UNIVERSIDADE DO VALE DO RIO DOS SINOS
CIÊNCIAS EXATAS E TECNOLÓGICAS
PROGRAMA INTERDISCIPLINAR DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA

Marcelo Augusto Cardozo

Anahy-DVM: um módulo de escalonamento distribuído

São Leopoldo
2006

Marcelo Augusto Cardozo

Anahy-DVM: um módulo de escalonamento distribuído

Dissertação submetida à avaliação como requisito parcial para a obtenção do grau de Mestre em Computação Aplicada

Orientador: Prof. Dr. Gerson Geraldo H. Cavalheiro

São Leopoldo
2006

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Cardozo, Marcelo Augusto

Anahy-DVM: um módulo de escalonamento distribuído / por Marcelo Augusto Cardozo. — São Leopoldo: Ciências Exatas e Tecnologicas da UNISINOS, 2006.

66 f.: il.

Dissertação (mestrado) — Universidade do Vale do Rio dos Sinos. Ciências Exatas e Tecnológicas Programa Interdisciplinar de Pós-Graduação em Computação Aplicada, São Leopoldo, BR–RS, 2006. Orientador: Cavalheiro, Gerson Geraldo H.

1. Processamento de Alto Desempenho. 2. Ambiente de Execução. 3. Escalonamento. I. Cavalheiro, Gerson Geraldo H.. II. Título.

"A celibate clergy is an especially good idea, because it tends to suppress any hereditary propensity toward fanaticism."
Carl Sagan

Agradecimentos

Gostaria de agradecer em primeiro lugar aos meus pais, pelo apoio e atenção dados, aos meus amigos por compreenderem as ausências e o eventual mau-humor. Também gostaria de agradecer ao meu orientador, professor Gerson Cavalheiro, por sua atenção e dedicação durante esse período, sempre tentando intercalar reuniões em sua já tão lotada agenda. Por fim, mas não menos importante, gostaria de agradecer a Hewlett-Packard por me proporcionar esta oportunidade.

Resumo

Atualmente o uso de aglomerados de computadores para fins de alto desempenho tem aumentado. Contudo, a programação desse tipo de arquitetura não é trivial. Pois, além de desenvolver a aplicação, detectar e explicitar a concorrência nela existente, o programador também é responsável por implementar o escalonamento de sua aplicação para efetivamente usar o paralelismo dos aglomerados. Existem ferramentas que se propõem a solucionar esses problemas; a ferramenta de programação Anahy é uma destas.

Este trabalho se propõe a implementar um módulo para Anahy com fins de provê-la de suporte à execução em ambientes dotados de memória distribuída. Para isso seu núcleo executivo foi estendido para que se possa ter acesso as estruturas de dados imprescindíveis à distribuição da carga computacional. Também será necessário desenvolver um mecanismo de comunicação entre os nós do aglomerado para que estes troquem as informações necessárias para o andamento da computação. Por fim, o módulo desenvolvido é avaliado através do uso de uma aplicação sintética. Através dessa avaliação, o módulo desenvolvido é analisado em relação a sua usabilidade no contexto do projeto Anahy e, em complementação, é apresentada uma análise preliminar de desempenho.

Palavras-chave: Processamento de Alto Desempenho, Ambiente de Execução, Escalonamento.

TITLE: “Anahy-DVM: a module for distributed scheduling”

Abstract

Lately, the usage of computer clusters has increased. However, programming for this class of architecture is non trivial. This happens due the fact that, besides programming the application, detecting and specifying its concurrency, the programmer is also responsible for coding the scheduler of the application so it can use computer clusters efficiently. There are programming tools that propose solutions for these problems, one of these tools is Anahy.

This work proposes an extension for Anahy runtime in order to provide support for distributed memory environments. In order to achieve this objective, the execution core of Anahy is extended so the necessary data structures can be accessed by this module. It is also necessary to develop a comunication mechanism among the nodes of the cluster so they can exchange the necessary information to complete the computation. Finally, the module is evaluated using a synthetic application. Through this evaluation, the module is analyzed relating to its usability in the Anahy's project context and a preliminary performance analysis is presented.

