Modelo para a Exploração Eficiente de Paralelismo em Aplicações Grão Fino*

Epifanio Dinis Benitez†
Otávio Corrêa Cordeiro§
Eduardo Moschetta‡
Gerson Geraldo H. Cavalheiro

Programa de Pós-Graduação em Computação Aplicada
Centro de Ciências Exatas e Tecnológicas
Universidade do Vale do Rio dos Sinos
São Leopoldo – RS – Brasil
{epifanio, eduardom, cordeiro, gersonc}@exatas.unisinos.br

Resumo

Quando executamos aplicações com alto grau de paralelismo em arquiteturas de processamento de alto desempenho (PAD) multiprocessadas, a melhor maneira de obter desempenho é projetar a aplicação de acordo com os recursos disponíveis na arquitetura. Para isto, é necessário classificar a aplicação em função de sua granularidade: fina, média e grossa, e identificar através de sua concorrência com que modelo de programação concorrente ela será desenvolvida. Entre estes modelos estão o paralelismo de tarefa e o paralelismo de controle. Neste contexto, o Anahy [2] apresenta um modelo de execução eficiente para aplicações que estão caracterizadas por esses modelos. Serão discutidos os modelos de execução do Anahy e do padrão POSIX *para threads. Também é apresentada uma análise de desempenho destas bibliotecas de threads através de aplicações paralelas.*

1. Introdução

O desenvolvimento de aplicações paralelas está motivado pelo ganho de desempenho que poderá ser obtido quando se encontra disponível um ambiente para o PAD. Torna-se então necessária a análise dos seguintes elementos: i) recursos disponíveis na arquitetura, ii) concorrência da aplicação e iii) conhecimento sobre técnicas de programação concorrente. Estes elementos estão fortemente relacionados, e quando combinados de forma correta, oferecem um bom ganho de desempenho, porém perdendo quesitos de portabilidade da aplicação.

Os recursos da arquitetura alvo (memória, cpu, latência) podem ser um fator determinante quando está sendo definida a granularidade da aplicação. Em linhas gerais, a granularidade de um programa pode ser relacionada ao tamanho médio das operações envolvidas neste programa, podendo ser classificada em fina, média e grossa, acompanhadas de suas variantes (muito fina, muito grossa, etc). Na análise da aplicação, a granularidade e a identificação do elemento delimitador de sua concorrência determinarão na maioria dos casos o modelo de programação concorrente a ser utilizado.

Entre os modelos descritos segundo o *ponto de vista do programador* [11] estão o *task parallelism* (paralelismo de tarefa), que tem como característica a execução de diferentes atividades sobre diferentes conjuntos de dados, e o *pipelinig* (paralelismo de controle), que se caracteriza por realizar operações em estágios ou segmentos, onde a saída de um segmento é a entrada de outro.

O objetivo deste artigo é avaliar o Anahy como ferramenta para a programação paralela seguindo um modelo para a exploração eficiente do paralelismo presente em aplicações com granularidade fina e paralelismo de controle. Também é mostrado que este modelo é aplicável a sistemas distribuídos. Na próxima seção se encontram caracterizadas as bibliotecas de threads Anahy e do padrão *POSIX* para threads. Na Seção 3 encontram-se a descrição e a modelagem das aplicações utilizadas para a obtenção dos resultados de desempenho e as análises desses resultados. A Seção 4 contém a conclusão deste estudo.

* Projeto Anahy – CNPq (55.2196/02-9). Este trabalho foi parcialmente desenvolvido em colaboração com a HP Brasil P&D
† BIC/Fapergs
‡ ITI/CNPq
§ DTI/CNPq

2. Modelos de Execução

Esta seção descreve o modelo de execução das bibliotecas citadas anteriormente. Algumas figuras são utilizadas para visualizar estes modelos com o intuito de complementar o que está descrito. O Anahy, por ser um ambiente de programação/execução, tem seu modelo de execução relacionado a outros componentes, entre eles, os processadores virtuais (PVs), o núcleo executivo e um grafo de dependência de dados entre threads.

2.1. Pthreads

A biblioteca de threads do padrão *POSIX* (Pthreads) [1, 9], mais conhecida pela implementação da Linux Threads, apresenta um modelo de execução onde cada thread consiste em um novo fluxo de execução (Figura 1) (processo leve). A criação do novo fluxo implica na geração de um contexto que deverá ser guardado quando o fluxo não estiver em execução e recuperado quando o fluxo é escalonado, gerando um *overhead* na troca de contextos. O escalonamento de um fluxo de execução é realizado pelo sistema. Um fluxo em execução significa que somente uma atividade será realizada dentro de sua fatia de tempo. Quando o fluxo realizar por completo sua atividade, isto é, quando ele terminar, o resultado por ele produzido poderá ser recuperado somente uma vez.

