Huang, Y., Kintala, C.,Kolettis e N., Fulton, N. D. (1995) “Software Rejuvenation: Analysis, Module and Applications”, In: Proceedings of the 25th Symposium on Fault Tolerant Computer Systems, p. 381-390, Pasadena, CA.

Jain, R. “The Art of Computer Systems Performance Analysis: *Techniques for Experimental Design, Measurement, Simulation, and Modeling*.”, John Wiley & Sons, 1991.

Kraus, I. F. (2000) “Agent Technology in Performance and Availability Management”, 26th International Computer Measurement Group Conference, p. 139-152, Orlando, FL, USA.

Li, L., Vaidyanathan, K. e Trivedi, K. S. (2002) “An Approach for Estimation of Software Aging in a Web Server”, International Symposium on Empirical Software Engineering, p. 91-100, Nara, Japan.

Mosberger, D. e Jin, T. (1998) “Httperf – A Tool for Measuring Web Server Performance”, In First Workshop on Internet Server Performance, Madison, WI.

Netcraft (2004) “Web Server Survey Archive”, http://news.netcraft.com/ archives/ web_ server_survey.html, Março.

Shereshevsky, M., Cukic, B., Crowel , J., Gandikota , V. e Liu, Y. (2003) "Software Aging and Multifractality of Memory Resources", The International Conference on Dependable Systems and Networks (DSN-2003),p. 721-730, San Francisco, CA,USA.

Trivedi, K. S., Vaidyanathan, K. e GosevaPopstojanova, K. (2000) “Modeling and Analysis of Software Aging and Rejuvenation”, In: Proceedings of the 33rd Annual Simulation Symposium, p. 270-279, Los Alamitos, CA, IEEE Computer Society Press.

Vaidyanathan, K. e Trivedi, K. S. (1998) “A Measurement-based Model for Estimation of Resource Exhaustion in Operational Software Systems”, In: Proceedings of the Tenth IEEE International Symposium on Software Reliability Engineering, p. 84–93,Boca Raton, FL,USA.

Vaidyanathan, K., Harper, E. S., Hunter , W. e Trivedi., K. S. (2001a) “Analysis and Implementation of Software Rejuvenation in Cluster Systems”, ACM SIGMETRICS 2001/Performance 2001, Cambridge, MA, USA.

Vaidyanathan, K. e Trivedi, K. S. (2001b) “Extended Classification of Software Faults Based on Aging”, *Fast Abstracts*, In: Proceedings of the 12th International Symposium on Software Reliability Engineering, Hong Kong.

# Avaliação de Desempenho de Anahy em Aplicações Paralelas*

**Epifanio Dinis Benitez†, Evandro Clivatti Dall'Agnol‡, Lucas Correia Villa Real§, Marcelo Augusto Cardozo Junior ¶, Gerson Geraldo H. Cavalheiro**

[1]Programa Interdisciplinar de Pós-Graduação em Computação Aplicada
Universidade do Vale do Rio dos Sinos
Avenida Unisinos, 950 – 93.022-000 São Leopoldo, RS

{epifanio,ecd,lucasvr,mcardozo,gersonc}@exatas.unisinos.br

***Abstract.*** *Several programming/execution environments seek to exploit hardware parallelism; this consists of efficient use of resources with shared memory. Anahy, as a library for light processes, is a tool for High Performance Computing (HPC) that offers an efficient model for the development of parallel applications. This is possible due to virtual processors, task graphs and application level scheduling. Code portability, dependency graph, kernel and Application Programming Interface (API) are also discussed. This article benchmarks the performance of Anahy through parallel applications in multi-processed architectures.*

***Resumo.*** *Diversos ambientes de programação/execução buscam aproveitar o paralelismo disponível em hardware; isto consiste na exploração eficiente de recursos em ambientes com memória compartilhada. Anahy, como biblioteca de processos leves, é uma ferramenta para o PAD (processamento de alto desempenho) que oferece um modelo eficiente para a implementação de aplicações altamente paralelas. Isto é possível através de uma estrutura baseada em processadores virtuais, grafo de tarefas e escalonamento ao nível de aplicativo. Assuntos como a portabilidade de código, grafo de dependências, núcleo executivo e a API disponível, também serão abordados. Este artigo avalia o desempenho de Anahy através de aplicações paralelas em arquiteturas multiprocessadas.*

*Projeto Anahy – CNPq (55.2196/02-9)
†BIC/FAPERGS.
‡ITI/CNPq.
§ITI/CNPq
¶DTI/CNPq

## 1. Introdução

Nos últimos anos, o surgimento de aplicações complexas e o número cada vez maior de dados a serem processados originaram a necessidade de maior poder computacional para reduzir o tempo deste processamento, levando estudantes e grupos de pesquisa a necessitarem de arquiteturas para o PAD. De custo relativamente baixo, comprovado pela incidência de aglomerados na lista das 500 máquinas[1] mais potentes em operação, os aglomerados de computadores (*clusters*) e os SMPs (*Symmetric Multi-Processors*) supriram esta necessidade, gerando sua popularização. Estas arquiteturas, quando juntas (*cluster de SMPs*), oferecem dois níveis de paralelismo a serem explorados: o nível intra-nó e o nível entre-nodos.

A utilização dessas arquiteturas requer o estudo do funcionamento das ferramentas disponíveis para a programação paralela e distribuída. Portanto, para conseguir ganho de desempenho frente à execução seqüencial através de arquiteturas paralelas multiprocessadas, além de estudar as ferramentas que exploram os dois níveis, é necessário identificar as regiões paralelizáveis da aplicação para que o mapeamento de sua concorrência nas unidades de processamento e cálculo (processador e memória) seja feito de maneira eficiente na arquitetura disponível. No entanto, na maioria dos casos, a concorrência da aplicação é maior que o paralelismo real da arquitetura impedindo que o mapeamento seja realizado de forma direta.

Na exploração dos níveis intra-nó e entre-nodos há uma questão importante a ser considerada, determinante para a exploração eficiente dos recursos: a distribuição de carga computacional (balanceamento de carga) entre os diferentes processadores e o compartilhamento de dados entre os nodos.

Utilizando ferramentas convencionais para a programação paralela ou ditribuída, tais como MPI (*Message Passing Interface*) [Snir et al., 1996] [Pacheco, 1997], Sockets (descritor de arquivos padrão UNIX) [Stevens, 1998], e multi-programação leve (*threads*) [Cohen et al., 1998], o programador deve explicitar a concorrência da aplicação, determinando o número de tarefas concorrentes de acordo com a arquitetura alvo para que a distribuição das tarefas e dos dados seja realizada de forma correta entre os processadores e módulos de memória presentes na arquitetura. Determinar a concorrência de acordo com os recursos de uma arquitetura específica impede a portabilidade da aplicação.

