GEORGE LUIZ MEDEIROS TEODORO

# SUPORTE A FLUXOS DE TRABALHO PARA APLICAÇÕES INTENSIVAS EM DADOS

Belo Horizonte, Minas Gerais
25 de outubro de 2006

GEORGE LUIZ MEDEIROS TEODORO

# SUPORTE A FLUXOS DE TRABALHO PARA APLICAÇÕES INTENSIVAS EM DADOS

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte, Minas Gerais

25 de outubro de 2006

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

# Suporte a Fluxos de Trabalho para Aplicações Intensivas em Dados

GEORGE LUIZ MEDEIROS TEODORO

Dissertação defendida e aprovada pela banca examinadora constituída por:

Prof. Renato Antonio C. Ferreira – Orientador
Universidade Federal de Minas Gerais

Prof. Dorgival Olavo Guedes Neto
Universidade Federal de Minas Gerais

Prof. Wagner Meira Junior
Universidade Federal de Minas Gerais

Prof. Guilherme Horta Travassos
Universidade Federal do Rio de Janeiro

Belo Horizonte, Minas Gerais, 25 de outubro de 2006

# Resumo

O recente desenvolvimento das tecnologias de aquisição de dados fez com que cada vez tivessemos mais dados disponíveis para processamento, entretanto, o aumento do volume de dados disponível criou uma demanda de processamento que extrapola a capacidade de apenas um computador, dessa forma, aplicações científicas foram forçada a utilizar recursos distribuídos e compartilhados. Entretanto, a maioria das aplicações científicas existentes foram desenvolvidas seqüencialmente e não eram capazes de utilizar recursos distribuídos. Em resposta a essas necessidades foram introduzidos os sistemas de fluxo de trabalho científicos, os quais permitiram a utilização de aplicações sequênciais em ambientes distribuídos, possibilitando a exploração de grandes bases de dados.

A introdução desses sistemas é baseada na observação de que aplicações científicas são construídas pela composição de múltiplos estágios de computação, como em *pipelines* tradicionais, que precisam ser executados em grandes coleções de dados. Dessa forma, os sistema de fluxo de trabalho permitiram que os estágios de computação da aplicação fossem mapeados em estágios de fluxos de trabalho, os quais são compartilhados entre usuários e executados em ambientes distribuídos.

Neste trabalho, apresentamos um sistema de fluxo de trabalho único no sentido de que o mesmo foi especialmente desenvolvido para facilitar a execução dessas aplicações em ambientes distribuídos utilizando bancos de dados para armazenamento de dados científicos. Nosso sistema é otimizado para execução fluxos de trabalho intensivos em dados, pois nos preocupamos com as tarefas de gerenciamento de dados. Os resultados experimentais obtidos com este sistema mostram que podemos alcançar *speedups* próximos do linear para aplicações sofisticadas, criadas por múltiplos componentes.

# Abstract

The increase of the demand of computation and data have forced the scientific applications to use distributed and shared resources. The scientific workflow systems have been introduced in response to the demand of researcher from several domains of science who need to process and analyse this increasingly larger experimental datasets.

The introduction of the workflow systems is based on the observation that scientific applications are constructed by the composition of multiple computation stages as a standard pipeline that need to be executed on very large data collection. In such a way, the scientific workflow systems had allowed the computation stages to be mapped into workflow stages, which can be efficiently executed in distributed systems.

In this work we present scientific workflow system that is unique in sence that it have been developed to facilitate the execution of scientific applications in distributed systems using databases to store scientifc data. Our system is optimized for data-intesive workflows, meaning that we are very concerned with data management issues. The experimental results with our system have shown that we can achieve linear speedups for fairly sophisticated application, created from multiple components.

*Dedico esta Dissertação a minha irmã e mãe Andréa, sem a qual nada disso seria possível, aos meus pais, meus irmãos, a minha linda sobrinha Giovanna, que encheu nossos corações de alegria, a toda minha família e em especial ao Francisco e Luciano que me acolheram e incentivaram durante essa caminhada.*

# Agradecimentos

Primeira, e inevitavelmente agradeço ao ser superior que me deu a oportunidade de viver. Mais uma vez agradeço a esse mesmo Deus por ter me dado a família e amigos que tenho. Não gostaria de falar aqui de sofrimento, mas isso também é parte dessa trajetória. Especialmente quando vi pessoas lindas se privarem de coisas para que eu pudesse estar aqui. Muito obrigado!!!!

Minha família, vocês foram essências para que eu pudesse vencer, meus irmãos, meus pais, mesmo que distantes estiveram sempre presentes nos meus pensamentos e principalmente tive sempre comigo o exemplo de determinação e honestidade acima de tudo. Também agradeço ao Luciano, pelo apoio e por ter sido fundamental para que seguisse essa profissão. Francisco você é um exemplo de homem e com certeza uma pessoa muito especial na minha vida, você já um grande amigo, obrigado pelo apoio e pela dedicação a minha irmã.

Não posso de forma alguma deixar de dizer a minha irmãe Andréa que ela é tão vitorioza nessa batalha quanto eu. Você foi sempre um ponto de referência, não tenho como dizer aqui o quanto te AMO, mas o importante é que sei que você sabe exatamente o tamanho desse meu amor e admiração. Você é minha vida. E é claro que não poderia deixar de falar sobre a aflilhada linda. Giovanna seus padrinhos te amam muito, você só nos alegra!!!!!

Aos meu companheiros de trabalho, meus orientadores o muito obrigado de um aprendiz que fez tudo para que esse trabalho fosse um sucesso. Agradeço a paciência que tiveram e as oportunidade que me abriram. Ao Meira que orientou aquele menino do terceiro período que de pesquisa não sabia nada, novamente ao Meira e ao Dorgival que me acompanharam em me projeto de fim de cursos e finalmente os dois últimos ao meu Orientador de Mestrado Renato pelo apoi, crença, dedicação e exemplo durante esse processo. Não poderia deixa de agradecer ao pessoal de OHIO que foram muito importantes, valeu Tashin e Saltz.

Finalmente, agradeço a todos os amigos que fiz e olha que não foram poucos. Ao Túlio, Zeniel, Serufest, Ismael, Leo Chaves1 e 2, Macambira, James, Gilberto, Coutinho, Barroca e tantos outros que fizem parte da minha vida. Obrigado!!!

# Sumário

**1 Introdução** **1**

1.1 Objetivo deste trabalho . . . 3

1.2 Contribuições do trabalho . . . 4

1.3 Organização do texto . . . 5

**2 Trabalhos Relacionados** **7**

2.1 Sistemas de fluxos de trabalho . . . 7

2.1.1 Prototipagem de fluxos de trabalho . . . 7

2.1.2 Execução de fluxos de trabalho . . . 8

2.2 Ambientes de suporte à execução . . . 11

2.2.1 DataCutter . . . 11

2.2.2 Anthill . . . 13

2.2.3 MapReduce . . . 14

2.2.4 Condor . . . 16

2.3 Mobius . . . 16

2.4 Entrada e saída distribuída de dados . . . 18

**3 Fluxos de Trabalho Científicos** **19**

3.1 Aplicações . . . 20

3.2 Requisitos . . . 20

3.3 Arquitetura proposta . . . 22

3.4 Sumário . . . 24

**4 Implementação** **25**

4.1 Repositório de executáveis . . . 25

4.2 Criador de fluxos de trabalho . . . 27

4.2.1 Descritor do fluxo . . . 27

4.2.2 Criador de filtros . . . 29

4.3 Ambiente distribuído de execução . . . 30

4.3.1 Sistema de suporte a execução . . . 30
4.3.2 Sistema de gerenciamento do fluxo de trabalho . . . 31
4.3.3 Gerenciador de armazenamento persistente de dados (GAPD) 33
4.3.4 Protocolo de comunicação entre os componentes do sistema de gerenciamento de fluxo de trabalho . . . 35
4.4 Sumário . . . 37

**5 Aplicação Exemplo** **39**
5.1 Descrição da aplicação . . . 39
5.2 Mapeamento da aplicação exemplo em fluxo de trabalho . . . 41
5.2.1 Desenvolvimento dos filtros . . . 42
5.2.2 Desenvolvimento do fluxo . . . 42
5.2.3 Fluxo de trabalho da aplicação exemplo . . . 42
5.3 Sumário . . . 44

**6 Experimentos** **45**
6.1 Configuração dos testes . . . 45
6.2 Resultados . . . 45

**7 Conclusões e trabalhos futuros** **51**
7.1 Conclusões . . . 51
7.2 Trabalhos futuros . . . 52
7.2.1 Compartilhamento de componentes em tempo de execução . . 52
7.2.2 Utilização de cache semântico . . . 53
7.2.3 Geração automática de fluxos de trabalho para integração de dados . . . 54

**Referências Bibliográficas** **56**

**Exemplo de código gerado** **63**
Estágio de FG/BG . . . 63
Arquivo de configuração . . . 63
Código do filtro A . . . 64
Makefile . . . 66

# Lista de Figuras

1.1 Fluxos de trabalho . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.2 Exemplo de aplicações de fluxo de trabalho . . . . . . . . . . . . . . . . 2

2.1 Visões oferecidas pelo paradigma Filtro-Fluxo . . . . . . . . . . . . . . . 12
2.2 Arquitetura do Anthill . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.3 MapReduce: Estágios das aplicações . . . . . . . . . . . . . . . . . . . 15

3.1 Arquitetura do Sistema . . . . . . . . . . . . . . . . . . . . . . . . . . 23

4.1 Repositório . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
4.2 Descritor do fluxo (Histogram Normalization) . . . . . . . . . . . . . . 28
4.3 Arquitetura do Mako . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.4 Comunicação entre os componentes do sistema quando o Filtro A lê algum dado . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.5 Comunicação entre os componentes do sistema quando o Filtro A escreve algum dado para o Filtro B . . . . . . . . . . . . . . . . . . . . . . . 37

5.1 Aplicação de segmentação da placenta de ratazana . . . . . . . . . . . . 41
5.2 Fluxo de trabalho da aplicação da placenta de ratazana . . . . . . . . . . 43

6.1 Estágio FG/BG . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
6.2 Estágio de normalização do histograma . . . . . . . . . . . . . . . . . . 47
6.3 Tempo de execução do estágio de normalização do histograma em delhalhes 48
6.4 Estágio de classificação de cores e segmentação de tecidos . . . . . . . . . 50

7.1 Aplicações compartilhando componentes em tempo de execução . . . . . 53
7.2 Aplicações de técnicas de cache semântico . . . . . . . . . . . . . . . . 53
7.3 Dados de mesma semântica armazenados com tipos diferentes . . . . . . 54
7.4 Repositório . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
7.5 Configuração de aplicações em alto nível para integração de dados . . . . 55

# Lista de Tabelas

6.1 Percentual do tempo em que o GMD esta esperando por requisição . . . 49

# Capítulo 1

# Introdução

Os recentes avanços em Ciência da Computação, em especial tecnologia da informação, vêm revolucionando a maneira de conduzir as diversas ciências, como constatado no amplo uso de novas técnicas, resultados e descobertas, que envolvem multi-disciplinas como bioinformática, geoinformática, entre outras. Se por um lado surgem várias oportunidades de descobertas com essa nova maneira de conduzir as ciências, dirigidas por informação e normalmente intensiva em dados, por outro, são criados novos desafios para gerenciar e processar dados científicos. Para responder uma pergunta científica, por exemplo, um cientista precisa não só dominar o conhecimento relacionado, mas, muitas vezes, é preciso ter acesso a dados e informações provenientes de bases de dados comunitárias e saber utilizar ferramentas que possibilitem extrair as informações necessárias.

A forma como cientistas constroem aplicações é tipicamente pela integração de múltiplos estágios de processamento executados ordenadamente, sendo a saída de um estágio utilizada como entrada do próximo. Esse modo de construção tem uma grande sobreposição de conceitos com a forma com que são criados fluxos de trabalho, onde os mesmos são um conjunto de serviços conectados e serviços são componentes que recebem dados de entrada e produzem alguma saída, conforme pode ser visto na figura 1.1. A existência desta intersecção de conceitos levou à utilização de sistemas de fluxo de trabalho(workflows) para a solução desse tipo de problema.

A demanda por fluxos de trabalho [12, 68, 41, 52, 24] impulsionou a criação de sistemas de gerência de fluxo de trabalho, ou seja, sistemas que permitem definir, executar e monitorar fluxos de trabalho [2]. Essa abordagem permitiu que os sistemas de gerência desenvolvidos pudessem executar estas três tarefas de forma eficiente, uma vez que as aplicações científicas tem a mesma demanda, ou seja, são intensivas em dados e executam por um longo período. Estes sistemas, por exemplo, permitem a rápida definição de fluxos a partir da composição de componentes bási-

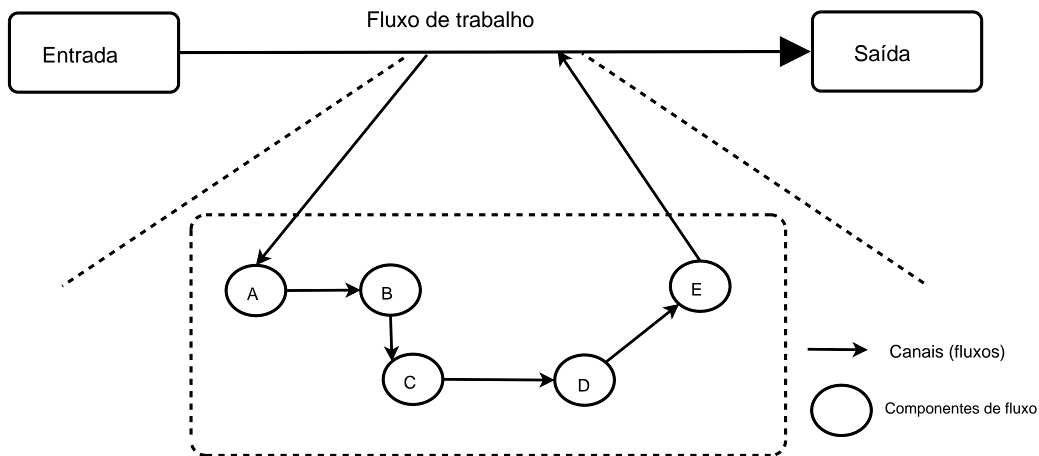



Figura 1.1: Fluxos de trabalho

cos, que por sua vez podem ser compartilhados entre aplicações, como é apresentado na figura 1.2. Essa solução foi muito atraente para pesquisadores que passaram não só a compartilhar dados, mas também componentes, que podem ser reutilizados na criação de outras aplicações.

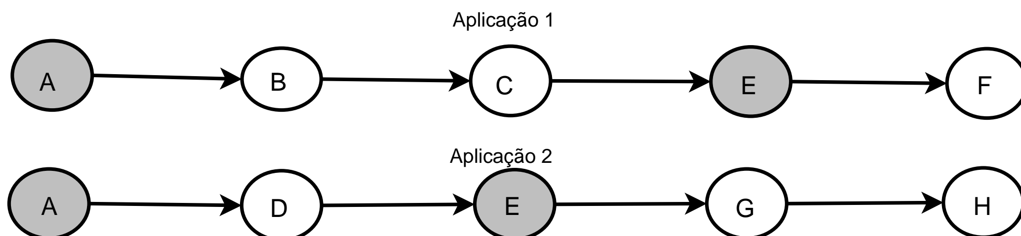



Figura 1.2: Exemplo de aplicações de fluxo de trabalho

Entretanto, o rápido crescimento do número de bases de dados e o seu grande volume de dados intensificam o desafio, qual seja, prover os cientistas do fácil uso de dados e ferramentas de análise disponíveis. A maioria dos dados científicos não estão armazenados em bancos de dados, existindo, portanto, dificuldades em manter, acessar, verificar validade, pesquisar, atualizar, controlar acesso multi-usuário e controle de transações. Um exemplo do volume dos dados armazenados pode ser dada pelo projeto Large Handron Collider (LCD) iniciado em 2006 no CERN, que deverá gerar petabytes de dados por ano [16].

Quando lidamos com volumes de dados dessa grandeza, todas as operações, por mais simples que pareçam, se tornam desafios; em particular, quando investigamos o problema de integração de dados, deparamo-nos com várias dificuldades que po-

deriam ser resolvidas ou amenizadas através da utilização de bancos de dados no armazenamento de dados científicos [42]. Estes problemas são resultado da heterogeneidade dos dados do ponto de vista de sintaxe, estrutura e semântica, além de dificuldades referentes à utilização de sistemas e serviços de infra-estrutura, como leitura de dados em sistemas heterogêneos.

Este problema é agravado pela incapacidade de funcionamento dos sistemas atuais de gerência de fluxo de trabalho, quando precisam lidar com a criação e transporte de grandes volumes de dados pela rede [37]. Todas estas limitações levam a crer que uma abordagem que trate do gerenciamento de dados, ou seja, armazenamento, pesquisa, leitura, controle de acesso, entre outras coisas, eficiente e escalável, utilizando bancos de dados distribuídos é um caminho inevitável.

Assim, neste trabalho, apresentamos um sistema de gerenciamento de fluxos de trabalho científicos, que reduz vários problemas relativos a armazenamento, junção e integração de dados através da utilização de bancos de dados distribuídos para o armazenamento de bases de dados científicas. A solução apresentada é baseada na construção de um sistema que promova a interação eficiente e escalável entre Mobius [33], um banco de dados XML distribuído para sistemas heterogêneos, e Anthill [27, 8, 10], um sistema para execução de workflows baseado no modelo de programação filter-stream. Ainda apresentamos uma proposta de prototipagem de fluxos de trabalho científicos, que provê a rápida criação de fluxo a partir de componentes compilados sem a necessidade de modificação de código.

Esta proposta de criação de um sistema de suporte a fluxo de trabalho utilizando bancos de dados XML distribuídos, integrado ao sistema de suporte à execução difere das soluções anteriormente propostas, que em geral desprezam as dificuldades inerentes ao gerenciamento de dados [34][1].

## 1.1 Objetivo deste trabalho

Projetar, implementar e avaliar um sistema com suporte a fluxos de trabalho para aplicações intensivas em dados, que aborda o gerenciamento e processamento de dados, integrando esses dois elementos de forma eficiente e escalável.

[1]No restante deste trabalho, sempre que nos referirmos a fluxos de trabalho estamos nos referindo a fluxos de trabalho científicos, a menos que digamos explicitamente o contrário.

## 1.2 Contribuições do trabalho

A primeira contribuição deste sistema é a criação de uma abstração para leitura de dados, a qual tem o intuito de facilitar a criação de aplicações que interagem com grandes bases de dados armazenadas em bancos de dados distribuídos. Utilizando esse paradigma de leitura, aplicações informam apenas o tipo de dados (nome da base) que desejam ler. De posse desta informação, o sistema de suporte a fluxo de trabalho cuida do acesso aos dados, onde quer que os mesmos estejam armazenados, retornando-os à aplicação. Alguns dos mecanismos criados para prover alto desempenho nesta tarefa são:

- Leitura paralela de dados em sistemas heterogêneos distribuídos: o sistema cria mecanismos de leitura paralela de dados, assim vários componentes do fluxo de trabalho podem ler dados ao mesmo tempo.
- Balanceamento de carga: o particionamento de dados é feito dinamicamente, ou seja, sempre que um componente, que lê dados provenientes do conjunto de dados de entrada, está disponível, o sistema é avisado e retorna dados para que ele processe. Assim, o trabalho é balanceado entre os componentes, permitindo que aqueles executados em máquinas mais potentes processem mais dados.

A segunda importante contribuição é a criação de mecanismos para armazenamento eficiente de resultados parciais, o que permite ao usuário habilitar o sistema para armazenar os dados de saída de cada um dos componentes do fluxo. Para que tivéssemos bom desempenho nesta tarefa introduzimos os seguintes mecanismos ao sistema:

- Salvamento de dados em segundo plano: cópias de todos os dados trocados entre os componentes do fluxo são enviadas transparentemente ao sistema de suporte a fluxo de trabalho, que primeiramente as armazena em memória primária e posteriormente cuida para que os dados sejam armazenados corretamente em bases de dados distribuídas. Essa estratégia permite que as aplicações não sejam interrompidas para execução do armazenamento dos dados parciais.
- Escrita paralela de dados: o sistema de armazenamento de dados é distribuído e instanciado nas várias máquinas disponíveis no ambiente de execução. Dessa forma, a responsabilidade de armazenar os dados enviados pelos diversos componentes da aplicação são divididos entre as cópias do sistema de armazenamento, o que permite a existência de escrita paralela nas bases de dados distribuídas.