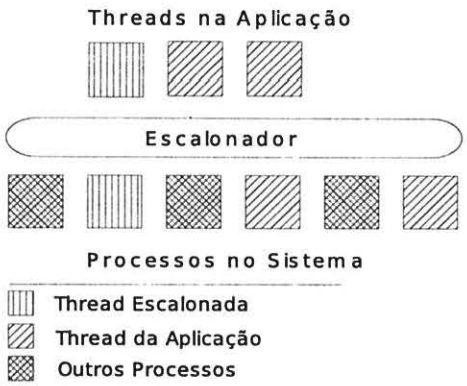

Figura 1. Modelo de execução das threads *POSIX*

Como visto na Figura 1, para cada thread criada na aplicação, uma thread é criada no sistema, portanto a estratégia utilizada é *one-to-one* (um para um).

2.2. Anahy threads

O Anahy é um ambiente de programação/execução que explora PAD em aplicações altamente paralelas [2]. O ambiente está formado basicamente por um grafo de dependência de dados entre threads, um núcleo executivo, processadores virtuais e também uma interface aplicativa (API) que segue a mesma sintaxe que a de Pthreads. No Anahy, existem dois tipos de 'processos': as threads que seguem o padrão *POSIX* (nível de sistema) e as threads Anahy (nível de aplicativo). Uma thread Anahy difere de uma Pthread na sua composição. Uma Pthread é composta por somente um fluxo de execução, já uma thread Anahy consiste no encapsulamento das tarefas explicitadas na aplicação (através da API). Este encapsulamento é realizado pelo ambiente Anahy seguindo a relação de paralelismo de controle existente entre elas. Como biblioteca de processos leves, Anahy oferece um modelo de programação eficiente para aplicações de granularidade fina e com paralelismo de controle. Abaixo segue uma breve descrição sobre o funcionamento de cada componente.

2.2.1. Grafo de dependência de dados Este grafo é construído a partir da concorrência explicitada na aplicação. Cada vértice do grafo é uma thread Anahy e as arestas representam a dependência entre as threads [12]. A execução destas começ a pelas folhas, chegando até a raiz quando todas as dependências forem satisfeitas. Portanto, a ordem execução no grafo segue no sentido da direita para a esquerda e de baixo pra cima (Figura 2).

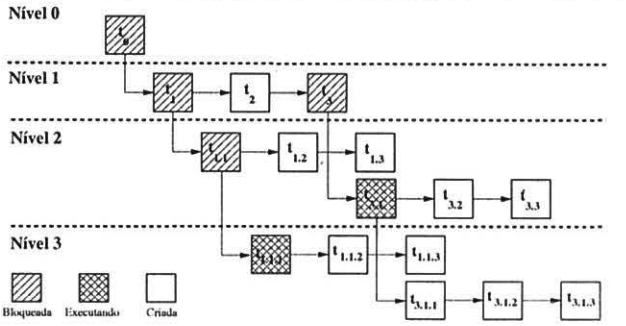

Figura 2. Grafo de dependência de dados entre threads

O sentido das flechas na Figura 2 indica a ordem de criação das threads e possibilita a visualização da dependência de dados entre estas. Cada dado produzido por uma thread é esperado pela thread que a criou, caracterizando a dependência.

2.2.2. Núcleo Executivo O escalonamento é realizado pelo núcleo executivo, respeitando a ordem de precedência do grafo. Aqui também é realizado o mapeamento da concorrência da aplicação na arquitetura alvo independentemente do paralelismo presente. Isto é possível pois o Anahy implementa escalonamento no nível de aplicativo, sendo

um grande diferencial frente a outras bibliotecas de threads. Este tipo de escalonamento busca explorar os recursos disponíveis na arquitetura ocupando-os com a execução do programa além de evitar o *overhead* que ocorre no momento da criação do fluxo de execução e durante a troca de contextos, tendo como custo de criação/escalonamento somente a leitura e escrita dos dados das threads em memória.

