Anahy-DVM: um Módulo de Escalonamento Distribuído*

Daniela S. Peranconi, Marcelo A. Cardozo, Gerson G. H. Cavalheiro, Otávio C. Cordeiro
Universidade do Vale do Rio dos Sinos - PIPCA
Av. Unisinos, 950 - São Leopoldo - RS - Brasil
{*daniela, mcardozo, gersonc, otavio*}*@anahy.org*

Resumo

O uso de aglomerados de computadores para fins de alto desempenho vem aumentando nos últimos anos. Porém, a programação dessas arquiteturas não é trivial. Além de desenvolver a aplicação, detectar e explicitar a concorrência nela existente, o programador também deve implementar o escalonamento de sua aplicação para explorar, efetivamente, o paralelismo da arquitetura. Algumas ferramentas se propõem a solucionar esses problemas, oferecendo recursos de escalonamento de tarefas; entre elas, Anahy. Este trabalho apresenta a implementação de um módulo para Anahy com fins de dotá-la de suporte à execução em ambientes com memória distribuída. Para tanto, seu núcleo executivo foi estendido para que se possa ter acesso às estruturas de dados imprescindíveis à distribuição da carga computacional. Também foi desenvolvido um mecanismo de comunicação para troca de informações entre os nós do aglomerado. Por fim, o módulo desenvolvido é avaliado através de seu uso em uma aplicação sintética.

1. Introdução

Nos últimos anos, o desenvolvimento do processamento de alto desempenho (PAD) encontrou grandes aliados nos aglomerados de computadores compostos de nós multiprocessados (SMPs). No entanto, a exploração dessas arquiteturas não é trivial, pois sua programação envolve, além da codificação do problema propriamente dito, o mapeamento das atividades concorrentes do programa e dos seus dados nas unidades disponíveis de suporte ao cálculo (processador e memória). Porém, na maioria dos casos, esse mapeamento não pode ser realizado de forma direta, pois a concorrência da aplicação normalmente é superior ao paralelismo suportado pela arquitetura. Assim, utilizando recursos convencionais de programação concorrente, é de responsabilidade do programador determinar o número de tarefas concorrentes que a arquitetura utilizada deve manter ativas simultaneamente e distribuir essas tarefas, e os dados por elas acessados, entre os processadores e os módulos de memória da arquitetura.

Transpor essas dificuldades, oferecendo tanto uma interface de programação de alto nível como mecanismos de gerência de recursos de hardware, implica abordar questões ligadas à portabilidade de código e desempenho dos programas [1]. Cilk [3], Athapascan-1 [11], Anahy [6] e Jade [16, 17] são ferramentas para o PAD inseridas nesse contexto. Essas ferramentas disponibilizam recursos de programação, para descrição da concorrência de uma aplicação, além de explorarem o conceito de escalonamento aplicativo, que permite tirar proveito dos recursos da arquitetura visando ao desempenho na execução de programas.

Este artigo aborda o desenvolvimento de um núcleo de execução dotado de recursos de escalonamento aplicativo para uma ferramenta de exploração de aglomerados de computadores: Anahy [5, 8]. A base deste núcleo é permitir a incorporação de algoritmos de escalonamento de listas (como [12]) em um ambiente de execução distribuído para suporte ao escalonamento de aplicações em tempo de execução.

O presente artigo encontra-se organizado como segue. Na primeira seção é discutida a utilização de grafos como base para escalonamento. Na seção seguinte é introduzido Anahy, um ambiente para PAD em aglomerados de computadores, sendo o modelo idealizado para suporte à execução de aplicações em ambientes dotados de memória distribuída apresentado na Seção 4 e a implementação deste modelo apresentado na Seção 5. Por fim, a Seção 6 apresenta uma análise de desempenho realizada sobre Anahy em sua versão distribuída e a Seção 7 conclui o artigo.

2. Grafos como Base para Escalonamento

A principal função do escalonamento é atribuir as unidades de cálculo da aplicação, denominadas tarefas, às unidades de execução da arquitetura. Escalonamento apli-

*Este trabalho foi parcialmente desenvolvido em colaboração com a HP Brazil R&D.

cativo, em particular, diz respeito a aplicar heurísticas de distribuição de trabalho considerando a estrutura do programa, ou seja, tirando proveito de informações relativas ao relacionamento (sincronizações e comunicações) entre as tarefas do programa [10]. Neste artigo, a abordagem é limitada ao escalonamento aplicativo dinâmico, ou seja, realizado em tempo de execução. Neste contexto, o núcleo de escalonamento deve ser concebido de forma a reagir à evolução do programa refletida nas modificações de um grafo representando as tarefas criadas e seus relacionamentos [7, 20]. Diversas heurísticas de escalonamento [13, 22, 14, 19, 18] exploram o conhecimento sobre a estrutura do programa para otimização de índices de desempenho. Embora as pesquisas sobre estas técnicas sejam populares, sua exploração prática em ambientes de execução ainda é bastante reduzida, existindo poucas opções (como Athapascan-1 [11] e Cilk [3]) desenvolvidas com esse propósito.

