MATHEUS FERREIRA RIBEIRO

UM AMBIENTE DE PROGRAMAÇÃO PARA SISTEMAS DISTRIBUÍDOS COM GRANDES VOLUMES DE DADOS

Belo Horizonte
Junho de 2005

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UM AMBIENTE DE PROGRAMAÇÃO PARA SISTEMAS DISTRIBUÍDOS COM GRANDES VOLUMES DE DADOS

Proposta de 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.

MATHEUS FERREIRA RIBEIRO

Belo Horizonte
Junho de 2005

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

Um Ambiente de Programação para Sistemas Distribuídos com Grandes Volumes de Dados

MATHEUS FERREIRA RIBEIRO

Proposta de dissertação defendida e aprovada pela banca examinadora constituída por:

Prof. RENATO ANTÔNIO CELSO FERREIRA – Orientador
Universidade Federal de Minas Gerais

Prof. DORGIVAL OLAVO GUEDES
Universidade Federal de Minas Gerais

Prof. WAGNER MEIRA JÚNIOR
Universidade Federal de Minas Gerais

Prof. WALFREDO CIRNE
Universidade Federal de Campina Grande

Belo Horizonte, Junho de 2005

Resumo

"Formigueiro" é uma implementação de um ambiente de programação paralelo, baseado no modelo fluxo-filtro. O fluxo-filtro é um modelo comprovadamente eficiente para o processamento de grandes volumes de dados e funciona como uma linha de processamento, onde cada estágio processa o dado e o modela para o estágio seguinte, seguindo o fluxo da linha. Estágios sobrecarregados podem ser replicados, obtendo-se uma linha uniforme, onde o fluxo de dados é constante.

Neste trabalho são apresentadas as principais características do modelo e do ambiente de programação: filtros e cópias transparentes, divisão dos dados por cópias, comunicação por fluxo, estados distribuídos através de fluxo rotulado e utilização do sistema. Também serão apresentados resultados de duas aplicações desenvolvidas sobre o "Formigueiro", mostrando que o sistema é escalável, podendo alcançar desempenhos superlineares.

A implementação e o modelo de programação do "Formigueiro" são baseados no DataCutter, um sistema desenvolvido pela University of Maryland e Ohio State University. A adição de novas características, como o fluxo rotulado e a interface de divisão de tarefas, permite que uma categoria maior de aplicações possa ser trabalhada no ambiente.

Abstract

“Anthill” is an implementation of a parallel programming environment, based on filter-stream model. Filter-stream is an efficient model for processing large ammounts of data and works as a network pipeline, where each stage processes data and transmits the result for the next one. Stages heavily loaded can be replicated, balancing the load among the pipeline components and having a constant data flow.

In this work, the features of the programming model and the environment are presented: filters and transparent copies, data parallelism interface, stream communication, distributed state through labeled streams and system usage. Results of two real applications that make use of our system will also be presented, showing “Anthill” can reach superlinear speedups.

“Anthill” implementation is based on DataCutter, a system developed by University of Maryland and the Ohio State University. The addition of new features, like labeled stream and work division interface, makes it possible for a greater range of applications to be run in the environment.

Para minha avó Adélia, pelos 80 anos de amor a todos ao seu redor.

Agradecimentos

Foi um longo caminho e, certamente, um caminho difícil. Muitas pessoas me acompanharam nessa trajetória e seria até um desrespeito não mencioná-las aqui.

Em primeiro lugar, a família: meu pai, Antônio, minha mãe, Águida, meus irmãos Marcelo e Gustavo. Todos contribuíram para o desfecho desse trabalho se não diretamente, através da educação, do caráter e das lições de vida.

Os amigos merecem atenção especial: o pessoal do Speed, de duras horas de trabalho mas também de muita diversão - Crocha, Juliano, Barroca, Andrec, Hugo, Ismael, Zeniel e ele, O Coutos; o Ehnbev, time de Futsal campeão do DCC - Marco Oreia, Helder, Flip, Velinho, Garrafa, Guerrinha, Serginho e André; os Qscara, grandes amigos de futebol, bate-papo e viagens- Thiaguinho, Rafinha, Salum, Filizzola, Tulio, Luiz; os amigos de RPG - Tipão, Bimbola, FC, Felipinho e Bill. E seria impossível deixar de fora dessa lista a Clara, que foi uma grande força nessa difícil etapa da minha vida e a Bia, pela revisão do texto. A lista é grande! Com certeza deixei alguns de fora, mas seria impossível falar de todos aqui.

Por último, gostaria de agradecer aos professores que participaram de minha vida acadêmica: Sérgio Campos, Berthier, Wagner Meira, Dorgival, Renato, Umit, Saltz e Tahsin. Em especial ao Renato, por me ajudar a realizar o sonho de estudar fora e ver como as coisas funcionam por lá (e como funcionam!), além de ter sido sempre um grande amigo.

A todos vocês, meu sincero obrigado.

Sumário

1 Introdução **1**

1.1 Contribuições 2

1.2 Organização do Trabalho 3

2 Trabalhos Relacionados **4**

2.1 PVM 5

2.2 MPI 6

2.3 Condor 7

2.4 Globus 8

2.5 DataCutter 9

2.6 MapReduce 10

2.7 Resumo 12

3 Modelo de Programação **13**

3.1 Discos Ativos e Modelo Baseado em Fluxos 13

3.2 DataCutter e o Modelo Filtro-Fluxo 14

3.3 Formigueiro e Manutenção de Estados Distribuídos 15

3.4 Resumo 18

4 Formigueiro **19**

4.1 Pré-Requisitos 19

4.2 Cenário 20

4.3 Trabalho 20

4.4 Interface do Programador 21

4.4.1 Gerente 21

4.4.2 Arquivo de Configuração XML 22

4.4.3 Filtros 24

4.5 Configuração Dinâmica 29

4.6 Reinício 30

4.7 Resumo 30

5 Aplicações **31**
5.1 Identificação de Conjuntos de Itens Frequentes em Bases de Dados Distribuídas . . . 31
5.1.1 Descrição do Problema . . . 31
5.1.2 Algoritmo . . . 31
5.1.3 Filtros . . . 32
5.1.4 O Problema da Terminação . . . 33
5.2 Segmentação da Camada do Labirinto em Placentas . . . 34
5.2.1 Particionamento das Imagens . . . 36
5.2.2 Filtro RedFinder (Identificador de Pixels Vermelhos) . . . 37
5.2.3 Filtro Counter (Contadores de Vizinhos) . . . 37
5.2.4 Filtro PCA (Análise do Componente Principal) . . . 41
5.3 Resumo . . . 44

6 Resultados **45**
6.1 Identificação de Conjuntos de Itens Frequentes em Bases de Dados Distribuídas . . . 45
6.2 Segmentação da Camada do Labirinto em Placentas . . . 46
6.3 Resumo . . . 52

7 Conclusão **53**
7.1 Trabalhos Futuros . . . 55
7.1.1 Multiplataforma . . . 55
7.1.2 Fluxo acumulado . . . 55
7.1.3 Integração . . . 56
7.1.4 Múltiplos trabalhos . . . 57
7.1.5 Configuração Dinâmica com Manutenção de Estados . . . 58
7.1.6 Tolerância a falhas . . . 60
7.1.7 Geração Automática de Programas Paralelos . . . 61

Referências Bibliográficas **62**

Lista de Figuras

3.1 Linha de processamento dos discos ativos. 14
3.2 Linha de processamento em fluxo de dados entre filtros. 15
3.3 Grafo direcionado do programa para contagem de cidades do estado próximas à capital. . 16
3.4 Cada mensagem do programa poderá ter mais de um destino. 17

4.1 Um cenário hipotético para execução de um programa Formigueiro. 20
4.2 Uma possível distribuição de filtros para o cenário da Figura 4.1. 20
4.3 Representação gráfica do XML apresentado na Listagem 4.2 24
4.4 Instâncias de um filtro leitor realizando paralelismo de dados 25
4.5 Representação gráfica do filtro divP4, mostrado na Listagem 4.3 27

5.1 Grafo direcionado do programa de identificação de conjuntos de itens frequentes. 32
5.2 Erro na contagem global de ocorrências do conjunto de itens A. 33
5.3 Todas as ocorrências de A são enviadas à mesma instância do Somador, gerando resultado correto (A é frequente). 34
5.4 Um *slide* de placenta digitalizado e normalizado, com o labirinto em destaque. 35
5.5 O grafo direcionado da aplicação da placenta. 35
5.6 Particionamento das imagens para a aplicação. 36
5.7 Distribuição de blocos entre os contadores. 38
5.8 Pixels nativos e de borda: os pontos A, B e D são nativos ao bloco 1. A região sombreada representa a borda dos blocos, portanto o ponto C é nativo ao bloco 5 e de borda para os blocos 1, 2 e 6, enquanto B é de borda ao bloco 2 e D é de borda para 2, 5 e 6. 40
5.9 Agregação de histogramas, suavização e corte. 41
5.10 Estágios do PCA: pontos recebidos, rotação no novo eixo e corte por limiar (com rotação de volta para o eixo original). 41
5.11 Execução do PCA distribuído. 43

6.1 Dados após cada estágio da execução. 47
6.2 Tempos dos RedFinders. 48
6.3 Tempos dos Contadores nas três configurações. 48
6.4 Tempo dos PCAs, nas três configurações. 49
6.5 Distribuição dos filtros entre as máquinas. 49
6.6 Tempo dos PCAs executados separados dos Contadores. 50

6.7 Resultado global dos experimentos. 52

7.1 Fluxo acumulado no Formigueiro. 56
7.2 Múltiplos trabalhos na linha, sincronizados por filtro. 58
7.3 Múltiplos trabalhos na linha, execução assíncrona. 59

Capítulo 1

Introdução

Atualmente, a programação paralela tem se mostrado como uma das áreas mais importantes na computação. Com o desenvolvimento da tecnologia de sensores, dados de variados tipos são coletados em quantidades enormes. Para se obterem informações significativas sobre esses dados são necessárias técnicas avançadas de análise, cujo custo de processamento em apenas uma máquina é, normalmente, alto. Programar paralelamente significa criar um programa que utilize diversos recursos simultaneamente na obtenção dessas informações, o que resultará em custos menores.

Existem vários graus de paralelismo e, nos computadores pessoais, eles estão presentes de maneira transparente ao programador. O acesso direto a memória, por exemplo, permite que dispositivos de disco acessem simultaneamente a memória. O foco deste trabalho foi o paralelismo no nível de máquinas, isto é, vários nodos independentes se comunicando via rede. Aplicações em execução nesse ambiente normalmente lidam com grandes volumes de dados e realizam, por exemplo, gerenciamento de reservas de petróleo através de imagens geradas para simulações [28] ou detecção de distúrbios em imagens de ressonância magnética [21].

A criação de programas paralelos não é uma tarefa trivial. Vários fatores, que não estão presentes em um programa serial, devem ser tratados pelo programador: o acesso a dados compartilhados de maneira síncrona para evitar condições de corrida (*race conditions*), os mecanismos de prevenção de esperas infinitas (*starvation*) e o travamento completo (*deadlocks*) da aplicação. Além disso, a utilização de mais recursos aumenta também o número de possíveis pontos de falha (por exemplo, cabos de rede danificados podem gerar falhas de comunicação). Lidar com essas dificuldades acarreta custos adicionais (*overheads*) ao programa, reduzindo seu desempenho. O objetivo de um programador paralelo é aproveitar seus recursos computacionais de maneira a sobrepassar os custos adicionais e obter um ganho de desempenho que justifique a utilização da execução paralela. Para tal, existem diversos sistemas de apoio ao programador, que diminuem as dificuldades do ambiente distribuído e permitem que o programador foque em seu problema.

Neste texto, será apresentada uma implementação de um ambiente de programação paralela denominado Formigueiro. Trata-se de uma evolução do DataCutter [3, 23], sistema desenvolvido pela University of Maryland e a Ohio State University. O Formigueiro é ao mesmo tempo um modelo de programação e um ambiente de execução auxíliar ao programador paralelo. O modelo de programação do Formigueiro é denominado filtro-fluxo (*filter stream*), por fazer com que o dado flua através de

diferentes programas (filtros) paralelamente, como em uma linha de processamento (*pipeline*). Esse modelo permite a execução de programas paralelos comutativos, com manutenção de estados distribuídos. O Formigueiro oferece algumas facilidades para se lidar com um ambiente distribuído, como a detecção de falhas e a comunicação confiável e ainda apresenta um modelo simples ao programador, que deve apenas dividir seu programa em tarefas distintas (filtros) e definir a maneira como estas se comunicam (fluxos).

1.1 Contribuições

As principais contribuições do Formigueiro são: estender um modelo comprovadamente eficiente de programação (filtro-fluxo) e simplificar a interface de programação do usuário, permitindo uma maneira mais fácil de se criar programas paralelos eficientes.

O Formigueiro evoluiu do modelo de programação do DataCutter, ao permitir a execução de aplicações com estado de uma maneira transparente. Isso só foi possível devido à adição de uma nova política de comunicação dos fluxos, controlada pelo usuário. No DataCutter, as políticas de comunicação consideram apenas o balanceamento de carga, mas não a semântica da aplicação. No Formigueiro, esse controle é passado para o usuário de maneira simples e modular, o que permite que uma nova classe de aplicações seja ser trabalhada: aplicações de redução com manutenção de estados. O impacto do controle do usuário é pequeno, como será mostrado nos resultados obtidos com experimentos.

O Formigueiro possui também mecanismos de instrumentação com pouco impacto na aplicação, permitindo ao programador detectar os pontos críticos de seu programa. A instrumentação é feita através de funções que mudam o estado geral da aplicação (como em uma máquina de estados). Ao fim da execução, o tempo gasto em cada um dos estados, inclusive os do sistema, são informados em arquivos para cada um dos filtros.

A criação de programas para o Formigueiro foi dividida em três partes bem definidas: a aplicação gerente, o arquivo de configuração e os filtros. A aplicação gerente é quem dispara os processos, enviando as unidades de trabalho aos filtros. O arquivo de configuração, feito em XML, determina quais máquinas serão utilizadas durante a execução, quais recursos estão disponíveis, quantas instâncias de filtros serão utilizadas e como os filtros se comunicam. Os filtros são os programas que processam os dados. Eles se comunicam através de fluxos contituídos por uma sequência de *buffers* de tamanhos pré-definidos. Suas funções são mapear dados de entrada para um formato de saída. A ligação de vários filtros é denominada *layout* e pode ser representada por um grafo direcionado.

O sistema se mostrou eficiente para lidar com grandes volumes de dados em aplicações reais. Em alguns casos, conseguiu-se desempenho superlinear, o que significa que não apenas os custos adicionais da paralelização foram compensados, como também mostra uma utilização eficiente dos recursos do sistema.

1.2 Organização do Trabalho

No Capítulo 2, serão apresentadas as principais características de alguns sistemas de auxílio ao programador paralelo já existentes. No Capítulo 3 será explicado o modelo filtro-fluxo e no Capítulo 4, o Formigueiro será mostrado em detalhes: o modelo de três componentes (configuração, gerente e filtros), a utilização do PVM, o sistema de detecção de falhas e de reinício, o fluxo rotulado permitindo a execução de programas com estados distribuídos e a instrumentação.

Duas aplicações que utilizaram o Formigueiro serão apresentadas no Capítulo 5. A primeira é uma aplicação de mineração de dados para identificar conjuntos de itens frequentes em bases grandes. A segunda aplicação se enquadra na categoria de segmentação de imagens, com o objetivo de segmentar a camada do labirinto no tecido de placentas de ratos. Os resultados dessas aplicações serão mostrados no Capítulo 6, onde será possível verificar que o sistema escala e pode atingir desempenhos superlineares.

O texto será concluído com um resumo das principais características do Formigueiro e possíveis extensões ao sistema. O Capítulo 7 mostrará como o Formigueiro poderá evoluir mais, sugerindo a criação de um sistema multiplataforma, a integração com o Condor para gerenciamento de requisitos de filtros, o ganho de desempenho através de múltiplos trabalhos simultâneos na linha, o fluxo acumulado e a geração automática de programas.

Capítulo 2

Trabalhos Relacionados

Muitas vezes, o programador paralelo desvia o foco de seu trabalho para lidar com problemas referentes à programação paralela. Um sistema auxiliar paralelo, que forneça serviços de transmissão confiável de dados e notificação de falhas, primitivas de sincronização e um modelo bem definido de programação, permite que o programador foque sua atenção em seu problema, aumentando a eficiência e produtividade do programador.

Alguns esforços já foram realizados para fornecer um ambiente mais amigável de programação. A necessidade vem da complexidade do ambiente de execução, distribuído e heterogêneo. Falhas podem ocorrer em diversos pontos, em qualquer momento da execução. Além disso, os recursos heterogênoes apresentam uma dificuldade ainda maior, devido a características como ordem dos bytes (*endianess*) em palavras, diferentes tamanhos de dados (32bit vs 64bit) e interfaces completamente distintas entre sistemas operacionais (em sistemas tipos UNIX, por exemplo, os dispositivos são arquivos com atributos especiais, enquanto no Windows eles são tratados separadamente do sistema de arquivos).

Atualmente, existem alguns desses sistemas voltados para diferentes propósitos. MPI (*Message Passing Interface*) [20] e PVM (*Parallel Virtual Machine*) [24] são os mais conhecidos. Ambos fornecem uma abstração para programas do tipo MIMD (múltiplas instruções, múltiplos dados). Outros sistemas são especializados em armazenar e recuperar, de maneira eficiente, dados de tamanho grande, como o Storm [28], ou armazenar dados para futura interação com bases de dados semanticamente semelhantes mas, sintaticamente diferentes, como o Mobius [18]. Outras abstrações ao modelo de programação são fornecidas pelo DataCutter [3, 23], um sistema que define um modelo de programação tipo linha de processamento (pipeline) e o MapReduce ([5]), que criou um modelo de programação simples para aplicações de redução generalizada.

O gerenciamento de grades computacionais (Grids [11]) é igualmente importante para a programação paralela. Grids são recursos heterogêneos interconectados por uma ou mais redes, parcialmente disponíveis para execução de aplicações distribuídas em larga escala. O Globus [10] é um sistema de gerenciamento de Grids, dotado de serviços como o de conexão segura a máquinas espalhadas por diferentes redes. O Condor [25] é um escalonador de processos distribuídos em redes heterogêneas, que pode funcionar juntamente com o Globus (Condor-G [12]).

Como exemplo de como esses serviços podem estar interligados, uma aplicação DataCutter, uti-

lizando MPI ou PVM para comunicação, pode ser disparada por um sistema de loteamento como o Condor, que por sua vez distribui os processos utilizando-se do sistema de segurança do Globus, que obtém informações sobre a utilização da rede através de um *plugin* tipo NWS (*Network Weather Service*) [30].

O Formigueiro é uma evolução do modelo de programação do Datacutter. O sistema se baseia em PVM para disparo de processos, comunicação e detecção de falhas em tempo de execução. Desde o princípio, a implementação foi voltada para um ambiente de configuração dinâmico e tolerante a falhas. Além disso, o Formigueiro é provido de um mecanismo de manutenção de estados distribuídos denominado fluxo rotulado. Neste capítulo, serão detalhados alguns trabalhos relacionados à programação paralela e, em especial, ao Formigueiro.

2.1 PVM

O PVM foi desenvolvido no final dos anos 80 e início da década de 90, quando a programação paralela começava a se mostrar como a solução mais promissora às exigências computacionais. Nessa época, houve discussões relativas ao modelo computacional (algoritmos paralelos e arquitetura de máquinas) mas muito pouco se falou sobre o ambiente de desenvolvimento de programas. O PVM preencheu esse vazio, oferecendo uma série de primitivas de programação que puderam ser incorporadas às linguagens procedurais existentes.

O sistema PVM é um ambiente de programação para desenvolvimento e execução de programas paralelos. Ele define o conceito de máquina virtual, ou seja, vários processos participantes da execução de um programa PVM, possivelmente em diferentes máquinas (heterogêneas), funcionam como se estivessem em uma máquina paralela. O PVM fornece recursos de programação que emulam características de uma máquina como: o envio de sinais (uma falha de segmentação em um dos processos participantes pode ser vista em todos os outros processos da máquina virtual), a notificação de falhas (como um desligamento de uma máquina participante), o controle dos processos remotos através de identificadores (semelhantes aos PIDs do UNIX), a comunicação confiável e o redirecionamento das saída padrão e de erro para um arquivo de registro.

A aplicação é híbrida, já que em um nível alto existe o paralelismo funcional (MIMD), mas cada parte do programa pode fazer uso de paralelismo de dados (SIMD, *single instruction, multiple data*). Isso ocorre porque cada processo PVM pode executar tarefas completamente distintas de um programa ou tarefas iguais sobre dados diferentes. A diferenciação sobre o que fazer é feita por parâmetros de execução enviados aos processos disparados.

O PVM é totalmente distribuído, suportando um conjunto dinâmico do máquinas e supondo apenas que uma comunicação não confiável (entrega de datagramas não sequenciais, mas com integridade de dados) esteja disponível para o sistema. Essas características fazem dele um sistema altamente flexível. Processos podem ser iniciados estaticamente (manualmente ou através de uma interface de administração) ou em tempo de execução. O controle é realizado por um processo de fundo (*daemon*), o pvmd, em cada nodo participante da máquina virtual. Todos os pvmds têm um conhecimento global da situação corrente. O controle distribuído aumenta a tolerância a falhas e reduz o gargalo de um sistema de mensagens centralizado.

Originalmente, o PVM usava comunicação ponto-a-ponto, com o número de processos participantes no envio de mensagens de difusão dobrando a cada rodada de envio de mensagens (*recursive doubling*). Versões mais recentes alteraram isso para o envio de mensagens por um único processo, piorando um pouco o desempenho, mas permitindo uma implementação mais simples de tolerância a falhas. Rotinas de seções críticas são baseadas em algoritmos de consenso distribuídos, utilizando o ranking do processo na máquina virtual para decidir entre operações conflitantes. Todos os pvmds têm informações sobre os recursos utilizados na máquina virtual.

Há também uma tentativa de balanceamento de carga, onde processos são iniciados nas máquinas menos carregadas. Mas o usuário tem a liberdade de sobrepassar tal decisão e escolher uma máquina qualquer, independente de sua carga. O algoritmo usado é alternação simples, mas quando um nodo tem a carga acima de determinado patamar, o PVM tenta disparar o processo em outro. Isso tudo resulta em um balanceamento de carga simples, mas flexível.

Todas estas características fazem do PVM uma boa escolha para aplicações paralelas, especialmente no cenário corrente, onde clusters se tornam mais comuns, tomando o lugar de processadores paralelos maciços (supercomputadores capazes de utilizar dezenas, ou centenas, de processadores simultaneamente) e onde há uma tendência de se encontrar recursos heterogêneos, muitas vezes disponíveis e não utilizados.

2.2 MPI

De 1990 a 1995, um grupo de pesquisadores de grandes corporações e instituições de governo dos Estados Unidos e Europa discutiram sobre um sistema padrão de passagem de mensagem. Este trabalho culminou com o MPI (*Message Passing Interface*). O MPI original foi estendido algumas vezes, resultando nas versões 1.1, 1.2 e 2.0. Além das tradicionais correções, as extensões adicionaram novos tipos de dados, entrada e saída de dados paralelos, comunicação unilateral e processos dinâmicos.