2.2.3. Processadores Virtuais Os processadores virtuais complementam o que é realizado no núcleo executivo. Cada processador virtual significa um processo leve em nível de sistema (*thread*) e esta camada segue o mesmo modelo de execução descrito na Seção 2.1. Quando um PV é escalonado pelo sistema, o núcleo executivo realiza o escalonamento das threads Anahy da aplicação durante a fatia de tempo que foi atribuída ao PV. Isto possibilita a execução potencial de todas as tarefas explicitadas na aplicação. O modelo de execução está representado na Figura 3.

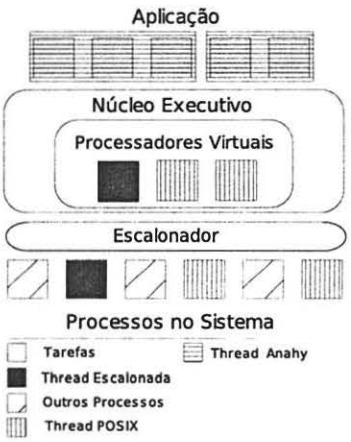

Figura 3. Modelo de execução do Anahy

3. Resultados de Desempenho

Nesta seção encontram-se descritas as aplicações que foram desenvolvidas para avaliar o modelo de execução do Anahy e os resultados dos testes de desempenho. As aplicações escolhidas foram: compactação de arquivos, convolução de imagens, ray tracer, fibonacci e um reconhecedor de imagens. O reconhecedor de imagens é uma aplicação paralela e distribuída que foi implementada com Pthreads e MPI [10] utilizando o modelo de execução do Anahy. As demais aplicações, em sua maior parte, se caracterizam pela presenç a de muitas atividades pequenas sobre um conjunto reduzido de dados. Para cada aplicação foram realizadas duas implementações: uma com Pthreads e outra com threads Anahy. Os experimentos foram realizados em duas arquiteturas: um Pentium IV de 1.8GHz com 512Mb de memória RAM (arquitetura mono-processada) e uma arquitetura bi-processada composta de 2 processadores XEON (Quad Xeon *Hyperthreaded*) de 2.8GHz operando com 1GB de memória RAM. Os testes foram compilados utilizando o GCC 3.2.2, rodando sobre o sistema operacional GNU/Linux, com kernel 2.4.20. Parte dos resultados se encontram em [4].

3.1. Compactação Paralela

Esta aplicação consiste na compactação de arquivos utilizando uma implementação paralela do algoritmo gzip. Para os testes de desempenho, foi utilizado um arquivo de 300MB que foi dividido em segmentos de mesmo tamanho. Tais segmentos foram atribuídos à atividades concorrentes seguindo o modelo de cada biblioteca de *threads*. Para que o resultado seja compatível com o algoritmo seqüencial do gzip é feito o cálculo de CRC 32 e a escrita seqüencial em disco. A leitura e escrita em disco foram desconsideradas na coleta dos resultados de desempenho.

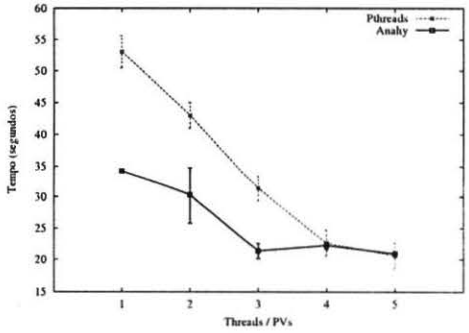

Figura 4. Gráfico comparativo dos tempos de execução do Gzip com o Anahy e Pthreads em arquiteturas bi-processadas

O gráfico da Figura 4 representa os tempos de execução utilizando o mesmo número de *threads*/PVs, e na implementação com o Anahy, a concorrência explorada na aplicação foi de 5 atividades concorrentes. Com um número baixo de *threads*/PVs, o *overhead* gerado por Pthreads é 50% maior que o gerado pelo Anahy (aproximadamente). A falta de determinismo na execução com o Anahy pode ser vista na execução com 3 PVs, onde a curva cai abruptamente e logo se normaliza já com 4 PVs. Com Pthreads, a execução se mostrou mais estável.

Na Figura 5, o gráfico mostra resultados interessantes nas diferentes implementações. Os resultados aqui apresentados foram tomados com uma Pthread, uma thread Anahy e um PV, acompanhados pela versão seqüencial.

