instantiate"/>
```

```
114             <invoke id_port="platform_gusty_1"      action="instantiate"/>
115             <invoke id_port="platform_gusty_2"      action="instantiate"/>
116
117         </parallel>
118
119         <parallel>
120
121             <!-- INSTANTIATE DATA SOURCES, COMPUTATIONS AND CONNECTORS -->
122             <invoke id_port="source"           action="instantiate"/>
123             <invoke id_port="shuffler_0"       action="instantiate"/>
124             <invoke id_port="gusty_0"          action="instantiate"/>
125             <invoke id_port="shuffler_1"       action="instantiate"/>
126             <invoke id_port="gusty_1"          action="instantiate"/>
127             <invoke id_port="shuffler_2"       action="instantiate"/>
128             <invoke id_port="gusty_2"          action="instantiate"/>
129             <invoke id_port="splitter_output" action="instantiate"/>
130
131             <!-- INSTANTIATE SERVICE BINDINGS -->
132             <invoke id_port="input_data"      action="instantiate"/>
133             <invoke id_port="global_pairs_0" action="instantiate"/>
134             <invoke id_port="local_pairs_1"  action="instantiate"/>
135             <invoke id_port="global_pairs_1" action="instantiate"/>
136             <invoke id_port="local_pairs_2"  action="instantiate"/>
137             <invoke id_port="global_pairs_2" action="instantiate"/>
138             <invoke id_port="result_pairs"   action="instantiate"/>
139             <invoke id_port="output_data"    action="instantiate"/>
140
141             <!-- INSTANTIATE ACTION BINDINGS -->
142             <invoke id_port="task_shuffle_collector_read_chunk_0" action="instantiate"/>
143             <invoke id_port="task_shuffle_feeder_read_chunk_0"    action="instantiate"/>
144             <invoke id_port="task_shuffle_feeder_chunk_ready_0"   action="instantiate"/>
145             <invoke id_port="task_gusty_0"                        action="instantiate"/>
146             <invoke id_port="task_shuffle_collector_read_chunk_1" action="instantiate"/>
147             <invoke id_port="task_shuffle_feeder_read_chunk_1"    action="instantiate"/>
148             <invoke id_port="task_shuffle_feeder_chunk_ready_1"   action="instantiate"/>
149             <invoke id_port="task_gusty_1"                        action="instantiate"/>
150             <invoke id_port="task_shuffle_collector_read_chunk_2" action="instantiate"/>
151             <invoke id_port="task_shuffle_feeder_read_chun_2k"    action="instantiate"/>
152             <invoke id_port="task_shuffle_feeder_chunk_ready_2"   action="instantiate"/>
153             <invoke id_port="task_gusty_2"                        action="instantiate"/>
154             <invoke id_port="task_split_collector_read_chunk"     action="instantiate"/>
155             <invoke id_port="task_split_feeder_read_chunk"        action="instantiate"/>
156             <invoke id_port="task_split_feeder_chunk_ready"       action="instantiate"/>
157
158         </parallel>
159
160     </sequence>
161
162     <parallel>
163         <invoke id_port="source"           action="run"/>
164         <invoke id_port="shuffler_0"       action="run"/>
165         <invoke id_port="gusty_0"          action="run"/>
166         <invoke id_port="shuffler_1"       action="run"/>
167         <invoke id_port="gusty_1"          action="run"/>
168         <invoke id_port="shuffler_2"       action="run"/>
169         <invoke id_port="gusty_2"          action="run"/>
170         <invoke id_port="splitter_output" action="run"/>
171     </parallel>
172
173     <!-- ORCHESTRATION OF COMPUTATIONS AND CONNECTORS -->
174     <parallel>
175
176         <!-- 1st STAGE LOOP -->
177         <sequence>
178           <iterate id_port="task_shuffle_collector_read_chunk_0" until="FINISH_CHUNK" loop="READ_CHUNK">
179               <sequence>
180                   <invoke id_port="task_shuffle_feeder_read_chunk_0" action="READ_CHUNK" />
181                   <invoke id_port="task_shuffle_feeder_chunk_ready_0" action="CHUNK_READY" />
182                   <invoke id_port="task_gusty_0" action="READ_CHUNK" />
183                   <invoke id_port="task_gusty_0" action="PERFORM" />
184               </sequence>
185           </iterate>
186           <invoke id_port="task_shuffle_feeder_read_chunk_0" action="FINISH_CHUNK" />
187           <invoke id_port="task_gusty_0" action="FINISH_CHUNK" />
188           <invoke id_port="task_gusty_0" action="CHUNK_READY" />
189         </sequence>
190
191         <!-- 2nd STAGE LOOP -->
192         <sequence>
193           <iterate id_port="task_shuffle_collector_read_chunk_1" until="FINISH_CHUNK" loop="READ_CHUNK">
194               <sequence>
195                   <invoke id_port="task_shuffle_feeder_read_chunk_1"  action="READ_CHUNK" />
196                   <invoke id_port="task_shuffle_feeder_chunk_ready_1" action="CHUNK_READY" />
197                   <invoke id_port="task_gusty_1" action="READ_CHUNK" />
198                   <invoke id_port="task_gusty_1" action="PERFORM" />
199               </sequence>
200           </iterate>
201           <invoke id_port="task_shuffle_feeder_read_chunk_1" action="FINISH_CHUNK" />
202           <invoke id_port="task_gusty_1" action="FINISH_CHUNK" />
203           <invoke id_port="task_gusty_1" action="CHUNK_READY" />
204         </sequence>
205
206         <!-- 3rd STAGE LOOP -->
207         <sequence>
208           <iterate id_port="task_shuffle_collector_read_chunk_2" until="FINISH_CHUNK" loop="READ_CHUNK">
209               <sequence>
210                   <invoke id_port="task_shuffle_feeder_read_chunk_2"  action="READ_CHUNK" />
211                   <invoke id_port="task_shuffle_feeder_chunk_ready_2" action="CHUNK_READY" />
212                   <invoke id_port="task_gusty_2" action="READ_CHUNK" />
213                   <invoke id_port="task_gusty_2" action="PERFORM" />
214               </sequence>
215           </iterate>
```