Se uma aplicação é decomposta em tarefas e estas são conectadas entre si seguindo o fluxo de dados que cada tarefa produz e consome, pode-se criar um grafo orientado (*DAG*) da execução da aplicação. Esse grafo pode ser considerado uma interface entre o programa em execução e o escalonamento [7, 4].

O tipo de grafo mais utilizado em escalonamento é o grafo de dependências. Um grafo de dependências $\mathcal{G}(\mathcal{T}, \mathcal{A})$ é composto por um conjunto $\mathcal{T} = \{\tau_1, \tau_2 \dots \tau_n\}$ de tarefas e um conjunto $\mathcal{A} = \{a_1, a_2 \dots a_m\}$, com $m \geq n - 1$, de arestas representando dados comunicados entre tarefas. Nesse grafo, a tupla (τ, a) representa um dado de saída produzido por τ e (a, τ) representa uma dependência de entrada de τ. Assim, um arco (τ_i, τ_j) implica a existência de uma aresta a_k tal que (τ_i, a_k) e (a_k, τ_j). Nesse caso, um arco (τ_i, τ_j) significa que τ_j não pode ser executada sem que τ_i tenha terminado sua execução, pois os dados gerados por τ_i serão utilizados em algum momento por τ_j.

Além da dependência entre tarefas, este tipo de grafo expõe diversas informações úteis sobre a estrutura do programa em execução. Por exemplo, o grau de paralelismo que pode ser atingido e o caminho crítico da execução (maior seqüência de tarefas a ser respeitada). No contexto deste artigo, a informação de maior relevância diz respeito à localidade de dados. Como o tempo de comunicação em sistemas com memória distribuída não é desprezível, convém considerar o custo de comunicação no momento em que tarefas são alocadas aos processadores. Cilk e Athapascan-1 obtêm informações a respeito da localidade de dados a partir das dependências entre tarefas, no entanto cada uma destas ferramentas propõe uma estratégia diferente para reduzir os custos de comunicação.

A estratégia de Cilk está baseada em agrupar seqüências de tarefas em unidades de execução de maior granularidade, as *threads* Cilk. Desta forma, comunicações entre tarefas localizadas em uma mesma *thread* são realizadas com custo nulo. Já em Athapascan-1, a estratégia adotada é explicitar os dados comunicados entre tarefas, de forma a considerar os custos de comunicação destes no momento de transferir tarefas de um processador a outro. Em comum é observado que ambas ferramentas realizam a manutenção do grafo de forma distribuída entre os processadores e que, em cada processador, é privilegiada a manipulação das tarefas locais à seção local deste grafo. Ambas ferramentas também exploram *multithreading* [21] para sobrepor custos de comunicação com cálculo efetivo.

3. Anahy

Esta seção introduz Anahy [5, 8], um ambiente para exploração de PAD em aglomerados de computadores.

3.1 Interface de Programação

Os serviços da interface de programação de Anahy oferecem ao programador mecanismos para explorar o paralelismo de uma arquitetura multiprocessada dotada de uma área de memória compartilhada. Tais serviços podem ser representados pelas operações *fork*/*join*, disponibilizando ao programador uma API bastante próxima ao modelo oferecido pela multiprogramação baseada em processos leves.

Uma operação *fork* consiste na criação lógica de um novo fluxo de execução, sendo o código a ser executado definido por uma função $\mathcal{F}$ definida no corpo do programa. Esse operador retorna um identificador ao novo fluxo criado. No momento da invocação da operação *fork*, a função a ser executada deve ser identificada e passados os parâmetros necessários à sua execução.

A sincronização com o término da execução de um fluxo é realizada pela operação *join*, através da identificação do fluxo. Essa operação permite que um fluxo bloqueie, aguardando o término de outro, de forma a garantir que a função $\mathcal{F}$ terminou, sendo possível recuperar seu resultado na memória compartilhada. Dessa forma, as operações de sincronização (*fork* e *join*) realizadas no interior de um fluxo de execução permitem definir novas tarefas que poderão vir a ser executadas de forma concorrente.

3.2 Núcleo Executivo

O algoritmo de escalonamento de listas explorado por Anahy manipula tarefas. A implementação de Anahy manipula *threads*. As tarefas em Anahy são escalonadas dentro do contexto das *threads*. Na Figura 1 pode-se ver a relação entre tarefas e *threads* em Anahy. Essa relação tarefa $\times$ *thread* implica que um *fork* gera efetivamente duas novas tarefas, porém, gera apenas uma modificação no grafo; de forma análoga, a operação *join* não realiza nenhuma modificação.