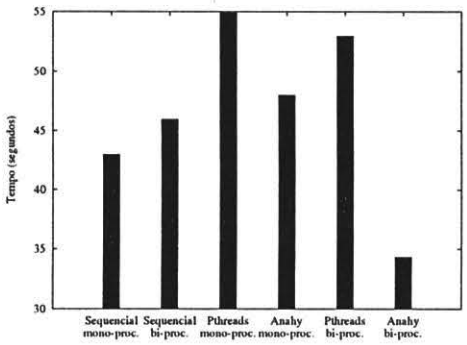

Figura 5. Gráfico comparativo dos tempos de execução do Gzip com o Anahy, Pthreads e seqüencial em arquiteturas mono e bi-processadas

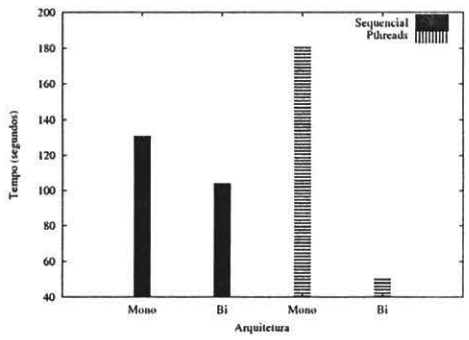

Figura 6. Gráfico comparativo dos tempos de execução do Ray-Tracer Seqüencial e Pthreads em arquiteturas mono e bi-processadas

A implementação com Pthreads não conseguiu alcanç ar um bom resultado com sua execução em uma arquitetura bi-processada, pois ficou quase duas vezes mais lenta que a implementação com o Anahy e muito próximo ao seu resultado alcanç ado na execução em arquitetura mono-processada. Com Anahy verficou-se um pequeno *overhead* na execução com arquitetura mono-processada sendo ainda um pouco mais eficiente que o resultado com Pthreads. O maior ganho de desempenho foi alcanç ado na execução em arquitetura bi-processada utilizando o Anahy, onde o tempo de processamento foi consideravelmente baixo frente as outras execuções.

3.2. Ray Tracer

O Ray Tracer aparece na literatura entre aplicações que são referência na área de programação paralela [3]. Esta aplicação se baseia em métodos de computação gráfica e sua implementação foi projetada de forma que uma cena fosse dividida em segmentos de mesmo tamanho para então calcular a iluminação e determinar a cor de uma região. Apesar desta divisão ser realizada da mesma forma, a carga de processamento de cada atividade concorrente é irregular, pois depende do número de objetos contidos em uma determinada região – isto é, quanto mais objetos, maior a carga de processamento. O cenário utilizado possui uma resolução de 800×800 e foi implementado com 256 atividades concorrentes.

Os gráficos das Figuras 6 e 7 mostram os tempos de execução de Pthreads, Anahy e seqüencial em arquiteturas mono e bi-processadas. Os resultados obtidos indicam que o Anahy é capaz de executar com sucesso o conjunto de atividades relacionadas a aplicação do Ray-Tracer. Na arquitetura mono-processada o Anahy realizou a execução sem introduzir *overheads* frente a execução seqüencial, já

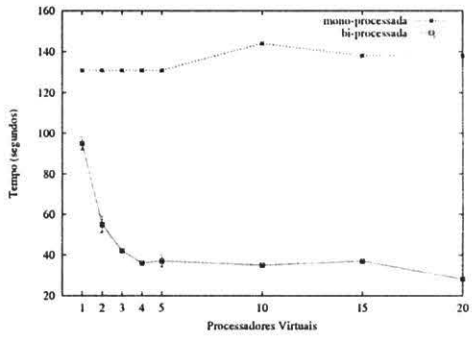

Figura 7. Gráfico comparativo dos tempos de execução do Ray-Tracer com Anahy em arquiteturas mono e bi-processadas

com Pthreads, devido ao seu modelo de execução, foi adicionado um *overhead* significativo no tempo de execução. No ambiente bi-processado o Anahy através de seu modelo de execução (visto na Seção 2.2) consegue obter mais de 50% de ganho de desempenho frente à execução com Pthreads a medida que aumenta o número de PVs.