A terceira contribuição decorre do modelo de programação suportado pelo ambiente de programação utilizado, pois o mesmo permite a concorrência segura entre os atores do fluxo em dois níveis:

- Várias instâncias de cada ator podem ser criadas durante a execução e passam a operar identicamente em dados diferentes;
- Existe concorrência entre os diferentes tipos de atores, pois, terminada a tarefa de um ator, o mesmo envia o resultado para o próximo ator no fluxo e passa a executar sobre outros dados, permitindo que o fluxo esteja "cheio" durante a execução.

A quarta contribuição é proveniente da necessidade de prover os cientistas de uma forma eficiente, ou seja, simples e rápida de criar aplicações que utilizem este sistema. Assim, foram desenvolvidos mecanismos para gerar automaticamente componentes de aplicação que sejam criados com base em código compilado (executáveis e bibliotecas compartilhadas). Para tanto, o sistema executa o código compilado nos componentes que o utilizam, provendo suporte à transformação dos dados lidos em tipos de entrada adequados e transformando os dados de saída em mensagens que são enviadas através do fluxo.

## 1.3 Organização do texto

O restante deste trabalho foi dividido em 6 capítulos, organizados da seguinte forma:

**Capítulo 2. [Trabalhos relacionados]** São apresentados os principais trabalhos da área de gerenciamento de fluxos de trabalho científicos, além de algumas abordagens de entrada e saída paralela de dados (parallel I/O).

**Capítulo 3. [Fluxos de trabalho Científicos]** Apresenta alguns dos dos principais requisitos atuais de sistemas de fluxo de trabalho, além da arquitetura do sistema desenvolvido nesta tese.

**Capítulo 4. [Implementação]** Detalha a implementação dos principais componentes do sistema proposto, assim como o protocolo de comunicação utilizado entre eles.

**Capítulo 5. [Aplicação exemplo]** Apresenta a aplicação de segmentação de placenta de ratazana utilizada durante a fase de testes do sistema desenvolvido.

A mesma seção ainda mostra o mapeamento da aplicação em um fluxo de trabalho.

**Capítulo 6. [Experimentos]** São apresentados os experimentos, assim como as formas de avaliações e resultados.

**Capítulo 7. [Conclusão e trabalhos futuros]** Apresenta as conclusões do trabalho e indica algumas sugestões de continuação do mesmo.

# Capítulo 2

# Trabalhos Relacionados

Neste capítulo, discutimos os trabalhos relacionados. Para melhor apresentá-los, estes foram divididos da seguinte forma: na seção 2.1, analisamos alguns dos principais artigos sobre prototipagem, execução e gerenciamento de fluxos de trabalhos científicos; na seção 2.2, apresentamos Anthill [27], um *middleware* para exploração e análise de bases de dados científicas em ambientes distribuídos heterogêneos, utilizado no suporte a execução das aplicações desenvolvidas com este sistema; na seção 2.3, são discutidas as principais características de Mobius [33, 38], um sistema de banco de dados XML distribuído, que é utilizado, no sistema proposto nesta tese, para o armazenamento de dados científicos; e, finalmente, na seção 2.4, são apresentados trabalhos da área de entrada-e-saída paralela, já que algumas destas técnicas são aplicadas no sistema projetado.

## 2.1 Sistemas de fluxos de trabalho

Nesta seção, são apresentadas algumas das principais soluções na área de suporte a execução de fluxos de trabalho. Para facilitar a explicação dividimos os trabalhos em duas classes: prototipagem e execução de fluxos de trabalho, que são descritas nas duas seções a seguir.

### 2.1.1 Prototipagem de fluxos de trabalho

Em [36] é apresentado um sistema de composição de fluxos de trabalho complexos formados por componentes simples. Esse sistema aplica técnicas de inteligência artificial utilizando semântica no auxílio à criação do fluxos de trabalho, o que não era feito por sistemas como os apresentados em [17, 57]. As principais funcionalidades adicionadas são: a verificação de erros entre os tipos de dados de entrada e

saída de cada componente; a qualquer momento o sistema pode gerar automaticamente sugestões de fluxos a partir de especificações incompletas. Também existe a possibilidade de oferecer sugestões durante a composição, tais como substituição de componentes por outros mais eficientes.

O trabalho apresentado por [39] propõe uma visão diferenciada de fluxos de trabalho, criados pela composição de serviços Web. No qual argumenta-se que serviços web são *softwares* tradicionais que podem ser acessados através da internet provendo as mais variadas funcionalidades. Dessa forma, são apresentados modos de composição de fluxos de trabalho utilizando ambientes *web* que dinamicamente identificam falhas e fazem reposição de componentes indisponíveis durante a execução. A solução apresentada é bastante interessante do ponto de estratégia de substituição e busca de serviços que compõem o fluxo, porém muito ineficiente para ser utilizada por sistema de fluxo de trabalho científico que utilizam grandes volumes de dados.

Manolakos e Funk [44] descrevem uma ferramenta para prototipagem de aplicações de processamento de imagens, que utiliza um sistema baseado em componentes implementando mecanismos mestre-escravo; apesar de eficiente do ponto de vista de criação de fluxos de trabalho, a solução é impraticável para utilização em processamento de grandes volumes de dados, já que o gerente se torna um ponto claro de contenção.

### 2.1.2 Execução de fluxos de trabalho

Nesta seção, apresentamos alguns dos principiais sistemas de gerenciamento de execução de fluxos de trabalho científicos.

#### 2.1.2.1 Kepler

Kepler, apresentado em [41], é um sistema de fluxos de trabalho científico desenvolvido com a noção de atores que compõem um fluxo, primeiramente apresentada em [55]. Em Kepler, os usuários desenvolvem fluxos de trabalho através da seleção de componentes ou atores e a colocação dos mesmos no devido estágio do fluxo, sendo os atores conectados por meio de fluxos de dados. Esse sistema apresenta a noção de fluxos de trabalho hierárquicos, ou seja, que podem ser formados por um conjunto de sub-fluxos. Os atores podem ser escritos em Java ou serem externos. Por exemplo, podemos utilizar atores de aplicações C/C++, Matlab, entre outros.

No sistema Kepler, fluxos de dados são sequências de *tokens* de dados, que são enviados de um ator a outro através das conexões disponíveis. Esta forma de con-

dução do processo de criação dos fluxos de trabalho criou uma maneira simples e eficiente de reutilizar os componentes, que podem fazer parte de vários fluxos.

Os recursos *Web* desenvolvidos por Kepler permitem que usuários utilizem os sistemas de fluxo de trabalho científico por meio de uma interação bastante simples. Dentre as vantagens deste método, estão a possibilidade de executar aplicações a partir de qualquer local, de executar uma aplicação sem a necessidade de permanecer conectado, etc.

#### 2.1.2.2 Chimera

O sistema Chimera [29] foi desenvolvido com o intuito de resolver problemas relativos à compartilhamento de dados. Quando analisamos o processo de pesquisa colaborativa, onde várias pessoas utilizam dados oriundos de diversas fontes, surgem problemas de proveniência de dados. Por exemplo, um cientista durante o processo de análise de um conjunto de hipóteses, pode acessar um conjunto de bases de dados distribuída e armazenar resultados localmente para serem utilizados em outros testes, ou até mesmo como resultados finais de seus experimentos. Entretanto, uma vez que estes dados foram extraídos, como poderia o cientista repetir os experimentos sem saber como e de onde estes dados foram derivados?

Na tentativa de amenizar esse problema, Chimera apresenta uma solução para rastrear dados derivados desde suas fontes. A precisão obtida neste processo é suficiente para que o Chimera seja capaz de recriar dados derivados a partir da reexecução do fluxo gerador. Para tanto, é implementado o suporte ao estabelecimento de catálogos virtuais que descrevem como dados gerados por uma aplicação foram derivados desde a fonte. Os catálogos, por sua vez exportam funcionalidades interessantes, tais como: recriar dados removidos, gerar dados derivados que foram definidos mas não executados, recriar dados derivados quando dependências são alteradas ou recriar dados em locais remotos quando isso seja mais eficiente que transferir os mesmos até o local desejado.

Para avaliar os benefícios de rastreamento dos dados gerados, foi criado um protótipo experimental que utiliza os catálogos virtuais propostos. Para tanto, Chimera foi acoplado a outros sistemas [18, 61] possibilitando a criação de fluxos que executam em dados obtidos de bases de dados armazenadas em *grid*. O sistema proposto nesta tese utiliza alguns conceitos de Chimera, tais como, a possibilidade de reexecução de fluxo de trabalho. Porém, a abordagem utilizada é diferente, uma vez que possibilitamos que usuários salvem dados intermediários em processamentos. Desta forma, somos capazes de fazer reexecuções mais eficientes, sendo necessário reexecutar somente a parte do fluxo afetada.

#### 2.1.2.3 Pegasus

O projeto Pegasus [24] desenvolve um sistema para mapear fluxos de trabalho em *grid* de computadores baseado nos recursos disponíveis. Durante a fase de geração e mapeamento das aplicações em fluxos de trabalho, são identificadas duas etapas principiais:

1. Mapeamento dos requisitos de aplicações em termos dos dados desejados para "fluxos de trabalhos abstratos", sendo o tipo de fluxo uma descrição em alto nível da aplicação com detalhes, por exemplo, sobre os tipos de dados de entrada e saída dos componentes da aplicação.

2. Mapeamento dos fluxo de trabalho nos recursos disponíveis em um *grid* de computadores.

Como solução a esses problemas, são apresentados dois tipos de mapeamento: CWG (*Concrete Workflow Generator*), que recebe um "fluxo de trabalho abstrato" (*abstract workflow*) e gera um fluxo concreto mapeado nos recursos disponíveis com otimizações modestas. O ACWG (*Abstract and Concrete Workflow Generator*) é capaz de gerar um "fluxo de trabalho abstrato", baseado em uma descrição incompleta do fluxo. Posteriormente, esse escalonador ainda mapeia os fluxos abstratos em fluxos de trabalho concretos explorando técnicas de planejamento em inteligência artificial.

Além disto, o sistema permite a criação de catálogos virtuais utilizando Chimera, que podem ser reexecutados, nessa tarefa, Condor DAGMan [20] e seus escalonadores são utilizados. Assim como Pegasus, o trabalho proposto nesta tese deve mapear e executar fluxos de trabalho em *Grid*, porém, este trabalho ainda tem maiores preocupações em resolver problemas de gerenciamento dos dados utilizados pela aplicação, criando uma interface transparente e eficiente entre aplicação e dados utilizados.

#### 2.1.2.4 Collection oriented scientific workflows

O trabalho [45] propõe um sistema de suporte a fluxo de trabalho derivado de Kepler, apresentando uma solução para integração de dados baseado em coleção de atores. O desafio de integração de dados está se tornando cada vez maior à medida que as ciências misturam multi-disciplinas utilizando dados provenientes de diversas fontes.

A solução apresentada no artigo é inspirada no modelo de programação baseado em fluxo [49], técnicas baseadas em coleção [22] e linguagens de programação fun-

cional. Nessa solução, os fluxos de trabalho gerados podem trabalhar com dados provindos de bases diferentes com tipos diferentes, isto é possível devido à criação de coleções de atores que executam a mesma tarefa em dados com tipos diferentes, mas com a mesma semântica.

Nesse trabalho, a tarefa de criação das coleções fica a cargo do usuário, assim antes do início da execução, o mesmo é obrigado a gerar atores para cada um dos tipos de dados de entrada. Em tempo de execução, o sistema escolhe o ator adequado de acordo com o tipo de dados de entrada. Por sua vez o suporte a execução foi desenvolvido utilizando Kepler, descrito na seção 2.1.2.1.

## 2.2 Ambientes de suporte à execução

Nesta seção, apresentamos Anthill, um ambiente de programação e suporte a execução de aplicações em ambientes distribuídos heterogêneos, o qual é utilizado no suporte a execução dos fluxos de trabalho. Também detalhamos alguns dos principais trabalhos relacionados ao mesmo.

### 2.2.1 DataCutter

O DataCutter [11, 9, 5, 50, 51, 60] é um *middleware* para exploração e análise de bases de dados científicas em ambientes distribuídos heterogêneos. Seu modelo de programação é chamado de programação filtro-fluxo (*filter-stream*), primeiramente proposto em Active Disks [1] e estendido para programação *grids* por DataCutter.

No modelo de programação filtro-fluxo, filtros são a representação de cada estágio da computação, onde existe transformação sobre dados, e os fluxos são abstrações de comunicação entre os filtros. Aplicações, neste modelo, são criadas por um processo de decomposição em filtros, ou seja, pela divisão da aplicação original em blocos de processamento que comunicam entre si através de um fluxo de dados uni-direcional sobre uma rede de computadores.

Este modelo de aplicação, naturalmente, cria paralelismo de tarefas, pois os filtros são executados como em um *pipeline* comunicando-se através da rede. Além disso, em tempo de execução, pode-se criar múltiplas cópias transparentes de cada um dos filtros que compõem a aplicação nas máquinas disponíveis, criando desta forma uma maneira de replicar cada um dos estágios do *pipeline*. Uma vez que os dados enviados a cada dos estágios também podem ser particionados, cria-se paralelismo de dados.

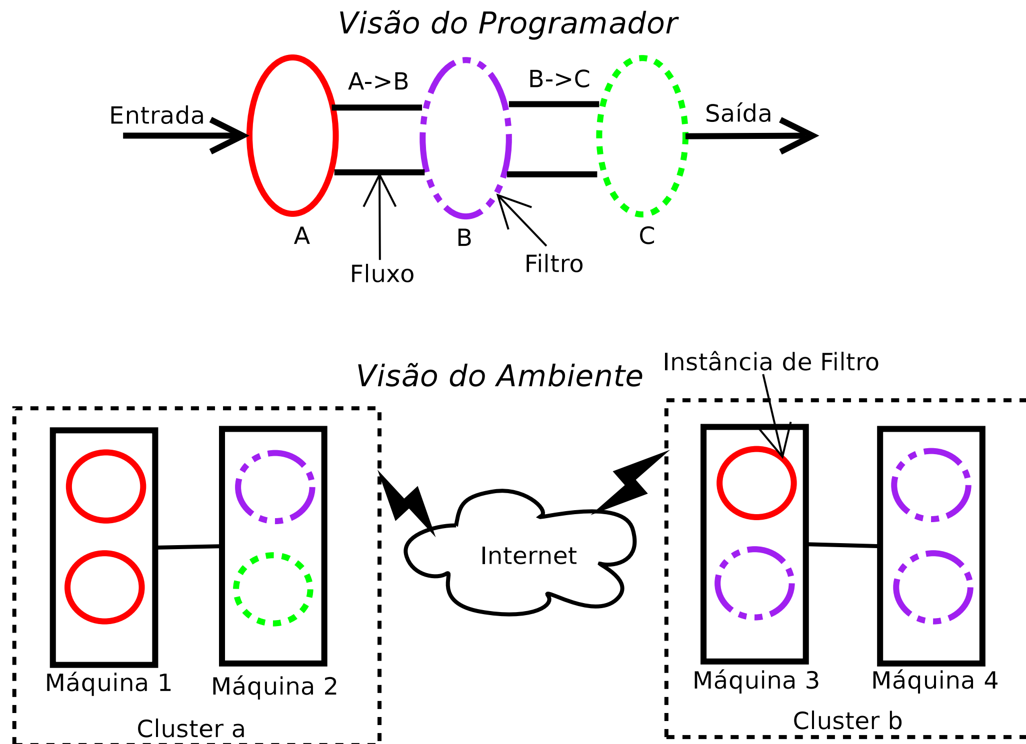



Figura 2.1: Visões oferecidas pelo paradigma Filtro-Fluxo

O paradigma filtro-fluxo oferece duas visões da aplicação, a utilizada pelo programador da aplicação e a visão do ambiente, conforme figura 2.1. Na visão utilizada pelo programador da aplicação, mais conhecida como disposição (*layout*), abstrai-se o número de instâncias de cada filtro e onde elas serão executadas. Já na visão do ambiente (utilizada pelo usuário da aplicação e escalonadores), também conhecida como colocação (*placement*) esses detalhes devem ser levados em conta para garantir o desempenho da aplicação.

No DataCutter, a idéia de processamento de dados sempre foi voltada a paralelismo de grão grosso, onde aplicações são um encadeamento de filtros que se comunicam trocando mensagens (que normalmente contém grande quantidade de dados), sendo que os filtros normalmente aplicam alguma função de redução sobre os dados recebidos e os enviam para o próximo estágio do *pipeline*. Está idéia reduz algumas oportunidades de assincronia, providas por paralelismo de grão fino, que foram resolvidas com a criação do Anthill [27], conforme discutido na seção 2.2.2.

### 2.2.2 Anthill

O Anthill [27] trata-se de uma extensão do DataCutter [11, 9, 5, 50, 51, 60], pois ambos são ambientes de programação em sistemas heterogêneos desenvolvidos utilizando o modelo de programação filtros-fluxo [27].

As extensões criadas neste sistema tem como objetivo explorar assincronia, provendo uma forma de manutenção de estados distribuída que permite a criação de aplicações representadas por grafos cíclicos direcionados, proporcionando a exploração de paralelismo de grão fino entre os ciclos. Esta forma de representação impede que a aplicação seja interrompida esperando que estágios anteriores terminem a execução, pois existe a possibilidade de tarefas de diferentes ciclos estarem sendo executadas concorrentemente.

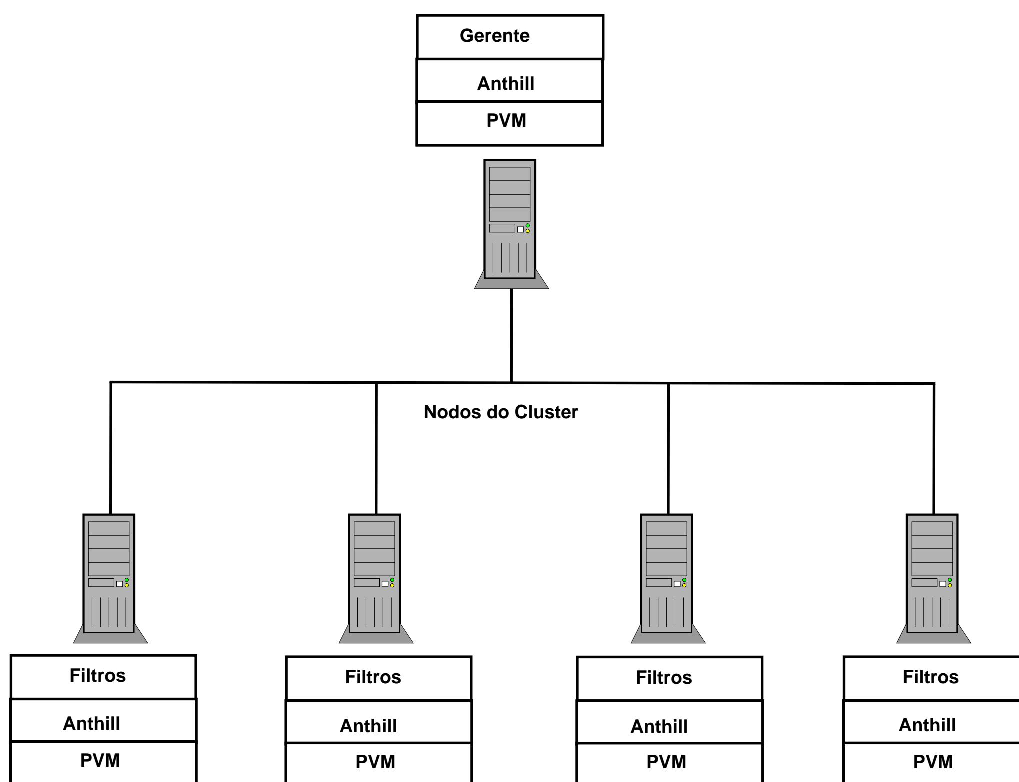



Figura 2.2: Arquitetura do Anthill

Este tipo de paralelismo só foi possível devido à criação de um novo tipo de comunicação chamado de *labeled stream*, apresentado em [56], que cria uma maneira natural de particionamento de dados. Pois o *labeled stream* faz o roteamento das mensagens enviadas por um fluxo para cópias específicas do filtro destino. A cópia para a qual a mensagem vai ser enviada, neste caso, depende exclusivamente do seu conteúdo. Assim, no ato do envio é extraído, da própria mensagem, um *label*

associado à mesma. Cada *label* tem a si uma função *hash* associada, que é invocada com a dupla mensagem e *label* para cada um das vezes que o *stream* é utilizado. O resultado deste *hash* determina a cópia do filtro que deve receber a mensagem.

