JULIANO DE CASTRO SANTOS

PARTICIONAMENTO SEMI-AUTOMÁTICO DE PROGRAMAS CÍCLICOS SEQÜENCIAIS PARA EXECUÇÃO EM ANTHILL

Belo Horizonte, Minas Gerais

15 de julho de 2007

JULIANO DE CASTRO SANTOS

PARTICIONAMENTO SEMI-AUTOMÁTICO DE PROGRAMAS CÍCLICOS SEQÜENCIAIS PARA EXECUÇÃO EM ANTHILL

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

Belo Horizonte, Minas Gerais

15 de julho de 2007

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

Particionamento Semi-Automático de Programas Cíclicos Seqüenciais para execução em Anthill

JULIANO DE CASTRO SANTOS

Dissertação defendida e aprovada pela banca examinadora constituída por:

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

Prof. Dorgival Olavo Guedes Neto
Universidade Federal de Minas Gerais

Prof. Wagner Meira Junior
Universidade Federal de Minas Gerais

Prof. Walfredo Cirne
Universidade Federal de Campina Grande

Belo Horizonte, Minas Gerais, 15 de julho de 2007

Resumo

Extrair informação de grandes bases de dados é um desafio das novas demandas da Ciência da Computação. Além disso, diversas técnicas de Mineração de Dados são propostas continuamente na literatura. Tais algoritmos são computacionalmente intensivos, além do fato de processarem entradas de dados muito grandes. Assim, há uma nova tendência em direção ao alto desempenho de processamento e montagem de *clusters* de computadores, a um custo mais razoável que os supercomputadores do passado. Essa tendência nos leva a um cenário em que o alto desempenho é alcançado por eficientes implementações de aplicações paralelas e distribuídas.

O *Anthill* é uma solução para processamento distribuído em *clusters* de computadores onde um alto desempenho é obtido em execuções eficientes e escaláveis de diversos algoritmos de Mineração de Dados. Ele fornece um *framework* para o desenvolvimento e a execução de aplicações em ambiente distribuído, onde as aplicações devem ser decompostas em filtros que se comunicam através de *streams* de dados, como em um *pipeline*. O processo de desenvolvimento das aplicações entretanto demanda do programador decompor as aplicações, que não é o caminho natural de desenvolvimento dos programas.

Este trabalho apresenta uma ferramenta de geração semi-automática de filtros a partir de uma implementação seqüencial do algoritmo. Esse processo é divido em duas etapas: a primeira é o particionamento em filtros do código seqüencial, seguido pela geração do código para cada um dos filtros identificados. Este trabalho tem como foco a segunda etapa, a geração do código. Para a primeira etapa, nós realizamos de forma semi-automática alguns passos a serem automatizados em trabalhos futuros. Esse processo determina as dependências de dados no código, e identifica os pontos de corte do mesmo. A partir deste passo, é gerado uma versão anotada do código seqüencial, contendo as informações para sua divisão em filtros.

A geração dos filtros é feita a partir do código anotado. Basicamente, as anotações contêm um grafo direcionado onde os vértices representam os filtros e as arestas representam os dados que são trafegados entre os filtros. Nós implementamos um gerador de código fonte onde a entrada é um algoritmos seqüencial escrito em **C** e a

saída são os filtros, também em **C**, com as extensões do *Anthill*. A maioria das adaptações necessárias para *Anthill* são geradas automaticamente por essa ferramenta.

O gerador de código automático foi validado usando três aplicações de Mineração de Dados que já haviam sido implementadas no *Anthill*. Foram gerados de forma automática o código dos filtros *Anthill* a partir das versões seqüenciais. Nós avaliamos o desempenho do código gerado e observamos que o resultado é similar ao código gerado manualmente na maioria dos casos. Este é um bom resultado, dado que o custo de implementação do código seqüencial é bem menor do que a implementação paralela dos filtros para o *Anthill*. Também foram observadas algumas otimizações presentes na implementação manual que podem ser realizadas automaticamente pela ferramenta para obtenção de um resultado mais otimizado. Como trabalhos futuros devemos prosseguir com a automatização de mais otimizações na geração do código.

Abstract

Extracting information from large datasets is one of the challenging new demands in Computer Science. To accomplish that, several Data Mining techniques are being proposed continually in the literature. Such algorithms are computationally intensive, in addition to the fact of having to process very large input data. Besides, the new trend towards high performance computing is gearing towards clusters of computers, a cost effective alternative to the supercomputers of the past. This trend leads to a scenario in which achieving high performance must be accomplished by efficient parallel and distributed implementations of the applications.

Anthill is a solution for distributed processing on clusters of workstations for which high performance have been achieved by efficient and scalable implementation of several Data Mining algorithms. In essence, it provides a framework for the development and execution of applications in distributed environments, where the applications must be decomposed into a set of filters that exchange information though streams in a pipeline fashion. The process of developing the applications, however, damands the programmers to perform such application decomposition which may not be the natural way in which they program.

This work presents a tool of semiautomatic generation of such fiters from a basic sequential implementation of the algorithms. This process is divided in two stages: the first being the partition into filters of the sequential code, followed by the code generation for each of the identified filters. This work focuses mostly on the latter, the code generation itself. For the first stage, we rely on some semi-automatic steps that could be implemented to be fully automatic in future work. These steps are based on determining data dependencies within the code, and finding good partition places. Using such steps we generate an annotated version of the sequential code, that contains the partitioning information.

The actual code generation is accomplished from the annotated code. Basically the annotations encompass a directed graph with vertices representing the filters and edges annotated with the data that should be communicated. We implemented a source-to-source compiler, where the initial sequential code is standard C, and the

output is also C, with Anthill extensions. Most of the necessary adaptations for Anthill are generated automatically by our compiler.

The compiler was validated using three different Data Mining applications that had previously been developed for Anthill. This time, we generated the Anthill code from sequential versions of the same algorithms. We evaluate the performance of the generated code and we observe that it is very similar to the hand-made implementations in most cases. This is a good result when noted that the effort to design the sequential code is much less than a fully parallel Anthill implementation. We also notice that there are some ad-hoc optimizations on the hand-made codes that could also be accomplished by a compiler in a further optimizing step. We plan to pursue such automatic optimizations as future work.

Agradecimentos

Várias foram as pessoas que contribuíram, tanto para este trabalho, como para a minha vida como um todo. Familiares, amigos, colegas de trabalho, professores e orientadores, todos direta ou indiretamente me ensinaram um pouco. Não vou colocar aqui uma lista de nomes, vou apenas dizer a todas essas pessoas um muito obrigado.

Dedico esta conquitsta a uma pessoa em especial, a minha Tia "Aia" que me acolheu por todos essas anos, desde antes do início da graduação até hoje, na conclusão do mestrado.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A principal revolução do Século XX foi a disseminação da informação. Devido a vários fatores, dentre eles a popularização da Internet, as pessoas e as corporações obtiveram acesso a uma infinidade de dados. Junto a essa avalanche de dados surgiu um novo desafio dos dias atuais: extrair conhecimento de tanta informação. A extração de conhecimento das grandes bases de dados existentes é um grande problema para a maioria das organizações. A cada dia mais dados são gerados e são necessários mais recursos computacionais para extrair informação dos mesmos. Soluções escaláveis e de baixo custo são imprescindíveis nesse cenário.

Nos últimos anos, o custo de armazenamento de dados foi reduzido consideravelmente. Com o advento de novas tecnologias e mídias, uma estação de trabalho comum possui uma capacidade de armazenamento de dados considerável. Hoje, um computador pessoal normalmente possui um *Hard Disk* com capacidade de armazenamento de dezenas de *gigabytes*. Um servidor com a finalidade de armazenamento de dados pode ter uma capacidade muito superior, com centenas ou milhares de *gigabytes*.

Uma conseqüência natural da evolução dos recursos computacionais é a geração de grandes massas de dados. Com o baixo custo de armazenamento e a visão de que a informação é a principal geradora de capital nos dias atuais, muitos tipos de dados passaram a ser armazenados. Registros de transações comerciais e financeiras, monitoração do meio ambiente, do corpo humano e dos próprios sistemas computacionais produzem uma enorme quantidade de dados. Alguns exemplos desses geradores de dados são: o sistema de compras pela Internet do governo federal [com], que gera registros de todas as transações; a constante coleta de informações do espaço pela NASA [nas]; as máquinas de busca da Internet, como o Google [goo] e o Yahoo [yah], que recebem milhões de pesquisas diárias. Todos esses dados na forma bruta podem ser transformados em conhecimento e utilizados para os mais diversos fins.

Uma ferramenta muito poderosa para o tratamento desses dados é a Mineração de Dados [HK01]. Através dela é possível extrair informação de grande quantidade de dados de forma automática e eficiente. Um algoritmo de extração de regras de associação, como o Apriori [VMF+04], pode ser usado para identificar padrões de comportamento. Estes padrões podem indicar, por exemplo, as preferências de consumidores de um site de comércio eletrônico. A Amazon [ama], uma das maiores livrarias virtuais, utiliza dessas técnicas para a personalização do site para seus clientes. Outra aplicação é a utilização de algoritmos de agrupamento para a criação de grupos de consumidores a fim de criar produtos ou serviços direcionados a um nicho. Esses são alguns exemplos de aplicações comerciais que podem ser exploradas a partir da utilização de técnicas de mineração sobre tais bases de dados.

Um outro aspecto a ser analisado é que o tratamento de grandes massas de dados, mesmo para as máquinas mais modernas, ainda é um desafio em relação ao tempo de execução. Aplicações como algoritmos de Mineração de Dados [HK01] são capazes de tratar centenas ou milhares de *gigabytes* de dados. Porém, a utilização desses algoritmos, computacionalmente intensos (I^3) [GGC+05], requer um grande poder de processamento e o armazenamento dos dados em uma forma de acesso rápido para sua utilização. O emprego de computadores de grande porte (*mainframes*) para tais atividades na maioria dos casos torna-se inviável. Dado o alto custos de aquisição e manutenção deste tipo de equipamento somente operações de grandes empresas com respeitado faturamento podem usufruir de tais equipamentos. O processamento paralelo e distribuído é uma solução economicamente mais viável. Com a aquisição de um conjunto de estações de trabalho é possível a construção de um "super computador" paralelo. O poder computacional dessas estações de trabalho também teve um crescimento significativo na última década. Existem trabalhos que descrevem o quão longe pode ir essa capacidade [BG97]. Estes dois fatos, o custo e a sua evolução, tornam a estação de trabalho uma nova opção para a realização de tarefas de processamento mais intenso.

Entretanto, aplicações de processamento paralelo e distribuído são mais complexas e conseqüentemente mais difíceis de serem implementadas do que uma aplicação seqüencial em um único computador. A execução de um algoritmo de forma paralela e distribuída pode gerar uma explosão combinatória de estados possíveis das operações entre os diversos processadores. A troca de mensagens, necessárias para a comunicação dos filtros nesse ambiente, deve ser tratada com cuidado pois existem o problema de sincronia das mesmas. Esse fato aumenta muito o nível de dificuldade de geração e de depuração de código, elevando o custo de produção do mesmo. Acredita-se que este seja um dos principais impecilhos à adoção de soluções paralelas

e/ou distribuídas. Como já foi mencionado, o custo de aquisição e manutenção dos equipamentos é muito inferior ao de computadores de grande porte. Dessa forma a implementação de sistemas que provêm ambientes distribuídos se tornam mais atrativos financeiramente do que a utilização de super computadores.

O *Anthill* [FJGD05] é um ambiente de suporte a processamento paralelo e distribuído. Ele provê um *framework* para desenvolvimento e execução dos algoritmos. Um algoritmo nesse ambiente pode ser dividido em "partes" (filtros) que se comunicam e executam de forma independente em *clusters* de computadores, elevando o poder de computação. Esse ambiente permite a implementação mais eficiente de algoritmos paralelos pois provê um rico arcabouço para a execução desses. Toda a gerência das execuções e uma interface de programação que abstrai vários aspectos da programação paralela são oferecidos no *Anthill*. As estações de trabalho se tornam nós em um *cluster* de processamento. Em muitos casos, o crescimento do poder de processamento é quase linear em relação ao número de processadores, e além disto permite a utilização de ambientes heterogêneos e fisicamente distribuídos. A grande capacidade de processamento e a escalabilidade do *Anthill* o tornam uma solução excelente para processamentos intensos.

Contudo a decomposição do algoritmo seqüencial em filtros não é intuitiva para programadores que não estão habituados com o ambiente do *Anthill*. O modelo de programação procedural ou orientada a objeto utilizados pela maioria dos programadores não possui os conceitos de programação utilizados no ambiente. A complexidade de implementação dos algoritmos de forma paralela e a sua organização em filtros não é trivial. A escrita de código seqüencial em uma linguagem de alto nível, que possa abstrair os detalhes de execução do *Anthill*, torna o mesmo uma opção mais próxima a mais programadores. É proposto, nesse trabalho, um modelo de programação onde os algoritmos são implementados nos processos tradicionais e, a ferramenta faz a transformação, de forma semi-automática, em filtros para a execução no *Anthill*. Isto torna esse ambiente uma opção mais simples e mais produtiva para a sua utilização.

Nesse trabalho é apresentado um processo de particionamento de código seqüencial em filtros *Anthill*. A partir da implementação seqüencial do algoritmo de Mineração de Dados esse processo gera de forma automática código dos filtros. Analisando as dependências de dados existentes no código seqüencial e extraindo um grafo da execução do algoritmo é possível identificar os pontos que melhor se prestam à a divisão do código seqüencial. Essa abordagem tem um resultado final semelhante ao código gerado manualmente pelo programador a um custo de desenvolvimento inferior. A qualidade do código gerado também é avaliada neste trabalho.

O foco deste trabalho é o particionamento e a geração do código dos filtros. Optou-se por não tratar de forma automática a identificação das variáveis e de suas dependências. Esse processo foi feito manualmente instrumentando o código dos algoritmos. Para a resolução desse problema existem trabalhos como [HK01].

A partir da execução do código seqüencial instrumentado é gerado o grafo de dependência de dados do algoritmo. O grafo é definido de forma que os vértices representem os dados durante a execução do algoritmo e as arestas as dependências entre eles. Sobre o grafo é feita a identificação de ciclos de forma a gerar um grafo mínimo que represente toda a execução do algoritmo. Esse grafo mínimo identifica as partes do código seqüencial e suas dependências de dados. A partir da análise desse grafo são identificadas as possíveis divisões do algoritmo em filtros. Para a divisão é considerado o tráfego de dados entre os vértices.

Identificadas as partes do algoritmo e o fluxo de dados entre as mesmas, o código dos filtros é gerado automaticamente. São usados como insumos na geração dos filtros o código seqüencial e as informações obtidas do Grafo de Dependência de Dados, como os pontos de particionamento do código. Os filtros gerados são específicos para o ambiente do *Anthill* [FJGD05]. Para a geração dos filtros é acrescido ao algoritmo o código adicional, para tratamento de mensagens entre os filtros e para o controle de execução.

1.1 Objetivo

Este trabalho tem como objetivo apresentar uma forma automática de geração de filtros *Anthill* a partir de um algoritmo seqüencial descrito em alto nível. A implementação da ferramenta de conversão de código seqüencial em filtros *Anthill* de forma automática é o resultado buscado. O código dos filtros a ser gerado deve ser de boa qualidade, eficiente e robusto, e obtido de maneira mais simples à implementação manual dos filtros.

Uma maior e melhor exploração dos recursos oferecidos pelo ambiente do *Anthill* é possível através deste trabalho. O *Anthill* tem apresentado ganhos significativos de desempenho na execução de algoritmos de Mineração de Dados.

1.2 Organização

O trabalho está dividido em 8 capítulos, incluindo essa introdução. No Capítulo 2 são apresentados os trabalhos relacionados que contribuíram para o desenvolvimento dessa dissertação. O Capítulo 3 apresenta o *Anthill*, a plataforma alvo dos

códigos gerados nesse trabalho. Nesse capítulo são descritas as funcionalidades do ambiente.

Uma visão geral do processo de geração de filtros é descrita no Capítulo 4. Ilustrado com dois exemplos os passos do processo são descritos com detalhes.

No Capítulo 5 é apresentada a ferramenta de geração automática do código para o filtros *Anthill* e no Capítulo 6 são mostrados três casos de uso com algoritmos de mineração de dados. Os experimentos e resultados obtidos são apresentados no Capítulo 7.

As conclusões e os trabalhos futuros são descritos no Capítulo 8.

Capítulo 2

Trabalhos Relacionados

Nesse capítulo são descritos os trabalhos relacionados ao assunto tratado nessa dissertação. O desafio de geração de código paralelo de forma automática já foi objeto de vários trabalhos. Extensões de linguagens, *frameworks*, rearranjo de código, várias são as técnicas empregadas na criação de programas paralelos. A seguir são apresentados os trabalhos considerados.

2.1 Ambiente de Execução

Os ambientes de execução paralelos estudados oferecem recursos para a implementação e execução de aplicações paralelas. Aliado ao ambiente de execução estão os escalonadores que determinam onde e como as aplicações devem ser executadas nos ambientes distribuídos. Esse trabalho utiliza o *Anthill* como ambiente de execução.

O *Anthill* [FJGD05] é um ambiente de execução desenhado para permitir a implementação de algoritmos em sistemas heterogêneos e distribuídos. Ele é baseado em filtros com fluxo rotulado (*labeled stream*) e permite a execução em paralelo dos algoritmos, como um *pipeline*. Através do *Anthill* é possível implementar e executar os algoritmos de mineração de dados abstraindo vários aspectos, como o controle de troca de mensagens e a detecção de terminação dos filtros. Utilizando a API provida pelo *Anthill* o desenvolvedor pode implementar os filtros, abstraindo vários controles que são necessários para esse tipo de aplicação distribuída.

O *DataCuter* [BFK+00] é um *framework* semelhante ao *Anthill*. O próprio *Anthill* foi desenhado e implementado a partir da experiência com o mesmo. A criação do *Anthill* veio a partir da necessidade de novas funcionalidades não existentes no *DataCuter*. O *DataCuter* também provê um ambiente de execução baseado em filtros e já foi utilizado em vários projetos [FMH+97] [AFS00]. Entretanto o *DataCuter* não

permite a identificação dos *streams* entre os filtros. Esta característica é fundamental para a agregação de dados, muito comum na classe de aplicações a serem tratadas neste trabalho.

O ambiente de execução provido pelo *Anthill* permite cópias transparentes dos filtros e através dos *label stream* o mesmo se encarrega de fazer o transporte e a entrega das mensagens a seu destino. Dessa forma, o programador ao desenvolver o código dos filtros, deve apenas especificar o *label* e a função *hash* que irá identificar qual filtro deve receber cada mensagem.

O escalonamento dos filtros é tratado no trabalho [GGC+05]. O *AnthillSched* determina o número de instâncias de cada filtro de acordo com a demanda de CPU e I/O. A abordagem utilizada é baseada em uma heurística e essa é eficiente mas não necessariamente possui a solução ótima para o problema. As decisões da heurística levam em consideração o fato de que aplicações I^3 são complexas e iterativas.

Outro trabalho que trata o problema de escalonamento é o LPSched [dN03]. Neste trabalho o autor apresenta uma escalonador para aplicações *DataCuter* que utiliza o modelo de programação linear para otimizar a utilização dos recursos disponíveis. A proposta apresentada é direcionada ao *DataCuter*, mas segundo o autor pode ser estendida para outros ambientes.

Outros ambientes de processamento como *Grids* [ACBR03] [Hof04] e *clusters* Linux *Beowulf* [CNHS01] também foram avaliados. O processamento de algoritmos em *Grids* também implementa a distribuição de tarefas aos nodos da rede. Este ambiente leva em consideração que os nodos de execução podem ficar indisponíveis a qualquer momento.

2.2 Extensões de Linguagens

O HPF (*High Performance Fortran*) [Forb] é uma extensão da linguagem Fortran 90 [Fora] [Cha04]. Foram feitas duas especificações, uma em 1993 [hpf93] e uma e em 1997 [hpf97] pelo *High Performance Fortran Forum*. A programação através do modelo de mensagens (como MPI e PVM [Mac]) é apontada como um dos dificultadores da programação paralela. A proposta do grupo era a criação de uma linguagem que facilitasse essa programação. Através de uma linguagem independente da arquitetura, um mesmo programa seria executado em cada processador sobre um subconjunto dos dados. A tarefa de distribuir os dados e controlar a computação em cada processador fica a cargo do compilador.

Esta proposta de extensão de linguagens teve uma boa aceitação, apesar de o HPF não ser muito utilizado. Vários trabalhos futuros vieram explorando essa

mesma técnica. Nos trabalhos [FAJS01] [AFS00] [AJL01] [AFJS00] os autores propõem uma pseudo-linguagem que permite a implementação de algoritmos paralelos de forma simples e eficiente. Uma extensão da linguagem Java e um compilador que permite a criação de algoritmos de redução de bases de dados multi-dimensionais foram desenvolvidos. A abordagem é a especificação em alto nível das aplicações que tratam porções dessas bases de dados multi-dimensionais.

Para a linguagem C foi apresentada a UPC [CDC+99] [EGCD01]. A UPC é uma extensão da linguagem C para paralelização de código para sistemas de múltiplos processadores com espaço de endereçamento global. Os principais objetivos da UPC são prover eficiente acesso a máquinas adjacentes e estabelecer uma sintaxe padronizada para paralelização de código C. O modelo de programação pode assumir duas abordagens distintas: *strict* ou *reladed*. Na primeira opção as *threads* são executadas em um modelo seqüencial e consistente e na segunda opção o processados assume apenas a consistência local. No trabalho [Che04] o autor apresenta suas experiências sobre a geração de código C a partir de UPC. Neste trabalho também são apresentadas algumas otimizações para a geração do código final.