3.3. Convolução Paralela

A operação de convolução é uma técnica de tratamento de imagens que consiste na multiplicação de um filtro [5] por uma imagem, onde ambos são representados através de matrizes. O processo de convolução consiste no deslizamento do filtro sobre a janela correspondente da imagem e dependendo do tipo do filtro (passa alta, passa baixa, etc) o resultado deve ser dividido pelo seu peso (soma de todos os elementos). Esta aplicação oferece um alto grau de paralelismo pois a convolução é realizada pixel a pixel permi-

tindo que a imagem seja dividida em um número arbitrário de blocos que irão estabelecer o nível de concorrência. A implementação com o Anahy utilizou o número padrão de PVs do ambiente (4). Os resultados dos testes realizados com esta aplicação se encontram nos gráficos das Figuras 8 e 9 e deve ser levado em conta que foram tomados os tempos completos da execução, isto é, o *overhead* das operações de leitura e escrita estão incluídos. Foi utilizado uma arquitetura bi-processada.

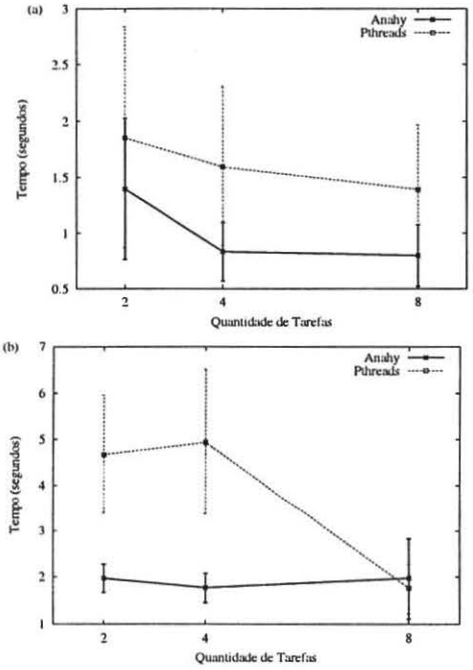

Figura 8. Convolução Paralela para imagens de dimensões a) 256 b) 512

Nos resultados com imagem de tamanho 256×256 e de tamanho 512×512 os resultados da implementação com Anahy foram superiores aos da implementação com Pthreads. Isto era esperado, pois o Anahy oferece um modelo eficiente para a situação onde existe um grande número de atividades concorrentes com pouca carga de processamento. Nas implementações com imagens de 1024×1024 e 2048×2048 a carga de processamento de cada atividade é maior e os tempos de execução das implementações estão próximos uns dos outros. Neste caso, as operações de leitura e escrita em disco representam uma parcela considerável no tempo total de execução da aplicação. Além disto, o Anahy reflete uma perda de desempenho pois o tamanho destas imagens aumenta a granularidade da aplicação.

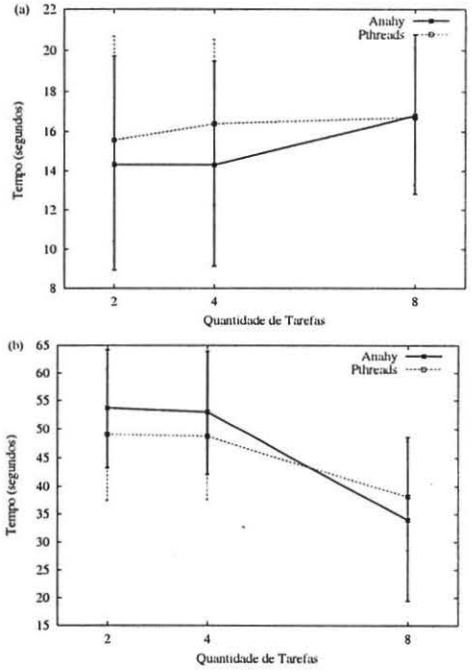

Figura 9. Convolução Paralela para imagens de dimensões a) 1024 b) 2048

3.4. Fibonacci

Os números de Fibonacci foram calculados de forma recursiva. Isto produz um elevado número de atividades concorrentes que exigem um grande número de sincronizações. A invocação recursiva da função Fibonacci gera a criação de uma nova atividade concorrente. Um fator ressaltante nesta aplicação é o fato da implementação com Pthreads limitar o cálculo a valores baixos da função pois cada atividade gera uma nova *thread*, e o número de *threads* suportadas por esta biblioteca é restringido. No Anahy este problema é contornado pelo fato de cada thread Anahy ser criada no nível de aplicativo, estando a concorrência limitada aos recursos da arquitetura. O fluxo de execução desta implementação está ilustrado na Figura 10.