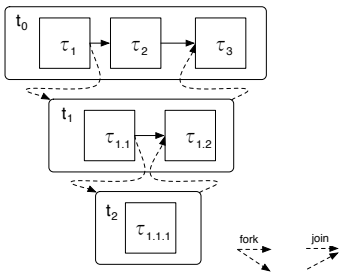

Figura 1. Exemplo de relação tarefa $\times$ *thread*.

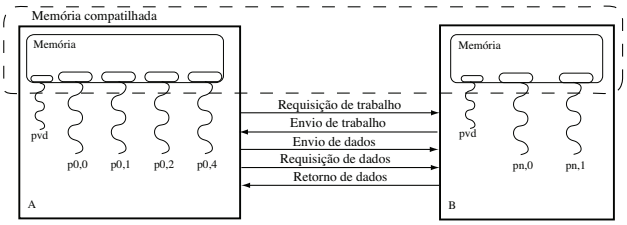

Figura 2. Suporte à comunicação em Anahy.

Da implementação do núcleo executivo, destaca-se sua organização do escalonamento em dois níveis. O primeiro é realizado pelo sistema operacional e consiste no mapeamento dos fluxos de execução associados aos processadores virtuais (PVs) aos recursos físicos de processamento (de forma equivalente, os dados manipulados em um nó na memória local).

O escalonamento aplicativo, no qual se dá a distribuição da carga computacional e o controle da execução do programa, é realizado no nível seguinte. Nele, o escalonador utiliza-se de um algoritmo de listas [12] para explorar de forma eficiente o grafo de dependências, percorrido em profundidade e em ordem lexicográfica por prover maior eficiência. Tendo como base esse grafo, realiza a atribuição de tarefas a cada PV, assim como controla a dependência de dados entre as tarefas. Dessa forma, obtém-se a localidade dos dados em cada PV, pois quando a execução de um fluxo é iniciada, esse fluxo potencialmente gera toda uma sub-árvore contendo as tarefas geradas por essa primeira. O escalonador baseia-se no grafo para obter uma ordem de execução que maximize a eficiência, já que toda vez que vai buscar uma nova tarefa a ser executada, ele evita tarefas que irão bloquear esperando o término de uma outra ainda em execução. A decisão da ordem de execução é calculada sempre que uma busca à lista de tarefas prontas é realizada.

4. Modelo de Escalonamento Distribuído

Nesta seção é apresentado o modelo da arquitetura de Anahy-DVM, um módulo que permite a execução de aplicações em ambientes dotados de memória distribuída, como os aglomerados de computadores.

4.1 Arquitetura Distribuída para Anahy

Anahy já apresenta um conjunto de primitivas de comunicação entre nós para a distribuição de carga de uma aplicação, mas um núcleo de escalonamento de uso geral não encontra-se disponível. Foi identificado que, para a correta comunicação entre nós Anahy, é necessário um conjunto de serviços que realizem o escalonamento e manutenção do grafo. Estes serviços atuam requisitando e enviando tarefas, além de enviar e receber dados de entrada e resultados das tarefas. Para manter compatibilidade com Anahy-SMP, foi incluído um *daemon* de comunicação (Figura 2), responsável por prover os serviços discutidos na seqüência. O *daemon* consiste em um PV baseado no algoritmo de Mensagens Ativas (MA), dedicado ao processamento da comunicação entre os nós da arquitetura (PV_d). Portanto, os PVs de Anahy ficam dedicados ao processamento das tarefas da aplicação, obtendo-se uma sobreposição de cálculos efetivos com comunicação para fins de ganho de desempenho [21]. No entanto, para tal ganho, os serviços executados pelo *daemon* devem ser rápidos e não bloqueantes.

O processo de escalonamento em Anahy é dividido em dois níveis, local e global, correspondendo às concorrências intra e entre-nós, respectivamente. A concorrência intra-nó refere-se a exploração dos recursos computacionais inerentes a um nó. Já, a concorrência entre-nós explora os recursos de dois ou mais nós interligados por rede. Assim, é necessário um conjunto de estruturas e primitivas responsáveis pelo controle da execução neste ambiente. Anahy-SMP oferece um conjunto de funcionalidades para controle de listas de tarefas, criação, sincronização e escalonamento em ambiente intra-nó. Assim, o mesmo conjunto de funcionalidades deve estar presente em Anahy-DVM para que o escalonamento e manutenção do grafo de tarefas seja possível em nivel global.