Os outros importantes pontos abordados por Anthill, que não tratados no Datacutter, são detecção de terminação e tolerância a falhas, conforme discutidos abaixo:

- Detecção de terminação: Em aplicações acíclicas a terminação é detectada pelo envio em cadeia de mensagens de terminação, partindo dos filtros iniciais do fluxo, que por sua vez tem uma visão clara de quando devem terminar sua execução, seguindo com a propagação aos demais filtros, à medida que cada estágio termina. Porém, quando se utiliza grafos cíclicos, como em algumas das aplicações Anthill [65], existe a possibilidade de nenhum estágio ter a idéia clara de que o mesmo chegou ao fim. Dessa forma não é possível iniciar a terminação naturalmente a partir de um certo estágio. Para resolver o problema foi proposto e implementado um algoritmo de detecção de terminação transparente ao usuário [27], que utiliza rodadas de terminação disparadas sempre que um filtro não tenha mais trabalho a fazer. Essas rodadas visam perguntar a todos os filtros do *pipeline* se estes concordam com a terminação da aplicação, ou seja, se também não tem nada a fazer. Assim que a condição é satisfeita o ciclo que forma a aplicação é quebrado e a mesma termina naturalmente.

- Tolerância a falhas: Assim como descrito por Coutinho em [21], criou-se um modelo de tolerância a falhas por *checkpoint* e recuperação baseado em tarefas. Nesse caso tarefa é um evento global com início e fim bem definidos em cada processador, determinada através de chamadas de função createTask() e endTask(). O modelo desenvolvido ainda permite que a aplicação continue progredindo em sua execução enquanto o sistema executa o *checkpoint* das informações, pois o salvamento das mesmas é feita em segundo plano, reduzindo o *overhead* adicionado à execução.

### 2.2.3 MapReduce

MapReduce [23] é outro ambiente de programação em sistemas distribuídos heterogêneos utilizado para processamento de dados em larga escala. O mesmo é bastante semelhante ao DataCutter e Anthill em vários fatores: arquitetura, modo de paralelização, escalonamento e monitoração de aplicações.

Seu modelo de programação permite que programas distribuídos sejam desenvolvidos de forma seqüencial, sendo que em tempo de execução, o sistema cuida da

iniciação de cópias do programa em um ambiente distribuído, além de gerenciar os detalhes relacionados à troca de mensagens.

Porém, quando analisa-se as funcionalidades providas por este sistemas nota-se que o mesmo é limitado quando comparado com DataCutter e Anthill. As maiores limitações são:

- Profundidade máxima do *pipeline*: No MapReduce, assim como diz o nome, as aplicações podem ter apenas duas etapas. A primeira é o Map, onde o filtro lê os dados e aplica algum tipo de mapeamento para particionar os dados entre os filtros da próxima etapa. Na segunda etapa, Reduce, aplica-se algum tipo de redução sobre os dados mapeados pelo estágio anterior, como pode ser visto na figura 2.3. Esta limitação é bastante grave quando analisamos as aplicações desenvolvidas em Anthill e DataCutter, que frequentemente tem um profundidade maior que 2.

- Troca de mensagens via disco: Os dados enviados na fase de Map são armazenados em disco para posteriormente serem lidos durante o Reduce.

- Sincronização: A segunda fase de processamento só pode ser iniciada após o fim da primeira.

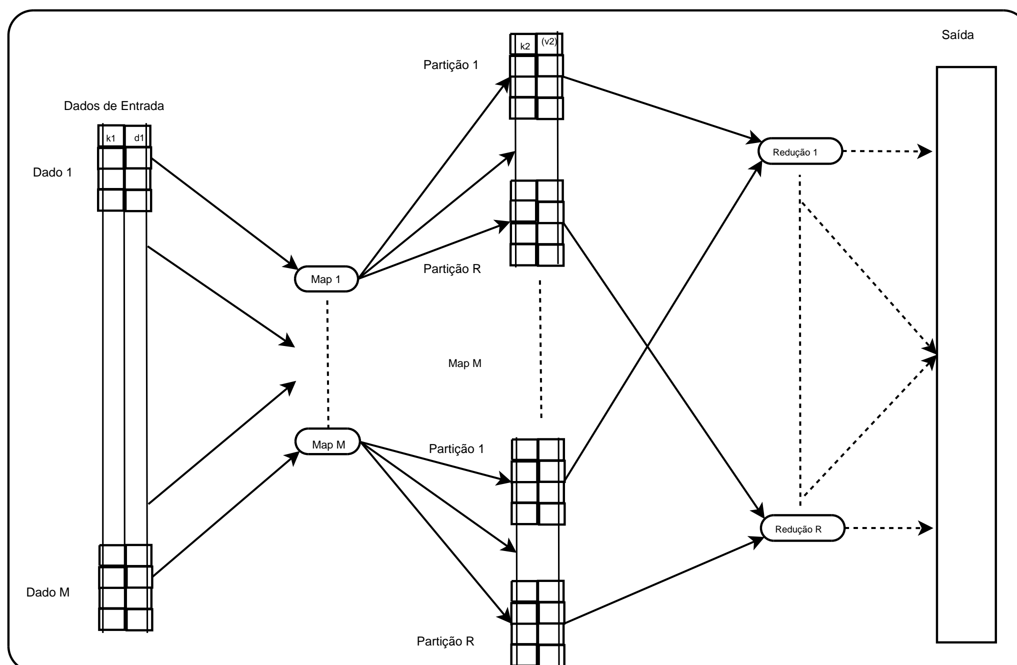



Figura 2.3: MapReduce: Estágios das aplicações

### 2.2.4 Condor

O sistema Condor [20], desenvolvido pela universidade de Wisconsin, teve seu inicio em 1988, tendo como objetivo de ser um grande sistema de gerenciamento de cargas distribuído. Esse sistema tem como funcionalidades fundamentais o escalonamento, monitoração, prioridades e gerenciamento de recursos em Grid de computadores. A aplicação do mesmo permite que máquinas com baixa utilização sejam alocadas para outras tarefas, passando a serem melhor aproveitadas.

O modo de interação do sistema com o usuário possibilita a submissão de trabalhos provindos dos mais diversos pontos da rede, permitindo que os mesmos sejam criados de forma assíncrona. A partir da lista de trabalhos (*jobs*) criados, Condor cuida de escalonar e iniciar todos os trabalhos, permitindo que os mesmos recebam determinação de prioridade. Condor ainda permite que as máquinas disponíveis possam ser configuradas para aceitar trabalhos dependendo de sua carga e que administradores atribuam a diferentes usuários prioridades distintas. Além disso, máquinas podem declarar os recursos disponíveis, enquanto trabalhos declaram seus requisitos. Para facilitar todas estas tarefas, foi criada uma linguagem para descrição da interação entre requisitos e recursos [63].

Condor também suporta processos paralelos, como programas desenvolvidos em PVM [30] e MPI [59]. Dessa forma, a execução dos mesmo não apresenta nenhuma anomalia. Para que esse sistema pudesse utilizar bem os recursos disponíveis, foi desenvolvido um esquema de balanceamento de carga baseado em migração transparente de processos. Assim, processos migrados são executados em máquinas diferentes daquela onde foi feita a chamada original de forma totalmente transparente ao usuário. Este processo é feito com uma sobreposição das chamadas de entrada e saída padrões da biblioteca C do sistema, uma camada adicional entre o programa do usuário e o sistema operacional.

Conforme apresentado, Condor é uma ótima solução para aproveitar melhor os recursos computacionais de instituições que possuem dezenas ou centenas de máquinas, usadas parcialmente, ou sub-utilizadas, com muitos recursos ociosos. Uma vez que o mesmo provê uma boa maneira de utilizar esses recursos para trabalhos computacionalmente caros, sem as dificuldades de executar e migrar os processos manualmente.

## 2.3 Mobius

Mobius [33, 38] é um sistema de banco de dados XML para sistemas distribuídos heterogêneos, que cuida de armazenamento de dados e gerenciamento de meta-dados

relacionados. Este sistema é projetado como um conjunto de serviços frouxamente conectados com protocolos bem definidos, os quais permitem que usuários possam definir e publicar modelos de dados. Possibilitando a execução eficiente de tarefas, tais como: armazenar; pesquisar; retornar e criar visões virtuais XML em bases de dados distribuídas. Este sistema foi construído com intuito de ser utilizado em ambientes de pesquisa, onde dados podem ser compartilhados por diferentes grupos de usuários. Nesse contexto, pesquisadores desejam acessar dados armazenados em bases de dados distribuídas, que muitas vezes apresentam tipos de dados complexos com representações diferentes, com alguma intersecção semântica. Dessa forma, conforme descrito em [33], o sistema que atende a esses requisitos deveria apresentar as seguintes características:

- criação de esquemas de descrição de modelos de dados, registro e compartilhamento dos modelos de descrição entre os usuários,
- mecanismos para facilitar a transformação entre modelos de dados que possuem a mesma semântica, mas representados através de estruturas diferentes,
- criação, integração e gerenciamento de bases de dados e dados conforme os modelos de dados.

Para cobrir os requisitos apresentados o Mobius foi dividido em três componentes, apresentados brevemente a seguir:

1. Global Model Exchange (GME) é o serviço projetados em resposta a demanda de gerenciamento de modelos de dados em *grid*. Através do mesmo os usuários podem criar e publicar modelos de dados, definidos utilizando esquemas XML.
2. Data Instance Management (Mako) é responsável por expor dados com serviços XML através de um conjunto de interfaces que são acessadas via rede. Os dados exportados por este sistema podem ser bases de dados relacionais, bases de dados XML ou até mesmo arquivos armazenados no sistema de arquivos.
3. Data Translation Sevice (DTS) é o componente responsável por fazer a transformação de dados entre tipos diferentes. O que é muito importante quando pesquisadores de instituições diferentes compartilham dados com semântica comum.

Neste trabalho utilizamos principalmente o Mako, uma vez que o mesmo é eficiente nas tarefas de armazenamento e recuperação de dados. Um dos grande motivos do uso desse sistema são as interfaces de interação via redes de computadores, que

facilita o trabalho de gerenciamento dos dados em sistemas de suporte à execução de fluxos de trabalhos executados em ambientes heterogêneos distribuídos. Na seção 4.3.3, discutimos em detalhes as funcionalidades do Mobius Mako e como elas são utilizadas no sistema proposto.

## 2.4 Entrada e saída distribuída de dados

Nesta seção, discutimos especificamente algumas técnicas muito especificas que tratam do problemas de fazer leitura e escrita paralela de dados em *grid* de computadores. Desta forma pretendemos destacar um subproblema de entrada e saída.

Em [6] são apresentados mecanismos de entrada e saída distribuída de dados para aglomerados de computadores, os quais permitem o compartilhamento do acesso a dados e recuperação de forma paralela. Esse sistema requer que os usuários utilizem um "caminho completo" até o arquivo que desejam ler, onde "caminho completo" significa nome da máquina e caminho do arquivo no sistema de arquivos do referido computador.

No artigo [25], são apresentadas outras técnicas de leitura de dados paralela, entretando o trabalho implementa essas funcionalidades de forma transparente ao usuário. Na solução, arquivos armazenados em um *grid* de computadores são acessado como se estivessem salvos localmente, o que cria uma forma mais elegante para aplicações acessarem seus dados.

Essas soluções, apesar de eficientes, não apresentam características interessantes para o armazenamento de grandes bases de dados, uma vez que não são capazes de prover ferramentas de pesquisa, acesso concorrente sobre os dados armazenados, o que pode impossibilitar, por exemplo, a utilização de uma parte apenas da base em uma execução. Contudo, as mesmas não podem ser ignoradas, pois apresentam técnicas de leitura de dados altamente escaláveis e eficientes. Desta forma, conforme descrito na seção 4, utilizamos algumas destes métodos para prover leitura e escrita paralela de dados em bancos de dados XML distribuídos.

# Capítulo 3

# Fluxos de Trabalho Científicos

Fluxos de trabalho podem ser definidos como um conjunto de serviços conectados, onde serviços são componentes que recebem dados de entrada e produzem alguma saída, sendo os dados produzidos e consumidos pelos serviços transportados entre os mesmos. Por sua vez, os sistemas de fluxo de trabalho têm a responsabilidade de definir como os serviços devem ser escalonados e os dados devem fluir entre os serviços [12, 68].

Com o surgimento de fluxos de trabalho científicos foi feita uma distinção entre eles e os fluxos de trabalho de negócios. A separação é decorrente das diferenças de demanda, pois o primeiro normalmente é intensivo em dados e computação, estando mais interessado com ritmo de processamento (*throughput*) dos dados através dos vários algoritmos e aplicações. O último se preocupa com tarefas de escalonamento de execuções, incluindo dependências que não são necessariamente dirigidas por dados e podem incluir agentes humanos [69].

Os sistemas de fluxos de trabalho científicos encontraram ampla aceitação em campos relacionados a ciências médicas a partir do ano 2000. Esse sucesso é decorrente da capacidade de suprir a necessidade por ferramentas inter-conectadas capazes de manipular grande quantidade de dados, reaproveitando programas existentes. O "casamento perfeito" criou uma demanda enorme por este tipo de sistemas, impulsionando a pesquisa na área, o que resultou na criação de dezenas de sistemas [37, 41, 24, 29, 52, 20, 15, 68, 12, 42, 2, 57, 18].

Entretanto, nos últimos anos, observou-se um enorme crescimento da quantidade de bases de dados comunitárias utilizadas no processo de pesquisa. Isso fez com que os sistemas não fossem mais capazes de atender às expectativas dos pesquisadores, pois surgiram novas demandas por gerenciamento de dados (armazenamento, pesquisa, atualização, verificação de tipo, etc) que não eram tratadas de forma satisfatória até então. Criando a demanda por um novo sistema que se preocupe em

criar soluções integrando processamento e gerenciamento de dados eficientemente.

O restante da seção é divido da seguinte forma: na seção 3.1, apresentamos a características das aplicações para as quais este sistema foi projetado, na seção 3.2 discutimos alguns dos requisitos de um sistema de fluxo de trabalho científico, na seção 3.3 apresentamos a arquitetura do sistema proposto, assim como a relação entre os benefícios e os requisitos apresentados.

## 3.1 Aplicações

Este sistema é projetado para suportar a execução eficiente de aplicações intensivas em dados em em computação. A seguir, nessa seção, detalhamos quais são as características dessas aplicações:

- Complexidade: as aplicações que envolvem um processo de pesquisa são normalmente complexas, tanto do ponto de vista dos algoritmos quanto da quantidade de etapas de processamento necessárias. Isso cria a necessidade de composição de aplicação que são formadas por meio de várias etapas de menores;
- Volume de dados: essas aplicações normalmente utilizam um grande volume de dados como entrada e algumas vezes produzem saídas da mesma ordem de grandeza. Assim, é necessário tenhamos mecanismos:
  - que façam entrada e saída de dados eficientemente, evitando que essa tarefa se torne um ponto de conteção do sistema;
  - que sejam capazes de transportar grandes volumes de dados pela rede, uma vez que os dados utilizados nem sempre estão armazenados em somente uma máquina.
- Tempo de execução: o tempo de execução durante o processamento de um grande conjunto de dados é invevitavelmente grande. Assim, é necessário que essas aplicações possam utilizar ambientes de execução distribuídos para melhorar o tempo de resposta.

## 3.2 Requisitos

A seguir, descrevemos alguns dos requisitos desejáveis nestes sistemas. Os mesmos foram provém de observações ou foram extraídos de [41].

**Gerenciamento de dados científicos:** nos últimos anos presenciamos o aumento na quantidade de dados disponíveis para processamento, fato que decorre tanto do aumento do número de bases compartilhadas por diversas corporações, quanto da capacidade de coletar dados, por exemplo, atualmente uma única imagem microscópica pode atingir dezenas de Gigabytes [40]. Este processo cria a necessidade de melhor organizar as bases existentes, provendo funcionalidades como:

- Existência de modelos bem definidos de descrição de dados, que possam ser compartilhados entre os diversos usuários.
- Provimento de armazenamento, leitura e pesquisa, eficiente e escalável em bases de dados distribuídas.
- Verificação de tipos de dados de acordo com o modelo definido, atualmente esta é uma grande dificuldade, pois a ausência de verificação gera bases com muitos ruídos.

**Escalabilidade e eficiência:** fluxos de trabalho científicos normalmente envolvem processamento de grande volume de dados, requerendo altos recursos computacionais. O suporte a fluxos de trabalho intensivos em dados e em processamento é uma tarefa complexa e exige cuidados especiais no desenvolvimento do sistema de suporte a execução, essa é uma preocupação cada vez mais evidente, pois aplicações costumam executar durante intervalos cada vez maiores. Assim, existe a necessidade de que o sistema seja eficiente e escalável em dois níveis:

- Execução eficiente de fluxos de trabalho: para melhorar o desempenho das aplicações, que geralmente aplicam operações complexas em dados para cada um dos parâmetros de estudo. O sistema deve ser capaz de suportar a execução de fluxos de trabalho em ambientes heterogêneos distribuídos, gerenciando a comunicação entre os componentes do fluxo.
- Leitura e escrita de dados: fluxos de trabalho científicos, os quais são normalmente intensivos em dados, exigem que a leitura e escrita dos dados sejam feitas de forma eficiente e escalável, caso contrário estas operações se tornam um ponto claro de contenção na execução das aplicações.

**Armazenamento de resultados parciais:** o processo tradicional de pesquisa consiste da análise de um conjunto hipóteses, que nesse tipo de sistema pode ser traduzido na reexecução do fluxo de trabalho com alteração de parâmetros. Desta forma, seria interessante a existência de uma abordagem que possibilite reexecutar novamente somente a parte do fluxo afetada pela alteração dos parâmetros. O que é possível com o armazenamento de resultados parciais, ou seja, dos dados trocados entre componentes do fluxo, permitindo que o fluxo seja recriado a partir do componente afetado.

**Composição de componentes:** a composição de componentes para geração de tarefas complexas é um tópico de pesquisa atual, esta etapa é fundamental na determinação de como os serviços disponíveis devem ser integrados para criar o fluxo de trabalho desejado. Desenvolver ferramentas que permitam a rápida e simples prototipagem de fluxos é fundamental para o sucesso do sistema, já que este é o ponto de maior interação com usuário.

## 3.3 Arquitetura proposta

Nesta seção, apresentamos a arquitetura do sistema desenvolvido, o qual é composto por duas partes principiais: O criador de fluxo de trabalho e o ambiente distribuído de execução, conforme pode ser visto na figura 3.1.

A primeira parte foi desenvolvida com a intenção de prover um modo simples de composição de componentes, que permite o rápido mapeamento de aplicações em componentes de um fluxo. Para tanto foi criada uma forma de prototipagem de fluxo de trabalho baseada em um arquivo XML bem definido, onde são especificados os requisitos da aplicação, tais como: dados de entrada de cada um dos componentes, funções de deserialização e serialização, tipo de comunicação entre os componentes e assim por diante. Essa forma de composição permite que aplicações, originalmente projetadas para operar centralizadamente, possam rapidamente ser mapeadas em componentes de um fluxo, sendo executadas eficientemente de forma distribuída.

O ambiente distribuído de execução foi projetado para prover suporte à execução de fluxos de trabalho de aplicações intensivas em dados. Esta parte do sistema é dividida em sistema de suporte a execução, gerenciador de meta-dados, armazenador de dados em memória e gerenciador de armazenamento persistente.

**O sistema de suporte a execução** utiliza Anthill [27] na execução dos fluxos de trabalho, permitindo a execução eficiente e escalável de fluxos de trabalho em sistemas distribuídos heterogêneos.

