Análise e Entendimento de Desempenho com a Ferramenta Antfarm

R. Pinho T. Teixeira Y. Faria G. Teodoro T. Tavares
D. Guedes R. Ferreira W. Meira Jr.
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
{rui,thiago,yurif,george,ttavares,dorgival,renato,meira}@dcc.ufmg.br

Resumo

O projeto e implementação de aplicações paralelizadas escaláveis e eficientes se mantem como um desafio, seja pela complexidade das aplicações e dos dados a serem analisados, seja pelas limitações dos paradigmas atuais de programação paralela para aplicações de alta demanda de processamento e comunicação como mineração de dados. O ambiente Anthill, baseado no paradigma filtro-fluxo identificado, tem se mostrado como um ambiente apropriado para paralelização dessas aplicações, permitindo explorar três estratégias de paralelização: dados, tarefas e assincronia. Entretanto, o entendimento e a depuração de desempenho dessas aplicações pode ser complexo, dada a diversidade de padrões de interação permitida pelo ambiente Anthill. A ferramenta Antfarm tem por objetivo não apenas apresentar o perfil de desempenho da aplicação mas identificar as interações entre os vários componentes de forma a justificar o desempenho observado. Neste artigo apresentamos as técnicas usadas na ferramenta Antfarm e um protótipo de implementação, que foi utilizado para analisar e entender o desempenho paralelo de uma aplicação popular de mineração de dados: a construção de árvores de decisão baseada em ganho de informação. Como será mostrado no artigo, a ferramenta foi capaz de detectar as peculiaridades da paralelização e explicar o seu desempenho. Em particular, fomos capazes de mensurar a assincronia entre os vários processos cooperando na solução de um problema e explicar o seu tempo de execução, incluindo interações entre as execuções das várias partes do algoritmo.

1. Introdução

O projeto e implementação de aplicações paralelas eficientes e escaláveis têm sido um desafio de pesquisa constante na área de Ciência da Computação. Diversas arquiteturas paralelas e paradigmas de programação têm sido propostos e utilizados em várias áreas de aplicação, onde fatores como a complexidade das aplicações e sua regularidade têm sido variáveis fundamentais no sucesso das paralelizações. A atividade de depuração de desempenho é intensa em todas essas situações e a construção de ferramentas de desempenho que permitam aos programadores analisar e entender o desempenho das aplicações também tem sido alvo de intensa pesquisa. Neste artigo apresentamos um conjunto de técnicas automatizadas de análise e entendimento de desempenho de programas paralelos, assim como mostramos a sua utilização para calibração de um programa paralelo real.

São bastante conhecidas as causas de degradação de desempenho em programas paralelos tais como desbalanceamento de carga, paralelismo insuficiente, computação adicional associada à paralelização (*overhead*), comunicação em excesso e sincronização em excesso, entre outras. É interessante notar que, via de regra, esses problemas podem ser agrupados em apenas dois grupos: problemas que causam ociosidade nos processadores e problemas que resultam em custos adicionais de processamento, armazenamento e tempo. O objetivo primordial de uma ferramenta de depuração de desempenho é auxiliar o programador a identificar as fontes de degradação de desempenho e, se possível, as estratégias de minimizá-las. Há um grande número de ferramentas de desempenho que auxiliam na identificação das fontes de degradação de desempenho, mas um número bem menor de ferramentas que permitem entender as origens dessa degradação. Podemos identificar duas causas para essa dificuldade no diagnóstico: modelos de programação semanticamente pobres e aplicações irregulares. Modelos de programação semanticamente pobres dão maior flexibilidade ao programador mas ao mesmo tempo dificultam a depuração de desempenho automatizada, uma vez que é muito complicado identificar de forma automatizada as oportunidades de paralelização e portanto o quão bem elas foram exploradas. A segunda fonte são aplicações irregulares, isto é, aplicações onde a quantidade de trabalho a ser realizada depende da natureza dos dados de entrada da aplicação, tornando esquemas de modelagem tradicio-

nais baseados em tamanho dos dados de entrada inviáveis.

Neste artigo apresentamos a ferramenta Antfarm. Essa ferramenta tem por objetivo facilitar a análise e entendimento de programas paralelos baseados no ambiente Anthill. Como será discutido na próxima seção, o ambiente Anthill permite a construção de aplicações paralelas escaláveis e eficientes por permitir a exploração de três tipos de paralelismo: dados, tarefa e assincronia (maximizando a sobreposição entre comunicação e computação). O paradigma de programação do Anthill permite a exploração de todo o potencial de paralelismo das aplicações, evitando, por exemplo, situações de sincronização em excesso. Por outro lado, toda essa flexibilidade torna a tarefa de análise e entendimento de desempenho mais complexa, exigindo que uma ferramenta de depuração de desempenho empregue técnicas especializadas, que explorem as informações providas pelo modelo de programação e sua instanciação.