4.2 Serviços de Comunicação

Os serviços de comunicação devem prover o suporte ao balanceamento de carga e migração de dados entre os nós da arquitetura. A Figura 2 identifica estes serviços através das possíveis interações entre dois nós – A e B – da arquitetura distribuída. Estes serviços são discutidos na seqüência:

- **Requisição de trabalho**: quando um nó não apresenta trabalho em sua lista local, sinaliza a outro nó que está ocioso e que pode receber tarefas, tirando proveito do paralelismo

da arquitetura. A Figura 2 ilustra o pedido de trabalho de A para B;

- **Envio de trabalho**: quando um nó possui *threads* que podem ser migradas, as envia para nós que sinalizaram ociosidade. Este envio consiste em uma mensagem com a descrição da *thread* a ser executada e os dados que esta tarefa manipula. Caso o nó não tenha *threads* que possam ser migradas, o fato é notificado através da mensagem. A Figura 2 ilustra o envio de trabalho de B para A;
- **Requisição de dados**: executado quando um PV necessita de dados produzidos por outro PV, estando no mesmo nó ou não. Este serviço fornece à primitiva *join* transparência de localização, permitindo ao programador explicitar apenas a *thread* que produziu os dados. A Figura 2 representa a requisição de A para B;
- **Retorno de dados**: quando um nó responde à alguma requisição. O nó que recebeu a requisição determina i) se a *thread* existe em seu conjunto local e ii) se os dados encontram-se disponíveis na memória. Em caso positivo, os envia ao nó requisitante. Caso contrário, aguarda o término da execução da *thread* para enviar os dados produzidos por ela. Na Figura 2 é representado pelo envio de dados de B para A; e,
- **Envio de dados**: quando um nó termina a execução de uma *thread* que foi migrada, envia os dados ao nó de origem, antecipando sua requisição, mesmo que esta não seja solicitada. Desta forma, otimiza-se o desempenho garantindo que os dados estarão no nó quando necessários sem e que não será preciso esperar a sua comunicação. Na Figura 2 é representado pelo envio de dados de A para B.

Além destes serviços, é necessário dotar a arquitetura virtual de Anahy de primitivas que permitam a migração transparente de *threads* entre os nós. Para tal, foi proposta a utilização de funções de empacotamento e desempacotamento de dados [15]. Quatro primitivas oferecem este suporte: *athread_ attr_pack_in_func*, *athread_attr_unpack_in_func*, *athread_attr_pack_out_func*, *athread_attr_unpack_out_func*. As duas primeiras são responsáveis por empacotar e desempacotar os dados de entrada e as duas últimas são responsáveis pelos resultados gerados pela *thread*. Como essas funções são relacionadas às *threads*, estas devem ser associadas no momento da sua criação, assim como também pode ser prevista a inserção de anotações no grafo. Estas anotações têm por objetivo associar custos às tarefas e a comunicação de dados entre elas de maneira a prover mais informações ao escalonador. Dessa forma, pode-se utilizar uma política de escalonamento que leve em consideração os custos previstos.

Havendo uma migração de *thread*, pode ocorrer que o custo de comunicação dos dados seja superior ao custo de execução da tarefa. Nesse caso, ocorre a inserção de custos na execução da aplicação. Para que o ambiente possa detectar essas situações, é necessário que o programador anote no grafo de dependência de dados os custos associados à migração dos dados e o custo estimado de execução. Portanto, também foi necessário estender as primitivas de criação de tarefas Anahy para que o programador possa associar esses custos às *threads*. A partir do grafo anotado serão executados os algoritmos que determinarão se uma migração de tarefa entre nós poderá ser executada ou não, dependendo se a mesma não adiciona custo a execução do caminho crítico.

4.3 Funcionamento do Escalonador

O mecanismo de escalonamento implementado na versão distribuída de Anahy deve seguir o mesmo conjunto de regras implementado na versão SMP que, por sua vez, obedece o modelo de execução proposto por Graham [12]. Portanto, o núcleo com suporte a ambientes distribuídos utilizará, em grande parte, o algoritmo de roubo de cargas [2]. A característica de minimizar os custos associados à execução do caminho crítico encontrada em Anahy deve ser adotada pela versão distribuída. Assim, é necessário evidenciar que as listas de tarefas dos nós serão mantidas de forma distribuída, isto é, cada nó manterá sua lista local e as interações entre as tarefas que estão em nós diferentes serão feitas através dos serviços apresentados anteriormente.

Embora Anahy-DVM possa tirar proveito de grafos anotados para tomar decisões sobre migração de tarefas, a análise feita a seguir não utiliza anotações no grafo e assume que a tarefa mais próxima da raiz possui, potencialmente, mais trabalho.

Inicialmente, quando a máquina virtual (MV) Anahy está sendo inicializada, apenas o nó que começou a executar o programa possui trabalho. Depois de criada a MV, o roubo de trabalho terá início pelos nós ociosos. Cada nó tem uma lista de todos os outros que compõem a MV e, ao acaso, escolhe outro para pedir trabalho. Caso o nó ao qual foi feito o pedido não possua trabalhos, este enviará uma mensagem ao requisitante informando que não os possui no momento. Caso haja trabalho, é de responsabilidade do nó requisitado de analisar se alguma de suas tarefas mais próximas à raiz pode ser migrada. Caso afirmativo, os dados de entrada da tarefa são empacotados e enviados ao nó requisitante. Este começará imediatamente a executar a tarefa recebida. Quando o nó de origem da tarefa necessitar sincronizar com a mesma, este enviará uma mensagem ao nó ao qual ela foi migrada pedindo os dados produzidos por ela. Nesse momento, o nó onde a tarefa foi migrada envia o seu estado atual ao nó de origem. Se já tiver sido completada, os dados produzidos são enviados também, mas se ela ainda estiver em execução, ou bloqueada, apenas a atualização do estado da tarefa é enviado. O nó de origem pega a atualização e toma a medida necessária para continuar a execução do programa. Esta pode ser bloquear a tarefa corrente e executar uma outra, ou apenas sincronizar com os dados recebidos.