Este trabalho apresenta resultados de desempenho obtidos utilizando Anahy, uma biblioteca de processos leves para aplicações altamente paralelas[Cavalheiro et al., 2003]. O objetivo de Anahy é permitir que o programador possa explicitar a concorrência de sua aplicação sem preocupar-se com a arquitetura na qual o programa poderá vir a ser executado; isto permite ao programador explicitar a concorrência da aplicação através de código preciso.

Na próxima seção, Anahy é apresentado em suas características principais, tais como modelo de execução, núcleo executivo, processadores virtuais e interface, entre outros. O restante deste artigo está organizado da seguinte forma: a Seção 3 descreve as aplicações desenvolvidas para a obtenção de resultados de desempenho, acompanhadas de seus respectivos resultados, e na Seção 4 uma conclusão é apresentada.

[1] Ver http://www.top500.org

## 2. Anahy

Esta seção resume os aspectos relevantes de Anahy, concentrando-se em assuntos que diferenciam o Anahy das ferramentas convencionais de programação concorrente. Anahy é um ambiente de programação/execução que oferece uma API (*interface de programação*) que torna possível explicitar a concorrência da aplicação através de tarefas, as quais são armazenadas em um grafo de acordo com a dependência de execução que existe entre elas, obedecendo uma classificação baseada nos níveis dessas dependências. O escalonamento das tarefas é realizado pelo núcleo executivo ou ainda kernel do Anahy, de acordo com o algoritmo e as políticas de balanceamento de carga que foram impostas.

### 2.1. Portabilidade

A maioria dos ambientes paralelos não aborda a questão da portabilidade, enquanto outros o fazem com um alto custo implicado através de uma camada de abstração, acarretando na perda de desempenho. Anahy trata da questão da portabilidade com o diferencial de preocupar-se com o desempenho e portabilidade das aplicações, além de permitir que um programa desenvolvido com o ambiente Anahy possa ser executado em diferentes configurações de aglomerados. O modelo de implementação de Anahy está baseado na dissociação entre a concorrência da aplicação e o paralelismo disponível na arquitetura alvo. Assim, o programador pode definir o número de atividades concorrentes de sua aplicação desconsiderando os recursos de processamento disponíveis.

A Figura 1 mostra a estrutura de Anahy, destacando as diferentes camadas que viabilizam a portabilidade da execução de uma aplicação, concorrente sobre uma arquitetura paralela composta de nodos multiprocessados. As camadas são discutidas a seguir.

A API de Anahy oferece recursos (*threads Anahy*) que podem ser utilizados na implementação de programas com um alto grau de paralelismo. O modelo proposto tomou como parâmetros alguns critérios considerado adequados e eficientes para linguagens de programação concorrentes. Um desses critérios consiste em facilitar o gerenciamento e sincronismo de um grande número de fluxos de execução e as comunicações realizadas entre estes fluxos.

Sendo a portabilidade de desempenho parte do objetivo de Anahy, adotou-se para o núcleo executivo um modelo [Cavalheiro et al., 1998] [Cavalheiro, 2001] que permitisse o suporte à inserção de diferentes algoritmos de escalonamento [Casavant and Kuhl, 1988], tanto para a execução paralela como para a execução distribuída de programas. Essa modelagem permite que diferentes critérios ou técnicas de balanceamento de carga possam ser criados de acordo às características da aplicação e da arquitetura alvo.

A portabilidade de código também é uma das preocupações de Anahy, portanto, utilizando o ambiente Anahy, o programador não precisa reescrever a aplicação quando há mudanças na arquitetura onde ela será executada; uma vez explicitada a concorrência, Anahy se encarrega de mapeá-la na arquitetura disponível. Os módulos dependentes de arquitetura foram implementados utilizando ferramentas padrões em aglomerados de computadores, que são as *threads* do padrão POSIX para nodos multiprocessados (SMPs) e MPI ou *sockets* para a comunicação de dados entre os nodos. Estas ferramentas possibilitam a exploração de dois níveis de paralelismo presentes em uma aglomerado de

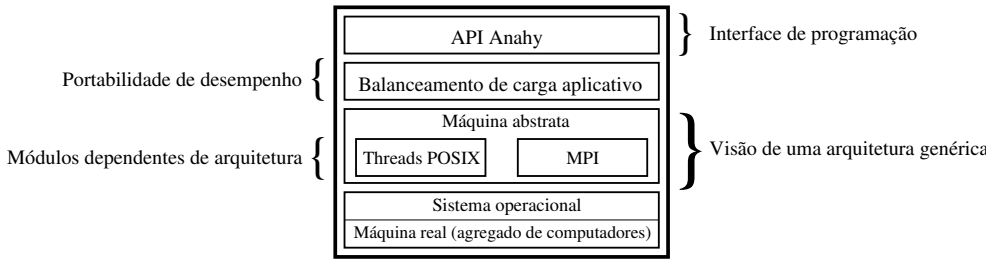


**Figura 1: Esquema estruturado do ambiente de programação Anahy**

nodos multiprocessados: intra (comunicação em memória compartilhada) e entre-nodos (comunicação via hardware de rede).

## 2.2. Núcleo Executivo

O núcleo executivo é responsável pelo escalonamento, que se encarrega de explorar a arquitetura que está sendo utilizada e também suporta a introdução de algoritmos e políticas de balanceamento de carga. Ele tem como função mapear as tarefas realizadas em paralelo de acordo com o paralelismo real disponível na arquitetura. Em ambientes multiprocessados é possível observar que o escalonamento é realizado em dois níveis: sistema e aplicativo. O escalonamento no nível de sistema busca explorar os recursos computacionais ocupando-os com a execução do programa, no nível de aplicativo a preocupação está em distribuir a carga computacional exigida pelo programa sobre os diferentes recursos disponibilizados pela arquitetura. No ambiente de execução Anahy o escalonamento é realizado a nível de aplicativo (*cf.* [Cavalheiro, 1999], cap.2). Na sub-seção 2.2.1, é explicado em detalhes como é realizado este escalonamento.