Este artigo está organizado em seis seções, sendo esta a introdução. Na próxima seção apresentamos o ambiente de programação Anthill seguido da descrição da ferramenta Antfarm. A Seção 4 apresenta um estudo de caso de aplicação da ferramenta para entendimento do desempenho de um algoritmo de mineração de dados. As duas últimas seções apresentam trabalhos correlatos e as conclusões e trabalhos futuros.

2. Paralelização de Aplicações no Anthill

O Anthill (Formigueiro) é nosso ambiente para desenvolvimento e execução de aplicações distribuídas escaláveis. O ambiente foi desenvolvido tendo em mente aplicações paralelas não regulares, intensivas em processamento e em entrada-e-saída de dados (E/S). Nessas aplicações, que manipulam grandes volumes de dados, estes se encontram distribuídos em várias máquinas do sistema. Mover os dados para outros nós para então serem processados é freqüentemente uma operação ineficiente. Isso é verdade especialmente porque à medida que o processamento avança, os dados resultantes tendem a ser muitas vezes menores que os dados de entrada. Dessa forma, uma alternativa interessante é levar a computação aonde o dado está, reduzindo a comunicação através da rede. O sucesso desse enfoque depende da facilidade com que a aplicação possa ser dividida em etapas que sejam passíveis de execução em nós diferentes do sistema. Cada etapa dessa forma executará parte das transformações sobre os dados, iniciando com o conjunto de dados de entrada, até que se atinja o conjunto de dados de saída.

Essa discussão indica que uma boa paralelização de qualquer aplicação nesse contexto deve considerar simultaneamente tanto paralelismo de dados quanto de tarefas. A estratégia do Anthill aplica os dois enfoques, agregando uma terceira dimensão que nos permite explorar o grau de assincronia existente entre diferentes tarefas independentes no sistema ao longo do tempo. Os benefícios dessas três dimensões combinadas nos permitem atingir *speedups* elevados experimentalmente [12, 4].

Alguns dos conceitos implementados no *Anthill* são derivados do *Datacutter*, um sistema de execução de aplicações distribuídas baseado no modelo de programação *filter stream* [1, 3, 10, 2]. Nesse modelo, o processamento é dividido em tarefas que operam sobre os dados que fluem pelo sistema. Cada filtro implementa uma tarefa que transforma os dados segundo a necessidade da aplicação e se comunicam com outros filtros pelos canais de comunicação responsáveis pela transmissão contínua de dados (*streams* ou fluxos). Essas duas abstrações podem ser combinadas formando grafos arbitrários que representem o processamento da aplicação.

Usando esse modelo, criar uma aplicação no *Anthill* é um processo de decomposição do processamento em filtros. Nesse processo, a aplicação é modelada como uma computação no modelo *dataflow* dividida em uma rede de filtros que transformam os dados. Durante a execução, o processo definido para cada filtro é instanciado em diferentes nós do ambiente distribuído. A esses processos dá-se o nome de cópias transparentes ou instâncias de um filtro. Dessa forma, cada estágio do processamento pode ser distribuído por muitos nós de uma máquina paralela e os dados que devem fluir por aquele filtro podem ser particionados pelas cópias transparentes, produzindo o paralelismo de dados desejado.

Além disso, para muitas aplicações, a execução consiste em múltiplas iterações da mesma cadeia de filtros. A aplicação se inicia com um conjunto de soluções possíveis que passam pelos filtros, sendo melhoradas e em alguns casos criando novas soluções que devem ser processadas novamente. Pela nossa experiência, muitas aplicações interessantes seguem esse modelo. Uma característica interessante nesse caso é que há muitas oportunidades para execução assíncrona, já que diversas soluções podem estar sendo testadas independentemente ao longo da cadeia de filtros a cada momento.

Essa assincronia e a natureza iterativa das aplicações levaram ao desenvolvimento de duas extensões do modelo *filter stream* original implementadas no *Anthill*. Verificamos que muitas vezes as aplicações precisam compartilhar certo estado global sobre a evolução da computação, o que nos levou a definir um *broadcast stream*, que permite esse padrão de comunicação para todas as cópias de um filtro. Além disso, muitas vezes é preciso garantir certa localidade de processamento: dados que compartilham uma certa característica na semântica da aplicação podem ter que ser processados pela mesma cópia transparente. Isso ocorre sempre que há algum tipo de estado local associado com al-

gumas instâncias dos dados, ou quando há dependência de dados no processamento, como pode ocorrer em qualquer processo iterativo. Para esse fim, o *Anthill* fornece o fluxo identificado (*labeled stream*) que permite exatamente que a cópia de destino de cada instância dos dados seja determinada em função de alguma propriedade (um identificador) derivado do seu conteúdo.