```
            <invoke id_port="task_shuffle_feeder_read_chunk_2" action="FINISH_CHUNK" />
            <invoke id_port="task_gusty_2" action="FINISH_CHUNK" />
            <invoke id_port="task_gusty_2" action="CHUNK_READY" />
          </sequence>

          <!-- MERGING AND SENDING OUTPUT -->
          <sequence>
            <iterate id_port="task_split_collector_read_chunk" until="FINISH_CHUNK" loop="READ_CHUNK">
                <sequence>
                    <invoke id_port="task_split_feeder_read_chunk"  action="READ_CHUNK" />
                    <invoke id_port="task_split_feeder_chunk_ready" action="CHUNK_READY" />
                </sequence>
            </iterate>
            <invoke id_port="task_split_feeder_read_chunk" action="FINISH_CHUNK" />
          </sequence>

    </parallel>

      <sequence>

        <parallel>

              <!-- RELEASE SERVICE BINDINGS -->
              <invoke id_port="input_data"     action="release"/>
              <invoke id_port="global_pairs_0" action="release"/>
              <invoke id_port="local_pairs_1"  action="release"/>
              <invoke id_port="global_pairs_1" action="release"/>
              <invoke id_port="local_pairs_2"  action="release"/>
              <invoke id_port="global_pairs_2" action="release"/>
              <invoke id_port="result_pairs"   action="release"/>
              <invoke id_port="output_data"    action="release"/>

        </parallel>

        <parallel>

              <!-- RELEASE ACTION BINDINGS -->
              <invoke id_port="task_shuffle_collector_read_chunk_0" action="release"/>
              <invoke id_port="task_shuffle_feeder_read_chunk_0"    action="release"/>
              <invoke id_port="task_shuffle_feeder_chunk_ready_0"   action="release"/>
              <invoke id_port="task_gusty_0"                        action="release"/>
              <invoke id_port="task_shuffle_collector_read_chunk_1" action="release"/>
              <invoke id_port="task_shuffle_feeder_read_chunk_1"    action="release"/>
              <invoke id_port="task_shuffle_feeder_chunk_ready_1"   action="release"/>
              <invoke id_port="task_gusty_1"                        action="release"/>
              <invoke id_port="task_shuffle_collector_read_chunk_2" action="release"/>
              <invoke id_port="task_shuffle_feeder_read_chun_2k"    action="release"/>
              <invoke id_port="task_shuffle_feeder_chunk_ready_2"   action="release"/>
              <invoke id_port="task_gusty_2"                        action="release"/>
              <invoke id_port="task_split_collector_read_chunk"     action="release"/>
              <invoke id_port="task_split_feeder_read_chunk"        action="release"/>
              <invoke id_port="task_split_feeder_chunk_ready"       action="release"/>