MPI é uma interface para programas MIMD, definindo rotinas de comunicação ponto-a-ponto e coletivas. O arcabouço não é tolerante a falhas, mas dispõe de comunicação confiável.

As rotinas incluem três diferentes tipos de envio: padrão, pronto e síncrono, em modos blocante ou não. O envio padrão (*standard*) é aquele que a rotina retorna imediatamente; o pronto (*ready*) requer uma recepção que esteja à espera do envio e o síncrono retorna apenas quando a recepção acontecer. No modo blocante, a chamada retorna assim que a mensagem tiver sido passada ao sistema de envio, enquanto que no não blocante a chamada retorna um objeto que permite a realização de consultas sobre a situação da operação. Por outro lado, existem apenas dois tipos de rotina de recepção, blocante e não blocante. A blocante retorna assim que o dado é recebido e a não blocante retorna um objeto que permite a consulta sobre a chegada dos dados, análogo ao envio. Por último, há também rotinas de comunicação coletiva, que devem ser chamadas por todos os processos envolvidos na operação coletiva. Os dados podem ser divididos entre um grupo, e mais tarde, unidos novamente. Chamadas de difusão (*broadcast*) também são possíveis. Todas as chamadas coletivas são blocantes.

O MPI definiu um conceito de grupo de processos, unindo-os para resolverem problemas similares (paralelismo de dados) mas com diferentes grupos resolvendo problemas diferentes ao mesmo tempo (paralelismo de tarefas). Um processo pode estar em mais de um grupo ao mesmo tempo, mas no MPI

original, a participação em grupos, uma vez definida, não pode ser alterada. Entretanto, na versão 2.0, essa limitação foi derrubada e o modelo estático da primeira versão se tornou um caso especial da segunda, fazendo que programas antigos continuassem compatíveis. A versão 2.0 adiciona ainda a criação de processos e mecanismos de estabelecimento de comunicação entre processos existentes e novos.

Outra adição à versão mais recente é a habilidade de realização de acesso remoto à memória (RMA). Todos os processos podem abrir uma janela em sua memória, permitindo que outros processos leiam e escrevam nessa área. O MPI acumula o máximo de chamadas RMA, por motivos de eficiência, realizando chamadas de sincronização periodicamente, ou quando necessário. Há três tipos de operações disponíveis: *get*, *put* e *accumulate*. A *get* lê os dados remotos, a *put* escreve os dados remotamente e a *accumulate* soma os dados remotos com a versão local. Há tambem chamadas de atualização local e remota. Esse modelo é conhecido como comunicação unilateral, pois permite realizar tanto a leitura quanto a escrita dos dados por apenas um dos processos envolvidos na operação.

Entrada e saída paralela também foram introduzidas na versão 2.0 do MPI. Os processos podem acessar diferentes porções de um arquivo (paralelismo de dados). Essa característica permite ao processo definir como será realizado o acesso ao arquivo, e então, o MPI abstrairá o dado, realizando o acesso como se o dado fosse sequencialmente organizado para o processo (um arquivo virtual, ou visão, na documentação do MPI). Todos os acessos às visões são traduzidos pelo sistema como deslocamentos reais nos arquivos fonte.

O MPI é uma plataforma excepcional em se tratando de sistemas homogêneos. Ainda faltam importantes características de tolerância a falhas, fazendo com que o sistema não seja tão apropriado para aplicações longas e que utilizam grandes quantidades de recursos distribuídos. Por outro lado, versões proprietárias de MPI para hardware comercializado são extremamente eficientes em seus próprios sistemas.

2.3 Condor

O Condor começou a ser desenvolvido pela universidade de Wisconsin no ano de 1988 e foi projetado para ser um grande sistema de gerenciamento de cargas distribuído. O Condor oferece escalonamento, monitoração, prioridades, gerenciamento de recursos, além de interagir com outros serviços de Grid, como o Globus. Através do sistema, máquinas com baixa atividade podem ser utilizadas, sob demanda, em outras tarefas sob demanda. Desde sua criação, o Condor tem sido usado nas máquinas da Universidade de Wisconsin fornecendo recursos aos pesquisadores e, atualmente, mais de mil máquinas estão sob a supervisão do sistema.

Usuários podem enviar trabalhos de qualquer lugar da rede de maneira totalmente assíncrona. É responsabilidade do Condor escalonar e iniciar todos os trabalhos enviados. Alguns desses trabalhos podem ter prioridades e um usuário pode sempre baixar a sua própria prioridade (como o *nice* no Unix). As máquinas podem ser configuradas para aceitar trabalhos dependendo de sua carga e administradores podem atribuir a diferentes usuários prioridades distintas.

Dependências entre trabalhos podem ser fornecidas por um usuário e, em tal caso, um trabalho só será iniciado pelo sistema depois que todas as suas dependências estiverem preenchidas (dependências devem ser representadas como um grafo direcionado acíclico). Além disso, as máquinas podem declarar seus recursos disponíveis enquanto que os trabalhos declaram seus requisitos. Esses requisitos podem ser tão específicos como "execute em uma determinada máquina" ou tão flexíveis como "dê preferência a máquinas com mais memória". O Condor tem sua própria linguagem para descrever a interação entre requisitos de trabalhos e recursos das máquinas (os *ClassAds* [26]).

Processos paralelos também são aceitos, logo programas PVM e MPI não apresentam problemas para o sistema. O Condor fornece migração transparente de processos, fazendo com que eles funcionem em outra máquina como se eles estivessem na máquina originária da chamada. Isso é feito através de 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. Desta forma, se uma chamada tradicional falha, Condor interfere e tenta obter o dado da máquina original.

É possível parar alguns tipos de processos em pontos de verificação (*checkpoints*) e migrá-los para outra máquina transparentemente. Tais pontos podem ainda ser usados como mecanismos de tolerância a falhas, já que os processos podem ser reiniciados do último ponto de verificação alcançado. O Condor pode ser programado para fazer pontos de verificação periodicamente.

Atualmente, é bastante comum ter departamentos em empresas e instituições com dezenas ou centenas de máquinas, usadas parcialmente ou sub-utilizadas, com muitos recursos ociosos. O Condor é uma boa maneira de se utilizar esses recursos para trabalhos computacionalmente caros, sem as dificuldades de executar e migrar os processos manualmente. O Condor funciona em diferentes plataformas e, na Universidade de Wisconsin, fornece diariamente mais de 650 *CPU days*[1].

2.4 Globus

Em meados dos anos 90, tornou-se bastante comum a utilização de grupos de computadores para a realização de computação paralela, ao invés de caros processadores especializados. A esse novo modelo, deu-se o nome de *cluster* e à ligação de vários *clusters* deu-se o nome de grade computacional (Grid [11]). Tratar todo este sistema como uma grande fonte de recursos foi o fator motivante da criação do Globus.

O Globus é definido como um conjunto de ferramentas de infra-estrutura para metacomputação, dotado de serviços básicos para comunicação, localização e escalonamento do uso de recursos, acesso a dados, autenticação e protocolos de comunicação. Forma-se, com o Globus, um sistema de recursos adaptável a um grande e complexo ambiente, conhecido como AWARE (*Adaptive Wide Area Resources Enviroment*) [10].

Em aplicações executadas nestes ambientes, podem-se notar as seguintes características, comparadas aos sistemas tradicionais mais simples:

- aumento do número de entidades envolvidas na resolução do problema e, como consequencia, a necessidade de se escolher corretamente quais entidades utilizarão quais recursos, através de

[1] Um *CPU day* equivale a um processador disponibilizado durante 24 horas.

critérios como conectividade, segurança, confiabilidade, e até custo de utilização.

- heterogeneidade dos recursos: em todos os níveis, existem os mais variados tipos de recursos, como banda de rede, latência de comunicação, poder computacional, sistemas de armazenamento e equipamentos variados.

- comportamento dinâmico e não previsível: ao se lidar com tantas variáveis, é natural que falhas ocorram e também que as condições de execução se alterem no decorrer da execução (afinal os recursos são compartilhados, e não utilizados exclusivamente). Em algumas situações, chega a ser impossível garantir uma qualidade de serviço.

- domínios administrativos distintos: cada rede tem suas próprias políticas de segurança, seus usuários e seus horários de disponibilidade.

Tendo em vista todos esses fatores, as aplicações necessitam de meios para saber a situação da rede em geral, para avaliar suas próprias configurações, executar e para se adaptar às mudanças que possam vir a ocorrer. O objetivo do Globus é fornecer uma infra-estrutura básica, de baixo nível, para que outros serviços possam funcionar sobre essa camada, tornando o ambiente do programador menos complexo. Assim, o programador pode focar-se apenas no seu problema e deixar para as camadas abaixo lidarem com as complexidades da topologia de uma rede Grid.

O Globus é composto por uma série de módulos, cada qual com uma interface de serviços. A implementação varia de sistema para sistema, mas a interface é sempre a mesma. Cada módulo é responsável por uma área distinta, de acordo com as características já mostradas. Por exemplo, existe um módulo próprio para a especificação de requisitos de um programa e a localização desses na rede, outro para autenticação (interagindo com sistemas já consolidados como NIS e LDAP), outro para acesso aos dados em sistemas de arquivos distribuídos, além de um módulo para realização de comunicação entre entidades da rede.

Esses módulos fazem de uma rede gerenciada por Globus uma grande máquina virtual, totalmente distribuída. É interessante notar que os módulos são dinâmicos, ou seja, um módulo de comunicação pode trocar de protocolo durante uma execução, adaptando-se às mudanças do ambiente. As mudanças podem ser feitas baseando-se em regras, ou forçadas explicitamente pelo sistema. Tudo isso faz do Globus um sistema dinâmico e versátil.

2.5 DataCutter

O DataCutter foi projetado para ser um sistema intermediário (*middleware*) de processamento de dados científicos em ambientes de armazenamento distribuídos tipo Grid. Essa idéia veio do modelo do ADR (*Active Data Repository* [4]), que realizava o processamento do dado no local de sua extração. Tal modelo apresentava os seguintes problemas:

- com o avanço da tecnologia de sensores e o aumento do número de experimentos de coleta de dados (como imagens de satélites, de ressonâncias magnéticas e outros), o tamanho do dado cresceu muito, tornando-se inviável, senão impossível, armazená-lo e processá-lo em um local

único. Além disso, dados semanticamente semelhantes, mas armazenados de maneira diferente, tornaram-se comuns e a integração de bases distintas não era possível.

- O processamento complexo de dados no local de sua extração não aproveita os recursos de forma ótima, devido às frequentes mudanças de contexto (leitura-processamento).

O DataCutter substituiu o ADR, tornando-se um sistema mais genérico. O sistema de recuperação de dados acabou se tornando o Storm [28], que utiliza o DataCutter como um de seus componentes. O DataCutter aplica um modelo de componentes a um ambiente Grid, fazendo com que diferentes tipos de tarefas possam ser executadas em locais propícios, obtendo maior desempenho. Este modelo se assemelha a uma linha de processamento (*pipeline*), onde cada estágio é distribuído no Grid, executando, paralelamente, tarefas diferentes entre si.

Cada estágio dessa linha de processamento é um componente, denominado filtro, que realiza operações sobre o dado. Os custos de comunicação e de computação podem ser diminuídos se os filtros forem estrategicamente colocados, lidando com a heterogeneidade de recursos. Outra maneira de se aumentar o desempenho é através do paralelismo, que pode ocorrer tanto entre filtros distintos (paralelismo de tarefas), quanto entre filtros iguais operando sobre partes distintas do dado (paralelismo de dados). A esse modelo, baseado em filtros, dá-se o nome de filtro-fluxo (*filter stream*).

Cada filtro possui três funções bases. A primeira se refere à inicialização de estruturas necessárias ao processamento. A segunda é o próprio processamento do dado, enquanto a terceira se refere a finalização das estruturas e liberação de recursos para uma próxima tarefa. Filtros são interligados por fluxos, um canal de comunicação lógico unidirecional (semelhante a um *pipe* do UNIX, em rede).

A implementação corrente do DataCutter oferece otimizações no disparo de filtros diferentes no mesmo local, diminuindo o custo da comunicação entre eles, além de facilitar a criação de diferentes grupos de filtros (um layout) ao mesmo tempo e a criação de cópias transparentes. Uma cópia transparente é uma instância de um filtro que possui a capacidade de processamento do filtro.

O modelo de programação do DataCutter permite o desenvolvimento de aplicações genéricas, especialmente as que realizam operações de redução generalizada: o dado é lido, transformado, mapeado e agregado. As operações de transformação e agregação não possuem estados e são independentes da ordem de chegada dos dados. Nesse modelo, os recursos heterogêneos de um Grid podem ser melhor aproveitados. Aparece aí, o problema de escalonamento dos filtros, trabalhado em [23, 6]. O objetivo é reduzir ao máximo o tempo de execução das tarefas, através de uma distribuição dos dados feita por métodos de previsão, como o Network Weather Service [30].

O resultado é um sistema bastante flexível e amigável ao programador em Grid, mas que ainda tem algumas deficiências, como a falta de configuração dinâmica e de tolerância a falhas, embora sua maior limitação seja a de não permitir o controle explícito dos fluxos que chegam em cópias transparentes, dificultando o desenvolvimento de aplicações com manutenção de estado.

2.6 MapReduce

O MapReduce [5] é um modelo de programação e uma implementação de um sistema paralelo para processamento de grandes bases de dados. O sistema foi desenvolvido por pesquisadores do Google

em 2004, para manipular dados contidos em bases com tamanhos superiores a terabytes, distribuídas em milhares de máquinas. A idéia do MapReduce é abstrair as dificuldades do ambiente paralelo para o programador, fornecendo uma interface simples de programação através de duas funções e oferecendo características como tolerância a falhas, distribuição de carga e comunicação dos dados.

O MapReduce aborda uma grande categoria de aplicações que realizam operações de redução de dados. O modelo de programação se baseia em duas funções fornecidas pelo usuário: mapeamento dos dados (*map*) e redução (*reduce*). Os dados de entrada devem ser compatíveis com o sistema, que aceita diversos formatos diferentes. Caso o formato seja diferente, o usuário deve fornecer também uma interface de leitura.

O sistema do MapReduce dispara um processo mestre e vários processos trabalhadores (*workers*) em máquinas disponíveis. Esses processos recebem tarefas de mapeadores e/ou redutores. Cada mapeador recebe uma partição (feita automaticamente pelo sistema) dos dados de entrada e cada tupla lida é passada para a função de mapeamento do usuário. Essa função recebe a tupla como entrada e a mapeia para um novo domínio, ou seja, para cada tupla do tipo $< c_i, v_i >$, onde c_i denota um tipo de chave e v_i o seu valor correspondente, a função *map* emite uma lista de tuplas intermediárias do tipo $< c_j, v_j >$ no domínio da aplicação. Cada redutor é reponsável por um conjunto de chaves das tuplas intermediárias. A função do redutor é transformar os dados intermediários em valores de saída do programa.

O MapReduce oferece várias características interessantes para o seu programador. A primeira é o particionamento automático dos dados em formato padrão, como já mencionado. Nessas partições é garantida a ordenação dos dados, ou seja, os mapeadores receberão as tuplas pela ordem de suas chaves. É possível também eliminar entradas problemáticas de dados. Se o usuário utilizar essa característica, o sistema pode detectar tuplas que causem falhas em diferentes máquinas e ignorá-las. Essa característica é interessante para aplicações que não necessitem de valores exatos. O sistema oferece ainda monitoramento, em forma de HTML, no processo mestre. Existe também uma opção de depuração local para detecção e resolução de problemas.

A tolerância a falhas é um dos pontos fortes do sistema, o qual foi desenvolvido para ser utilizado em grandes cluster de centenas ou alguns milhares de máquinas. Parte da tolerância a falhas do sistema se baseia no fato do MapReduce funcionar sobre o sistema de arquivos GFS (Google File System) [14]. A aplicação mestre verifica periodicamente todos os trabalhadores e, se não houver resposta depois de determinadas tentativas, aquele trabalhador é considerado morto e suas tarefas de mapeamento (completas ou incompletas) e/ou redução (incompletas) são repassadas a outro trabalhador . Isso só é possível porque o GFS oferece redundância de dados, de maneira que os dados de entrada para um mapeador nunca estão em um só local. Se houver um ponto de falha central (aplicação mestre), a aplicação é abortada. No entanto, por ser apenas em um ponto, é fácil previnir falhas comuns. Além disso, a aplicação mestre registra *checkpoints* periodicamente, sendo possível que uma aplicação se utilize desses *checkpoints* para retornar ao último ponto registrado.

Outra característica bastante interessante do sistema são as tarefas reservas (*backup tasks*). Trabalhadores que terminam suas tarefas recebem tarefas de outros para realizarem. Esse trabalho duplicado não altera o resultado final da aplicação, mas é importante por causa da existência de processos desgarrados (*stragglers*). Um processo desgarrado é aquele que se atrasa devido a alguma falha

não fatal. Por exemplo, em *clusters* de centenas de máquinas, é comum que algumas delas tenham problemas de disco e alguns sistemas operacionais (Linux, por exemplo) desabilitam o acesso direto à memória quando isso ocorre. Como resultado, aplicações que utilizam o disco acabam sendo atrasadas enormemente. As tarefas reservas acabam reduzindo esse problema e, sem elas, algumas execuções grandes podem ter seu tempo de execução acrescido em até 44%.

Atualmente, o MapReduce é um modelo consolidado pelo Google e sua eficiência é comprovada por várias aplicações de mineração de dados coletados por robôs do Google. O modelo se mostra extremamente interessante por apresentar ao usuário uma interface padronizadae e simples de ser utilizada, por suas características de tolerância a falhas, pelas suas otimizações de execução e por seu gerenciamento de máquinas em grandes *clusters*. As primitivas de mapeamento e redução se assemelham muito ao fluxo rotulado do Formigueiro, permitindo a execução de aplicações que possuam estados relativos à chave do dado.

2.7 Resumo

Neste capítulo, foi mostrado que vários esforços já foram realizados para auxiliar o programador paralelo e que cada sistema ajuda a abstrair uma parte da tarefa como: a comunicação, o acesso a recursos, a modelagem do programa ou a distribuição de carga. No Capítulo 3 será apresentado o modelo de programação filtro-fluxo com mais detalhes, desde a sua origem até o estágio atual, ou seja, o Formigueiro.

Capítulo 3

Modelo de Programação

O modelo de programação adotado pelo Formigueiro teve origem com os discos ativos (*active disks* [4, 1]). O objetivo dos discos ativos é utilizar os processadores já embutidos nos discos para processamento dos dados em sua fonte. Esse modelo de programação é baseado em fluxos para comunicação (*stream based model*). O DataCutter substituiu o modelo dos discos ativos, criando uma linha de processamento de profundidade ilimitada e distribuída. O Formigueiro evoluiu do modelo do DataCutter, criando a possibilidade de manutenção de estados distribuídos.

3.1 Discos Ativos e Modelo Baseado em Fluxos

A idéia por trás dos discos ativos é dar mais poder à leitura de dados, processando-os em sua fonte. Um programa especial, denominado *disklet*, é instalado no dispositivo controlador dos discos. O *disklet* possui várias limitações por motivos de segurança, como, por exemplo, não permitir alocar ou desalocar memória, nem tampouco realizar operações de entrada/saída (I/O), evitando que programas maliciosos danifiquem os dados já armazenados em um disco. O controle sobre o código do *disklet* é realizado pelo programa de carregamento da mesma na controladora do disco, sendo que o sistema operacional controla a interação do *disklet* com as chamadas de usuário.

Os dados vindos do disco são passados ao *disklet* como fluxo, sequências de *buffers* de tamanho pré-determinado. Da mesma forma, a saída do *disklet* também é na forma de fluxos. As disklets podem ser parametrizadas, tendo seu funcionamento alterado de acordo com os parâmetros recebidos. Ao ser instalada, a função `init` do *disklet* é chamada, permitindo uma inicialização do sistema antes de sua utilização (por exemplo, atribuições de valores de variáveis de acordo com parâmetros). A função `read` é chamada sempre que dados forem lidos do disco e esses são então passados ao *disklet* em forma de fluxo. Similarmente, a função `write` é chamada em situações de escrita. Opcionalmente, pode haver uma função de finalização (para operações de descarga, como *flush*) e um espaço de armazenamento de dados temporários chamado *scratch space*.

O comportamento final dos discos ativos é similar a uma linha de processamento (Figura 3.1). O programa faz a requisição de dados ao sistema operacional como se eles estivessem armazenados de forma adequada a seu processamento. O disco recupera os dados e esses são passados ao *disklet* para processamento. O processamento mapeia o dado de um domínio de armazenamento, onde ele

normalmente se encontra de maneira crua, para o domínio da aplicação. Após o processamento, o volume de dados transmitidos ao programa pelo sistema operacional pode ser reduzido consideravelmente e a carga sobre o processador diminui. A operação inversa ocorre quando os dados são escritos em disco: o programa escreve os dados em seu formato, esses são repassados ao *disklet* para mapeamento inverso da leitura e armazenados em disco.

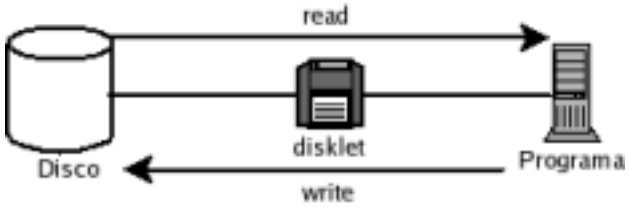

Figura 3.1: Linha de processamento dos discos ativos.

Em simulações realizadas [4], os discos ativos mostraram-se eficientes e escaláveis em situações onde normalmente os processadores ficavam sobrecarregados. As melhoras de desempenho variam entre 3 e 30 vezes, dependendo da quantidade de processamento de dados por byte. Além disso, os programas escalam de acordo com o número de discos: nas arquiteturas convencionais, 4 discos foram suficientes para manter o processador ocupado, enquanto com os discos ativos, os programas escalavam com até 16 discos.

3.2 DataCutter e o Modelo Filtro-Fluxo

O DataCutter evoluiu do modelo de programação dos discos ativos. Os discos ativos se baseavam-se na premissa de que os dados deveriam ser processados em sua fonte, criando uma linha de processamento de três níveis: leitura do disco, processamento pelo *disklet* e utilização do dado pelo programa. Com o DataCutter, a profundidade da linha se tornou ilimitada. Foi criado o conceito de filtro, substituindo o de *disklet*. O filtro é um programa com o objetivo de processar dados de um domínio de entrada para outro de saída. Filtros se comunicam por meios de fluxos de dados, assim como as disklets. No entanto, os filtros funcionam no espaço do usuário e não possuem as limitações do ambiente restrito das disklets.

O DataCutter permitiu o processamento distribuído do dado em uma linha de profundidade ilimitada. Em sua fonte, por exemplo, um filtro pode ser responsável por ler os dados e fazer um pré-processamento simples. Operações mais complexas podem ser feitas por outras máquinas, livrando o filtro leitor do processamento pesado. Com isso, o filtro leitor pode servir a outras requisições mais eficientemente, aumentando a vazão de dados (*throughput*). Similarmente, outros filtros especializam-se em sua tarefa, realizando-a de maneira eficiente e aumentando o paralelismo de tarefas.

Esse modelo de programação foi chamado de filtro-fluxo. Além do paralelismo de tarefas, ele introduziu também o paralelismo de dados através das cópias transparentes. A cópia transparente pode ser pensada como uma instância de um filtro. O filtro define a função da instância e várias instâncias de um filtro podem ser criadas, executando o mesmo código sobre dados diferentes.