Os resultados aqui apresentados comparam o *overhead* gerado pelas bibliotecas Pthreads e Anahy onde, neste, utilizou-se quatro (4) PVs. Observando o gráfico da Figura 11, é possível constatar que os *overheads* gerados na execução com Anahy são muito baixos em relação aos gerados na execução com Pthreads. A execução de uma aplicação com o Anahy é não determinística. Isto é importante, pois apesar dos resultados terem sido muito favoráveis em relação a Pthreads, em aplicações semelhantes ou até mesmo com o cálculo de um fibonacci maior, o ganho obtido pelo Anahy pode diminuir. Foram utilizados os

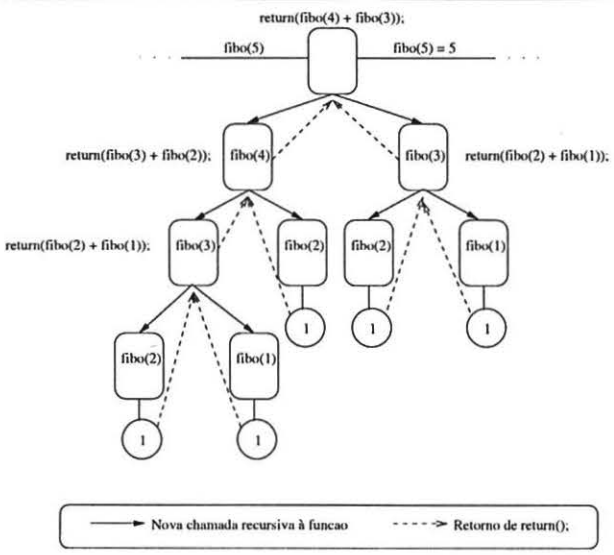

Figura 10. Fluxo de execução da implementação do Fibonacci

resultados de fibonacci 15 e 16, pois oferecem o maior grau de concorrência que pode ser explorado na implementação com Pthreads.

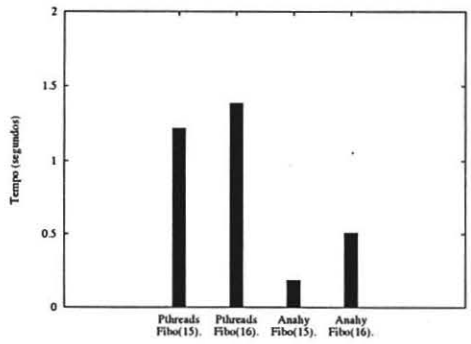

Figura 11. Gráfico comparativo dos tempos de execução do Fibonacci com Pthreads e Anahy em arquitetura mono-processada

3.5. Reconhecedor de Imagens

Esta aplicação possui um alto custo computacional e foi desenvolvida para validar o uso do modelo de execução do Anahy. Foram realizadas duas implementações: uma concorrente [8] [7] [6] (Phtreads e Anahy) e outra distribuída (MPI e Pthreads). Com isto é possível verificar a compatibilidade do modelo de execução com ferramentas clássicas de programação (MPI e *threads* POSIX). O Reconhecedor de Imagens é uma aplicação que representa um problema na área da computação gráfica: sistemas de recuperação de imagens (CBIR). O alto custo computacional está relacionado à pesquisa no banco de dados (de imagens) e ao algoritmo de *matching* utilizado para realizar as comparações de semelhanç a entre o fragmento procurado e as imagens do banco de dados. Os resultados devolvidos pelo algoritmo de *matching* obedecem uma margem de erro previamente estabelecida. As comparações foram realizadas utilizando os algoritmos de *matching* Ponto-a-Ponto, Histograma, Ponto a Ponto Escalas e Histogramas Escalas, no sistema de cores RGB utilizando fragmentos pequenos de imagem para a busca.

A implementação com Pthreads seguiu o modelo de execução do Anahy. Para isso foi necessário implementar um algoritmo de balanceamento de carga, um escalonador e um pool de *threads*. Esta modelagem está materializada na Figura 12 e pode ser descrita da seguinte forma: o banco de dados de imagens é distribuído entre as *threads* e cada imagem desse conjunto é processada por um conjunto de *threads* filhas, que produzem um resultado a ser recuperado posteriormente no momento da sincronização – caracterizando a dependência de dados entre as atividades concorrentes. O modelo de escalonamento utilizado é o centralizado, onde o mestre realiza a distribuição das imagens do banco e coleta os resultados parciais ao longo da execução. O pool de execução está composto por n *threads*, onde cada *thread* pega uma tarefa pronta para executar e após executá-la, recupera seus resultados. Este pool de threads está formado por 4 *threads*, pois na implementação com o Anahy utilizou-se 4 Pvs. A implementação com o Anahy brinda resultados que são comparados aos da implementação com Pthreads. Os resultados destas aplicações são mostrados em forma de tabela, e foram tomados os tempos de execução e também foi calculado o ganho (tempo em seqüencial/tempo em paralelo). O banco de dados utilizado possui 505 imagens de dimensões médias de 571 × 583 e foram selecionados fragmentos de tamanho pequeno (135 × 135), médio (272 × 272) e grande (540 × 540). Foram realizadas 20 execuções considerando desvios padrões menores de 10%.