              <!-- RELEASE DATA SOURCES, COMPUTATIONS AND CONNECTORS -->
              <invoke id_port="source"          action="release"/>
              <invoke id_port="shuffler_0"      action="release"/>
              <invoke id_port="gusty_0"         action="release"/>
              <invoke id_port="shuffler_1"      action="release"/>
              <invoke id_port="gusty_1"         action="release"/>
              <invoke id_port="shuffler_2"      action="release"/>
              <invoke id_port="gusty_2"         action="release"/>
              <invoke id_port="splitter_output" action="release"/>

        </parallel>

        <parallel>

              <!-- RELEASE PLATFORMS -->
              <invoke id_port="platform_data_source" action="release"/>
              <invoke id_port="platform_gusty_0"     action="release"/>
              <invoke id_port="platform_gusty_1"     action="release"/>
              <invoke id_port="platform_gusty_2"     action="release"/>

        </parallel>

      </sequence>

    </sequence>

</workflow>
```

Código-fonte 6 – Orquestração para mapreduce.workflow.linear.singlestage.iterative. WORKFLOW

```
<workflow>

  <sequence>

      <parallel>

          <!-- RESOLVE PLATFORMS -->
          <invoke id_port="platform_source"     action="resolve"/>
          <invoke id_port="platform_gusty_0"    action="resolve"/>
```