### 2.2.1. Escalonamento em Anahy

O escalonamento em Anahy é realizado através da manipulação de um grafo de precedência de execução que pode ser visto na Figura 2. Este grafo apresenta as dependências entre as tarefas; é possível visualizar esta dependência analisando cada nível da Figura 2. Podemos ver que no nível 0 existe somente uma tarefa, isto é, é o "pai" de todas as tarefas nos níveis seguintes e a continuidade de sua execução se dará quando todas as tarefas "filhas", ou seja, dos níveis 1,2 e 3 estiverem terminado. Então quando a tarefa "pai" $\mathcal{T}0$ solicitar o resultado de sua tarefa filha $\mathcal{T}1$, na verdade $\mathcal{T}0$ estará solicitando o último resultado recebido pela tarefa filha $\mathcal{T}1$ . A ordem de execução neste grafo é da direita para a esquerda e de baixo para cima, o que facilita a visualização da dependência de dados entre as tarefas. Cada tarefa presente no grafo está em distintos tons de cinza, para que sejam identificadas as tarefas que estão **executando**, as que estão **bloqueadas** e as que foram criadas **criadas**.

Basicamente, o algoritmo de escalonamento consiste em gerenciar quatro listas de tarefas: a primeira contém as tarefas **prontas**, ou seja, aptas a serem executadas; a segunda as tarefas **terminadas** cujos resultados ainda não foram solicitados, ou seja, a

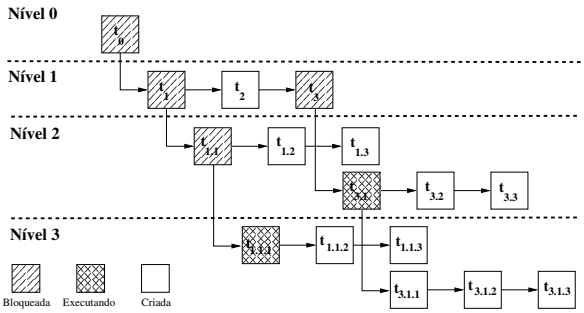


**Figura 2: Grafo de criação de threads Anahy e relações de precedência de execução**

operação de *join* sobre estas tarefas ainda não foi realizada; a terceira contém a lista de tarefas **bloqueadas** e a quarta, as tarefas **desbloqueadas**.

O esquema de funcionamento do algoritmo de escalonamento está representado no seguinte exemplo: a arquitetura de execução é uma máquina monoprocessada; o processador, inicialmente vazio, recebe do escalonador a tarefa $\mathcal{T}1$ da lista de tarefas prontas e inicia sua execução; as operações $\mathcal{T}1$ são realizadas normalmente e as operações que envolvem concorrência na aplicação são administradas pelo escalonador. Caso uma operação *fork* seja executada, uma nova tarefa $\mathcal{T}2$ é criada e armazenada na lista de tarefas prontas e $\mathcal{T}1$ continua sua execução normalmente. Caso algum resultado for requerido, isto é, se for realizado um *join*, por exemplo, em $\mathcal{T}2$, $\mathcal{T}1$ termina e uma nova tarefa $\mathcal{T}3$ é criada ($\mathcal{T}1$ torna-se $\mathcal{T}3$, pois a partir do join em $\mathcal{T}2$, um novo conjunto de dados passa a ser a entrada de $\mathcal{T}1$), sendo o início de seu código as operações ou instruções que sucedem o join, o estado inicial de $\mathcal{T}3$ é bloqueado, pois $\mathcal{T}3$ somente poderá executar quando $\mathcal{T}2$ terminar sua execução, tendo produzido os dados necessários e solicitados por $\mathcal{T}3$: isto é uma relação de dependência entre as tarefas, e é denotada por $\mathcal{T}2 \prec \mathcal{T}3$. A lista de tarefas prontas e a de tarefas terminadas são atualizadas com a retirada da tarefa $\mathcal{T}2$ da lista de tarefas prontas e terminadas. Após o término de $\mathcal{T}2$, $\mathcal{T}3$ é desbloqueada e iniciada, tendo como dados de entrada os dados que foram produzidos por $\mathcal{T}2$. Se existe uma grande quantidade de dependência de execução entre as tarefas, satisfazer as relações de precedência pode ser uma operação recursiva.

Caso a arquitetura for multiprocessada, dotada de dois ou mais processadores, cada processador executa o mesmo algoritmo descrito anteriormente, mas há um porém: caso uma tarefa $\mathcal{T}i$ solicitar o resultado ou executar um *join* na tarefa $\mathcal{T}j$, duas situações poderão ocorrer; ou a tarefa $\mathcal{T}j$ está em execução ou ela já terminou. Caso ela já tenha terminado, os dados que resultaram de seu processamento são recuperados e passados à tarefa $\mathcal{T}i$. A tarefa $\mathcal{T}i{+}1$ é executada e a tarefa $\mathcal{T}j$ é retirada da lista de tarefas terminadas. Se $\mathcal{T}j$ está sendo executada a tarefa $\mathcal{T}i + 1$ permanece bloqueada e o escalonador busca uma nova tarefa na lista de tarefas prontas e repassa ao processador. A tarefa $\mathcal{T}i + 1$ será desbloqueada quando a tarefa $\mathcal{T}j$ terminar seu fluxo de execução.

### 2.3. Processadores Virtuais (PV)

Portabilidade de desempenho de programas é um dos objetivos de Anahy. Para que isso seja possível o módulo de escalonamento deve ser independente à execução da aplicação.

Com escalonamento independente, caso seja necessário manipular de outra forma as tarefas devido a mudanças de arquitetura ou políticas de balanceamento de carga, o código do aplicativo não precisa ser modificado. O núcleo executivo limita o número de atividades concorrentes da aplicação que estão em execução simultânea em função dos recursos computacionais disponíveis na arquitetura. O limite é dado pela quantidade de processadores virtuais ativos no núcleo.

O programador, ao utilizar PVs em sua aplicação, tem a visão de uma arquitetura *virtual* multiprocessada dotada de memória compartilhada. Essa visão, que está representada na Figura 3, é explicada como segue: a arquitetura real ou destino está formada por um conjunto de nodos de processamento que possuem sua memória e processadores. Entre os componentes da arquitetura *virtual*, estão um conjunto de PVs alocados sobre a arquitetura do nodo e uma memória que é compartilhada entre eles. A arquitetura real impõe as limitações ao número de PVs e ao tamanho da memória compartilhada (utilizada para a comunicação entre os PVs através de instruções providas pela arquitetura virtual). Cada PV possui a capacidade de executar um fluxo de execução seqüencial que foi atribuído a ele: enquanto ele estiver executando, nenhuma sinalização será tratada por ele. Caso o PV estiver ocioso, ele pode ser reativado caso haja uma alguma atividade apta a ser executada. Os dados locais às atividades do usuário que serão executadas são guardadas em um espaço próprio do PV.