Foi definida [FSA01] uma forma canônica de representação da classe de algoritmos aos quais esse processo pode ser aplicado [FAS00]. O compilador que gera o código paralelo das aplicações também foi desenvolvido baseado no ambiente de execução chamado Active Data Repository (ADR) [CSS98]. Aplicações que utilizam bases de dados multidimensionais normalmente possuem tanto a entrada como a saída armazenada em discos. O ADR trata este tipo de computação de forma eficiente provendo algumas abstrações, facilitando o desenvolvimento da aplicação. O dialeto de Java especifica coleções de objetos, *loops* paralelos e variáveis de redução. A partir dessas estruturas é possível a geração de código paralelo por fissão de *loops*. Essa técnica permite identificar os pontos de "corte" ao algoritmo. A aplicação dessa técnica na paralelização de aplicações foi utilizada em trabalhos como o Microscópio Virtual [FMH+97].

Outros trabalhos também apresentam outras extensões de linguagens. Em [ALE01] foi desenvolvido um compilador e uma linguagem para simplificar o desenvolvimento e melhorar o desempenho de aplicações distribuídas adaptativas. Eles apresentam um conjunto de extensões de linguagem que permitem adaptação Uma interface de desenvolvimento específica para algoritmo de mineração de dados é apresentada no trabalho [JA01]. A interface trabalha tanto com memória compartilhada quanto distribuída. É provida uma forma fácil e rápida de implementação dos algoritmos nessa interface. Uma extensão de C++ para processamento distribuído [BBG+93] com bons ganhos de desempenho e facilidade de implementação são apresentadas no

trabalho.

2.3 Estratégias de Paralelização

A paralelização de um algoritmo pode ser realizada de várias formas, desde a completa implementação pelo programador até a utilização de técnicas que realizam essa tarefa de forma automática. A criação de código paralelo é mais complexa porque os algoritmos são normalmente escritos de forma seqüencial. É preciso fazer uma engenharia sobre o mesmo, de forma que a versão paralela obtenha o mesmo resultado da versão serial. Outro fator, talvez o principal é a implementação do algoritmo e correção de erros do código paralelo. Em uma implementação paralela é necessário o tratamento de alguns elementos que não existem no código seqüencial, como estado global e concorrência.

Em [RD95] os autores utilizam como técnica de paralelização a análise de "comutatividade" de programas seqüenciais. Essa técnica consiste na análise do código em um "grão grosso" na tentativa de comutação do código, a fim de identificar trechos cuja execução independe da ordem na qual são executados. Quando um código possui essa propriedade ele pode ser executado de forma paralela. Dessa maneira os trechos de código paralelizáveis são identificados. Para que um bloco seja comutado ele deve atender às seguintes condições: *Instance Variables*, o valor novo de cada variável da instância dos objetos do receptor de A e de B sob a ordem AB da execução deve ser o mesmo que o valor novo sob a ordem BA da execução ; *Invoked Operations*, o conjunto das operações invocadas diretamente por A ou por B sob a ordem AB da execução deve ser o mesmo que o conjunto das operações invocadas diretamente por A ou por B sob a ordem BA da execução. Os compiladores paralelizadores normalmente mantêm as dependências de dados existentes; essa abordagem é mais conservadora. Com a técnica é possível ter maiores ganhos comutando as operações e identificando os pontos onde o programa pode ser paralelizado. No trabalho [PSC93] o autor também explora essa mesma abordagem de rearranjo do código pelo compilador.

Algumas estratégias exploram casos específicos em classes de algoritmos. Em [KLA+03] foram desenvolvidas e avaliadas estratégias de execução para aplicações que processam dados em repositórios. As aplicações que utilizam tais dados freqüentemente realizam operações de agregação e redução sobre os mesmos. A característica mais importante é que as reduções e agregações são associativas e comutativas. Isso permite diferentes estratégias de paralelização. A aplicação de programação paralela estruturada, baseada em esqueletos a problemas de mineração de dados

é desenvolvida no trabalho [CV01]. Foram feitas análises sobre vários algoritmos procurando identificar padrões nessas aplicações. Identificaram-se variações nos padrões de acesso para três algoritmos. [GAK02] apresentam uma abordagem de paralelização de código seqüencial em código paralelo explorando *loops* aninhados. A estratégia é criar *loops* mais externos que podem ser paralelizados. Cada parte é enviada para uma processador. Eles usam MPI para a geração do código paralelo.

As estratégias de paralelização possuem duas formas principais, a análise estática e a análise sobre a execução. O trabalho [HAM+95] apresenta uma avaliação de uma ferramenta de paralelização. A experimentação mostra que ele detecta paralelismo de grão-grosso. Como técnicas usadas, estão a análise avançada de vetores, buscando reduções ou casos que permitam replicação de dados, e análise interprocedural, eliminando as chamadas de funções. Em alguns casos a análise estática não foi suficiente para se obter as informações necessárias para a paralelização. A maioria das estratégias estudadas exploram a identificação estática de paralelismo no código dos algoritmos. A utilização de redes neurais para a paralelização de código é explorada no trabalho [VPP97]. Segundo o autor essa abordagem fornece uma paralelização em baixo nível, ao contrário das estratégias de alto nível oferecida pelos demais sistemas.

No trabalho [WD05] os autores tratam o problema de decomposição em filtros como paralelismo de grão-grosso. São propostos nesse trabalho três algoritmos de tempo polinomial para a decomposição dos filtros: MIN ONETRIP, MIN BOTTLENECK e MIN TOTAL. Os dois primeiros são algoritmos de programação dinâmica que diferem na sua proposta de otimização. O primeiro avalia o custo da translado de um pacote através do pipeline e o segundo minimiza o tempo do passo de contenção. O ultimo é um algoritmo guloso que tenta minimizar o tempo de execução total.

Algumas estratégias de paralelização foram desenvolvidas especificamente para algoritmos de Mineração de Dados. No trabalho [HPY00] o autor propõe a manutenção de uma estrutura condensada de itens freqüentes para a geração eficiente de regras de associação. [SN95] discute duas formas de otimizar a agregação: na primeira cada "nodo" faz a sua "contagem" local e um "nodo" agregador faz o cálculo global; a segunda forma é distribuir a base em classes de forma que cada nodo possua todos os elementos da mesma classe. No trabalho [Sar91] os autores usam como estratégia de paralelização a identificação dos pontos de corte o grafo de dependências. O objetivo do trabalho é a geração de código para execução em sistemas multi-processados. O desafio é encontrar o ponto onde a paralelização não aumenta o overread. [LJA] apresenta uma pseudo-linguagem usada para implementação de

algoritmos de mineração de dados. Tais algoritmos, segundo os autores, processam grandes entradas de dados. Eles apresentam um compilador que gera o código distribuído dos algoritmos e a sua aplicação em alguns casos.

2.3.1 *AnthillPart*

O artigo [GSFJ05] apresenta a proposta do algoritmo *AnthillPart*. Esse algoritmo realiza a geração de código paralelo baseado em filtros a partir do código seqüencial. Os autores mostram uma seqüência de passos e um algoritmo em alto nível para a resolução do problema. O processo definido no trabalho tem como passos a geração de um Grafo de Dependência de Dados e a paralelização do código baseado nesse grafo. Esse trabalho entretanto não define a linguagem a ser utilizada na geração de código seqüencial nem o código dos filtros *Anthill* gerados. As suas contribuições são a proposta apresentada e uma prova formal de que é possível a realização do processo e a sua avaliação para o algoritmo de itemsets freqüentes. Como trabalhos futuros os autores propõem o mapeamento de outros programas (I^3) ; a implementação de um mecanismo de extração automática do grafo de tarefas e um compilador para a geração automática de código paralelo de aplicações *Anthill*.

2.4 Sumário

Neste capítulo foram apresentados os trabalhos para o ambiente de execução do *Anthill*, trabalhos sobre extensões de linguagens para geração de código paralelo e/ou distribuídos e as estratégias de paralelização de código estudadas. No capítulo seguinte será apresentado em maiores detalhes o ambiente do *Anthill*.

Capítulo 3

Ambiente *Anthill*

Nesse capítulo é apresentado o ambiente de execução *Anthill* e o modelo de programação para a construção de filtros para o mesmo.

3.1 Programação *Anthill*

O *Anthill* [FJGD05] é um ambiente de execução paralelo e distribuído baseado em filtros. Ele faz parte do projeto Tamanduá [Tam05] e é utilizado para a execução de algoritmos de Mineração de Dados. A utilização do *Anthill* tem apresentado bons resultados para a escalabilidade dos algoritmos de Mineração de Dados. Ele provê uma interface de programação que abstrai o tratamento de alguns problemas, como entrega de mensagens.

No *Anthill* os algoritmos são decompostos em filtros. Cada filtro do algoritmo contem uma parte do código do mesmo, ou seja, realiza uma parte do processamento. Os filtros possuem um canal de comunicação, um *stream*, onde o dado flui entre os mesmos. O diferencial desse tipo de ambiente é que cada filtro pode conter várias cópias, possibilitando um particionamento de tarefas que são "gargalos" da execução nas várias instâncias do filtro. Assim cada tarefa é distribuída através das máquinas onde os filtros estão instanciados. As várias instâncias de um mesmo filtro são transparentes entre si.

Os filtros *Anthill* são como estágios de um *pipeline* em um supercomputador. Cada estágio do *pipeline* pode ser replicado de forma transparente aumentando o poder de processamento. O *stream* de comunicação entre os filtros pode ser identificado garantindo a sua entrega a um destino específico. Através do *labeled stream* é possível realizar a partição de um estado global entre os filtros sem que seja necessário a replicação e sincronização de variáveis globais. Esse recurso fornece uma forma mais rica de programação ao *Anthill*. A criação e ativação dos filtros é gerenciada

pelo próprio *Anthill*. Ele lê as informações pré-definidas no arquivo de configuração e executa o código dos filtros na configuração descrita.

O paralelismo no *Anthill* é explorado em três níveis: paralelismo de dados, paralelismo de tarefas e assincronia. O paralelismo de tarefas é natural nesse ambiente pois cada filtro é responsável por uma parte da execução do algoritmo. O *Anthill* ao executar os filtros paralelamente gera um *pipeline* do algoritmo aumentando sua eficiência e escalabilidade. Outra dimensão de paralelismo é o de dados. Esse também é natural ao tipo de aplicações tratadas. Como as entradas de dados são muito grandes, cada filtro trata uma parte dos dados e os resultados locais de cada execução serão agregados por outros filtros. A assincronia é uma terceira dimensão muito presente nos algoritmos de mineração de dados. Normalmente esses algoritmos são iterativos e a cada iteração um resultado intermediário pode gerar ou não mais processamento a ser realizado. Dessa forma a sincronização das atividades pode fazer com que muito poder de processamento seja desperdiçado quando o processamento atinge uma barreira de sincronia.

A utilização do *Anthill* em um processamento possui algumas premissas. Normalmente para se ter ganho em eficiência na execução com o *Anthill* os dados da entrada devem estar distribuídos entre as máquinas que irão instanciar os filtros que farão o primeiro estágio do processamento, ou seja, a leitura dos dados. Os filtros correspondentes à aplicação devem ser implementados no *Anthill* e devem ser definidos quais e quantas instâncias dos mesmos deverão existir. O último passo é a execução do algoritmo e coleta dos resultados.

Os dados consumidos pelos filtros normalmente são grandes e movê-los até os filtros é ineficiente. Em operações de redução, apesar do tamanho das entradas, os resultados obtidos normalmente são de tamanho tratável. O resultado desses processamentos é ordens de grandeza menor que a entrada, assim o ideal é que os filtros sejam instanciados nas máquinas onde os dados estão armazenados.

Um algoritmo pode ser decomposto em N filtros *Anthill*. Em uma definição mais genérica, cada operação do algoritmo pode ser implementada em um filtro. Entretanto essa abordagem pode ser extremamente ineficiente. O particionamento ideal do algoritmo deve levar em consideração dois pontos principais: a estrutura de funcionamento do algoritmo e a troca de mensagens. Apesar da troca de mensagens ser gerenciada pelo *Anthill* ela ainda tem um alto custo, principalmente porque os filtros normalmente estão instanciados em máquinas distintas que se comunicam pela rede.

A estrutura do algoritmo é específica de cada aplicação e o particionamento é dependente desta forma. Para operações de redução por exemplo, o algoritmo deve

ser particionado no ponto onde apresenta uma iteração sobre uma grande quantidade de dados. Dessa forma várias instâncias realizarão reduções locais e um filtro fará a agregação dos resultados em um valor global.

Entretanto, a utilização do *Anthill* exige o conhecimento de sua API e de seu modo de funcionamento. Os algoritmos Apriori [VMF+04], Kmeans, ID3 [FJGD05] e CobWeb [GRG+05] já foram implementados no ambiente, sendo que todos foram feitos manualmente.

3.1.1 *Labeled Stream*

A abstração provida pelo *labeled stream* permite uma poderosa forma de comunicação entre os filtros. O *labeled stream* é um fluxo rotulado entre os filtros onde a origem é capaz de identificar o filtro destino. Esse destino é um filtro que detenha um identificador específico. Cada filtro gerado tem um identificador único atribuído pelo próprio ambiente. Dessa forma o filtro de origem não precisa ter conhecimento sobre quantos filtros ou onde esses filtros foram instanciados, o ambiente *Anthill* se encarrega de realizar a entrega dos dados. Podem existir várias instâncias de um mesmo filtro em cada nível da execução. A existência dessas várias cópias é transparente tanto para os filtros do mesmo nível quanto para os de níveis distintos.

Uma mensagem no *Anthill* pode ser definida como uma tupla $< l\ ,\ m >$, onde l representa o *labeled* e m a mensagem a ser enviada. Dessa maneira é possível submeter dados de um grupo de instâncias de filtros uma instância específica de um outro grupo de filtros. Para a resolução de qual filtro deve receber a mensagem do *labeled* l existe uma função *HASH* que faz o mapeamento. A função *HASH* possui um tratamento padrão onde faz a distribuição fixa das mensagens pelo número de instâncias do filtro alvo. Entretanto essa função pode ser redefinida de acordo com o tipo da aplicação.

Um ambiente de comunicação que não provê esse recurso de *labeled stream* exige que nos casos onde um resultado de um nível tenha um destino específico seja tratado pelo filtro. Em casos de redução isto é muito comum. Se o *stream* não tiver um destino pré-definido ele pode ser recebido por qualquer filtro do nível seguinte. Assim, os filtros de um mesmo nível teriam de se comunicar repassando as mensagens a seus destinos.

Esse recurso provê um modelo de programação mais rico, pois através dele é possível fazer o particionamento do estado global em cópias transparentes de filtros. O *labeled stream* aumenta consideravelmente o nível de abstração para o programador em relação ao tempo de execução do algoritmo. Definido o *labeled* e a sua função

HASH o código será o mesmo para qualquer número de instâncias do filtro ou de seus vizinhos.

3.1.2 Fluxo de Dados

Como descrito anteriormente o *Anthill* provê uma forma de endereçamento identificado de mensagem. Entretanto, as mensagens entre os filtros também podem ser enviadas em *broadcast*. Nesse mecanismo, a mensagem é replicada em todas as instâncias do filtro alvo e não apenas na instância que atenda a condição da função *HASH*. Essa outra forma de envio também é muito útil, pois em processos iterativos o resultado global final obtido no último estágio do *pipeline* deve ser enviado a todos os filtros do início do *pipeline*.

Outra característica do fluxo de dados no *Anthill* é que instâncias de um mesmo filtro não se comunicam. Assim o processamento que cada instância de um filtro realiza é independente das demais. A criação de novas instâncias depende apenas da escalabilidade do ambiente ou da semântica do algoritmo.

O fluxo padrão entre os filtros *Anthill* segue a forma ilustrada na Figura 3.1. Como pode ser observado o filtro inicial do *pipeline* obtém os dados de uma fonte de armazenamento persistente. Essa fonte pode ser um banco de dados, um arquivo ou até mesmo um *stream* gerado por um filtro especial leitor de dados.

À medida que o filtro (no exemplo o filtro A) faz o tratamento dos dados lidos ele envia o resultado ao filtro seguinte (B). Para cada resultado obtido esse pode ser enviado a uma instância diferente do filtro seguinte. Esse processo é repetido N vezes até que se atinja o filtro final (Z), que faz a agregação global. Dada a característica iterativa dos algoritmos o resultado gerado no filtro final pode realimentar os filtros iniciais (A).

O número de instância de cada filtro em cada nível depende da estrutura do algoritmo a ser tratado. Para se fazer uma agregação global de uma única classe de dados seriam necessários N filtros do tipo A (de acordo com o tamanho da entrada e a eficiência desejada) e apenas um filtro do tipo B que seria responsável por agregar os valores gerados pelos filtros do nível anterior.

3.2 Implementação dos Filtros

Um filtro *Anthill* possui em sua estrutura três sub-partes: *initFilter*, *processFilter* e *finalizeFilter*. Essas três funções são processadas em momentos específicos da execução dos filtros. A primeira função, o *initFilter*, é executada na criação do

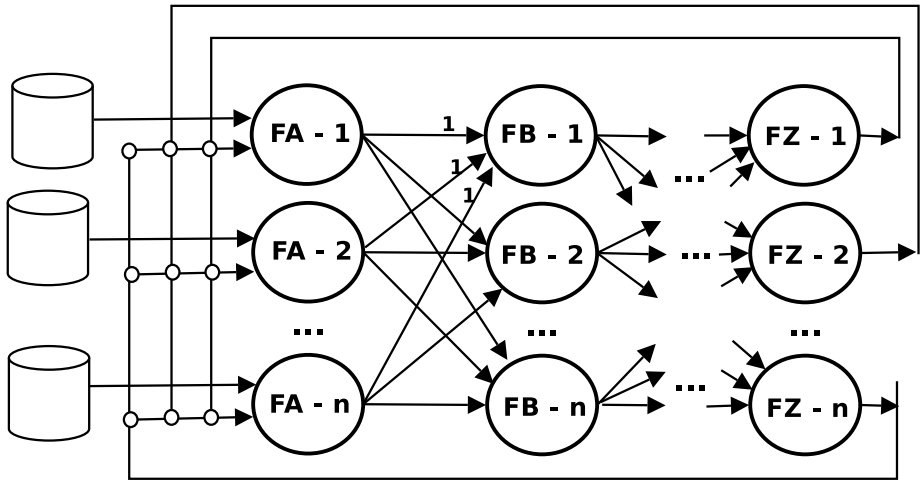

Figura 3.1: Esquema de Troca de Mensagens entre Filtros Anthill

filtro. Ela é usada para inicialização de variáveis, alocação de *stream* de dados entre os filtros, alocação memória e quaisquer pré-processamentos que devam ser feitos no carregamento dos mesmos. A função *processFilter* é executada ao final da *initFilter* e contém o corpo do algoritmo. Nessa função é feita a checagem de terminação do algoritmo.

Nos filtros que fazem a agregação de valores, normalmente essa parte se mantêm em um *loop* infinito e é finalizada pelo algoritmo de terminação, a ser descrito na próxima seção. A terceira e última função, a *finalizeFilter*, é executada no término do filtro. Normalmente essa função é usada para a geração da saída do resultado do processamento.

A API (*Application Programming Interface*) do *Anthill* conta com uma lista de funções para a implementação dos algoritmos. Essas funções fornecem aos algoritmos várias informações sobre o estado da execução e dos filtros como a identificação do filtro ou a quantidade de memória disponível na máquina.

As funções *dsWriteBuffer* e *dsReadBuffer* são usadas para o envio e recebimento de mensagens, respectivamente. Essas funções recebem como parâmetro o *stream* entre os filtros e uma estrutura pré-definida que contém a mensagem a ser trocada entre os filtros. Esta estrutura deve conter um campo numérico que representa o *labeled* de mensagem para que a mesma possa ser endereçada de maneira correta.

3.2.1 Problema de Terminação

A conclusão do algoritmo se dá no momento em que o fluxo entre os filtros se encerra. Entretanto essa condição não é identificada de maneira trivial. Um dos

filtros pode estar realizando um processamento e pode voltar a transmitir gerando novamente o fluxo de mensagens.

A execução do algoritmo em filtros pode ser modelada em um grafo direcionado. Cada filtro está ligado a todos os demais filtros do nível seguinte e a nenhum filtro do mesmo nível. Os filtros do último nível estão ligados aos filtros do primeiro nível, gerando um ciclo, que dificulta ainda mais a identificação da terminação da execução.

Para a solução do problema foi implementado um algoritmo de terminação. Foram definidas três tipos de mensagens: SUSPECT(R), TERMINATE(R) e END. Quando um filtro encerra seu processamento ele envia uma mensagem SUSPECT(R) para os filtros vizinhos. Observe que a mensagem possui um parâmetro que informa qual a rodada que este está executando. Se todas as instâncias do filtros de um mesmo nível concordarem com esse estado é enviada uma mensagem de TERMINATE(R) para o processo gerente do *Anthill*. Se o processo gerente receber a mensagem de TERMINATE(R) de todos os níveis de filtros em uma mesma rodada é assumido que o processamento foi concluído e é enviada uma mensagem END em *broadcast* a todos os filtros.

3.2.2 Arquivos de Configuração e Principal

O grafo de execução que será gerado pelas instâncias dos filtros *Anthill* é definido no arquivo de configuração. Para cada tipo de filtro, em cada nível, são definidos os sentidos dos *streams* entre os filtros e o número de instâncias que serão criadas.

Nesse arquivo também é definido o tipo de comunicação entre os filtros, se as mensagens serão de *labeled stream* ou *broadcast*.

O arquivo principal possui a função *main* que instância e dispara os filtros. No *main* é criado o *work*, uma estrutura que é passada a todos os filtros com as informações relevantes à execução do algoritmo. Esse processo também é responsável por remover as instâncias dos filtros quando todos os *works* são finalizados.

