Mapeamento de Programas I^3 para Aplicações Anthill Paralelas de Fluxos de Dados baseadas em Filtros

Luís F. W. Góes, Ítalo Giovani, Renato Ferreira, Wagner Meira Jr.
Universidade Federal de Minas Gerais – Belo Horizonte
{lfwgoes,italost,renato,meira}@dcc.ufmg.br

Resumo

Aplicações atuais de mineração de dados, simulação e visualização científica oferecem várias oportunidades de paralelismo por serem iterativas, irregulares e intensivas em termos de E/S (programas I^3). O mapeamento e escalonamento de programas I^3 para aplicações paralelas de fluxos de dados baseadas em filtros é bastante complexo, pois eles devem considerar aspectos de localidade, dependência de dados e tarefas. A plataforma Anthill provê um modelo de programação adequado para implementação e execução de aplicações paralelas baseadas em filtros. Portanto, neste trabalho, nossos ***objetivos principais*** *são: a proposta e implementação do algoritmo AnthillPart para o mapeamento de um grafo de tarefas de um programa I^3 em filtros; a análise do desempenho das aplicações mapeadas pelo AnthillPart e escalonadas pelo AnthillSched.*

1. Introdução

O crescimento do poder de processamento, largura de banda das redes, memória primária e capacidade dos discos têm permitido o paralelismo eficiente e escalável de um grande número de classes de aplicações, dentre elas a mineração de dados [9] [21], visualização [4] [17] e simulação científica [18].

Essas aplicações não somente demandam muitos recursos computacionais, mas também elas são um desafio para a extração de paralelismo, pois são geralmente irregulares, intensivas em termos de E/S e iterativas. Essas aplicações pertencem a uma classe de programas que chamamos de programas I^3 [16].

Um programa ser irregular significa que o seu tempo de execução não pode ser predito, além do programa mudar seu comportamento durante uma execução. O fato de que os programas são intensivos em termos de E/S, faz com que o desempenho seja bastante afetado pelos componentes do sistema e pela quantidade de sobreposição entre o processamento e a comunicação atingida durante a execução do programa. Além disso, os programas I^3 realizam computação por meio de vários domínios, não apenas consumindo dados desses domínios, mas também gerando dados e aumentando o volume de dados manipulados em tempo real [16].

Finalmente, a iteratividade implica em duas questões que afetam a extração de paralelismo: localidade de referência e grau de paralelismo. A localidade é afetada pelo padrão de acesso das iterações ao longo do tempo. Já o grau de paralelismo é função das dependências entre os dados e iterações [16]. Como conseqüência dessas características, o mapeamento de programas I^3 em aplicações paralelas é um problema bastante complexo, já que ele deve considerar a localidade, tamanho da entrada de dados, dependência de dados e tarefas.

Geralmente, programas seqüências são mapeados em aplicações paralelas que seguem modelos de programação como memória compartilhada e troca de mensagens. Existem poucos trabalhos lidando com o mapeamento de aplicações seqüências no modelo de programação baseado em filtros [4] [21].

Além disso, os escalonadores de aplicações paralelas são projetados para lidar com aplicações de processamento intensivo [3] [5] [6] [7] [12] [13] [14] [15] [20] [24] [25]. Algumas pesquisas propuseram políticas para lidar com programas intensivos e irregulares de E/S [1] [2] [17] [18] [19] [22] [23], mas não com programas I^3.

Neste trabalho, nós investigamos o mapeamento de programas I^3 em aplicações Anthill paralelas de fluxo de dados baseadas em filtros [4]. A estrutura dessas aplicações é composta de filtros que se comunicam usando fluxos de dados identificados, que garantem a consistência entre os filtros. Cada filtro pode ter uma ou mais instâncias, permitindo um alto nível de paralelismo. Essas aplicações são assíncronas e implementadas usando um paradigma baseado em eventos. A plataforma Anthill fornece o suporte necessário para a programação deste tipo de aplicação.

Para o mapeamento de programas seqüenciais em aplicações Anthill, um grafo de dependência de tarefas, ou seja, um grafo no qual cada vértice representa uma ou parte de uma etapa do programa e as arestas a seqüencialidade entre duas etapas, é essencial para a identificação dos filtros da aplicação. Após a implementação da aplicação, uma política de escalonamento específica deve ser utilizada para a determinação do número de cópias ou instâncias de cada filtro [16].