Através dos PVs o programador pode definir um número de atividades concorrentes que ultrapasse o paralelismo real da arquitetura disponível. O núcleo executivo se encarrega de ativar a execução das atividades sobre os recursos de processamento virtuais disponíveis.

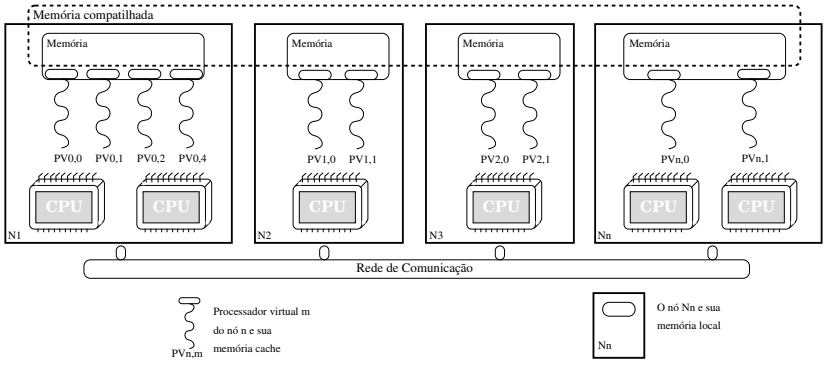


**Figura 3: Modelo lógico e físico da arquitetura de suporte à Anahy**

Para facilitar o entendimento segue um exemplo: o programador explicita 4 tarefas na aplicação, 4 processadores virtuais em Anahy e a arquitetura disponível é multiprocessada dotada de dois processadores. Durante a execução, quando a aplicação ganhar sua "fatia" de tempo para executar suas tarefas, serão executadas potencialmente 4 tarefas de forma simultânea, isto é, apesar da arquitetura prover somente dois processadores, o escalonador irá alternar as tarefas entre os processadores virtuais dentro da "fatia" de tempo da aplicação. Caso o número de tarefas for maior que o número de PVs, o algoritmo de escalonamento determinará como será a execução das tarefas. Para cada processador virtual (*thread de nível aplicativo*) criado, uma *thread* sistema é criada. Será visto na

Seção 3 que o uso de PVs conduz a um bom ganho de desempenho tanto em arquiteturas monoprocessadas como em arquiteturas bi-processadas.

### 2.4. Interface aplicativa (API)

Através de sua interface de programação, Anahy provê serviços que oferecem ao programador mecanismos para explorar o paralelismo de uma arquitetura multiprocessada dotada de memória compartilhada (*SMPs*), permitindo que a sincronização das tarefas concorrentes da aplicação seja realizada implicitamente via troca de dados entre elas. Os serviços suportados consistem em operações de *fork* e *join*, disponibilizando ao programador uma interface de programação próxima ao modelo oferecido pela multiprogramação baseda em processos leves (criação e sincronização com o término de fluxos de execução).

A função *fork* consiste na criação de um novo fluxo de execução, sendo o código a ser executado definido por uma função $\mathcal{F}$ presente no corpo do programa. Esse operador retorna um identificador *pid* ao novo fluxo criado. No momento da invocação da operação fork a função a ser executada deve ser identificada e os parâmetros $\mathcal{X}$ necessários à sua execuçao devem passados. O programador não tem nenhuma informação sobre o momento em que este fluxo será disparado: sabe-se que após seu término, um resultado $\mathcal{Y}$ será produzido, ou seja, $\mathcal{Y} = \mathcal{F}(\mathcal{X})$.

Com a operação *join* é realizada a sincronização com o término da execução de um fluxo, onde o fluxo a ser sincronizado deve ser identificado. Essa operacão permite que um fluxo bloqueie, aguardando o término de outro fluxo, de forma a recuperar os resultados produzidos. Com o uso das operações de sincronização (*fork* e *join*) realizadas no interior de um fluxo de execução:w pode ser identificada alguma atividade que possa ser executada de forma concorrente.

Deve ser observado o fato de que operadores de sincronização não estão presentes. Isto é justificado pelo aumento da capacidade de execução seqüencial do programa sem esses operadores. Portanto, o resultado da aplicação concorrente é igual ao resultado da execução seqüencial do mesmo código, o que facilita o desenvolvimento e a depuração da aplicação.

#### 2.4.1. Sintaxe utilizada

Por questões de compatibilidade, o ambiente Anahy está sendo desenvolvido com o padrão POSIX para *threads*, portanto, os serviços e estruturas oferecidos por Anahy são um subconjunto das oferecidas pelo padrão POSIX. Por questões de desempenho operações de sincronização, tais como semáforos e variáveis de condição, não foram implementadas, mas estuda-se a entrada delas em um novo conjunto de serviços oferecidos por Anahy.

O "corpo" de um fluxo de execução é escrito como uma função em linguagem C. No trecho de código abaixo se encontra uma declaração de corpo de função:

```
void* func( void * in ) {
    /* código da função */
    return out;
```

```
}
```

A função *func* é o fluxo de execução que será executado, que recebe como parâmetro a variável *in* que corresponde ao endereço de memória onde se encontram os dados de entrada da função. Os dados resultantes deste fluxo de execução são retornados em (*return out*), onde *out* aponta para a área de memória que contém os dados de retorno da tarefa.

O sincronismo permitido em Anahy está formado basicamente por operações de *fork* e *join*, que correspondem as operações de criação e espera por término de *threads* respectivamente. Os operadores do padrão POSIX *pthread_create* e *pthread_join* são substituídos pelo seu equivalente em Anahy, e estão exemplificados abaixo por:

```
int athread_create( athread_t *th,
                    athread_attr_t *atrib,
                    void * (*func) (void *),
                    void *in );
int athread_join( athread_t th, void **res);
```

A explicação do código é bastante simples: athread_create cria um novo fluxo de execução para a função *func*; os dados de entrada desta função estão presentes no endereço de memória *in*. O valor de *th* consiste em um identificador único e pode ser utilizado como referência ao fluxo criado. Os atributos são passados em *atrib*. Em se tratando de uma *thread* (tarefa) Anahy, existe alguns atributos extras não presentes no padrão POSIX, entre eles: os atributos *athread_attr_setjoinnumber*, que e *athread_attr_getjoinnumber* servem para explicitar e recuperar a qauntidade de operações join podem ser realizadas sobre uma determinada tarefa; *athread_attr_setdatalen* e *athread_attr_getdatalen* servem para dizer e recuperar o tamanho dos dados das tarefas, entre outras que estão sendo experimentadas para o envio e recebimento de tarefas entre nodos de um aglomerado de computadores.