```
10          <invoke id_port="platform_gusty_1"     action="resolve"/>
11
12          <!-- RESOLVE SERVICE BINDINGS -->
13          <invoke id_port="input_data"             action="resolve"/>
14          <invoke id_port="global_input_pairs_0"  action="resolve"/>
15          <invoke id_port="global_input_pairs_1"  action="resolve"/>
16          <invoke id_port="global_output_pairs_0" action="resolve"/>
17          <invoke id_port="global_output_pairs_1" action="resolve"/>
18          <invoke id_port="output_data"           action="resolve"/>
19
20          <!-- RESOLVE ACTION BINDINGS -->
21          <invoke id_port="task_shuffle_collector_active_status_0" action="resolve"/>
22          <invoke id_port="task_shuffle_collector_active_status_1" action="resolve"/>
23          <invoke id_port="task_shuffle_collector_active_status_2" action="resolve"/>
24          <invoke id_port="task_shuffle_collector_read_chunk_0"    action="resolve"/>
25          <invoke id_port="task_shuffle_collector_read_chunk_1"    action="resolve"/>
26          <invoke id_port="task_shuffle_collector_read_chunk_2"    action="resolve"/>
27          <invoke id_port="task_shuffle_feeder_read_chunk_0"       action="resolve"/>
28          <invoke id_port="task_shuffle_feeder_read_chunk_1"       action="resolve"/>
29          <invoke id_port="task_shuffle_feeder_read_chunk_2"       action="resolve"/>
30          <invoke id_port="task_shuffle_feeder_chunk_ready_0"      action="resolve"/>
31          <invoke id_port="task_shuffle_feeder_chunk_ready_1"      action="resolve"/>
32          <invoke id_port="task_shuffle_feeder_chunk_ready_2"      action="resolve"/>
33          <invoke id_port="task_gusty_0"                          action="resolve"/>
34          <invoke id_port="task_gusty_1"                          action="resolve"/>
35          <invoke id_port="reduce_function_action_port_0"          action="resolve"/>
36          <invoke id_port="reduce_function_action_port_1"          action="resolve"/>
37
38          <!-- RESOLVE SOLUTION COMPONENTS CONTRACTS -->
39          <invoke id_port="source"          action="resolve"/>
40          <invoke id_port="shuffler"        action="resolve"/>
41          <invoke id_port="gusty_0"         action="resolve"/>
42          <invoke id_port="gusty_1"         action="resolve"/>
43
44      </parallel>
45
46      <sequence>
47
48          <parallel>
49
50              <!-- DEPLOY PLATFORMS -->
51              <invoke id_port="platform_source" action="deploy"/>
52              <invoke id_port="platform_gusty_0"     action="deploy"/>
53              <invoke id_port="platform_gusty_1"     action="deploy"/>
54
55          </parallel>
56
57          <parallel>
58
59              <!-- DEPLOY SERVICE BINDINGS -->
60              <invoke id_port="input_data"      action="deploy"/>
61              <invoke id_port="global_input_pairs_0"  action="deploy"/>
62              <invoke id_port="global_input_pairs_1"  action="deploy"/>
63              <invoke id_port="global_output_pairs_0" action="deploy"/>
64              <invoke id_port="global_output_pairs_1" action="deploy"/>
65              <invoke id_port="output_data"  action="deploy"/>
66
67              <!-- DEPLOY ACTION BINDINGS -->
68              <invoke id_port="task_shuffle_collector_active_status_0" action="deploy"/>
69              <invoke id_port="task_shuffle_collector_active_status_1" action="deploy"/>
70              <invoke id_port="task_shuffle_collector_active_status_2" action="deploy"/>
71              <invoke id_port="task_shuffle_collector_read_chunk_0"    action="deploy"/>
72              <invoke id_port="task_shuffle_collector_read_chunk_1"    action="deploy"/>
73              <invoke id_port="task_shuffle_collector_read_chunk_2"    action="deploy"/>
74              <invoke id_port="task_shuffle_feeder_read_chunk_0"       action="deploy"/>
75              <invoke id_port="task_shuffle_feeder_read_chunk_1"       action="deploy"/>
76              <invoke id_port="task_shuffle_feeder_read_chunk_2"       action="deploy"/>
77              <invoke id_port="task_shuffle_feeder_chunk_ready_0"      action="deploy"/>
78              <invoke id_port="task_shuffle_feeder_chunk_ready_1"      action="deploy"/>
79              <invoke id_port="task_shuffle_feeder_chunk_ready_2"      action="deploy"/>
80              <invoke id_port="task_gusty_0"                           action="deploy"/>
81              <invoke id_port="task_gusty_1"                           action="deploy"/>
82              <invoke id_port="reduce_function_action_port_0"          action="deploy"/>
83              <invoke id_port="reduce_function_action_port_1"          action="deploy"/>
84
85              <!-- DEPLOY SOLUTION COMPONENTS -->
86              <invoke id_port="source"    action="deploy"/>
87              <invoke id_port="shuffler"  action="deploy"/>
88              <invoke id_port="gusty_0"   action="deploy"/>
89              <invoke id_port="gusty_1"   action="deploy"/>
90
91          </parallel>
92
93      </sequence>
94
95      <sequence>
96
97          <parallel>
98
99              <!-- INSTANTIATE PLATFORMS -->
100             <invoke id_port="platform_source" action="instantiate"/>
101             <invoke id_port="platform_gusty_0"     action="instantiate"/>
102             <invoke id_port="platform_gusty_1"     action="instantiate"/>
103
104         </parallel>
105
106         <parallel>
107
108             <!-- INSTANTIATE SERVICE BINDINGS -->
109             <invoke id_port="input_data"             action="instantiate"/>
110             <invoke id_port="global_input_pairs_0"  action="instantiate"/>
111             <invoke id_port="global_input_pairs_1"  action="instantiate"/>
112             <invoke id_port="global_output_pairs_0" action="instantiate"/>
```