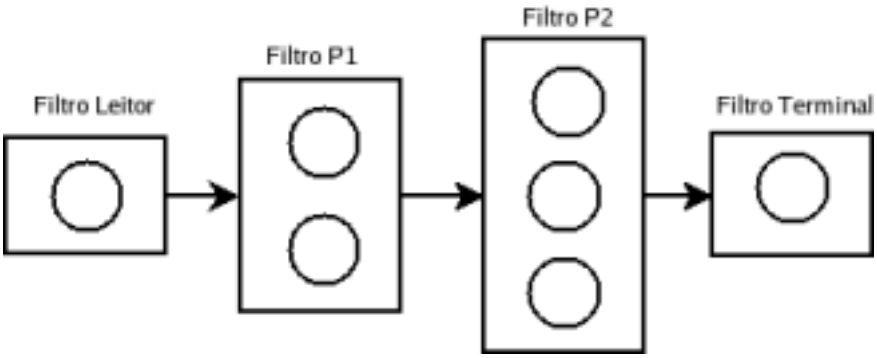

Figura 3.2: Linha de processamento em fluxo de dados entre filtros.

Na Figura 3.2, são apresentados quatro filtros. O primeiro (filtro $Leitor$) é responsável por realizar a leitura dos dados em sua fonte e realizar algum pré-processamento. Todo dado lido é repassado ao filtro de processamento $P1$, que realiza o primeiro processamento caro sobre o dado. Apenas uma instância de $P1$ recebe o dado lido. Através de políticas de comunicação entre os filtros, os dados vão sendo repassados a instâncias diferentes de $P1$. Similarmente, o dado processado por esse filtro é repassado às instâncias de $P2$, que realizam a segunda parte do processamento do dado. O último filtro é terminal na linha de processamento e é responsável por armazenar os dados finais ou mostrá-los ao usuário. Quando todas as instâncias de todos os filtros estiverem executando, a linha estará completamente cheia e o aproveitamento máximo da mesma.

Ao conjunto de filtros e fluxos é dado o nome de *layout*, representado como um grafo direcionado. Neste texto, os filtros serão representados como retângulos e suas instâncias como círculos. A comunicação entre os filtros (fluxos) será representada por setas indicando sua direção. Em casos especiais, a comunicação entre instâncias também será mostrada. A política de comunicação, quando omitida, será considerada como alternância simples (*round-robin*): a cada escrita, uma instância receberá o dado em sequência. Em outros casos, será etiquetada à seta o tipo de comunicação utilizada.

3.3 Formigueiro e Manutenção de Estados Distribuídos

O Formigueiro estendeu o modelo de fluxo de dados. O modelo de fluxo de dados entre filtros é simples e bem definido, tornando-se um grande atrativo para o programador. A tarefa de dividir o programa em tarefas bem definidas acaba se refletindo também no desempenho da aplicação, que é automaticamente paralelizada pelo sistema. No entanto, algumas aplicações não podem ser desenvolvidas no ambiente do DataCutter (ou seu desenvolvimento não é prático).

O problema ocorre quando os filtros executam tarefas com manutenção de estado. Como instâncias de um mesmo filtro não se comunicam e não há um controle determinado por usuário sobre o destino dos dados, não há uma maneira simples de se implementar filtros com estado no DataCutter (se houver cópia transparente do filtro com estado).

O Formigueiro apresenta uma solução simples e eficiente para esse problema: o fluxo rotulado. O DataCutter apresenta políticas de comunicação para os fluxos, mas não dá ao programador uma

forma de controlar o destino dos dados à sua maneira. O fluxo rotulado do Formigueiro permite ao programador exatamente isso: o controle sobre o destino dos dados.

Em um fluxo rotulado, toda mensagem enviada é analisada por uma função de identificação de uma informação especial, o rótulo (*label*). O formato do rótulo é dependente de cada aplicação, sendo definido pelo programador. Após a identificação do rótulo da mensagem, outra função é aplicada, dessa vez sobre o rótulo, para determinar o destino da mensagem. O *hash* é uma função global entre instâncias de um mesmo filtro, significando que dado um determinado rótulo, independente de qual instância aplique o *hash*, a saída será sempre a mesma. Essas duas funções são passadas ao sistema pelo usuário e permitem um controle explícito sobre o destino dos dados.

Considere, por exemplo, uma aplicação cujo objetivo seja identificar quantas cidades de um estado estão a uma distância máxima da capital. Cada cidade possui como atributos seu estado, sua latitude e sua longitude.

A Figura 3.3 mostra o grafo direcionado da aplicação. O filtro leitor (L) tem duas tarefas: a primeira é realizar a leitura das capitais e enviá-las aos classificadores; na segunda, o restante das cidades da base é lido e, para cada uma das cidades, o leitor converte suas latitudes e longitudes para um formato cartesiano (X,Y) e envia essas informações ao filtro classificador (C), responsável pela capital do estado. Estes computam quais cidades se situam próximas à sua capital e realizam a contagem (a contagem é o estado que o filtro deve manter). O número de cidades próximas à capital é repassado ao filtro de saída (S), que escreve em um arquivo de saída o resultado. O programa é iniciado quando o usuário passa a distância máxima de interesse (DMI).

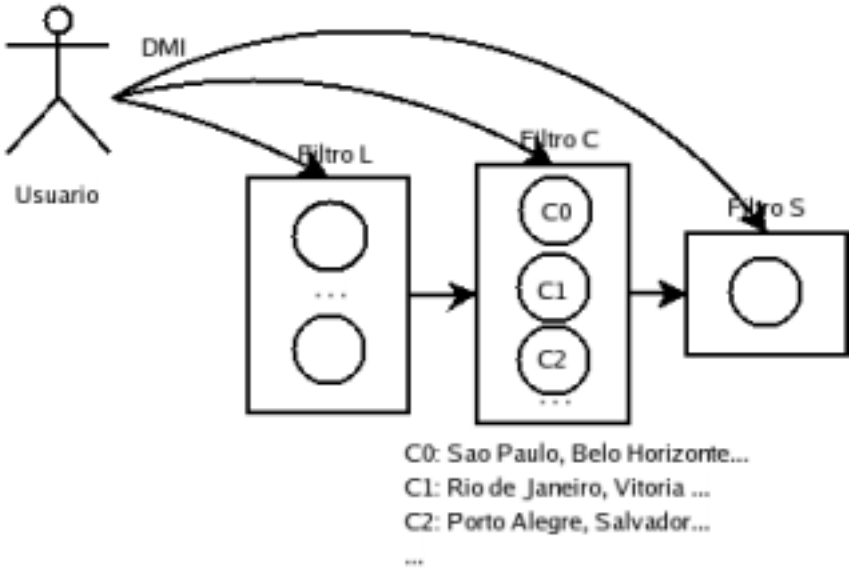

Figura 3.3: Grafo direcionado do programa para contagem de cidades do estado próximas à capital.

Ao ler as tuplas das capitais, os leitores identificam o estado e enviam as capitais aos classificadores responsáveis por cada uma. Cada classificador pode ser responsável por mais de uma capital. O estado da cidade é utilizado como o rótulo da mensagem e as cidades de um mesmo estado são sempre enviadas ao mesmo classificador, pois a função de *hash* é global. Quando o leitor lê a cidade da base,

a latitude e longitude são convertidas para valores de cálculo compatíveis com a distância máxima de interesse. A mensagem do leitor para o classificador consiste na tupla (nome da cidade, estado, coordenadas). Para cada mensagem recebida, o classificador extrairá as coordenadas e o estado e fará o cálculo da distância da cidade até a capital. Caso seja menor do que DMI, o contador de cidades próximas será incrementado. Ao fim da leitura de todas as cidades da base, o resultado de cada estado é enviado ao filtro S para ser mostrado ao usuário.

Independentemente do número de filtros classificadores ou leitores, o resultado da aplicação será sempre o mesmo. A existência do estado no filtro classificador (o número de cidades próximas) é tratada pelo fluxo rotulado, que garante que cidades de um mesmo estado sejam sempre enviadas a uma mesma instância. Portanto, o fluxo rotulado é uma maneira transparente da aplicação manter um estado distribuído. O controle sobre o destino dos dados é realizado por funções a parte do código do filtro, passadas como bibliotecas dinâmicas criadas pelo usuário.

Uma variação do fluxo rotulado é a difusão seletiva (ou fluxo rotulado múltiplo). Assim como no fluxo rotulado, o controle sobre o destino dos dados é realizado pelo usuário. A diferença, no entanto, está no número de instâncias do filtro do lado receptor que receberá a mensagem. No fluxo rotulado, apenas uma instância recebe a mensagem. No caso da difusão seletiva, mais de uma instância poderá receber a mensagem.

Para exemplificar, considere uma variação da aplicação mostrada na Figura 3.3[1]. Cada cidade na base de dados conterá, além do atributo estado, o atributo estados próximos. O objetivo agora é contar o número de cidades (pertencentes ao estado ou não) próximas à capital do estado.

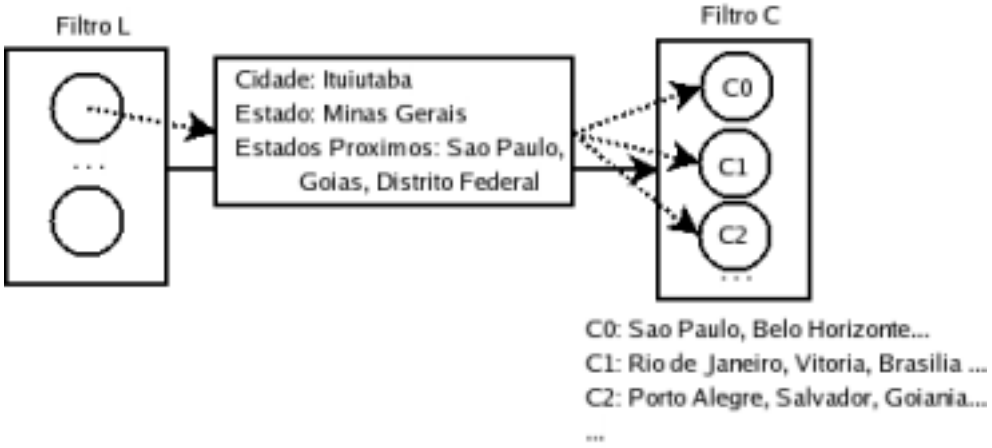

Figura 3.4: Cada mensagem do programa poderá ter mais de um destino.

Para realizar a mudança, o filtro leitor deverá agora extrair o estado e os estados próximos de cada cidade (já armazenados na base de dados). Cada mensagem do leitor para o classificador terá como rótulo esses estados. A função getLabel extrairá todos os estados próximos à cidade, enquanto a função hash encontrará as instâncias dos filtros classificadores responsáveis por cada um desses estados.

A mensagem será enviada a cada uma dessas instâncias, conforme mostrado na Figura 3.4: o filtro leitor L_0 encontra a cidade de Ituiutaba; esta cidade está situada no triângulo mineiro e está

[1]Para simplificar, o Distrito Federal será considerado como um estado.

próxima dos estados de Goiás, São Paulo e do Distrito Federal. Portanto, ela será enviada para cada um dos filtros responsáveis pelas capitais desses estados (C_0 para Belo Horizonte e São Paulo, C_1 para Brasília e C_2 para Goiânia). Cada instância computará a cidade recebida normalmente e, caso esteja próxima da capital, será contabilizada pelo filtro.

3.4 Resumo

Neste capítulo foi apresenada a evolução do modelo filtro-fluxo, uma abstração de modelagem ao programador paralelo. O modelo iniciou-se com os discos ativos, onde tinha um formato simples de processamento de dados no local de sua extração. Este modelo foi substituído pelo DataCutter, possibilitando a criação de programas totalmente distribuídos e independentes do local da extração dos dados. Por fim, o Formigueiro elevou mais ainda a abstração, facilitando o desenvolvimento de aplicações filtro-fluxo com manutenção de estados. No Capítulo 4 será apresentado o sistema desenvolvido neste trabalho.

Capítulo 4

Formigueiro

O Formigueiro é um sistema que estende o DataCutter, elevando seu nível de abstração para o programador. Inicialmente, a idéia era trabalhar sobre o código do DataCutter, mas após um período de análise desse código, chegou-se à conclusão de que seria menos trabalhoso fazer uma implementação baseada em PVM do que alterar o DataCutter, já que a implementação do DataCutter não previa tolerância a falhas ou configuração dinâmica.

O PVM foi escolhido por ser um sistema mais adequado a ambientes heterogêneos do que o MPI, além de já ter embutidas características importantes de notificação de falhas e configuração dinâmica [13]. Utilizando-se o PVM, a implementação do Formigueiro foi bastante simplificada, pois toda a parte de inicialização de processos e comunicação baseou-se em funções similares às do PVM, o que tornou possível focar nas áreas mais importantes que eram buscadas.

No Capítulo 3 foram apresentados o modelo do Formigueiro e como ele aumentou o poder do DataCutter através do fluxo rotulado. Neste capítulo, será mostrado o funcionamento do Formigueiro.

4.1 Pré-Requisitos

O Formigueiro é um sistema desenvolvido para Linux. Para seu funcionamento, são necessárias as seguintes bibliotecas listadas a seguir, embora outras versões tambemḿpossam funcionar, desde que mantenham a consistência de interface de funções exportadas:

- PVM versão 3.0 [pvm];
- Expat 0.5 [exp];
- Glibc 2.3.0 com carregamento dinâmico (libdl.so), matemática (libm.so) e POSIX threads (libpthread.so).

Não é necessária a existência de um sistema de arquivos compartilhado, mas as bibliotecas dos filtros devem estar disponíveis em todas as máquinas que forem utilizá-las, assim como qualquer arquivo que seja acessado durante a execução.

4.2 Cenário

Na Figura 4.1, vê-se um cenário típico de aplicações Formigueiro. Tem-se uma linha de três filtros, $F1$, $F2$ e $F3$. Em cada estágio, o dado é mapeado de um domínio pertencente ao filtro para o domínio do filtro seguinte. O filtro $F1$ executa o processamento inicial, lendo os dados de sua fonte (um arquivo ou banco de dados, por exemplo). Todo dado processado por $F1$ é repassado a $F2$, que por sua vez processa e repassa o resultado a $F3$. Quando os três estágios estão processando, a linha está completamente ocupada. Estão disponíveis para a aplicação 6 máquinas, $M1$ a $M6$. Na Figura 4.2, foi feita uma distribuição de filtros (layout), onde há cópias transparentes dos filtros $F2$ e $F3$, possivelmente os mais caros computacionalmente. Quando $F1$ processa um dado, ele o envia a alguma instância do filtro $F2$. O mesmo ocorre com as instâncias de $F2$, que escrevem para $F3$.

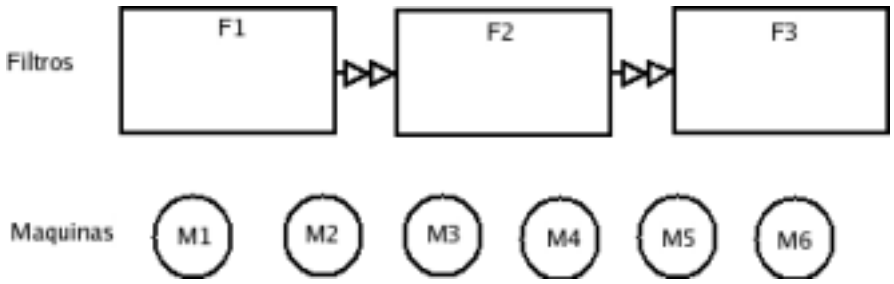

Figura 4.1: Um cenário hipotético para execução de um programa Formigueiro.

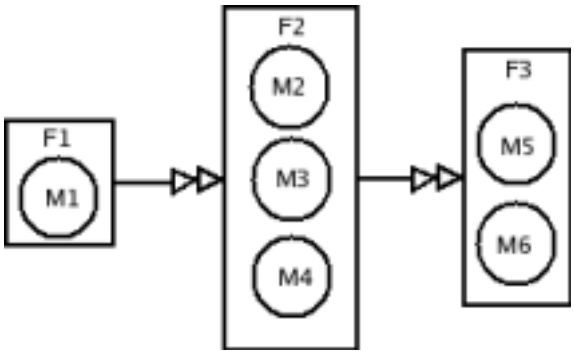

Figura 4.2: Uma possível distribuição de filtros para o cenário da Figura 4.1.

O Microscópio Virtual [7, 9], uma aplicação de visualização de imagens digitalizadas de microscópios eletrônicos, se encaixa nesse cenário. Outro exemplo deste tipo de aplicação é o gerenciamento de reservas de petróleo [28], onde simulações são realizadas para saber qual a maneira mais barata e segura de se realizar uma escavação, através da manipulação de múltiplas variáveis.

4.3 Trabalho

O início do funcionamento da linha de processamento do Formigueiro acontece quando um trabalho (*work*) é criado e enviado para a linha. Um trabalho é definido semanticamente pela aplicação e pode ser visto como a unidade que contém os parâmetros de entrada para uma aplicação. Múltiplas

execuções de uma aplicação com um mesmo trabalho sobre os mesmos dados devem gerar o mesmo resultado.

Em aplicações de tratamento de imagens, o nome de um arquivo a ser tratado e quais as operações a serem realizadas poderiam estar contidas no trabalho. Já em aplicações de bancos de dados, o trabalho poderia ser a consulta a ser realizada pelos filtros leitores. No exemplo das cidades, do Capítulo 3, o trabalho é simplesmente a distância máxima para uma cidade ser considerada próxima à capital de um estado.

Após serem iniciados, todos os filtros aguardam a recepção do trabalho para começarem a trabalhar. Suas rotinas de inicialização são chamadas para alocação de *buffers* e os descritores das portas de comunicação com os fluxos são abertos. Em seguida, as rotinas de processamento de cada um dos filtros são chamadas. Quando todas as instâncias de um determinado filtro terminam de processar os dados relativos ao trabalho, suas rotinas de finalização são chamadas e são enviadas mensagens de fim do trabalho (*EOW*) ao próximo filtro da linha. Quando todos os filtros terminam o trabalho, a aplicação gerente é notificada e o sistema pode executar um próximo trabalho ou encerrar o funcionamento.

Outra função do trabalho é definir a unidade mínima de processamento. Em caso de falha, o sistema é reiniciado a partir do último trabalho não realizado. Finalmente, no modelo de programação do Formigueiro, supõe-se que o resultado de um trabalho dependa apenas do próprio trabalho, ou seja, caso a execução do trabalho seja reiniciada, o resultado obtido será o mesmo de uma execução não interrompida. Os filtros não mantêm estados entre trabalhos distintos.

4.4 Interface do Programador

A interface de programação do Formigueiro foi separada em 3 componentes bem definidos, simplificando o código do programador. É tarefa do prograd or escrever:

- a aplicação que dispara os trabalhos, chamada de *Gerente* (*Manager*)[1];
- as bibliotecas dinâmicas: código dos filtros e, se utilizadas, as funções de fluxo rotulado;
- o arquivo de configuração do sistema, apresentando o layout da aplicação (distribuição de filtros e comunicação entre eles, fluxos) e, opcionalmente, as demandas dos filtros e os recursos disponíveis nas máquinas.

4.4.1 Gerente

A aplicação que dispara os filtros é extremamente simples, como mostrado na Listagem 4.1.

Listagem 4.1: Código do Gerente

```
int main (int argc, char *argv[]) {
  //inicia o sistema
  Layout *layout = initDs("nome_arquivo.xml", argc, argv);
```

[1]No DataCutter, tal aplicação é chamada de console, assim como no PVM.

```
  //cria o trabalho a ser enviado... multiplos podem
  //ser feitos e enviados
  Work *w = (Work*)malloc(sizeof(Work));

  //envia o trabalho a linha de producao
  appendWork(layout, w, sizeof(Work));

  //finaliza o sistema
  finalizeDs(layout);

  return 0;
}
```

A chamada initDs cria o sistema do Formigueiro, lendo o arquivo de configuração passado como primeiro argumento. Em seguida, são criados os trabalhos, que são enviados a todos os filtros antes da execução. O trabalho é enviado com appendWork. Por fim, é chamado finalizeDs, que finaliza todas as instâncias de filtros.

4.4.2 Arquivo de Configuração XML

O arquivo de configuração XML tem a tarefa de descrever quais filtros o usuário utilizará, quais recursos e máquinas estarão disponíveis para o sistema e como os filtros se comunicam. O nome do arquivo é passado à função initDs, e contém três seções. A primeira seção é a declaração de máquinas com seus recursos. A segunda seção é a declaração de filtros com suas instâncias e requisitos. Por último, a declaração de fluxos, ligando os filtros entre si. A Listagem 4.2 mostra um arquivo de configuração genérico, e a Figura 4.3, sua representação gráfica.

Listagem 4.2: Arquivo de Configuração XML

```
<config>
  <hostdec>
    <host name="maquinaA">
      <resource name="memoria"/>
    </host>
    <host name="maquinaB">
      <resource name="memoria"/>
      <resource name="base_dados" />
    </host>
  </hostdec>
  <placement>
    <filter name="leitor_base" libname="leitor.so">
      <instance demands="base_dados" />
    </filter>
    <filter name="processa_base" libname="processa.so" numinstances="5" >
      <instance demands="maquinaB" />
    </filter>
  </placement>
  <layout>
    <stream>
```

```
      <from filter="leitor_base" port="envia_dados" policy="ls" policylib="ls_lib
          .so"/>
      <to filter="processa_base" port="recebe_dados" />
    </stream>
  </layout>
</config>
```

Neste arquivo, são declaradas duas máquinas disponíveis dentro do elemento hostdec, $maquinaA$ e $maquinaB$. As duas máquinas possuem o recurso $memoria$, que pode significar, por exemplo, que ambas possuem quantidade abundante de memória primária. A segunda possui ainda outro recurso, denominado $base\_dados$. A semântica de recursos é definida pelo usuário, e pode significar qualquer coisa, ou até ser completamente ignorada. Neste caso, o recurso $memoria$ não é utilizado por nenhum filtro. O recurso $base\_dados$ pode significar que aquela máquina é a única que possui acesso à base, e portanto, o filtro leitor da base deve ser obrigatoriamente executado nela. Além disso, pode ser dado um peso diferente a uma máquina através do atributo weight, havendo maior chance de ser ela a escolhida pelo Formigueiro como um local de execução de filtro.

A declaração placement declara os filtros e seus locais de execução (implícita ou explicitamente). Caso um filtro não apresente demandas, o Formigueiro escolherá aleatoriamente[2] onde este filtro será executado, levando em consideração o peso das máquinas. Na listagem acima, o filtro $leitor\_base$ possui apenas uma instância, a qual demanda $base\_dados$. Já o filtro $processa\_base$ possui 5 cópias e, dessas, apenas uma faz alguma demanda (ser executado na $maquinaB$). O Formigueiro está livre para escolher onde as outras quatro instâncias serão executadas.