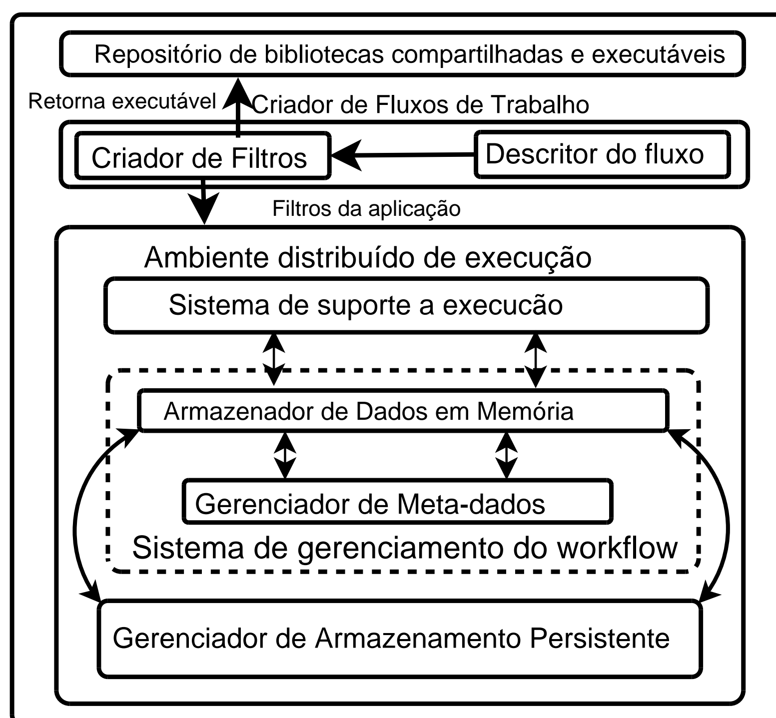



Figura 3.1: Arquitetura do Sistema

**Gerenciador de meta-dados** é responsável por gerenciar o fluxo de execução como um todo, gerenciando a informação sobre os dados lidos e escritos pela aplicação. Sendo responsável por decidir, sob-demanda, qual parte dos dados de entrada são processados por cada uma das cópias dos filtros. Para criar o conjunto de dados de entrada o mesmo recebe um pesquisa do tipo Xpath, a qual é repassada ao armazenador de dados persistente que retorna os meta-dados dos elementos que satisfazem a requisição.

**O armazenador de dados em memória** funciona como um intermediário entre a aplicação e o gerenciador de armazenamento persistente. Baseado nos meta-dados providos pelo gerenciador, ele executa a leitura dos dados necessários do gerenciador de armazenamento persistente e escreve as saídas de cada componente. Para evitar que a aplicação seja interrompida durante o armazenamento de resultados parciais, esta tarefa é executada em segundo plano, ou seja, os dados enviados entre componentes são salvos em memória primária e posteriormente repassados ao gerenciador de armazenamento persistente.

**O gerenciador de armazenamento persistente (GAP)** foi construído com a ferramenta Mobius [33, 38], apresentada na seção 2.3, no armazenamento de dados de entrada e criados durante a execução pela aplicação.

A solução proposta utiliza banco de dados no armazenamento de dados científicos, pois essa abordagem resolve a maioria dos problemas de gerenciamento dos mesmos [37]. Isso ocorre porque bancos de dados tem formas eficientes e bem definidas de manipular dados de maneira geral. A construção do GAP sobre o Mobius [33, 38] foi motivada pela capacidade dessa ferramenta executar tais tarefas eficientemente em ambientes distribuídos.

O gerenciador de meta-dados e o armazenador de dados em memória foram criados com intuito de prover um interação eficiente e escalável entre o sistema de suporte à execução e o gerenciador de armazenamento persistente. Os dois foram desenvolvidos como filtros Anthill e podem ser replicados tantas vezes quanto necessárias, criando um ambiente sem pontos de contenção.

Neste sistema, conforme descrito em detalhes na seção 4.3.2, sempre que um filtro precisa ler dados, ele avisa a um dos armazenadores de dados em memória. O último pergunta por um dado disponível ao gerenciador de meta-dados, que decide em tempo de execução qual dado deve ser processado pelo filtro. Assim que o gerenciador de meta-dados recebe a resposta, ele executa a leitura do dado. Esta estratégia desenvolvida cria uma forma eficiente de leitura de dados distribuída.

## 3.4 Sumário

Neste capítulo, definimos formalmente sistemas de fluxo de trabalho científicos, assim como distinguimos os mesmos dos sistemas de fluxo de trabalho comerciais. Em seguida, baseado em nossas observações, listamos os requisitos desejáveis aos sistemas de gerenciamento de fluxo científicos, que são: gerenciamento de dados científicos; escalabilidade e eficiência; armazenamento eficiente de resultados parciais; composição de componentes. Finalmente, baseado nos requisitos observados, propomos a arquitetura de um sistema de fluxo de trabalho, o qual tenta atender requisitos desejados e suprir as deficiências dos sistemas semelhantes.

# Capítulo 4

# Implementação

Este capítulo, explica em detalhes a implementação e o funcionamento do sistema de suporte a fluxo de trabalho desenvolvido. O sistema foi dividido em três partes principais, como pode ser visto na figura 3.1, são elas: repositório de bibliotecas compartilhadas e executáveis, criador de programas e ambiente distribuído de execução, descritos nas seções 4.1, 4.2 e 4.3.

## 4.1 Repositório de executáveis

Este componente do sistema foi desenvolvido com o intuito de atingir dois objetivos: o primeiro é o compartilhamento, entre os diversos usuários, dos programas, que podem ser executáveis ou bibliotecas compartilhadas, o segundo é a ferramenta de auxilio na fácil e rápida prototipagem de fluxos de trabalho a partir desses programas. Assim, o repositório cria mecanismos para que desenvolvedores possam compartilhar programas e informações sobre os mesmos. As tarefas executadas para atingir esses objetivos são: salvar, pesquisar e recuperar os programas disponíveis que estejam acessíveis a todos os usuários, como pode ser visto na figura 4.1.

Para possibilitar fácil interação do usuário com o repositório, o mesmo foi implementado utilizando Mobius [33], um banco de dados XML distribuído, descrito em detalhes na seção 4.3.3. Esse tipo de banco de dados provê ótimos mecanismos para definição dos elementos armazenados, o que é fundamental ao componente, uma vez que programas armazenados são compartilhados entre vários usuários. Essa importância decorre do fato que, quanto melhor definidos os programas, menos dúvidas existirão durante sua utilização. Esse banco de dados ainda incorpora ferramentas que permitem aos seus usuários acessarem bases armazenadas em um *grid* de computadores, a partir de um ponto único, e facilmente executar as três operações básicas necessárias aos usuários do repositório: salvar, pesquisar e recuperar programas e

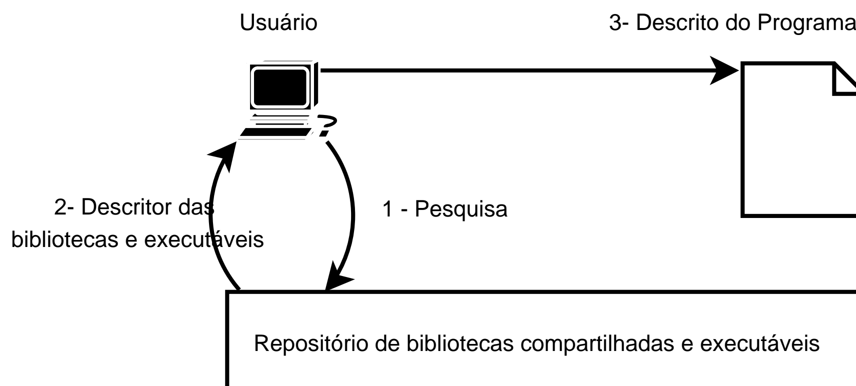



Figura 4.1: Repositório

os XML que descrevem os mesmos. A seguir descrevemos os pontos importantes na execução destas operações:

1. Salvar programa: esta operação requer a criação de um arquivo XML com os meta-dados que descrevem o programa a ser salvo no repositório. Esse arquivo contém todas informações necessárias para identificação dos tipos de dados sobre os quais o programa pode operar, as estruturas usadas como parâmetro para o mesmo, as funções de de-serialização e serialização utilizadas para criar os argumentos a partir das mensagens de entrada ou transformar as saídas em mensagens, as quais são enviadas aos próximos componentes do fluxo. Além das informações que são diretamente utilizadas para a criação do fluxo a partir dos programas armazenados, são salvos dados sobre os requisitos para a execução do programa, tais como: compilador utilizado para geração do programa, sistema operacional, etc. Uma vez que as informações sejam preenchidas o usuário está apto a submeter seu programa, sendo o arquivo com os meta-dados e o programa salvos no banco de dados.

2. Pesquisar: durante esta fase o usuário utiliza o *Virtual Mako(Vmako)*, que é uma das ferramentas inclusas no Mobius, a qual cria visualização centralizada para pesquisa nas diversas bases de dados armazenadas utilizando Mobius. Através desta interface o usuário pode executar pesquisas Xpath e visualizar os meta-dados dos elementos armazenados.

3. Recuperar: a última operação é executada utilizando uma das ferramentas criadas nesse sistema, que recebe o identificador do programa e salva o mesmo no diretório corrente.

Uma vez que o usuário tenha terminado sua interação com este componente, todos os programas utilizados na composição do fluxo devem estar salvos no diretório onde o mesmo será criado. Também requisita-se que esse diretório seja o mesmo em todas as máquinas, pois a ferramenta para disparo de aplicações usa caminho absoluto em cada uma das máquinas. Para evitar maiores problemas, relacionados à coerência entre os programas armazenados, é recomendado o uso de sistemas do tipo NFS -*Network File System* para armazenamento dos arquivos.

## 4.2 Criador de fluxos de trabalho

Nesta seção, descrevemos o criador de fluxos de trabalho, uma ferramenta que permite a incorporação de programas compilados em fluxos de trabalho, sem a necessidade de reescrita de código. Para possibilitar a reutilização desses programas nos fluxos, a ferramenta cria códigos para cada elemento do fluxo de trabalho, os quais são suportados pelo ambiente de execução e cuidam das chamadas aos programas compilados, criando as estruturas de entrada conforme descrito e redirecionam as saídas para mensagens. Dois são os elementos que compõem o criador de fluxos de trabalho: "Descritor do fluxo" e "Criador de filtros".

### 4.2.1 Descritor do fluxo

O "Descritor do fluxo" é o arquivo de configuração que descreve o fluxo de trabalho como um todo, o qual é dividido em quatro seções: *hostDec*, *placement*, *layout* e *compiledFilters*, conforme figura 4.2.

- *hostDec*: Essa seção é utilizada para declarar todas as máquinas que estão disponíveis para a execução das aplicações. Pode-se ainda associar recursos a cada uma das máquinas, assim componentes do fluxo que demandam de tal recurso são executados onde os mesmos estão disponíveis.

- *placement*: No *placement*, o usuário deve declarar quais são os componentes (filtros) do fluxo e quantas instâncias de cada um devem ser criadas.

- *layout*: A seção *layout* define como os componentes do fluxo estão conectados. Para tanto, são criados canais de comunicação (*streams*) entre os mesmos, os quais são direcionais e executam uma das políticas de troca de mensagens disponíveis:

  - *Random*: Escolhe-se de forma aleatória o destino a cada vez que uma mensagem.

```xml
<programDescriptor>
  <hostDec>
    <host name="mymachine.bmi.ohio-state.edu"/>
  </hostDec>
  <placement>
    <filter name="HistogramNomalization" libName="HistogramNormaliztionFilter.so" instances="2">
    <filter name="Writer" libName="WriterFilter.so" instances="2">
  <placement/>
  <layout>
    <stream>
      <from filter="HistogramNormalization" port="histOut" policy="roundRobin"/>
      <to filter="Writer" port="writerInput" />
    </stream>
  </layout>
  <compiledFilters>
    <matLabFilter name="HistogramNormalization" matLabLibName="libMyHistogramNormalization" firstFiler="yes">
      <function headerName="mflMyHistogramNormalization" numoutputs="2" numinputoutpus="0" numinputs="6">
        <arg argType="mxArray*" inputType="userArg" userArgIndex="1" deserializeFunc="stringToMxArray" deserializeLib="libdeserialize.so" order="5"/>
        <arg argType="mxArray*" inputType="userArg" userArgIndex="2" deserializeFunc="stringToMxArray" deserializeLib="libdeserialize.so" order="6"/>
        <arg argType="mxArray*" inputType="userArg" userArgIndex="3" deserializeFunc="stringToMxArray" deserializeLib="libdeserialize.so" order="7"/>
        <arg argType="mxArray*" inputType="userArg" userArgIndex="4" deserializeFunc="stringToMxArray" deserializeLib="libdeserialize.so" order="8"/>
        <arg argType="mxArray*" inputType="msg" msgIndex="1" deserializeFunc="uint8MatrixMToMxArray" deserializeLib="libdeserialize.so" order="3"/>
        <arg argType="mxArray*" inputType="msg" msgIndex="2" deserializeFunc="uint8MatrixMToMxArray" deserializeLib="libdeserialize.so" order="4"/>
        <arg argType="mxArray**" serializeFunc="mxArrayToDCBuffer" serializeLib="libserialize.so" order="1" msgIndexOut="1"/>
        <arg argType="mxArray**" serializeFunc="mxArrayToDCBuffer" serializeLib="libserialize.so" order="2" msgIndexOut="2"/>
      </function>
    </matLabFilter>
  </compiledFilters>
</programDescriptor>
```

Figura 4.2: Descritor do fluxo (Histogram Normalization)

  – *Round-robin*: Nesta política escolhe-se de forma aleatória a primeira instância do destino que receberá a mensagem inicial. A partir desse momento, as mensagens enviadas vão para a instância cujo índice seja igual ao atual mais um, a menos que a instância atual corresponda à última, neste caso a primeira instância do filtro destino deve receber a mensagem.
  – *Broadcast*: No caso do *broadcast* todas as instâncias do filtro destino devem receber a mensagem enviada.
  – *Labeled stream*: A instância do filtro que deve receber a mensagem é calculada em função de um *label* extraído da mensagem. Assim o usuário deve definir uma função que recebe o *label* e retorna um valor inteiro, que por sua vez é mapeado na instância, através da função módulo que utiliza a quantidade total de instância do filtro destino.
  – *Multicast labeled stream*: A política é parecida com a anterior, entretanto a função que recebe o *label* retorna um vetor de inteiros, e cada um dos valores armazenados é mapeado em uma instância do filtro destino.

- *compiledFilters*: É a seção utilizada para que o usuário declare toda a informação necessária para que o sistema possa gerar os códigos dos componentes do fluxo de trabalho que interagem com as bibliotecas compartilhadas e os exe-

cutáveis. Para cada um dos componentes que o sistema deve gerar, o usuário precisa declarar:

- biblioteca ou executável que o mesmo utiliza;
- o número de argumentos;
- estrutura utilizada pelo argumento, por exemplo, *int* ou *mxArray*.
- o tipo do argumento, que pode ser *input, inputOutput, output*;

  Para cada argumento dos tipos *input ou inputOutput*:

  * a fonte, ou seja, se o argumento vem de linha de comando ou mensagem;
  * o índice do argumento em sua fonte;
  * o índice do argumento na função ou executável utilizado;
  * a função de deserialização utilizada para criar o argumento de entrada no formato correto;

  Para cada argumento dos tipos *inputOutput ou output*

  * o índice do argumento na função ou executável utilizado;
  * a função de serialização para transformar o argumento de saída em mensagem;
  * a ordem com que o argumento deve ser adicionado na mensagem;

### 4.2.2 Criador de filtros

O criador de filtros é responsável pela geração do código do filtro de cada um dos componentes do fluxo de trabalho. Essa ferramenta recebe como entrada o "Descritor do fluxo", detalhado na seção 4.2.1. Esse arquivo contém informação sobre quais funções são executadas por cada um dos filtros que compõe o fluxo, assim como todos as outras informações necessárias para ligar os filtros às bibliotecas utilizadas

etc.

```
while existe dado para ser processado do
    lê(dado);
    dadoEntrada = deserialize(dado);
    dadoSaida = processamento(dadoEntrada);
    if se existe dado de saída then
        mensagemSaida = serialize(dadoSaida);
        escreve(mensagemSaida);

end
```

**Algoritmo 1**: Filtros da aplicação

Os filtros criados pela primeira versão da ferramenta, como pode ser visto no algoritmo 1, cria um *loop* que lê dados do sistema de gerenciamento de fluxos de trabalho enquanto existem dados disponíveis para processamento, executa as funções de deserialização e processamento. Se existe algum dado de saída, o filtro executa a função de serialização sobre o mesmo e o envia para o próximo filtro do fluxo. Exemplos de códigos gerados podem ser vistos no apêndice 7.2.3, os quais correspondem a códigos gerados para a aplicação exemplo, detalhada na seção 5.

A ferramenta cria os códigos, como descrito anteriormente, para cada um dos filtros especificados na sub-seção *compileFilters* do "Descritor do fluxo" e o *Makefile* utilizado para compilar e ligar os filtros com as bibliotecas apropriadas. Para que o *Makefile* ligue os filtros corretamente é requerida a criação de uma variável de ambiente referindo-se ao diretório onde bibliotecas utilizadas estão armazenadas.

## 4.3 Ambiente distribuído de execução

Nesta seção, descrevemos, em detalhes, as funcionalidades de cada um dos componentes responsáveis pela execução dos fluxos de trabalho, também apresentamos o protocolo utilizado entre eles para a manutenção da consistência durante a execução das aplicações. A seguir, detalhamos cada uma das três partes integrantes do ambiente de execução:

### 4.3.1 Sistema de suporte a execução

Conforme discutido anteriormente, esse sistema foi desenvolvido utilizando Anthill [27], o qual é responsável por instanciar os componentes do sistema de suporte a fluxo de trabalho em ambientes distribuídos e gerenciar a comunicação entre eles.

Neste trabalho, estendemos Anthill para prover comunicação transparente entre a aplicação e o sistema de gerenciamento do fluxo de trabalho, descrito na seção 4.3.2. As modificações dão suporte a troca de informações necessárias para o gerenciamento do fluxo. As informações trocadas são, por exemplo, quais filtros estão disponíveis para processar dados e quais dados já foram processados.

### 4.3.2 Sistema de gerenciamento do fluxo de trabalho

O sistema de gerenciamento do fluxo de trabalho é composto de dois componentes, o gerenciador de meta-dados (GMD) e o armazenador de dados em memória (ADM). Ambos foram desenvolvidos utilizando a ferramenta de suporte a execução Anthill [27], discutida na seção 4.3.1, desta forma, pode-se, durante a execução, instanciar tantas cópias de cada um dos componentes, quantas sejam necessário. A seguir descrevemos detalhadamente o GMD e o ADM.

#### 4.3.2.1 Gerenciador de meta-dados(GMD)

É o componente responsável pela monitoração e gerenciamento dos meta-dados relacionados a todos dados que transitam durante a execução do fluxo de trabalho. Todas as operações de leitura e escrita de dados dependem deste elemento, pois o mesmo controla os dados que devem ser processados e os criados pelo fluxo. Dessa forma, todas as vezes que o primeiro filtro do fluxos precisa ler dados, o GMD é comunicado sobre essa necessidade, decidindo qual documento ou dados deve ser entregue ao filtro. Sendo assim, toda "inteligência" relacionada a leitura de dados está contida no GMD, o que o torna importantíssimo no processo de execução das aplicações.

Em nosso trabalho, definimos uma unidade de dados, lida do conjunto de entrada ou trocada entre filtros da aplicação, como um documento. Sendo que, no caso do conjunto de entrada, por exemplo, um documento corresponde a cada um dos registros armazenados no gerenciador de armazenamento persistente de dados (GAPD), descrito em detalhes na seção 4.3.3.

Assim, quando a execução do fluxo de trabalho é iniciada, o GMD recebe como entrada uma pesquisa do tipo Xpath, que é utilizada para delimitar o conjunto de dados ou documentos de entrada da aplicação. A primeira ação do GMD é repassar a requisição de pesquisa ao gerenciador de armazenamento persistente de dados (GAPD), descrito em detalhes na seção 4.3.3, que retorna os meta-dados de cada um dos documentos que satisfazem a requisição.

Nesse momento, o GMD, com base nos meta-dados retornados, tem informação sobre localização de cada um dos documento do conjunto de entrada, o mesmo cria um identificador único para cada documento e armazena seus meta-dados. Em seguida, sempre que existir uma requisição de leitura o GMD utiliza esses meta-dados para decidir qual documento deve ser retornado ao filtro.