3.3 Sumário

Nesse capítulo foi apresentado o *Anthill*. Esse é o ambiente alvo do código gerado automaticamente nesse trabalho. Foram descritas as características do ambiente *Anthill* e o seu funcionamento.

No próximo capítulo é definida a classe de aplicações sobre as quais o processo de geração de filtros será realizado.

Capítulo 4

Modelo de Aplicação

Neste capítulo é apresentado a natureza das aplicações sobre as quais esse trabalho é focado, algoritmos de Mineração de Dados. Estes algoritmos normalmente trabalham com grandes massas de dados na busca de informações. Os resultados gerados pela mineração é ordens de grandeza menor do que as entradas utilizadas.

4.1 Classe de Algoritmos

Nesta seção é descrita a classe de algoritmos às quais o processo definido nesse trabalho pode ser aplicado. Algoritmos de Mineração de Dados em sua maioria podem ser modelados no algoritmo padrão que será definido. Para a geração automática dos filtros é necessário identificar nos algoritmos uma forma padrão. Feito o mapeamento do algoritmo no formato a ferramenta poderá gerar o código dos filtros.

A classe de algoritmos aos quais esse trabalho se aplicam pode ser definida com a de algoritmos cíclicos seqüenciais. Um algoritmo que possui uma seqüencia de passos definida e executa esse pessoas seqüencialmente em um *loop*. Estruturas de decisão alteram o fluxo de dados somente localmente não alterando o fluxo do algoritmo como um todo. Um exemplo desta classe de algoritmos seria um grafo onde existe apenas um caminho a ser percorrido e um ciclo. Cada vértice do grafo representa uma função sendo que internamente ela pode ter vários fluxos alternativos. O fluxo de um vértice para outro sempre segue pela mesma aresta.

O algoritmo padrão pode ser descrito como uma seqüência de funções. O resultado da execução de cada função pode ser usado como entrada na execução da função seguinte ou para iniciar um novo ciclo. Essas são características necessárias para o funcionamento desta proposta de geração de código. O Algoritmo 1 demonstra, em uma linguagem de alto nível, a estrutura desse Algoritmo Padrão.

Algoritmo 1: Forma Padrão do Algoritmo

```
1 begin
2     Trabalho = Entrada(DADOS)
3     foreach T ∈ Trabalho do
4         T1 = ExecutaA(T)
5         T2 = ExecutaB(T1)
6         ...
7         Tn = ExecutaN(Tn-1)
8         Incluir(Tn, Trabalho)
9 end
```

A partir dos DADOS de entrada é gerado um **Trabalho** a ser executado. O **Trabalho** é basicamente o dado na sua forma bruta ou armazenado em alguma estrutura de redução. Em um *loop* cada item **T** de **Trabalho** é processado por uma função. Esta função gera um novo produto a ser processado por uma outra função. O algoritmo pode ter uma seqüência de funções onde o resultado gerado por uma dessas é insumo da seguinte. Para algoritmos iterativos o resultados do último processamento pode gerar uma entrada para a primeira função. Assim uma nova rodada é disparada sobre o novo **Trabalho** gerado.

Algoritmos que possuem esta forma padrão permitem a aplicação do gerador proposto neste trabalho. Para algoritmos que possuem uma forma diferente desta os passos de identificação dos filtros podem não encontrar uma solução correta.

4.1.1 Forma geral dos Filtros

A partir do algoritmo apresentado na seção anterior foi definida uma forma padrão para os filtros. Um algoritmo que possui a forma definida pode ter os filtros gerados de forma automática. Os algoritmos de Mineração de Dados normalmente realizam algum tipo de redução dos dados, buscando um padrão ou uma característica comum nos mesmos. O Algoritmo 1 mostra uma forma padrão de um algoritmo de redução.

Um processo de particionamento pode ser definido para os algoritmos que podem ser modelados nessa forma padrão. O filtro inicial trata o recebimento de **Trabalho**, que pode ser a leitura de um arquivo e a contagem das ocorrências de um determinado padrão. Os filtros seguintes recebem os resultados e fazem sucessivas reduções ou geram mais Trabalhos. A forma padrão dos filtros gerados é mostrado nos Algoritmos 2 e 3. Como pode ser visto cada filtro processa uma função **Executa** e o resultado gerado é enviado ao filtro seguinte e assim sucessivamente.

Algoritmo 2: Filtro 01 de redução padrão

```
   Entrada: Dados
   Resultado: Tn
1  begin
2      Trabalho = Entrada(DADOS)
3      foreach T ∈ Trabalho do
4          T1 = ExecutaN(T)
5          SendMsg(T1)
6          RecvMsg(Tn)
7          Incluir(Tn, Trabalho)
8  end
```

O filtro do Algoritmo 2 além de executar a primeira função faz a inicialização do **Trabalho**. Normalmente a replicação desse filtro é responsável pelo paralelismo de dados, onde cada instância irá tratar uma parte da entrada. Outra particularidade desse filtro é que, por ele ser o primeiro estágio do *pipeline* ele possui uma seqüência de envio e recebimento de dados diferente dos demais. O filtro realiza o processo, envia seu resultado e aguarda um possível retorno que irá gerar mais processamento.

Algoritmo 3: Filtro N de redução padrão

```
   Entrada: T
   Resultado: Tn
1  begin
2      while RecvMsg(T) do
3          Tn = Executa(T)
4          SendMsg(Tn)
5  end
```

O Algoritmo 3 mostra a forma padrão dos demais filtros. A variação entre os filtros é a função **Executa**. Esta função corresponde à tarefa de cada filtro do algoritmo. Os filtros desse tipo, como pode ser visto no algoritmo, somente são "ativados" com o recebimento de dados.

4.2 Processo de Geração de Filtros *Anthill*

Nesta seção serão apresentados dois exemplos que apresentam o processo de geração de filtros a partir do algoritmo seqüencial. Os algoritmos serão mapeados na forma padrão e serão decompostos em filtros no modelo do *Anthill*.

4.2.1 Algoritmo de Contagem

Para apresentar o processo de geração dos filtros será utilizado como exemplo um algoritmo seqüencial de contagem de moedas apresentado em alto nível no Algoritmo 4. Este algoritmo recebe como entrada um arquivo onde cada tupla possui um valor que representa uma moeda. O objetivo do algoritmo é realizar a leitura de todos os valores do arquivo de entrada e calcular o valor total das moedas lidas.

Algoritmo 4: Algoritmo Seqüencial

```
   Entrada: Arquivo contendo as moedas, uma por linha
   Resultado: Valor Total das Moedas
 1 begin
 2     Arq = Open(entrada)
 3     foreach item ∈ Arq do
 4         if item = 5 then
 5             Cont5++
 6         else if item = 10 then
 7             Cont10++
 8         else if item = 50 then
 9             Cont50++
10     Final = (5 * Cont5) + (10 * Cont10) + (50 * Cont50)
11     Print(Final)
12 end
```

Sendo o arquivo de entrada suficientemente grande, esse algoritmo simples terá um tempo de execução que pode ser inaceitável para o tipo de aplicação. Ou, no pior caso, o arquivo de entrada ser muito grande e ser necessário armazena-lo de forma distribuída entre várias máquinas.

Nesse cenário o *Anthill* é uma solução simples e efetiva. O mesmo algoritmo pode ser decomposto em três filtros *Anthill*. No primeiro filtro, o Filtro Contador do Algoritmo 5, é adicionado praticamente o mesmo código da versão seqüencial, exceto o calculo final. Assumindo que a entrada está distribuída entre algumas máquinas esse filtro irá operar somente sobre a base de dados local na máquina em que ele está instanciado. O Filtro Contador ao contrário do Algoritmo Seqüencial, não possui um resultado global e envia a sua contagem parcial para um outro filtro, que irá realizar a redução de todos os resultados parciais para as moedas de cada valor.

O Filtro Agregador, apresentado no Algoritmo 6, recebe mensagens com os valores contagens parciais de cada Filtro Contador. Esse filtro, como apresentado anteriormente fica aguardando uma mensagem para o processamento. Cada mensagem possui como *label* o valor da moeda correspondente à quantidade calculada.

Algoritmo 5: Filtro Contador

```
   Entrada: Parte do arquivo contendo as moedas, uma por linha
   Resultado: Valor Parcial do número de moedas
 1 begin
 2     Arq = Open(entrada-par)
 3     foreach item ∈ Arq do
 4         if item = 5 then
 5             Cont5++
 6         else if item = 10 then
 7             Cont10++
 8         else if item = 50 then
 9             Cont50++
10     SendMsg(5, Cont5)
11     SendMsg(10, Cont10)
12     SendMsg(50, Cont50)
13 end
```

Dessa forma, todos os valores parciais para um mesmo valor são encaminhados sempre ao mesmo filtro. Estes valores são somados gerando um resultado global para cada um dos valores de moeda. Após o recebimento de todas as mensagens dos filtros contadores é enviada uma nova mensagem com o resultado global de quantidade de moedas que o filtro agregou. Instanciando três cópias do Filtro Agregador cada um desses será responsável por fazer a redução das moedas de um dos valores possíveis: 5, 10 e 50.

Algoritmo 6: Filtro Agregador

```
   Entrada: Total parcial do número de moedas de um valor
   Resultado: Total do número de moedas de um valor
1 begin
2     while RecvMsg(id, valor) do
3         Cont+ = valor
4     SendMsg(id, Cont)
5 end
```

O terceiro filtro, o Filtro Final do Algoritmo 7 recebe as quantidades globais de cada valor de moedas dos filtros agregadores e calcula o valor total das moedas. Este resultado será então enviado para a saída padrão. Para esse algoritmo especificamente, somente uma instância desse filtro deve existir, ou uma seqüência hierárquica que levará novamente a um único filtro. Esta é uma característica do algoritmo e não do ambiente. Um único filtro em algum momento da execução precisa receber

as quantidades totais de todos os valores para o cálculo do resultado final.

Algoritmo 7: Filtro Final

Entrada: Total parcial do número de moedas de um valor
Resultado: Total do número de moedas de um valor

```
1 begin
2 |  while RecvMsg(id, valor) do
3 |  |  Final+ = id * valor
4 |  Print(Final)
5 end
```

Buscando uma maior eficiência na execução desse algoritmo no *Anthill* poderão ser utilizadas vários servidores. Para o Filtro Contador podem ser instanciados tantos filtros quanto o número de máquinas que contenham uma fração da base de dados. Para o filtro Agregador a quantidade de instâncias é limitada pelo número de valores de moedas, pois a função *hash* apenas irá endereçar para um filtro que vai agregar uma das opções de moedas existe. O Filtro Final deve ter somente uma instância. Na Figura 4.1 é ilustrado em um grafo direcionado como seria a organização e o fluxo de dados dos filtros em execução no *Anthill*. Nesse modelo os filtros são os vértices e os *streams* de dados são as arestas.

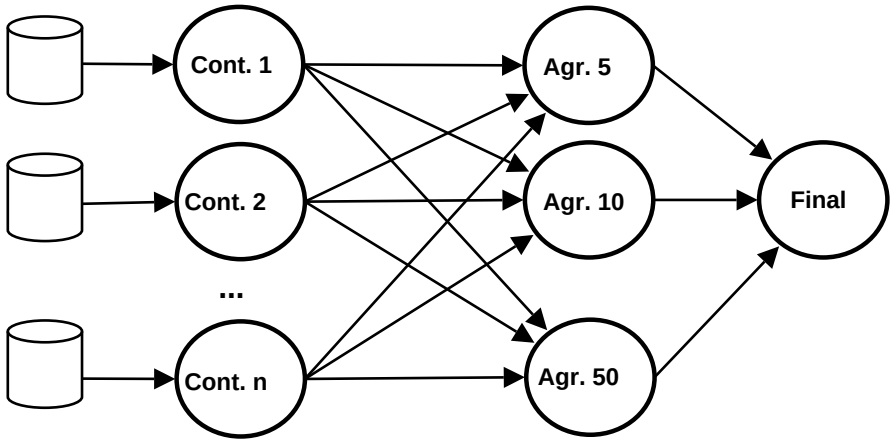

Figura 4.1: Instâncias dos Filtros no *Anthill*

Nesse exemplo o algoritmo de contagem foi particionado em três filtros *Anthill* permitindo o tratamento de bases de dados maiores e de forma mais eficiente. Também foram explorados o paralelismo de dados e o paralelismo de tarefas. Na seção seguinte será utilizado um exemplo de algoritmo iterativo.

4.2.2 Algoritmo de Contagem Iterativo

No algoritmo de contagem de moedas da seção 4.2.1 foi possível demonstrar a ocorrência de uma redução e o seu tratamento. São gerados resultados parciais sobre os dados existentes em cada computador e filtros de agregação fazem a soma desses valores gerando o resultado global para o cálculo final. Este algoritmo entretanto não é iterativo. Os resultados gerados no Filtro Agregador do exemplo dado são enviados ao Filtro Final e não geram mais computação. Os algoritmos de Mineração de Dados aos quais se pretende utilizar esse trabalho são iterativos.

Estendendo o exemplo anterior, foi gerado o Algoritmo 8 de contagem iterativo onde é feita a contagem apenas da moeda de maior valor. Caso o número de moedas total gere um valor inferior a um parâmetro informado, serão contadas as moedas de valor inferior e calculado um novo valor para os dois tipos de moedas, e assim sucessivamente até utilizar todos os valores existentes.

Algoritmo 8: Algoritmo Seqüencial Iterativo

```
   Entrada: Arquivo contendo as moedas, uma por linha
   Resultado: Valor das Moedas
 1 begin
 2     Arq = Open(entrada)
 3     maiorvalor = 50
 4     while Final < NUMERO do
 5         foreach item ∈ Arq do
 6             if item = maiorvalor then
 7                 Cont++
 8         Final+ = (maiorvalor * Cont)
 9         maiorvalor = NextVal()
10     Print(Final)
11 end
```

Para esse novo algoritmo serão gerados apenas dois filtros: um filtro Contador e um Agregador. Esta versão do Filtro Agregador irá, após receber todas as mensagens dos filtros contadores, enviar esse resultado a todos os filtros contadores para que os mesmos possam decidir se irão continuar a contagem de moedas de outro valor.

O código do Filtro Contador, no Algoritmo 9, ao final da contagem envia o resultado parcial como no exemplo anterior. Entretanto nesse caso ele recebe do Filtro Agregador o valor global da contagem e calcula o valor total das moedas já contadas. Caso esse valor não seja suficiente ele irá reiniciar o processo. Esta mesma decisão será tomada por todas as instâncias do Filtro Contador.

Algoritmo 9: Filtro Contador Iterativo

```
   Entrada: Total parcial do número de moedas de um valor
   Resultado: Total do número de moedas de um valor
 1 begin
 2     Arq = Open(entrada-parc)
 3     maiorvalor = 50
 4     while Final < NUMERO do
 5         foreach item ∈ Arq do
 6             if item = maiorvalor then
 7                 Cont++
 8         SendMsg(maiorvalor, Cont)
 9         RecvMsg(Final)
10         maiorvalor = NextVal()
11     Print(Final)
12 end
```

Algoritmo 10: Filtro Agregador Iterativo

```
   Entrada: Total parcial do número de moedas de um valor
   Resultado: Total do número de moedas de um valor
1 begin
2     while TRUE do
3         while RecvMsg(id, valor) do
4             Total+ = id ∗ valor
5         SendMsg(id, Total)
6 end
```

O Filtro Agregador, do Algoritmo 10, se difere do exemplo anterior apenas por possuir um *loop* infinito, de forma a estar disponível quando o Filtro Contador identificar a necessidade de mais uma iteração. Nesse exemplo, pode existir apenas uma instância desse filtro porque é feita a contagem de apenas um valor de moeda a cada iteração. Na Figura 4.2 é mostrado o grafo de execução desses filtros no *Anthill*.

Esses exemplos mostram como podem ser gerados manualmente os filtros *Anthill*. A partir da implementação do algoritmo seqüencial, processo normal de desenvolvimento, são identificados os pontos onde o paralelismo pode ser aplicado.

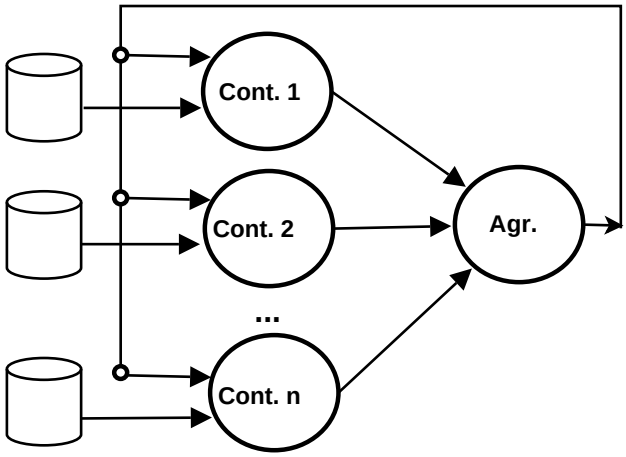

Figura 4.2: Instâncias dos Filtros no *Anthill* do Algoritmo Iterativo

4.3 Linguagem

A linguagem utilizada nos filtros *Anthill* é **C**. Neste trabalho os algoritmos devem ser implementados em C, com algumas restrições. A implementação seqüencial não pode conter funções. Esta restrição se deve ao fato de o ponto de corte poderia ocorrer no código de um desses procedimentos. Em trabalhos futuros essa restrição pode ser reavaliada se for feito um passo que serialize todo o código.

A segunda e principal restrição é a não existência de recursividade no algoritmo. Caso o algoritmo seja naturalmente recursivo ele deve ser implementado em uma versão que não explore tal facilidade. Essa restrição tem o mesmo principio da anterior. Nessa implementação não foi tratado o caso de uma chamada recursiva, todavia não há nenhuma restrição que impeça a partição do algoritmo em filtros nesse caso.

A existência de estruturas complexas não foi totalmente avaliada, podendo assim existir alguma que venha a dificultar o particionamento dos filtros nessa implementação. Essas restrições entretanto não limitam a capacidade da linguagem sendo que neste trabalho o resultado será com aplicações reais.

4.4 Sumário

O objetivo desse capítulo é definir um processo para realizar a geração dos filtros de forma automática. Como foi mostrado existe uma classe de algoritmos que possuem uma forma padrão que possibilitam a geração de filtros de forma automática. São necessários dois passos para a obtenção dos filtros: a identificação das funções

existentes nos algoritmos e a geração do código dos mesmos.

Capítulo 5

Geração dos Filtros

A Geração Automática de Filtros é um problema complexo e pode ser particionado em dois sub-problemas: a identificação dos filtros e a geração do código dos mesmos. Nesse capítulo será descrito como resolver o primeiro problema de forma semi-automática, pois serão feitos alguns passos manualmente. A partir de um código seqüencial com algumas anotações dos pontos de divisão do algoritmo, o códigos dos filtros será gerado automaticamente.

5.1 Análise do Programa Seqüencial

A partição do código seqüencial em filtros *Anthill* depende de cada algoritmo. De acordo com a semântica do algoritmo podem ser gerados dois, três ou N filtros. Entretanto, podem ser identificados alguns padrões no algoritmo que indicam pontos de divisão dos mesmos. A abordagem usada para a identificação dos filtros é a de dependência de dados na execução do algoritmo. A análise do programa seqüencial foi feita a partir da instrumentação do código, a execução do mesmo e extração do grafo de dependência de dados.

5.1.1 Instrumentação do Código Seqüencial

O primeiro passo na identificação dos filtros é a instrumentação do código seqüencial. Neste trabalho esse é um passo executado de forma manual, apesar de que o processo pode ser feito de forma automática. Esta decisão foi tomada porque esse é um pré-processamento e está fora do escopo proposto para esse trabalho.

Para a instrumentação do código o programador deve identificar as variáveis que mantêm os dados do algoritmo e as suas dependências. Uma dependência é identificada através de atribuição ou de cálculos onde os resultados são armazenados

em outra variável. Caso o dado de entrada seja atribuído a variável X e é realizada uma operação sobre X resultado em uma saída armazenada em Y essa relação deve ser identificada e assinalada.

Dessa relação de dependência é gerada uma saída de forma a criar um grafo demonstrando tais dependências durante execução do algoritmo. Na linguagem **C** a instrumentação será nada mais que um comando de impressão (*fprintf*) direcionado a um arquivo de saída. Esta saída deve ser no formato do DOT [KN], um padrão de geração de grafos. Este padrão possui uma ferramenta o Graphviz [gra] que gera as imagens a partir da definição dos grafos.

No Algoritmo 11 é mostrado um exemplo de como deve ser feita a instrumentação do código seqüencial. O grafo de execução irá mostrar que o conteúdo da variável **Final** é gerado a partir do conteúdo das variáveis: **Cont5**, **Cont10** e **Cont50**. Caso o valor da variável **Final** seja computado em um filtro diferente do qual foram calculados os valores dos contadores, esses serão enviados ao filtro que realiza o cálculo da variável **Final**. Outra informação importante é o tipo de conteúdo atribuído de uma variável a outra. No exemplo foi um valor *Int* (Inteiro).