Este trabalho é parte de um projeto que consiste no mapeamento e execução de programas seqüenciais I^3 em aplicações Anthill paralelas de fluxo de dados baseadas em filtros. O projeto é composto das seguintes etapas: extração do grafo de tarefas; mapeamento do grafo de tarefas em filtros; geração do código da aplicação Anthill; escalonamento e execução da aplicação Anthill. Neste artigo, os **principais problemas** abordados são o mapeamento do grafo de tarefas em filtros e o escalonamento de aplicações Anthill.

Os **objetivos principais** deste artigo são: proposta e implementação do algoritmo AnthillPart para o mapeamento do grafo de tarefas em filtros; análise do desempenho das aplicações mapeadas pelo AnthillPart e escalonadas pelo AnthillSched, uma política já proposta em outra trabalho [16].

Este artigo está organizado da seguinte forma. Nós apresentamos os trabalhos relacionados na seção 2. Na terceira seção, nós introduzimos a plataforma Anthill e na seção 4, a nossa proposta, análise de complexidade e verificação do algoritmo AnthillPart. Então, nós apresentamos o planejamento de experimentos e a análise de desempenho de aplicações Anthill geradas pelo AnthillPart nas seções seguintes. Por último, apresentamos as conclusões e trabalhos futuros.

2. Trabalhos Relacionados

Esta seção apresenta alguns trabalhos relacionados com o mapeamento e escalonamento de programas I^3 em aplicações baseadas em filtros.

Um trabalho fundamental para a nossa pesquisa foi o desenvolvimento do DataCutter [4]. Ele é um middleware que permite que aplicações sejam executadas de forma eficiente em ambientes distribuídos heterogêneos. Ele foi projetado baseado no modelo de programação em filtros, no qual os programas são compostos por um conjunto de filtros que se comunicam por meio de fluxos de dados [4]. O DataCutter permite a instanciação de várias cópias de um filtro em tempo de execução. A abstração de um fluxo mantém a ilusão de comunicação ponto a ponto entre os filtros e quando uma cópia gera uma saída pra um fluxo, o middleware cuida para que o dado seja entregue a uma das cópias transparentes do filtro destino.

Um trabalho bastante relacionado é o LPSched, uma política de escalonamento com programação linear que lida com aplicações assíncronas e intensivas de E/S baseadas em filtros [17]. Apesar disso, o mapeamento dos programas seqüências para aplicações baseadas em filtros tem sido feita manualmente pelo programador. Depois de implementadas, as aplicações são escalonadas pelo LPSched baseado em informações extraídas apriori. Ele procura maximizar o fluxo de dados entre os processos e minimizar o número de processadores utilizados.

Em [19], os programas de mineração de dados são mapeados em aplicações do tipo BoT (bag of tasks) e escalonadas em uma grade computacional. Os processos da aplicação são agrupados em grupos maiores e escalonados para os mesmos processadores de acordo com os dados de entrada. Esta política procura minimizar a quantidade de dados transferidos entre os nodos de uma grade.

Como apresentado, de acordo com a nossa revisão bibliográfica, nenhum trabalho que faça o mapeamento de programas I^3 em aplicações paralelas baseadas filtros foi encontrado.

3. Plataforma Anthill

Nós estendemos o DataCutter e criamos a plataforma Anthill [27]. Ela provê um mecanismo chamado fluxo identificado (labeled stream) que permite a seleção de uma cópia particular baseada nos dados relacionados com as mensagens (labels). Essa extensão permite um modelo de programação mais rico, facilitando a partição do estado global de cópias transparentes. Além disso, o Anthill provê um framework, no qual a execução da aplicação pode ser decomposta numa seqüência de tarefas intermediárias que precisam ser realizadas e também pode criar múltiplos filtros em um ambiente iterativo. Ele explora o paralelismo em vários níveis: tempo, espaço e assincronia [16][27].

Como podemos observar na Fig. 1, uma aplicação Anthill explora o paralelismo de tempo como um pipeline, porque ela é composta de N filtros (estágios ou fases de processamento) conectados por fluxos de dados (canais de comunicação). Esse modelo de aplicação força explicitamente o programador a dividir o programa em fases bem definidas (filtros), nos quais o dado é transformado, por um filtro, em um dado de

outro domínio ou espaço de dados, que é necessário para o próximo filtro.