Dessa maneira, o programador não precisa codificar a migração de tarefas para explorar o paralelismo da máquina virtual, pois o algoritmo também é coerente com o escalonador implementado pela versão SMP, apresentado na Seção 3, evitando, assim, diferenças nas políticas de escalonamento global e local.

5. Implementação

A implementação de Anahy-DVM se deu em duas etapas. A primeira composta da adequação da ferramenta de MA para o ambiente Anahy e a segunda pela implementação das primitivas necessárias pelo escalonador distribuído dentro do próprio núcleo executivo.

5.1 Mensagens Ativas

Primeiramente, foram feitas alterações no código original das MA para permitir a esse mecanismo lidar com os tipos de dados utilizados no núcleo executivo de Anahy. Isso possibilitou que as MA [9] manipulassem tais estruturas de dados para fins de migração de tarefas e de dados entre nós da MV Anahy-DVM.

Após, foram criadas as funções de tratamento, chamadas quando uma MA chega, que têm por finalidade efetivar os serviços mostrados na Figura 2. Entretanto, como os serviços devem ter acesso a dados locais ao escalonador de Anahy, eles foram implementados dentro do núcleo executivo. Dessa forma, o mecanismo de MA apenas as chama, passando os parâmetros recebidos na mensagem.

5.2 Funções do Usuário

Anahy mantém compatibilidade com o padrão POSIX para *threads* trabalhando com tipos de dados abstratos, ou seja, através de ponteiros sem tipo definido. Portanto, para fins de manter a compatibilidade, é necessário que Anahy-DVM também trabalhe com esse tipo de dados.

Entretanto, isso gera um problema para o núcleo de comunicação de dados, pois não é possível saber que tipos de dados estão sendo trabalhados e como deverão ser empacotados para efetuar a comunicação, uma vez que somente o usuário tem conhecimento dos dados manipulados sob sua aplicação e, portanto, somente ele tem a capacidade de empacotar, ou desempacotar, os dados para comunicação.

A solução adotada por Anahy-DVM foi considerar o usuário responsável pela criação de funções que serão utilizadas pelo núcleo executivo para empacotar e desempacotar os dados utilizados por uma tarefa quando tiver de ser migrada. Assim, a comunicação é possível, mesmo que os tipos de dados não sejam conhecidos, pois seu tratamento está sob jurisdição das funções do usuário.

Por outro lado, essas funções devem seguir um padrão rígido, para que possam ser utilizadas de maneira correta pelo núcleo executivo. Em Anahy-DVM, as funções recebem como entrada um ponteiro sem tipo definido contendo o dado com o qual a função do usuário trabalhará. No término da função, esta deve retornar um ponteiro sem tipo definido que contém o resultado de sua computação. No caso de ser uma função de empacotamento, o retorno deve ser para o pacote criado; já no caso de desempacotamento, o retorno deve apontar para a região de memória onde os dados foram colocados.

As funções de empacotamento e desempacotamento necessitam inicializar o pacote a ser enviado antes de começar a mover os dados do *buffer* local para dentro dele, também, necessitam usar primitivas próprias do ambiente para realizar acessos a ele. Isso foi feito de maneira a minimizar a quantidade de cópias feitas para fins de envio do pacote. As primitivas necessárias à manipulação de pacotes são apresentadas na Tabela 1.

Tabela 1. Primitivas para manipulação de pacotes.

Primitiva	Descrição
athread_msg_init	Inicializa o pacote, alocando o espaço necessário na memória. Retorna o ponteiro para o pacote criado.
athread_pack	Faz a cópia da quantidade de dados indicada no *buffer* para dentro do pacote.
athread_unpack	Responsável pelo acesso de leitura dentro do pacote, com a qual o usuário pode retirar uma quantidade arbitrária de dados a partir do deslocamento passado para dentro do *buffer* local.

5.3 Núcleo Executivo

A estrutura de dados das tarefas precisou ser estendida para dar suporte aos serviços necessários à distribuição de tarefas e dados como proposto em [15]. O suporte às funções do usuário teve de ser feito, modificando a estrutura que especifica o descritor de uma tarefa.

5.3.1 Extensão dos Atributos

A utilização de ponteiros sem tipo definido se torna um problema para o núcleo executivo quando executando em um ambiente de memória distribuída, já que o ambiente não sabe como tratar os dados apontados de maneira que possam ser migrados.