A última seção deste arquivo é o layout. Aqui são declarados os fluxos que conectam os filtros. O elemento stream possui dois sub-elementos, a origem (from) e o destino (to). Os atributos determinam qual filtro enviará os dados e qual os receberá, além de especificar a porta. Dois atributos adicionais (policy e policylib) mostram que a política de envio das instâncias do filtro $leitor\_base$ realizará o envio às instâncias de $processa\_base$, utilizando um fluxo rotulado (ls), onde as funções getDest e hash estão contidas na biblioteca dinâmica $ls\_lib.so$. Outras políticas possíveis são: difusão (broadcast), difusão seletiva (mls), alternância simples (rr, ou round robin, se omitido, esta é utilizada como padrão) e aleatória (random).

Na Figura 4.3 pode-se ver o possível resultado deste arquivo XML. Diz-se possível, porque o filtro $processa\_base$ não define explicitamente onde disparar quatro de suas instâncias, sendo estas distribuídas aleatóriamente entre as máquinas[3]. Note que a única instância de $leitor\_base$, preenchida de cinza claro, é disparada em $maquinaB$, devido à sua demanda explícita. Ao não declarar o número de instâncias de um filtro, o Formigueiro assume uma instância como padrão, ou o número de instâncias declarado dentro do elemento filtro. Em relação ao filtro $processa\_base$, apenas uma das instâncias possui demanda (mostrada na Figura 4.3 em cinza escuro). Essa instância é executada em $maquinaB$, enquanto as restantes são aleatoriamente escolhidas pelo Formigueiro.

[2] Um algoritmo de *round-robin* pode ser mais apropriado neste caso, mas não foi implementado ainda. A distribuição deve levar em consideração também as demandas dos filtros, tornando o algoritmo mais complexo que uma simples alternância de máquinas.

[3] O Formigueiro utiliza um algoritmo aleatório, baseado nos pesos dos nodos. Um nodo sem peso declarado é considerado como peso 1.

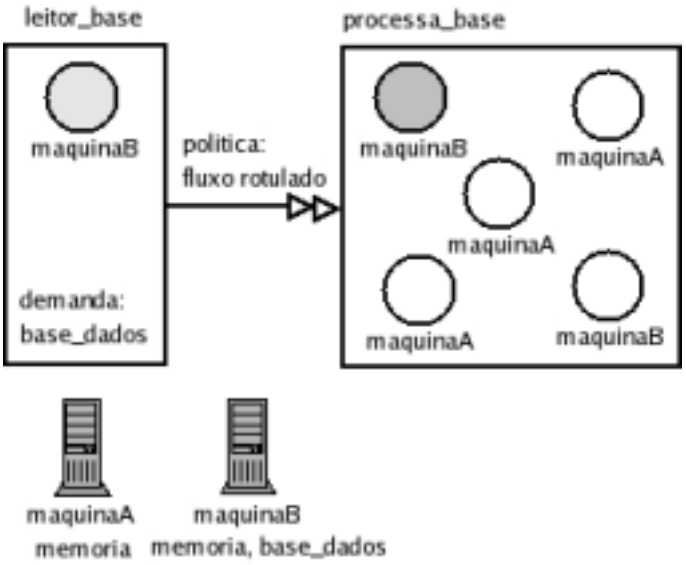

Figura 4.3: Representação gráfica do XML apresentado na Listagem 4.2

4.4.3 Filtros

Os filtros são a parte mais significativa do programa. Neles, o programador insere seu código, realizando as tarefas de processamento. No DataCutter, um filtro é representado por uma classe virtual C++, onde o programador define as funções de inicialização, processamento e finalização. No Formigueiro, o programador deve fornecer uma biblioteca dinâmica [29] que implemente as mesmas funções. Essa biblioteca deve então ser passada ao sistema pelo arquivo XML (veja 4.4.2 para mais detalhes), para que, em tempo de execução, os filtros possam carregar a biblioteca correspondente e realizar suas tarefas.

Para cada trabalho enviado pelo *Gerente*, é executada a função `initFilter`, que fará a inicialização de estruturas. Em seguida, a função `processFilter` é chamada, sendo esta responsável pelo processamento. Em ambas as funções, são passados dois argumentos: um ponteiro genérico, que aponta para o trabalho, e um inteiro, representando seu tamanho. Tal mecanismo permite que o trabalho não possua nenhuma restrição, podendo o programador utilizá-lo para qualquer fim, ou até ignorá-lo. Por fim, a função `finalizeFilter` será chamada, liberando os recursos utilizados pelo filtro para aquele trabalho. A partir deste ponto, o filtro estará apto a receber dados do próximo trabalho enviado pelo *Gerente*.

Como toda biblioteca compartilhada, duas outras funções podem ser definidas: uma para uma inicialização única quando a biblioteca for carregada e outra para uma finalização única, quando a biblioteca for descarregada pelo sistema. Tais funções podem vir a ser úteis para uma aplicação que realize uma tarefa comum a todos os trabalhos enviados para a linha de produção como por exemplo, para uma aplicação Formigueiroque utilize uma máquina virtual Java. Nesse caso, é interessante iniciar a máquina uma única vez, já que o custo de fazê-lo a cada trabalho pode ser significativo e limitar o ganho de desempenho do sistema. Estas duas funções não possuem um nome pré-definido, mas devem ser declaradas com os atributos de compilação `__attribute__ (( constructor ))` e `__attribute__ (( destructor ))` [29, 15], para a função chamada durante a carga e para a função chamada durane a

liberação da biblioteca, respectivamente.

4.4.3.1 Interface de Programação para Filtros

O programador tem à sua disposição uma suíte de funções para auxiliá-lo. Em primeiro lugar, estão as funções de manipulação das portas dos filtros. Portas são as conexões dos filtros com os fluxos, indentificadas por nomes, como mostrado no arquivo XML (ver Listagem 4.2). Assim como os fluxos, as portas são unidirecionais, podendo ser de entrada (apenas recepção de dados) ou saída (apenas escrita). A função de escrita em portas de saída é chamada de dsWriteBuffer. Em portas de leitura, a função de leitura é chamada de dsReadBuffer e a de verificação de dados é chamada de dsProbe.

Todas estas funções utilizam um descritor para identificar a porta, pois referenciá-la pelo nome é pouco eficiente em tempo de execução, além de pouco prático para o programador. Para se obter o descritor de uma porta, há duas funções que recebem o nome da porta como parâmetro e retornam este descritor: para portas de saída, a função é dsGetOutputPortByName, e para portas de entrada, dsGetInputPortByName. Quando uma porta de saída não for mais utilizada em um trabalho, existe uma operação para fechá-la, dsCloseOutputPort. Tal operação, não obrigatória, envia a informação de fim de dados para o fluxo (*EOW*[4]). Ao final do processFilter , todas as portas de escrita são fechadas. Alguns algoritmos, no entanto, se utilizam desta função, pois escrevem em diferentes portas dados distintos ao invés de utilizar um fluxo único.

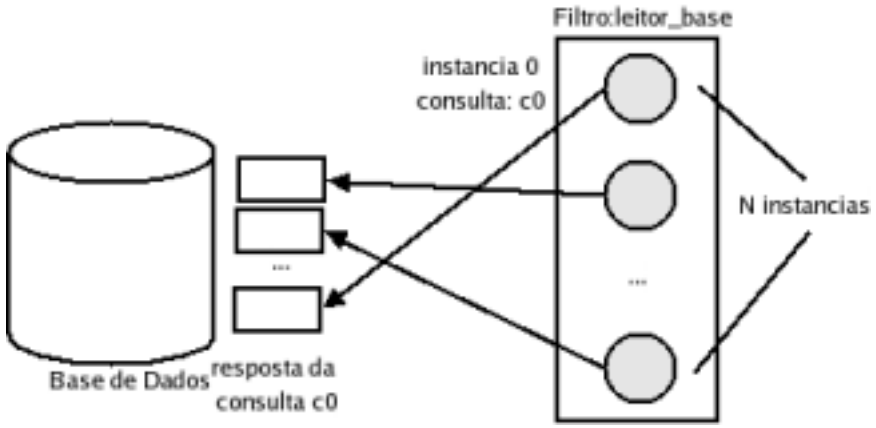

Figura 4.4: Instâncias de um filtro leitor realizando paralelismo de dados

Para realizar a divisão de tarefas entre os filtros, existem duas funções: dsGetNumInstances e dsGetMyRank . A primeira função retorna quantas instâncias o filtro possui e a segunda retorna o identificador da instância entre todas elas. Um exemplo de como utilizar essas funções é um filtro leitor de uma base de dados. Cada instância é responsável por ler apenas parte da base. Para isso, elas se utilizam das informações passadas pelas duas funções e realizam a consulta com parâmetros próprios, recebendo apenas uma parcela dos dados.

A Figura 4.4 ilustra uma possível utilização destas funções. Cada uma das instâncias i do filtro descobre qual é seu *ranking* (r) entre todas as instâncias (n) e faz a consulta de acordo com esta

[4]Para um filtro que possua várias portas de envio fechar uma porta indica que aquela porta não enviará mais dados do trabalho corrente, mas as outras estão livres para continuar enviando dados até serem fechadas também.

informação. A consulta é uma função $f_i(r, n, o)$ do *ranking*, número de instâncias, e algum outro parâmetro qualquer de interesse da aplicação (o). Para cada uma dessas consultas f_i, obtem-se uma resposta (a_i).

O *ranking* (r) e o número de instâncias (n) servem apenas para dividir os dados entre as instâncias. O parâmetro o é um filtro que varia de acordo com a aplicação. Por exemplo, numa base de dados contendo números, o poderia ser a condição de que o número seja primo. Consequentemente, se o parâmetro o for vazio, toda a base (B) será lida:

$$a_0 \cup a_1 \cup \ldots \cup a_{n-1} = B \qquad \text{e} \qquad a_0 \cap a_1 \cap \ldots \cap a_{n-1} = \phi$$

Quando um programa encontra um erro, ele pode notificá-lo ao *Gerente* através da função dsExit. Esta notificação faz com que o *Gerente* termine a execução de todos os filtros e retorne um código de erro para o sistema. Por exemplo, na Figura 4.4, a base de dados pode não estar disponível e então o filtro, ao não conseguir se conectar à mesma, notificará o *Gerente* e a aplicação retornará um erro para o usuário.

A Figura 4.5 mostra a representação gráfica do filtro apresentado na Listagem 4.3. Este filtro realiza a simples operação de receber um número na porta dividendoP, dividí-lo por quatro, enviar o resto da divisão para a porta restoP e o quociente para a porta quocienteP. Durante a inicialização, os descritores das portas são armazenados em variáveis. Durante o processamento, para cada valor que chega à porta dividendoP, é feito o cálculo mencionado acima e o resultado enviado às portas correspondentes.

Listagem 4.3: Filtro divP4

```
#include <stdlib.h>
#include <stdio.h>
#include <FilterDev/FilterDev.h>

InputPortHandler dividendoP;
OutputPortHandler restoP;
OutputPortHandler quocienteP;

int initFilter(void *work, int size){
  //pega os handlers de saida
  dividendoP       = dsGetInputPortByName("dividendoP");
  restoP           = dsGetOutputPortByName("restoP");
  quocienteP       = dsGetOutputPortByName("quocienteP");

  return 1;
}

int processFilter(void *work, int size){
  int dividendo;
  int divisor = 4;
  int resto;
  int quociente;
```

```
  //le o dividendo
  while ((dsReadBuffer(dividendoP, &dividendo, sizeof(int))) != EOW) {
    quociente = dividendo / divisor;
    resto = dividendo % divisor;

    //escreve pra filtro resto
    dsWriteBuffer(restoP, &resto, sizeof(int));
    //escreve para filtro quociente
    dsWriteBuffer(quocienteP, &quociente, sizeof(int));
  }

  return 1;
}

int finalizeFilter(void){
  return 1;
}
```

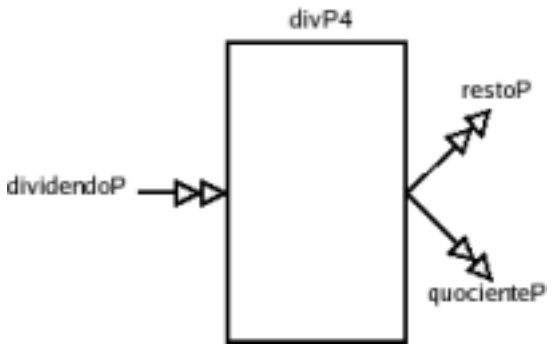

Figura 4.5: Representação gráfica do filtro divP4, mostrado na Listagem 4.3

4.4.3.2 Instrumentação

A interface de programação para filtros também oferece funções para instrumentação do usuário. Por padrão, o Formigueiro oferece os tempos gastos em cada uma das três funções da biblioteca dinâmica, nas funções de leitura e escrita para filtros, e nas funções do fluxo rotulado. Ao final da execução, cada instância de filtro gera seus próprios dados em um arquivo na máquina que executou o filtro.

As funções de usuário permitem que ele forneça estágios adicionais para serem calculados. A função dsInstSetStates deve ser chamada uma única vez, durante a inicialização do sistema. Essa função declara os tempos de usuário em um vetor de *strings*, além de zerar todos os tempos computados pela instrumentação. Para o programador entrar em um estado seu, ele deverá chamar a função dsInstEnterState . Essa função empilha o estado corrente e o altera para o novo estado fornecido. Ao se empilharem estados, deve-se sempre desempilhá-los, em algum momento, com dsInstLeaveState .

Quando o programador deseja entrar em uma sequência de estados próprios, chama-se dsInstEnterState uma única vez, e se alternam os estados com dsInstSwitchState , até o final da sequência, quando é chamada dsInstLeaveState , retornando-se ao estado do Formigueiro. Isso permite que o impacto da

instrumentação seja menor, pois dsInstSwitchState apenas altera o estado no topo da pilha, enquanto cada chamada dsInstEnterState empilha estados e deve ser correspondida por dsInstLeaveState.

A Listagem 4.4 mostra o código do $divP4$ instrumentado. Neste caso, são declarados dois estados de usuário: no primeiro ($DIVISAO_QUO$) se calcula o quociente e no segundo ($DIVISAO_RES$) se calcula o resto. Quando se entra no estado $DIVISAO\_QUO$, empilha-se este sobre um estado do Formigueiro (neste caso, o $process$). Ao sair de $DIVISAO_QUO$, o sistema volta ao seu estado anterior.

Listagem 4.4: Exemplo de Instrumentação

```
#define DIVISAO_QUO 0
#define DIVISAO_RES 1

int __attribute__ ((constructor)) iniciaInstrumentacao(){
  char *estados[2] = {"quociente", "resto"};

  dsInstSetStates(estados, 2);

  return 1;
}

//...

int processFilter(void *work, int workSize){
  //...

  //le o dividendo
  while ((dsReadBuffer(dividendoP, &dividendo, sizeof(int))) != EOW) {
    dsInstEnterState(DIVISAO_QUO);
    quociente = dividendo / divisor;
    dsInstSwitchState(DIVISAO_RES);
    resto = dividendo % divisor;
    dsInstLeaveState();

    //escreve pra filtro resto
    dsWriteBuffer(restoP, &resto, sizeof(int));
    //escreve para filtro quociente
    dsWriteBuffer(quocienteP, &quociente, sizeof(int));
  }

  //...
}
```

Um exemplo de resultado desta instrumentação é mostrado a seguir (os tempos são apenas ilustrativos). A instância é identificada como $divP4.0$, que significa a instância 0 do filtro $divP4$. O estado $timer\_w\_bkd$ mostra o tempo em que um filtro bloqueado durante a escrita. O bloqueio acontece quando há contenção na recepção e o *buffer* de envio enche. O tempo $timer\_void$ mostra quanto tempo o Formigueiro gastou em seus estados internos (por exemplo, entre o process e o finalize). Este

tempo deve sempre ser pequeno, mostrando que a sobrecarga do Formigueiro é pequena.

Listagem 4.5: Exemplo de saída da instrumentação

```
Execution time for divP4.0
timer_init          :          0.000002 seconds
timer_proc          :          0.000003 seconds
timer_read          :          0.000542 seconds
timer_write         :          0.002831 seconds
timer_w_bkd         :          0.000000 seconds
timer_ls            :          0.000000 seconds
timer_finalize      :          0.000001 seconds
timer_void          :          0.000004 seconds
User States:
quociente           :          0.000001 seconds
resto               :          0.000001 seconds
```

4.5 Configuração Dinâmica

O Formigueiro foi estruturado visando a possibilidade de um funcionamento dinâmico, cuja importância é permitir que o número de filtros varie durante a execução e que mais recursos sejam adicionados em tempo real. Outra característica importante é permitir que haja um nível de tolerância a falhas, pois se houver falha de equipamento em uma execução, a instância do filtro que executava naquele local tem que ser excluída ou substituída.

Os filtros do Formigueiro são filtros genéricos, capazes de executar qualquer código em tempo de execução. Ao serem disparadas nos locais de execução, as instâncias dos filtros ficam em um estado de espera, durante o qual receberão os dados do Gerente, dizendo qual tarefa será desempenhada. Todas as informações necessárias para a execução do filtro são enviadas por mensagens do Gerente, como por exemplo, qual biblioteca dinâmica carregar, quantas instâncias do filtro existem, qual sua posição entre elas (*rank*), dados dos fluxos etc.

Atualmente, a implementação do Formigueiro permite que o número de filtros seja alterado em tempo de execução, mas ainda não há um mecanismo que realize tal mudança. O grande problema de uma alteração na configuração se deve a filtros com estado. Neste caso, se acontecer uma mudança de configuração no decorrer de um trabalho, o resultado da aplicação provavelmente será incorreto. Suponha, por exemplo, que uma execução da aplicação de contagem de cidades próximas (ver Capítulo 3) seja interrompida e alterada e que a responsabilidade de contar os dados de Minas Gerais tenha mudado de uma instância do filtro classificador i para outra j. Como a contagem já havia sido iniciada e não há comunicação entre instâncias de um mesmo filtro, a contagem do estado estará dividida. Para solucionar este problema, será necessário que os dados do estado sejam transmitidos entre instâncias de um mesmo filtro.

4.6 Reinício

Na seção 4.3, foram apresentadas a definição e a utilização do trabalho do Formigueiro. Uma das características desse trabalho é que o seu resultado depende apenas dele mesmo, não havendo manutenção de estados entre trabalhos distintos.

O Formigueiro implementa um mecanismo automático de reinício. Em caso de falha do sistema, os filtros são reiniciados. Em execuções em série de vários trabalhos, esse mecanismo funciona como uma tolerância a falhas, ao permitir que a execução seja retomada do último trabalho não finalizado.

O Formigueiro reiniciará os filtros atendendo a todos os requisitos passados no arquivo de configuração XML (instâncias sem requisitos podem ser disparadas em máquinas diferentes da primeira execução). Caso não seja possível atendê-los (devido a uma falha de uma máquina que continha os dados da execução, por exemplo), o sistema é finalizado, retornando um erro. Para evitar que programas defeituosos fiquem eternamente em execução, o reinício é feito apenas três vezes, para qualquer execução.

4.7 Resumo

Neste capítulo foi apresentado o Formigueiro, um sistema que permite a programadores paralelos o desenvolvimento de aplicações através do modelo filtro-fluxo. As aplicações têm disponíveis facilidades para a distribuição de dados e tarefas entre filtros, comunicação, notificação de falhas, instrumentação, verificação de término para aplicações com grafo direcionado cíclico e configuração do sistema distribuído. No Capítulo 5 serão apresentadas duas aplicações desenvolvidas sobre o Formigueiro.

Capítulo 5

Aplicações

Neste capítulo serão apresentadas duas aplicações intensivas em dados desenvolvidas sobre o Formigueiro. Serão mostrados os filtros utilizados e como o fluxo rotulado tornou possível distribuir a carga entre os nodos disponíveis.

5.1 Identificação de Conjuntos de Itens Frequentes em Bases de Dados Distribuídas

Essa aplicação foi desenvolvida por um grupo de pesquisadores da Universidade Federal de Minas Gerais e da Ohio State University. Mais detalhes podem ser encontrados em [27]. Neste texto, será mostrado como o Formigueiro permitiu a construção de uma aplicação escalável com manutenção de estado distribuído através do fluxo rotulado.

5.1.1 Descrição do Problema

Uma tarefa bastante comum em mineração de dados é a de encontrar o conjunto de itens (*itemsets*) frequentes, ou padrões, em bases de dados. Um conjunto de itens é frequente em uma base de dados se ele aparece mais vezes do que um dado número de transações (determinado pelo usuário) chamado suporte mínimo. Por exemplo, em uma base de compras de supermercado, cada compra pode ser considerada como uma transação. Deseja-se encontrar todos os produtos, ou combinações de produtos, cujas vendas superem o valor de suporte mínimo.

5.1.2 Algoritmo

O algoritmo inicialmente realiza a leitura de uma base distribuída de transações (B). Cada transação contém um conjunto de itens simples (*1-itemsets*).

O primeiro passo do algoritmo é construir a lista invertida de itens. Para cada *1-itemset* X da base, cria-se uma lista de identificadores de transações nas quais o item aparece. Um $k - itemset \quad X_k$, onde $k \geq 2$, pode ser frequente, se, e somente se, todos os subconjuntos $(k-1) - itemsets$ de X_k forem frequentes.

A interseção de duas listas invertidas de itens simples (X_i e X_j) nos fornece a lista do *2-itemset* contendo todas as transações com X_i e X_j. Similarmente, a interseção das listas de um *k-itemset* com a de um *l-itemset*, onde ambas sejam formadas por conjuntos de itens distintos, nos fornece uma nova lista de um *(k+l)-itemset*.

O tamanho da lista invertida de um conjunto de itens X, $|\sigma(X, B)|$ determina o suporte do conjunto. Minerar conjuntos de itens frequentes significa encontrar todos os conjuntos cujo suporte ultrapasse um valor mínimo passado pelo usuário (σ^{min}), ou seja, encontrar:

$$F(\sigma^{min}, B) = \{X_1, X_2, \ldots, X_N\} \qquad \Longleftrightarrow \qquad \sigma(X_i, B) > \sigma^{min}$$

Para realizar a mineração, calculam-se os conjuntos simples (*1-itemset*) e os frequentes encontrados são combinados entre si, em busca de conjuntos cada vez maiores.

5.1.3 Filtros

Existem três tarefas bem definidas para a execução da mineração de dados:

- leitura das transações e cálculo do suporte local de um candidato;
- soma dos suportes locais, encontrando o suporte global do candidato;
- descobrimento de novos candidatos a partir de conjuntos frequentes.

Nesta implementação, a verificação dos novos candidatos é realizada pelo mesmo filtro que calcula o suporte local. Portanto, é criado um grafo de dois filtros, Contador e Somador, conforme mostrado na Figura 5.1.

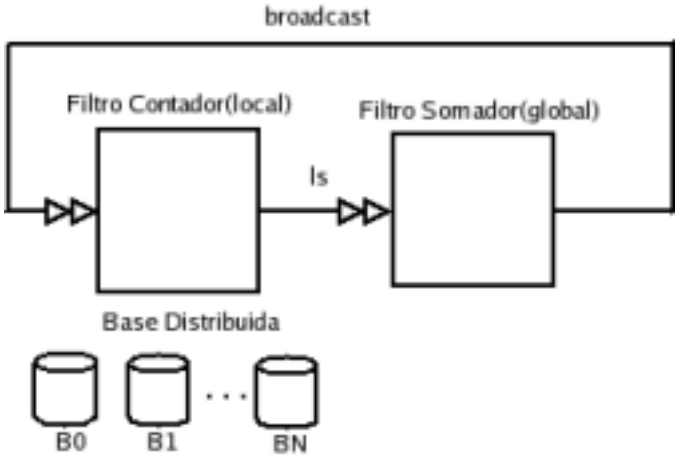

Figura 5.1: Grafo direcionado do programa de identificação de conjuntos de itens frequentes.