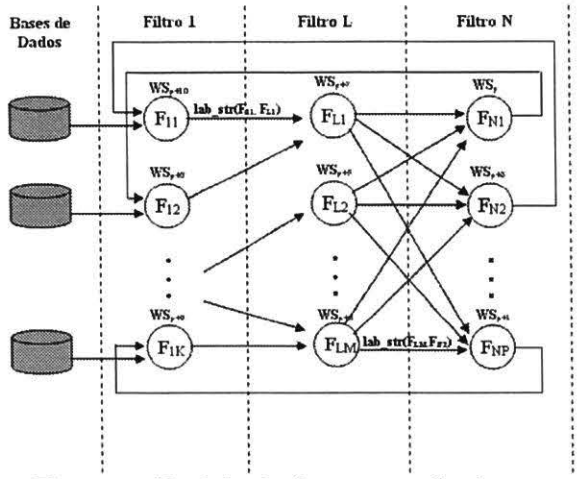

Figura 1: Modelo de Programação de uma Aplicação Anthill.

O modelo de programação do Anthill também explora o paralelismo de espaço, pois cada filtro possui múltiplas cópias ou instâncias. Cada canal de comunicação entre filtros pode ser nomeado para direcionar um fluxo de dados para uma cópia específica de um filtro (ponto a ponto) ou para todas as cópias de um filtro (broadcast). Uma consequência do paralelismo de espaço é o paralelismo de dados, porque um banco de dados é automaticamente dividido entre as cópias dos filtros. Juntamente com os fluxos, o paralelismo de dados provê um mecanismo eficiente de divisão da demanda de E/S entre os filtros.

Além disso, o Anthill tira vantagem da assincronia e provê iteratividade. Cada aplicação possui um conjunto de pedaços de trabalho (work slices) a serem executados. De acordo com os conjuntos de dados lidos das bases de dados, um pedaço de trabalho é criado. Eles podem ser executados independentemente, respeitando apenas a dependência de dados. Depois de processados, eles também podem gerar novos pedaços de trabalho, esse fato justifica a necessidade de um processo iterativo.

4. Proposta do AnthillPart

Na Fig. 2, podemos observar todas as etapas necessárias para o mapeamento de um programa seqüencial I^3 para um aplicação Anthill e sua execução na plataforma Anthill. Dado um programa seqüencial com algumas diretivas indicando etapas importantes e uma análise de código, torna-se possível extrair automaticamente um grafo de dependência de tarefas.

Um grafo de tarefas G = (V,A) é um grafo direcionado acíclico que representa um algoritmo, ou seja, uma seqüência de operações, ou tarefas, representadas pelos vértices $v_i \in V$. Uma tarefa v_i é uma função que mapeia variáveis de entrada de um determinado domínio em variáveis de saída de um outro domínio. Se o domínio de entrada e saída é o mesmo, então não há transição de domínio, ou seja, necessidade de comunicação. Uma tarefa pode ser desde uma simples soma até um processamento complicado como um *join* de duas listas. Uma aresta $a_i \in A$ representa a dependência entre duas tarefas, ou seja, as variáveis do domínio de saída de uma tarefa v são variáveis do domínio de entrada da tarefa v'. Apesar de ser uma etapa importante, atualmente, a extração do grafo de tarefas é realizada manualmente. Neste trabalho, a partir de um grafo de dependência de tarefas, propomos um algoritmo chamado AnthillPart, responsável pela identificação, particionamento e mapeamento do grafo em filtros. Após esse mapeamento, o programa seqüencial é codificado como uma aplicação Anthill. Finalmente, o número de cópias de cada filtro e onde essas serão executadas são determinados em tempo de execução pelo AnthillSched.