Nosso modelo de programação, dessa forma, nos permite extrair o máximo de paralelismo das aplicações através das três possibilidades discutidas anteriormente: paralelismo de dados, de tarefas e assincronia. Já que as unidades de processamento são na verdade cópias de estágios de um *pipeline*, pode-se ter um paralelismo de grão fino. Como todo esse processamento pode ocorrer assincronamente, a execução não terá qualquer tipo de retenção (*bottleneck*) devido ao sistema. Para garantir a redução da latência durante o processamento, a granulosidade das tarefas deve ser definida pelo projetista da aplicação.

3. A ferramenta de desempenho Antfarm

Nesta seção apresentamos a ferramenta Antfarm, que tem por objetivo auxiliar no entendimento e depuração de desempenho de aplicações paralelizadas no modelo de programação filtro-fluxo identificado. Essa ferramenta permite aos usuários avaliar o desempenho das suas aplicações sob duas perspectivas. A primeira é a de filtros e fluxos. Esta avalia a interação entre os filtros e como os vários fluxos entre eles são utilizados em termos de comunicação. A segunda é a perspectiva de tarefas. Ela permite ao programador entender a dinâmica da aplicação, como e quando tarefas são criadas, assim como a interação entre diferentes tarefas. Nas próximas seções descrevemos em mais detalhe a ferramenta e as técnicas que ela integra.

3.1. Instrumentação

O ponto de partida da ferramenta é a instrumentação do ambiente Anthill. Aplicações instrumentadas geram um registro (*log*) de execução por filtro, contendo eventos tanto da biblioteca Anthill quanto da aplicação em si. São registrados, além da categoria do evento (que pode ser computação, comunicação, computação adicional ou ocioso) e da operação (quando pertinente), a localização de código origem do evento e a tarefa associada, além de outros dados específicos de eventos, como, por exemplo, processos origem ou destino de mensagens.

3.2. Análise de Desempenho Agregado

A ferramenta Antfarm permite a análise de desempenho utilizando medidas agregadas nas dimensões de filtros e tarefas. A Figura 2 mostra, na sua parte superior, a "Visão de Filtro", que é uma das metáforas visuais providas pela ferramenta. Neste caso, cada filtro é representado por um gráfico do tipo "pizza", onde se pode visualizar a quantidade de tempo associada a cada categoria de processamento. Os filtros que se comunicaram estão conectados por setas que também informam sobre o padrão de comunicação entre eles. Mais especificamente, a sua largura indica o número de mensagens trocadas e o tom de cinza o tamanho médio das mensagens. Outras metáforas (não mostradas por restrições de espaço) incluem a desagregação da Visão de Filtro para "Visão de Instância de Filtro", além da visão da representatividade de cada categoria por tarefa executada, o que pode ser desagregado por filtro, ou por operação dentro de cada categoria.

3.3. Análise de Desempenho Estrutural

A análise de desempenho estrutural avalia a adequação da estrutura de filtros e fluxos em termos de balanceamento e agilidade na execução das tarefas. Um aspecto fundamental do paradigma é minimizar ao máximo o tempo que um dado é retido em um filtro sem que gere algum resultado. Nesse caso, quanto menor a retenção, menor o nível de multiprogramação do filtro e melhor será o desempenho do sistema como um todo. Esse aspecto é medido pela análise de disparo, descrita a seguir.

Uma das bases da eficiência das aplicações paralelizadas no Anthill é a dinâmica que a aplicação pode assumir se corretamente balanceada e distribuída. Essa situação é evidenciada pelo índice de retenção de dados em cada um dos filtros. Quanto mais dados estiverem retidos nos filtros, pior. A análise de disparo avalia quanto tempo é necessário para que uma mensagem recebida por um filtro resulte em uma nova mensagem sendo enviada a outro filtro. Quanto mais rápido as mensagens forem enviadas, melhor. Altos tempos de disparo podem estar associados a desbalanceamento de carga, contenção e mesmo irregularidades das massas de dados. Um primeiro nível de análise compreende identificar as mensagens que apresentem longos tempos de disparo em um filtro. O segundo nível compreende explicar porque esses tempos de disparo são longos, seja pelo filtro estar lidando com muitas mensagens, ou por razões externas. Em geral, há uma mensagem originadora que vai resultar na mensagem que está sendo analisada. A toda mensagem há um tempo de disparo associado. O nosso objetivo, ao realizar essa análise, é entender o que tem que ser mudado para que a retenção seja a menor possível. Assim, construímos o grafo com todas as mensagens que chegam e a sua dependência entre elas. Esse grafo é percorrido identificando quem são as mensagens que explicam o tempo de disparo.