O algoritmo é iniciado com a contagem local dos *1-itemsets* da base. Os valores encontrados pelos Contadores são enviados ao Somador. A partir do segundo conjunto frequente, os Somadores encontram novos candidatos, enviados aos Contadores. Estes combinam as listas dos dois conjuntos envolvidos

para a verificação de novo *2-itemsets*. O algoritmo progride, combinando as listas de frequentes existentes para gerar conjuntos de items cada vez maiores, até que não haja mais candidatos.

Cada *1-itemset* é identificado unicamente entre todos os Contadores. A informação dos novos candidatos passada pelos Somadores é transmitada por difusão. Portanto, os candidatos novos são também criados com identificadores globais. Esta propriedade é imprescindível para que múltiplas instâncias do Somador sejam possíveis, como será visto na seção 5.1.3.1.

Note que a identificação de conjuntos frequentes não necessita de todos os resultados locais. Assim que a soma atingir o valor de suporte mínimo (σ^{min}), o conjunto já é identificado como frequente e novos candidatos contendo o mesmo conjunto podem ser criados. Esta otimização é conhecida como antecipação.

5.1.3.1 Fluxo Rotulado

Assim como mostrado na seção 3.3, a instanciação de múltiplos Contadores e Somadores pode levar a um resultado errado. Todos os valores locais de um conjunto de itens X devem ser enviados ao mesmo Somador, ou o conjunto poderá não ser considerado frequente, e, consequentemente, todos os conjuntos de itens derivados do mesmo. Este erro é ilustrado na Figura 5.2.

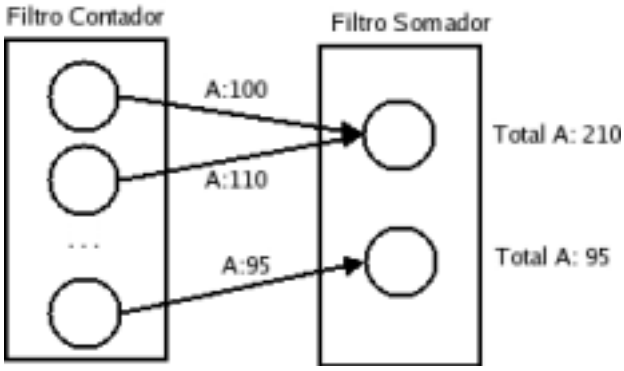

Figura 5.2: Erro na contagem global de ocorrências do conjunto de itens A.

O suporte mínimo passado pelo usuário é 250. O conjunto de itens representado por A aparece em 3 partições diferentes da base e cada filtro calcula seu resultado local. Uma das instâncias do filtro Contador envia seu resultado para a instância do filtro Somador errada e, como resultado, A é considerado infrequente. Na Figura 5.3, todos os dados referentes a A são enviados à mesma instância de cálculo global, obtendo-se o resultado correto.

Para permitir o funcionamento correto do algoritmo, é utilizado um fluxo rotulado, conforme apresentado na seção 3.3. O rótulo da mensagem é o próprio identificador do conjunto, que é global entre todos os Contadores. A função *hash*, aplicada sobre o mesmo rótulo, em qualquer instância de Contador, gerará o mesmo destino.

5.1.4 O Problema da Terminação

A condição de terminação do algoritmo se torna complexa para o caso de haver mais de um filtro Somador. O problema ocorre porque os conjuntos infrequentes não são transmitidos (por motivos de

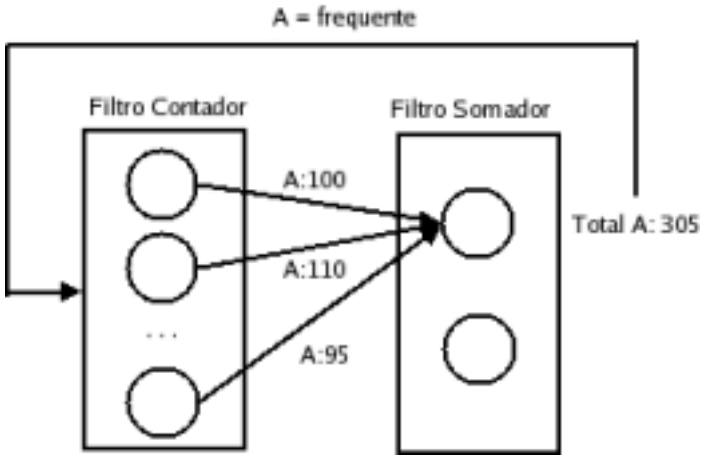

Figura 5.3: Todas as ocorrências de A são enviadas à mesma instância do Somador, gerando resultado correto (A é frequente).

desempenho) e também por causa do ciclo no grafo direcionado da aplicação.

Suponha o caso em que haja mais de uma instância de cada um dos filtros. Ao enviarem a contagem de todos os seus candidatos, os Contadores nunca sabem se os Somadores consideraram o conjunto infrequente, ou se ainda estão processando. Por sua vez, as instâncias deste último também não têm conhecimento se as instâncias do primeiro estão calculando novos itens ou não. Como resultado, chega-se a uma paralisação completa (*deadlock*), onde cada filtro espera por um resultado do outro.

Este problema foi tratado por um grupo de pesquisadores da Universidade Federal de Minas Gerais e da Universidade Federal do Rio de Janeiro e a solução implementada pode ser encontrada em [8].

5.2 Segmentação da Camada do Labirinto em Placentas

Essa aplicação tem como objetivo segmentar, em imagens microscópicas digitalizadas, a camada do labirinto de placentas de ratos. A aplicação foi parte de um trabalho desenvolvido pela Ohio State University em conjunto com a Universidade Federal de Minas Gerais [21, 19]. As seguintes características foram observadas nessa região:

- grande concentração de sangue, levando a um número maior de pixels vermelhos neste local;
- formato alongado, podendo a região ser confinada em uma faixa fina da imagem.

A Figura 5.4 mostra a imagem de uma placenta com a região do labirinto em destaque.

A implementação deste trabalho possui uma linha de três filtros principais e dois auxiliares para agregação de resultados. O grafo direcionado pode ser visto na Figura 5.5. O primeiro filtro, denominado RedFinder, identifica os pixels vermelhos na imagem. Os Contadores (filtro Counter) recebem os pixels vermelhos e fazem uma análise de vizinhança entre eles, buscando a maior área de densidade

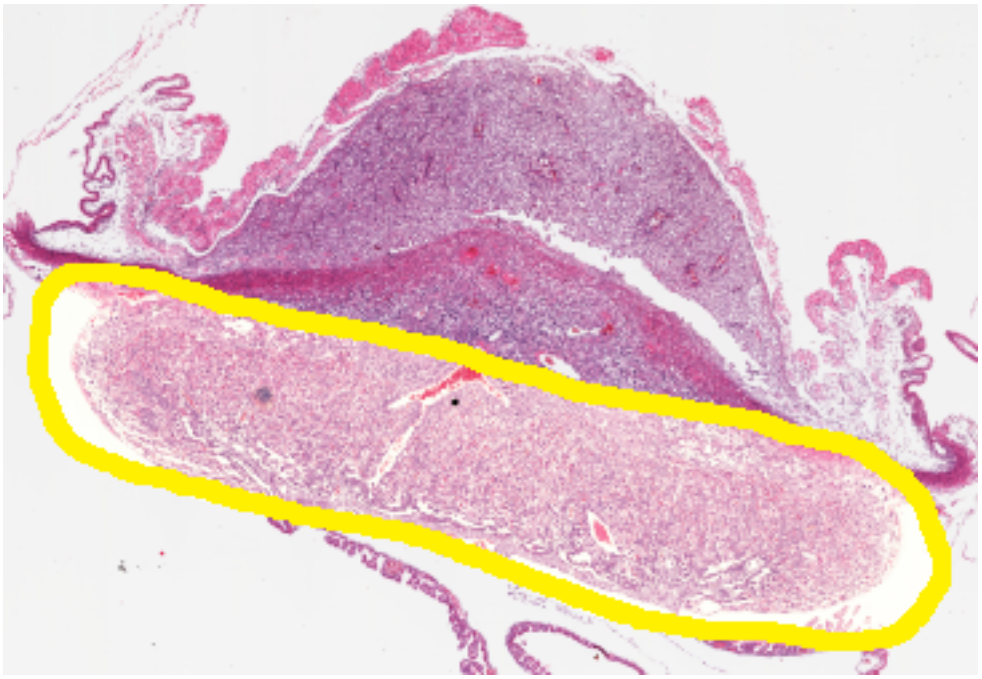

Figura 5.4: Um *slide* de placenta digitalizado e normalizado, com o labirinto em destaque.

semelhante na imagem. O terceiro estágio (filtro PCA) é responsável por encontrar, através da análise do componente principal, a faixa onde o labirinto está situado.

As imagens possuem resoluções, tonalidades e orientações distintas, tornando o problema ainda mais complexo. Imagens com tonalidades distintas dificultam o reconhecimento de pixels vermelhos pelos RedFinders, enquanto resoluções diferentes podem acarretar diferentes balanceamentos de carga entre os Contadores. As orientações distintas são tratadas pelas instâncias do filtro PCA.

Tanto o filtro Contador quanto o filtro PCA precisam de um filtro auxiliar para produzirem resultados globais. No caso do Contador, esse filtro auxiliar é um agregador de histograma, por isso chamado de HistAgg. Já o PCA precisa de um ponto central para seu cálculo, além de uma matriz auxiliar única entre todas as suas instâncias. Por isso existe o filtros PCAagg.

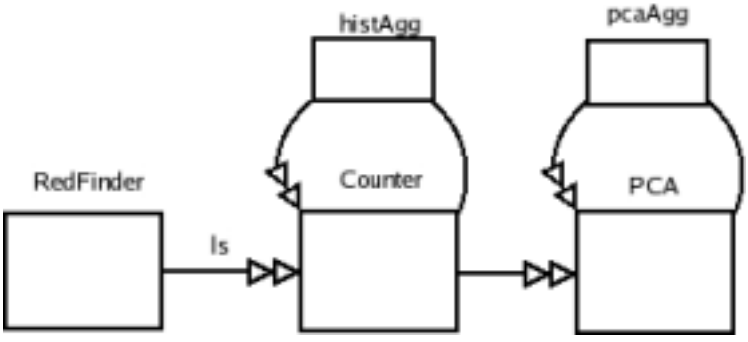

Figura 5.5: O grafo direcionado da aplicação da placenta.

Os estágios serão detalhados nas seções a seguir.

5.2.1 Particionamento das Imagens

Para o sistema funcionar, as imagens devem ser particionadas entre as máquinas onde os filtros RedFinders serão executados. Isso é necessário por se estar lidando com arquivos cujos tamanhos superam o limite do kernel utilizado nos experimentos (2GB). Portanto, o sistema de armazenamento é distribuído entre várias máquinas.

Foi então criado um programa de particionamento de imagens para o formato da aplicação. A imagem é partida em blocos de pixels de tamanho definido pelo usuário, sendo estes lidos de uma vez pelo filtro leitor. Para cada imagem lida, é gerado um arquivo de cabeçalho com os dados sobre a imagem particionada e vários arquivos que contém os blocos da mesma. Esse programa recebe uma imagem como entrada e parâmetros do usuário como tamanho do bloco, número de arquivos de blocos e fator de escala. Este último parâmetro foi usado apenas para se gerar uma imagem de tamanho grande (42GB), já que não havia imagens reais disponíveis para realizar tal teste. A Figura 5.6 ilustra o funcionamento do programa convertendo uma imagem para o formato da aplicação.

O resultado desta aplicação com múltiplas imagens menores pode ser encontrado em [21]. Nesta versão, foi utilizada uma imagem escalada para mostrar que é possível utilizar o programa com imagens cujos tamanhos superem as memórias das máquinas envolvidas (imagens *out-of-core*). A demanda por imagens de altíssima resolução (140000 x 140000) pode acontecer em um futuro próximo, com a evolução da tecnologia de coleta de dados e armazenamento da informação.

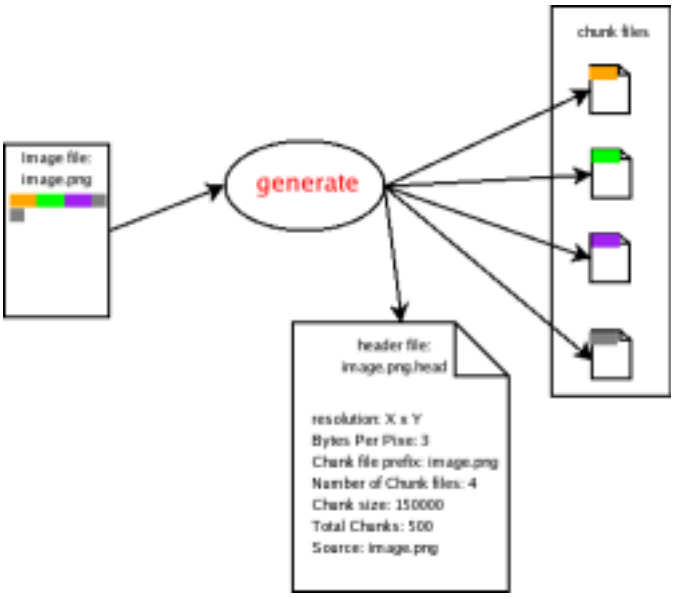

Figura 5.6: Particionamento das imagens para a aplicação.

O arquivo de cabeçalho contém metadados da imagem necessários à execução do programa. A Listagem 5.1 mostra o formato do arquivo. A resolução (resolution) é passada na primeira linha, seguida da profundidade de cores dos pixels (BPP). O parâmetro Chunk file prefix indica o prefixo dos arquivos de blocos de dados. Se são gerados N arquivos de blocos (valor de Number of Chunk files), eles são nomeados prefixo .0, prefixo .1, ... prefixo .(N-1). O tamanho dos blocos (TB) é passado através de Chunk size (valor em número de pixels). O tamanho físico do bloco, em bytes, é dado por $BPP * TB$.

O valor de Total chunks indica quantos blocos foram gerados no total, e Source indica qual o nome da imagem que gerou os dados.

Listagem 5.1: Arquivo de cabeçalho

```
resolution: 1602 x 1167
Bytes Per Pixel: 3
Chunk file prefix: 04-4351_603.he.2.png.chk
Number of Chunk files: 2
Chunk size: 100000
Total chunks: 19
Source: 04-4351_603.he.2.png
```

5.2.2 Filtro RedFinder (Identificador de Pixels Vermelhos)

Os RedFinders são responsáveis por ler os blocos gerados pelo programa apresentado em 5.2.1. Cada instância deste filtro é responsável por ler um conjunto de blocos, não havendo interseção entre eles. Todos os blocos devem ser lidos. O ideal seria que cada RedFinder fosse responsável por um único arquivo de blocos, mas a aplicação aceita qualquer combinação *número de instância x quantidade de blocos*.

Ao ler um bloco, o RedFinder encontra todos os pixels vermelhos presentes e os envia para o próximo filtro, estando pronto para ler mais blocos. Um pixel p, pertencente a imagem I, é vermelho se, e somente se, seu valor de vermelho, p_v, estiver acima de um limite mínimo, e seu valor de azul, p_a, estiver abaixo de um limite máximo. A função booleana[1] $v(p)$ indica se um pixel é vermelho ou não. Os valores V e A são parâmetros passados ao usuário do programa, definindo o valor mínimo de vermelho e máximo de azul, respectivamente. Normalmente são usados valores entre 170 e 200 para V, e 80 e 130 para A (255 é o máximo).

$$v(p) = 1 \iff p_v > V \wedge p_a < A, \qquad \forall p \in I$$

Note que os valores de verde são ignorados, já que pelas características morfológicas da placenta e dos corantes utilizados para obter os slides, estes valores podem ser desprezados.

Apenas os pixels considerados vermelhos são enviados ao próximo estágio, sendo esta a primeira filtragem da imagem pela aplicação. O envio dos pixels sobreviventes é feito através de um fluxo rotulado, pois o próximo filtro demanda que todos os pixels vermelhos vizinhos de um pixel p_v qualquer estejam na mesma instância que recebe p_v, como será visto na seção 5.2.3.3. Note que após esta primeira filtragem, serão enviadas apenas as coordenadas dos pixels aos Contadores, já que a cor só interessa à aplicação para determinar se o pixel é vermelho ou não.

5.2.3 Filtro Counter (Contadores de Vizinhos)

A função deste filtro é retirar todos os pixels de áreas que tenham densidade diferente da maioria dos pixels. Para isso, ele computa o número de vizinhos vermelhos ($nv(p)$) que cada pixel vermelho da

[1] Neste texto o valor 1 será utilizado para denotar verdadeiro e 0 para falso, a fim de evitar confusões com parâmetros passados pelo usuário, como V (valor mínimo de vermelho).

imagem possui. Um pixel vermelho p_i é considerado vizinho de outro pixel vermelho p_j se a distância (d) entre eles é menor do que um parâmetro (D) passado pelo usuário.

$$viz(p_i, p_j) = 1 \iff d(p_i, p_j) \leq D$$

Feito o cálculo de vizinhos para cada pixel, é criado um gráfico de densidade, através do qual é possível encontrar a densidade mais comum em toda a imagem. Pixels com número de vizinhos próximo desta densidade são passados ao próximo estágio.

5.2.3.1 Múltiplas Instâncias

Para a utilização de múltiplas instâncias de contadores, a imagem foi dividida em blocos retangulares, ficando cada Contador responsável por um ou mais blocos. Cada bloco mostrado na Figura 5.7 é representado por uma árvore 2-D (*kdTree* [22], onde $k = 2$).

A árvore 2-D otimiza operações de cálculo de vizinhança entre os pontos, descartando vários pontos em cada comparação realizada. No entanto, caso os pixels sejam inseridos em sequência, a árvore acaba se tornando uma lista encadeada, e seu desempenho cai. Para evitar este efeito, os pixels de um mesmo bloco são armazenados em um *buffer*, que é embaralhado ao atingir determinada capacidade. Após o embaralhamento, os pixels são inseridos em suas árvores.

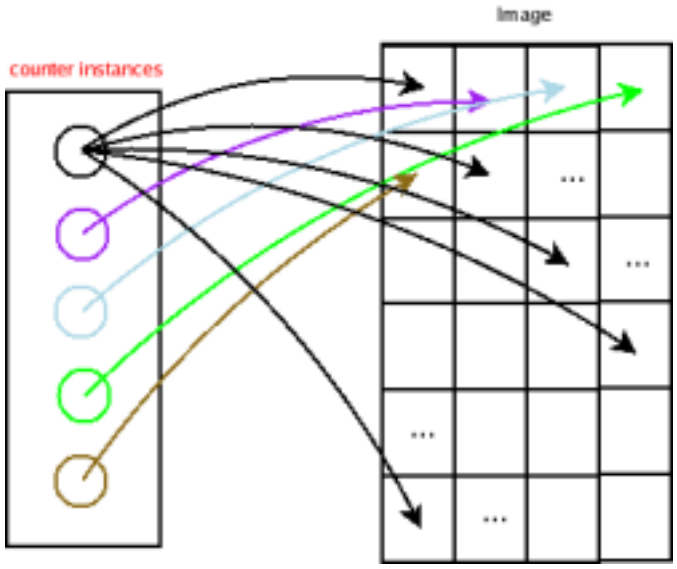

Figura 5.7: Distribuição de blocos entre os contadores.

A divisão de blocos adiciona outra dificuldade ao problema. Um pixel pertencente a um bloco depende de pixels de outros blocos para sua contagem de vizinhos, como será visto na seção 5.2.3.2. Como blocos vizinhos pertencem a Contadores diferentes, os pixels pertencentes à borda dos blocos devem ser enviados a múltiplos contadores, como mostrado na seção 5.2.3.3.

Os limiares de corte da densidade devem ser os mesmos para todos os os contadores, por isso a necessidade do filtro agregador, descrito na seção 5.2.3.4.

Existe um compromisso entre o tamanho do bloco e a sobrecarga de armazemanto. Quanto menor o bloco, mais balanceados serão os contadores, já que o número de pixels enviados a cada contador tende a se igualar. O tamanho reduzido das árvores também tende a uma inserção de pixels mais eficiente e a cálculos rápidos. Por outro lado, mais árvores ocupam mais espaço em memória, aumentando o número de bordas de blocos e gerando mais comunicação entre RedFinders e Contadores. O valor ideal do tamanho do bloco varia de imagem para imagem, mas empiricamente, chegou-se à conclusão que um valor até 10 vezes o valor da distância (D) passada pelo usuário resulta em um bom compromisso.

5.2.3.2 Pixels Nativos e Pixels de Borda

Para facilitar o entendimento do problema da divisão de blocos, serão definidos os conceitos de pixel nativo e pixel de borda de um bloco.

Define-se um pixel p como *nativo* a B aquele situado dentro dos limites do bloco B. A função booleana $pn(p, B)$ representa este conceito, sendo verdadeira se, e somente se, um pixel p, de coordenadas (p_x e p_y), pertencente à imagem I, é nativo ao bloco B, de coordenadas superior esquerda (b_{xes}, b_{yes}) e inferior direita (b_{xdi}, b_{ydi}).

$$pn(p, B) = 1 \iff p_x \geq b_{xes} \wedge p_x \leq b_{xdi} \wedge p_y \geq b_{yes} \wedge p_y \leq b_{ydi} \quad \forall p \in I$$

Chama-se de *pixel de borda* de um bloco B àquele que não seja nativo a B e esteja a uma distância da borda de B menor ou igual à distância de vizinhança entre pixels (D). Esse conceito é representado pela função booleana $pb(p, B)$:

$$pb(p, B) = 1 \iff pn(pb) = 0 \wedge d(p, B) \leq D \qquad \forall p \in I$$

Os contadores precisam dos pixels nativos e de borda de um bloco para computarem $nv(p)$ para todos os seus pixels (p) nativos. Este problema é ilustrado na Figura 5.8. Nela, podem-se ver quatro pontos, A, B, C e D. A região sombreada representa a borda entre os blocos e tem largura $D \times 2$. Os pixels A, B e D são nativos ao bloco 1, e B é um pixel de borda do bloco 2. O pixel C é nativo ao bloco 5 e de borda dos blocos 1, 2 e 6. Para computar $nv(D)$, o contador responsável por 1 necessita do pixel de borda C, nativo ao bloco 5, pois $viz(D, C) = 1$.

5.2.3.3 Difusão Seletiva e o Problema da Borda

Quando um RedFinder encontra um pixel vermelho p, tal que $pn(p, B_i) = 1$, o mesmo deve ser enviado ao contador responsável pelo bloco B_i. Caso o pixel também seja de borda de outro bloco B_j, o contador responsável por B_j também deve recebê-lo.