A operação *athread_join* serve para sincronizar tarefas, e recebe como parâmetro a identificação do fluxo com quem será realizada a sincronização, enquanto o endereço *res* aponta para um endereço de memória que irá conter os dados retornados pelo fluxo. Ambos operadores podem retornar um código de erro.

## 3. Resultados de Desempenho

Para avaliar o desempenho de Anahy foram implementados quatro algoritmos fazendo uso de *threads* POSIX e do ambiente Anahy. As aplicações que foram escritas são um *Ray-Tracer*, onde um cenário descrito através de objetos geométricos é renderizado, uma implementação paralela para compactação de arquivos baseada na *Zlib*, e o *ConvoP*, onde está implementado um algoritmo paralelo para tratamento de imagens: a convolução de imagens. A quarta e última implementação teve por objetivo avaliar o uso de Anahy em uma situação em que a aplicação possui um elevado grau de concorrência e um grande número de sincronizações: o cálculo de um valor na série de Fibonacci. Parte destes resultados encontram-se em [Dall'Agnol et al., 2003].

Para a realização dos experimentos foram utilizadas duas arquiteturas: um Pentium IV de 1.8GHz com 512Mb de memória RAM (arquitetura mono-processada), uma

arquitetura bi-processada composta de dois processadores XEON (Quad Xeon *Hyper-threaded*) 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.

### 3.1. Ray-Tracer

O *Ray-Tracer* foi implementado com base em métodos de computação gráfica e consiste em dividir o cálculo da iluminação e determinação de cor de uma determinada região de uma cena entre diferentes tarefas. A estratégia utilizada foi *split-compute-merge* e consiste nas seguintes etapas: na etapa *split*, a imagem é sub-dividida em uma determinada quantidade de linhas consecutivas de acordo ao número de *tarefas*, que são os dados de entrada de cada *tarefa*; na etapa *compute*, é realizado o processamento em si, e na etapa *merge* o resultado é retornado à uma tarefa responsável pela escrita da imagem em disco. O número de *tarefas* a serem criadas foi fixado em 256, utilizando um ambiente de descrição de cenário com resolução de $800 \times 800$.

Apesar da estratégia adotada descrita acima dividir os dados de forma equalitativa entre as *tarefas*, a carga de processamento atribuída à elas é irregular, ou seja, a carga computacional atribuída a cada *tarefa* depende dos objetos contidos nesse conjunto de linhas: isto é, quanto mais objetos, maior a quantidade de cálculo a ser realizado.

Na Figura 4, encontra-se o esquema das *tarefas* criadas pela aplicação. Observando a figura, vemos que não existe precedência de execução entre as tarefas a não ser a sincronização dos dados conseqüente pelo processamento do conjunto de linhas.

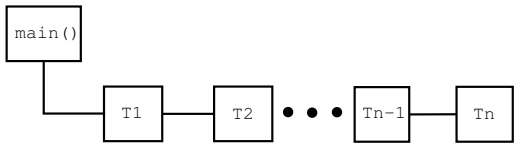


**Figura 4: Modelo de tarefas independentes: Ray-Tracer, agzip e ConvoP**

Os resultados da execução deste aplicativo encontram-se nas Tabelas 1, 2, 3 e 4. Nestas tabelas são apresentados os tempos, em segundos, e os desvios padrões obtidos para 100 execuções da aplicação em diferentes situações. As colunas PVs (processadores virtuais) indicam o grau de concorrência da aplicação suportado pelo núcleo executivo de Anahy.

Os resultados obtidos indicam que o ambiente Anahy é capaz de manipular com sucesso a quantidade de trabalho associada ao modelo de paralelismo do Ray-Tracer. Realizando uma análise comparativa entre as tabelas é possível observar que: em uma arquitetura mono-processada, Anahy permitiu uma execução sem introduzir *overheads* em relação à execução seqüencial. Chegou-se a esta conclusão comparando os resultados da Tabela 1 (execução seqüencial) com os da Tabela 3(execuções com Anahy). Já a execução da implementação utilizando PThreads, introduz *overhead* significativo na execução em arquitetura mono-processada: isto pode ser visto nos resultados apresentados na Tabela 2. Na execução em um ambiente bi-processado, observa-se que Anahy permite obter um maior ganho de desempenho em relação à implementação com PThreads; isto ocorre pela

existência dos PVs, que restringem a geração de custos adicionais na criação de *threads* gerenciadas pelo sistema. Comparando os resultados das tabelas 1, 2 e 4, observa-se que com PThreads o ganho de desempenho foi de aproximadamente 50%, mas analisando a Tabela 4, à medida que aumenta o número de PVs, vemos que o ganho de desempenho é superior a 50% na maioria dos casos. Isto ocorre pois como foi descrito na Seção 2.3, o número de tarefas executando é potencialmente maior que o paralelismo presente na arquitetura.

**Tabela 1: Tempos em segundos para a execução (seqüencial) do Ray-Tracer**

| Arquitetura | Média | Desvio Padrão |
|---|---|---|
| Mono-proc | 131,615 | 0,126 |
| Bi-proc | 104,922 | 7,173 |

**Tabela 2: Tempos em segundos do Ray-Tracer com PThreads**

| Arquitetura | Média | Desvio Padrão |
|---|---|---|
| Mono-proc | 181,799 | 0,115 |
| Bi-proc | 50,646 | 0,460 |

**Tabela 3: Tempos em segundos para a execução do Ray-Tracer em Anahy em uma arquitetura mono-processada.**

| PVs | Média | Desvio Padrão |
|---|---|---|
| 1 | 131,552 | 0,124 |
| 2 | 131,542 | 0,118 |
| 3 | 131,550 | 0,127 |
| 4 | 131,543 | 0,122 |
| 5 | 131,533 | 0,120 |
| 10 | 144,066 | 0,105 |
| 15 | 138,328 | 0,116 |
| 20 | 138,504 | 0,122 |

**Tabela 4: Tempos em segundos do Ray-Tracer em arquitetura bi-processada com Anahy**

| PVs | Média | Desvio Padrão |
|---|---|---|
| 1 | 95,180 | 3,3016 |
| 2 | 55,229 | 4,5509 |
| 3 | 42,216 | 1,4201 |
| 4 | 36,781 | 0,5676 |
| 5 | 37,452 | 3,1633 |
| 10 | 35,760 | 0,3675 |
| 15 | 37,627 | 0,8435 |
| 20 | 28,923 | 0,7287 |

### 3.2. Compactação de arquivos

Outro tipo de aplicação em que o uso de alto desempenho torna-se desejável é a compactação de arquivos. A implementação realizada divide o arquivo a ser compactado em *streams* de mesmo tamanho e as distribui entre as tarefas da aplicação desenvolvida. Em cada *stream* é feito o cálculo de CRC-32 e sua compactação. A escrita em disco é seqüencial e pré-determinada para manter a compatibilidade com aplicativos como o GZip. Este mesmo algoritmo foi implementado fazendo-se uso das *threads* POSIX. A Figura 4 representa o esquema de execução desta aplicação.