O nosso alvo são mensagens que têm um grande tempo de disparo, o que significa que há um grande intervalo de

tempo entre a recepção das mensagens necessárias para o envio da mensagem resultante que inicia as atividades associadas à tarefa no filtro subseqüente. Há duas causas fundamentais para grandes tempos de disparo. A primeira é contenção, ou seja, a existência de um grande número de tarefas sendo tratadas pela instância do filtro, o que aumenta o tempo de execução da tarefa. Neste caso, a análise do grafo mostra que o tempo entre o envio e a recepção de mensagens é grande e há várias outras tarefas sendo processadas entre a recepção de mensagens consecutivas associadas à mesma tarefa. O segundo caso é desbalanceamento de carga, quando as mensagens chegam à instância de filtro espaçadas no tempo tendo em vista terem sido enviadas espaçadas no tempo. Neste caso, o tempo entre envio e recepção de cada mensagem é baixo, a diversidade de tarefas é baixa e a explicação é baseada entre as diferenças entre as instâncias. A análise é baseada no grafo que contém todas as atividades dos filtros e mensagens trocadas entre eles. Para cada mensagem enviada, verificamos quais são as mensagens das quais elas dependem, qual o tempo de disparo e diagnosticamos se o tempo de disparo é motivado por contenção ou desbalanceamento de carga. Como a análise é feita no contexto de uma tarefa, identificamos as tarefas com maior tempo de disparo pelo nível de concorrência que a sua execução apresenta, ou seja, o número de instâncias de filtros que estão trabalhando na execução da tarefa durante o seu ciclo de vida. Na Figura 2 à direita e abaixo, o "Perfil de Concorrência"apresenta o referido perfil de uma tarefa com baixo tempo de disparo, parte do estudo de caso discutido na Seção 4. O gráfico de dispersão temporal da execução de uma tarefa (não mostrado por falta de espaço) nas várias instâncias é o ponto de partida como mecanismo de foco para identificar as tarefas cujo nível de concorrência deva ser avaliado.

3.4. Análise de Desempenho Dinâmico

A análise de desempenho dinâmico tem por objetivo identificar como as características dos dados influenciam o desempenho observado das aplicações. Neste contexto, aspectos como contenção, desbalanceamento de carga e atraso de sincronização são fundamentais. Como mencionado, uma tarefa é definida semanticamente dentro de um problema sendo resolvido no ambiente, como calcular a freqüência de um conjunto de entidades. A tarefa normalmente tem um ciclo de vida que se inicia com a sua criação, quando são definidas as suas dependências, ou seja, o conjunto de tarefas das quais a tarefa em questão depende. Uma vez que a tarefa tenha sido criada e todas as tarefas das quais ela depende concluídas, ela pode iniciar a execução no filtro que a criou. Não só o início, como a execução da tarefa em qualquer filtro depende de haver ou não disponibilidade de recursos de processamento, além de eventual atrasos de comunicação. Enquanto ativa em uma instância de filtro, a tarefa pode realizar o processamento propriamente dito da aplicação ou então despender tempo em chamadas do Anthill, que são quantificadas como custo adicional de computação.

Tarefa	Criada	Pronta	Suspensa
Computação Própria		Computação	
Outra Computação	Sinc	Contenção	Reciclada
Custos Adicionais		Custos Adicionais	
Comunicação		Comunicação	
Processador Ocioso	Ocioso	Ocioso	Desbal carga

Tabela 1. Perfis de Desempenho das Tarefas

Como mencionado na Seção 1, as fontes de degradação de desempenho em programas paralelos podem ser divididas em dois grupos: processador ocioso e computação adicional. No paradigma filtro-fluxo identificado a agilidade na execução das tarefas é fundamental para a eficiência das aplicações paralelas. A análise de desempenho dinâmico tem por objetivo avaliar o quanto a multiprogramação afeta o tempo de execução de uma tarefa. Entretanto, devemos diferenciar duas situações de ocorrência de multiprogramação. Se a instância de filtro processa várias tarefas, mas todas, à exceção de uma, estão suspensas, isso é benéfico, pois evita a ociosidade do processador. Se, por outro lado, a tarefa está pronta para executar e outra tarefa está sendo executada, então se caracteriza contenção. Podemos avaliar essas situações de forma sistemática como descrito a seguir. Cada tarefa pode estar em um de três estados em uma dada instância de filtro: (i) criada – a tarefa foi criada e não começou a executar pois uma ou mais das tarefas das quais ela depende não estão concluídas; (ii) pronta – a tarefa está pronta para executar; e (iii) suspensa – a tarefa está suspensa, normalmente à espera de uma mensagem. Com relação à instância de filtro e uma dada tarefa, ela pode estar em uma de cinco situações de execução: (i) computação própria – está realizando computação da própria tarefa; (ii) outra computação – está realizando algum processamento associado a outra tarefa; (iii) custos adicionais – está realizando alguma chamada à biblioteca do Anthill associada à tarefa; (iv) comunicação – a instância está recebendo ou enviando mensagens associadas à tarefa; e (v) processador ocioso – não está executando.