Algoritmo 11: Exemplo de Código Instrumentado

```
1 begin
2 |   ...
3 |   Final = (5 * Cont5) + (10 * Cont10) + (50 * Cont50)
4 |   Print("grafo.dot","Cont5 → Final[Int]")
5 |   Print("grafo.dot","Cont10 → Final[Int]")
6 |   Print("grafo.dot","Cont50 → Final[Int]")
7 |   ...
8 end
```

A instrumentação do Algoritmo 4 deve conter as dependências existentes entre as variáveis: **tupla**, **Cont5**, **Cont10**, **Cont50** e **Final**. Cada variável **tupla** corresponde a uma linha do arquivo de entrada. Na Figura 5.1 é mostrado o grafo de uma execução do algoritmo. Nesse exemplo a entrada possuía apenas cinco tuplas. Como pode ser visto, as variáveis **item** e **Arq** não estão presentes nesse esquema de execução. Estas variáveis não contêm informações durante a execução, elas são apenas auxiliares para a iteração sobre os dados do arquivo e para a referência ao arquivo. Existem técnicas que permitem identificar tais tipos de variáveis

As anotações devem informar a variável origem do dado, a variável destino do dado e o tipo de conteúdo atribuído. Devem ser identificados um dos tipos: *Int*, *Float*, *String*, *Array* e *Struct*. O tipo da aresta é identificado pelo tipo da variável que está sendo usada como fonte do dado. No caso das variáveis **ContX** elas recebem

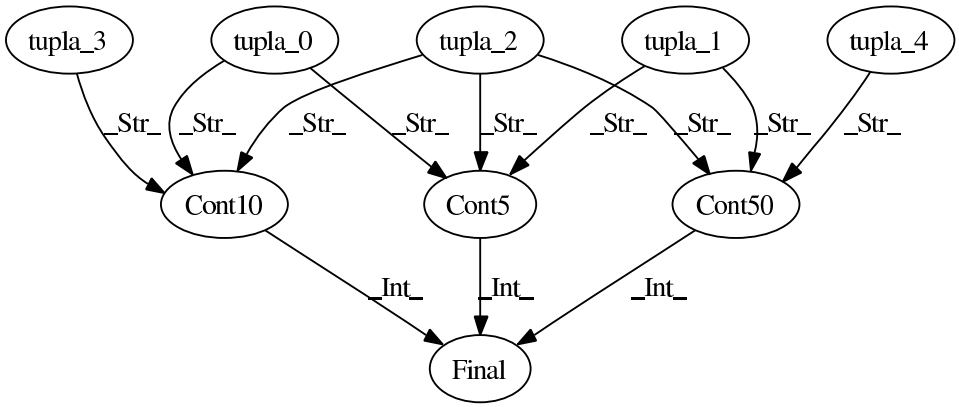

Figura 5.1: Grafo Dependências Algoritmo de Contagem

o conteúdo de um *String*, pois o dado é a tupla lida da entrada de dados. Outra característica especial das variáveis **ContX** é que elas são variáveis de redução. Isto se deve ao fato de que seu valor é calculado dentro de um *loop* que faz a iteração sobre todos os registros da entrada de dados. A variável de redução também deve ser assinalada como tal para que seja tratada.

Como pode ser observado essa técnica de extração das dependências entre as variáveis foi feita manualmente. Como já mencionado, não é o foco desse trabalho a identificação automática dessas dependências e optou-se por esse método.

5.1.2 Particionamento do Grafo de Dependências

O segundo passo no particionamento do algoritmo em filtros é a partição do grafo de dependências. No exemplo do Algoritmo 4, a partir da entrada são contadas as ocorrências de cada moeda de cada valor e armazenados nas variáveis contadores. Ao final da contagem os valores nos contadores são usados para o cálculo do valor final. O grafo possui três tipos de vértices: tupla, contadores e final. O fluxo entre os vértices **Tupla** e **Contadores** é igual a todo conteúdo lido do arquivo de dados. A divisão desses dois vértices em dois filtros distintos iria gerar um grande fluxo de mensagens entre os mesmos. Como apresentado no algoritmo padrão da Seção 4.1.1, a geração de **T**rabalho é feita no primeiro filtro. A melhor opção é transformar os vértices **Tupla** e **Contadores** em um único vértice.

Em uma execução sobre uma base real existirão milhares de tuplas. Os vértices **Contadores** possuem uma variável de redução. Como apresentado no algoritmo de redução padrão serão feitas contagens locais sobre cada parte da base de dados e esses

resultados serão agregados em um segundo filtro. Para isso, o vértice **Contador** deve ser particionado em dois níveis. O primeiro nível faz as reduções parciais, relativas aos dados presentes na máquina, e o segundo nível faz a agregação em um resultado global. O vértice **Final** representa o terceiro filtro. Este filtro irá receber uma mensagem de cada filtro agregador e realizar o cálculo final.

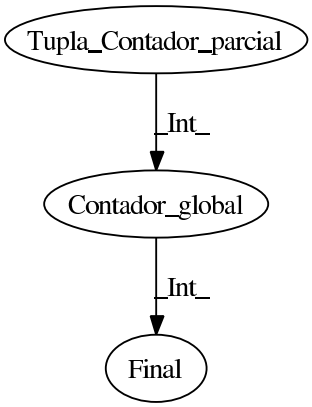

Figura 5.2: Filtros Algoritmo de Contagem

Assim, os filtros, seu conteúdo e o fluxo de comunicação entre os mesmos são dados pelo grafo da Figura 5.2. Este é o resultado dos filtros que foram implementados nos Algoritmos 5, 6 e 7. O primeiro filtro faz a redução parcial, envia o resultado para o segundo filtro realizar a agregação, e o terceiro faz o cálculo final.

Foi implementado um algoritmo que, a partir do grafo de dependências de dados, gera um novo grafo com o conjunto de filtros nos quais o algoritmo deve ser particionado. Os vértices do grafo final gerado representam os filtros e a sua identificação no código seqüencial é feita pela variável ou variáveis principais contidas no vértice.

O algoritmo de geração do código observa as seguintes características no grafo de dependências de dados: tráfego entre os vértices; *loops* entre os vértices; variáveis de redução. Arestas que possuem um grande fluxo de dados, como um *Array*, fazem com que os vértices das suas extremidades sejam agrupados. Um vértice que possua uma variável de redução é desmembrado em dois outros vértices, sendo que esses possuem uma aresta entre os mesmos no sentido do vértice parcial para o vértice global. O terceiro tratamento é a identificação de *loops* nas dependências de dados.

O Algoritmo 8, por ser iterativo, possui repetições do padrão de execução. Este padrão pode ser identificado fazendo a união dos vértices que operam sobre as mesmas variáveis com valores distintos. O grafo de dependências pode ser reduzido a uma lista encadeada onde as seqüências de variáveis se repetem em ciclos. No exem-

plo citado existe um ciclo entre os vértices que representam os filtros. O contador global do segundo filtro possui uma dependência do contador parcial para o cálculo da soma. Porém, o contador parcial possui uma dependência do contador global para identificar se será necessária a contagem de moedas de mais um valor.

Uma outra característica que depende do escalonamento dos filtros para que seja identificada são as relações de **um-para-um** e **um-para-muitos** (ou **muitos-para-um**). No exemplo tratado, somente existem relações do segundo tipo, quando ocorre distribuição de informação de um vértice para outros ou um vértice recebe dados de vários outros. A relação de **um-para-um** ocorre quando uma variável depende unicamente de um valor de uma outra variável. Por exemplo, no Algoritmo 8, após o cálculo do valor de **Final** poderia-se testar se o mesmo é maior que o valor de **NUMERO**. O resultado, um booleano, seria armazenado na variável **Teste**. Seria criado um vértices da variável **Teste**, sendo um novo filtro. Apesar disso, a relação entre os vértices **Final** e **Teste** não permite uma distribuição de carga nem uma agregação. Devido a essa característica a existência de dois filtros nesta situação não trazem ganho de desempenho ao algoritmo, podendo ser também transformados em um único vértice.

As anotações são inseridas manualmente no código seqüencial. Identificados os filtros, os mesmos podem ser implementados para o ambiente *Anthill*. Esta implementação exige a inclusão no código das variáveis de controle do ambiente e das mensagens para o fluxo de dados entre os filtros. A partir das informações obtidas do grafo dos filtros será apresentado o gerador automático dos filtros na próxima seção.

5.1.2.1 Algoritmo de Particionamento do Grafo

O algoritmo que extrai os filtros apresentados na Figura 5.2 a partir do grafo apresentado na Figura 5.1 será apresentado nesta seção.

O algoritmo está fundamentado em algumas premissas que permitem seu funcionamento correto. A primeira premissa é de que o algoritmo a ser particionado em filtros *Anthill* é cíclico e que a será possível identificar um padrão de repetição nas variáveis de controle de dados. A segunda é que serão algoritmos de redução, sendo que as variáveis de redução podem ser identificadas.

Como dito anteriormente os vértices da Figura 5.1 representam as variáveis principais de controle de fluxo de dados nos algoritmos. Tais variáveis estão classificadas em: de redução e simples. Uma variável de redução tem uma característica especial, ela recebe uma grande quantidade de dados e gera um resultado pequeno, como um inteiro.

O funcionamento do algoritmo segue os passos descritos a seguir. O algoritmo recebe como entrada um grafo onde os vértices representa as variáveis de armazenamento de dados e as arestas o fluxo de dados entre as mesmas. Nas variáveis também estão anotadas seu tipo: de redução ou simples.

O primeiro passo é identificar os vértices que estão no mesmo nível hierárquico do grafo de execução. Assumimos que a raiz do grafo é dada pela entrada de dados e a partir deste vértice os demais são classificados quanto ao nível, ou seja, a profundidade em uma busca pelo grafo.

No segundo passo, para cada nível são agrupados os vértices que tem apenas um vértice destino em comum ou que tem o mesmo vértice origem em comum. Este passo é repetido até que não seja possível mais agrupar dois vértices do mesmo nível. Concluído o agrupamento "horizontal" do grafo é dado início à identificação de padrões de repetição na lista encadeada gerada.

Para a identificação de padrões é usado uma técnica de identificação de padrões em *strings*. Sempre que uma seqüencia de vértices é identificada e esta mesma seqüencia se repete pelo grafo de dependências pode ser caracterizado um *loop* no código do algoritmo a ser particionado. Identificados os padrões no grafo os vértices que se repetem regularmente podem ser reduzido a um único ciclo.

O passo seguinte é para os vértices que contem as variáveis de redução. Esse são divididos em outros dois vértices. Isso se deve ao fato de que para que seja explorado paralelismo no *Anthill* deve ser possível estanciar várias copias de um mesmo filtro. Para isso em uma redução deve existir um único filtro que sumariza todos os resultados parciais dos filtros do nível anterior.

No ultimo passo do algoritmo os vértices que antecedem o inicio do ciclo ou após a conclusão do mesmo são mantidos. Caso o fluxo de dados entre o vértice inicial e o primeiro vértice do ciclo seja intenso, tais vértices são agrupados. Normalmente esses vértices são responsáveis pela leitura de dados e neste caso assumimos que a leitura é feita pelo primeiro filtro da execução.

No próximo capítulo esse algoritmo será exemplificado em três casos de uso.

5.2 Geração Automática de Código

Esta seção apresenta a principal contribuição desse trabalho: uma ferramenta que recebe como insumo um algoritmo implementado em **C** com anotações expressando o grafo que representa o conjunto de filtros e gera o código desses filtros para o ambiente *Anthill*.

5.2.1 Código Seqüencial

A implementação dos filtros será baseada no código seqüencial do algoritmo. No particionamento do algoritmo algumas características devem ser identificadas de forma que os filtros obtenham o mesmo resultado ao final do processamento. Devido a replicação do código nas instâncias dos filtros, código adicional deve ser inserido para manter a semântica original do algoritmo.

Algoritmo 12: Algoritmo Seqüencial Contador de Notas

Entrada: Arquivo contendo as notas de 10 e 20
Resultado: Valor Total das notas

```
1 begin
2     Arq = Open(entrada)
3     foreach item ∈ Arq do
4         Cont[item] + +
5     Final = (10 * Cont[10]) + (20 * Cont[20])
6     Print(Final)
7 end
```

O Algoritmo 12, semelhante aos exemplos anteriores, faz a contagem de notas de 10 e 20. Ele lê um arquivo de entrada, faz a redução em uma variável e calcula o valor final. A Figura 5.3 mostra o conjunto de filtros e fluxos de dados, representados pelos vértices e arestas respectivamente. Este conjunto de filtros foi extraído pelo processo já descrito.

Fazendo a distribuição do código do Algoritmo 12 nos filtros identificados, é gerada o seguinte resultado: as linhas de 2 à 4 fazem parte do primeiro filtro e as linhas 5 e 6 do terceiro filtro. No primeiro filtro é feita a redução das tuplas e no terceiro filtro é realizado o cálculo final.

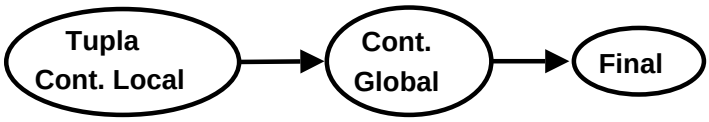

Figura 5.3: Filtros Algoritmo 12

O segundo filtro não receberá nenhuma linha desse código seqüencial. Isto se deve ao fato do mesmo ter sido gerado a partir de uma variável de redução. Assumimos que a entrada de dados é suficientemente grande de forma a demandar o paralelismo de dados. O arquivo de entrada será dividido em N partes. Reescrevendo o código para que o mesmo possa ler todos os arquivos e gerar o mesmo resultado é obtido o Algoritmo 13.

Algoritmo 13: Algoritmo Seqüencial Contador de Notas

Entrada: Vários arquivos contendo as notas de 10 e 20
Resultado: Valor Total das Notas

```
1 begin
2     foreach entrada ∈ Entradas do
3         Arq = Open(entrada)
4         foreach item ∈ Arq do
5             Cont[item] + +
6         ContTot[]+ = Cont[]
7     Final = (10 ∗ ContTot[10]) + (20 ∗ ContTot[20])
8     Print(Final)
9 end
```

Nessa nova versão do algoritmo, a agregação dos contadores de cada arquivo está explícita no código. Existem contadores parciais para cada arquivo e um contador global que agrega todos os valores em uma função de soma. A função de agregação é dependente do algoritmo. Fazendo uma nova distribuição do código seqüencial o primeiro filtro irá receber as linhas 2 à 6, o segundo filtro a linha 7 e o terceiro filtro as linhas 8 e 9. Observe que o *loop* da linha 1 envolve o código do primeiro e do segundo filtro. Assim, esse *loop* deve ser replicado nos dois filtros. Porém, a iteração é sobre os arquivos existentes. No caso do primeiro filtro é esperado que existam instâncias desse na mesma quantidade de arquivos de entrada. Pelas restrições de *hardware* é esperado que tais arquivos estejam armazenados em máquinas distintas. Já para o segundo filtro, o *loop* deve ser mantido para que o mesmo receba todos os resultados parciais e calcule a soma global do resultado.

O novo código gerado realiza a mesma função do original. Entretanto, ele é capaz de ler vários arquivos distintos. O código adicional garante que as reduções parciais sejam agregadas em um resultado global. Dado que as entradas serão repartidas, para se obter o valor global da redução, é necessário somar os resultados. Seja qual for a função de redução, essa seria identificada no código seqüencial como parte do segundo filtro e a mesma seria executada sobre o valor global da redução dos filtros do nível anterior. Esse processo é feito automaticamente pelo gerador de código no particionamento do algoritmo seqüencial em filtros.

5.2.2 Anotações

A partir do grafo resultante que representa o conjunto de filtros para o algoritmo é possível gerar as anotações no código seqüencial. As anotações contêm as informações obtidas do grafo final do processo anterior e de identificações das variáreis e

suas dependências do código. A identificação foi feita manualmente, mas pode ser gerada de forma automática pela análise de dependência de dados.

A identificação do código de cada filtro na versão seqüencial é feita por anotações no mesmo, indicando a qual filtro o código pertence e as arestas entre eles. As anotações são feitas como comentários **C** e são utilizadas apenas pela ferramenta de geração dos filtros. Toda anotação deve iniciar com /****ANT:** e ser finalizada com *****/. Outra característica é que não existem blocos de código aninhados, ou seja, se uma bloco de código pertence a um filtro um sub-bloco não pode ser definido a outro filtro. Todavia, os blocos de código podem ser de um filtro específico ou a todos os filtros. Este caso ocorre principalmente na declaração de variáveis e de definições.

As anotações:

Inicio do Arquivo : Esta anotação indica quantos filtros existem no código seqüencial e qual a forma padrão de cada filtro. No Exemplo são dois filtros sendo que, o primeiro tem a forma do Algoritmo 15 e o segundo do Algoritmo 15.

```
/**ANT: NU_FILTROS 2 */
/**ANT: TP_FILTRO F01 INICIO */
/**ANT: TP_FILTRO F02 FIM */
```

Parâmetros : Esta anotação indica os parâmetros que serão informados no *work*. Esta é uma estrutura padrão que é fornecida a todos os filtros instanciados. Para a execução de um algoritmo no *Anthill* cada filtro recebe um *work*. Dessa estrutura são extraídos os parâmetros de execução.

```
/**ANT: INI ALL */
/**ANT: PARTE RecebeParametro 1 */
/**ANT: PAR01 nome_arquivo */
```

Inicio/Fim de um bloco : Os filtros *Anthill* possuem três funções em sua estrutura, como descrito anteriormente, *initFilter*, *processFilter* e *finalizeFilter*. Cada uma dessas é indicada para um passo de processamento. Esta anotação identifica qual o filtro e a função que deve receber essa parte do código. o exemplo é marcado o início de parte do código da função *processFilter*

```
/**ANT: INI FIM F01 */
/**ANT: PARTE processFilter */
```

Aresta : Esta anotação indica uma aresta entre os filtros. Deve ser inserida no ponto entre a atribuição de uma variável e a sua utilização como fonte de

variável destino. São identificadas nessa anotação os filtros de origem e destino, a variável a ser enviada e se será através de *label stream* ou *broadcast*. No caso a variável XX é enviada com o *label* YY do filtro 01 para o filtro 02.

```
/**ANT: ARESTA F01 F02 */
/**ANT: DADO XX LABEL YY */
```

Fim do algoritmo : Esta anotação apenas indica que o algoritmo foi finalizado.

```
/**ANT: FIM PROG */
```

Algoritmo 14: Algoritmo Seqüencial Contador de Notas - Anotado

Entrada: Vários arquivos contendo as notas de 10 ou 20
Resultado: Valor Total das Notas

```
1  begin
2      /**ANT: NU-FILTROS 3 */
3      /**ANT: TP-FILTRO F01 INICIO */
4      /**ANT: TP-FILTRO F02 MEIO */
5      /**ANT: TP-FILTRO F03 FIM */
6      .
7      /**ANT: INI ALL */
8      /**ANT: PARTE RecebeParametro 1 */
9      /**ANT: PAR01 nome-arquivo */
10     .
11     /**ANT: INI F01 */
12     /**ANT: PARTE procesFilter */
13     Arq = Open(entrada)
14     foreach item ∈ Arq do
15         Cont[item] + +
16     /**ANT: FIM F01 */
17     .
18     /**ANT: ARESTA F01 F02 */
19     /**ANT: DADO Cont LABEL id */
20     .
21     /**ANT: ARESTA F02 F03 */
22     /**ANT: DADO Cont LABEL id */
23     .
24     /**ANT: INI F03 */
25     /**ANT: PARTE procesFilter */
26     Final = (10 * Cont[10]) + (20 * Cont[20])
27     Print(Final)
28     /**ANT: FIM F03 */
29     /**ANT: FIM PROG */
30 end
```

O Algoritmo 14 apresenta o exemplo dado com as anotações. Como pode ser observado, nenhum código foi atribuído ao segundo filtro. Entretanto existem duas arestas de comunicação com o mesmo.

5.2.3 Geração de Código

Nessa seção será mostrado o processo de geração do código a partir das anotações. A ferramenta desenvolvida faz a leitura do arquivo com as anotações e cria o código fonte dos filtros para o ambiente *Anthill*.

5.2.3.1 Criação dos filtros

O primeiro passo da geração do código dos filtros é a criação dos mesmos. A primeira anotação informa quantos serão os filtros e qual o tipo de cada um. São três opções: **INICIO**, **MEIO**, **FIM**. O filtro de tipo **INICIO** tem a forma do Algoritmo 2. Nesse filtro é feita a leitura dos dados e quando necessário a criação de mais trabalho. Os filtros de tipo **MEIO** e **FIM** possuem a mesma estrutura, a do Algoritmo 3. Entretanto os filtros de tipo **MEIO** enviam mensagens como *label stream* e os filtros de tipo **FIM** enviam mensagens em *broadcast*. Isto se deve ao fato de que o mesmo valor gerado no filtro final deve ser enviado a todos os filtros de tipo **INICIO**.

A identificação destes filtros é feito pelo conteúdo de cada vértice do grafo que representa os filtros e a sua organização. O filtro de tipo **INICIO** é o vértice que recebe a entrada (tuplas). Os demais filtros **MEIO** e **FIM** são classificados pela sua seqüência no fluxo de dados. O vértice anterior ao vértice de inicio é um filtro do tipo **FIM**. Os demais filtros que possam existir são do tipo **MEIO**.

Criados os filtros e o tipo de cada um, é feita a leitura do código seqüencial e cada bloco de código identificado é adicionado ao filtro especificado nas anotações. Além do código do algoritmo seqüencial, são inseridos outros códigos adicionais, como: *includes* de bibliotecas do *Anthill*; criação de variáveis de controle dos filtros; criação de variáveis de *stream*; conexão dos *streams* entre os filtros; e recebimento de parâmetros do *work*.

Cada filtro tem um número que o diferencia dos demais filtros do mesmo nível. Assim é possível definir que apenas uma das instâncias irá executar alguma operação. Por exemplo, qual filtro irá gerar o resultado final do processamento. Pode ser definido que apenas o filtro de ID 0 (por exemplo) irá gerar a saída. Todos os filtros recebem a informação de quantidade de instâncias de cada tipo de filtro nos demais níveis de execução. Esta informação pode ser obtida de outras formas em tempo

de execução, mas optou-se por fornecer este dado. Um filtro deve saber quantas instâncias existe dos demais para controlar o recebimento de mensagens.