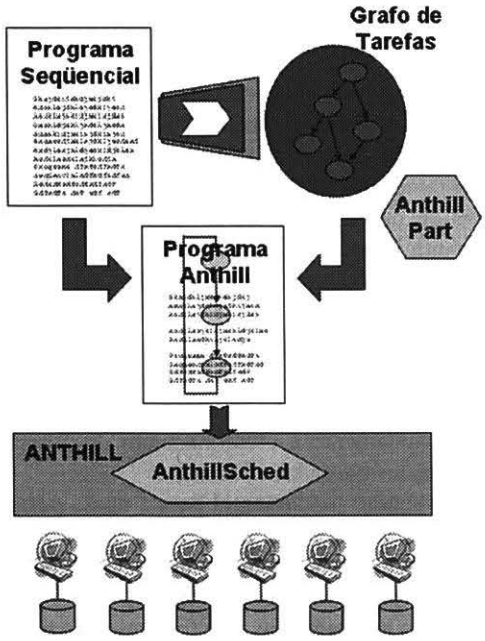

Figura 2: Diagrama do mapeamento de um Programa I^3 para uma Aplicação Anthill.

Como exemplo, mostramos na Fig. 3, o grafo de dependência de tarefas para o algoritmo Apriori de mineração de dados. Ele procura encontrar relações

interessantes entre uma grande quantidade de itens, baseando-se na seguinte regra: todos os subconjuntos não-vazios de conjuntos freqüentes de itens também precisam ser freqüentes [26]. As etapas principais do Apriori são: i) contar o número de ocorrências de cada conjunto nas transações; ii) descartar os conjuntos que não possuem um suporte mínimo (verificar se é frequente); criar as combinações de K+1 itens com os conjuntos freqüentes restantes [26].

No grafo da Fig. 3, mostramos apenas duas iterações do Apriori para um subconjunto mínimo de itens. Os identificadores dos vértices ou tarefas do grafo não são necessários para o funcionamento do AnthillPart, desde que o grafo respeite a precedência das tarefas.

Neste exemplo, o primeiro nível do grafo, ou seja, o primeiro conjunto independente de vértices, pode ser identificado como a primeira etapa de contagem das ocorrências de cada conjunto de transações. O segundo nível (composto de dois vértices) é responsável por descartar os conjuntos não freqüentes. Por último, no terceiro e quarto níveis (compostos por um vértice cada), está a etapa responsável por criar novas combinações de itens. Note que após esta última etapa, no quinto nível do grafo, existe uma operação de broadcast para todos os vértices que são dependentes dos vértices do primeiro nível do grafo.

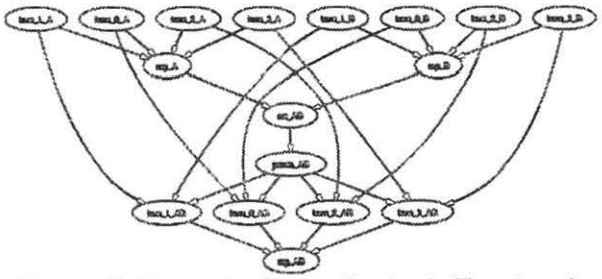

Figura 3: Exemplo de um Grafo de Tarefas de um Programa I^3.

Observando as características do grafo do programa Apriori e de outros programas I^3, nós propomos e apresentamos o algoritmo AnthillPart, na Fig.4, para a identificação e mapeamento de grafos de dependência de tarefas para aplicações Anthill baseadas em filtros.

Resumidamente, o nosso algoritmo AnthillPart parte de princípio que aplicações baseada em filtros permitem cópias transparentes ou instâncias. Portanto, identificando os conjuntos independentes de vértices que estão em um mesmo domínio, torna-se óbvio que cada conjunto se torna um filtro.

A partir da identificação dos conjuntos independentes, devemos reduzir ao máximo o número de filtros existentes utilizando dois passos: determinando as seqüências de tarefas e os *loops*. Por último, determinamos o padrão de comunicação entre os filtros como redução ou *broadcast*.

```
Entrada: Grafo: grafo
Saída: Grafo: filtros

Grafo subGrafo;

subGrafo = Det_Conjuntos_Indep(grafo);
subGrafo = Det_Aresta_Avanço(subGrafo);
subGrafo = Agrupa_Sequenciais(subGrafo);
filtros = Identifica_Filtros(subGrafo);
```

Figura 4: Pseudocódigo do Algoritmo AnthillPart.

Abaixo, nós apresentamos algumas formalizações necessárias para demonstrar a validade do algoritmo proposto. Dado um grafo de dependência de tarefas de entrada G(V,A), onde V é o conjunto de vértices e A o conjunto de arestas, o AnthillPart detecta as arestas de avanço do grafo, que evidenciam a presença de um *loop*.