Quem realiza este roteamento é uma função especial de fluxo rotulado múltiplo, conhecida também como difusão seletiva (política mls). Portanto, quando o RedFinder escreve um pixel p para o fluxo, é aplicado a ele uma função de descobrimento de destino $dd(p)$, que retorna um conjunto de blocos que necessitam do pixel p.

$$dd(p) = B_0 \cup B_1 \cup \ldots \cup B_n \quad | \quad pn(p, B_i) = 1 \vee pb(p, B_i) = 1$$

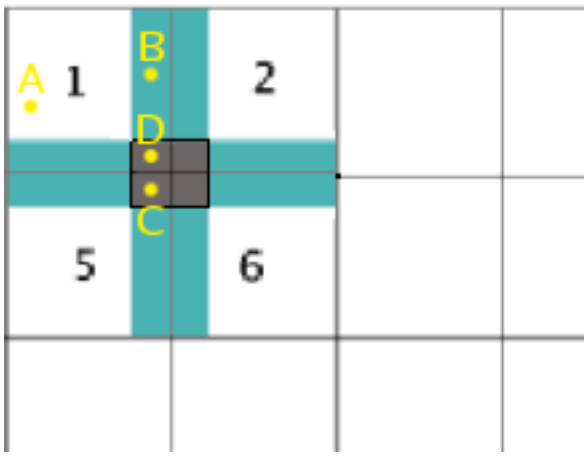

Figura 5.8: Pixels nativos e de borda: os pontos A, B e D são nativos ao bloco 1. A região sombreada representa a borda dos blocos, portanto o ponto C é nativo ao bloco 5 e de borda para os blocos 1, 2 e 6, enquanto B é de borda ao bloco 2 e D é de borda para 2, 5 e 6.

5.2.3.4 Histograma de Densidade e Filtro HistAgg (Agregador de Histogramas)

Ao término do cálculo do número de vizinhos, cada Contador cria um histograma de densidade. Este histograma pode ser visto como um gráfico de duas dimensões, cujo eixo X representa o número de vizinhos vermelhos (v) que um pixel vermelho pode ter e o eixo Y (função h) representa o número de pixels vermelhos que possuem aquele número de vizinhos de sua coordenada X. Por exemplo, se $h(3) = 5$, apenas 5 pixels vermelhos possuem 3 vizinhos vermelhos.

Para obter o histograma da imagem, basta somar os histogramas locais. Cada instância do Contador cria o histograma de cada um de seus blocos e depois os soma, obtendo o histograma de todos os seus blocos. O Contador armazena em cada v do histograma uma referência para os pixels com aquele número de vizinhos, para que estes possam ser identificados rapidamente caso v esteja na faixa de interesse da aplicação.

Para obter a faixa de interesse, deve-se somar os histogramas de todas as instâncias de Contadores. Através do pico deste gráfico global, encontra-se a densidade mais comum na imagem.

O filtro Agregador de Histogramas recebe todos os histogramas e os soma. Após a agregação, o histograma global é suavizado, a fim de evitar picos e vales repentinos. Em seguida, encontra-se a faixa de interesse da aplicação, ou seja, os pixels que têm um número de vizinhos próximo à maioria.

Defe-se $h'(v)$ como a função $h(v)$ após a suavização para evitar os desníveis súbitos da mesma. Uma vez feito $h'(x)$, aplica-se um filtro passa-faixa (veja Figura 5.9), retirando-se da função as regiões de densidade diferente da maioria. Parte-se do pico do gráfico para ambos os lados até se encontrarem os limites esquerdo (le) e direito (ld), cujos valores estão abaixo de 30% do valor do pico. A faixa de interesse é delimitada por estes limites.

Os Agregadores de Histograma enviam a faixa de interesse de volta aos contadores, que recuperam, através de suas referências, todos os pixels contidos na faixa de interesse de seus histogramas. Todos os pixels recuperados que forem nativos são enviados para o próximo estágio. A filtragem dos contadores é definida pela função booleana $c(p)$ que é verdadeira se, e somente se, o pixel p, nativo ao bloco B, sob responsabilidade do contador C, é passado adiante.

Figura 5.9: Agregação de histogramas, suavização e corte.

$$c(p) = 1 \iff le \leq nv(p) \leq ld \quad | \quad B \in C, pn(p, B) = 1$$

5.2.4 Filtro PCA (Análise do Componente Principal)

Neste ponto da linha de processamento, existem apenas pontos vermelhos de regiões com densidades bastante similares. A grande maioria destes pontos pertence à região do labirinto. Este filtro elimina os pontos fora dessa região.

A região do labirinto possui forma alongada, conforme mostrado na Figura 5.4. A análise do componente principal, também conhecida como transformada de Hotelling ou discreta de Karhunen-Loeve[16], nos permite encontrar a orientação do labirinto com erro desprezível, já que a grande maioria dos pixels da imagem estão presentes nesta faixa.

Encontrada a orientação, realiza-se a mudança para o eixo baseado no centróide da imagem rotacionada pelo ângulo encontrado pelo PCA, de maneira que a camada do labirinto fique paralela ao eixo y. Após a rotação, os pontos são projetados no eixo x, gerando-se o gráfico $p(x)$ (se $p(100) = 200$, há duzentos pontos com abscissa 100 no novo sistema de coordenadas).

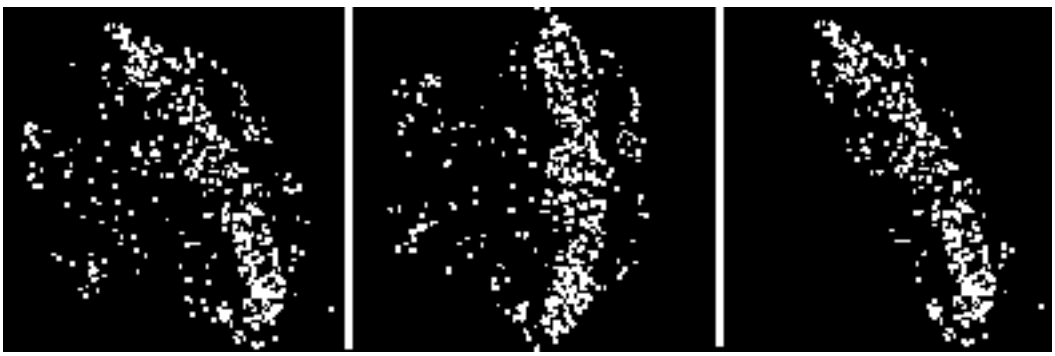

Figura 5.10: Estágios do PCA: pontos recebidos, rotação no novo eixo e corte por limiar (com rotação de volta para o eixo original).

A filtragem realizada pelo filtro PCA pode ser vista na Figura 5.10. À esquerda, os pontos são recebidos pelo filtro. No meio, o ângulo de rotação foi encontrado e os pontos foram rotacionados

para o novo sistema de coordenadas. À direita, após projeção e corte por limiar, vêm-se os pontos sobreviventes.

Após a projeção no eixo x, obtém-se a função $p(x)$ que, suavizada para evitar desníveis súbitos, gera $p'(x)$. Achando-se o pico ($p'(x_p)$), encontra-se o limiar de corte (L_c), que corresponde a 30% do pico. Partindo-se de x_p em ambas as direções, encontram-se os limiares esquerdo (l_e) e direito (l_d), onde $p(l_e) < L_c$ e $p(l_d) < L_c$. A filtragem do filtro PCA pode ser, então, definida como a função booleana $fp(p')$, que é verdadeira se o pixel está contido na faixa do labirinto:

$$fp(p) = 1, \iff l_e < p'_x < l_d \tag{5.1}$$

onde p'_x corresponde à coordenada X do pixel após a transformada.

5.2.4.1 PCA Distribuído

Assim como os Contadores, este filtro pode ter seu desempenho melhorado através de múltiplas cópias transparentes. Novamente, alguns dados globais são necessários para o cálculo do componente principal como, por exemplo, a média dos pontos. O filtro auxiliar, PCAAgg, é responsável por agregar as informações dos PCAs.

O algoritmo utilizado nesta implementação distribuída é o seguinte: dados os pontos $p_i = [x_i, y_i]^T \quad \forall i = 1, \ldots, N$, o ponto médio é calculado por:

$$p_m = \begin{bmatrix} p_{mx} \\ p_{my} \end{bmatrix} = \frac{1}{N} \sum_{i=1}^{N} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

A matriz dos pontos transladados ao centro é dada por

$$D = \begin{bmatrix} x_1 - p_{mx} & \ldots & x_N - p_{mx} \\ y_1 - p_{my} & \ldots & y_N - p_{my} \end{bmatrix} \tag{5.2}$$

Ao invés de calcular a decomposição do valor singular (SVD) de D, é utilizado outro método equivalente para o cálculo do ângulo da componente principal em paralelo. Dada a matriz:

$$E = DD^T = \begin{bmatrix} a & b \\ b & c \end{bmatrix} = \begin{bmatrix} \sum (x_i - p_{mx})^2 & \sum (x_i - p_{mx})(y_i - p_{my}) \\ \sum (x_i - p_{mx})(y_i - p_{my}) & \sum (y_i - p_{my})^2) \end{bmatrix} \tag{5.3}$$

o maior autovalor de E, é dado por:

$$e_m = \frac{(a + c) + \sqrt{(a - c)^2 + 4b^2}}{2} \tag{5.4}$$

o autovetor correspondente a e_m é dado por:

$$a_m = [v, w]^T = \begin{bmatrix} \frac{b}{e_m - a} \\ 1 \end{bmatrix} \tag{5.5}$$

Se $e_m = a$, $a_m = [1, 0]^T$.

O ângulo entre a direção do componente principal e o eixo Y é dado por:

$$\theta = arctan(\frac{w}{v} - \frac{\pi}{2}) \tag{5.6}$$

A compensação de 90 graus acontece porque o ângulo encontrado se refere ao eixo X.

5.2.4.2 Agregador de PCA

A interação entre os filtros PCA e o filtro agregador é mostrada no diagrama da Figura 5.11.

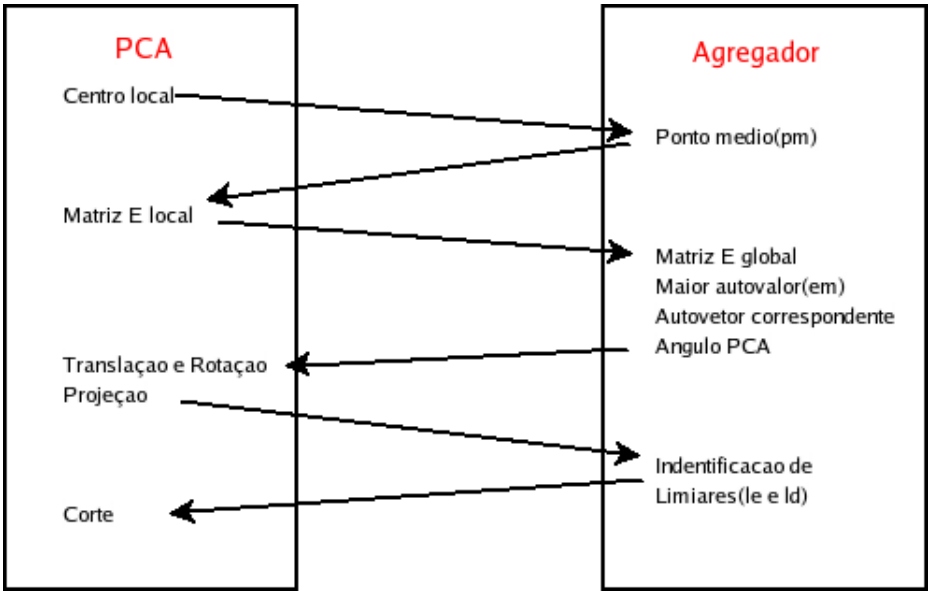

Figura 5.11: Execução do PCA distribuído.

1. Cada filtro PCA encontra seu centro local e envia para o agregador uma mensagem tipo (pm_{loc}, n), onde pm_{loc} é o ponto médio local e n o número de pontos pertencentes ao filtro.
2. O agregador soma todos os pontos médios recebidos e divide pela soma do número de pontos, obtendo o ponto médio global (p_m).
3. p_m é utilizado pelos PCAs para transladar todos seus pontos, como mostrado na equação 5.2.
4. PCAs constroem a matriz E (equação 5.3) e as enviam aos agregadores.
5. O agregador, após somar as matrizes recebidas, calcula o maior autovalor (e_m, equação 5.4) e seu autovetor correspondente (a_m, equação 5.5). Então, o ângulo entre o eixo x e o vetor do componente principal é calculado segundo a equação 5.6.
6. Os PCAs recebem o ângulo, rotacionam seus pontos e projetam no eixo x, enviando o gráfico resultante ao agregador.
7. O agregador identifica os limiares esquerdo (l_e) e direito (l_d) e os envia aos filtros PCAs.

8. Os filtros PCAs realizam o corte dos pontos fora dos limiares, como mostrado na equação 5.1.

Após o corte, apenas os pontos pertencentes à região do labirinto permanecem e são enviados para alguma aplicação seguinte ou armazenados em um banco de dados.

5.3 Resumo

Neste capítulo foram apresentadas uma aplicação de mineração de dados e outra de segmentação de imagens. As aplicações têm características bastante distintas: a execução da primeira segue um grafo direcionado cíclico, enquanto a da segunda possui uma linha alongada, com ciclos de uma iteração apenas. As duas aplicações exemplificam o uso do fluxo rotulado e do fluxo rotulado múltiplo, respectivamente. Apesar de suas diferenças, no Capítulo 6 será mostrado que ambas apresentam desempenho superlinear em condições ideais.

Capítulo 6

Resultados

Neste capítulo serão apresentados os resultados obtidos com as aplicações, mostradas no Capítulo 5, que utilizaram o Formigueiro como base. Esses resultados mostram que o sistema suporta aplicações intensivas em dados distribuídos e que o desempenho melhora à medida em que novos recursos são adicionados.

Os resultados da aplicação de mineração de dados podem ser encontrados em [27]. Além disso, em [8], podem ser encontrados resultados de outros algoritmos de mineração de dados (árvore de decisão [id3] e agrupamento [k-means]) que utilizaram o Formigueiro como base. A aplicação de segmentação de placentas possui outras duas versões, uma delas trabalha com múltiplas imagens [21] e a outra interage com o banco de dados Mobius [19].

6.1 Identificação de Conjuntos de Itens Frequentes em Bases de Dados Distribuídas

A análise de desempenho deste algoritmo foi feita em relação à distribuição de dados, tamanho dos dados e grau de paralelismo. Foram utilizadas três bases de transações para a execução da aplicação. A primeira base é real, denominada KOSARAK (K), contendo aproximadamente 900.000 transações de um portal de notícias húngaro, enquanto as outras duas são sintéticas, geradas conforme [2] e de tamanhos diferentes (560MB [S_1] e 2,2GB [S_2]).

A distribuição dos dados foi realizada de duas maneiras distintas. Em ambos os casos, são criados blocos de dados de tamanhos iguais. A diferença está na maneira como as transações são distribuídas entre os blocos.

Na Distribuição Aleatória (D_R), as transações são distribuídas entre os blocos aleatoriamente, enquanto na Distribuição Original (D_O), os dados são simplesmente particionados em blocos de tamanhos iguais, mantendo as irregularidades originais da base.

O problema da Distribuição Original é que a possibilidade de algumas partições terem conjuntos de itens frequentes é maior do que na aleatória (*data skewness*), levando a um desbalanceamento de carga.

A tabela 6.1 sumariza os resultados encontrados para esta aplicação. O valor de $\mu_{p,q}$ mostra a eficiência paralela da execução realizada com q máquinas em relação à execução com p. Uma

Base de Dados	Distribuição	Antecipação	τ_8 (s)	$\mu_{8,16}$	$\mu_{16,32}$
S_1	D_O	NÃO	126,38	1,07	0,88
S_1	D_O	SIM	123,11	1,07	1,06
S_1	D_R	NÃO	94,93	1,02	0,89
S_1	D_R	SIM	92,13	1,02	0,99
S_2	D_O	NÃO	194,74	1,05	0,98
S_2	D_O	SIM	188,18	1,07	1,07
S_2	D_R	NÃO	168,35	1,00	0,97
S_2	D_R	SIM	165,96	1,02	1,01
K	D_O	NÃO	642,82	0,96	0,85
K	D_O	SIM	639,14	1,00	0,95
K	D_R	NÃO	640,87	0,97	0,94
K	D_R	SIM	639,81	0,99	0,95

Tabela 6.1: Resultados do algoritmo de mineração de dados a-priori.

eficiência paralela igual a 1 indica 100% de aproveitamento, enquanto valores acima de 1 indicam desempenho superlinear. A eficiência paralela é dada por:

$$\mu_{p,q} = \frac{\tau_p \times p}{\tau_q \times q}$$

onde τ_p é o tempo da execução para p processadores, e p e q o número de processadores utilizados.

Os resultados mostram claramente que o algoritmo escalou de maneira satisfatória diante do aumento do número de máquinas envolvidas, atingindo desempenho superlinear em alguns casos.

6.2 Segmentação da Camada do Labirinto em Placentas

Na aplicação da placenta, foi utilizado um agrupamento de 23 computadores Pentium III, 933MHz, cada um com 512MB de memória primária e 3 discos IDE, com capacidade entre 100GB e 120GB. Foi escolhida uma imagem de placenta aleatoriamente do grupo de imagens disponíveis e esta foi redimensionada, a fim de se obter uma imagem de 42GB. A imagem foi dividida em 32 arquivos de blocos que, por sua vez, foram distribuídos entre as máquinas que executam o filtro RedFinder.

Os parâmetros utilizados para o experimento foram:

- máximo de azul (A): 130
- mínimo de vermelho (V): 170
- distância máxima entre vizinhos (D): 25
- tamanho do bloco dos Contadores: 250x250 pixels

Os experimentos foram executados com três configurações diferentes e cada configuração foi executada três vezes, a fim de se obter um resultado consistente. Em todas as configurações, variou-se apenas o número de Contadores e PCAs, sendo fixo o número de RedFinders (8) e o de Agregadores (1 de cada).

Em primeiro lugar, será mostrada a quantidade de dados que é passada por cada estágio. Inicialmente, são lidos 42GB de dados. A Figura 6.1 mostra a quantidade de dados que sobrevive após cada

estágio. Note que a maior filtragem ocorre justamente no primeiro filtro que descarta pouco mais de 40GB do total dos dados. Este valor varia com a escolha dos limiares de azul (A) e vermelho (V).

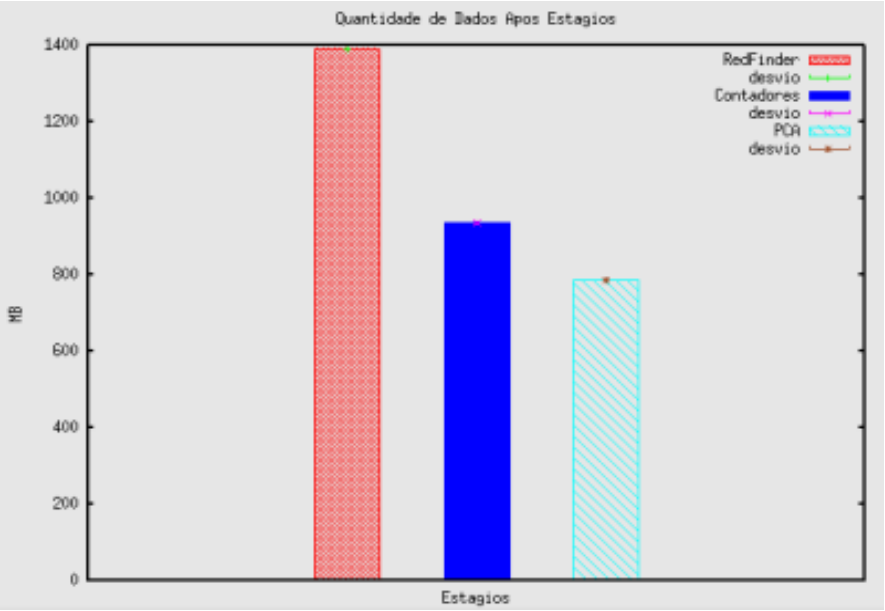

Figura 6.1: Dados após cada estágio da execução.

Os tempos individuais dos filtros são mostrados a seguir. Como o número de RedFinders não foi variado, o resultado das diferentes execuções não variou significativamente para esse filtro. Para os Contadores e PCA, serão apresentados os resultados para as três configurações empregadas. As Figura 6.2, 6.3 e 6.4 mostram os resultados para os três filtros. Os tempos são apresentados como uma média dos filtros, e o desvio padrão dos valores é mostrado como uma linha nas barras. Quanto menor o desvio, maior o balanceamento de carga entre os filtros.

Os tempos do RedFinder são divididos entre os estágios de: leitura dos dados em disco, identificação de vermelhos e escrita dos pixels vermelhos encontrados para o próximo filtro. Este último tempo tende a ser pequeno mas, quando os Contadores não são rápidos o suficiente para receber os dados, ocorre contenção de envio e esse tempo sobe. Como esperado, para uma quantidade de dados grande, o tempo de leitura de dados acaba se tornando o maior de todos para este filtro.

Os Contadores têm dois tempos significativos: a inserção dos dados nas árvores 2-D e o cálculo do número de vizinhos que cada pixel possui. A maior parte da execução do programa ocorre neste estágio durante o cálculo do número de vizinhos. No caso da imagem escalonada, por ser artificial, este cálculo se torna ainda mais caro (um pixel original gera vários pixels vizinhos). Ainda assim, é possível notar que o tempo é bastante sensível ao número de filtros, mostrando que a implementação com fluxo rotulado desempenha bem seu papel.

O tempo de execução dos PCAs é surpreendentemente rápido. No entanto, nota-se que o filtro não escala aumentando-se o número de suas instâncias. Pela análise do desvio, vê-se um grande desbalanceamento de carga entre as instâncias.

A Figura 6.5) nos mostra uma planilha com a distribuição de filtros das três configurações utilizadas no experimento. Cada configuração é representada por uma grande coluna cujo nome tem

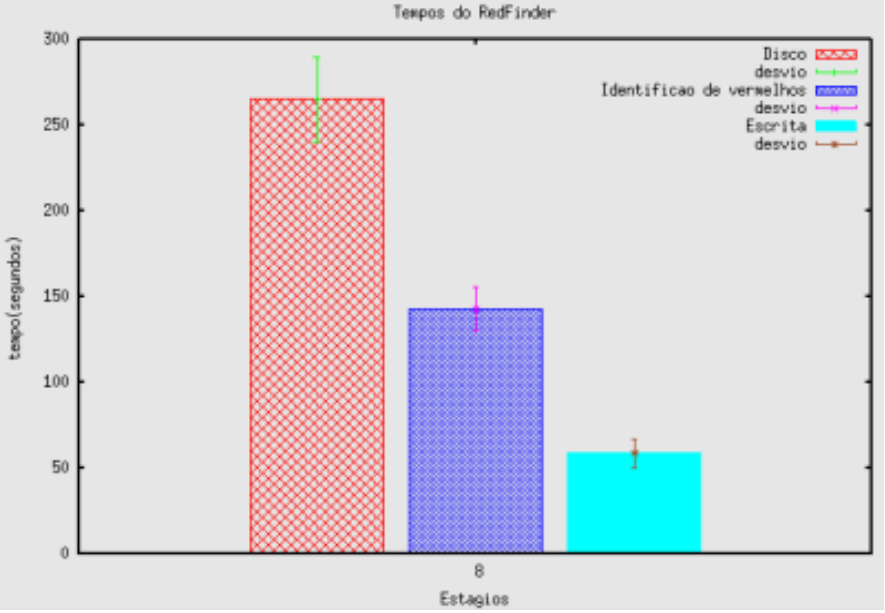

Figura 6.2: Tempos dos RedFinders.