A partir das várias combinações de categorias e situações

de execução temos os perfis apresentados na Tabela 1. Identificamos quatro categorias nesses perfis, além das quatro já citadas: (i) desbalanceamento de carga – algumas instâncias estão trabalhando enquanto outras estão ociosas; (ii) atraso de sincronização – a tarefa foi criada mas ainda está aguardando as suas dependências para ser executada; (iii) contenção – a tarefa está pronta para executar, mas há outra tarefa executando na mesma instância; e (iv) computação reciclada – a tarefa está temporariamente suspensa e outra tarefa está executando. Note que a diferença entre contenção e computação reciclada é definida pela situação da tarefa, e cabe ressaltar que o primeiro degrada o desempenho da tarefa, mas o segundo aumenta o desempenho global da aplicação, e não afeta a tarefa, que não estaria executando de qualquer forma. As informações da análise de desempenho dinâmico são apresentadas pelo perfil de desempenho da tarefa, que pode ser visualizado na Figura 2 a esquerda e abaixo. A partir dessas categorias, podemos também visualizar quais são as tarefas que se relacionam com a tarefa analisada em cada caso, por exemplo, quais são as tarefas que exploram computação reciclada e quantidade de tempo reciclado (não mostrado por restrições de espaço).

Na próxima seção apresentamos um estudo de caso de utilização do Antfarm para analisar o desempenho de uma aplicação paralelizada de mineração de dados.

4. Estudo de Caso: ID3

Nesta seção ilustramos a utilização da ferramenta Antfarm analisando o desempenho de um algoritmo paralelizado usando o Anthill. O algoritmo avaliado é o ID3, que tem por objetivo determinar árvores de decisão a partir de bases de dados. Este algoritmo é apresentado na Seção 4.1. A seguir apresentamos alguns exemplos de análise do seu desempenho experimental usando o Antfarm.

4.1. O Algoritmo ID3

Em uma árvore de decisão, os nós folha são os elementos de dados individuais, cada um contendo uma ou mais tuplas com os respectivos valores para os seus atributos. Os nós internos identificam um atributo e cada valor possível do mesmo para os descendentes que são identificados por aquele valor. A profundidade dessa árvore é o maior número de perguntas sobre os valores dos atributos que precisam ser feitas para se encontrar qualquer dado individual. A idéia básica do algoritmo ID3 é usar uma busca gulosa e *top-down* dos dados a fim de encontrar o atributo que permita a maior discriminação dos dados a cada nível da árvore.

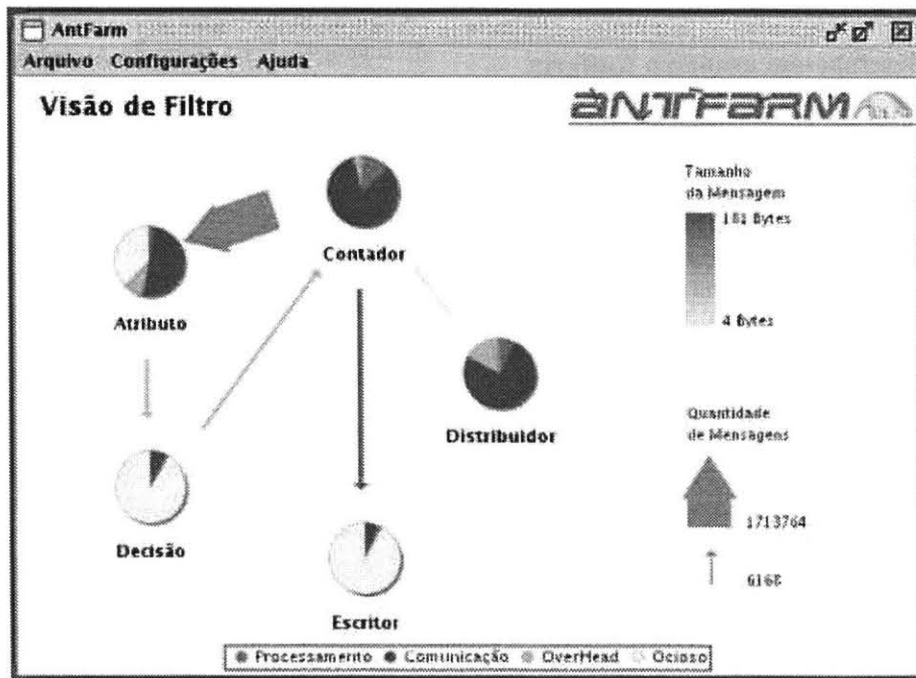
O ponto de partida do ID3 é um conjunto de m tuplas, cada uma contendo instâncias dos n atributos e uma entre c classes possíveis. Cada atributo a pode assumir v_a valores.