Hipótese 1: Uma aresta de avanço *(v,v')* mostra que um vértice *v* de um nível inferior possui uma relação de dependência com um vértice *v'* de um nível k superior, sendo k > 1, que indica uma iteração do algoritmo.

Para provar essa hipótese, os vértices adjacentes à aresta devem possuir as mesmas funcionalidades e uma aresta de retorno não pode ser adjacente à dois vértices que pertençam a iterações distintas.

Prova 1.1: Supondo que os vértices adjacentes à aresta de retorno (v,v') são vértices que realizam a mesma operação. A aresta indica que um vértice v precisa da saída do vértice v'. Como queremos modelar a aplicação baseada em uma seqüência de filtros, cada filtro deve ser independente do outro. Então, a aresta de avanço indica que devemos passar o dado de v, nível a nível (sem nenhuma modificação do dado) até chegar no vértice v'. Portanto, a sobrecarga para transportar este dado seria alta. Podemos então considerar que v = v' e criar um loop, tornando a sobrecarga nula.

Prova 1.2: Se existir v e v' que dependem da saída de dados de u e v' depende de v, teríamos três arestas de avanço, (u,v), (u,v') e (v,v'). Porém, a aresta (u,v) passa a ser desnecessária, pois a informação redundante fica explícita e o dado pode chegar a v' passando por v. Então não pode-se considerar as arestas do tipo (v,v'),

evitando que várias iterações sejam interpretadas como uma só.

Para executar detecção de um *loop*, o algoritmo AnthillPart possui um custo de O (|V|+|A|), ou seja, o custo de uma busca em profundidade. No exemplo do algoritmo Apriori, o subGrafo de saída G'(V',A') seria composto de 4 níveis com um vértice em cada um deles, mas conservando-se os graus de entrada e saída de cada vértice. A cada passagem de nível na busca em profundidade, a transmissão de dados é avaliada para definir se é feita por meio de uma operação de broadcast ou não, bastando contar as arestas. Sendo um nível x com i vértices e o nível seguinte y com j vértices, a verificação de broadcast é na verdade verificar se existem i*j arestas ligando um nível ao outro. Esta informação é armazenada para construção da aplicação Anthill no final do algoritmo.

A próxima etapa consiste no agrupamento de vértices seqüenciais, ou seja, os vértices que possuem apenas um de grau de entrada são agrupados com os que possuem apenas um de grau de saída, sendo eles adjacentes. Esta etapa possui um custo de O(V').

Por último, para o subGrafo resultante, o conjunto de vértices é o número mínimo de filtros necessários para a aplicação Anthill. Esta operação tem custo O(V'). Na Fig. 5, podemos observar o grafo resultante do algoritmo AnthillPart para o exemplo do Apriori.

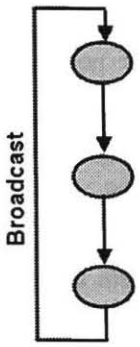

Figura 5: Aplicação Anthill para o Programa Apriori.

A complexidade total para a execução do algoritmo AnthillPart é igual a O (|V|+|A|) + O (V') + O (V'), que é dominado assintoticamente por **O(|V|+|A|).**

5. Resultados

Nesta seção, nós analisamos o desempenho de uma aplicação Anthill baseada em um grafo gerado pelo AnthillPart e escalonada pelo AnthillSched.

5.1 Configuração dos Experimentos

Para realização dos experimentos, utilizamos o programa irregular, intensivo de E/S e iterativo ID3 para mineração de dados. O ID3 é utilizado para classificação de dados por meio de árvores de decisão. Por questões de espaço, o grafo de dependência de tarefas do ID3 será omitido. Mas o grafo resultante é composto por 3 vértices e um *loop broadcast* como no Apriori.

O ID3 possui um parâmetro de entrada chamado tamanho mínimo do nodo, que é utilizado para determinar se um nodo possui o número mínimo de ocorrências de um atributo para a sua criação. O ID3 está sendo executado no ambiente Tamanduá [27]. Ele é uma plataforma de mineração de dados escalável e orientada por serviços que utiliza algoritmos eficientes para clusters com grandes bases de dados. Atualmente, o Tamanduá é utilizado para mineração em bases governamentais.