```
113             <invoke id_port="global_output_pairs_1" action="instantiate"/>
114             <invoke id_port="output_data"           action="instantiate"/>
115
116             <!-- INSTANTIATE ACTION BINDINGS -->
117             <invoke id_port="task_shuffle_collector_active_status_0" action="instantiate"/>
118             <invoke id_port="task_shuffle_collector_active_status_1" action="instantiate"/>
119             <invoke id_port="task_shuffle_collector_active_status_2" action="instantiate"/>
120             <invoke id_port="task_shuffle_collector_read_chunk_0"    action="instantiate"/>
121             <invoke id_port="task_shuffle_collector_read_chunk_1"    action="instantiate"/>
122             <invoke id_port="task_shuffle_collector_read_chunk_2"    action="instantiate"/>
123             <invoke id_port="task_shuffle_feeder_read_chunk_0"       action="instantiate"/>
124             <invoke id_port="task_shuffle_feeder_read_chunk_1"       action="instantiate"/>
125             <invoke id_port="task_shuffle_feeder_read_chunk_2"       action="instantiate"/>
126             <invoke id_port="task_shuffle_feeder_chunk_ready_0"      action="instantiate"/>
127             <invoke id_port="task_shuffle_feeder_chunk_ready_1"      action="instantiate"/>
128             <invoke id_port="task_shuffle_feeder_chunk_ready_2"      action="instantiate"/>
129             <invoke id_port="task_gusty_0"                          action="instantiate"/>
130             <invoke id_port="task_gusty_1"                          action="instantiate"/>
131             <invoke id_port="reduce_function_action_port_0"          action="instantiate"/>
132             <invoke id_port="reduce_function_action_port_1"          action="instantiate"/>
133
134             <!-- INSTANTIATE SOLUTION COMPONENTS -->
135             <invoke id_port="source"    action="instantiate"/>
136             <invoke id_port="gusty_0" action="instantiate"/>
137             <invoke id_port="gusty_1" action="instantiate"/>
138             <invoke id_port="shuffler"  action="instantiate"/>
139
140         </parallel>
141
142     </sequence>
143
144     <!-- RUN -->
145     <parallel>
146       <invoke id_port="source"    action="run"/>
147       <invoke id_port="gusty_0" action="run"/>
148       <invoke id_port="gusty_1" action="run"/>
149       <invoke id_port="shuffler"  action="run"/>
150     </parallel>
151
152     <!-- ORCHESTRATION OF COMPUTATIONS AND CONNECTORS -->
153     <sequence>
154
155       <!-- DISABLE FACETS 0 e 1 OF shuffler FOR THE FIRST ITERATION -->
156       <parallel>
157           <sequence>
158               <invoke id_port="task_shuffle_collector_active_status_0" action="CHANGE_STATUS_BEGIN" />
159               <invoke id_port="task_shuffle_collector_active_status_0" action="INACTIVE" />
160               <invoke id_port="task_shuffle_collector_active_status_0" action="CHANGE_STATUS_END" />
161            </sequence>
162            <sequence>
163               <invoke id_port="task_shuffle_collector_active_status_1" action="CHANGE_STATUS_BEGIN" />
164               <invoke id_port="task_shuffle_collector_active_status_1" action="INACTIVE" />
165               <invoke id_port="task_shuffle_collector_active_status_1" action="CHANGE_STATUS_END" />
166            </sequence>
167       </parallel>
168
169       <!-- FIRST ITERATION (shuffler READS INPUT PAIRS ONLY FROM SOURCE) -->
170       <iterate id_port="task_shuffle_collector_read_chunk_2" until="FINISH_CHUNK" loop="READ_CHUNK"/>
171            <parallel>
172                <sequence>
173                   <invoke id_port="task_shuffle_feeder_read_chunk_1" action="READ_CHUNK" />
174                   <invoke id_port="task_shuffle_feeder_chunk_ready_1" action="CHUNK_READY" />
175                   <invoke id_port="task_gusty_0" action="READ_CHUNK" />
176                   <invoke id_port="task_gusty_0" action="PERFORM" />
177                </sequence>
178                <sequence>
179                   <invoke id_port="task_shuffle_feeder_read_chunk_2" action="READ_CHUNK" />
180                   <invoke id_port="task_shuffle_feeder_chunk_ready_2" action="CHUNK_READY" />
181                   <invoke id_port="task_gusty_1" action="READ_CHUNK" />
182                   <invoke id_port="task_gusty_1" action="PERFORM" />
183                </sequence>
184            </parallel>
185       </iterate>
186
187       <parallel>
188           <invoke id_port="task_shuffle_feeder_read_chunk_0" action="FINISH_CHUNK" />
189           <invoke id_port="task_shuffle_feeder_read_chunk_1" action="FINISH_CHUNK" />
190           <invoke id_port="task_shuffle_feeder_read_chunk_2" action="FINISH_CHUNK" />
191           <invoke id_port="task_gusty_0" action="FINISH_CHUNK" />
192           <invoke id_port="task_gusty_1" action="FINISH_CHUNK" />
193       </parallel>
194
195       <parallel>
196           <invoke id_port="task_gusty_0" action="CHUNK_READY" />
197           <invoke id_port="task_gusty_1" action="CHUNK_READY" />
198       </parallel>
199
200       <!-- ENABLE FACETS 1 e 2 and DISABLE FACET 0 OF shuffler FOR THE NEXT ITERATIONS -->
201       <parallel>
202           <sequence>
203               <invoke id_port="task_shuffle_collector_active_status_0" action="CHANGE_STATUS_BEGIN" />
204               <invoke id_port="task_shuffle_collector_active_status_0" action="ACTIVE" />
205               <invoke id_port="task_shuffle_collector_active_status_0" action="CHANGE_STATUS_END" />
206           </sequence>
207           <sequence>
208               <invoke id_port="task_shuffle_collector_active_status_1" action="CHANGE_STATUS_BEGIN" />
209               <invoke id_port="task_shuffle_collector_active_status_1" action="ACTIVE" />
210               <invoke id_port="task_shuffle_collector_active_status_1" action="CHANGE_STATUS_END" />
211           </sequence>
212           <sequence>
213               <invoke id_port="task_shuffle_collector_active_status_2" action="CHANGE_STATUS_BEGIN" />
214               <invoke id_port="task_shuffle_collector_active_status_2" action="INACTIVE" />
```