Os filtros de tipo **MEIO** e **FIM** também têm o conteúdo de sua função *processFilter* dentro de um *loop* infinito. Como os algoritmos são normalmente iterativos esses filtros estão sempre aguardado a chegada de uma nova mensagem de dados. Como já descrito na seção 3.2.1, o algoritmo de terminação do *Anthill* se encarrega de identificar que tais filtros estão ociosos e encerrar o processamento.

5.2.3.2 Tratamento das Arestas

Durante a leitura do código seqüencial, quando é identificada uma anotação de aresta, é feito o seguinte tratamento: são lidos os filtros de origem e destino da aresta e o conteúdo a ser comunicado. No filtro de origem é incluído o código de envio de mensagem, sendo o conteúdo e o *label* da mensagem os parâmetros fornecidos na anotação. Uma estrutura padrão de mensagem contendo dois campos, um *label* e um valor, é preenchida e atribuída à função *dsWriteBuffer*. No filtro destino é feito o processo inverso. É incluído o código de recebimento de uma mensagem. Da função *dsReadBuffer* é lida a estrutura padrão e seu conteúdo é copiado para variáveis locais.

Tanto no recebimento quanto no envio da mensagem existe uma diferenciação na forma de tratar o valor recebido ou enviado. Se o filtro destino for do tipo **INICIO** então o valor é atribuído diretamente à variável local. Se o filtro for dos tipos **MEIO** ou **FIM** então o valor recebido é acumulado em uma variável de controle. Como os filtros desse tipo realizam agregações de reduções de filtros do nível anterior deverão ser recebidas as mensagens com as reduções parciais. A variável de controle é indexada pelo *label* das mensagens de forma que se um mesmo filtro de agregação realizar a agregação de vários itens os resultados serão mantidos independentes. O resultado acumulado é atribuído a uma variável local. Esta contém um valor global após o recebimento das mensagens de todos os filtros do nível anterior.

Quando o filtro de origem é do tipo **INICIO**, a mensagem é enviada assim que a execução chegar à esse ponto do algoritmo. Para os filtros os tipos **MEIO** ou **FIM** somente são enviadas mensagens após o recebimento de um número específico de mensagens. Ou seja, somente será enviada a mensagem com o resultado global da agregação após receber todas as reduções parciais.

A função de agregação é característica de cada algoritmo. No exemplo dado é feita a soma dos resultados parciais. Por padrão os valores das mensagens recebidas são acumulados. Quando o algoritmo utilizar alguma outra função de agregação a mesma estará presente no código seqüencial, por exemplo uma média dos valores lidos. O código desse cálculo irá ser atribuído ao filtro agregador que, após o re-

cebimento de todas as mensagens, irá processar o cálculo sobre os valores globais agregados.

5.2.3.3 Código Gerado

O código final dos filtros para a execução no *Anthill* para o Algoritmo 14 é apresentado nos Algoritmos 15, 16, e 17.

Algoritmo 15: Filtro 01

Entrada: Arquivo parcial contendo as notas de 10 ou 20, uma por linha
Resultado: Quantidade de cada Nota

```
1  begin
2      initFilter(work) begin
3          f2 = Conect(F2)
4      end
5      processFilter(work) begin
6          Arq = Open(entrada)
7          foreach item ∈ Arq do
8              Cont[item]++
9          msg = (item, Cont[item])
10         SendMsg(f2, msg)
11     end
12     finalizeFilter(work) begin
13     end
14 end
```

No Algoritmo 15 é feita a leitura dos dados, a redução dos mesmos para cada valor de nota e o envio para o filtro dois. Podem ser alocados tantos filtros desse tipo quanto o número de máquinas com arquivo de dados.

No código do Algoritmos 16 ocorre a agregação dos valores lidos no filtro um. O resultado acumulado para cada valor de nota é enviado ao terceiro filtro. Para esse filtro só podem ser ser criadas no máximo duas instâncias, pois existem apenas dois valores de notas.

O terceiro filtro, do Algoritmos 17, recebe os valores totais dos filtros do segundo nível e calcula o valor final.

Esse exemplo mostra como é gerado o código dos filtros pela ferramenta.

5.3 Sumário

O processo e a ferramenta de geração automática do código dos filtros foi apresentada nesse capítulo. Foram descritas também as decisões tomadas no processo

Algoritmo 16: Filtro 02

```
Entrada: Redução parcial
Resultado: Agregação Total das Notas
1  begin
2      initFilter(work) begin
3          f1 = Conect(F1)
4          f3 = Conect(F3)
5      end
6      processFilter(work) begin
7          while TRUE do
8              while TRUERecvMsg(f1, msg) do
9                  id = msg.id
10                 ContTot[id] = msg.valor
11             msg.id = id
12             msg.valor = ContTot[id]
13             SendMsg(f3, msg)
14     end
15     finalizeFilter(work) begin
16     end
17 end
```

Algoritmo 17: Filtro 03

```
Entrada: Arquivo parcial contendo as notas de 10, uma por linha
Resultado: Valor Total das Notas
1  begin
2      initFilter(work) begin
3          f2 = Conect(F2)
4      end
5      processFilter(work) begin
6          while RecvMsg(f2, msg) do
7          id = msg.id
8          ContTot[id] = msg.valor
9          Final = (10 * ContTot[10]) + (20 * ContTot[20])
10     end
11     finalizeFilter(work) begin
12         Print(Final)
13     end
14 end
```

de geração do código. No capítulo seguinte serão apresentados três algoritmos de Mineração de dados e esse mesmo processo será aplicado aos mesmos.

Capítulo 6

Casos de Uso

Esse capítulo contém os três casos de uso nos quais o processo e a ferramenta desenvolvidos foram aplicados. Foram utilizados três algoritmos de Mineração de Dados: Itemsets Freqüentes, Kmeans e ID3. Cada um destes é utilizado para descoberta de um tipo de padrão: regras de associação, agrupamentos e modelos de classificação. Como cada um destes algoritmos tem uma função distinta, será mostrado que tanto o processo quanto a ferramenta desenvolvidos são bem abrangentes.

Para cada caso será apresentado o algoritmo e os resultados do processo, o grafo de execução e o grafo final com o grupo mínimo de filtros. Será também apresentado o resultado final dos filtros gerados.

6.1 Algoritmo Itemsets Freqüentes

O algoritmo de Itemsets Freqüentes calcula o suporte dos itens e suas combinações possíveis. Ele recebe como entrada ocorrências de itens e calcula quais destes ocorreram em uma quantidade acima do valor de suporte fornecido. Esse algoritmo serve de base para vários outros como a geração de regras de associação.

Entradas de dados de maior porte, com uma considerável quantidade de itens, gera uma explosão combinatória de possibilidades. Para uma base de apenas 100 itens distintos, se forem permitidas combinações de 10 em 10 itens já são aproximadamente 19.000.000.000.000 tipos de ocorrências para serem contadas. Em um caso real, como um hiper-mercado onde existem alguns milhares de itens disponíveis e uma compra pode chegar a ter 100, itens estamos falando de uma quantidade computacionalmente inviável. Porém, uma característica muito freqüente observada nesse tipo de base de dados real é que a ocorrência dos itens na base não é uniforme. Exatamente por esse motivo se busca encontrar os itens mais freqüentes.

Esse algoritmo tem um bom desempenho por justamente explorar a ocorrência do sub-item freqüente. Para que um registro com itens de dois elementos ocorra acima de um suporte definido é necessário que os itens de um elemento que compõem o registro duplo sejam freqüentes. Por exemplo, para que existam vendas de AB acima de um suporte devem existir vendas de A e de B acima do suporte. Caso A ou B não atenda a esse requisito não é necessário fazer a contagem de AB na base.

Será dado um exemplo do funcionamento do algoritmo para uma base que contém apenas cinco produtos: pão, café, leite, açúcar e biscoito. É possível uma venda com até os cinco itens em uma única operação. São trinta e uma combinações de registros de venda possíveis. A base de exemplo possui seis registros. As tuplas desta base representam as vendas desses produtos. Para esse caso é pedido os itens com suporte acima de três ocorrências.

```
venda 01 : pão café biscoito
venda 02 : pão açúcar biscoito
venda 03 : açúcar
venda 04 : pão café leite açúcar biscoito
venda 05 : leite
venda 06 : pão
```

O cálculo dessa base com o suporte mínimo de 3 é apresentado na Tabela 6.1. A maneira mais simples seria para cada uma das 31 possibilidades verificar cada uma das 6 tuplas. Seriam 186 comparações para esse exemplo simples.

Item	Suporte
pão	4
açúcar	3
biscoito	3
pão e biscoito	3

Tabela 6.1: Suporte dos Itens

Adotando a estratégia do algoritmo de Itemsets Freqüentes é feita a contagem apenas dos itens de um elemento, sendo feitas 30 comparações. Assim, somente pão, açúcar e biscoito estão acima do suporte. A partir desse ponto somente os registros de dois elementos composto por esses itens serão contados, ou seja, 18 comparações. Como somente um item de dois elementos possui suporte acima do valor mínimo não serão contados itens com três ou mais elementos. O resultado final foi obtido pelo Itemsets Freqüentes com apenas 38 comparações.

6.1.1 Algoritmo Seqüencial e Grafo de Dependências

Para a utilização do algoritmo de Itemsets Freqüentes foi feita a implementação seqüencial do mesmo em **C**. O Algoritmo 18 apresenta o funcionamento do Itemsets Freqüentes em um pseudo-código em alto nível. A implementação real em **C** está no Apêndice A. Esse algoritmo foi desenvolvido de acordo com as restrições definidas nesse trabalho. Essa versão também não teve como preocupação principal a otimização do código seqüencial, pois isto foge ao escopo desse trabalho.

Algoritmo 18: Itemsets Freqüentes Seqüencial

```
   Entrada: Arquivo contendo os registros
   Resultado: Calculo do Suporte
 1 begin
 2     Initialize(Candidates, UmItemsets)
 3     foreach tupla ∈ Transactions do
 4         if candidate ∈ tupla then
 5             invertedList(candidate) ← tupla
 6     foreach candidate ∈ Candidates do
 7         Length(counter, invertedList(candidate))
 8         if counter > threashold then
 9             Frequents ← candidate
10         foreach frequent ∈ Frequents do
11             if frequentenablesnewCandidates then
12                 candidates ← newCandidates
13                 invertedList(candidate) ← newCandidates
14 end
```

A execução do código do Itemsets Freqüentes instrumentado para uma entrada de 4 tuplas e com três itens gera o grafo da Figura 6.1 . Como pode ser observado é feito o cálculo dos itens A e B. A partir do resultado obtido o item AB é computado.

Utilizando a ferramenta de particionamento do grafo de dependências de dados é obtido o grafo da Figura 6.2 que representa os filtros nos quais o código deve ser particionado. Os filtros representados pelos vértices **Tupla** e **CandidatoListaInvSuporte** serão agrupados em um único vértice. Isto se deve ao fato de que o primeiro vértice apenas faz a leitura do arquivo de entradas do qual a lista invertida dos candidatos é extraída.

Esse grafo e o algoritmo seqüencial serão as entradas para a geração automática do código dos filtros.

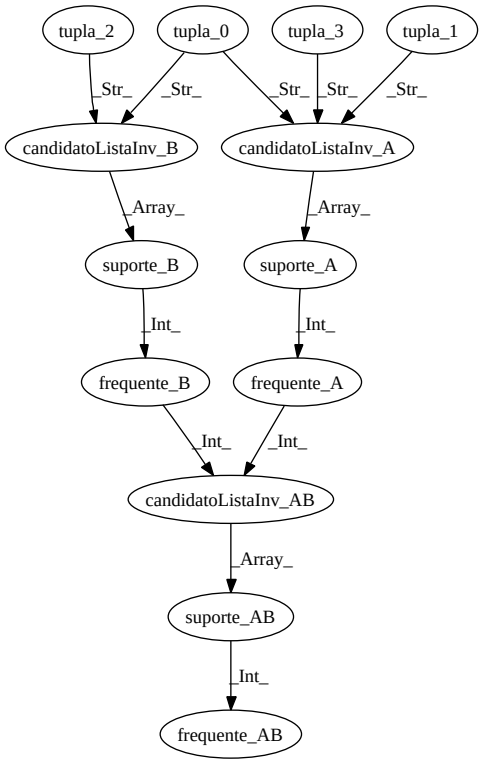

Figura 6.1: Grafo de Dependência de Dados Itemsets Freqüentes

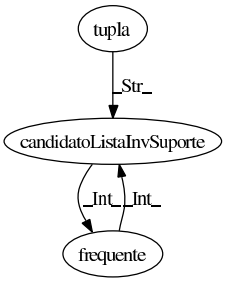

Figura 6.2: Grafo com o conjunto mínimo de Filtros

6.1.2 Filtros Gerados

Incluindo as anotações referentes ao grafo, do conjunto de filtros obtido, é feita a geração automática dos filtros do algoritmo de Itemsets Freqüentes. Os filtros obtidos desse processo estão nos Apêndices B e C.

Foram gerados dois filtros. O primeiro filtro, o **Contador**, realiza a leitura do arquivo de entrada e a geração dos candidatos iniciais, os itens de um elemento. Para cada candidato é gerada uma lista invertida da entrada e feita a contagem do suporte. O valor dessa redução é enviado ao segundo filtro com o ID do item.

O segundo filtro, o **Agregador**, recebe as reduções parciais de todos os filtros do nível um e envia de volta a todos os filtros do nível um, em *broadcast*, o valor global do suporte de cada item. Novamente no primeiro filtro é identificado se o item habilita a contagem de novos itens. Caso seja possível, um novo item é calculado, dando início a um novo ciclo.

Para o filtro **Contador** podem ser instanciadas cópias até o total do número de máquinas que contenham parte da entrada. Já para o segundo filtro o limite é o número de itens possíveis, pois apenas um filtro pode fazer a agregação global de cada item. Um mesmo filtro **Agregador** pode agregar vários itens distintos.

6.2 Algoritmo Kmeans

O Kmeans é um algoritmo de agrupamento (ou clusterização). Ele trabalha com coordenadas espaciais e agrupa os objetos de acordo com a proximidade. Ele recebe como entradas os pontos (as coordenadas), o número de grupos nos quais a base deve ser particionada (os pontos centrais de cada grupo) e quantas iterações devem ser feitas. O número de iterações é um parâmetro de segurança pois esse algoritmo possui alguns casos onde não é possível identificar a terminação.

Seu funcionamento é simples, para cada ponto é calculada a distância a cada um dos pontos centrais dos grupos existentes. O ponto é atribuído ao grupo cuja distância ao centro do mesmo seja menor. Após a atribuição de todos os pontos a um dos grupos, para cada grupo é calculado um novo centro. O novo centro é o ponto médio entre todos os pontos. Para o novo conjunto de centros calculado é feita a nova distribuição dos pontos e um novo cálculo dos centros. O algoritmo irá realizando tais iterações até o limite informado.

Na Figura 6.3 é mostrado um conjunto de pontos distribuídos em um espaço bi-dimensional. Os pontos, círculos, devem ser agrupados. Os centros, quadrados, são dados de forma aleatória.

Realizando as iterações do Kmeans os pontos são atribuídos aos grupos de acordo com os centros e o centro irá sendo "movido" para o centro de cada grupo. Ao final da execução são identificados os três grupos ilustrados Figura 6.4. Cada ponto foi atribuído ao grupo ao qual estava mais próximo de seu centro.

6.2.1 Algoritmo Seqüencial e Grafo de Dependências

O Algoritmo 19 descreve em alto nível o funcionamento do Kmeans que foi implementado em **C**. Esse algoritmo realiza iterações buscando agrupar os pontos que

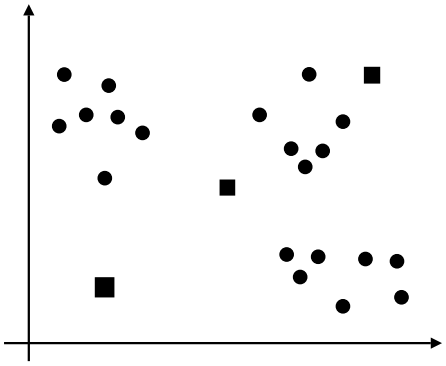

Figura 6.3: Pontos e Centros Iniciais

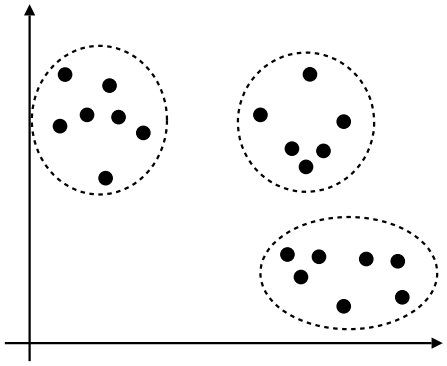

Figura 6.4: Pontos Agrupados

Figura 6.5: Exemplo Kmeans

estão mais próximos. É dado o número **N** de iterações e um conjunto de centros iniciais ao algoritmo. No primeiro *loop* cada ponto é atribuído ao grupo ao qual o ponto está à menor distância de seu centro. Para cada grupo gerado é calculado o ponto médio central desse conjunto, no segundo *loop*. Esse ponto médio passa a ser o novo centro do grupo. O contador controla o número de iterações do algoritmo.

Algoritmo 19: Kmeans Seqüencial

```
   Entrada: Arquivo contendo os pontos
   Resultado: Grupos de Pontos
 1 begin
 2     Initialize(Pontos)
 3     Initialize(Grupos,centros)
 4     contador = N
 5     while contador do
 6         foreach ponto ∈ Pontos do
 7             foreach centro ∈ Centros do
 8                 dist = Distancia(pont, centro)
 9                 novogrupo = Menor(dist)
10             Inclui(ponto,Grupo,novogrupo)
11         foreach grupo ∈ Grupos do
12             foreach ponto ∈ Grupos[grupo] do
13                 novo_centro = CalculaMedia(ponto)
14             Grupo[grupo] = novo_centro
15         contador − −
16     Saida(Grupos)
17 end
```

Cada ponto central, de cada grupo, tende a se aproximar dos locais de maior

concentração de pontos. Entretanto o algoritmo pode encontrar resultados que não representam da melhor forma os agrupamentos. Uma forma de melhorar o resultado final é ter uma escolha mais elaborada dos pontos iniciais. Nessa implementação optou-se por uma escolha aleatória pois esse fato não influencia no comportamento do algoritmo e sim no resultado dos grupos, que não é o foco desse trabalho.

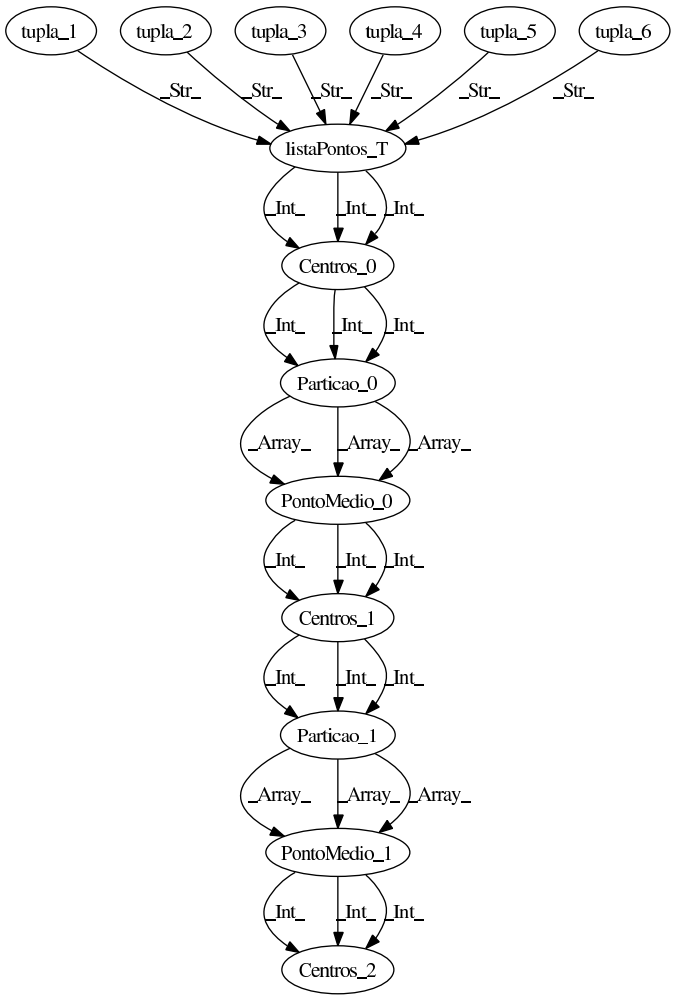

Figura 6.6: Grafo de Dependência do Kmeans

O grafo de dependência de dados é apresentado na Figura 6.6. Para essa execução foi utilizada uma entrada com 7 pontos e 3 grupos. Como pode ser observado nesse grafo as variáveis estão ligadas como uma lista. A cada iteração todos os pontos são distribuídos nos grupos, mas não há o particionamento da base de pontos.

O conjunto mínimo de filtros é mostrado na Figura 6.7. São gerados dois filtros para esse algoritmo, pois nesse caso novamente existem vértices que serão agrupados. Os vértices **Tupla** e **ListaPontos** realizam a leitura dos dados e serão integrados

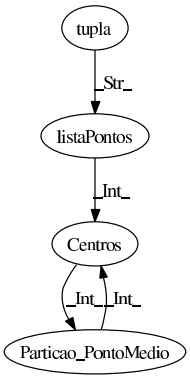

Figura 6.7: Grafo com o conjunto mínimo de Filtros

ao vértice **Centros**. Os vértices **Centros** e **Partição** também tem o mesmo valor semântico no algoritmo. Um centro define uma partição, pois o grupo de pontos é relativo ao centro.