Para tornar a migração dos dados possível, definiu-se que o programador fornece ao ambiente um conjunto de rotinas para tratar os dados de maneira a serem empacotados e migrados a algum dos nós. A maneira escolhida de informar

o ambiente das funções foi a extensão dos atributos de uma tarefa para acomodar ponteiros para essas funções criadas pelo usuário. Além disso, foram criadas as funções *athread_attr_pack_in_func*, *athread_attr_unpack_in_func*, *athread_attr_pack_out_func* e *athread_attr_unpack_out_func* já citadas. Também foi necessário estender os atributos da *thread* para que fosse possível armazenar os custos estimados sobre sua execução e sua comunicação de dados. Foram criados, para tanto, os atributos *execution_cost* e *communication_cost*, acessíveis, respectivamente, através das funções *athread_attr_set_execution_cost* e *athread_attr_set_communication_cost*. Estes devem ser associados à *thread* durante sua criação e serão utilizados pelo escalonador para a tomada de decisão durante o processo de migração.

5.3.2 Serviços

Para que as MA possam ter acesso às estruturas necessárias para a migração das tarefas e dos dados, os serviços por elas instanciados tiveram de ser implementados dentro do núcleo executivo. Tais serviços estão descritos na Tabela 2.

Tabela 2. Serviços para acesso ao núcleo executivo.

Primitiva	Descrição
steal_job	Implementa o serviço de requisição de trabalho, descrito na Seção 4.2.
athread_join_remote	Implementa o serviço de requisição de dados.
deliver_job_service	Entrega uma tarefa recebida ao escalonador.
reply_join_service	Implementa o serviço de retorno de dados.
steal_job_service	Implementa o serviço de envio de trabalho.
rcv_job_back_service	Implementa o serviço de envio de dados.

6. Resultados Obtidos

Para avaliar o escalonador distribuído, foram conduzidos testes utilizando-se uma aplicação para cálculo do Número de Fibonacci. Este cálculo, quando executado de forma recursiva, gera um fluxo de execução como pode ser visto na Figura 3. Quando programado para Anahy, o cálculo é realizado gerando uma *thread* para calcular cada nó do grafo, deixando a cargo do ambiente de execução o escalonamento das *threads* e a migração delas em caso de roubo de tarefas.

Os experimentos foram realizados em um aglomerado de computadores composto por oito nós biprocessados (2×Xeon de 2.8 Ghz), com 2 GB de RAM, interligados por uma rede Gigabit Ethernet, executando Linux Gentoo kernel 2.6.8. Os experimentos foram repetidos vinte vezes para obtenção de média e desvio padrão. Os números de Fibonacci escolhidos para os experimentos foram 10, 15 e 20,

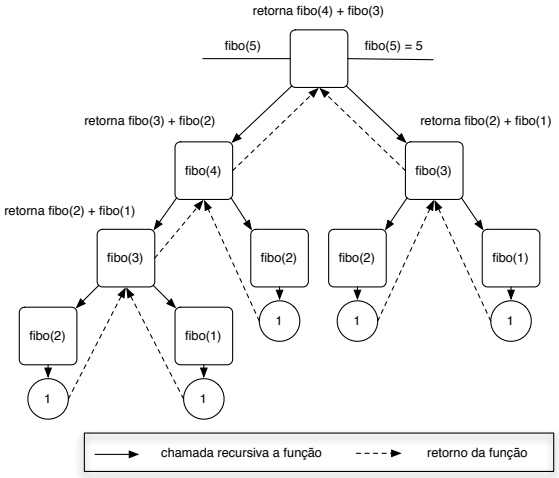

Figura 3. Fluxo de execução recursiva de Fibonacci.

pois representam uma quantidade pequena, média e grande de tarefas geradas pela aplicação.

A realização dos experimentos considerou dois casos, apresentados na seqüência.

O primeiro teste foi feito para o caso onde não se adiciona carga sintética de comunicação ao cálculo de Fibonacci. O custo de comunicação, portanto, corresponde a 4 bytes, identificando o número de Fibonacci a ser calculado. Dessa forma, testa-se se o comportamento é consistente com a versão SMP. Os resultados obtidos são mostrados na Tabela 3, com núcleo de execução configurado com 1 e 2 PVs. Nesta tabela é possível reparar que, mesmo variando o custo computacional aplicado (número de Fibonacci), o comportamento de execução é reproduzido conforme são variados os recursos de processamento, mostrando que o mecanismo de escalonamento garante estabilidade do comportamento de execução independente do número de tarefas geradas no caso de estudo.

A Figura 4 complementa a avaliação de desempenho do Caso 1 apresentando o *speed up* obtido pelas execuções paralelas para o cálculo de Fibonacci de 20. Neste gráfico, foi considerado como referência para o cálculo o tempo de execução da aplicação em 1 nó com 1 PV. Observa-se que o núcleo de escalonamento reproduz o comportamento da execução variando o suporte de concorrência representado pelos PVs. Ainda na figura, pode-se reparar que, apesar de existir um ganho, ele não é tão acelerado quanto o esperado para o número de nós, podendo ser devido ao mecanismo de roubo de trabalho escolhido. Tal mecanismo, por escolher o nó de quem vai roubar trabalho de forma aleatória, permite que no início da computação os nós sem trabalho

Tabela 3. Resultados obtidos no Caso 1.