Keywords: High Performance Computing, Execution Environment, Scheduling.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

ACM Association for Computing Machinery
API Application Programming Interface
APN Arbitrary Processor Network
BNP Bounded Number of Processors
BSP Bulk Syncronous Parallel
COW Cluster of Workstations
DAG Direct Acyclic Graph
DSC Dominant Sequence Clustering
DSM Distributed Shared Memory
FCFS First Come First Served
FIFO First In First Out
NOW Network of Workstations
NUMA Non Uniform Memory Access
PAD Processamento de Alto Desempenho
PRAM Parallel Random Access Machine
PV Processador Virtual
SAC Symposium on Applied Computing
SBC Sociedade Brasileira de Computação
SMP Symmetric Multi-Processor
TDB Task Duplication Based
UMA Uniform Memory Access
UNC Unbounded Number of Clusters

Capítulo 1

Introdução

Nos últimos anos, o desenvolvimento do processamento de alto desempenho (PAD) encontrou um grande aliado nos aglomerados de computadores (*clusters*) e nas arquiteturas multiprocessadoras com memória compartilhada (*Symmetric Multi-Processors*, ou SMPs). No entanto, a exploração dessas arquiteturas com o intuito de obtenção de bom desempenho de execução não é trivial, tendo sido desenvolvidos diversos ambientes de execução, dotados ou não de mecanismos de escalonamento para auxiliar o programador nessa tarefa.

De custo relativamente baixo, os aglomerados e os SMPs têm aumentado sua participação como suporte ao desenvolvimento de programas para aplicações com alto custo computacional. Dentre as razões que motivam esse fato, além do custo, está o potencial de desempenho que pode vir a ser obtido, conforme dados facilmente comprováveis através dos preços aplicados pelo mercado aos microcomputadores bi- e quadri-processados e pela incidência de aglomerados na lista das 500 máquinas mais potentes em operação[1]. Entretanto, a programação dessas máquinas envolve, além da codificação do problema propriamente dito, o mapeamento da concorrência da aplicação, ou seja, as atividades concorrentes no programa, nas unidades de suporte ao cálculo (processador e memória) da arquitetura. A esse mapeamento estão ligadas questões referentes à repartição da carga computacional entre os diferentes processadores e ao compartilhamento de dados entre os nós.

Dessa forma, o uso efetivo de aglomerados e de arquiteturas SMP para o PAD requer a realização do mapeamento da concorrência da aplicação sobre os recursos computacionais disponíveis. Porém, cabe observar que, 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. Portanto, utilizando recursos convencionais de programação concorrente, paralela ou distribuída, é de responsabilidade do programador determinar o número de tarefas concorrentes que a arquitetura utilizada

[1]Lista pode ser encontrada em http://www.top500.org (visitada em 01/2006)

deve manter ativas simultaneamente, assim como 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 de desempenho dos programas [1]. Cilk [2], Athapascan-1 [3], Anahy[4] e PM^2 [5] são ferramentas para o PAD inseridas nesse contexto. Essas ferramentas provêem tanto recursos de programação, para descrição da concorrência de uma aplicação, como também introduzem núcleos executivos capazes de tirar proveito dos recursos da arquitetura visando ao desempenho na execução de programas.

1.1 Definição do problema

A ferramenta de programação Anahy encontra-se em desenvolvimento, não possuindo um mecanismo de escalonamento dinâmico para ambientes de memória distribuída. O atual protótipo para essa arquitetura conta com primitivas, permitindo o desenvolvimento de algoritmos estáticos de escalonamento [6]. Portanto, é necessário desenvolver um escalonador de tarefas dinâmico para Anahy o qual deverá suportar a execução plena de Anahy, tanto em máquinas SMP como em aglomerados de computadores. Os resultados obtidos com a implementação realizada também fornecerão subsídios para a identificação dos custos associados ao escalonamento de tarefas em Anahy utilizando tal núcleo de execução, além de finalizar a atual implementação de Anahy para aglomerados.

Em particular, é buscada uma solução diferenciada das já propostas na literatura, que permite a utilização de um mecanismo de escalonamento de tarefas proporcionando a exploração da localidade no acesso aos dados, com o objetivo de reduzir a sobrecarga de comunicação de tarefas em ambientes com memória distribuída.

1.2 Objetivos

O principal objetivo desse trabalho é a implementação de um escalonador distribuído para a ferramenta Anahy, para que esta contemple as necessidades de execução de uma aplicação paralela sobre aglomerados de computadores. A abordagem de escalonamento dinâmico adotada para a distribuição de tarefas, faz uso de técnicas de escalonamento baseada em algoritmo de lista [7, 8] explorando de roubo de tarefas [2] iniciado pelos nós ociosos. A coleta de resultados de desempenho fornecerá dados quantitativos para a análise das sobrecargas relacionadas às operações de escalonamento.

Pontualmente os objetivos desse trabalho são:

- Estender o núcleo de Anahy para suportar tanto arquiteturas SMP como aglomerados;

- Implementar no núcleo de execução distribuído uma estratégia de escalonamento baseada em um algoritmo de listas;
- Implementar um mecanismo de roubo de trabalho para ambientes de aglomerados;
- Experimentar para consolidar os resultados.