6.2.2 Filtros Gerados

O resultado final do processo para o Kmeans foram dois filtros. Do grafo apresentado na Figura 6.7 foram agrupados os vértices **tupla**, **listaPontos** e **Centros**. Isto foi feito pois os dois primeiros vértices são a geração de **Trabalho**, até o ponto de identificação dos centros. Dessa forma o processamento ficou definido da maneira descrita a seguir. O primeiro filtro faz a leitura dos pontos e sorteia aleatoriamente alguns centros. A partir desse ponto é dado início ao algoritmo. Cada ponto é atribuído ao grupo cuja distância do ponto ao centro seja menor. Para os grupos gerados em cada filtro **Contador** é identificado o ponto médio "local" (soma das coordenadas e o número de pontos). Esse resultado é enviado ao filtro **Agregador** onde é feita a agregação global. A partir dos dados locais de cada filtro do nível um é calculado o ponto médio (ponto central) de cada grupo para todos os pontos existentes. Os novos centros são enviados pelo filtro **Agregador** aos filtros **Contador**.

Para cada parte do arquivo de entrada pode ser instanciado um filtro do tipo **Contador**. O resultado para cada grupo é enviado usando o identificador do grupo como *label*. Todos os pontos médio parciais do mesmo grupo são agregados por um filtro do nível dois. Assim o filtro do tipo **Agregador** pode ter tantas instâncias quanto o número de grupos.

6.3 Algoritmo ID3

Uma tarefa de classificação utiliza um conjunto mínimo de características conhecidas de um determinado objeto a fim de identificar o mesmo. O ID3 é um algoritmo de classificação que a partir de uma base de dados já agrupada em classes gera uma árvore de decisão que permite uma rápida classificação de novos itens. Novamente nos deparamos com um problema que possibilita uma explosão computacional com o crescimento da entrada. Identificar quais características irão ser mais relevantes para a classificação dos objetos para uma base de muitos registros é uma aplicação computacionalmente intensa.

Esse algoritmo funciona da seguinte maneira: para uma base de dados onde são conhecidos os domínios de cada propriedade do objeto é calculado o ganho de informação para cada uma dessas propriedades, sendo que o ganho de informação mede o quão bem uma propriedade identifica os itens da base. A propriedade de maior ganho de informação para a base dada é usada como o vértice raiz da árvore de decisão. São gerados vértice filhos para cada valor possível no domínios dessa propriedade. Para cada vértice filho dessa raiz são atribuídos os registros que possuem o valor da propriedade correspondente ao valor do vértice filho. Assim, nos vértice filhos, o processo é repetido.

No arquivo de entrada de exemplo são apresentados 2 propriedades e 6 registros. Os registros estão divididos em 3 classes. Também é informado ao algoritmo o domínio de cada propriedade. No exemplo a primeira propriedade pode ter os valores A, B e C.

```
propriedade 1: A B C
propriedade 2: X Y
regra 1 : A X  - C1
regra 2 : A X  - C1
regra 3 : C X  - C1
regra 4 : B Y  - C2
regra 5 : B Y  - C2
regra 6 : C Y  - C3
```

Utilizando o ID3 são identificados os ganhos de informação para cada propriedade até que seja possível identificar todas as classes. O resultado para essa entrada é a árvore da Figura 6.8. Como pode ser observado nas ocorrências da segunda propriedade onde o valor é "X", é possível identificar as tuplas da Classe 1. Caso o valor da segunda propriedade seja "Y", é analisada a primeira propriedade. Para os valores dessa propriedade são identificadas as Classes 2 e 3. Esse é um exemplo

simples para a descrição do problema, mas para grandes bases de dados, com várias propriedades e milhares de registros essa é uma tarefa complexa.

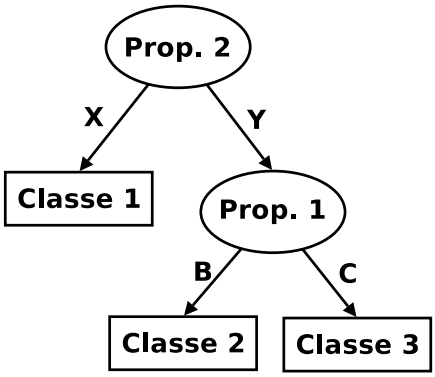

Figura 6.8: Árvore de Decisão

A escolha das propriedades para a criação dessa árvore no ID3 busca identificar o item de forma mais eficiente possível. Os vértices mais próximos à raiz são os de maior ganho de informação na identificação dos itens.

6.3.1 Algoritmo Seqüencial e Grafo de Dependências

O Algoritmo 20 descreve em alto nível o funcionamento do ID3 que foi implementado em **C**. Nesse algoritmo para cada propriedade da base de dados é calculado o ganho de informação. Essa redução deve ser realizada sobre os valores totais dos contadores da base, pois a função é baseada na entropia dos itens. A propriedade de maior contribuição será atribuída à árvore e a base é dividida de acordo com o domínio dessa propriedade. Para cada novo vértice é novamente realizado o processo.

Quando não há mais propriedades que têm ganho de informação para a árvore de decisão o algoritmo é finalizado e é informada a árvore de decisão encontrada.

O grafo de dependência de dados da execução desse algoritmo é apresentado na Figura 6.9. Para essa execução foi utilizada uma entrada com 10 registros, 3 propriedades e 4 classes. O grafo da execução assim como o resultado tem o formato de uma árvore. A base de dados é particionada à medida que a árvore de decisão é gerada.

O tratamento dos dados no ID3, no decorrer do processamento ocorre de forma diferente dos outros dois algoritmos. No Itemsets Freqüentes os resultados são fundidos para o calcula de novos itens, no Kmeans os pontos são separados e agrupados constantemente. Nas iterações do ID3 as operações são feitas com sub-conjuntos dos dados iniciais, e tais sub-conjuntos são independentes.

Algoritmo 20: ID3 Seqüencial

Entrada: Arquivo contendo os registros e os domínios
Resultado: Calculo do Suporte

```
1  begin
2      Initialize(PropriedadesValores)
3      Initialize(Tuplas)
4      foreach vertice ∈ Arvore do
5          foreach prop ∈ Prop[vertice] do
6              foreach tupla ∈ Tuplas[vertice] do
7                  contador++
8              ganhoInf[prop] = Calculo(contador)
9          prop = Maior(ganhoInf)
10         IncluiArvore(Arvore, prop)
11     Saida(Arvore)
12 end
```

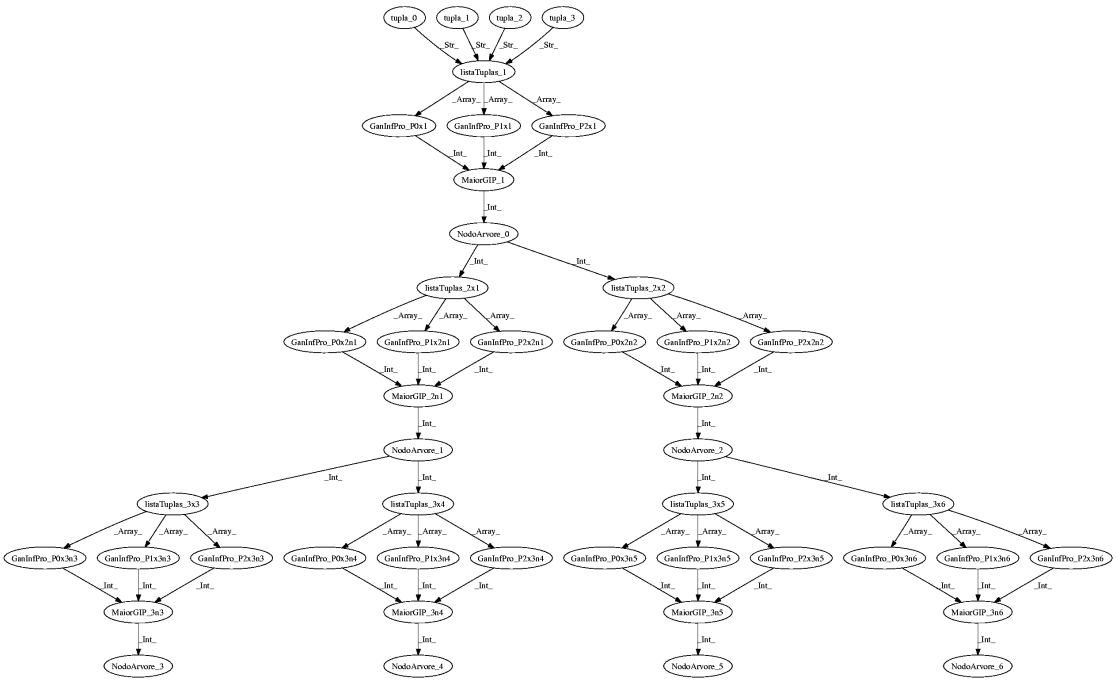

Figura 6.9: Grafo de Dependência do ID3

O conjunto mínimo de filtros é mostrado na Figura 6.10. Como pode ser visto o ID3 é particionado em três filtros. O filtro **listaTuplas_NodoArvore** faz a redução das tuplas para cada propriedade. Esses resultados são enviados aos filtros **GanInfPro** onde são feitos os cálculos do ganho de informação para cada propriedade. O terceiro filtro, o **MaiorGIP**, define qual a propriedade do novo vértice da

árvore.

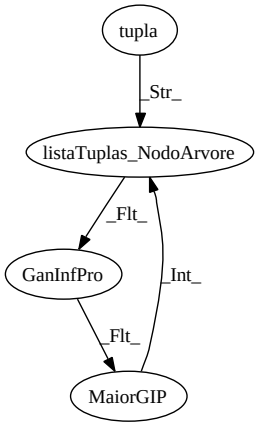

Figura 6.10: Grafo com o conjunto mínimo de Filtros

Os vértices **tupla** e **listaTuplas_NodoArvore** serão agrupados em um único filtro.

6.3.2 Filtros Gerados

O ID3 é particionado em três filtros, diferentemente dos outros dois algoritmos descritos. Do grafo da Figura 6.10 podem ser identificados os três filtros que compõem o ID3. Os três filtros fazem a redução o cálculo do ganho de informação e seleção do maior valor, respectivamente.

O filtro inicial percorre a base, ou a sub-base correspondente ao vértice da árvore de decisão, e conta as ocorrências de cada classe para cada propriedade. Essas reduções são enviadas aos filtros do segundo nível identificadas pela propriedade, ou seja, o *label* usado é o identificador da propriedade.

Cada filtro do segundo nível recebe as reduções de uma ou mais propriedades. O ganho de informação global gerado no segundo filtro a partir dos dados recebidos é para cada propriedade. Para o novo vértice da árvore é buscado o maior entre os valores calculados. Entretanto esses valores estão distribuídos pelas instâncias dos filtros do segundo nível.

O terceiro filtro é responsável por gerar o resultado final (e global) da iteração. Esse filtro recebe os resultados de ganho de informação de cada propriedade e identifica o maior valor. O resultado é então enviado aos filtros do primeiro nível em *broadcast* para a geração da árvore de decisão.

Cada parte da base de dados que estiver em uma maquina do cluster pode receber uma instância do filtros de agregação local, o filtro do nível um. Os filtros do nível

dois têm como limite superior de instâncias o número de propriedades da base de dados. O terceiro filtro deve ter apenas uma instância. Isto porque ele deve receber todos os valores de ganho de informação para cada propriedade e gerar o resultado global para a árvore de decisão.

6.4 Sumário

O capítulo apresentou o processo definido no Capítulo aplicado a três a algoritmos de Mineração de Dados. Foram gerados filtros para os três algoritmos e os mesmos serão avaliados no próximo capítulo.

Capítulo 7

Experimentos

Esse capítulo apresenta os experimentos realizados com o código gerado automaticamente para os três casos de uso apresentados. Através dos resultados das execuções poderemos avaliar a qualidade do código gerado.

7.1 Definição dos Experimentos

Essa seção apresenta a definição dos experimentos realizados para a comprovação da qualidade do código gerado.

O objetivo desse trabalho é a geração automática do código dos filtros *Anthill*. Entretanto o código dos filtros gerados de forma automática deve ser eficiente e escalável. Se a qualidade dos filtros não atender a alguns requisitos mínimos o resultado obtido não será muito proveitoso.

A avaliação da qualidade será feita através de medida do *Speedup* e *Scaleup* dos filtros. Também foi feita uma comparação entre os resultados dos filtros gerados automaticamente e de implementações manuais existentes.

Outra comparação a ser realizada é entre a execução seqüencial do algoritmo e a execução com uma instância de cada filtro em uma única máquina. Essa medida mostra o *overhead* acrescido ao algoritmo pelo código adicional dos filtros e pela troca de mensagens.

Para todos os itens que serão avaliados será feita a comparação entre os tempos de execução. Não será avaliado a utilização dos recursos de cada servidor, pois tais máquinas são de uso exclusivo para os experimentos. Uma análise mais detalhada sobre a utilização dos recursos dos servidores durante os experimentos é proposta como trabalho futuro.

7.1.1 Ambiente

Para a execução dos experimentos será usado um *cluster* com 15 máquinas. As máquinas são homogêneas e possuem processadores AMD Athlon 64 3200+ de 2 GHz, com 2 GBytes de memória RAM e discos SATA. A rede é ethernet de 1 G bits e são utilizados dois switchs para a ligação das máquinas. Os equipamentos tem dedicação exclusiva para a execução dos experimentos.

Todas as máquinas utilizam como sistema operacional o GNU/Linux. O kernel é o 2.6.15 e a distribuição Debian Testing. O *Anthill* utilizado foi a versão 3.1 .

7.1.2 Carga e Montagem de Experimentos

Os dados utilizados nos experimentos com os algoritmos de Itemsets Freqüentes e o ID3 são uma base de dados real. Para o Kmeans foi usada uma base sintética gerada aleatoriamente. Os detalhes das bases de dados usadas nos algoritmos são apresentados na Tabela 7.1.

Algoritmo	Número de Colunas/Dimensões	Número de Registros
Itemsets Freqüentes	4	79.053
Kmeans	2	5.000.000
ID3	10	316.212

Tabela 7.1: Dados utilizados

Para o algoritmo de Itemsets Freqüentes foram utilizadas 4 colunas da base original. Essas 4 colunas possuem 20 diferentes itens, gerando um total de 6.195 combinações possíveis para serem contadas. O ID3 irá trabalhar com 10 colunas da base. Os dados do Kmeans foram gerados e os pontos possuem duas dimensões.

Cada execução de cada experimento foi repetida cinco vezes. Foi selecionado o maior tempo de todos os filtros para cada execução e feita a média entre as execuções. Para a medida do tempo de execução dos algoritmos foi usada a função *gettimeofday* do **C**. Essa função retorna o tempo em segundos desde uma data pré-definida quando executada. O tempo foi marcado nos inicio e final de cada algoritmo e calculada a diferença.

Nas próximas sub-seções são descritos as montagens dos experimentos realizados, definindo a distribuição de filtros em cada uma das máquinas. Cada máquina irá receber apenas uma instância de um tipo de filtro. Assim a execução de um filtro em uma máquina não irá influenciar nos demais.

7.1.2.1 *Speedup*

Para medir a escalabilidade dos filtros para o Itemsets Freqüentes foram utilizados grupos de 2 a 11 máquinas. Pela avaliação inicial do consumo de recursos pelos filtros foi observado que o filtro leitor tem uma alta demanda de processamento enquanto o segundo filtro fica ocioso por boa parte do tempo. Para uma melhor avaliação foram montados dois experimentos.

ID	Número de Máquinas	Número de Filtros 1	Número de Filtros 2	Número de Registros
01	2	1	1	79.053
02	3	2	1	39.527
03	4	3	1	26.351
04	5	4	1	19.763
05	6	5	1	15.811
06	7	6	1	13.176
07	8	7	1	11.293
08	9	8	1	9.882
09	10	9	1	8.784
ID	Número de Máquinas	Número de Filtros 1	Número de Filtros 2	Número de Registros
10	9	7	2	11.293
11	10	8	2	9.882
12	11	9	2	8.784

Tabela 7.2: Distribuição de Máquinas e filtros - Itemsets Freqüentes

Na primeira metade da Tabela 7.2, as máquinas são acrescidas de 1 unidade para o primeiro filtro e sempre é usada 1 máquina para o segundo filtro. Dessa forma será avaliado a escalabilidade explorando o paralelismo de dados. Também é apresentado na tabela o número de registros a ser tratado por cada filtro do nível um.

Na segunda metade da Tabela 7.2, do segundo experimento, são usados dois filtros de agregação (tipo 2). Esse caso foi usado apenas para execuções com uma maior quantidade de filtros leitores (tipo 1) para que a demanda para os filtros de agregação seja considerável.

Para medir a escalabilidade dos filtros para o Kmeans foi utilizada a mesma abordagem. A Tabela 7.3 mostram as quantidades de filtros e registros de cada execução dos experimentos. O Kmeans também tem a carga de processamento concentrada no primeiro filtro e será utilizada a mesma distribuição do Itemsets Freqüentes.

O ID3 possui três filtros e a distribuição dos mesmos terá outro formato. A concentração de processamento esta principalmente no primeiro filtro, o leitor, depois no segundo que realiza alguns cálculos, mais complexos que no caso do Itemsets Freqüentes e Kmeans. O terceiro filtro tem a menor demanda de processamento pois nesse é feita apenas a seleção do maior valor e somente pode existir uma instância do

ID	Número de Máquinas	Número de Filtros 1	Número de Filtros 2	Número de Registros
01	2	1	1	5.000.000
02	3	2	1	2.500.000
03	4	3	1	1.666.666
04	5	4	1	1.250.000
05	6	5	1	1.000.000
06	7	6	1	833.333
07	8	7	1	714.286
08	9	8	1	625.000
09	10	9	1	555.555
ID	Número de Máquinas	Número de Filtros 1	Número de Filtros 2	Número de Registros
01	9	7	2	714.286
02	10	8	2	625.000
03	11	9	2	555.555

Tabela 7.3: Distribuição de Máquinas e filtros - Kmeans

mesmo. Assim a distribuição e o aumento dos filtros nas execuções dos experimento será feita obedecendo essa ordem de demanda.

ID	Número de Máquinas	Número de Filtros 1	Número de Filtros 2	Número de Registros
01	2	1	1	316.212
02	3	2	1	158.108
03	4	3	1	105.405
04	5	4	1	79.054
05	6	5	1	63.243
06	7	6	1	52.703
07	8	7	1	45.174
08	9	8	1	39.527
09	10	9	1	35.135
ID	Número de Máquinas	Número de Filtros 1	Número de Filtros 2	Número de Registros
10	6	4	2	79.054
11	7	5	2	63.243
12	8	6	2	52.703
13	9	7	2	45.174
14	10	8	2	39.527
15	11	9	2	35.135

Tabela 7.4: Distribuição de Máquinas e filtros - ID3

A Tabela 7.4 apresenta a organização dos filtros do ID3 nos experimentos. O terceiro filtro sempre terá uma instância e utilizará uma das máquinas que contem o segundo filtro. A demanda de processamento do segundo filtro é maior do que nos demais casos. Serão executados 6 experimentos com duas instâncias do segundo filtro.

7.2 Resultados

Nessa seção são apresentados os resultados da execução dos experimentos descritos na seção anterior e a análise dos mesmos.

7.2.1 *Overhead* dos Filtros

A primeira comparação a ser feita sobre os filtros gerados é o *overhead* dos controles e código adicional inseridos pela ferramenta em relação ao algoritmo seqüencial. Para cada um dos algoritmos foram feitas execuções das versões seqüenciais dos algoritmos e medidos os tempos totais de execução. Para as mesmas estradas e nas mesmas máquinas foram executados os algoritmos com o código dos filtros gerados. Foram instanciados um filtro de cada nível em uma mesma máquina.

Algoritmo	Tempo Execução Seqüencial (Segundos)	Tempo Execução Filtros (Segundos)	Aumento Tempo Execução
Itemsets Freqüentes	308	331	6,9%
Kmeans	390	423	7,5%
ID3	276	592	114,0%

Tabela 7.5: Dados utilizados

Como pode ser visto na Tabela 7.5 a implementação em filtros aumenta o tempo de execução dos algoritmos. Isto se deve principalmente às trocas de mensagens realizadas entre os filtros. No caso do algoritmo de Itemsets Freqüentes o aumento é pouco significativo, cerca de 7%. Para o Kmeans, que realiza mais trocas de mensagens esse valor já é um pouco maior, 7,5% de aumento no tempo de execução.

O custo adicional para o ID3 foi muito alto, mais que o dobro. Isso se deve principalmente a dois fatores: as mensagens trocadas são estruturas maiores e existem três níveis de filtros, ou seja, há mais comunicação de dados entre os filtros.

7.2.2 *Speedup* e *Scaleup*

O *Speedup* é calculado com a divisão do tempo de execução seqüencial pelo tempo de execução com **N** processadores. No caso ideal o resultado é igual ao número de processadores, onde a escalabilidade é linear em relação ao aumento de processadores, ou seja não existe custo adicional pelo paralelismo.

Para o *Scaleup* é acrescido proporcionalmente o número de processadores e o número de registros. No caso ideal, o resultado obtido deve ser constante. Dado uma quantidade de máquinas e uma quantidade de dados a serem processados, se

forem dobrados os dados e dobrado a quantidade de máquinas, o tempo de execução deveria ser o mesmo.

7.2.2.1 Itemsets Freqüentes

A Tabela 7.6 apresenta os tempos medidos dos experimentos de *Speedup* com os filtros do algoritmo de Itemsets Freqüentes.