O GMD não só controla a ordem com que os documentos do conjunto de entradas do sistema devem ser lidos, mas também os documentos criados durante a execução. Temos como definição que cada mensagem enviada entre filtros do fluxo é tida como um documento. Esse controle visa, entre outras coisas, a leitura de dados, como discutido acima, o armazenamento de dados intermediários, ou seja, os dados criados em tempo de execução e o armazenamento de resultados. Para facilitar o controle definiu-se que os documentos podem, durante a fase de execução, atingir três estados:

- "Não processado": todos os documentos do conjunto de dados de entrada são considerados "não processados" no início da execução, o que significa que os mesmos estão disponíveis para serem processados.

- "Sendo processado": o documento assume este estado quando é retornado para algum filtro que o tenha requisitado dados. Além disso, todos os documentos criados durante a execução estão nesse estado, uma vez que eles foram criados e enviados para serem processados por outro filtro.

- "Processado": um documento é dito processado somente quando o mesmo já foi processado por um filtro e o resultado de seu processamento já tenha sido enviado para outro filtro e ao ADM, descrito na seção a seguir.

Como dito anteriormente, o GMD é responsável por decidir, em tempo de execução, qual documento deve ser processado por cada filtro. Assim, cada vez que um dos filtros, que lêem dados provenientes do conjunto de entrada, executam uma função de leitura o GMD é avisado transparentemente, por meio do ADM. A ação do GMD durante a escolha do documento é sempre tentar retornar um documento que esteja localizado na mesma máquina do filtro. Caso não seja possível, escolhe-se um documento aleatoriamente. Essa estratégia visa minimizar o uso de rede, uma vez que o transporte de grandes bases de dados é uma tarefa cara.

#### 4.3.2.2 Armazenador de dados em memória (ADM)

O armazenador de dados em memória é o componente responsável por prover a interface de leitura e escrita de dados entre os filtros da aplicação e o GAPD. Durante a inicialização do sistema de fluxo de trabalho são criadas cópias do componente em

cada uma das máquinas disponíveis no sistema, de acordo com a especificação do usuário.

Cada uma das cópias criadas tem seu limite de memória, também definido pelo usuário, o que limita a quantidade de espaço em memória primária utilizada na gravação de documentos na execução do fluxo de trabalho. Durante a inicialização da aplicação, os filtros são agrupados aos ADM disponíveis. Esse agrupamento faz uma amarração de cada filtro ao ADM que responderá suas requisições durante toda a execução do fluxo. A amarração faz com que os filtros de uma certa máquina $A$, por exemplo, sejam atendidos pelo ADM localizado na mesma. Caso não exista ADM na mesma máquina o filtro é ligado a um ADM qualquer.

Durante a execução, todas as vezes que o primeiro filtro do fluxo executa uma leitura em um dos seus canais de comunicação esta requisição é repassada ao ADM ligado ao mesmo. Uma vez recebida a requisição, existe uma comunicação com o GMD, o qual é responsável pela escolha do documento que será lido. Logo que o ADM recebe os meta-dados que descrevem o documento ele acessa o GAPD, para ler os dados e repassa os mesmos à aplicação.

Além da leitura, o ADM também cuida, quando requisitado, da escrita de dados intermediários enviados através dos canais de comunicação existentes. Esse mecanismo é responsável pela criação de bases de dados distribuídas em tempo de execução para cada um dos canais de comunicação existentes, para então fazer o salvamento de todas as mensagens enviadas entre os filtros. Nessa tarefa, esse componente do sistema faz uso de uma memória própria, permitindo que a aplicação continue sua execução enquanto os são salvos em segundo plano, reduzindo assim o custo com essa operação.

Os dados intermediários salvos em uma execução qualquer podem ser utilizados por outras aplicações sem a necessidade de reexecução do fluxo do trabalho que os gerou. Isso é muito interessante para aplicações cujos resultados são fortemente modificados por seus parâmetros, pois dessa forma pode-se utilizar os resultados parciais como entrada de outro fluxo sem a necessidade de execução de todo o fluxo.

### 4.3.3 Gerenciador de armazenamento persistente de dados (GAPD)

O gerenciador de armazenamento persistente de dados é o componente responsável pelo armazenamento das bases de dados de entrada e de saída dos fluxos de trabalho. Assim esse mecanismo deve ser capaz, dentre outras coisas, de comunicar através da rede com componentes da aplicação espalhados pelo ambiente de execução

e prover armazenamento distribuído de dados escalável e eficiente.

Devido às exigências, optamos pela utilização do Mobius [33, 38] nessa tarefa. Mobius é um sistema projetado para gerenciamento de dados eficientemente e gerenciamento de meta-dados dinamicamente em ambientes distribuídos. Esse sistema provê um conjunto genérico de serviços e protocolos distribuídos para suportar criação, controle de versão e gerenciamento de modelos e instâncias de dados, criação de bases de dados sob demanda, pesquisa de dados em ambientes distribuídos. Esses serviços são constituídos utilizando esquemas XML para representar a definição dos dados e documentos XML para representar as instâncias de dados armazenadas.

O Mobius é composto por três serviços fundamentais: *Global Model Exchange (GME), Data Instance Management(Mako) e Data translation Service(DTS)*, dos quais utilizamos apenas o Mako, o qual é descrito detalhadamente a seguir:

#### 4.3.3.1 Data Instance Management (Mako)

O Mako é responsável por tarefas como criação de bases de dados, pesquisa, recuperação e validação de dados armazenados conforme definido no seu esquema XML. A exposição das operações são feitas através de um conjunto de interfaces bem definidas baseadas em seu protocolo, esses recursos são utilizados através de "requisições XML", feitas por meio arquivo XML enviado via rede ao Mako, o qual define o tipo de operação executada e os parâmetros da mesma. Por exemplo, uma vez criadas, as bases de dados podem ser pesquisadas através do Mako utilizando XPath [7], para tanto é criado um XML de pesquisa que detalha os valores de atributos que atendem a necessidade do usuário. Essa forma de interação cria um padrão, tornando fácil sua a utilização por aplicações executadas em sistemas heterogêneos. Abaixo apresentamos a arquitetura do Mako e seu protocolo de comunicação:

Toda a comunicação entre o Mako e seu clientes é feita através da rede, conforme pode ser visto na figura 4.3, para tanto é utilizado um conjunto de ouvintes com protocolos de comunicação distintos, o que permite que clientes utilizando, por exemplo, TCP, *Globus Security Infrastructure (GSI)*, etc., comuniquem com Mako.

Uma vez que um pacote é recebido, o mesmo é repassado ao roteador de pacotes, que é responsável por identificar, dentre as interfaces suportadas (tipos de operações executadas) qual a responsável por tratar cada pacote. Quando isso é feito, a interface escolhida recebe o pacote, identifica os parâmetros da requisição, processa a mesma e responde diretamente ao cliente que a requisitou.

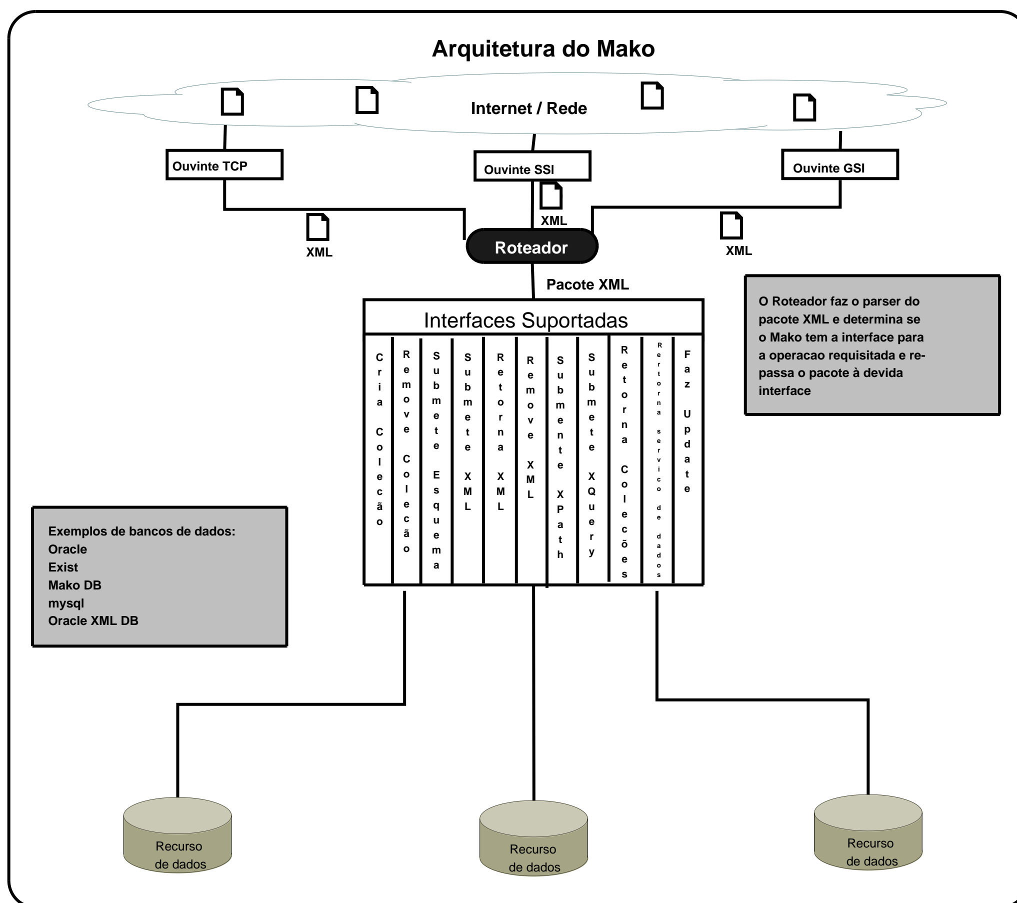



Figura 4.3: Arquitetura do Mako

### 4.3.4 Protocolo de comunicação entre os componentes do sistema de gerenciamento de fluxo de trabalho

Todas aplicações executadas por este sistema de suporte a fluxo de trabalho utilizam Anthill como ambiente de execução, conforme discutido anteriormente na seção 4.3.1. Essas aplicações são implementadas como um conjunto de filtros, que tem o formato similar ao descrito no algoritmo 1. Ou seja, cada filtro recebe um fluxo de dados como entrada, executa algum processamento sobre esses dados e se necessário envia os resultados para o próximo componente do fluxo de trabalho.

Assim, as aplicações executando sobre o Anthill, que utilizem o sistema de fluxo de trabalho, podem fazer uso de todos os recursos do mesmo, tais como, leitura transparente de dados de entrada, controle de escrita de resultados intermediários, escrita de dados de saídas etc. Para que o sistema de controle possa executar suas tarefas existe uma comunicação transparente entre cada componente do fluxo de trabalho (filtros) e o sistema de suporte. Essa comunicação pode ser visualizada

através das figuras 4.4 e 4.5. Na primeira das figuras, descrevemos o processo de leitura de dados utilizando o sistema de suporte a fluxo de trabalho, na segunda descrevemos os passos de uma troca de mensagem entre dois filtros, quando existe armazenamento de resultados parciais, ou seja, os dados enviadas entre filtros.

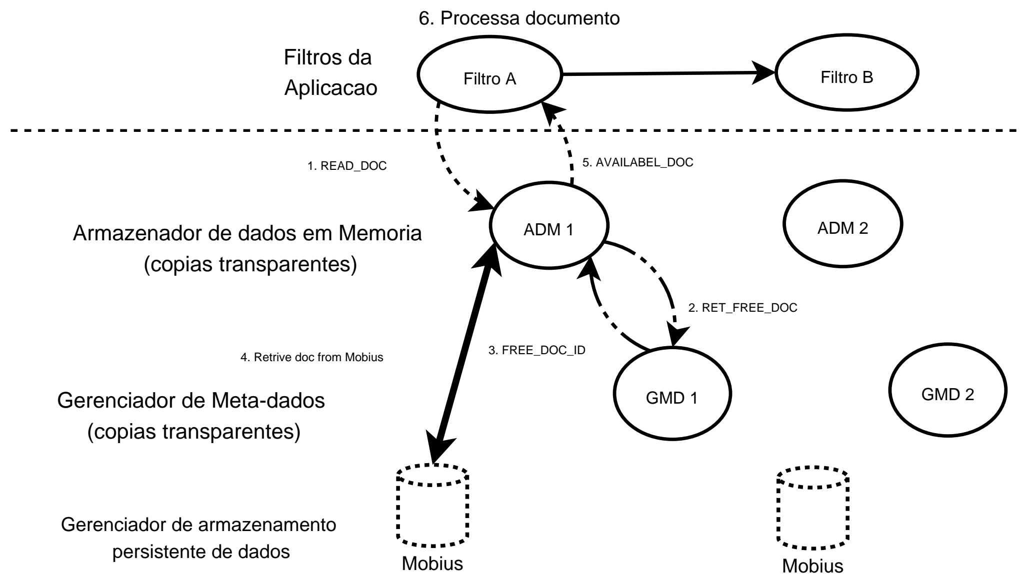



Figura 4.4: Comunicação entre os componentes do sistema quando o Filtro A lê algum dado

Quando o filtro A precisa ler algum dado, conforme pode ser visto na figura 4.4, transparentemente o sistema é avisado, por meio do ADM, através de uma mensagem do tipo *READ_DOC*. O ADM por sua vez comunica-se com o GMD enviando-lhe uma requisição do *RET_FREE_DOC*. Assim que o GMD recebe a última mensagem ele decide qual documento deve ser processado pelo filtro da aplicação que executou a requisição, para em seguida retornar a descrição deste documento ao ADM utilizando uma mensagem *FREE_DOC_ID*, que contém os meta-dados relacionados a este documento. Neste momento o ADM lê o documento do Mobius e repassa o mesmo ao filtro da aplicação que o requisitou, isto é feito nos passos 4 e 5 do protocolo de leitura de dados.

Uma vez que o filtro A tenha terminado o processamento do documento lido e, por exemplo, envie o resultado ao filtro B, se o usuário requisitou o salvamento de resultados parciais, existirá a troca de mensagens apresentada na figura 4.5. Assim, quando o Filtro B recebe uma mensagem, o sistema é avisado dessa ação através da mensagem WRITE_DOC, que contém uma cópia da mensagem recebida por B. Nesse momento existe uma comunicação entre o ADM e o GMD que visa a criação de um novo documento no sistema, que contém a mensagem recebida por

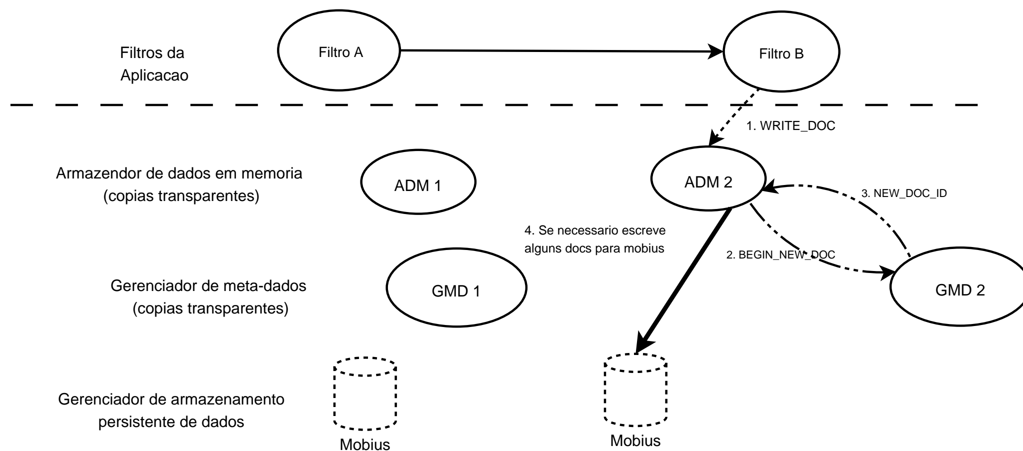



Figura 4.5: Comunicação entre os componentes do sistema quando o Filtro A escreve algum dado para o Filtro B

B. Quando conclui-se essa comunicação, o ADM averigua a existência de espaço em memória para armazenar o documento, havendo espaço o documento é armazenado em memória primária, caso contrário documentos são enviado ao Mobius, utilizando a política FIFO, até que haja espaço suficiente para armazenar o documento criado.

A estratégia de salvamento de dados intermediários é bastante eficiente, uma vez que a tarefa é executada em segundo plano, ou seja, a aplicação não precisa ser interrompida para que os dados sejam salvos. Com isso temos uma sobreposição de processamento e I/O(entrada-e-saída), o que se mostra bastante eficiente para aplicações intensivas em dados.

## 4.4 Sumário

Neste capítulo, apresentamos os principais detalhes da implementação e o funcionamento do sistema de suporte a fluxo de trabalho desenvolvido. Esse sistema foi dividido em três partes principais: repositório de bibliotecas compartilhadas e executáveis, criador de programas e ambiente distribuído de execução.

O primeiro dos componentes do sistema foi desenvolvido com o intuito de atingir dois objetivos: compartilhamento de programas e auxílio na fácil e rápida prototipagem de fluxos de trabalho a partir de programas seqüencias. O criador de programas, por sua vez, é uma ferramenta que permite a incorporação de programas compilados em fluxos de trabalho, sem a necessidade de reescrita de código. E finalmente, desenvolvemos o ambiente distribuído de execução, que é responsável por instanciar e executar fluxos de trabalho em ambientes heterogêneos distribuídos, fazendo a inte-

ração eficiente e escalável entre o os fluxos em execução e um banco de dados XML distribuído.

# Capítulo 5

# Aplicação Exemplo

Neste capítulo, apresentamos a aplicação exemplo [53], utilizada na avaliação desse sistema. Também detalhamos os passos de seu mapeamento em fluxos de trabalho, utilizando as ferramentas desenvolvidas nesse trabalho.

## 5.1 Descrição da aplicação

A aplicação exemplo utiliza imagens microscópicas de alta resolução para estudar mudanças, induzidas por manipulações genéticas, no fenótipo de placenta de ratazanas. O objetivo da aplicação é fazer a segmentação de imagens, que compõem visões 3D de placentas de ratazanas, em regiões correspondentes aos três tipos de tecidos: labirinto, espongioblasto e glicogênio.

Na figura 5.1, temos uma visão completa da aplicação. A mesma foi dividida em 6 estágios, sendo que alguns deles podem ter mais de uma etapa. A seguir, descrevemos a fase de pré-processamento para aquisição das imagens, para então apresentar os estágios da aplicação.

**Pré-processamento:** a placenta é dividida em fatias de aproximadamente $3\mu$, posteriormente cada uma delas são digitalizadas em um *slide* inteiro utilizando um *scanner* do tipo ScanScope CS [62]. As imagens geradas apresentam diferentes cores e texturas para cada um dos três tipos de tecidos que desejamos identificar.

**Separação do plano da frente (Foreground/Background Separation (FG/BG)):** as imagens são convertidas do formato RGB para CMYK e as combinações das cores dos canais são limitadas a valores estipulados pelo usuário. O resultado dessa operação é o plano da frente da imagem.

Diferentes *slides* normalmente apresentam variação na cor, isto ocorre devido a diferenças na espessura, tempo de coloração ou concentração do corante. Assim, antes da separação dos tecidos é preciso fazer uma correção na variação das cores, o que ocorre em três etapas a seguir:

**Teste de cores:** calcula as médias das cores vermelha, verde e azul para cada uma das imagens. Ao fim, faz o cálculo da média dessas cores para todas as imagens, a qual é chamada de média de cores da placenta.

**Imagem referência:** uma imagem é escolhida como base para a normalização das cores de todas as outras imagens. Isso é feito comparando a médias das cores da placenta com a média de cada fatia. Nesse processo, a fatia que tem as médias mais próximas da placenta é escolhida como referência.

**Normalização do histograma:** durante esse processo é gerado um histograma, para cada uma três cores, da imagem referência. Em seguida o restante das imagens é corrigido para os histogramas da imagem referência utilizando as funções padrão de equalização de histogramas existentes em Matlab [70].

**Classificação das cores:** nesse estágio da aplicação cada *pixel* da imagem é qualificado como pertencendo a uma de 8 classes: núcleo escuro, núcleo de densidade mediana, núcleo claro, núcleo extra claro, células de sangue vermelho, citoplasma claro, citoplasma escuro e fundo.

**Interação humana:** o classificador é treinado através da intervenção humana. Primeiramente, especialistas escolhem de forma aleatória algumas imagens e identificam os tecidos, a partir disso são gerados histogramas para as três cores de cada uma das 8 classes descritas anteriormente.