1 PV				2 PVs			
Nós	Número Fibonacci	Média (s)	Desvio padrão	Nós	Número Fibonacci	Média (s)	Desvio padrão
1	10	2,67	0,003	1	10	1,65	0,026
1	15	30,08	0,020	1	15	18,59	0,032
1	20	541,25	0,187	1	20	334,60	0,087
2	10	1,75	0,002	2	10	0,95	0,016
2	15	19,52	0,014	2	15	10,69	0,020
2	20	292,86	0,116	2	20	171,02	0,048
4	10	0,94	0,001	4	10	0,55	0,010
4	15	10,55	0,008	4	15	6,20	0,012
4	20	158,15	0,066	4	20	99,19	0,029
8	10	0,50	0,001	8	10	0,29	0,006
8	15	5,54	0,005	8	15	3,29	0,007
8	20	83,03	0,038	8	20	52,67	0,016

tentem roubar trabalho de outro nó que também não possui nenhum trabalho. Assim, até que o mecanismo roube trabalho de um nó que possua algum, os nós ficam ociosos. Com o aumento do número de nós na MV, esse problema se potencializa, explicando os ganhos atingidos.

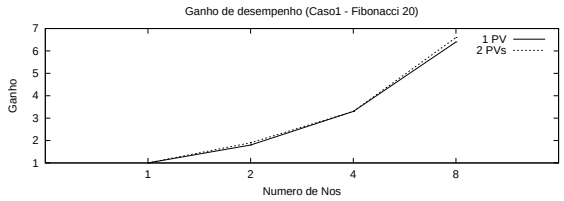

Figura 4. Ganhos de desempenho obtidos no Caso 1.

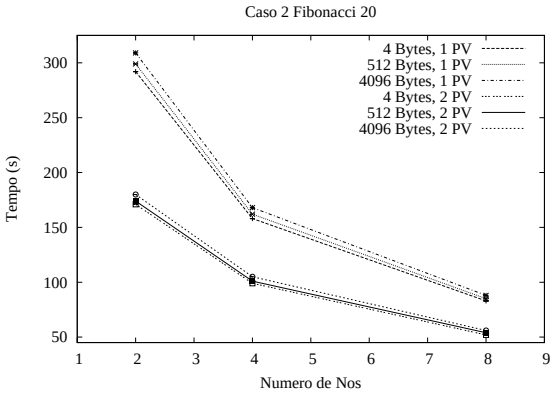

Figura 5. Resultados obtidos no Caso 2.

No segundo caso, varia-se a carga de comunicação extra a ser comunicada a cada tarefa, com valores de 512 e 4092 bytes. O objetivo, neste caso, é testar o impacto da comunicação (influência do *daemon* de comunicação) no tempo de execução da aplicação. Os resultados obtidos são representados na Figura 5. Embora não haja um impacto significativo no tempo de execução, pela sobreposição de comunicação com cálculo, pois o *daemon* de comunicação é, na verdade, um PV dedicado, destaca-se que o comportamento da execução manteve-se estável, independente do número de tarefas criadas.

Na Figura 5 pode-se ver que a curva representa o tempo de execução da aplicação com uma mesma carga de cálculo quando esta é dotada de uma carga de comunicação. Também se observa que, quando há poucos nós na arquitetura virtual, o peso da comunicação pode causar um aumento do tempo de execução da aplicação. Isto ocorre pelos poucos PVs na arquitetura e, quando um ou mais PVs bloqueiam esperando a sincronização dos dados, causam um impacto negativo na execução da aplicação. Entretanto, quando se aumenta o número de nós da arquitetura virtual, a relação da quantidade de nós que vão bloquear, esperando sincronização com a quantidade de nós que estão executando a aplicação, cairá e o impacto da comunicação será menor. Cabe ressaltar que as curvas presentes na Figura 5 possuem o mesmo comportamento, mostrando um impacto homogêneo do *daemon* na execução da aplicação.

7. Conclusão

Para que seja realizado um uso efetivo de aglomerados de computadores e arquiteturas SMP, é necessário realizar um mapeamento da concorrência da aplicação que está sendo desenvolvida para os recursos computacionais existentes na arquitetura sobre a qual esta aplicação está sendo executada. Na maioria dos casos, esse mapeamento não pode ser realizado de forma direta, pois a concorrência da aplicação é maior do que o paralelismo fornecido pela arquitetura, ficando, então, a cargo do programador determinar o número de tarefas concorrentes que a arquite-

tura utilizada deve manter em execução simultânea. Para transpor essas dificuldades, foram desenvolvidas ferramentas que auxiliam o programador no desenvolvimento de sua aplicação, dentre elas Anahy.