1 p.atr = None
2 p.instance = None
3 p.dataset = T
4 while ($\exists p \in Partitions$)
5 $\quad Partitions- = p$
6 $\quad q = \{t \in p.dataset \wedge p.atr = p.instance\}$
7 $\quad \forall a \in Attributes$
8 $\quad\quad \forall v \in Values(a)$
9 $\quad\quad\quad prob_c = \frac{|\{t \in q \wedge t.a = t.v \wedge t.c = c\}|}{|\{t \in q \wedge t.a = t.v\}|}$
10 $\quad\quad\quad info_{a,v} = \sum_{c \in Classes} -prob_c * log(prob_c)$
11 $\quad \forall a \in Attributes$
12 $\quad\quad prob_v = \frac{|\{t \in q \wedge t.a = t.v\}|}{|q|}$
13 $\quad\quad gain_a = \sum_{v \in Values(a)} info_{a,v} * prob_v$
14 $\quad disc = a|gain_a isMaximum$
15 $\quad \forall v \in Values(disc)$
16 $\quad\quad p.atr = disc$
17 $\quad\quad p.instance = v$
18 $\quad\quad p.dataset = q$
19 $\quad\quad Partitions+ = p$

Figura 1. Algoritmo ID3.

O processo de geração da árvore baseia-se no uso de discriminantes. Cada discriminante é um teste de um atributo específico que é usado para dividir o conjunto de tuplas em dois ou mais sub-conjuntos (dependendo do número de valores diferentes que possam ocorrer no discriminante). Inicialmente, não há nenhum discriminante e a partição é o conjunto de todas as tuplas. À medida que novos discriminantes são encontrados nós particionamos o conjunto de tuplas em subconjuntos, até que todas as tuplas em uma partição pertençam a uma mesma classe.

O pseudo-código do algoritmo é apresentado na Figura 1. As linhas 1–3 inicializam as partições, sendo que a primeira corresponde à base completa. O *loop* da linha 4 executa até que não hajam mais novas partições. Para cada uma delas (linha 5), selecionamos as tuplas que comporão a partição q na linha 6. Em seguida calculamos a informação para cada atributo e valores de instâncias usando uma métrica de entropia (linhas 7–10). As linhas 11–13 calculam o ganho de informação para cada atributo e, no último passo, determinam o atributo que gera o maior ganho de informação, adicionando as novas partições resultantes ao conjunto $Partitions$. Em termos de paralelização, há duas reduções multi-objetivo nas linhas 10 e 13, as quais usamos para identificar as fronteiras entre os filtros, produzindo assim três filtros. O primeiro deles, denominado *Contador*, realiza as operações associadas com as linhas de 4 a 9, sendo responsável pela contagem do número de instâncias de cada valor de cada atributo. O segundo filtro, *Atributo*, realiza as operações associadas com as linhas 10 a 12, computando o ganho de informação de cada atributo testado no filtro anterior. Finalmente, o terceiro filtro, *Decisão*,



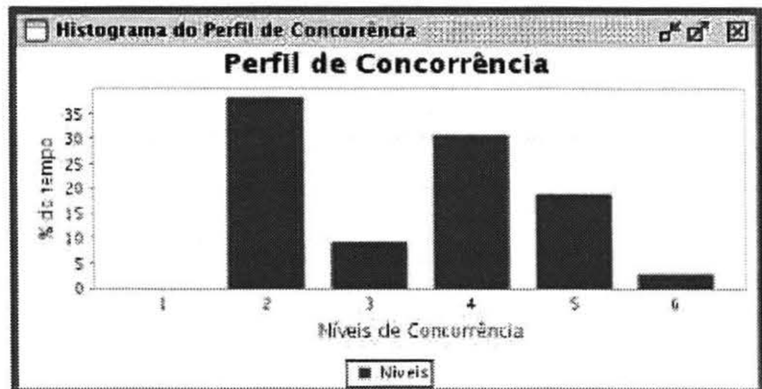


Figura 2. Exemplos de visualizações da ferramenta Antfarm

realiza as operações restantes, que incluem a seleção do atributo apropriado e a comunicação dessa escolha de volta ao primeiro filtro, onde o processo continua, selecionando novos atributos discriminantes para cada uma das classes produzidas.

Em nosso algoritmo nós exploramos as três dimensões de paralelismo como discutidas anteriormente. Na dimensão do paralelismo de tarefas, identificamos as operações de cada filtro no *pipeline*. Na dimensão do paralelismo de dados, temos diversas instâncias dos filtros, cada uma processando um subconjunto da base de dados original. A granulosidade do particionamento pode ser bem fina. A dimensão final, assincronia, é explorada na medida em que as diferentes cópias dos filtros podem processar vários nós da árvore de decisão simultaneamente, não havendo cópias ociosas. Na próxima seção

apresentamos uma análise de desempenho da versão paralelizada do ID3 usando o Antfarm.