Observando os resultados apresentados nas Tabelas 1 e 2, vemos que o ganho de desempenho obtido na implementação com Pthreads foi semelhante ao obtido na implementação com o Anahy. Esta aderência visível nos resultados aqui mostrados era esperada, pois o modelo de execução empregado na implementação do reconhecedor de imagens com Pthreads é igual ao modelo de execução do Anahy. Os resultados apresentados na execução mostraram-se mais favoráveis ao Anahy pois na implementação com Pthreads não foram realizadas as mesmas otimizações existentes no Anahy.

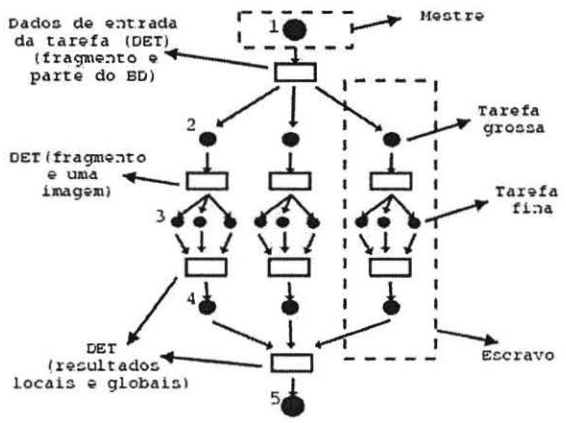

Figura 12. Modelo da implementação concorrente do Reconhecedor de Imagens

Algoritmo	Tempo	Ganho
Ponto a Ponto RGB	1.996 s	1,99
Histogramas RGB	611 s	1,92
P a P Escalas RGB	5.503 s	1,97
Histogramas Escalas RGB	1.120 s	1,93

Tabela 1. Tempos de execução paralela utilizando *threads* POSIX (Fragmento pequeno)

A versão distribuída do ambiente Anahy está em fase de desenvolvimento, por isso os resultados de desempenho não se encontram disponíveis. Como parâmetro de desenvolvimento, optou-se pela implementação distribuída do reconhecedor de imagens utilizando MPI e Pthreads. Isto foi motivado pela oportunidade de avaliar o que está sendo desenvolvido, baseando-se na aderência dos resultados apresentados pela solução concorrente com Pthreads aos resultados obtidos com o Anahy. Além da disponibilidade de um aglomerado de computadores bi-processados para realizar os testes. Para a comunicação entre os nodos do aglomerado utilizou-se o paradigma de troca de mensagens dis-

Algoritmo	Tempo	Ganho
Ponto a Ponto RGB	1.996 s	1,99
Histogramas RGB	593 s	1,98
P a P Escalas RGB	5.498 s	1,97
Histogramas Escalas RGB	1.106 s	1,95

Tabela 2. Tempos de execução paralela utilizando o Anahy (Fragmento pequeno)

Algoritmo	Tempo	Ganho
Ponto a Ponto RGB	488 s	8,14
Histogramas RGB	131 s	8,96
P a P Escalas RGB	1.394 s	7,77
Histogramas Escalas RGB	229 s	9,43

Tabela 3. Tempos de execução utilizando Pthreads e MPI (Fragmento pequeno)

ponibilizado pela biblioteca MPI, e para a comunicação entre os processadores dos nodos, foi utilizado Pthreads. Esta aplicação foi desenvolvida baseando-se no modelo de execução do Anahy e segue o mesmo esquema da figura citada nesta seção, com a diferenç a que o banco de dadosé dividido entre os nodos que compõem o aglomerado. A divisão das atividades e as sincronizações dos resultados são realizadas pelo nodo mestre.