1.3 Metodologia

O presente trabalho foi desenvolvido no contexto do projeto Anahy, cuja arquitetura é apresentada na Figura 1.1. Nesta figura, encontram-se destacados com tons escuros os módulos que se encontram associados ao desenvolvimento do ambiente de escalonamento distribuído proposto.

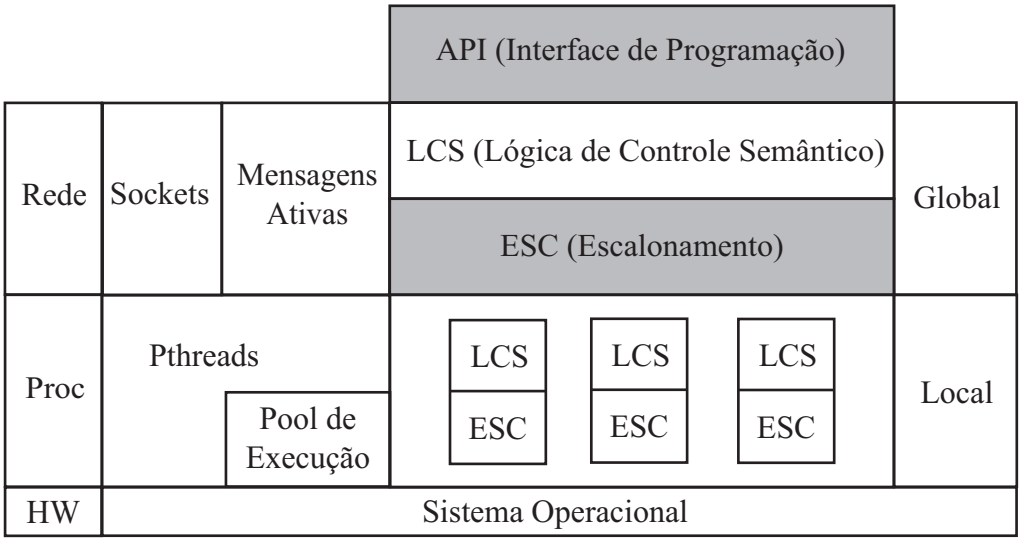

FIGURA 1.1 – Arquitetura de Anahy

O trabalho foi iniciado com o estudo das técnicas de escalonamento envolvendo relações de dependências de dados entre tarefas e de ferramentas de apoio ao desenvolvimento de programas que façam uso dessas técnicas. O estudo teve por objetivo identificar os mecanismos utilizados para criação e manipulação de grafos de dependência entre tarefas e também para suporte à execução (Capítulo 2), permitindo a modelagem e a construção de uma infra-estrutura de execução distribuída de Anahy em ambientes com memória distribuída dotada de mecanismos que permitam otimizar índices de desempenho na execução de aplicações (Capítulo 4).

A interface de programação de Anahy foi estendida para permitir a identificação dos custos relativos à execução das tarefas: custos de processamento e de comunicação. Como premissa de desenvolvimento desse novo conjunto de primitivas, foi considerada a

adotada pelo projeto de Anahy, a qual prevê compatibilidade com o padrão POSIX para *threads*.

O núcleo executivo de Anahy também foi estendido de forma a oferecer ao programador transparência de localização de tarefas e dados na execução de aplicações em ambientes dotados de memória distribuída (Capítulo 5). Para implementação dessa extensão, a premissa de portabilidade de código adotada no projeto Anahy foi igualmente respeitada, assim como a adoção de ferramentas padrões para seu desenvolvimento.

Para a obtenção de dados sobre o comportamento desse novo escalonador de Anahy, uma aplicação sintética foi escolhida. A qual deverá descrever um fluxo de dados em suas tarefas concorrentes de maneira a explorar o escalonador e evidenciar os custos envolvidos em seus algoritmos (Capítulo 6).

1.4 Organização

No Capítulo 2 é apresentado um breve relato do estado da arte na pesquisa de escalonamento em sistemas de memória distribuída, mostrando como aplicações são escalonadas e como heurísticas são utilizadas para otimizar algum índice de desempenho. Também são apresentadas algumas ferramentas de programação que utilizam tais heurísticas, sendo destacado o uso de grafos no escalonamento, assim como seus atributos. Por fim, apresenta-se o modelo de Graham, o qual permite inferir qual o tempo mínimo de execução de uma aplicação em um ambiente de execução que utilize algoritmo de listas.