No trabalho [16], baseado em logs reais obtidos do Tamanduá, foi possível determinar o intervalo de chegada entre requisições e o padrão do tamanho mínimo do nodo. Então usou-se um modelo de carga com as seguintes distribuições de probabilidade: i) o intervalo de chegada é uma distribuição exponencial com parâmetro λ = 0.00015352; ii) o padrão do tamanho mínimo de nodos é uma distribuição de Pareto com parâmetros θ = 0.61815 e a = 0.00075019, onde θ é um parâmetro contínuo da forma e a um parâmetro contínuo de escala.

Usando um gerador de cargas, nós variamos a semente e geramos 10 cargas de trabalho. Cada carga é composta por 1000 aplicações, onde todas são implementações do programa ID3 com diferentes tamanhos mínimos de nodos e tempos de submissão, que tornam as execuções diferentes.

Para avaliar o desempenho das aplicações, nós usamos duas métricas de desempenho: tempo de execução médio (Eq.1) e tempo de espera médio (Eq.2).

O tempo de execução médio (em segundos) é o somatório do tempo de execução de todas as aplicações de cada carga de trabalho dividido pelo número de cargas (Equação 1).

$$\text{TempoExecução} = \frac{\sum \text{TempoExecução}_i}{\text{NúmeroCargas}}$$

(Equação 1)

O tempo de espera médio (em segundos) é o somatório do intervalo entre a submissão e o início da

execução das aplicações de cada carga de trabalho dividido pelo número de cargas (Equação 2).

$$\text{TempoEspera} = \frac{\sum \text{TInicio}_i - \text{TSubmissão}_i}{\text{NúmeroCargas}}$$

(Equação 2)

O computador paralelo utilizado foi um cluster Linux composto de 16 nodos Pentium 4 3.0Ghz com memórias primárias de 1GB e 120 GB de disco, interconectados por uma rede Fast Ethernet. Por último, é importante ressaltar que para todos os testes experimentais realizados foram calculados os intervalos de confiança com nível e confiança igual a 95%.

5.2 Resultados Experimentais

Para compararmos o desempenho do AnthillSched Otimizado (OAHS) [16], nós também utilizamos outras políticas de escalonamento: AnthillSched Não Otimizado (NOAHS), Estratégia Balanceada (BS) e Estratégia Todos (AS). Todas as políticas utilizam o número máximo de processadores disponíveis e procuram balancear o número de cópias de cada filtro.

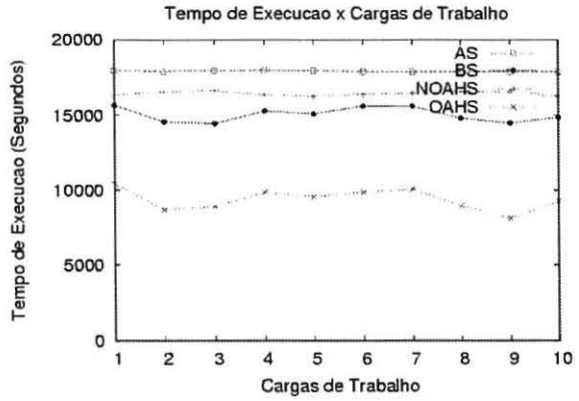

Figura 6: Tempo de Execução Médio para todas as cargas vs políticas de escalonamento.

Quando uma aplicação é submetida, o Anthill invoca o AnthillSched com os parâmetros de entrada necessários como o tamanho mínimo dos nodos e o número de processadores livres. O AnthillSched procura determinar o número de cópias de cada filtro de acordo com a porcentagem do tempo ocupada por cada filtro com E/S e processamento, medidos em execuções controladas. Por exemplo, se temos uma aplicação com três filtros, sendo que, respectivamente, cada filtro gaste 60%, 20% e 20% do tempo total e o número de processadores livres seja igual a 10. Então, o primeiro filtro tem 6 cópias, o segundo e o terceiro com 2 cópias cada. A versão otimizada do AnthillSched detecta se uma aplicação deve ser executada em paralelo ou seqüencial.

A política BS tenta balancear o número de cópias de cada filtro considerando que todos os filtro possuem uma porcentagem igual de participação no tempo total. Por exemplo, se temos 15 processadores livres, cada filtro tem 5 cópias. Por último na política AS, todos os filtros possuem cópias em todos os processadores de forma concorrente.