Entretanto, Anahy não possuia um escalonador para ambientes com memória distribuída, tais como aglomerados de computadores. Para este fim, foi necessário estender o núcleo executivo de Anahy para suportar tanto arquiteturas SMPs quanto aglomerados. Foram criadas e implementadas novas chamadas de API que permitem ao programador desenvolver aplicações para ambientes com memória distribuída.

A estratégia de escalonamento de Anahy é uma combinação das estratégias de Cilk e Athapascan-1. Em Cilk, as seqüências de tarefas são agrupadas em *threads* Cilk e tarefas de uma mesma *thread* se comunicam com custo nulo. Em Athapascan-1, os dados comunicados entre tarefas são explicitados, sendo considerados os custos de comunicação destes na migração de tarefas. Já, em Anahy, as tarefas são agrupadas em *threads* Anahy e sua migração entre nós considera os custos de comunicação de dados entre tarefas. Além de manipularem o grafo de forma distribuída entre os processadores e privilegiarem, em cada processador, a manipulação das tarefas locais à seção local deste grafo, as ferramentas exploram *multithreading* [21] para sobrepor custos de comunicação com cálculo efetivo.

Os resultados de desempenho obtidos mostraram que o núcleo executivo implementado funciona dentro das restrições impostas pelas premissas. Além disso, o núcleo provê meios para manter um comportamento de execução estável mesmo que os custos de comunicação sejam alterados. Os dados apresentados também mostraram como as informações de tempo podem ser utilizadas para a análise de sobrecarga de escalonamento. Por fim, observou-se que novas combinações de custos, como, por exemplo, adicionar uma carga sintética à execução de uma *thread*, pode aumentar o espectro de análises que podem ser realizadas com esses resultados.

Referências

[1] G. Alverson, W. Griswold, C. Lin, and L. Snyder. Abstractions for portable, scalable parallel programming. *IEEE TPDS*, 9(1):71–86, Jan. 1998.

[2] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico.*, pages 356–368, November 1994.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, 30(8):207–216, Aug. 1995.

[4] G. G. H. Cavalheiro. A general scheduling framework for parallel execution environments. In *CCGrid 2001*, Brisbane, Australia, May 2001.

[5] G. G. H. Cavalheiro, E. D. Benitez, D. S. Peranconi, and E. Moschetta. Dynamic list scheduling of threads on clusters. In *DSM 2006/CCGrid 2006*, 2006.

[6] G. G. H. Cavalheiro, E. C. Dall'Agnol, and L. C. Villa Real. Uma biblioteca de processos leves para a implementação de aplicações altamente paralelas. In *IV WSCAD*, pages 117–124, São Paulo, Brasil, Nov. 2003.

[7] G. G. H. Cavalheiro, Y. Denneulin, and J.-L. Roch. A general modular specification for distributed schedulers. *Lecture Notes in Computer Science*, 1470:373–377, 1998.

[8] G. G. H. Cavalheiro, L. P. Gaspary, M. A. Cardozo, and O. C. Cordeiro. Anahy: a programing environment for cluster computing. In *VecPar 2006 (A aparecer)*, 2006.

[9] E. C. Dall'Agnol, L. C. Villa Real, D. S. Peranconi, M. A. Cardozo Jr., and G. G. H. Cavalheiro. Construção de um mecanismo de comunicação para ambientes de processamento de alto desempenho. In *V WSCAD*, pages 169–175, Foz do Iguaçú, Brasil, Oct. 2004.

[10] D. G. Feitelson. Job scheduling in multiprogrammed parallel systems. IBM Research Report RC 19790 (87657), Aug. 1997.

[11] F. Gallilée, J.-L. Roch, G. G. H. Cavalheiro, and M. Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In *PACT'98*, pages 88–95, Paris, Oct. 1998.

[12] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, Mar. 1969.

[13] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 19(6):841–848, 1961.

[14] M. Iverson and F. Ozguner. Dynamic, competitive scheduling of multiple dags in a distributed heterogeneous environment. In *HCW'98*, page 70, Washington, USA, 1998.

[15] D. S. Peranconi. Alinhamento de Seqüências Biológicas em Arquiteturas com Memória Distribuída. Master's thesis, PIPCA - Unisinos, São Leopoldo, Brasil, Mar. 2005.

[16] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, May 1998.

[17] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A high-level machine-independent language for parallel programming. *Computer*, 26(6):28–38, 1993.

[18] R. Sakellariou and H. Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. *ipdps*, 02:111b, 2004.

[19] O. Sinnen and L. Sousa. List scheduling: extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. *Parallel Computing*, 30(1):81–101, 2004.

[20] E. Tärnvik. Dynamo - a portable tool for dynamic load balancing on distributed memory multicomputers. In *CONPAR'92/VAPP V*, pages 485–490, London, UK, 1992.

[21] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.

[22] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. Technical Report TRCS94-12, 20, 1994.