**Classificação de bayes:** para cada *pixel* na imagem são calculadas 8 probabilidades, uma para cada classe, utilizando o histograma gerado na fase anterior. O *pixel* é então classificado em uma das 8 classes utilizando o critério de classificação *Maximum A Posteriori (MAP)*, ou seja, a maior probabilidade indica a classe a qual o *pixel* deve pertencer.

**Segmentação dos tecidos:** as imagens são divididas em regiões de 40x40 *pixels*, para as quais são calculadas 3 probabilidades baseadas na densidade da área, sendo esses valores utilizados na classificação da região em um dos 3 tecidos.

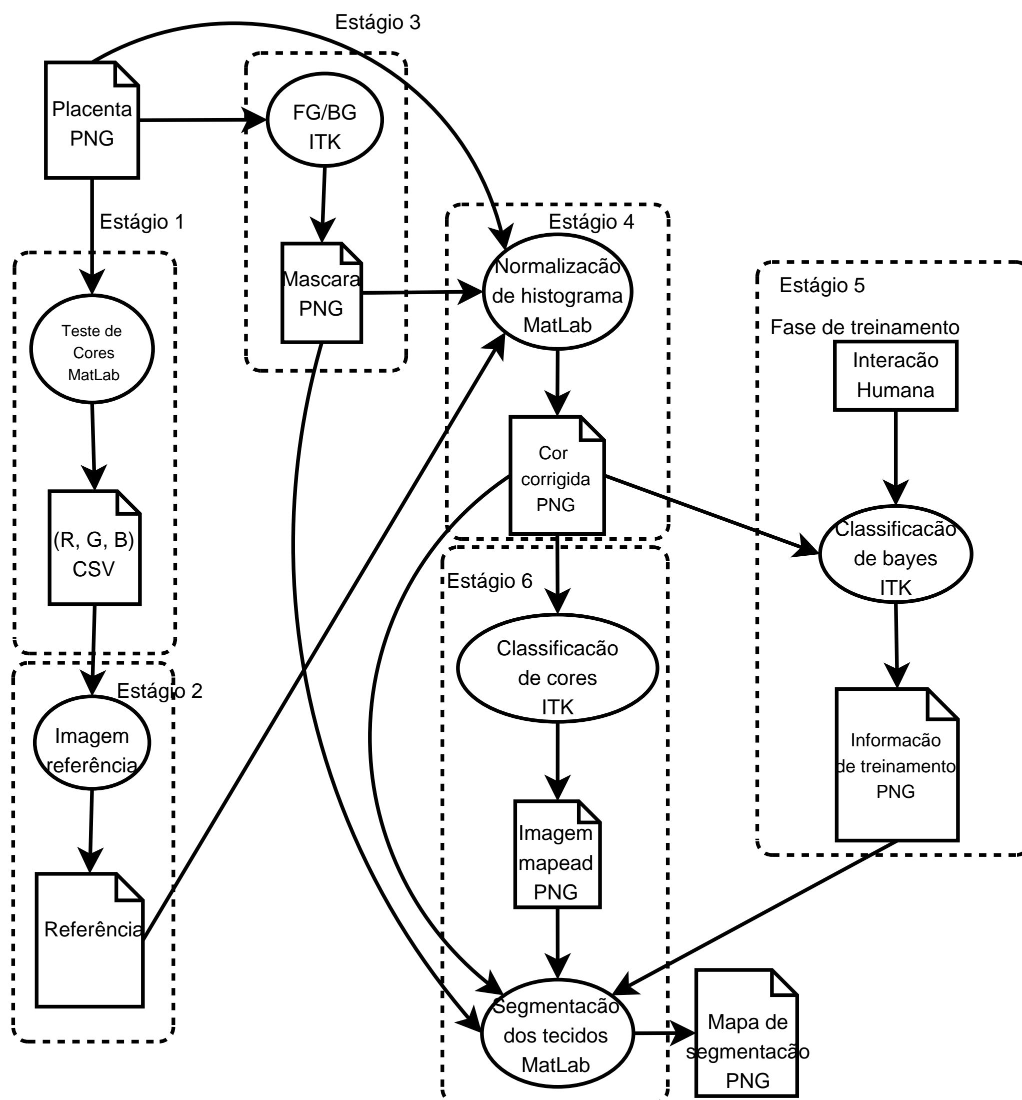



Figura 5.1: Aplicação de segmentação da placenta de ratazana

## 5.2 Mapeamento da aplicação exemplo em fluxo de trabalho

Esse sistema foi desenvolvido para prover fácil incorporação de programas compilados como componentes de um fluxo de trabalho sem a necessidade de modificação de código. Para atingir tal objetivo, desenvolvemos uma ferramenta, descrita na seção 4.2, para criar os códigos Anthill. Os códigos criados contém os filtros do fluxo responsáveis por interagir com programas compilados (bibliotecas compartilhadas e executáveis) sem necessidade de modificação dos programas do usuários. No restante desta seção, descrevemos como é feito o mapeamento da aplicação exemplo

em fluxos de trabalho.

### 5.2.1 Desenvolvimento dos filtros

O principal trabalho na integração de uma aplicação para utilização deste sistema consiste da construção da seção *compiledFilters* do arquivo "Descritor do fluxo", apresentado na seção 4.2.1. A seção *compiledFilters* é utilizada para descrever detalhes sobre os filtros que executam funções provenientes de programas compilados.

O repositório de bibliotecas compartilhadas e executáveis, descrito na seção 4.1, foi criado para reduzir o trabalho associado a tarefa. Através do mesmo usuários pode-se armazenar códigos compilados, assim como, toda a informação relacionada a esses. Desta forma, uma vez armazenados no repositório, a tarefa de incorporar estes programas em um fluxo de trabalho resume-se a copiar as informações a respeito do programa e colar no "Descritor do fluxo" na ordem correta.

A seção *compiledFilters* do "Descritor do fluxo" para o estágio "Normalização de histograma" da aplicação exemplo é apresentada na figura 4.2. Como pode ser visto, precisamos especificar o nome da biblioteca, a função que será utilizada no filtro, as entradas e seus tipos, a informação a respeito das funções de serialização e deserialização utilizadas. Nesse texto nos restringimos ao exemplo apresentado, pois a criação dos outros estágios do fluxo são similares.

### 5.2.2 Desenvolvimento do fluxo

Os fluxos de trabalho suportados por esse sistema são criados utilizando o modelo de programação filtro-fluxo, primeiramente apresentado em Active Disks [1] e estendido para *grid* por DataCuter [8, 27], conforme descrito anteriormente na seção 2.2.1.

Na fase de composição do fluxo de trabalho o usuário precisa especificar quais filtros fazem parte dos fluxos, além das conexões entre eles. A informação é retirada, respectivamente, das seções *placement* e *layout* do "Descritor do fluxo", conforme pode ser visualizado na figura 4.2.

A última interação do usuário consiste da execução do fluxo, para tanto o mesmo deve utilizar um *script*, gerado pelo nosso sistema, com os parâmetros da aplicação e a consulta XML utilizada para identificar o conjunto de dados de entrada.

### 5.2.3 Fluxo de trabalho da aplicação exemplo

Nesta seção, descrevemos o fluxo de trabalho gerado a partir da aplicação exemplo, descrita na seção 5.1. Primeiramente, como pode ser visto na figura 5.1, dividi-

mos a aplicação em 6 estágios e posteriormente mapeamos os 3 computacionalmente mais intensos em fluxos de trabalho, como pode ser visto na figura 5.2.

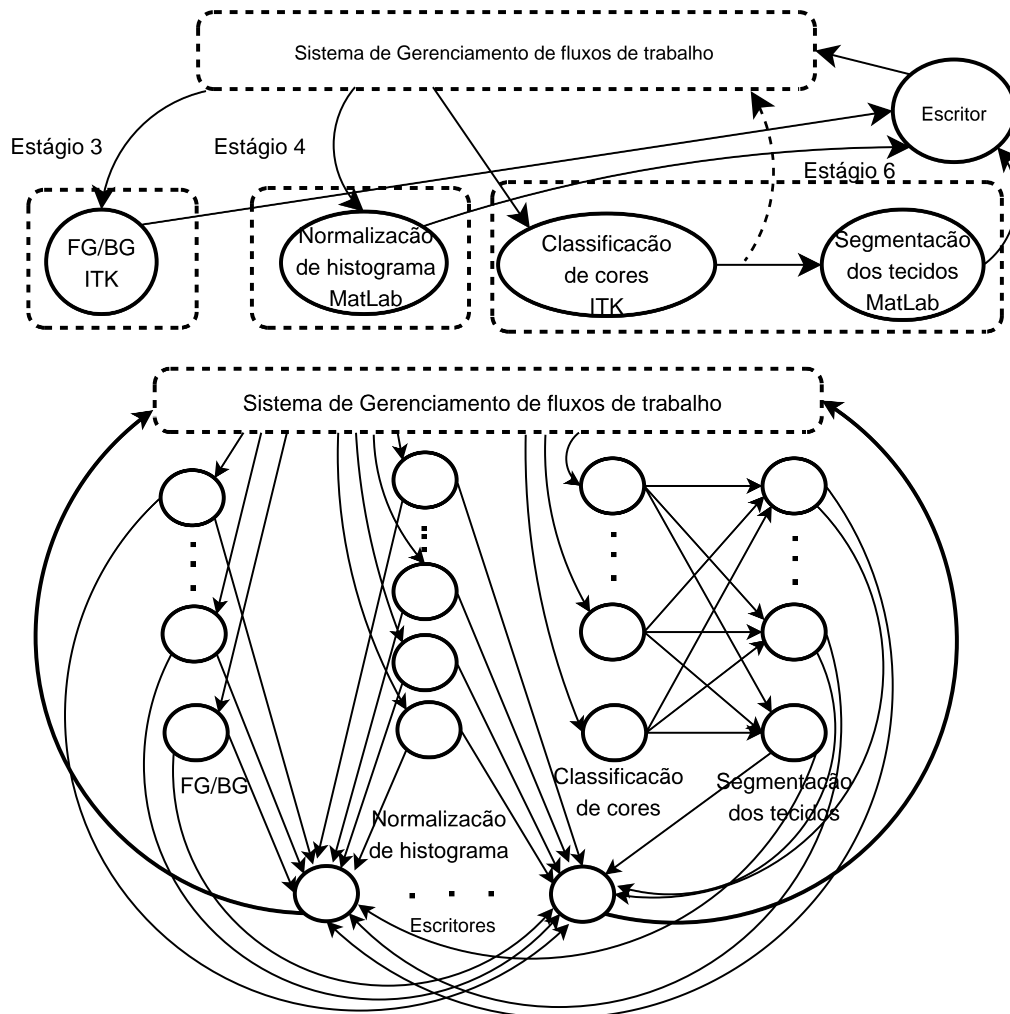



Figura 5.2: Fluxo de trabalho da aplicação da placenta de ratazana

A entrada do estágio 4 da aplicação depende do fim do processamento do estágio 2. Porém os estágios 2 e 3 podem ser executados concorrentemente, dessa forma construímos um sub-fluxo para cada um dos estágios 3 e 4, assim quando 3 é executado o sistema cuida da criação de bases distribuídas e do armazenamento de sua saída, que é utilizada como entrada de 4.

O estágio 6, nosso terceiro sub-fluxo, tem um *stream* entre dois filtros. A figura 5.2 mostra um *stream* e uma seta pontilhada desse para os sistema de gerenciamento de fluxos de trabalho. A seta representa a opção de armazenamento eficiente de resultados parciais, ou seja, de mensagens enviadas entre filtros. Esses resultados podem ser utilizados para reiniciar fluxos de trabalho sem a necessidade de executar

totalmente o mesmo, o que pode ser muito útil para aplicações que são fortemente afetadas por parâmetros.

## 5.3 Sumário

Neste capítulo, foi apresentada a aplicação exemplo [53] utilizada durante a fase de avaliações experimentais desta ferramenta, a qual utiliza imagens microscópicas de alta resolução para estudar mudanças, induzidas por manipulações genéticas, no fenótipo de placenta de ratazanas. Tendo como objetivo a segmentação de imagens, que compõem visões 3D de placentas de ratazanas, em regiões correspondentes ao três tipos de tecidos: labirinto, espongioblasto e glicogênio.

Posteriormente, apresentamos a forma como a aplicação estudada foi dividida em seis estágios, para, a seguir, detalhar o mapeamento de um deles em fluxo de trabalho. Nesse processo, exemplificamos os passos do uso das ferramentas que permitem incorporação de programas compilados como componentes de um fluxo de trabalho, sem a necessidade de modificação de código.

# Capítulo 6

# Experimentos

## 6.1 Configuração dos testes

Os experimentos foram executados utilizando um aglomerado de computadores com 20 máquinas conectadas por meio de um *switch Fast Ethernet*. Cada um dos nodos tem um processador AMD Athlon(tm) 64 Processor 3200+, 2GB de memória primária e sistema operacional Linux 2.6.

Os testes foram executados com a aplicação de segmentação de placenta de ratazanas, mapeada em fluxos de trabalho utilizando as ferramentas disponíveis neste sistema, conforme apresentado anteriormente no capítulo 5.

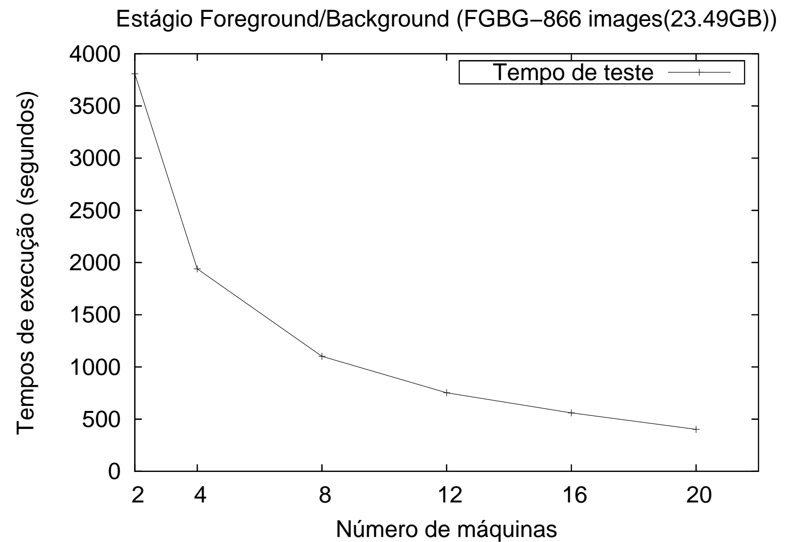
Durante a avaliação do sistema utilizamos uma base de dados com 866 imagens (23.49GB) como entrada, a qual foi criada pela divisão de uma placenta de ratazana em fatias, conforme descrito em [53]. O conjunto de dados foi armazenado no gerenciador de armazenamento persistente de dados. Nos experimentos executamos um Mako em cada uma das máquinas disponíveis e dividimos as imagens igualmente entre os mesmos.

Nos experimentos criamos um cópia de ADM em cada uma das máquinas, uma cópia de GMD em uma delas e uma cópia de cada um dos filtros do fluxo por máquina. As execuções em nossos experimentos utilizam um minímo de duas máquinas, em virtude de não termos uma versão serial que execute sobre todas as imagens de uma vez.

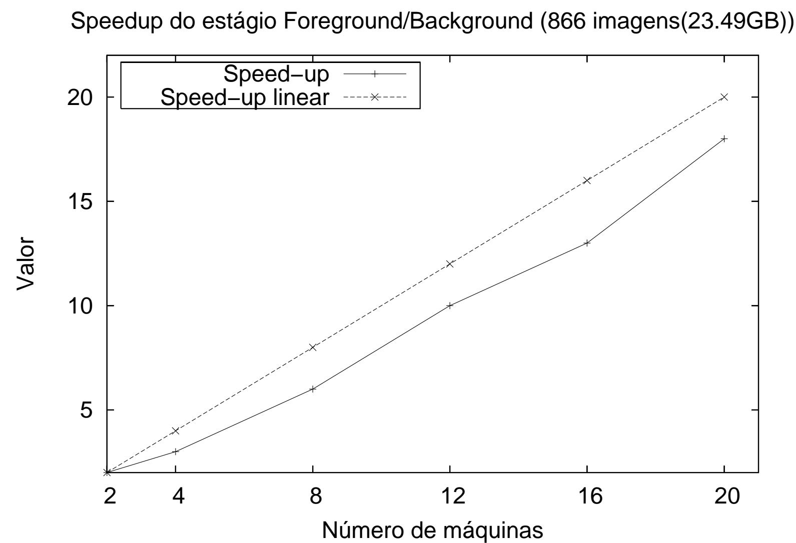
## 6.2 Resultados

Nas Figuras 6.1(a) e 6.1(b), são apresentados os resultados experimentais do estágio FG/BG. Como podemos ver, o tempo de execução utilizando duas máquinas

é de aproximadamente 38.000 segundos e decai quase linearmente com o aumento do número de nodos, o que é refletido nos valores de *speedups* alcançados.




(a)




(b)

Figura 6.1: Estágio FG/BG

As Figuras 6.2(a) e 6.2(b), mostram os resultados obtidos durante a execução do estágio normalização do histograma. Como discutido anteriormente na seção 5, esse componente utiliza as imagens de placenta de ratazanas originais e a máscara como entrada, sendo o tamanho total das máscaras de 488MB. O tempo de execução utilizando duas máquinas é de cerca de 7.000 segundos, mais uma vez o *speedup* obtido durante nossos experimentos é próximo do linear.

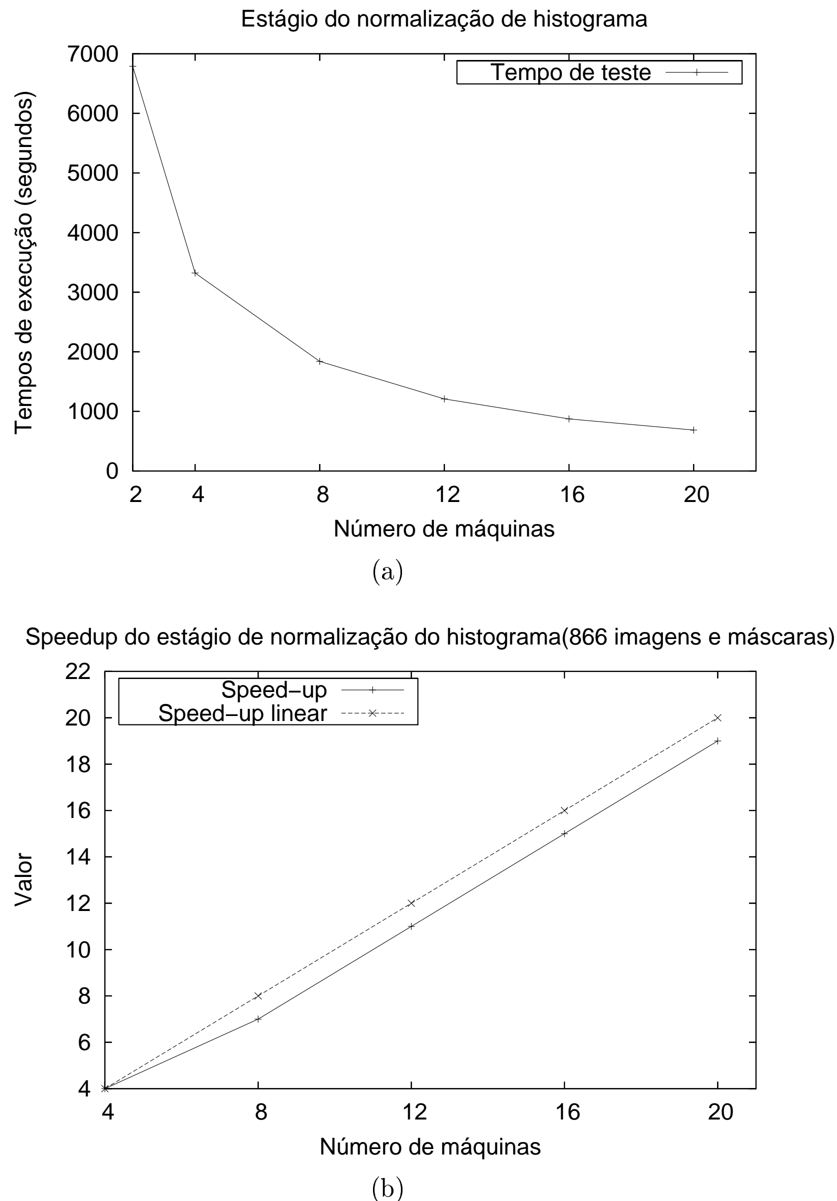
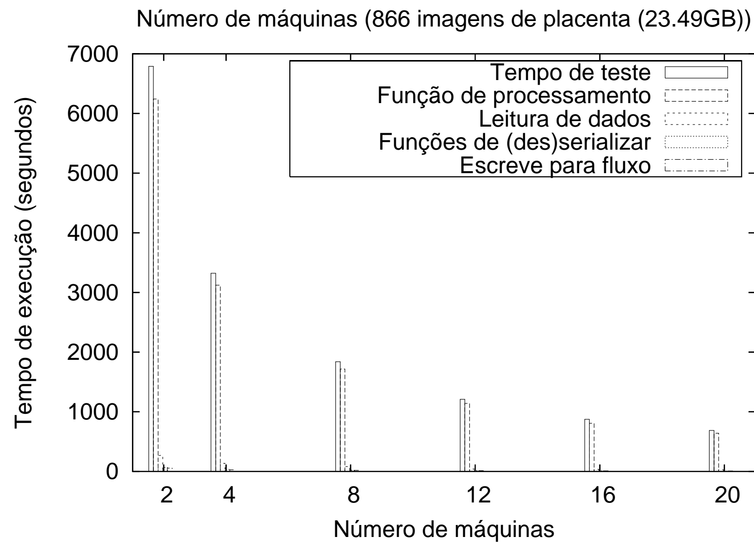