4.2. Análise de Desempenho usando o Antfarm

Nesta seção apresentamos uma análise de desempenho de execuções experimentais do algoritmo ID3 usando a ferramenta Antfarm. Os experimentos foram executados em um *cluster* de PCs conectados por uma rede Fast Ethernet (100 Mbps). Cada nó é composto por um computador Pentium IV a 3 GHz com 1 GB de memória principal usando o sistema operacional Linux, *kernel* 2.6. Para avaliar nosso algoritmo utilizamos dados sintéticos gerados a partir de funções específicas descritas na literatura [13]. Em particular utilizamos a função 7 que produz árvores de decisão mais profundas, onde há maior paralelismo potencial a ser explorado. A configuração usada nos experimentos possui 4 instâncias do filtro Contador, duas instâncias do filtro Atributo e uma instância de cada um dos demais filtros (Decisão, Distribuidor e Escritor). Os filtros Distribuidor e Escritor são utilizados apenas na inicialização e terminação do algoritmo.

O ponto de partida da nossa análise é a "Visão de Filtro"(Figura 2, janela superior), onde podemos clara e rapidamente perceber que há intensa comunicação entre os filtros Contador e Atributo. Mais ainda, fica claro que o filtro mais sobrecarregado nessa configuração é o Contador, tendo em vista a quantidade mínima de ociosidade que é observada nesse filtro. A Visão de Tarefa (não mostrada por restrições de espaço) indica uma grande variabilidade no tempo de execução das tarefas, independentemente do nível da árvore sendo calculado e outros atributos estáticos das tarefas.

Continuamos a nossa análise investigando o perfil de desempenho das tarefas executadas em cada um dos filtros. No caso do filtro contador, mostrado no canto inferior à esquerda da Figura 2, a maior fonte de degradação de desempenho é contenção. Entretanto, uma quantidade de tempo três vezes maior é associada a computação reciclada, o que nos leva a concluir que, embora os filtros estejam sobrecarregados, a computação realizada por eles poderia ser ainda mais distribuída, com a ressalva do quanto o custo de comunicação aumentaria com o número de instâncias. A análise dos outros filtros mostra que a ociosidade aumenta para as instâncias do filtro Atributo e domina o tempo de execução do filtro Decisão.

Uma outra vertente de observação é a análise de disparo, a qual será utilizada para entender a grande variabilidade do tempo de execução das tarefas. Neste caso comparamos duas tarefas, A e B. A tarefa A foi executada rapidamente, enquanto a tarefa B demandou 10 vezes mais tempo que ela. Analisando os perfis de desempenho das tarefas nos filtros (não mostrados por restrições de espaços), verificamos que a quantidade de computação reciclada aumenta significativamente, em particular do filtro Atributo. A análise de disparo gera, entre outros, uma análise do nível de concorrência na execução de uma tarefa, ou seja, quantas instâncias de filtro estavam trabalhando na tarefa durante um dado intervalo. Quanto maior for este número, menor o tempo de disparo e a multiprogramação. A Figura 2 a direita e abaixo mostra o perfil de concorrência da tarefa A, onde vemos que o nível de concorrência chega a 6, mas há uma porcentagem significativa do tempo quando o nível de concorrência é 4. Neste caso, os filtros Contadores estavam trabalhando na resolução das tarefas. Por outro lado, avaliando o nível de concorrência da tarefa B (não mostrado por restrições de espaço), verificamos que ele raramente ultrapassa 2. O problema neste caso é que uma das instâncias do filtro Contador havia estado sobrecarregada e realizou a computação associada à tarefa B após todos os outros filtros, resultando em um atraso global da tarefa e um tempo de disparo maior, em particular para os filtros Atributo. Note que a nossa ferramenta foi capaz de detectar com precisão uma situação que pode ser transiente no sistema como um todo mas que causou significativo impacto durante um período da execução.

Em suma, podemos dizer que o paralelismo potencial do algoritmo ID3 ainda não foi totalmente explorado, em particular no filtro contador. Entretanto, há um compromisso a ser explorado em termos do volume de processamento e comunicação neste filtro, tendo em vista o impacto da comunicação realizada pelo filtro.

5. Trabalhos Relacionados

A dificuldade da depuração do desempenho em aplicações paralelas motivou a construção de um grande número de ferramentas para os mais variados ambientes. Essas ferramentas apresentam os mais variados perfis de desempenho, em geral focando nos aspectos temporais e estruturais (por exemplo, máquina, segmento de código da aplicação) da execução de programas paralelos. Exemplos de ferramentas desta natureza são Vampir [7], TAU [9] e Pablo [8]. AntFarm também provê tais perfis de desempenho mas explora uma nova dimensão, tarefa, que é inerente ao ambiente AntHill e cujo entendimento é fundamental para analisar o desempenho dos programas, como discutido na Seção 3.