ID	Filtros 1	Filtros 2	Tempo (Segundos)		ID	Filtros 1	Filtros 2	Tempo (Segundos)
1	1	1	331		10	7	2	49
2	2	1	269		11	8	2	45
3	3	1	136		12	9	2	42
4	4	1	93					
5	5	1	76					
6	6	1	64					
7	7	1	49					
8	8	1	51					
9	9	1	46					

Tabela 7.6: Tempos Itemset Freqüentes

As primeiras colunas apresentam os tempos para a execução com apenas um filtro de agregação (nível dois). Nas outras colunas estão os tempos de execução com dois filtros de agregação.

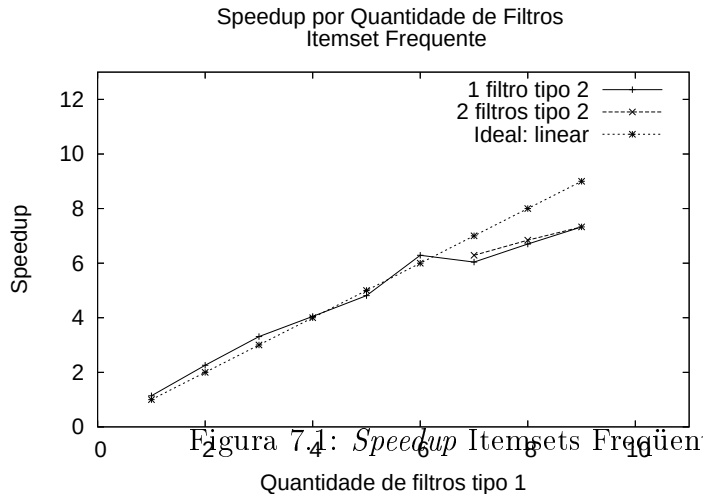

Figura 7.1: *Speedup* Itemsets Freqüentes

O gráfico das Figura 7.1 mostra os *speedups* em comparação ao caso ideal, o linear. Como pode ser visto a escalabilidade é próxima da ideal. Quando o tamanho da base fica menor a curva se afasta um pouco, mas a derivada ainda é positiva.

No gráfico foram usadas como escala o número de instâncias de filtro do nível um, sendo que cada instância se encontra em um processador.

A execução com 2 filtros de nível dois só foi feita a partir de 7 instâncias do filtro um. Isso foi feito para avaliar se o afastamento da curva ideal não estava sendo causado por contenção no segundo filtro. Entretanto como é mostrado no gráfico não ouve nenhum ganho significativo que justifica-se a utilização de mais um processador para o segundo filtro.

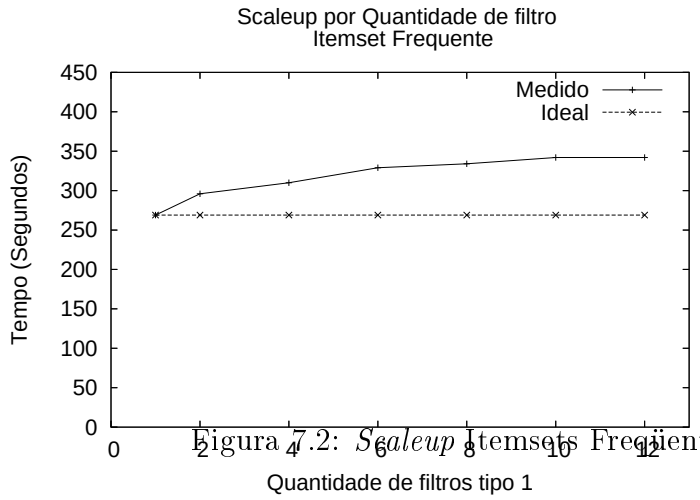

Figura 7.2: *Scaleup* Itemsets Frequentes

No *scaleup* apresentado no gráfico da Figura 7.2 é possível observar que a escalabilidade se mantém para grandes bases de dados. Nesse caso o número de processadores foi acrescido de 2 em 2 e foram utilizados até 12 instâncias do filtro de nível um. Para esse caso a medida que mais filtros leitores são acrescidos a base inicial e replicada e não dividida entre os mesmos.

Pelo gráfico é identificado que existe um aumento do tempo de execução com o aumento da entrada, mas para uma entrada 12 vezes maior o esse aumento foi de apenas 27%. Esse aumento se deve em parte pelo fato que apesar do aumento da entrada foi mantido o valor mínimo de suporte e provavelmente mais item foram computados.

7.2.2.2 Kmeans

A Tabela 7.7 apresenta os tempos medidos dos experimentos de *Speedup* com os filtros do algoritmo Kmeans.

Pelos valores o *Speedup* para esse algoritmo foi muito abaixo do esperado. Na da Figura 7.3 é mostrado o gráfico comparativo entre os valores medidos e o valor

ID	Filtros 1	Filtros 2	Tempo (Segundos)		ID	Filtros 1	Filtros 2	Tempo (Segundos)
01	1	1	464		10	9	2	167
02	2	1	274		11	10	2	156
03	3	1	217		12	11	2	152
04	4	1	183					
05	5	1	169					
06	6	1	158					
07	7	1	172					
08	8	1	152					
09	9	1	152					

Tabela 7.7: Tempos Kmeans

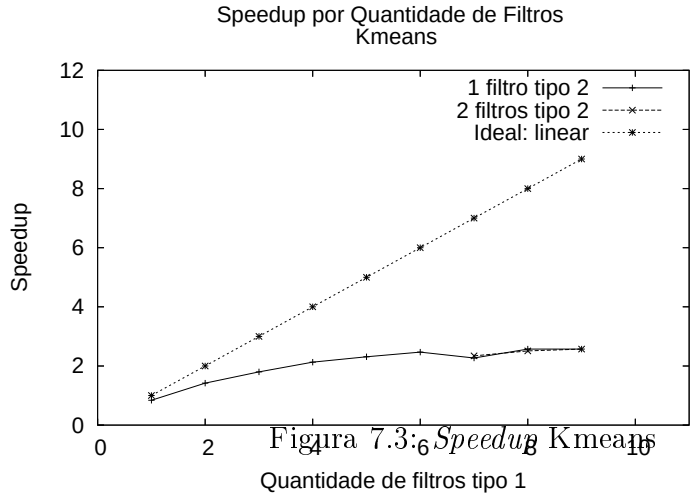

Figura 7.3: *Speedup* Kmeans

ideal esperado. Com a divisão da base em vários processadores a escalabilidade não apresentou ganhos que justificassem o uso de mais processadores. Pelo comportamento do algoritmo foi levantada a hipótese de que com a redução da base de dados a carga para cada filtro ficou muito pequena, não apresentando ganhos.

Novamente foi observado que o aumento de instâncias do segundo filtro não apresentaram ganho de desempenho. Não houve contenção pelo segundo filtro durante a execução, os gráficos praticamente coincidem nos pontos.

A hipótese levantada, no experimento de *speedup*, de que a base ao ser dividida ficou gerou pouca demanda de processamento pode ser comprovada pelo resultado do *Scaleup* para o algoritmo Kmeans. O gráfico da Figura 7.4 mostra que para bases maiores o algoritmo teve uma escalabilidade melhor do que a observada no Itemsets Freqüentes. Pelo gráfico pode ser observado que o tempo execução está se mantendo com o aumento da base e do número de processadores.

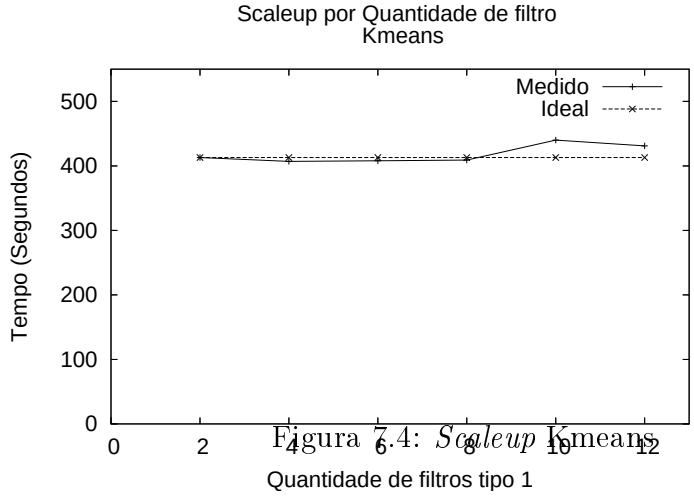

Figura 7.4: *Scaleup* Kmeans

7.2.2.3 ID3

A Tabela 7.8 apresenta os tempos medidos dos experimentos de *Speedup* com os filtros do algoritmo ID3.

ID	Filtros 1	Filtros 2	Tempo (Segundos)		ID	Filtros 1	Filtros 2	Tempo (Segundos)
01	1	1	679		10	6	2	579
02	2	1	394		11	7	2	677
03	3	1	453		12	8	2	706
04	4	1	610		13	9	2	925
05	5	1	1011		14	10	2	967
06	6	1	1491		15	11	2	1036
07	7	1	1833					
08	8	1	2133					
09	9	1	2207					

Tabela 7.8: Tempos ID3

Os valores de *Speedup* para esse algoritmo foi muito abaixo do esperado. Na da Figura 7.5 é mostrado o gráfico comparativo entre os valores medidos e o valor ideal.

Como pode ser observado o algoritmo teve uma escalabilidade muito ruim. Com esse desempenho o paralelismo piorou a execução do algoritmo. O *scaleup* apresentado no gráfico da Figura 7.6 também apresentou um resultado in-satisfatório, pois teve um aumento de mais de 350% no tempo de execução.

Foi feita uma avaliação mais detalhada para esse caso, para entender esse desempenho tão ruim, diferentemente dos dois outros algoritmos. Os primeiros fatores já apresentados para a explicação do *overhead* da execução em filtros não deveriam influenciar nas execuções com mais processadores.

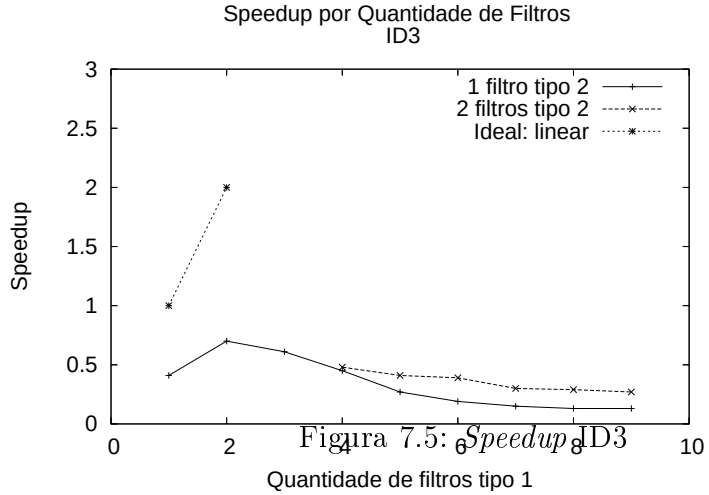

Figura 7.5: *Speedup* ID3

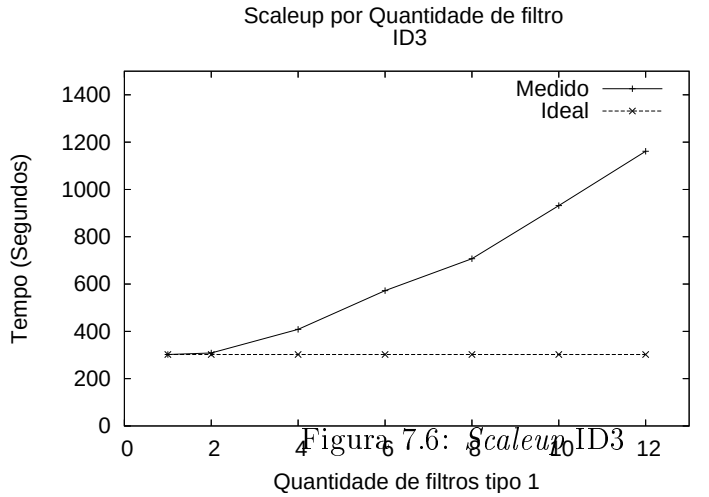

Figura 7.6: *Scaleup* ID3

O acréscimo de mais um filtro do segundo nível apresentou melhores resultados, como pode ser visto na Tabela 7.8. Entretanto a escalabilidade ainda se manteve insatisfatória.

A versão seqüencial do ID3 calcula o ganho de informação para cada propriedade de cada vértice. Na implementação feita, os vértices da árvore são processados por níveis, um a um. A cada novo vértice, seus filhos são inseridos em um fila para serem tratados.

Para a versão paralela, foi mantida essa mesma ordem na execução. Um novo vértice somente tem seus itens contados e o ganho de informação das propriedades calculado após a conclusão do anterior.

Como a cada vértice inserido na árvore de decisão a base é particionada, cada

novo filho pode seguir sua execução sem aguardar qualquer resultado de outro vértice do seu nível na árvore ou de outros vértices de níveis inferiores, filhos dos demais vértices.

O código gerado automaticamente pela ferramenta utiliza funções de envio e recebimento de mensagens blocantes. Para algoritmos com essa característica deve ser gerado um código com outro formato de filtros, onde as trocas de mensagens não sejam blocantes. Isso corresponde a dizer que o paralelismo a nível de assincronia deve ser explorado nesse caso.

O resultado obtido se deve ao fato de que quanto a quantidade de vértices aumenta, ou seja, nos níveis mais baixos da árvore, os filtros ficam muito tempo ociosos. As sub-bases, geradas pelo particionamento natural do algoritmo, ficam pequenas e o tempo de tráfego das mensagens é maior do que o tempo de processamento dos filtros.

O compilador atual não trata esse tipo de caso. É proposto como trabalho futuro a utilização de mensagens não blocantes em casos de algoritmos como o do ID3.

7.2.3 Código Manual x Automático

Nessa seção foi feita a comparação dos resultados obtidos da experimentação do código gerado de forma automática com algoritmo implementado manualmente. No trabalho [FJGD05], que apresenta o *Anthill*, os autores implementaram três algoritmos de Mineração de Dados e avaliaram a eficiência dos mesmos. Foram utilizadas as mesmas métricas, *speedup* e *scaleup*, no trabalho de apresentação do *Anthill* e nesse trabalho.

A comparação busca apresentar o quão eficiente foi o resultado da geração automática em relação a um processo manual. Um fato importante a ser destacado é que no caso das implementações manuais os algoritmos passaram por um processo de otimização do código. No caso da geração automática o código dos filtros é a versão seqüencial acrescida de código adicional para controle e comunicação. É um dos trabalhos futuros a esse a otimização do código gerado automaticamente.

Não foi possível obter informação sobre o custo de homens/hora para o desenvolvimento dos algoritmos em filtros *Anthill*. Todavia, é sabido que escrita de código seqüencial é mais simples que a implementação de filtros *Anthill*. A análise de custos para os dois processos também é proposta como trabalho futuro.

7.2.3.1 Algoritmo Itemsets Freqüentes

No Artigo [FJGD05] foi implementado o algoritmo de geração de regras de associação, o Apriori. Nesse trabalho foi utilizados o algoritmo de calculo de Itemsets Freqüentes, que é a base do Apriori. Não pode ser feita uma comparação direta entre os dois casos. Entretanto, os resultados de *speedup* e *scaleup* obtidos para a execução do código gerado automaticamente foram satisfatórios. A implementação do Apriori seqüencial e a aplicação do processo de geração de filtros automático é proposto como trabalho futuro.

7.2.3.2 Algoritmo Kmeans

Os resultados de *scaleup* obtidos na experimentação da versão gerada automaticamente foram tão bons quanto a implementação manual. No artigo é feita uma normalização do valor e é dito que o mesmo esteve constante em uma unidade. Fazendo a normalização do Gráfico 7.4 apresentado esse valor está variando entre 0,99 e 1,07 .

O *speedup* na versão manual também obteve bom resultados. A versão seqüencial implementada da qual foram gerados os filtros não foi feita com foco na sua otimização. Acredita-se que um refinamento do código seqüencial, e a utilização de uma base de dados com mais dimensões irá gerar filtros *Anthill* de forma automática com o mesmo valor de *speedup* semelhante a versão manual.

7.2.3.3 ID3

No Artigo [FJGD05] foram apresentados resultados de *speedup* muito bons. A escalabilidade foi próxima da ideal, ou seja, linear. Em um dos resultados foi obtida uma escalabilidade super linear. No Artigo também foi feita uma avaliação do tempos de processamento e de troca de mensagens entre os filtros.

Ao analisar o código gerado manualmente foi possível identificar que as trocas de mensagens não são blocantes. Vários vértices tem seus itens contados e seus ganhos de informação calculados simultaneamente. A assincronia do algoritmo é fortemente explorada nessa implementação.

Baseado nos resultados obtidos da implementações manuais e do resultado obtido pela geração automática dos filtros foi comprovada a causa da não escalabilidade do código desse trabalho. Para essa aplicação é essencial a utilização de outro tipo de funções para trocas de mensagens. É proposto como trabalho futuro essa expansão no gerador de código automático.

7.3 Sumário

Nesse capítulo foi feita a avaliação experimental dos resultados da geração automática de código. Foi avaliado o *speedup* e o *scaleup* para três algoritmos. Os dois primeiros algoritmos apresentaram bom resultados em termos de escalabilidade. Para o terceiro caso o resultado gerado foi ineficiente.

Capítulo 8

Conclusão e Trabalhos Futuros

O *Anthill* é uma solução eficiente para a execução de algoritmos intensos de forma distribuída e paralela. Os algoritmos de Mineração de dados estão sendo cada dia mais usados e eles tem forte demanda por poder de processamento. O resultado obtido nesse trabalho vem facilitar uma maior exploração do *Anthill* por uma classe de algoritmos como os de Mineração de Dados.

Nesse trabalho foi feito o desenvolvimento de um processo e uma ferramenta para a geração automática de filtros para o *Anthill* a partir de um algoritmo seqüencial. A automatização da geração do código foi dividida em duas etapas: identificação dos filtros, ou seja, o particionamento do código e a geração do código dos filtros. O foco desse trabalho foi na segunda etapa, sendo a mesma feita de forma automática pela ferramenta gerada. A primeira etapa teve um processo definido, mas a realização foi manual, pois estava fora do escopo do trabalho.

O processo de identificação dos filtros e a anotação do código seqüencial foi aplicada a três algoritmos de Mineração de Dados com sucesso. Foram gerados os filtros *Anthill* para o Itemsets Freqüentes, Kmeans e ID3 a partir da implementação seqüencial dos mesmos. O código gerado foi avaliado quanto a sua escalabilidade, mostrando resultados satisfatórios. No caso do algoritmo ID3 deve ser feita uma avaliação com mensagens não blocantes.

Esse trabalho tem como contribuição, além do processo e ferramenta gerada, o primeiro passo em uma pesquisa que se mostrou muito promissora. Os resultados obtidos levam a acreditar de que mais esforços podem ser invertidos nesse processo e que poderão ter bons resultados.

8.1 Trabalhos Futuros

A geração dos código dos filtros foi baseada nas anotações inseridas no código seqüencial dos algoritmos. A automatização desse passo é o primeiro passo a ser dado como trabalho futuro. A identificação das principais variáveis dos algoritmos e a dependência de dados entre as mesmas se mostrou um caminho promissor para a identificação dos filtros. Também podem ser exploradas outras abordagens para a identificação dos filtros.

A avaliação mais detalhada das características dos algoritmos também pode ser útil para a geração de códigos de filtros mais eficientes. Uma forma mais genérica para a troca de mensagens entre os filtros pode aceitar uma estrutura na transferência de dados entre os mesmos. A partir da ferramenta atual podem ser feita uma extensão nas anotações utilizadas incluindo mais detalhes e informações para a geração do código dos filtros *Anthill*. Essas são apenas algumas das várias otimizações que podem ser feitas no resultado do gerador automático de código dos filtros.

Outras classes de algoritmos podem ser avaliadas com a ferramenta. Dado a existência do Projeto Tamanduá foi explorado nesse trabalho algoritmos de Mineração de Dados. A utilização da ferramenta para outras aplicações pode ser feita com outros tipos de algoritmos. Uma maior generalização do formato padrão dos algoritmos definido nesse trabalho pode aumentar o número de classes de algoritmos sobre os quais essa aplicação pode atuar.

Informações para o escalonamento dos filtros de maneira mais eficiente podem ser obtidas a partir do processo de identificação de dependência de dados. Baseado na dependências identificadas pode-se obter de forma automática os limites de instâncias para cada filtro, bem como uma distribuição otimizada de filtros para o processamento. O caminho contrário também pode se percorrido. Informações geradas pelo escalonador podem ser utilizadas para a geração do código dos filtros. O escalonador traria dados da execução do algoritmo em filtros, de forma a gerar uma nova distribuição do algoritmo em filtros, mais eficiente.