Para os testes seqüenciais foi utilizada uma implementação já existente do GZip. As Tabelas 5, 6, 7, 8 e 9 mostram os resultados da compactação de um arquivo binário de 300MB com os tempos em segundos obtidos de 100 execuções em cada caso. O acesso a arquivo não foi considerado na tomada de tempo por poder variar de acordo com a arquitetura. O algoritmo seqüencial mantém um histórico da taxa de compressão de todo o arquivo, o que resulta em uma complexidade maior em relação à versão concorrente.

Índices satisfatórios de desempenho foram obtidos com 5 a 10 *tarefas* executando sobre a arquitetura bi-processada com o uso de PThreads que pode ser visualizado nas Tabelas 6 e 8, demonstrando um ganho de desempenho onde o mapeamento de atividades concorrentes encontra-se adaptado à capacidade de processamento da arquitetura.

Nas execuções sobre o ambiente Anahy destaca-se uma característica da implementação de seu núcleo: de fato não é criada nenhuma *thread*, não existindo *overheads* de execução. Como conseqüência tem-se que os tempos obtidos para a execução com 1 tarefa e 1 PV são inferiores aos tempos equivalentes nas execuções com PThreads, podendo ser verificados nas Tabelas 7 e 9. Nota-se que uma nova coluna foi inserida nessas tabelas, indicando a quantidade de atividades concorrentes definidas pela aplicação a serem mapeadas nos PVs para execução.

**Tabela 5: Tempos em segundos da execução (seqüencial) do compactador GZip**

| Arquitetura | Média | Desvio Padrão |
|---|---|---|
| Mono-proc | 43,698 | 2,829 |
| Bi-proc | 46,104 | 3,561 |

**Tabela 6: Tempos em segundos do compactador paralelo com Pthreads em uma arquitetura mono-processada**

| Threads | Média | Desvio Padrão |
|---|---|---|
| 1 | 54,924 | 0,224 |
| 2 | 53,440 | 0,957 |
| 3 | 53,030 | 1,111 |
| 4 | 52,349 | 0,919 |
| 5 | 52,394 | 1,026 |
| 10 | 51,896 | 0,592 |
| 15 | 51,976 | 0,509 |
| 20 | 51,744 | 0,428 |

**Tabela 7: Tempos em segundos do compactador paralelo em Anahy em uma arquitetura mono-processada**

| PVs | Tarefas | Média | Desvio Padrão |
|---|---|---|---|
| 1 | 1 | 48,988 | 2,003 |
| | 2 | 49,822 | 1,086 |
| | 3 | 53,070 | 2,559 |
| | 4 | 57,387 | 1,759 |
| | 5 | 61,465 | 3,859 |
| 2 | 1 | 49,824 | 3,859 |
| | 2 | 52,584 | 2,700 |
| | 3 | 54,745 | 1,894 |
| | 4 | 56,715 | 1,795 |
| | 5 | 57,750 | 35,117 |
| 3 | 1 | 48,898 | 2,158 |
| | 2 | 49,384 | 1,121 |
| | 3 | 53,437 | 2,333 |
| | 4 | 60,477 | 1,580 |
| | 5 | 61,750 | 4,73 |
| 4 | 1 | 46,054 | 1,651 |
| | 2 | 48,778 | 2,504 |
| | 3 | 51,425 | 1,107 |
| | 4 | 59,707 | 1,949 |
| | 5 | 59,917 | 3,114 |
| 5 | 1 | 46,432 | 1,934 |
| | 2 | 49,658 | 2,592 |
| | 3 | 54,787 | 2,099 |
| | 4 | 61,752 | 4,208 |
| | 5 | 63,922 | 3,815 |

**Tabela 8: Tempos em segundos do compactador paralelo com Pthreads em uma arquitetura bi-processada**

| Threads | Média | Desvio Padrão |
|---|---|---|
| 1 | 53,043 | 2,592 |
| 2 | 43,023 | 2,023 |
| 3 | 31,348 | 2,023 |
| 4 | 22,592 | 2,097 |
| 5 | 20,592 | 2,097 |
| 10 | 20,716 | 0,238 |
| 15 | 21,561 | 0,171 |
| 20 | 21,985 | 0,381 |

### 3.3. ConvoP

Esta aplicação implementa uma técnica de tratamento de imagens: operação de convolução. A convolução consiste em multiplicar um filtro por uma imagem, os dois representados através de uma matriz. O processo consiste no deslizamento do filtro sobre a região correspondente na imagem, multplicando os elementos do fltro sobre a janela correspondente da imagem, e dividindo pelo peso da máscara (soma de todos os elementos); o deslocamento da janela é feito da esquerda para a direita, de cima para baixo. Esta aplicação oferece um alto grau de paralelismo, pois a operação de convolução é realizada

**Tabela 9: Tempos em segundos do compactador paralelo em Anahy em uma arquitetura bi-processada**

| PVs | Tarefas | Média | Desvio Padrão |
|---|---|---|---|
| 1 | 1 | 37,596 | 2,493 |
| | 2 | 35,185 | 0,735 |
| | 3 | 34,411 | 0,590 |
| | 4 | 34,446 | 0,217 |
| | 5 | 34,314 | 0,235 |
| 2 | 1 | 37,218 | 3,101 |
| | 2 | 30,645 | 6,474 |
| | 3 | 28,763 | 7,236 |
| | 4 | 24,053 | 7,066 |
| | 5 | 30,284 | 4,491 |
| 3 | 1 | 37,696 | 1,916 |
| | 2 | 26,823 | 2,331 |
| | 3 | 22,428 | 1,504 |
| | 4 | 21,292 | 1,326 |
| | 5 | 21,322 | 1,206 |
| 4 | 1 | 36,858 | 0,171 |
| | 2 | 24,438 | 1,817 |
| | 3 | 22,366 | 0,726 |
| | 4 | 22,274 | 0,725 |
| | 5 | 22,202 | 0,557 |
| 5 | 1 | 35,910 | 0,505 |
| | 2 | 28,156 | 0,431 |
| | 3 | 19,731 | 0,564 |
| | 4 | 24,465 | 0,416 |
| | 5 | 20,950 | 0,129 |

pixel a pixel permitindo o uso da estratégia *split-compute-merge* descrito na Seção 3.1. O processo consiste em dividir a imagem em blocos de acordo ao número de tarefas explicitadas pelo programador; em algumas ocasioes, caso o tamanho da imagem não seja múltiplo do número de tarefas, a última tarefa pode receber algumas linhas a mais para serem processadas.