```
            <invoke id_port="task_shuffle_collector_active_status_2" action="CHANGE_STATUS_END" />
        </sequence>
    </parallel>

    <parallel>

        <!-- NEXT ITERATIONS (shuffler READS INPUT PAIRS ONLY FROM Gust AGENTS) -->
        <iterate  id_port="reduce_function_action_port_0" >

            <select action="TERMINATE">
               <sequence>
                    <invoke id_port="reduce_function_action_port_1" action="TERMINATE" />
                    <break/>
               </sequence>
            </select>

            <select action="CONTINUE">
                <sequence>
                    <invoke id_port="reduce_function_action_port_1" action="CONTINUE" />

                    <iterate id_port="task_shuffle_collector_read_chunk_0">
                        <select action="FINISH_CHUNK">
                            <invoke id_port="task_shuffle_collector_read_chunk_1" action="FINISH_CHUNK" />
                        </select>

                        <select action="READ_CHUNK">
                            <sequence>
                                <invoke id_port="task_shuffle_collector_read_chunk_1" action="READ_CHUNK" />
                                <parallel>
                                   <sequence>
                                      <invoke id_port="task_shuffle_feeder_read_chunk_1"  action="READ_CHUNK" />
                                      <invoke id_port="task_shuffle_feeder_chunk_ready_1" action="CHUNK_READY" />
                                      <invoke id_port="task_gusty_0"                      action="READ_CHUNK" />
                                      <invoke id_port="task_gusty_0"                      action="PERFORM" />
                                   </sequence>
                                   <sequence>
                                      <invoke id_port="task_shuffle_feeder_read_chunk_2"  action="READ_CHUNK" />
                                      <invoke id_port="task_shuffle_feeder_chunk_ready_2" action="CHUNK_READY" />
                                      <invoke id_port="task_gusty_1"                      action="READ_CHUNK" />
                                      <invoke id_port="task_gusty_1"                      action="PERFORM" />
                                   </sequence>
                                </parallel>
                            </sequence>
                        </select>
                    </iterate>

                    <parallel>
                        <invoke id_port="task_shuffle_feeder_read_chunk_0" action="FINISH_CHUNK" />
                        <invoke id_port="task_shuffle_feeder_read_chunk_1" action="FINISH_CHUNK" />
                        <invoke id_port="task_shuffle_feeder_read_chunk_2" action="FINISH_CHUNK" />
                        <invoke id_port="task_gusty_0"                     action="FINISH_CHUNK" />
                        <invoke id_port="task_gusty_1"                     action="FINISH_CHUNK" />
                    </parallel>