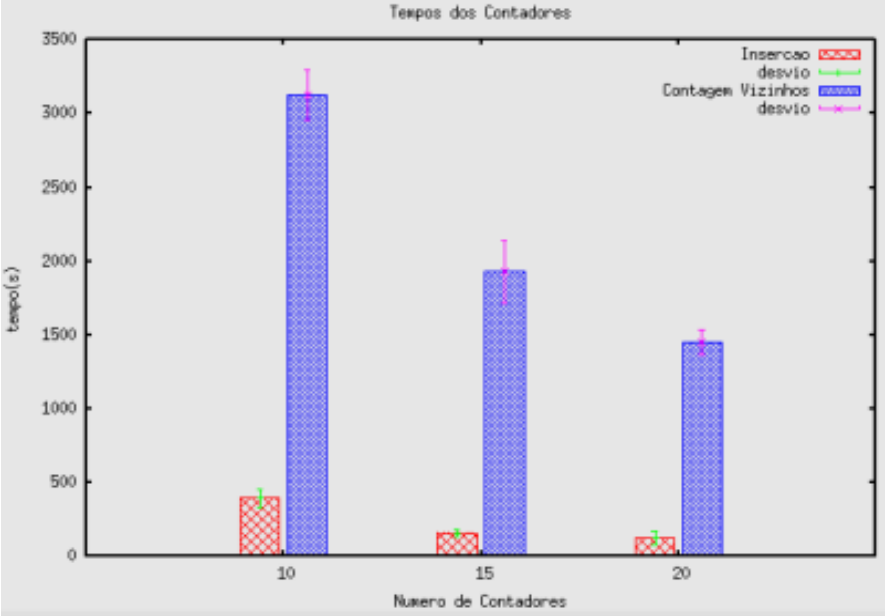

Figura 6.3: Tempos dos Contadores nas três configurações.

o formato R-C-H-P-A, onde 8 é o número de RedFinders (R) utilizado, C o número de Contadores, P o número de PCAs e os dois números 1 que aparecem referem-se aos Agregadores de Histograma (H) e Agregadores de PCAs (A). Por exemplo, a coluna 8-10-1-10-1 refere-se à execução com 8 RedFinders, 10 Contadores e 10 PCAs, além dos dois agregadores.

Dentro de cada uma dessas colunas, as colunas menores R, C, H, P e A indicam onde as instâncias de cada um dos filtros foi executada. Por exemplo, na execução 8-10-1-10-1, pode ser observado que os nodos 1 a 8 executaram uma instância de RedFinder, enquanto os nodos 9 a 18 executaram instâncias

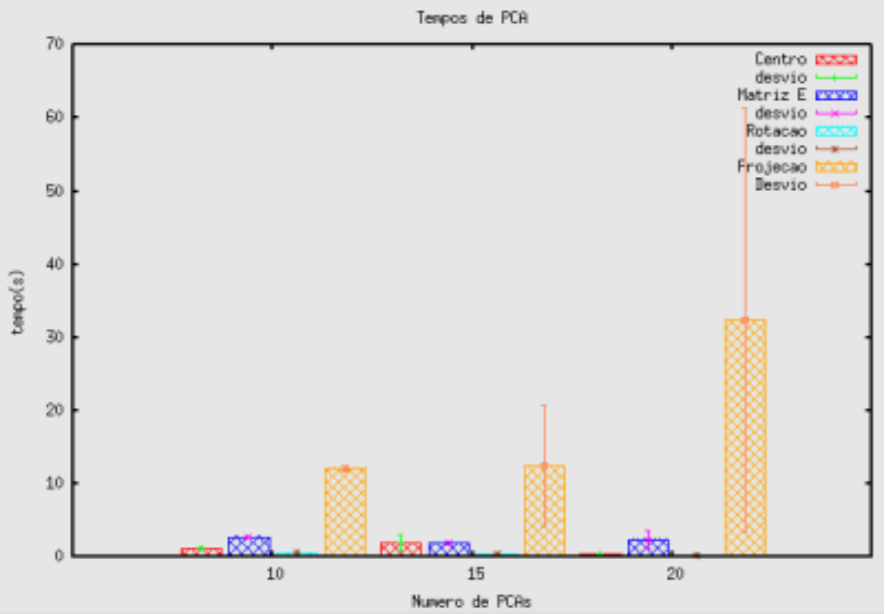

Figura 6.4: Tempo dos PCAs, nas três configurações.

de Contadores. As colunas intituladas por C+P mostram quantas instâncias de Contadores ou PCAs foram executadas naquele nodo. Por exemplo, na execução 8-15-1-15-1, pode-se ver que os nodos 9 a 17 executaram uma instância de cada um desses dois filtros. Por último, a coluna All mostra quantas instâncias, de qualquer filtro, foram executados naquele nodo. Por exemplo, na execução 8-20-1-20-1, os nodos 1 a 6 executaram 3 instâncias de filtros.

	8-10-1-10-1							8-15-1-15-1							8-20-1-20-1						
Node	**R**	**C**	**H**	**P**	**A**	**C+P**	**All**	**R**	**C**	**H**	**P**	**A**	**C+P**	**All**	**R**	**C**	**H**	**P**	**A**	**C+P**	**All**
1	1			1		1	2	1	1				1	2	1	1		1		2	3
2	1			1		1	2	1		1			0	2	1	1		1		2	3
3	1			1		1	2	1			1		1	2	1	1		1		2	3
4	1			1		1	2	1			1		1	2	1	1		1		2	3
5	1			1		1	2	1			1		1	2	1	1		1		2	3
6	1			1		1	2	1			1		1	2	1	1			1	1	3
7	1			1		1	2	1			1		1	2	1		1			0	2
8	1				1	0	2	1			1		1	2	1			1		1	2
9		1				1	1		1		1		2	2		1		1		2	2
10		1				1	1		1		1		2	2		1		1		2	2
11		1				1	1		1		1		2	2		1		1		2	2
12		1				1	1		1		1		2	2		1		1		2	2
13		1				1	1		1		1		2	2		1		1		2	2
14		1				1	1		1		1		2	2		1		1		2	2
15		1				1	1		1		1		2	2		1		1		2	2
16		1				1	1		1		1		2	2		1		1		2	2
17		1				1	1		1		1		2	2		1		1		2	2
18		1				1	1		1			1	1	2		1		1		2	2
19			1			0	1		1				1	1		1		1		2	2
20				1		1	1		1				1	1		1		1		2	2
21				1		1	1		1				1	1		1		1		2	2
23				1		1	1		1				1	1		1		1		2	2
Tt Inst	8	10	1	10	1			8	15	1	15	1			8	20	1	20	1		

Figura 6.5: Distribuição dos filtros entre as máquinas.

Observando a planilha, nota-se que o aumento do número de instâncias dos filtros acarretou a

execução de mais de uma instância de filtro a um mesmo nodo. No caso do filtro RedFinder, isso não é tão problemático, já que, enquanto ele está lendo, os filtros PCAs estão ociosos (os Contadores só podem passar seus dados adiante após terem recebido todos os pontos). No entanto, a execução simultânea de instâncias de PCAs e Contadores no mesmo nodo acabam por degradar o desempenho do filtro, pois ambos utilizam o mesmo recurso (são caros computacionalmente).

Tendo em vista esse fato, foi feita a análise dos tempos de instâncias de PCAs que executaram em nodos sem Contadores. O resultado é mostrado na Figura 6.6 e pode-se notar, que nestas circunstâncias, não só o filtro escala como esperado como também o desvio padrão entre as instâncias é bem menor, indicando bom balanceamento de carga.

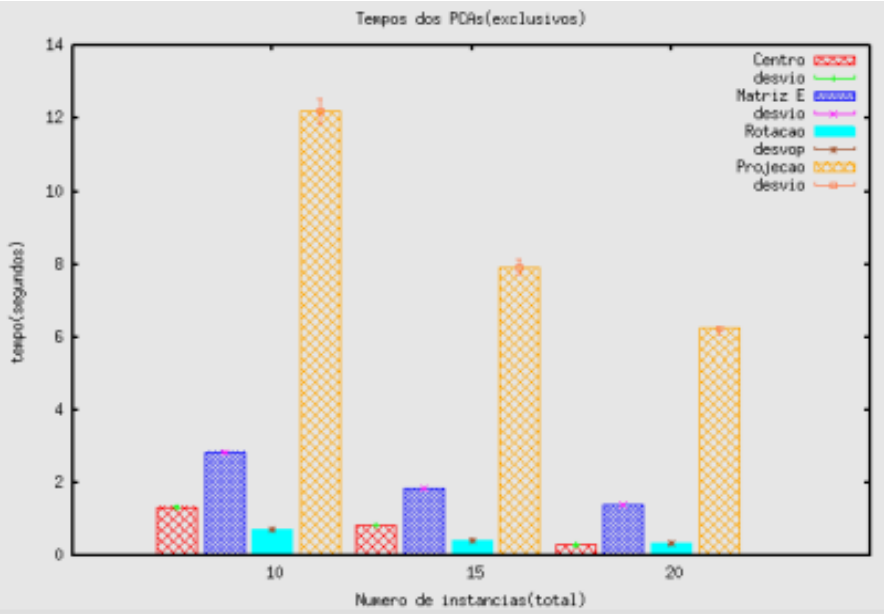

Figura 6.6: Tempo dos PCAs executados separados dos Contadores.

Por fim, na Figura 6.7 é apresentado o resultado global do experimento, onde se vê claramente que o sistema escala com o número de filtros utilizados. O tempo de execução caiu de aproximadamente 4350 com a utilização de 30 instâncias de filtros, para 2400 com 50 instâncias. Esse número indica um *speedup* de 1,8 em relação à execução com 30 filtros e uma eficiência paralela de $mu_{30,50} = 1,087$.

O desempenho superlinear se explica pela divisão do dado por mais instâncias de filtros, o que acarreta uma menor utilização de memória virtual ao sistema, especialmente na etapa dos Contadores , a mais cara computacionalmente. As grandes reduções de tempo ocorrem nesta etapa. O tempo de contagem de vizinhos de cada pixel vermelho, com 10 Contadores, cai de aproximadamente 3200 segundos para 1450 segundos com 20 contadores. O tempo de inserção nas árvores também cai de 400 segundos com 10 contadores para 150 segundos com 20 Contadores.

Durante esta etapa, lida-se com aproximadamente 1,4GB de dados. Dividindo esse valor por 10 máquinas, obtém-se aproximadamente 140MB por máquina, no caso de um balanceamento perfeito. As estruturas de dados utilizados são mostradas na Listagem 6.1. Cada ponto tem tamanho de dois *doubles* (16 bytes total) e será inserido em um nodo da árvore. Cada nodo possui ponteiros para o nodo pai, para os dois nodos filhos, a estrutura de conteúdo do nodo e inteiros para o tamanho do conteúdo,

para o nível na árvore, para o indicador de nodo visitado e para o número de vizinhos (32 bytes). Além disso, os Contadores mantém referências aos pixels vermelhos na contagem do histograma. As referências são armazenadas em listas duplamente encadeadas (3 ponteiros, incluindo a referência ao elemento, num total de 12 bytes).

Portanto, para cada 16 bytes recebidos, são adicionados mais 44 (ou 2,75 bytes a mais para cada byte recebido). Para os 140MB recebidos em um caso ideal (distribuição perfeita de pontos), o programa geraria 525MB de dados em estruturas de armazenamento de pixels pelos Contadores e os próprios pixels. Levando-se em consideração que existem vários nodos executando mais de uma instância de filtro, vê-se que a memória utilizada pelos contadores ultrapassa os valores de memória da máquina (512MB). A adição de novas instâncias de Contadores reduz a utilização de memória por instância para aproximadamente a metade (no caso de 20 Contadores), o que pode ser contido na memória primária dos nodos. Como consequência, ocorre o ganho de desempenho superlinear.

Listagem 6.1: Estruturas de dados utilizadas pelos Contadores

```
// estruturas da arvore 2-D
typedef struct {
  double x;
  double y;
} Coord;

typedef struct _NodeContent {
  void *userContent;
  int size;
  int numNeighbors;
  Coord coords;
  int visited;
} NodeContent;

typedef struct _NodeStruct {
  struct _NodeStruct *children[2], *parent;
  int level;
  NodeContent nodeContent;
} NodeStruct;

// estruturas da lista encadeada
typedef struct _DbLkListNodeContent {
  void *content;
} DbLkListNodeContent;

typedef struct _DbLkListNode {
  struct _DbLkListNode *previous, *next;
  DbLkListNodeContent nodeContent;
} DbLkListNode;
```

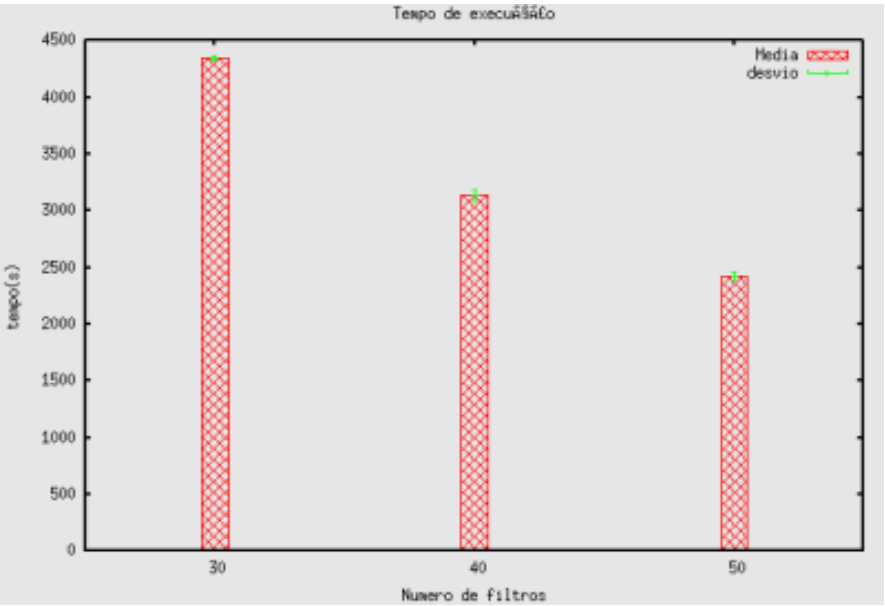

Figura 6.7: Resultado global dos experimentos.

6.3 Resumo

Neste capítulo foram mostrados os resultados de duas aplicações distintas. Ambas lidam com grandes volumes de dados, mas têm processamentos diferentes. Os resultados obtidos mostraram que, em condições ideais (um filtro por máquina), o sistema escalou de maneira linear ou superior. Os tempos gastos pelas funções de fluxo rotulado também foram baixos em relação aos tempos de processamento das aplicações. Os resultados mostram que o Formigueiro permitiu o desenvolvimento de aplicações eficientes, sem interferir significativamente no desempenho da aplicação.

Capítulo 7

Conclusão

A programação paralela distribuída é o caminho para a execução de aplicações que lidam com grandes volumes de dados e/ou computacionalmente caras, para as quais ainda não se tem uma solução serial viável. Este nicho da programação oferece oportunidades únicas para otimização de programas através de assincronismo entre tarefas distintas, concorrência entre tarefas iguais e distribuição da carga de trabalho.

Por outro lado, essas oportunidades de otimização acarretam um preço. A complexidade de se fazer um programa paralelo é maior e a depuração é ainda mais complexa. Condições de corrida (*race conditions*), travamentos completos (*deadlocks*), inanição de processos (*starvation*) são alguns dos problemas mais conhecidos. Existem outras dificuldades como: falhas nos meios de comunicação (frequentes em execuções distribuídas), ambientes dinâmicos, sistemas operacionais distintos (com ordem de bytes diferentes), tudo isso formando uma rede bastante caótica e heterogênea de recursos para a execução de programas paralelos distribuídos.

O Formigueiro é um sistema que facilita a programação paralela para uma grande variedade de aplicações. Além de abranger as aplicações que podem ser executadas no DataCutter (aplicações estilo *bag of tasks*), o Formigueiro permite ainda a criação de programas com manutenção de estados distribuídos. Para o programador, ele oferece também uma abstração que facilita a criação do programa. O modelo do Formigueiro divide a criação de programas paralelos nas seguintes etapas:

- Determinação de etapas: um programa é modelado como uma sequência de etapas distintas, como em uma linha de processamento (uma etapa depende apenas do que ela recebe, ou já recebeu). Cada etapa é conduzida de maneira independente da outra, criando um paralelismo de tarefas. As etapas são conhecidas como filtros.

- Alívio de gargalos: etapas mais caras computacionalmente podem ser distribuídas e executadas paralelamente. Etapas replicadas são conhecidas como instâncias de um mesmo filtro. A replicação de filtros permite o paralelismo de dados.

- Comunicação: as etapas devem se comunicar. A comunicação entre elas é chamada de fluxo, pois existe uma transmissão constante de dados entre etapas, mantendo a linha de processamento sempre ocupada.

- Descrição do trabalho: todas as etapas devem ter o conhecimento sobre o que estarão trabalhando. Esse conhecimento é representado pelo trabalho no Formigueiro, uma estrutura definida pelo programador que descreve o que os filtros executarão.

- Utilização de recursos: por último, o usuário deve escolher quais recursos serão utilizados para sua execução.

Sem um ambiente auxiliar, o programador teria bastante dificuldades com a inicialização distribuída dos processos, a comunicação, a modelagem do programa, o tratamento de condições de corrida e até com a medição de resultados. Alguns ambientes oferecem serviços para facilitar essas tarefas, desde os protocolos de rede (TCP/IP) até bibliotecas de comunicação e disparo de processos (PVM). O Formigueiro preenche a lacuna no nível mais alto, fornecendo também as facilidades oferecidas nos níveis mais baixos de comunicação.

O sistema utiliza uma linguagem amplamente difundida (XML) para a declaração de recursos a serem utilizados (máquinas), para a associação de máquinas a tarefas distintas (filtros), para a comunicação (fluxos) e para a replicação de etapas caras (instâncias). Com isso, a adaptação de programadores novos é mínima. Os filtros são escritos em linguagem C, também amplamente conhecida por programadores, especialmente os que lidam com aplicações de alto-desempenho. O Formigueiro ainda oferece facilidades para depuração paralela, utilização de recursos por um filtro e uma interface para instrumentação de código. Essas facilidades são mostradas no Capítulo 4.

O modelo de fluxo de dados do Datacutter é bastante eficiente, mas possui a limitação de não permitir estados distribuídos. Com o Formigueiro, o modelo é estendido, atendendo a uma variedade maior de programas. Neste trabalho, o Formigueiro foi utilizado em uma aplicação real de processamento de um grande volume de dados (uma imagem de 40GB). Os resultados apresentados no Capítulo 6 mostram que o sistema escala satisfatoriamente enquanto existem recursos disponíveis. Em algumas situações, é possível utilizar uma mesma máquina para a execução de tarefas distintas e ainda obter bons resultados.

A implementação do Formigueiro foi voltada desde o ínicio para possíveis ampliações de suas funções. Várias extensões são possíveis para o sistema, dentre as quais destacam-se um ambiente de execução dinâmico (onde as máquinas participam de apenas parte de uma execução) e tolerante a falhas. Integração com outros sistemas de gerenciamento de recursos também poderão ser implementadas no código. Muitas dessas idéias serão apresentadas na Seção 7.1.

O desenvolvimento de programações paralelas é a chave para aplicações que resolvem questões complexas, como bases de dados extremamente grandes ou aplicações caras computacionalmente. Facilitar o desenvolvimento deste tipo de programa significa acelerar o desenvolvimento dessas aplicações. O objetivo pode ser simples, como mostrar o mapa de uma região qualquer do mundo em tempo real, ou pode ser complexo, como determinar milhares de combinações de fatores genéticos que levam a uma doença como o câncer.

7.1 Trabalhos Futuros

Como o Formigueiro é uma extensão do DataCutter, a adição de várias características ao arcabouço pode ser interessante. Nesta seção serão discutidas algumas possíveis extensões para o Formigueiro.

7.1.1 Multiplataforma

Neste texto, foram consideradas as características mais importantes do DataCutter, como o modelo de fluxo de dados, a comunicação, os fluxos e o trabalho. No entanto, o DataCutter é um sistema multiplataforma, enquanto o Formigueiro se limita a apenas uma plataforma operacional, o Linux. Apesar da crescente participação do Linux no mercado de computadores, especialmente na área acadêmica, há grande disponibilidade de sistemas, como o Windows, em empresas e instituições não governamentais. Uma implementação do Formigueiro para tais sistemas pode abrir uma nova possibilidade de utilização de recursos.

7.1.2 Fluxo acumulado

Atualmente, para cada chamada de envio de mensagem do programador, o Formigueiro gera uma chamada de envio do PVM. Nem sempre esta opção é satisfatória, podendo levar a um baixo desempenho para o programa. Não existe um *buffer* de armazenamento das mensagens enviadas em um período curto de tempo. Tal artifício poderia diminuir o número de mensagens do programa e também a sobrecarga de envio.

Um exemplo de como esse sistema pode ser útil está na própria aplicação da placenta, apresentada no Capítulo 5. Quando o filtro RedFinder escreve para um Contador, o correto seria enviar o pixel vermelho assim que este fosse encontrado. No entanto, tal alternativa geraria um grande número de mensagens, além de uma alta sobrecarga de comunicação. Como cada pixel ocupa o equivalente às suas coordenadas, ou seja, 2 *doubles* (16 bytes) e um datagrama IP possui 24 bytes de cabeçalho, a quantidade de dados enviados estaria mais do que dobrando.

Para a aplicação da placenta, o acúmulo de dados foi implementado no nível da aplicação, através de *buffers*. O diagrama mostrado na Figura 7.1 apresenta uma proposta para a implementação de fluxo acumulado.

A e B representam dois filtros quaisquer. A instância k de A escreve para B(um ou mais destinos). Após aplicação da política de envio, que pode ser um fluxo rotulado múltiplo por exemplo, chega-se aos destinos i e j. Ao invés de enviar os dados diretamente para estas instâncias, a mensagem é armazenada em um *buffer* de envio, descarregado em quatro situações:

1. Passagem de uma quantidade de tempo determinada pelo programador(temporização).
2. Acúmulo de mensagens cujo tamanho total ultrapasse a quantidade fornecida pelo programador.
3. Chamada da função dsFlushOutputPort.
4. Fechamento da porta, por dsClose.

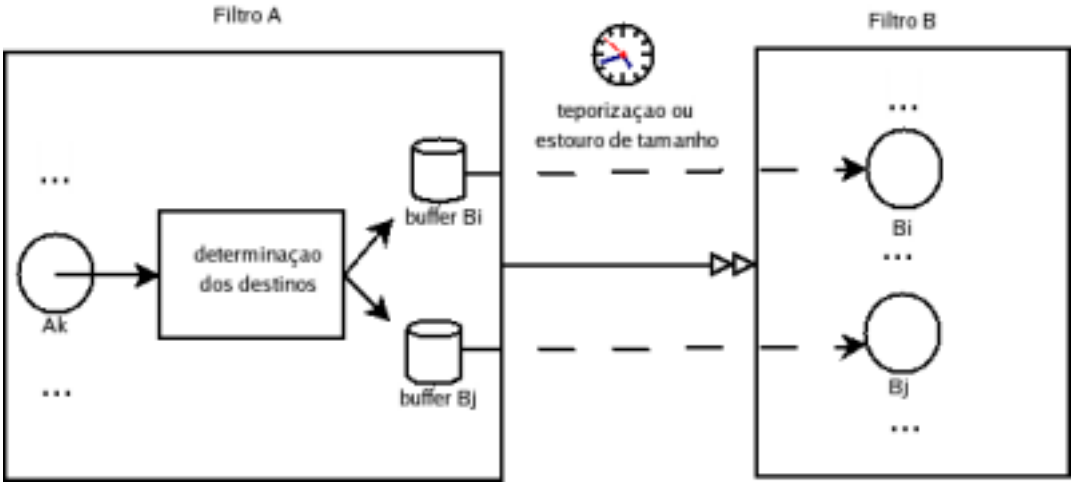

Figura 7.1: Fluxo acumulado no Formigueiro.