Foram realizadas duas versões, uma com PThreads e outra com Anahy. Para o ambiente Anahy, foi utilizado o número de PVs padrão da biblioteca, que é formado por 4 PVs. Os tempos de execução desta aplicação estão na Tabela 12 expressos em segundos, e foram obtidos medindo o tempo completo de execução. Analisando os resultados, vemos que na imagem de tamanho $256 \times 256$, os resultados de Anahy foram superiores aos da implementação com PTreads, nas implementações com 2, 4 e 8 *threads*. Isto ocorre pois Anahy tem um esquema de execução que faz uma boa utilização do tempo que é concedido à aplicação. Na imagem de tamanho $512 \times 512$, já podemos observar alguma perda de desempenho de Anahy em relação à Pthreads. Com uma imagem de tamanho $1024 \times 1024$ vemos que os tempos de execução das duas biliotecas estão muito próximos uns dos outros, para as 3 quantidades de *threads*. A perda de desempenho refletida nos tempos de execução de Anahy, é justificado, pois a medida que o tamanho da imagem aumenta, torna-se mais demorado o processo de escrita em disco. Neste caso, as duas bibliotecas podem apresentar um comportamento muito parecido.

**Tabela 10: Tempos em segundos para Pthreads em Fibonacci**

| Arquitetura | Fibo | Média | Desvio Padrão |
|---|---|---|---|
| mono proc | 15 | 1,221 | 0,0540 |
| | 16 | 1,391 | 0,0584 |
| bi proc | 15 | 1,095 | 0,1092 |
| | 16 | 1,414 | 0,1873 |

**Tabela 11: Tempos em segundos de Fibonacci em Anahy em uma arquitetura mono-processada**

| PVs | Fibo | Média | Desvio Padrão |
|---|---|---|---|
| 1 | 15 | 0,186 | 0,0819 |
| | 16 | 0,509 | 0,2502 |
| | 17 | 1,482 | 0,7466 |
| | 18 | 5,170 | 0,0092 |
| | 19 | 13,877 | 0,0526 |
| | 20 | 36,285 | 0,0862 |
| 2 | 15 | 0,179 | 0,0794 |
| | 16 | 0,501 | 0,2427 |
| | 17 | 1,461 | 0,7424 |
| | 18 | 5,204 | 0,0301 |
| | 19 | 14,042 | 0,0434 |
| | 20 | 36,866 | 0,0889 |
| 3 | 15 | 0,059 | 0,0295 |
| | 16 | 0,098 | 0,0621 |
| | 17 | 0,177 | 0,2004 |
| | 18 | 0,302 | 0,2926 |
| | 19 | 0,374 | 0,0832 |
| | 20 | 0,778 | 0,0625 |
| 4 | 15 | 0,055 | 0,0338 |
| | 16 | 0,132 | 0,0883 |
| | 17 | 0,284 | 0,1924 |
| | 18 | 0,528 | 0,1710 |
| | 19 | 0,743 | 0,1606 |
| | 20 | 1,788 | 3,3887 |
| 5 | 15 | 0,092 | 0,0582 |
| | 16 | 0,177 | 0,1084 |
| | 17 | 0,391 | 0,3147 |
| | 18 | 0,834 | 0,6495 |
| | 19 | 0,797 | 0,1365 |
| | 20 | 1,315 | 0,3752 |

### 3.4. Fibonacci

Os últimos resultados apresentados nesta avaliação referem-se à execução de uma aplicação com elevado número de atividades concorrentes e sincronizações entre estas. Trata-se da implementação recursiva de um algoritmo de Fibonacci, que tem seu fluxo de execução ilustrado na Figura 5. Nota-se que para um valor pequeno (5), a quantidade de criações e sincronizações de tarefas realizadas é considerável.

A análise feita compara a execução do algoritmo paralelo suportado por PThreads (Tabela 10) e por Anahy (Tabelas 11 e 13).

**Tabela 12: Tempos da aplicação ConvoP, expressos em segundos.**

| Tamanho da imagem | Quantidade de tarefas | Anahy | | Pthreads | |
|---|---|---|---|---|---|
| | | Média | DesvPad | Média | DesvPad |
| 256 | 2 | 1.397667 | 0.631565 | 1.855849 | 0.984134 |
| | 4 | 0.834735 | 0.263164 | 1.594874 | 0.707890 |
| | 8 | 0.800374 | 0.276784 | 1.392401 | 0.576007 |
| 512 | 2 | 1.966468 | 0.307431 | 4.668790 | 1.294266 |
| | 4 | 1.763698 | 0.305904 | 4.937353 | 1.576778 |
| | 8 | 1.972571 | 0.857244 | 1.756903 | 0.526590 |
| 1024 | 2 | 14.331558 | 5.383365 | 15.561152 | 5.152146 |
| | 4 | 14.317198 | 5.153535 | 16.369576 | 4.153191 |
| | 8 | 16.796820 | 3.972121 | 16.705836 | 3.427423 |
| 2048 | 2 | 53.734340 | 10.450171 | 48.985360 | 11.615081 |
| | 4 | 53.034336 | 10.927343 | 48.694584 | 11.140005 |
| | 8 | 33.988656 | 14.538091 | 38.153152 | 9.683560 |

Na implementação realizada, cada invocação recursiva da função Fibonacci gera a criação de uma nova tarefa concorrente. Como o suporte sobre PThreads gera uma nova *thread* para cada atividade, o cálculo fica limitado a valores baixos da série (o número de *threads* suportadas pelo sistema operacional é limitado). Já em Anahy, o número de atividades em execução simultânea é controlado pelo número de PVs do núcleo.