Apêndice A

ItemSet Frequente Serial

Código do algoritmo de Itemsets frequentes versão serial comentado.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/time.h>
#include <sys/resource.h>
#define   NUITENS  1000
#define   NUTUPLAS 200000
#define  TAMLINHA 1000000
typedef struct Item
{
  int     ativo;
  int     id_item;
  char    label[5];
  int     suporte;
  int     card;
  int     dep[NUITENS];
} t_Item;

int main (int argc, char *argv[]) {
  FILE    *Arq;
  char    *linha;
  int     suporte;
  int     nu_tupla;
  t_Item  *itemsets;
  int     nu_itemsets;
  int     i, j ,l , k;
  int     n, ini, iniTam;
  int     teste1, teste2;
  int     exceto, c;
```

```
int     sub1, sub2;
char    str_tmp[200];
int     **lista_inv;
int     *lista_candidato;
int     *lista_frequente;
int     TamMenor = 1;
int     blocos;
char   nome_arquivo[100];
if(argc != 3){
    printf("Executar:  ./a.out <nome arquivo>  <suporte> \n");
    exit(0);
}
sprintf(nome_arquivo, "%s", argv[1]);
sscanf(argv[2], "%d", &suporte) ;
itemsets = malloc( sizeof(t_Item) * NUITENS );
linha = malloc( sizeof(char) * TAMLINHA );
lista_inv = malloc( sizeof(int) * NUITENS );
for (i=0 ; i<NUITENS ; i++) {
    lista_inv[i] = malloc( sizeof(int) * NUTUPLAS );
}
lista_candidato = (int*) malloc( sizeof(int) * NUITENS );
lista_frequente = (int*) malloc( sizeof(int) * NUITENS );
nu_itemsets = 0;
nu_tupla  = 0;
for (i=0 ; i<NUITENS ; i++) {
  lista_candidato[i] = 0;
  lista_frequente[i] = 0;
  for (j=0 ; j<NUITENS ; j++) {
    itemsets[i].dep[j] = 0;
  }
  for (j=0 ; j<NUTUPLAS ; j++) {
    lista_inv[i][j] = 0;
  }
}
/* -----  Leitura de Dados - Inicio  */
if ( (Arq = fopen(nome_arquivo,"r+") ) )
{
  fgets(linha, (TAMLINHA-1), Arq);

     (...)
        while ( !(feof(Arq)) ) {
          linha[0] = '\0';
          fgets(linha, (TAMLINHA-1), Arq);
    (...)

       (...)
```

```
                for (j=0 ; j<nu_itemsets ; j++) {
                  if ( (itemsets[j].ativo==1) && (itemsets[j].card==TamMenor) ) {
                    if ( strcmp(itemsets[j].label,str_tmp ) == 0 ) {
                      lista_inv[j][nu_tupla] = 1;
                    }
                  }
       (...)

     }
     nu_tupla ++;
   }
}
printf ("\n -- Tuplas Lidas: %d  ", nu_tupla);
fclose(Arq);
/* -----  Leitura de Dados - Fim  */
for (i=0 ; i<nu_itemsets ; i++)
{
  if ( lista_candidato[i] ) {
    for (j=0 ; j<nu_tupla ; j++) {
      if ( lista_inv[i][j] ) {
          itemsets[i].suporte ++;
      }
    }
    if ( ! ( itemsets[i].suporte < suporte ) ){
        lista_frequente[i] = 1;
    }
  }
  if (lista_frequente[i]) {
    for (n=0 ; n<nu_itemsets ; n++) {
      if (lista_candidato[n]==0) {
        k = 1;
        sub1 = -1;
        sub2 = -1;
        for (j=0 ; j<nu_itemsets ; j++) {
          if ( itemsets[n].dep[j]==1 ) {
            if ( itemsets[j].card == (itemsets[n].card-TamMenor) ) {
              if (sub1 == -1) {
                sub1 = j;
              } else if (sub2 == -1) {
                sub2 = j;
              }
            }
            if (lista_frequente[j]==0) {
              k = 0;
              lista_frequente[n]=0;
```

```
              j=nu_itemsets+1;
            }
          }
        }
        if (k) {
          for (j=0 ; j<nu_tupla ; j++) {
            if ( lista_inv[sub1][j] && lista_inv[sub2][j] ) {
              lista_inv[n][j] = 1;
            }
          }
          lista_candidato[n] = 1;
        }
      }
    }
  }
  printf ("\n --- Resultado ----------- ");
  for (i=0 ; i<nu_itemsets ; i++)
  {
    if ( lista_frequente[i]) {
      printf ("\n - Item:%s - sup(%d)",itemsets[i].label,itemsets[i].suporte);
    }
  }
  return 0;
}
```

Apêndice B

ItemSet Frequente Filtro 01

Código do filtro 1 do algoritmo de Itemsets frequentes gerado automáticamante.

```
// ----- Código gerado automáticamante             -----
#include "FilterDev.h"
#include "estrutura-filtro.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define   NUITENS  1000
#define   NUTUPLAS 100000
#define   TAMLINHA 1000000
typedef struct Item
{
  int     ativo;          // Se item ativo
  int     id_item;        // id
  char    label[5];
  int     suporte;        // id
  int     card;           // Quantos sub itens
  int     dep[NUITENS];   // mapa de dependencias
} t_Item;
FILE    *Arq;
char    *linha;
int     suporte;
int     nu_tupla;
t_Item  *itemsets;
int     nu_itemsets;
int     i, j ,l , k;
int     n, ini, iniTam;
int     teste1, teste2;
int     exceto, c;
int     sub1, sub2;
char    str_tmp[200];
```

```
int     **lista_inv;
int     *lista_candidato;
int     *lista_frequente;
int     TamMenor = 1;
int     blocos;
char   nome_arquivo[100];
OutputPortHandler saida_f1;
InputPortHandler  entrada_f1;
Work     tmp_work;
int      filtro_id;
t_Msg    *msg_f1;
t_Msg    *msg_f2;
t_Msg    *msg_f3;
int      nu_filtros ;
int initFilter(void *work, int worksize){
  filtro_id = dsGetMyRank();
  nu_filtros = NUFILTROS_F1 ;
  msg_f1 = malloc( sizeof(t_Msg) ) ;
  msg_f2 = malloc( sizeof(t_Msg) ) ;
  tmp_work = (*(Work *) work);
  saida_f1  = dsGetOutputPortByName("saida-f1");
  entrada_f1  = dsGetInputPortByName("entrada-f1");
  suporte = tmp_work.parametro1 ;
  sprintf( nome_arquivo, "entrada/entrada.txt" ) ;
  itemsets = malloc( sizeof(t_Item) * NUITENS );
  linha = malloc( sizeof(char) * TAMLINHA );
  lista_inv = malloc( sizeof(int) * NUITENS );
  for (i=0 ; i<NUITENS ; i++) {
    lista_inv[i] = malloc( sizeof(int) * NUTUPLAS );
  }
  lista_candidato = (int*) malloc( sizeof(int) * NUITENS );
  lista_frequente = (int*) malloc( sizeof(int) * NUITENS );
  nu_itemsets = 0;
  nu_tupla  = 0;
  for (i=0 ; i<NUITENS ; i++) {
    lista_candidato[i] = 0;
    lista_frequente[i] = 0;
    for (j=0 ; j<NUITENS ; j++) {
      itemsets[i].dep[j] = 0;
    }
    for (j=0 ; j<NUTUPLAS ; j++) {
      lista_inv[i][j] = 0;
    }
  }
  return 1;
```

```
}

int processFilter(void *work, int worksize) {
  int id, valor;
  /* -----  Leitura de Dados - Inicio  */
  if ( (Arq = fopen(nome_arquivo,"r+") ) )
  {
    fgets(linha, (TAMLINHA-1), Arq);

      nu_tupla ++;
    }
  }
  fclose(Arq);

  /* -----  Leitura de Dados - Inicio  */
  for (i=0 ; i<nu_itemsets ; i++)
  {
    if ( lista_candidato[i] ) {
      for (j=0 ; j<nu_tupla ; j++) {
        if ( lista_inv[i][j] ) {
          itemsets[i].suporte ++;
        }
      }
      msg_f1->id_item = i;
      msg_f1->valor = itemsets[i].suporte ;
      dsWriteBuffer(saida_f1, (void*)(msg_f1), ( sizeof(t_Msg) ) );

      if ( (dsReadBuffer(entrada_f1, (msg_f2), ( sizeof(t_Msg) ) )) == EOW) {
        return 1;
      } else  {
        id = msg_f2->id_item;
        valor = msg_f2->valor;
      }
      lista_frequente[i] = valor ;

      if ( (dsReadBuffer(entrada_f1, (msg_f2), ( sizeof(t_Msg) ) )) == EOW) {
        return 1;
      } else  {
        id = msg_f2->id_item;
        valor = msg_f2->valor;
      }
      i = id ;
      itemsets[i].suporte = valor ;

      if (lista_frequente[i]) {
```

```
        for (n=0 ; n<nu_itemsets ; n++) {
          if (lista_candidato[n]==0) {
            k = 1;
            sub1 = -1;
            sub2 = -1;
            for (j=0 ; j<nu_itemsets ; j++) {
              if ( itemsets[n].dep[j]==1 ) {
                if ( itemsets[j].card == (itemsets[n].card-TamMenor) ) {
                  if (sub1 == -1) {
                    sub1 = j;
                  } else if (sub2 == -1) {
                    sub2 = j;
                  }
                }
                if (lista_frequente[j]==0) {
                  k = 0;
                  lista_frequente[n]=0;
                  j=nu_itemsets+1;
                }
              }
            }
            if (k) {
              for (j=0 ; j<nu_tupla ; j++) {
                if ( lista_inv[sub1][j] && lista_inv[sub2][j] ) {
                  lista_inv[n][j] = 1;
                }
              }
              lista_candidato[n] = 1;
            }
          }
        }
      }
    }
  }
  return 1;
}

int finalizeFilter(void) {
  if (filtro_id == 0 ) {
    for (i=0 ; i<nu_itemsets ; i++)
    {
      if ( lista_frequente[i]) {
        if (SAIDA) printf ("\n- Item:%s - sup(%d)",itemsets[i].label,itemsets[i].suporte);
      }
    }
```

```
  }
  return 1;
}

// --- Fim Filtro ----
```

Apêndice C

ItemSet Frequente Filtro 02

Código do filtro 2 do algoritmo de Itemsets frequentes gerado automáticamante.

```
// ----- Código gerado automáticamante pelo:       -----
#include "FilterDev.h"
#include "estrutura-filtro.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/time.h>
#include <sys/resource.h>
#define   NUITENS  1000
#define   NUTUPLAS 100000
#define   TAMLINHA 1000000

typedef struct Item
{
int     ativo;          // Se item ativo
int     id_item;        // id
char    label[5];
int     suporte;        // id
int     card;           // Quantos sub itens
int     dep[NUITENS];   // mapa de dependencias
} t_Item;

FILE    *Arq;
char    *linha;
int     suporte;
int     nu_tupla;
t_Item  *itemsets;
int     nu_itemsets;
int     i, j ,l , k;
```

```
int     n, ini, iniTam;
int     teste1, teste2;
int     exceto, c;
int     sub1, sub2;
char    str_tmp[200];
int     **lista_inv;
int     *lista_candidato;
int     *lista_frequente;
int     TamMenor = 1;
int     blocos;
char    nome_arquivo[100];

OutputPortHandler saida_f2;
InputPortHandler  entrada_f2;

Work     tmp_work;
int      filtro_id;
t_Msg    *msg_f1;
t_Msg    *msg_f2;
int      nu_filtros ;

int initFilter(void *work, int worksize){
  filtro_id = dsGetMyRank();
  nu_filtros = NUFILTROS_F1 ;
  msg_f1 = malloc( sizeof(t_Msg) ) ;
  msg_f2 = malloc( sizeof(t_Msg) ) ;
  tmp_work = (*(Work *) work);
  saida_f2  = dsGetOutputPortByName("saida-f2");
  entrada_f2  = dsGetInputPortByName("entrada-f2");
  suporte = tmp_work.parametro1 ;
  pegaTempo(&tempoSec, &tempoUsec);
  itemsets = malloc( sizeof(t_Item) * NUITENS );
  lista_frequente = (int*) malloc( sizeof(int) * NUITENS );

  return 1;
}

int processFilter(void *work, int worksize) {
  int id, valor;
  int lista_valores[3500];
  int lista_recebimento[3500];
  int x;
  for (x=0; x<3500 ; x++){
    lista_recebimento[x] = 0;
    lista_valores[x] = 0;
```

```
  }
  while(1) {
    if ( (dsReadBuffer(entrada_f2, (msg_f1), ( sizeof(t_Msg) ) )) == EOW) {
      return 1;
    } else  {
      id = msg_f1->id_item;
      valor = msg_f1->valor;
    }
    lista_valores[id] += valor;
    lista_recebimento[id] ++;

    i = id ;
    itemsets[i].suporte = lista_valores[id] ;
    if ( ! ( itemsets[i].suporte < suporte ) ){
      lista_frequente[i] = 1;
    }

    if ( lista_recebimento[id] == nu_filtros ) {
      msg_f2->id_item = i;
      msg_f2->valor = lista_frequente[i] ;
      dsWriteBuffer(saida_f2, (void*)(msg_f2), ( sizeof(t_Msg) ) );

      msg_f2->id_item = i;
      msg_f2->valor = itemsets[i].suporte ;
      dsWriteBuffer(saida_f2, (void*)(msg_f2), ( sizeof(t_Msg) ) );
    }
  }
  return 1;
}

int finalizeFilter(void) {
  return 1;
}

// --- Fim Filtro ----
```

Referências Bibliográficas

[ACBR03] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. OurGrid: An approach to easily assemble grids with equitable resource sharing. In *9th Workshop on Job Scheduling Strategies for Parallel Processing*, 2003.

[AFJS00] Gagan Agrawal, Renato Ferreira, Ruoming Jin, and Joel H. Saltz. High level programming methodologies for data intensive computations. In *In Proceedings of the Fifth Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, páginas 32–43, 2000.

[AFS00] G. Agrawal, R. Ferriera, and J. Saltz. Language extensions and compilation techniques for data intensive computations. *In Proceedings of Workshop on Compilers for Parallel Computing*, 2000.

[AJL01] Gagan Agrawal, Ruoming Jin, and Xiaogang Li. Compiler and middleware support for scalable data mining. *9th Workshop on Compilers for Parallel Computers*, 2001.

[ALE01] Vikram S. Adve, Vinh Vi Lam, and Brian Ensink. Language and compiler support for adaptive distributed applications. In *Languages, Compilers, and Tools for Embedded Systems - LCTES/OM*, páginas 238–246, 2001.

[ama] Amazon. http://www.amazon.com.

[BBG+93] Francois Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic Ideas for an object parallel language. *Scientific Programming*, 2(3), 1993.

[BFK+00] Michael Beynon, Renato Ferreira, Tahsin M. Kurc, Alan Sussman, and Joel H. Saltz. Datacutter: Middleware for filtering very large scientific datasets on archival storage systems. In *IEEE Symposium on Mass Storage Systems*, páginas 119–134, 2000.

[BG97] G. Bell and J. Gray. The revolution yet to happen. *In P.J. Denning and R. M. Metcalfe, editors, Beyond Calculation, páginas 5–32. Primavera Verlag*, 1997.

[CDC+99] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to upc and language specification. Technical report, 1099.

[Cha04] Stephen J. Chapman. *Fortran 90/95 for Scientists and Engineers*. McGraw-Hill Professional, 2004.

[Che04] Wei-Yu Chen. Building a source-to-source upc-to-c translator. Technical Report UCB/CSD-04-1369, EECS Department, University of California, Berkeley, 2004.

[CNHS01] P. Christen, O. Nielsen, M. Hegland, and P. Strazdins. Parallel data mining on a beowulf cluster. *High-Performance Computing (HPC) Asia*, 2001.

[com] Comprasnet. http://www.comprasnet.gov.br/.

[CSS98] Chialin Chang, Alan Sussman, and Joel Saltz. Infrastructure for building parallel database systems for multi-dimensional data. Technical Report CS-TR-3894, Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing, 1998.

[CV01] Massimo Coppola and Marco Vanneschi. High-performance data mining with skeleton-based structured parallel programming. Technical Report TR-01-06, 08 2001.

[dN03] Luiz Thomaz do Nascimento. Lpsched: Escolamento de aplicações de fluxo de dados em grids. In *Dissertação de Mestrado - Universidade Federal de Minas Gerais*, 2003.

[EGCD01] T. El-Ghazawi, W. Carlson, and J. Draper. Upc language specifications v, 2001.

[FAJS01] Renato Ferreira, Gagan Agrawal, Ruoming Jin, and Joel Saltz. Compiling data intensive applications with spatial coordinates. *Lecture Notes in Computer Science*, 2001.

[FAS00] Renato Ferreira, Gagan Agrawal, and Joel Saltz. Compiling object-oriented data intensive applications. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, páginas 11–21, New York, NY, USA, 2000. ACM Press.

[FJGD05] Renato Ferreira, Wagner Meira Jr., Dorgival Guedes, and Lucia Drumond. Anthill: A scalable run-time environment for data mining applications. In *SBAC-PAD'05: The 17th International Symposium on Computer Architecture and High Performance Computing*, Rio de Janeiro, Brazil, Outubro 2005.

[FMH+97] Renato Ferreira, Bongki Moon, Jim Humphries, Alan Sussman, Joel Saltz, Robert Miller, and Angelo Demarzo. The virtual microscope. In *American Medical Informatics Association, 1997 Annual Fall Symposium*, páginas 449–453, Nashville, TN, 1997.

[Fora] Fortran. http://pt.wikipedia.org/wiki/fortran.

[Forb] HPF High Performance Fortran. http://www.netlib.org/hpf/.

[FSA01] Renato Ferreira, Joel H. Saltz, and Gagan Agrawal. Compiler and runtime analysis for efficient communication in data intensive applications. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, páginas 231–242, Washington, DC, USA, 2001. IEEE Computer Society.

[GAK02] G. Goumas, M. Athanasaki, and N. Koziris. Automatic Code Generation for Executing Tiled Nested Loops Onto Parallel Architectures. In *Proceedings of the ACM Symposium on Applied Computing (SAC 2002)*, páginas 876–881, Madrid, Spain, Mar 2002.

[GGC+05] Luís Fabrício Góes, Pedro Guerra, Bruno Coutinho, Leonardo Rocha, Wagner Meira, Renato Ferreira, Dorgival Guedes, and Walfredo Cirne. AnthillSched: A scheduling strategy for irregular and iterative I/O-intensive parallel jobs. In Dror G. Feitelson, Eitan Frachtenberg, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, páginas 108–122. Primavera Verlag, 2005. Lect. Notes Comput. Sci. vol. 3834.

[goo] Google. http://www.google.com.

[gra] Graphviz. http://www.graphviz.org/.

[GRG+05] Ferreira G, Araujo R, Orair G, Gonsalves L, D O Guedes, Ferreira Renato Antônio Celso, Furtado V, and Meira JR Wagner. Paralelização eficiente de um algoritmo de agrupamento hierárquico. In *Anais do I Workshop sobre Algoritmos de Mineração de Dados, 2005, Uberlândia.*, 2005.

[GSFJ05] Luís F. W. Góes, Italo G. A. Stefani, Renato Ferreira, and Wagner Meira Jr. Mapeamento de programas i3 para aplicações anthill paralelas de fluxos de dados baseadas em filtros. *VI Workshop em Sistemas Computacionais de Alto Desempenho WSCAD'2005*, páginas 145–152, 2005. Rio de Janeiro - RJ - Brasil.

[HAM+95] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *ACM Press - New York, NY, USA*, 1995.

[HK01] J. Han and M. Kamber. *Data Mining - Concepts and Techniques.* Morgan Kaufmann Publishers, 2001.

[Hof04] J. Hofer. Distributed decision tree induction within the grid data mining framework gridminer-core. *Dissertação de Mestrado, Fakult at f ur Wirtschaftswissenschaften und Informatik, Universit at Wien*, 2004.

[hpf93] Hpf language specification, version 1.0. In *Contains the HPF 1.1 Language and the HPF 1.1 Approved Extensions*, 1993.

[hpf97] Hpf language specification, version 2.0. In *Contains the HPF 2.0 Language and the HPF 2.0 Approved Extensions*, 1997.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In Weidong Chen, Jeffrey Naughton, and Philip A. Bernstein, editors, *2000 ACM SIGMOD Intl. Conference on Management of Data*, páginas 1–12. ACM Press, 05 2000.

[JA01] R. Jin and G. Agrawal. A middleware for developing parallel data mining applications. *Proceedings of the First SIAM International Conference on Data Mining*, 2001.

[KLA+03] Tahsin Kurc, Feng Lee, Gagan Agrawal, Umit Catalyurek, Renato Ferreira, and Joel Saltz. Optimizing reduction computations in a distributed environment. 15th Supercomputing Conference, 2003.

[KN] Eleftherios Koutsofios and Stephen C. North. *Drawing graphs with dot.* Murray Hill, NJ.

[LJA] Xiaogang Li, Ruoming Jin, and Gagan Agrawal. Exploiting domain specific high-level runtime support for parallel code generation.

[Mac] PVM: Parallel Virtual Machine. http://www.csm.ornl.gov/pvm/pvm_home.html.

[nas] Nasa. http://www.nasa.gov.

[PSC93] Ravi Ponnusamy, Joel H. Saltz, and Alok N. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Supercomputing*, páginas 361–370, 1993.

[RD95] Martin Rinard and Pedro Diniz. Automatically parallelizing serial programs using commutativity analysis. Technical Report TRCS95-13, 6, 1995.

[Sar91] V. Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. *IBM J. Res. Dev.*, 35(5-6):779–804, 1991.

[SN95] Ambuj Shatdal and Jeffrey F. Naughton. Adaptive parallel aggregation algorithms. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, páginas 104–114, New York, NY, USA, 1995. ACM Press.

[Tam05] Projeto Tamanduá. Projeto tamanduá, 2005. http://tamandua.speed.dcc.ufmg.br.

[VMF+04] A. Veloso, W. Meira, R. Ferreira, D. Guedes, and S. Parthasarathy. In proceedings of asynchronous and anticipatory filter-stream based parallel algorithm for frequent itemset mining. 8th European Conference on Principles and Practice of Knowledge Discovery in Databases, 2004.

[VPP97] Purnell V., Corr P.H., and Milligan P. A novel approach to loop parallelization. In *In Proceedings of EUROMICRO 97. 'New Frontiers of Information Technology'. Short Contributions., Proceedings of the 23rd Euromicro Conference*, 1997.

[WD05] Gagan Agrawal Wei Du. Filter decomposition for supporting coarse-grained pipelined parallelism. *International Conference on Parallel Processing (ICPP'05)*, páginas 539–546, 2005.

[yah] Yahoo. http://www.yahoo.com.