                    <parallel>
                        <invoke id_port="task_gusty_0" action="CHUNK_READY" />
                        <invoke id_port="task_gusty_1" action="CHUNK_READY" />
                    </parallel>

                </sequence>
            </select>

        </iterate>

        <!-- RELEASE UNUSEFUL RESOURCES WHILE PROCESSING NEXT ITERATIONS -->
        <sequence>

            <parallel>

                <!-- RELEASE UNUSEFUL SERVICE BINDINGS -->
                <invoke id_port="input_data"     action="release"/>

                <!-- RELEASE UNUSEFUL ACTION BINDINGS -->
                <invoke id_port="task_shuffle_collector_active_status_0" action="release"/>
                <invoke id_port="task_shuffle_collector_active_status_1" action="release"/>
                <invoke id_port="task_shuffle_collector_active_status_2" action="release"/>
                <invoke id_port="task_shuffle_collector_read_chunk_2"    action="release"/>

                <!-- RELEASE UNUSEFUL SOLUTION COMPONENTS -->
                <invoke id_port="source"    action="release"/>

            </parallel>

            <!-- RELEASE UNUSEFUL PLATFORMS -->
            <invoke id_port="platform_source" action="release"/>

        </sequence>

    </parallel>

    <parallel>
        <invoke id_port="task_shuffle_collector_read_chunk_0" action="READ_CHUNK" />
        <invoke id_port="task_shuffle_collector_read_chunk_1" action="READ_CHUNK" />
    </parallel>

    <sequence>
        <invoke id_port="task_shuffle_feeder_read_chunk_0"  action="READ_CHUNK" />
        <invoke id_port="task_shuffle_feeder_chunk_ready_0" action="CHUNK_READY" />
    </sequence>

    <parallel>
        <invoke id_port="task_shuffle_collector_read_chunk_0" action="FINISH_CHUNK" />
```

```
317             <invoke id_port="task_shuffle_collector_read_chunk_1" action="FINISH_CHUNK" />
318           </parallel>
319
320       </sequence>
321
322
323       <!-- RELEASE REMAINING RESOURCES -->
324       <sequence>
325
326           <parallel>
327
328               <!-- RELEASE SERVICE BINDINGS -->
329               <invoke id_port="global_input_pairs_0"  action="release"/>
330               <invoke id_port="global_input_pairs_1"  action="release"/>
331               <invoke id_port="global_output_pairs_1" action="release"/>
332               <invoke id_port="global_output_pairs_0" action="release"/>
333               <invoke id_port="output_data"  action="release"/>
334
335               <!-- RELEASE ACTION BINDINGS -->
336               <invoke id_port="task_shuffle_collector_read_chunk_0"    action="release"/>
337               <invoke id_port="task_shuffle_collector_read_chunk_1"    action="release"/>
338               <invoke id_port="task_shuffle_feeder_read_chunk_0"       action="release"/>
339               <invoke id_port="task_shuffle_feeder_read_chunk_1"       action="release"/>
340               <invoke id_port="task_shuffle_feeder_read_chunk_2"       action="release"/>
341               <invoke id_port="task_shuffle_feeder_chunk_ready_0"      action="release"/>
342               <invoke id_port="task_shuffle_feeder_chunk_ready_1"      action="release"/>
343               <invoke id_port="task_shuffle_feeder_chunk_ready_2"      action="release"/>
344               <invoke id_port="task_gusty_0"                          action="release"/>
345               <invoke id_port="task_gusty_1"                          action="release"/>
346               <invoke id_port="reduce_function_action_port_0"          action="release"/>
347               <invoke id_port="reduce_function_action_port_1"          action="release"/>
348
349               <!-- RELEASE SOLUTION COMPONENTS -->
350               <invoke id_port="gusty_0"  action="release"/>
351               <invoke id_port="gusty_1"  action="release"/>
352               <invoke id_port="shuffler" action="release"/>
353
354           </parallel>
355
356           <parallel>
357
358               <!-- RELEASE PLATFORMS -->
359               <invoke id_port="platform_gusty_0"    action="release"/>
360               <invoke id_port="platform_gusty_1"    action="release"/>
361
362           </parallel>
363
364       </sequence>
365
366   </workflow>
```