Na Fig. 6, os resultados mostram que a política AS apresentou o pior tempo de execução médio para todas as cargas de trabalho. Enquanto, as políticas NOAHS e BS apresentaram desempenho similar, com uma pequena vantagem para o BS. O mesmo acontece com o tempo de espera médio apresentado na Fig. 7. Também em ambos os casos, o OAHS apresenta um desempenho muito superior por considerar aspectos específicos da aplicação durante o escalonamento. Portanto, as aplicações Anthill terminam mais rápido e conseqüentemente diminuem o tempo de espera das outras aplicações na fila.

A estratégia AS se mostrou inviável, portanto não podemos assumir que cópias de filtros diferentes são complementares (E/S e CPU) para aplicações ID3.

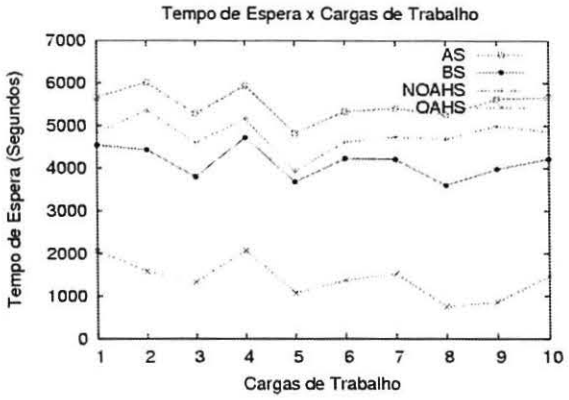

Figura 6: Tempo de Espera Médio para todas as cargas vs. políticas de escalonamento.

Com estes experimentos, nós podemos observar que a execução de uma aplicação Anthill gerada a partir do grafo resultante do AnthillPart e escalonada pelo AnthillSched apresenta bom desempenho comparada com outras políticas de escalonamento.

6. Conclusão

O mapeamento de programas I^3 em aplicações Anthill de fluxo de dados baseadas em filtros ainda está no seu estágio inicial, mas os resultados são bastante promissores. O algoritmo AnthillPart provou realizar o mapeamento de um grafo de dependência de tarefas em uma aplicação Anthill da mesma forma que um programador experiente, respeitando as etapas fundamentais dos programas analisados. Depois de implementada, uma aplicação foi escalonada pelo AnthillSched que apresentou desempenho superior aos demais escalonadores.

Para a geração automática de código paralelo para Anthill, ainda faltam definir as diretivas de compilação necessárias, os parâmetros de entrada das aplicações, a ligação entre o grafo gerado pelo AnthillPart e a o código seqüencial etc.

As **principais contribuições** deste trabalho foram: a proposta e implementação do algoritmo AnthillPart para o mapeamento de um grafo de tarefas de um programa I^3 em filtros; a análise do desempenho das aplicações mapeadas pelo AnthillPart e escalonadas pelo AnthillSched.

Como **trabalhos futuros** podemos destacar: o mapeamento de outros programas I^3; a implementação de um mecanismo de extração automática do grafo de tarefas; um compilador para a geração automática de código paralelo de aplicações Anthill etc.

7. Referências Bibliográficas

[1] Andrade, N., Cirne, W., Brasileiro, F., Roisenberg, P., "OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing", Job Scheduling Strategies for Parallel Processing, 2003.

[2] Batat, A., Feitelson, D., "Gang Scheduling with Memory Considerations", IEEE International Parallel and Distributed Processing Symposium, 2000, pp. 109-114.

[3] Beaumont, O., Boudet, V. and Robert, Y., "A Realistic Model and an Efficient Heuristic for Scheduling with Heterogeneous Processors", IEEE Heterogeneous Computing Workshop, 2002.

[4] Beynon, C. M., Ferreira, R., Kurc, T., Sussmany, A. and Saltz, J., "DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage Systems", IEEE Mass Storage Systems, 2000.

[5] Chapin, S.J. et al, "Benchmarks and Standards for the Evaluation of Parallel Job Schedulers", Job Scheduling Strategies for Parallel Processing, 1999, pp. 67-90.

[6] Feitelson, D. and Nitzberg, B., "Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860", Job Scheduling Strategies for Parallel Processing, 1995, pp. 337-360.