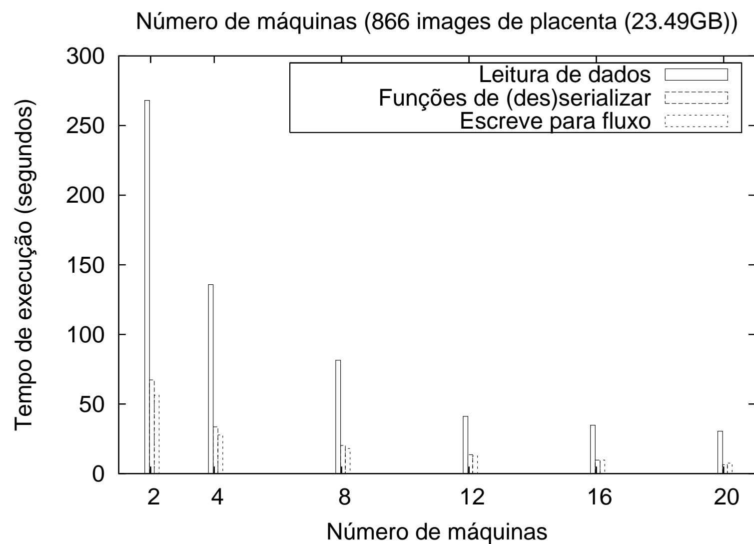

Figura 6.2: Estágio de normalização do histograma

A Figura 6.3(a), mostra os tempos de execução do teste em detalhes, esses tempos foram divididos seguindo o algoritmo 1. Como pode ser visto, o tempo total gasto é dominado pelo tempo gasto na função de processamento, que representa o tempo entre a chamada do executável utilizado nesse estágio e a finalização do mesmo. O *overhead* adicionado pela utilização desse sistema é muito pequeno, o mesmo pode ser calculado pela diferença entre o tempo total de execução e o tempo gasto na função de processamento.

Devido a grande diferença entre o tempo da função de processamento e os demais, é difícil analisar todos os tempos apresentados na Figura 6.3(a). Dessa forma,




(a)




(b)

Figura 6.3: Tempo de execução do estágio de normalização do histograma em delhalhes

adicionamos a Figura 6.3(b), onde são apresentados somente os demais tempos de execução.

Os gastos com "Leitura de dados" representa o tempo para nosso sistema retornar os dados requisitados pela aplicação. Como pode ser visualizado na figura 6.3(b), esse tempo é reduzido à medida que aumentamos o número de máquinas. O resultado mostra a boa escalabilidade das ferramentas de leitura paralela de dados desenvolvidas, uma vez que a redução nos tempos é proporcional ao número de máquinas utilizadas. Como era de se esperar, o valor gasto com deserialização e

serialização vistos na mesma figura, acompanham a mesma tendência. Isso por que essas operações dependem somente da quantidade de dados de entrada, que nesse caso é dividida entre as várias máquinas.

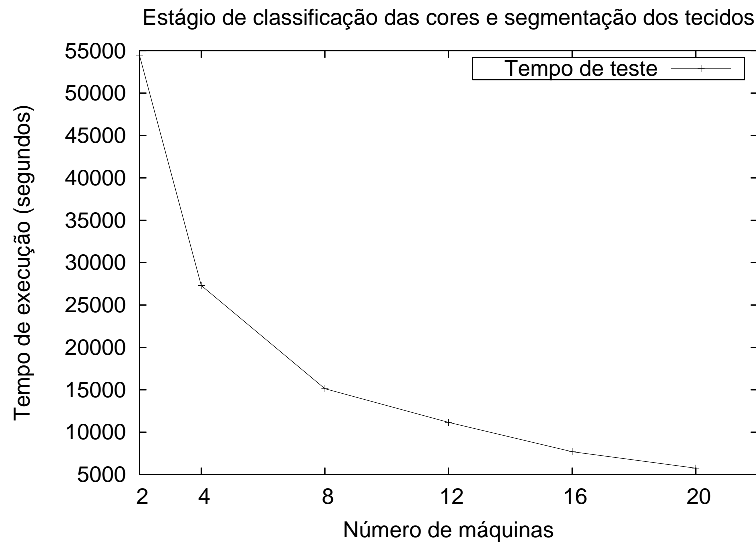
Tabela 6.1: Percentual do tempo em que o GMD esta esperando por requisição

| Número de máquinas | Percentual |
|---|---|
| 2 | 99.98% |
| 8 | 99.80% |
| 16 | 99.61% |

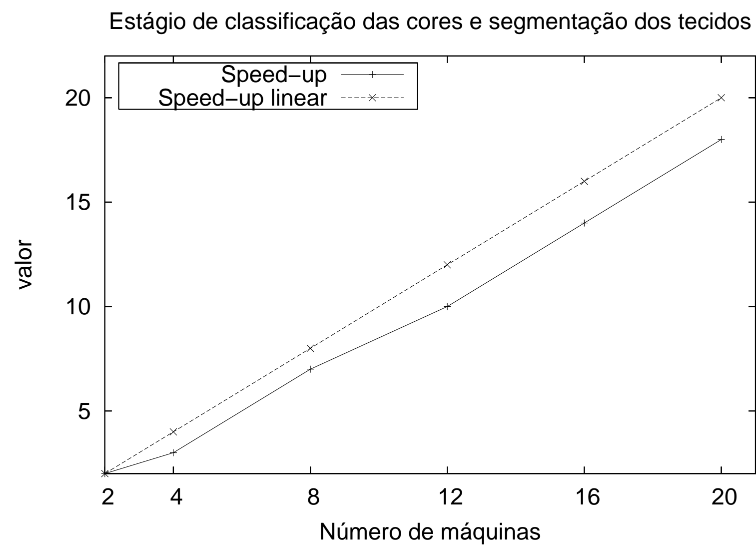
Anteriormente, na Figura 6.3, mostramos a forma com que o tempo de leitura no sistema escala com o aumento do número de máquinas. Entretanto, essa figura não discute em detalhes o nível de saturação do gerenciador de meta-dados (GMD). Dessa forma, na Tabela 6.1, apresentamos o percentual do tempo total de teste que o GMD está aguardando requisições. Como pode ser visto, esse componente passa quase todo o tempo do teste ocioso, ou seja, esperando por requisições do armazenador de dados em memória (ADM). Apesar de termos usado apenas uma cópia desse compontente durante nossos experimentos, os resultados mostram que o componente está distante de se tornar um ponto de contenção.

Na Figura 6.4, mostramos os resultados obtidos durante a execução do fluxo do estágio “Classificação das cores” e “segmentação dos tecidos”. Este é o estágio mais demorado da execução, o tempo gasto com a utilização de duas máquinas é de aproximadamente 55.000 segundos e apresenta *speedup* mais uma vez próximo do linear.

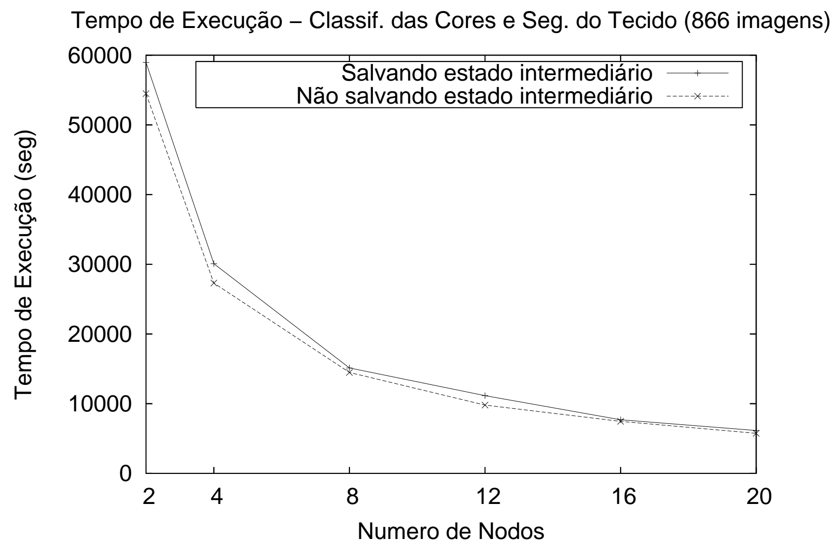
Finalmente, na Figura 6.4(c), apresentamos o resultado do último estágio da aplicação exemplo. Porém, desta vez, comparamos o tempo de execução quando salvamos e não salvamos os resultados parciais que são enviados do passo de “Classificação das cores” para o de “segmentação dos tecidos”. Esses experimentos são utilizados para avaliar a abordagem de armazenamento de resultados parciais desenvolvida. A média da diferença entre os mesmos é de apenas 5%, o que comprova a eficiência da aboradagem.




(a)




(b)




(c)



Figura 6.4: Estágio de classificação de cores e segmentação de tecidos

# Capítulo 7

# Conclusões e trabalhos futuros

## 7.1 Conclusões

Diversos foram os sistemas de gerenciamento de fluxos de trabalho [12, 68, 41, 52, 24] criados nos últimos anos. Inicialmente os mesmos eram capazes de atender às demandas impostas pelos cientistas, entretanto o rápido crescimento do número de bases de dados e o seu grande volume de dados intensificaram o desafio, qual seja, prover os cientistas de ferramentas que sejam capazes de gerenciar e processar tais bases de dados.

Esses desafios foram agravados pelo fato da maioria dos dados científicos não estarem armazenados em bancos de dados, existindo, portanto, dificuldades em manter, acessar, verificar validade, pesquisar, atualizar, controlar acesso multi-usuário e transações. Este problema é agravado pela incapacidade de funcionamento dos sistemas atuais de gerência de fluxo de trabalho, quando precisam lidar com a criação e transporte de grandes volumes de dados pela rede [37].

Neste trabalho, apresentamos um sistema de suporte a fluxos de trabalho para aplicações intensivas em dados para ambiente distribuídos heterogêneos. O sistema de suporte à execução foi construído sobre Anthill [27], consistindo de filtros Anthill criados automaticamente a partir de uma descrição alto nível dos componentes da aplicação do usuário. Os filtros gerados podem executar códigos arbitrários de usuário através de uma interface simples.

Para provermos gerenciamento de dados eficiente, como discutido na seção 3.3, utilizamos Mobius [33, 38] na tarefa de armazenar dados científicos de entrada, criados em tempo de execução e de saída.

Os dois *frameworks* citados anteriormente (Anthill e Mobius) foram integrados, provendo uma interação transparente, eficiente e escalável entre eles, por meio da criação do sistema de gerenciamento de fluxo de trabalho distribuído implementado

como filtros Anthill, discutido na seção 4.3.2. O último componente é responsável pela gerência dos dados que fluem no fluxo de trabalho, assim como leitura e escrita paralela do gerenciador de armazenamento persistente.

A utilização de Anthill no suporte a execução permite a concorrência segura entre os componentes do fluxo, assim, várias instâncias de cada componente podem ser criados durante a execução e diferentes tipos de componentes podem estar executando concorrentemente.

As avaliações experimentais, discutidas em detalhes na seção 6, mostram que o sistema desenvolvido é capaz de executar aplicações sofisticadas, com múltiplos componentes, alcançando *speed-ups* lineares. Os resultados destacam o baixo *overhead* introduzido pelo sistema na execução da aplicação. Os resultados mostram que os custos introduzidos no armazenamento de resultados parciais, ou seja, dados enviados entre componentes de um fluxo, atingem a média de apenas 5%, indicando a eficiência do sistema nessa tarefa.

## 7.2 Trabalhos futuros

Durante a elaboração e execução deste trabalho, vislumbramos várias oportunidades de trabalhos futuros a partir das contribuições e resultados alcançados nesta dissertação. Algumas ds oportunidades são apresentadas nas seções a seguir.

### 7.2.1 Compartilhamento de componentes em tempo de execução

Atualmente, conforme descrito na seção 3, os sistemas de fluxo de trabalho científicos são criados pela composição de componentes simples formando aplicações complexas. Esses, por sua vez, podem ser compartilhados entre aplicações, como pode ser visto na figura 1.2, onde dois fluxos hipotéticos tem a mesma fase de processamento $A$ no início.

A forma de criação de aplicações foi muito importante para a reutilização e compartilhamento de código entre usuários, entretanto, existem outras oportunidades decorrentes da forma como as aplicações são construídas que não foram aproveitadas. Uma delas é o compartilhamento de componentes em tempo de execução, ou seja, aplicações que utilizam os mesmos componentes e estão sendo executadas no mesmo conjunto de dados podem processar a parte comum do fluxo apenas um vez, enviando a saída para o próximo componente no fluxo de cada aplicação, conforme pode ser visto na figura 7.1.

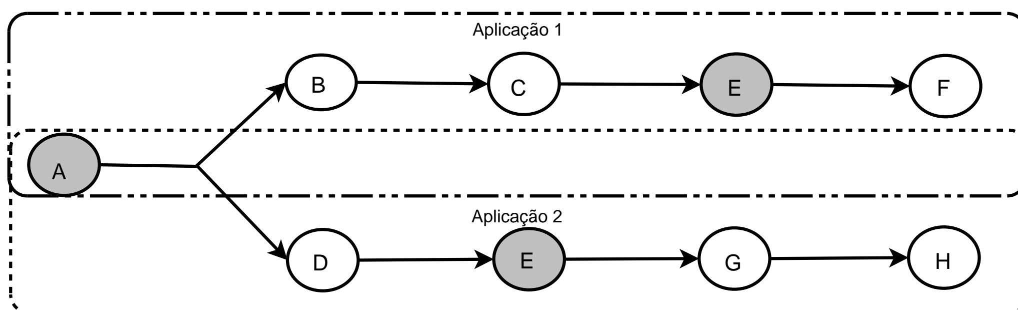



Figura 7.1: Aplicações compartilhando componentes em tempo de execução

Esse trabalho ainda poderia ser beneficiado de escalonadores que identificassem aplicações disparadas para execução com partes comuns, fazendo com que fossem executadas concorrentemente sem canibalização.

### 7.2.2 Utilização de cache semântico

Conforme discutido na seção 3.2, a reutilização de dados parciais é importante para prover melhor desempenho na reexecução de aplicações. Na figura 7.2, apresentamos uma alternativa de reuso de dados, onde estes dados são utilizados de várias formas.

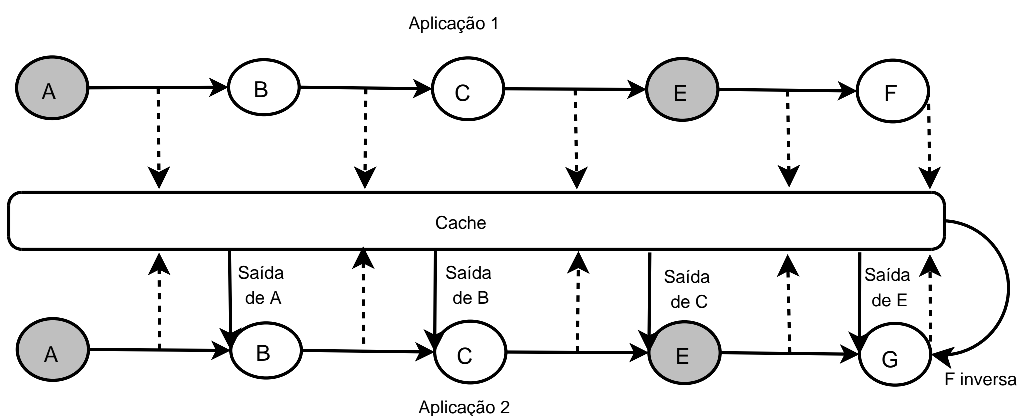



Figura 7.2: Aplicações de técnicas de cache semântico

Todas a possíveis soluções são baseadas na utilização de *cache* semântico entre os vários componentes do fluxo. Dado que a aplicação 1 tenha sido executada, ver figura 7.2, quais seriam as oportunidades reuso de dados criados, na execução da aplicação 2?

- A primeira forma de melhorar o tempo de execução é pela leitura de dados intermediários do *cache*, assim não precisamos executar o fluxo todo para processar os dados. Na figura 7.2, mostramos como a aplicação 2 poderia utilizar dados intermediários gerados pela aplicação 1 em cada um de seus estágios.

- Na segunda possível abordagem, aplica-se uma função que desfaz certa operação aplicada pela aplicação 1 não utilizada pela aplicação 2, por exemplo, como pode ser visto na figura 7.2, aos dados de saída de 1 é aplicada a função inversa de F, sendo a saída da última utilizada pelo componente G da aplicação 2.

### 7.2.3 Geração automática de fluxos de trabalho para integração de dados

Atualmente, durante o processo de pesquisa, cientistas acessam dados de diversas fontes, em especial nas ciências ligadas à área biológica, onde múltiplas bases de dados e repositórios podem prover informação relevante sobre saúde. Porém, a exploração de diversas fontes de dados requer a utilização de um grande leque de técnicas de integração de dados.

Grande parte das dificuldades referentes a esse assunto decorrem de problemas de armazenamento de dados com mesma semântica utilizando tipos diferentes. Na figura 7.3 mostramos um exemplo deste problema, onde temos as bases de dados 1 e 2 armazenando dados de semântica X com tipos Y e Z diferentes.

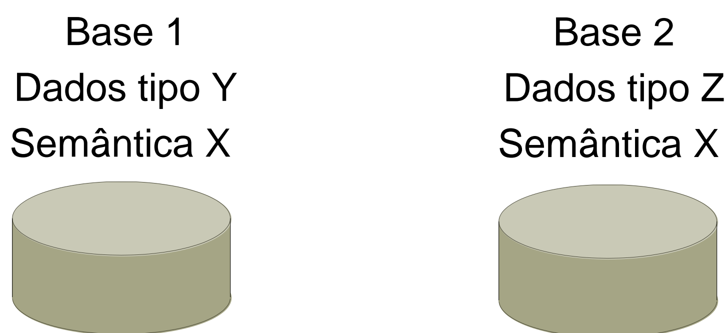



Figura 7.3: Dados de mesma semântica armazenados com tipos diferentes

Dado o cenário em questão, o problema seria criar um fluxo de trabalho capaz de processar durante a mesma execução dados das duas bases, armazenados usando tipos diferentes.

Figura 7.4: Repositório

Se existem funções para processarem os dados com tipos diferentes, conforme pode ser visto na figura 7.4, onde a "função A" tem duas versões com entrada de tipos Y e Z. Uma solução seria oferecer ao usuário a possibilidade de descrição em alto nível do fluxo de trabalho, como mostrado na figura 7.5, sendo que a partir dela o sistema poderia interagir com o repositório e verificar a existência das funções adequadas e montar o fluxo de trabalho.

```
<programDescriptor>
  <hostDec>
    <host name="mymachine.dcc.ufmg.br"/>
  </hostDec>
  <placement>
    <filter name="funcaoA" location="Repositorio" instances="2">
       <input="Base1"/>
       <input="Base2">
    <filter name="Writer" libName="WriterFilter.so" instances="2"/>
  <placement/>
  <layout>
    <stream>
      <from filter="funcaoA" port="funcaoAOut" policy="roundRobin"/>
      <to filter="Writer" port="writerInput" />
    </stream>
  </layout>
</programDescriptor>
```

Figura 7.5: Configuração de aplicações em alto nível para integração de dados

Uma vez executado, o componente receberia o dado e escolheria qual função deveria ser executada baseado no tipo de entrada. Esse funcionamento é análogo a ligação tardia, onde em tempo de execução decide-se função qual deve ser executada.