A passagem de parâmetros do fluxo seria feita através do XML, onde a declaração do fluxo(elemento stream) aceitaria os atributos time e size. A Listagem 7.1 mostra como ficaria a declaração do fluxo da Figura 7.1, usando-se parâmetro de temporização de 2 segundos e um acúmulo máximo de 512KB.

Listagem 7.1: Declaração de fluxo acumulado no XML.

```
<!—...—>
<stream time="2s" buffersize="512KB">
  <from filter="A" port="portA" policy="mls" policylib="..."/>
  <to filter="V" port="portB"/>
</stream>
<!—...—>
```

Neste caso, a cada dois segundos, o fluxo que liga A e B através das portas $portA$ e $portB$ seria descarregado. Caso uma mensagem, ou o total de dados acumulados, ultrapasse o limite de 512KB, os dados são enviados. Por último, uma chamada de dsFlushOutputPort(portA) ou dsClose(portA) realiza a descarga imediatamente. A diferença está em que a primeira permite a reutilização da porta, enquanto a última fecha a porta para o trabalho corrente.

7.1.3 Integração

A implementação corrente do Formigueiro já é integrada ao sistema PVM. Tal integração permitiu abstrair-se a parte de comunicação confiável, o disparo de processos, o tratamento de falhas, além de uma interface de programação mais amigável que a interface de *sockets*. Outros sistemas podem também apresentar ganhos significativos ao Formigueiro.

Por exemplo, o Condor poderia gerenciar um grupo de máquinas dinâmico e gerar a declaração de *hosts* automaticamente para o Formigueiro, de acordo com o sistema disponível no momento da execução. Para tal, além da integração no código do Formigueiro, seria necessário uma alteração no arquivo de configuração XML, permitindo uma declaração dinâmica. Além disso, os filtros teriam que declarar seus recursos segundo a sintaxe do Condor. Um exemplo de trecho do XML proposto é visto na Listagem 7.2.

Listagem 7.2: Declaração dinâmica através do Condor

```
<!--...-->
<hostdec type="condor" url="condorserver:port" />
<!--...-->
```

O Formigueiro, ao ler o elemento hostdec com atributo type igual a condor, geraria a declaração de *hosts* segundo a resposta do servidor do Condor, localizado no endereço do atributo url. A resposta do Condor conteria as máquinas disponíveis e também seus recursos. A sintaxe da declaração dos filtros é a mesma dos *classAd*s de trabalhos(*jobs*) do Condor[26]. Por exemplo, um filtro poderia declarar:

Listagem 7.3: Requisitos de filtros através do Condor

```
<placement>
  <!--...-->
  <filter name="A" libname="a.so" instances="10">
    <requirements>
    [
      ...
    MyType = "Job"
    Requirements = (DatabaseA) && (Memory > 512*1024)
      ...
    ]
    </requirement>
  </filter>
  <!--...-->
```

Neste caso, o filtro está declarando que necessita do recurso DatabaseA e mais de 512MB de memória. Note que a semântica de DatabaseA é livre, já que o Condor não define um esquema para os *ClassAds*. Portanto, o programador pode definir livremente seus requisitos, desde que as máquinas disponíveis também sejam configuradas de acordo com eles. O trabalho de casamento de filtros com máquina passaria a ser do *matchmaker* do Condor, e não mais do Formigueiro.

7.1.4 Múltiplos trabalhos

Uma deficiência apresentada pelo Formigueiro é a limitação de apenas um trabalho na linha em cada instante de tempo. O próximo trabalho é iniciado apenas após o término do corrente. Esta barreira é desnecessária, impedindo um paralelismo maior de tarefas. A função da barreira é evitar que mensagens de trabalhos diferentes se misturem. Por exemplo, no caso da aplicação da placenta, cada imagem a ser tratada é considerada um trabalho. Caso um RedFinder termine sua parte antes das outras instâncias, ele poderia iniciar imediatamente a leitura da próxima imagem. No entanto, ao enviar dados para os Contadores, estes seriam misturados com os dados da imagem anterior, gerando um resultado errado.

Uma alternativa simples à barreira é esperar que todas as instâncias de um filtro finalizem o trabalho corrente antes que qualquer uma delas inicie um novo trabalho. Para isso, uma mensagem de fim de trabalho (EOW) sempre separaria as mensagens de trabalhos diferentes e quando todas as instâncias de um filtro terminassem o trabalho corrente, o EOW seria enviado a todas as portas de

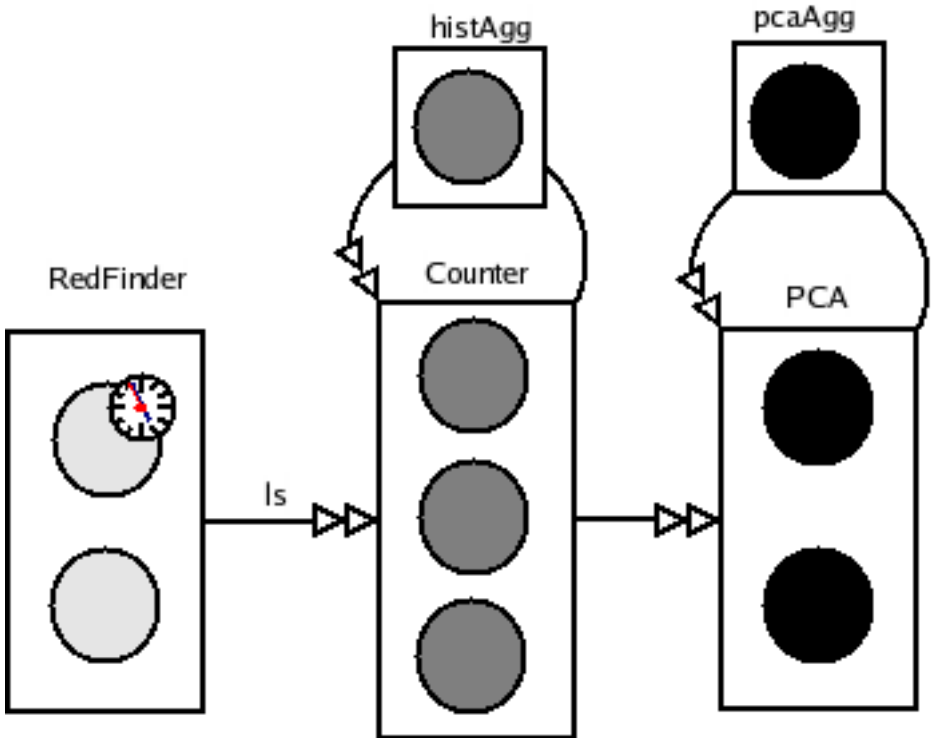

Figura 7.2: Múltiplos trabalhos na linha, sincronizados por filtro.

comunicação de saída. Mesmo assim, seria possível haver instâncias ociosas no processo, aguardando que as outras instâncias do mesmo filtro terminem. Esta implementação está representada na Figura 7.2, onde as sombras mais escuras representam os trabalhos mais antigos na linha. A instância superior do RedFinder está ociosa e, já tendo terminado seu trabalho, aguarda a outra instância terminar.

Outra opção é a de se marcar todas as mensagens com um dado adicional, identificando a qual trabalho a mensagem pertence. Caso a instância receptora esteja processando outro trabalho, a mensagem é guardada em um espaço de armazenamento, ou a instância responsável pelo envio é bloqueada até o receptor estar processando o mesmo trabalho. A vantagem desta implementação é permitir que trabalhos distintos sejam executados em filtros do mesmo tipo, gerando um paralelismo ainda maior, e uma execução assíncrona. A Figura 7.3 ilustra a linha com múltiplos trabalhos em instâncias de um mesmo filtro e, novamente, as sombras mais escuras representam trabalhos mais antigos.

Em qualquer um dos casos, o Formigueiro deve manter o estado dos trabalhos, para possibilitar o reinício, em caso de falhas, a partir do trabalho não finalizado mais antigo dentro da linha. Elaborando-se um pouco mais, é possível criar uma estrutura de dependências entre os trabalhos, onde o programador pode solicitar que um trabalho só seja iniciado após o término de outro(s).

7.1.5 Configuração Dinâmica com Manutenção de Estados

Atualmente, a configuração dinâmica pode ser executada entre trabalhos distintos sem maiores problemas, já que o resultado da aplicação depende apenas do trabalho (ver seção 4.3). Também é possível realizar a mudança de configuração de filtros em programas sem manutenção de estados.

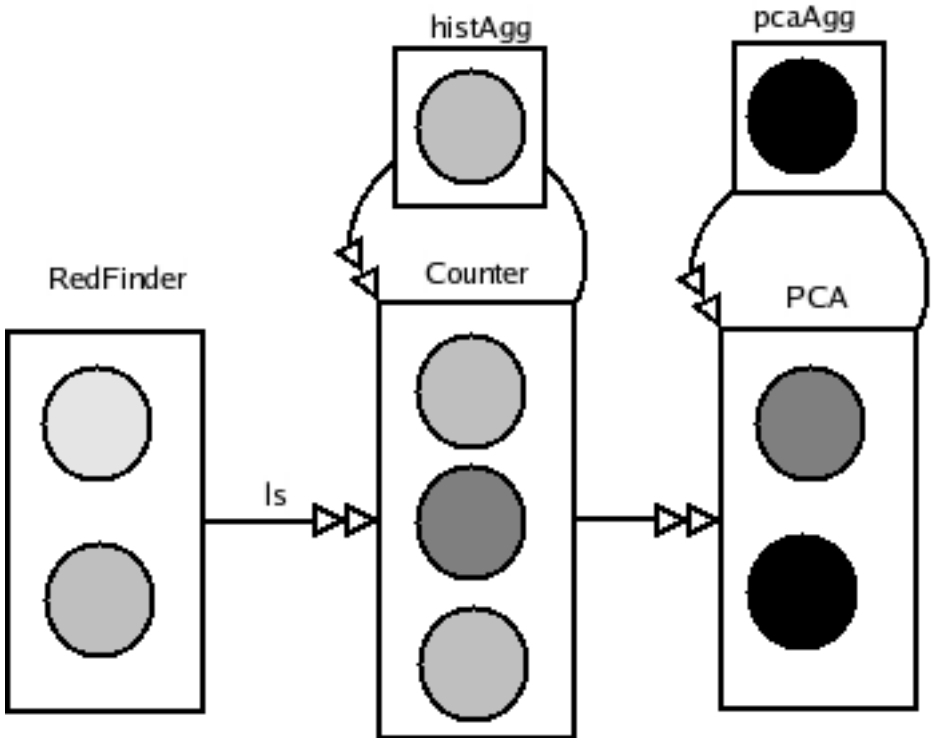

Figura 7.3: Múltiplos trabalhos na linha, execução assíncrona.

No entanto, programas que possuem estados distribuídos apresentam dificuldades de mudança de configuração, em especial, os programas com fluxo rotulado.

Voltando ao exemplo de contagem de cidades, apresentado no Capítulo 3, onde o objetivo é descobrir quantas cidades de um estado estão situadas a uma distância máxima da capital, o estado mantido é a contagem do filtro classificador. Se houver uma mudança de configuração durante a execução, possivelmente instâncias desse filtro passarão a ser responsáveis por outros estados. Neste caso, ocorrem dois problemas:

- descobrir quem é o novo responsável por esta parte do estado mantido;
- repassar o estado mantido até o momento para o novo responsável.

Para exemplificar, suponha que o filtro C_i seja o responsável pelo estado de Tocantins e São Paulo até o momento da mudança de configuração. No momento da mudança, uma nova máquina é adicionada e um filtro classificador C_j é nela alocado. Quem determina qual, ou quais estados serão de responsabilidade desta nova instância (e as outras também) é o fluxo rotulado, através das funções getLabel e hash. A configuração poderá mudar completamente dependendo dessas funções. Para este exemplo, suponha que apenas o estado de São Paulo tenha passado para a nova instância e as outras permaneceram inalteradas. Deve-se descobrir este fato e repassar a contagem de São Paulo à nova instância.

Para que seja possível realizar esta mudança, o Formigueiro deve manter uma lista de todos os rótulos já utilizados pelo fluxo rotulado até o momento da mudança. Se o estado relativo ao rótulo

for mantido em uma variável do Formigueiro, este valor poderá ser passado à nova instância em caso de mundança. Adicionado-se também uma função de *callback* do usuário, para executar quaisquer outras operações necessárias pelo programa (como realocação e liberação de memória), é possível mudar a configuração das instâncias.

Considere a seguinte estrutura de dados, que armazena dados relativos a um estado, e as funções do Formigueiro para manipulação da estrutura:

Listagem 7.4: Estruturas de armazenamento para dados de estado

```
typedef struct {
  char *label; //o rotulo, que define o estado
  void *data;  //os dados deste estado
  int dataSize; //tamanho da estrutura data
} StateData;

List <StateData> stateDataList; //lista de dados de estado

void dsSetStateData(char *label, void *data, int size); //cria o estado com os
    dados
void dsGetStateData(char *label, void **data, int *size); //recupera os valores
    armazenados pelo estado
```

Todo dado de estado relativo a um rótulo deverá ser colocado nesta estrutura usando a função dsSetStateData. Observe que o dado é genérico, sendo definido pelo usuário. O tamanho é utilizado para realizar a cópia dos dados, no caso da mudança.

Quando houver a mudança de configuração, o Formigueiro chamará as funções do fluxo rotulado *getLabel* e *hash* em cada uma das instâncias com estado, usando todos os rótulos utilizados até o momento, para descobrir quais estados pertencem agora a cada instância.

Uma vez descobertos, a lista stateDataList de cada instância será percorrida e os dados de estado ali encontrados serão enviados à nova instância responsável. Ao término, a operação de *callback* do usuário (se houver tal função) será chamada para fazer qualquer operação necessária (como alocação de variáveis dinâmicas, por exemplo).

Com isso, os estados serão transferidos à nova instância e a execução poderá presseguir normalmente. Este procedimento vale apenas para filtros que usem fluxo rotulado para distribuir o estado. Para a difusão seletiva o processo não funciona.

7.1.6 Tolerância a falhas

Tolerância a falhas é uma área que tem se tornado cada vez mais importante em se tratando de programação paralela. Ao aumentar-se o número de recursos envolvidos em uma execução, aumenta-se também o risco de falhas, pois lida-se com diversas variáveis (equipamentos, meios de comunicação, sistemas operacionais etc) e, normalmente, o emprego de programas paralelos se aplica a programas cuja execução leva um tempo longo, aumentando ainda mais a chance de falhas ocorrerem.

O Formigueiro apresenta um mecanismo de reinício a partir do último trabalho falho (ver seção 4.6). Esse mecanismo, apesar de útil, é limitado, pois alguns tipos de programas funcionam apenas

com um trabalho, mas de tamanho grande. Uma falha nesses programas implica no reinício completo da execução.

A implementação em PVM do Formigueiro permite a notificação ao programador, de falhas em tempo de execução e a recuperação da execução. A utilização de pontos de verificação pode ser utilizada para a retomada da execução a partir do último ponto ultrapassado. Esta alternativa implica na necessidade de uma implementação com consenso global distribuído para determinar o estado de um *checkpoint*. Juntamente com a utilização de funções de recuperação do usuário(*callbacks* de falhas), este modelo pode se tornar uma alternativa poderosa para recuperação da execução em caso de falhas.

7.1.7 Geração Automática de Programas Paralelos

A geração automática é responsável por transformar um programa serial em um programa paralelo no modelo de fluxo de dados. O programa serial é feito em uma linguagem instrumentada de alto nível. O objetivo é abstrair do programador a necessidade de lidar diretamente com as dificuldades de um ambiente paralelo, passando essa responsabilidade ao sistema intermediário de geração.

Alguns tipos de aplicações vêm sendo estudadas recentemente em mineração de dados. Estas aplicações têm como características marcantes o fato de serem:

- intensivas em dados;
- irregulares(tempo de execução variável e não previsível);
- iterativas, gerando suas tarefas em tempo de execução.

Tais aplicações são conhecidas como I^3[17] e podem ser descritas por um grafo direcionado de tarefas acíclico $G = (V, A)$, onde cada vértice $v_i \in V$ representa uma tarefa a ser executada pelo algoritmo. Uma aresta $a_i \in A$ que liga v_i a v_{i+1} indica que existe uma dependência de dados da tarefa representada por v_i para v_{i+1}.

Transformar estes tipos de aplicações em programas de fluxo de dados consiste nas seguintes etapas:

1. Extração do grafo de tarefas da aplicação.
2. Mapeamento do grafo de tarefas em filtros.
3. Geração do código dos filtros.
4. Escalonamento dos filtros no ambiente paralelo.

O trabalho proposto em [17] apresenta uma maneira de se transformar o grafo de tarefas em filtros, além de uma política de escalonamento de filtros baseada em informação de execuções anteriores controladas. A extração do grafo de tarefas e a geração automática de código de filtros dependem de diretivas adicionadas ao código instrumentado. Um trabalho futuro, importante, seria definir estas meta-instruções, completando as duas etapas remanescentes.

Referências Bibliográficas

[exp] Expat: Xml parser. `http://expat.sourceforge.net/`.

[pvm] Pvm - parallel virtual machine. `http://www.csm.ornl.gov/pvm/pvm_home.html`.

[1] Acharya, A.; Uysal, M. e Saltz, J. H. (1998). Active disks: Programming model, algorithms and evaluation. In *Architectural Support for Programming Languages and Operating Systems*, pp. 81–91.

[2] Agrawal, R. e Srikant, R. (1994). Fast algorithms for mining association rules. In Bocca, J. B.; Jarke, M. e Zaniolo, C., editores, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pp. 487–499. Morgan Kaufmann.

[3] Beynon, M.; Ferreira, R.; Kurc, T. M.; Sussman, A. e Saltz, J. H. (2000). Datacutter: Middleware for filtering very large scientific datasets on archival storage systems. In *IEEE Symposium on Mass Storage Systems*, pp. 119–134.

[4] Chang, C.; Sussman, A. e Saltz, J. (1998). Infrastructure for building parallel database systems for multi-dimensional data. Technical Report CS-TR-3894.

[5] Dean, J. e Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *OSDI*, pp. 137–150.

[6] do Nascimento, L. T. (2004). Escalonamento de aplicações de fluxo de dados. Master's thesis, Universidade Federal de Minas Gerais.

[7] Ferreira, R.; Moon, B.; Humphries, J.; Sussman, A.; Saltz, J.; Miller, R. e Demarzo, A. (1997). The virtual microscope. Technical report, College Park, MD, USA.

[8] Ferreira, R. A.; Meira, W.; Guedes, D.; Drummond, L.; Coutinho, B.; Teodoro, G.; Tavares, T.; Araújo, R. e Ferreira, G. (2005). Anthill: A scalable run-time environment for data mining applications. In *SBAC-PAD 2005*.

[9] Ferreira, R. A. C. (2001). *Compiler Techniques for Data Parallel Applications Using Very Large Muti-Dimensional Datasets*. PhD thesis, University of Maryland.

[10] Foster, I. e Kesselman, C. (1997). Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128.

[11] Foster, I. e Kesselman, C., editores (1999). *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[12] Frey, J.; Tannenbaum, T.; Foster, I.; Livny, M. e Tuecke, S. (2001). Condor-G: A computation management agent for multi-institutional grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC)*, pp. 7–9, San Francisco, California.

[13] Geist, G. A.; Kohla, J. A. e Papadopoulos, P. M. (1996). PVM and MPI: A Comparison of Features. *Calculateurs Paralleles*, 8(2):137–150.

[14] Ghemawat, S.; Gobioff, H. e Leung, S. (2003). The google file system.

[15] GNU Project (2005). *Using the GNU Compiler Collection*. http://gcc.gnu.org/onlinedocs/gcc-4.0.1/gcc/Function-Attributes.html#Function-Attributes.

[16] Gonzales, R. e Woods, R. (1992). *Processamento Digital de Imagens*, chapter 3, pp. 106–110. Addison-Wesley.

[17] Góes, L. F. W.; Giovani, I.; Ferreira, R.; Guedes, D. e Jr, W. M. (2005). Mapeamento de programas i3 para aplicações anthill paralelas de fluxos de dados baseadas em filtros. Aceito para publicação no WSCAD 2005.

[18] Hastings, S.; Langella, S.; Oster, S. e Saltz, J. (2004). Distributed data management and integration framework: The mobius project. In *Global Grid Forum 11 (GGF11) Semantic Grid Applications Workshop*, pp. 20–38.

[19] Hastings, S.; Ribeiro, M.; Langella, S.; Oster, S.; Catalyurek, U.; Pan, T.; Huang, K.; Ferreira, R.; Saltz, J. e Kurc, T. (2005). Xml database support for distributed execution of data-intensive scientific workflows. ACM SIGMOD Record, Special Section on Scientific Workflows, accepted for publication.

[20] MPI, F. (1994). Mpi: A message-passing interface. Technical report.

[21] Ribeiro, M.; Kurc, T.; Pan, T.; Huang, K.; Catalyurek, U.; Zhang, X.; Langella, S.; Hastings, S.; Oster, S.; Ferreira, R. e Saltz, J. (2005). Tools for efficient subsetting and pipelined processing of large scale distributed biomedical image data. In Grandinetti, L., editor, *Grid Computing: New Frontiers of High Performance Computing*. Elsevier.

[22] Samet, H. (1984). The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260.

[23] Spencer, M.; Ferreira, R.; Beynon, M.; Kurc, T.; Catalyurek, U.; Sussman, A. e Saltz, J. (2002). Executing multiple pipelined data analysis operations in the grid. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pp. 1–18, Los Alamitos, CA, USA. IEEE Computer Society Press.

[24] Sunderam, V. S. (1990). PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340.

[25] Tannenbaum, T.; Wright, D.; Miller, K. e Livny, M. (2001). Condor – a distributed job scheduler. In Sterling, T., editor, *Beowulf Cluster Computing with Linux*. MIT Press.

[26] Thain, D.; Tannenbaum, T. e Livny, M. (2004). Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*.

[27] Veloso, A.; Meira, W.; Ferreira, R.; Guedes, D. e Parthasarathy, S. (2004). Asynchronous and anticipatory filter-stream based parallel algorithm for frequent itemset mining.

[28] Weng, L.; Agrawal, G.; Catalyurek, U. V.; Kurc, T. M.; (K2), S. N. e Saltz, J. H. (2004). An approach for automatic data virtualization. In *IEEE Symposium on High-Performance Distributed Computing*, pp. 24–33.

[29] Wheeler, D. A. (2003). Program library howto.

[30] Wolski, R.; Spring, N. T. e Hayes, J. (1999). The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768.