[7] Feitelson, D., Rudolph, L., "Evaluation of Design Choices for Gang Scheduling using Distributed Hierarchical Control", Journal of Parallel and Distributed Computing, 1996, pp. 18-34.

[8] Feitelson, D.G., "A Survey of Scheduling in Multiprogrammed Parallel Systems", Research Report RC 19790 (87657), IBM T. J. Watson Research Center, 1997.

[9] Paul E. Utgoff and Carla E. Brodley., "An Incremental Method for Finding Multivariate Splits for Decision Trees", Seventh International Conference on Machine Learning, Morgan Kaufman, 1990.

[10] Feitelson, D., Rudolph, L., "Metrics and Benchmarking for Parallel Job Scheduling", Job Scheduling Strategies for Parallel Processing, 1998, pp. 1-24.

[11]Feitelson, D., "Metric and Workload Effects on Computer Systems Evaluation", IEEE Computer, 2003, pp. 18-25.

[12] Franke, H., Jann, J, Moreira, J., Pattnaik, P., Jette, M., "An Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific", ACM/IEEE Conference on Supercomputing, 1999.

[13] Frachtenberg, E., Feitelson, D.G., Petrini, F. and Fernandez, J., "Flexible CoScheduling: Mitigating Load Imbalance and Improving Utilization of Heterogeneous Resources", 17th International Parallel and Distributed Processing Symposium, 2003.

[14] Góes, L. F. W., Martins, C. A. P. S., "Proposta e Desenvolvimento de um Algoritmo Reconfigurável de Escalonamento Paralelo de Tarefas", Dissertação de Mestrado, PUC-Minas, Belo Horizonte, Brasil, 2004.

[15] Góes, L. F. W., Martins, C. A. P. S., "Reconfigurable Gang Scheduling Algorithm", Job Scheduling Strategies for Parallel Processing, 2004.

[16] Góes, L. F. W. et al, "AnthillSched: A Scheduling Strategy for Irregular and Iterative I/O-Intensive Parallel Jobs", Job Scheduling Strategies for Parallel Processing, 2005.

[17] Nascimento, L.T., Ferreira, R., "LPSched – Escalonamento de Aplicações de Fluxos de Dados em Grids", Dissertação de Mestrado, UFMG, Belo Horizonte, Brasil, 2004.

[18] Neto, E. S., Cirne, W., Brasileiro, F., Lima, A., "Exploiting Replication and Data Reuse to Efficiently

Schedule Data-intensive Applications on Grids , Job Scheduling Strategies for Parallel Processing, 2004.

[19] Silva, F. A. B., Carvalho, S., Hruschka, E.R., A Scheduling Algorithm for Running Bag-of-Tasks Data Mining Applications on the Grid , EuroPar, 2004.

[20] Streit, A., A Self-Tuning Job Scheduler Family with Dynamic Policy Switching , Job Scheduling Strategies for Parallel Processing, 2002, pp. 1-23.

[21] Veloso, A., Meira, W., Ferreira, R., et al., Asynchronous and Anticipatory Filter-Stream Based Parallel Algorithm for Frequent Itemset Mining , European Conference on Principles of Data Mining and Knowledge Discovery, 2004.

[22] Wiseman, Y., Feitelson, D., Paired Gang Scheduling , IEEE Transactions Parallel and Distributed Systems, 2003, pp. 581-592.

[23] Zhang, Y., H. Franke, Moreira, E.J., Sivasubramaniam, A., Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques , IEEE International Parallel and Distributed Processing Symposium, 2000.

[24] Zhang, Y., Yang, A., Sivasubramaniam, A., Moreira, J., Gang Scheduling Extensions for I/O Intensive Workloads , Job Scheduling Strategies for Parallel Processing, 2003.

[25] Zhou, B. B., Brent, R. P., Gang Scheduling with a Queue for Large Jobs , IEEE International Parallel and Distributed Processing Symposium, 2001.

[26] Han, J. & Kamber, M., Data Mining: Concepts and Techniques , Morgan Kaufmann, 2001.

[27] Ferreira, R. A., W. Meira Jr., Guedes, D., Drumond, D. Anthill: A Scalable Run-Time Environment for Data Mining Applications , SBAC-PAD, 2005.