Observando a Tabela 3, é possível constatar a eficiência do modelo de execução empregado, pois o ganho de desemepenho foi próximo a 10, número equivalente à quantidade de processadores presentes no aglomerado de computadores(5 nodos bi-processados). Existe a expectativa de que os resultados da implementação distribuída do Anahy possam apresentar um grau de ganho de desempenho próximo do que foi obtido com a implementação concorrente. Os resultados que não foram próximos ao resultado ótimo 'ganho 10' acontecem justamente onde a granularidade não se encontrava conforme com a capacidade de processamento do hardware, isto ocorre na utilização do algoritmo de *matching* Ponto-a-Ponto.

4. Conclusão

O desenvolvimento de aplicações altamente paralelas exige a utilização de ferramentas que auxiliem na exploração eficiente da arquitetura disponível. Um fator importante a ser observado no momento da escolha de alguma ferramenta de programação concorrente é seu modelo de execução. Neste trabalho foi possível observar através dos resultados apresentados, que um modelo de execução eficiente pode melhorar e muito o ganho de desempenho das aplicações.

As aplicações que foram desenvolvidas utilizando o modelo de execução disponibilizado através do ambiente de programação/execução Anahy apresentaram melhores resultados que as desenvolvidas com o modelo de Pthreads. A implementação do reconhecedor de imagens com a biblioteca de *threads* POSIX projetada com o modelo do Anahy, permitiu validar o uso deste modelo de execução aplicado sobre o modelo de execução de outra ferramenta de programação concorrente. Os resultados da implementação distribuída do reconhecedor de imagens estão sendo utili-

zados para comparar com os resultados obtidos com os testes que estão sendo realizados sobre o protótipo do Anahy para aglomerados.

O Anahy mostrou um comportamento uniforme e elegante nos resultados obtidos. Atualmente, além de estar sendo desenvolvida uma versão distribuída do Anahy, a equipe também está trabalhando sobre a estrutura dos processadores virtuais e do funcionamento do núcleo executivo, com o intuíto de realizar otimizações que possam oferecer resultados ainda mais favoráveis, e que também possam oferecer grau de 'determinismo' na execução.

Referências

[1] American National Standards Institute. *IEEE standard for information technology: Portable Operating Sytem Interface (POSIX). Part 1, system application program interface (API) — amendment 1 — realtime extension [C language].* IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Std 1003.1b-1993 (formerly known as IEEE P1003.4; includes IEEE Std 1003.1-1990). Approved September 15, 1993, IEEE Standards Board. Approved April 14, 1994, American National Standards Institute.

[2] G. G. H. Cavalheiro, L. C. V. Real, and E. C. Dall'Agnol. Uma biblioteca de processos leves para a implementação de aplicações altamente paralelas. In *Anais do IV Workshop em Sistemas Computacionais de Alto Desempenho*, São Paulo, Brasil, novembro 2003.

[3] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann, San Francisco, 1999.

[4] E. C. Dall'Agnol, L. C. Villa Real, E. D. Benitez, and G. G. H. Cavalheiro. Portabilidade na programação para o processamento de alto desempenho. In *IV Workshop em Sistemas Computacionais de Alto Desempenho*, São Paulo, Nov. 2003.

[5] R. C. Gonzalez. *Processamento de imagens digitais.* Addison-Wesley, São Paulo, 1993.

[6] E. Moschetta, F. S. Osório, and G. G. H. Cavalheiro. Reconhecedor de imagens concorrente. *Scientia*, 13(2), 2002.

[7] E. Moschetta, F. S. Osório, and G. G. H. Cavalheiro. Reconhecimento de imagens em aplicações críticas. In *III Worshop em Sistemas Compucionais de Alto Desempenho*, Vitória – ES, Oct. 2002.

[8] E. Moschetta, F. S. Osório, and G. G. H. Cavalheiro. Avaliação de desempenho na recuperação de imagens concorrente. In A. C. Yamin and J. L. V. Barbosa, editors, *4 Escola Regional de Alto Desempenho*, Pelotas – RS, Jan. 2004.

[9] Prentice-Hall, editor. *UNIX Network Programming,Networking APIs: Sockets and XTI.* Stevens W. Richard, Upper Saddle River, NJ, 1998.

[10] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: the complete reference.* MIT Press, Cambridge, MA, USA, 1996.

[11] J. Wiley and Sons, editors. *Parallel and Distributed Computing A Survey of Models, Paradigms, and Approaches.* Claudia Leopold, New York, 2001.

[12] A. Y. Zomaya. *Parallel and Distributed Computing Handbook.* McGraw-Hill, New York, 1996.