# Referências Bibliográficas

[1] A. Acharya, M. Usysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *Eighth International Conference on Architectural Support for Programming Languages and Operations Systems (ASPLOS VIII)*, pages 81–91, Oct 1998.

[2] Anastassia Ailamaki, Yannis E. Ioannidis, and Miron Livny. Scientific workflow management by database management. In *Statistical and Scientific Database Management*, pages 190–199, 1998.

[3] George S. Almasi, Calin Cascaval, and David A. Padua. Matmarks: A shared memory environment for matlab programming. In *Proceedings of International Symposium on High Performance Distributed Computing (HPDC)*, 1999.

[4] Gustavo Alonso, Berthold Reinwald, and C. Mohan. Distributed data management in workflow environments. In *Proc. 7th International Workshop on Research Issues in Data Engineering (RIDE'97), Birmingham, England, April 1997.* .

[5] Henrique Andrade, Tahsin Kurc, Alan Sussman, and Joel Saltz. Efficient execution of multiple query workloads in data analysis applications. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 53–53, New York, NY, EUA, 2001. ACM Press.

[6] Troy Baer and Pete Wyckoff. A parallel I/O mechanism for distributed systems. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 63–69, San Diego, CA, September 2004. IEEE Computer Society Press.

[7] Anders Berglund, Scott Boag, Don Chamberlim, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. Xml path language (xpath). In *World Wide Web Consortium (W3C)*, August 2003.

[8] M. Beynon, C. Chang, U. Catalyurek, T. Kurc, A. Sussman, H. Andrade, R. Ferreira, and J. Saltz. Processing large-scale multidimensional data in parallel and distributed environments. In *Parallel Computing*, 2002.

[9] Michael Beynon, Chialin Chang, Umit Catalyurek, Tahsin Kurc, Alan Sussman, Henrique Andrade, Renato Ferreira, and Joel Saltz. Processing large-scale multi-dimensional data in parallel and distributed environments. *Parallel Computing*, 28(5):827–859, 2002.

[10] Michael Beynon, Renato Ferreira, Tahsin M. Kurc, Alan Sussman, and Joel H. Saltz. Datacutter: Middleware for filtering very large scientific datasets on archival storage systems. In *IEEE Symposium on Mass Storage Systems*, pages 119–134, 2000.

[11] Michael D. Beynon, Tahsin Kurc, Umit Catalyurek, Chialin Chang, Alan Sussman, and Joel Saltz. Distributed processing of very large datasets with datacutter. *Parallel Computing*, 27(11):1457–1478, Novembro 2001.

[12] S. Bowers and B. ascher. An ontology-driven framework for data transformation in scientific workflows. In *The 1st Intl. Workshop on Data Integration in the Life Sciences (DILS)*, volume 2994, pages 1–16, 2004.

[13] Shawn Bowers, Timothy McPhillips, Bertram Ludaescher, Shirley Cohen, and Susan Davidson. A model for user-oriented data provenance in pipelined scientific workflows. In *International Provenance and Annotation Workshop (IPAW'06)*, Chicago, Illinois, USA, May 2006.

[14] Dongarra J. Casanova H. Netsolve: A network enabled server for solving computational science problems. In *International Journal of Supercomputer*, pages 212–223, 1997.

[15] Maria Cláudia Cavalcanti, Rafael Targino, Fernanda Baião, Shaila C., Paulo M. Bisch, Paulo F. Pires, Maria Luiza M. Campos, and Marta Mattoso. Managing structural genomic workflows using web services. *Data Knowl. Eng.*, 53(1):45–74, 2005.

[16] CERN. Large hadron collider (lhc) - http://www.interactions.org/lhc/, 2006.

[17] L. Chen, N.R. Shadbolt, C. Goble, F. Tao, S.J. Cox, C. Puleston, and P.R. Smart. Towards a knowledge-based approach to semantic service composition. In *2nd International Semantic Web Conference*, October 2003.
A knowledge-based approach to workflow composition.

[18] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and Tuecke S. The data grid: Towards an architecture for distributed management and analysis of large scientific data sets. In *J. Network and Computer Applications*, pages 187–200, 2001.

[19] A. Choy, R. Edelman. Parallel matlab: doing it right. In *Proceedings of the IEEE*, pages 331–341, Feb 2005.

[20] Condor. The directed acyclic graph manager, 2003.

[21] Bruno Coutinho. Desempenho e disponibilidade em sistemas distribuídos em larga escala. Tese de mestrado, Universidade Federal de Minas Gerais, Brasil, 2005.

[22] S. Davidson, C. Hara, and L. Popa. Querying an object-oriented databases using cpl. In *Brazilian Symposium on Databases (SBBD)*, 1997.

[23] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation (OSDI'04)*, pages 137–150, Dezembro 2004.

[24] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda. Mapping abstract complex workflows onto grid environments. *In Journal of Grid Computing*, pages 25–39, 2003.

[25] Thomas Duessel, Norbert Eicker, Florin Isaila, Thomas Lippert, Thomas Moschny, Hartmut Neff, Klaus Schilling, and Walter Tichy. Fast parallel i/o on cluster computers, 2003.

[26] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[27] Renato Ferreira, Wagner Meira Jr., Dorgival Guedes, Lucia Drummond, Bruno Coutinho, George Teodoro, Tulio Tavares, Renata Araujo, and Guilherme Ferreira. Anthill:a scalable run-time environment for data mining applications. In *Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD)*, 2005.

[28] Layna Fischer. Workflow handbook: Future strategies, 2005.

[29] Ian Foster, Jens Vockler, Michael Wilde, and Yong Zhao. Chimera: Avirtual data system for representing, querying, and automating data derivation. In *Proceedings of The 14th International Conference on Scientific and Statistical Database Management (SSDBM'02)*, 2002.

[30] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing.* MIT Press, Cambridge, MA, EUA, 1994.

[31] Rafael C. Gonzalez, Richard E. Woods, and Steven L. Eddins. *Digital Image Processing Using MATLAB.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.

[32] Steven D. Gribble. Robustness in complex systems. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 21–26, May 2001.

[33] Shannon Hastings, Stephen Langella, Scott Oster, and Joel Saltz. Distributed data management and integration framework: The mobius project. In *Global Grid Forum 11 (GGF11) Semantic Grid Applications Workshop*, pages 20 – 38. IEEE Computer Society, 2004.

[34] Shannon Hastings, Scott Oster, Stephen Langella, Tahsin M. Kurc, Tony Pan, Umit V. Catalyurek, and Joel H. Saltz. A grid-based image archival and analysis system. In *Journal of Americam Medical Informatics Association*, Dec 2005.

[35] T.; Langella S.; Catalyurek U.; Pan T.; Saltz J. Hastings, S.; Kurc. Image processing for the grid: a toolkit for building grid-enabled image processing applications. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, pages 36 – 43, Washington, DC, USA, 2003. IEEE Computer Society.

[36] Jihie Kim, Marc Spraragen, and Yolanda Gil. An intelligent assistant for interactive workflow composition. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interface*, pages 125–131, New York, NY, USA, 2004. ACM Press.

[37] George Kola, Tevfik Kosar, Jaime Frey, and Miron Livny. Disc: A system for distributed data intensive scientific computing. In *First Workshop on Real, Large Distributed Systems (WORLDS'04)*, San Francisco, CA, December 2004.

[38] Stephen Langella. Distributed data management and integration, the mobius project. In *The Global Grid Forum (GGF11) Semantic Grid Applications Workshop, Honolulu, HI*, Jun 2004.

[39] Mikko Laukkanen and Heikki Helin. Composing workflows of semantic web services. In *Proceedings of the Workshop on Web-Services and Agent-based Engineering*, 2003.

[40] K Likos, A. Burger, and R Bladock. A scalable mediator approach to process large biomedical 3-d images. In *Information Technology in Biomedicine*, pages 21–26, Sept. 2004.

[41] B. Ludascher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, and Y. Zhao J. Tao. Scientific workflow management and the kepler system, to appear, 2005.

[42] B Ludascher, K Lin, S Bowers, E Jaeger-Frank, B Brodaric, and Chaitan Baru. Managing scientific data: From data integration to scientific workflows. In *GSA Today, Special Issue on Geoinformatics*, 2005.

[43] Antonioletti M, Atkinson M, and Malaika S. Grid data service specification. In *Global Grid Forum*, 2003.

[44] Funk A. Manolakos E. Rapid prototyping of component-based distributed image processing applications using javaports. In *Workshop on Computer-Aided Medical Image Analysis, CenSSIS Research and Industrial Collaboration.*, 2002.

[45] Timothy McPhillips, Shawn Bowers, and Bertram Ludaescher. Collection-oriented scientific workflows for integrating and analyzing biological data. In *3rd International Workshop on Data Integration in the Life Sciences (DILS'06)*, Hinxton, UK, July 2006. European Bioinformatics Institute (EBI).

[46] Daniel A. Menascé and Virgilio A. F. Almeida. *Planejamento de Capacidade Para Serviços na WEB*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[47] C. Moler. Why there isn't a parallel matlab. http://www.mathworks.com/company/newsletter, 1995.

[48] J. Montagnat, V. Breton, and I. E. Magnin. Using grid technologies to face medical image analysis challenges. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 588, Washington, DC, USA, 2003. IEEE Computer Society.

[49] J. Paul Morrison. Flow-based programming. In *1st International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 25–29, Berlin, March 1996.

[50] Sivaramakrishnan Narayanan, Umit Catalyurek, Tahsin Kurc, Xi Zhang, and Joel Saltz. Applying database support for large scale data driven science in distributed environments. In *GRID '03: Proceedings of the Fourth International Workshop on Grid Computing*, pages 141–148, Washington, DC, EUA, 2003. IEEE Computer Society Press.

[51] Sivaramakrishnan Narayanan, Tahsin Kurc, Umit Catalyurek, and Joel Saltz. Database support for data-driven scientific applications in the grid. *Parallel Processing Letters*, 13(2):245–271, 2003.

[52] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, Pocock, A. Wipat, and P. Li. Taverna:a tool for the composition and enactment of bioinformatics workflows. In *GGF10*, volume 2994, pages 3045–3054, Berlin, Germany, 2004.

[53] Tony C. Pan and Kun Huang. Virtual mouse placenta: Tissue layer segmentation. *Proceedings of the 27th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC2005)*, Sep 2005.

[54] Rui Pinho, Thiago Teixeira, Yuri Faria, George Teodoro, Tulio Tavares, Dorgival Guedes, Renato Ferreira, and Wagner Meira Jr. Análise e entendimento de desempenho com a ferramenta antfarm. In *VI Workshop em Sistemas Computacionais de Alto Desempenho (WSCAD-2005)*, 2005.

[55] US Berkeley PTOLEMYII project, Deparment of EECS. http://ptolemy.eecs.berkeley.edu/ptolemyII/, 2004.

[56] Matheus Ribeiro. Um ambiente de programação para sistemas distribuídos com grandes volumes de dados. Dissertação de mestrado, Universidade Federa de Minas Gerais, Brasil, 2005.

[57] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic composition of web services using semantic descriptions. In *Web Services: Modeling, Architecture and Infrastructure workshop in conjunction with ICEIS2003*, 2003.

[58] Sleepycat-Software-Inc. Berkeley db xml. http://www.sleepycat.com/products/bdbxml.html, 2006.

[59] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: The Complete Reference.* MIT Press, Cambridge, MA, EUA, 1995.

[60] Matthew Spencer, Renato Ferreira, Michael Beynon, Tahsin Kurc, Umit Catalyurek, Alan Sussman, and Joel Saltz. Executing multiple pipelined data analysis operations in the grid. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Los Alamitos, CA, EUA, 2002. IEEE Computer Society Press.

[61] H. Stockinger, A. Samar, W. Allcock, I. Foster, K. Holtman, and B. Tierney. File and object replication in data grids. In *10th IEEE Intl. Symp. on High Performance Distributed Computing*, pages 76–86. IEEE Press, 2001.

[62] Aperio Technologies. Aperio - http://www.aperio.com, 2006.

[63] T. Livny M. Thain, D.; Tannenbaum. Distributed computing in practice: The condor experience. concurrency and computation: Practice and experience., 2004.

[64] Dave Turner. Scientific applications on workstation clusters vs supercomputers. In *APS March 2000 Meeting*, Minneapolis, MN, March 2000.

[65] Adriano Veloso, Wagner Meira Jr., Renato Ferreira, Dorgival Guedes Neto, and Srinivasan Parthasarathy. Asynchronous and anticipatory filter-stream based parallel algorithm for frequent itemset mining. In *Knowledge Discovery in Databases: PKDD 2004, 8th European Conference on Principles and Practice of Knowledge Discovery in Databases, Proceedings*, volume 3202 of *Lecture Notes in Computer Science*, pages 422–433, Pisa, Italy, Setembro 2004. Springer-Verlag GmbH.

[66] Johnston W, Guojun J, and Hoo G. Distributed environments for large data-objects: Broadband networks and a new view of high performance, large scale storage-based applications. In *Internetworking*, Nara, Japan, September 1996.

[67] L. Weng, G. Agrawal, U. Catalyurek, T. Kurc, M. T, K2, N. S, and J. Saltz. An approach for automatic data virtualization. In *IEEE Symposium on High-Performance Distributed Computing*, 2004.

[68] M. Weske, G. Vossen, and C. Medeiros. Scientific workflow management: Wasa architecture and applications. In *Fachbericht Angewandte Mathematik und Informatik 03/96-I*, 1996.

[69] Wikipedia, the free encyclopedia. ENIAC. http://www.wikipedia.org/wiki/Eniac, 2006.

[70] The Math Works. Matlab. http://www.mathworks.com/products/matlab, 2006.

[71] Ran Zheng, Hai Jin, Qin Zhang, Ying Li, and Jian Chen. Ipge: Image processing grid environment using components and workflow techniques. In *GCC*, pages 671–678, 2004.

# Exemplo de código gerado

## Estágio de FG/BG

Nesta seção, apresentamos exemplos de código gerado para estágio de separação do filtro da frente. Na seções 7.2.3, 7.2.3 e 7.2.3, apresentamos o arquivo de configuração, o código do filtro e o Makefile criado criado para o primeiro estágio da aplicação exemplo.

### Arquivo de configuração

```
<?xml version="1.0"?>
<config>
  <hostdec>
    <host name="uzi01">
      <resource name="cache1"/>
    </host>
  </hostdec>
  <placement>
    <filter name="filterA" libname="filterA.so" instances="1">
      <instance demands="cache1"/>
    </filter>
    <filter name="writter" libname="writter.so" instances="1">
      <instance demands="cache1"/>
    </filter>
  </placement>
  <layout>
    <stream>
      <from filter="filterA" port="output" />
      <to filter="writter" port="input"/>
    </stream>
    </layout>
    <compiledFilters>
      <compiledFilter name="filterA" firstFilter="yes">
        <executable name="/home/speed/george/bmi/toolkit/matlab/samples/
removeBackground" numoutputs="1" numinputoutpus="0" numinputs="1">
```

```
          <argument argtype="string" inputtype="msg" msgindexin = "1"
deserializefunction="charArrayToFileWithoutSize"
deserializelibname="libexec-deserialize.so" order="1"/>

          <argument argtype="string" serializefunction="fileToDCBuffer"
serializelibname="libexec-serialize.so" order="2" msgIndexOut="1"/>

        </executable>
       </compiledFilter>
     </compiledFilter>
</config>
```

## Código do filtro A

```
#include <iostream.h>
#include "api_cpp.h"
#include "api_cpp_cache.h"

#include "exec-serialize.h"
#include "exec-deserialize.h"

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <iostream>
#include <stdio.h>

class filterA : public DCFilterCache
{

public:
  filterA(){}
  ~filterA(){}

  int init();
  int process();
  int fini();
};

int filterA::init() {
  return 0;
```

```
}

int filterA::process() {

  // Input arguments
  string arg1; // message argument

  // Output arguments
  char *aux2 = (char *) malloc(50);
  sprintf(aux2, "/tmp/doc.out.%d.2.png", getpid());
  string arg2 = aux2;

  // Receives the program arguments
  DCBuffer* buffer = this->get_init_filter_broadcast();

  // System arguments: <first collection> <first stream type> <mobius hosts file>
  string firstCollection, firstStream, mobiusHostFiles;
  buffer->unpack("sss", &firstCollection, &firstStream, &mobiusHostFiles);

  // Choose the correct cache transparent copy to communicate.
  createLinkToCache(dcmpi_get_hostname());


  while (1) {
    string mobiusXML;

    // Receives message from cache
    DCBuffer * inBuffer = read(firstStream, "filterAToCache", "cacheTofilterA", &mobiusXML,
true);

    if(inBuffer == NULL) break;
    arg1 = charArrayToFileWithoutSize(inBuffer);

    // MatLab Function
    int ret, pid, status;
    if ((pid = fork()) < 0) {
      perror("fork");
      exit(1);
    }
    if (pid == 0) {
      char *cmd[] = { "./removeBackground", (char *) arg1.c_str(), (char *) arg2.c_str()};
      ret = execv ("/home/speed/george/bmi/toolkit/matlab/samples/removeBackground", cmd);

      perror("Not done\n");
      exit(1);
```

```
    }
#ifdef VOID_INST
  dsInstEnterState(PROCESSING_FUNCTION);
#endif

    if (waitpid(pid, &status, 0) == -1) {
      perror("wait interrupted");
      exit(1);
    }
    if (WIFEXITED(status)) {
      cout << "Child process completed. Status is " <<  WEXITSTATUS(status) << endl;
    }else {
      perror("wait failed");
      exit(1);
    }
#ifdef VOID_INST
  dsInstLeaveState();
#endif

    // Deserializes arguments that will be sent
    DCBuffer *outBuffer = new DCBuffer();
    fileToDCBuffer(outBuffer, arg1);
    fileToDCBuffer(outBuffer, arg2);

    // Sends output arguments to next filter
    write(outBuffer, "output", "filterAToCache", "cacheTofilterA", mobiusXML);

    // Consume input buffer.
    inBuffer->consume();
    outBuffer->consume();
  }

  return 0;
}

int filterA::fini() {

        return 0;
}

provide1(filterA)
```

## Makefile

```
AH_API_CPP = ${ANTHILL_ROOT}/api_c++
```

```
AH_SRC = ${ANTHILL_ROOT}

MT_LIB = ${TOOLKIT_ROOT}/libs/lib
MT_LIBINLUCDE = ${TOOLKIT_ROOT}/libs/include
MT_INCLUDE = ${TOOLKIT_ROOT}/libs/matlabinclude

CACHE_ROOT = ${TOOLKIT_ROOT}/cache

CC = g++ -g -Wall

CFLAGS = -Wno-deprecated -I${AH_API_CPP} -I${AH_SRC} -I${AH_SRC}/FilterDev
-I${PVM_ROOT}/include -I${MT_LIBINLUCDE} -I${MT_INCLUDE} -I${CACHE_ROOT}
-I${TOOLKIT_ROOT}/makoConnect -I${TOOLKIT_ROOT}/libs/OSCVR/include/OSCVR/
-I${TOOLKIT_ROOT}/libs/OSCVR/include/ -DVOID_INST #-DUSE_CACHE -DBMI_FT

CLIBS = -L${AH_API_CPP} -ldscpp -L${AH_SRC} -lds -lexpat -L. -L${MT_LIB}
-lexec-serialize -lexec-deserialize -L${TOOLKIT_ROOT}/makoConnect/

all: main filterA.so writter.so

filterA.so: filterA.cpp
  ${CC} ${CFLAGS} ${CLIBS} -fPIC -shared -o filterA.so filterA.cpp

writter.so: writter.cpp
  ${CC} ${CFLAGS} ${CLIBS} -fPIC -shared -o writter.so writter.cpp -lMakoConnect

main: matlab_filters_main.cpp
  cp ${AH_API_CPP}/DCBuffer.o .
  cp ${AH_API_CPP}/api_util.o .
  ${CC} ${CFLAGS} -L${AH_SRC} -lds DCBuffer.o api_util.o matlab_filters_main.cpp -o main

clean:
  rm -f *.o main filterA.so writter.so
```