**Tabela 13: Tempos em segundos de Fibonacci em Anahy em uma arquitetura bi-processada**

| PVs | Fibo | Média | Desvio Padrão | PVs | Fibo | Média | Desvio Padrão |
|---|---|---|---|---|---|---|---|
| 1 | 15 | 0,171 | 0,0213 | 4 | 15 | 0,198 | 0,0277 |
| | 16 | 0,443 | 0,0321 | | 16 | 0,431 | 0,0680 |
| | 17 | 1,239 | 0,1064 | | 17 | 0,962 | 0,1759 |
| | 18 | 3,634 | 0,1989 | | 18 | 2,383 | 0,5147 |
| | 19 | 10,429 | 0,4364 | | 19 | 6,114 | 1,4462 |
| | 20 | 27,829 | 1,0544 | | 20 | 16,115 | 3,8569 |
| 2 | 15 | 0,134 | 0,0151 | 5 | 15 | 0,221 | 0,0096 |
| | 16 | 0,285 | 0,0404 | | 16 | 0,495 | 0,0486 |
| | 17 | 0,613 | 0,0639 | | 17 | 1,146 | 0,0723 |
| | 18 | 1,452 | 0,1982 | | 18 | 2,885 | 0,3529 |
| | 19 | 3,837 | 0,5850 | | 19 | 7,535 | 0,4952 |
| | 20 | 10,219 | 0,9775 | | 20 | 19,486 | 1,3612 |
| 3 | 15 | 0,162 | 0,0223 | | | | |
| | 16 | 0,337 | 0,0422 | | | | |
| | 17 | 0,723 | 0,0813 | | | | |
| | 18 | 1,749 | 0,1622 | | | | |
| | 19 | 4,621 | 0,5051 | | | | |
| | 20 | 11,900 | 1,5037 | | | | |

Também chama atenção nos resultados da execução em arquitetura bi-processada que, aumentando o número de atividades concorrentes em execução simultânea (maiores números de PVs e de tarefas), o *overhead* gerado pelas sincronizações afeta o desempenho final. Este característica justifica, ainda mais, que o número de atividades concorrentes em execução simultânea deve ser adaptado aos recursos de processamento da arquitetura.

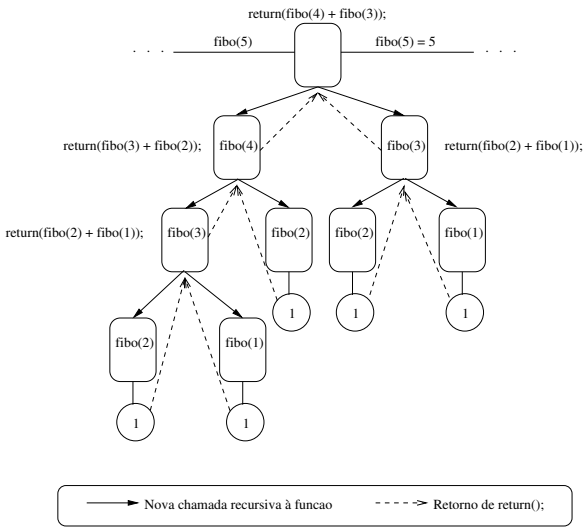


**Figura 5: Modelo de execução de Fibonacci**

## 4. Conclusão

Este trabalho apresentou os principais componentes de Anahy. Entre eles, o núcleo executivo (explicando o funcionamento do escalonador), os processadores virtuais (aumenta o número de tarefas potencialmente em execução) e a interface aplicativa (API), que disponibiliza ao programador serviços que permitam utilizar o ambiente. Estes componentes foram descritos para apresentar o funcionamento interno de Anahy.

O Anahy é um ambiente que explora os recursos computacionais (memória e dados) disponíveis em uma arquitetura multiprocessada através de uma camada de serviços, que além de manter a portabilidade de código, também mantem a portabilidade de desempenho das aplicações. Com estas características, Anahy apresenta-se como uma ferramenta eficiente para a exploração de PAD (processamento de alto desempenho) em arquiteturas dotadas de dois ou mais processadores.

Atualmente o ambiente Anahy encontra-se disponível em uma versão para computadores multiprocessados (SMPs) e com um protótipo para aglomerados de computadores (permite a migração de tarefas entre os nodos). A versão atual de Anahy permite que o programador possa definir o número de processadores virtuais, explicitar o número de tarefas concorrentes em sua aplicação entre outras funcionalidades descritas na 2.4.1.

Observando os resultados de desempenho na Seção 3 podemos ver que Anahy comportou-se de forma elegante nas distintas aplicações, mantendo melhor desempenho na maior parte dos casos; fator este, conclusivo no momento de escolher uma ferramenta de programação concorrente.

Os trabalhos futuros estão centrados em portar de forma total Anahy para um aglomerado de computadores onde, além de poder realizar a troca de mensagem entre os nodos, será possível enviar e receber tarefas a serem executadas.

## Referências


Casavant, T. L. and Kuhl, J. G. (1988). A taxonomy of scheduling general-purpose distributed computing systems. *IEEE Trans. on Software Engineering*, 14(2):141–154.

Cavalheiro, G. G. H. (1999). *Athapascan-1: Interface générique pour l'ordonnancement dans un environnement d'exécution parallèle*. Thèse de doctorat, Institut National Polytechnique de Grenoble, Grenoble, France.

Cavalheiro, G. G. H. (2001). A general scheduling framework for parallel execution environments. In *Proceedings of SLAB'01*, Brisbane, Australia.

Cavalheiro, G. G. H., Denneulin, Y., and Roch, J.-L. (1998). A general modular specification for distributed schedulers. In Springer Verlag, L. ., editor, *Proceedings of Europar'98*, Southampton, England.

Cavalheiro, G. G. H., Real, L. C. V., and Dall'Agnol, E. C. (2003). 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.

Cohen, W. E., Yalamanchili, N., Tewari, R., Patel, C., and Kazi, T. (1998). Exploitation of multithreading to improve program performance. In *Proceedings of The Yale Multithreaded Programming Workshop*, New Haven, EUA.

Dall'Agnol, E. C., Real, L. C. V., Benitez, E. D., and Cavalheiro, G. G. H. (2003). Portabilidade na programação para o processamento de alto desempenho. In *Anais do IV Workshop em Sistemas Computacionais de Alto Desempenho*, São Paulo, Brasil.

Pacheco, P. S., editor (1997). *Parallel programming with MPI*. Morgan Kaufmann, San Francisco.

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

Stevens, R. W., editor (1998). *UNIX Networking Programming: Networking APIS*. Prentice Hall.