Além de apresentar perfis de desempenho, várias ferramentas provêm metáforas que facilitam o entendimento do desempenho dos programas, fornecendo indícios para as causas de degradação de desempenho. Neste contexto podemos citar Carnival [5], as ferramentas do projeto Paradyn [6] e SCALEA [11]. AntFarm introduz duas novas técnicas de entendimento de desempenho que novamente exploram o conceito de tarefa, facilitando a avaliação

de como a assincronia das tarefas afeta o desempenho da aplicação, tanto positiva quanto negativamente.

Finalmente temos ferramentas para visualização de registros de desempenho, tais como MPE/Jumpshot [14]. O AntFarm possui integração transparente com tais ambientes, permitindo ao usuário visualizar os dados coletados no maior nível de detalhe possível.

6. Conclusões e Trabalhos Futuros

Neste artigo apresentamos a ferramenta AntFarm, que tem por objetivo auxiliar programadores a analisar e entender, de forma rápida e objetiva, o desempenho de aplicações paralelas no ambiente AntHill. A ferramenta incorpora medidas usuais de análise de desempenho de programas paralelos, como tempo despendido em comunicação, processamento e mesmo o tempo que os processadores ficam ociosos. Todas essas informações são apresentadas de forma gráfica, explorando variadas metáforas visuais. AntFarm também introduz duas novas técnicas de análise de desempenho de aplicações que seguem o paradigma filtro-fluxo identificado: estrutural e dinâmico. Utilizamos a ferramenta para analisar o desempenho de um algoritmo de classificação, mais especificamente ID3. A análise demonstrou significativa contenção em um dos filtros, enquanto pôde também determinar ociosidade nos outros filtros, muitas vezes resultando em computação reciclada.

Em termos de trabalhos futuros, distinguimos a incorporação de novas análises e visualizações, além de permitir que as análises de desempenho sejam feitas em tempo real, como subsídio para ajustes imediatos ou mesmo ajuste de parâmetros das aplicações paralelas para execuções posteriores.

Referências

[1] A. Acharya, M. Uysal, and J. Satlz. Active disks: Programming model, algorithms and evaluation. In *Internation Conference on Architectural Support for programming Languages and Operating Systems (ASPLOS VIII)*, pages 81–91. ACM Press, Oct 1998.

[2] M. Beynon, C. Chang, U. Çatalyürek, T. Kurç, A. Sussman, H. Andrade, R. Ferreira, and J. Saltz. Processing large-scale multi-dimensional data in parallel and distributed environments. *Parallel Computing*, 28(5):827–859, 2002.

[3] M. Beynon, T. Kurc, A. Sussman, and J. Saltz. Design of a framework for data-intensive wide-area applications. In *Heterogeneous Computing Workshop (HCW)*, pages 116–130. IEEE Computer Society Press, May 2000.

[4] R. Ferreira, W. M. Jr., D. Guedes, L. Drummond, B. Coutinho, G. Teodoro, T. T. andR. Araújo, and G. Ferreira. Anthill: A scalable run-time environment for data mining applications. In *Proc. of the Symp. on Computer Architecture and High Performance Computing, SBAC*, Rio de Janeiro, Brazil, October 2005. IEEE.

[5] W. Meira Jr., T. LeBlanc, and V. Almeida. Using cause-effect analysis to understand the performance of distributed programs. In *Proceedings of SPDT98: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 101–111, Welches, OR, August 1998. ACM.

[6] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, 1995.

[7] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[8] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. Scalable Parallel Libraries Conf.*, pages 104–113. IEEE Computer Society, 1993.

[9] S. Shende, A. D. Malony, J. Cuny, P. Beckman, S. Karmesin, and K. Lindlan. Portable profiling and tracing for parallel, scientific applications using C++. In *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 134–145, New York, NY, USA, 1998. ACM Press.

[10] M. Spencer, R. Ferreira, M. Beynon, T. Kurc, U. Catalyurek, A. Sussman, and J. Saltz. Executing multiple pipelined data analysis operations in the grid. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18. IEEE Computer Society Press, 2002.

[11] H.-L. Truong, T. F. G. Madsen, A. D. Malony, H. Moritsch, and S. Shende. On using SCALEA for performance analysis of distributed and parallel programs. In *SC 01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, November 2001.

[12] A. Veloso, W. M. Jr., R. Ferreira, D. Guedes, and S. Parthasarathy. Asynchronous and anticipatory filter-stream based parallel algorithm for frequent itemset mining. In *Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, 2004.

[13] M. Zaki, C. Ho, and R. Agrawal. Parallel classification for data mining on shared-memory multiprocessors. In *ICDE '99: Proceedings of the 15th International Conference on Data Engineering*, page 198, Washington, DC, USA, 1999. IEEE Computer Society.

[14] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with Jumpshot. *The International Journal of High Performance Computing Applications*, 13(3):277–288, Fall 1999.