No Capítulo 3 expõe-se o ambiente de execução Anahy, identificando-se a arquitetura alvo desse ambiente, assim como sua interface de programação, sendo também mostrada sua sintaxe e seu núcleo de escalonamento. O capítulo conclui mostrando os serviços básicos detectados em Anahy, os quais devem estar presentes em sua implementação distribuída.

No Capítulo 4 apresenta-se o modelo da extensão de Anahy para contemplar o suporte a arquiteturas distribuídas, que consiste em serviços de comunicação que devem ser criados. Além disso, é explicado o funcionamento do escalonador distribuído, bem como a apresentação das estratégias nele utilizadas.

No Capítulo 5, a implementação do suporte é detalhada apresentando as primitivas inseridas no ambiente para permitir o programador tirar proveito da arquitetura distribuída. Também são detalhados como os serviços detectados no Capítulo 3 foram implementados.

No Capítulo 6 são apresentados os resultados atingidos com essa implementação e, finalmente, no Capítulo 7, as conclusões tomadas.

Capítulo 2

Estado da Arte

Neste capítulo são apresentados conceitos básicos utilizados no escalonamento de aplicações, como também as estratégias utilizadas para otimizar o escalonamento. São mostradas algumas ferramentas de programação que auxiliam na criação de aplicações para ambientes dotados de mais de um recurso de processamento.

2.1 Escalonamento de aplicações

O escalonamento [9] é parte fundamental de um ambiente de execução, pois são os mecanismos associados a ele que permitem a exploração dos recursos da máquina para efetivar a execução de aplicações. A tarefa principal do escalonamento é atribuir as tarefas da aplicação que devem ser executadas às unidades de execução da arquitetura. A literatura apresenta também o termo alocação no contexto da discussão sobre o escalonamento. Então, Casavant e Kuhl [9] caracterizam o uso desses termos: escalonamento está associado à aplicação e trata dos custos computacionais gerados pela aplicação em execução, como atividades de processamento ou dados e da utilização definida pelo algoritmo de escalonamento; Já, o termo alocação, ou mapeamento, refere-se ao mesmo problema, porém sob a ótica do recurso de processamento a ser utilizado. Nesse trabalho a visão privilegiada é a de escalonamento, sendo adotada a taxonomia proposta em [9].

Existem duas classes de algoritmos de escalonamento: estáticos e dinâmicos. A adequação de uma aplicação a uma ou outra classe depende da sua estrutura, assim como do seu comportamento durante a evolução na computação. Aplicações cujas relações de dependência entre tarefas sejam conhecidas antes de sua execução, podem ser submetidas a estratégias de escalonamento estático. Já, as aplicações, cujo grafo descrevendo o relacionamento entre tarefas somente é conhecido durante a evolução do programa, devem ser submetidas a técnicas de escalonamento dinâmico. Nesse caso, o algoritmo de escalonamento é concebido de forma a reagir à evolução do programa refletida nas modificações do grafo [10, 11].

Considerando a evolução do grafo gerado pelas aplicações dinâmicas, Konig e Roch observam que estas podem ser classificadas em dois grupos distintos: regulares e irregulares [12]. As aplicações ditas regulares possuem uma estrutura de execução previsível. Dessa forma, a estratégia de escalonamento pode ser desenvolvida considerando um padrão no relacionamento entre tarefas, embora não sejam conhecidos nem o número de tarefas a serem executadas, nem os custos computacionais associados. Para as aplicações irregulares, não é possível identificar um padrão na estrutura de execução, dificultando as operações de escalonamento, em particular quando se busca otimizar algum índice de desempenho.

Deve-se notar que otimização de índices de desempenho é resultado de alguma política de distribuição de carga empregada pelo mecanismo de escalonamento. Durante o escalonamento podem ser aplicadas estratégias com objetivo de distribuir a carga computacional gerada pelo programa em execução pelos recursos de processamento disponíveis. Nesse trabalho as estratégias trabalhadas serão voltadas a minimizar o tempo total de processamento, sendo manipuladas as atividades de cálculo geradas pelo programa, embora técnicas semelhantes possam ser aplicadas a qualquer outra fonte de custo, como dados ou comunicações. Diversas heurísticas de escalonamento [13, 7, 14, 15, 16, 8] exploram conhecimento sobre a estrutura do programa para otimização de índices de desempenho. Tais estratégias servirão de base de estudo para implementação de um suporte ao escalonamento dinâmico em arquiteturas do tipo aglomerado de computadores [17].

Feitelson relatou em 1995 [18] que, embora as pesquisas sobre técnicas de escalonamento explorando a estrutura de execução de programas fossem populares, sua exploração prática em ambientes de execução era bastante reduzida. Esta realidade é ainda observada, existindo poucas opções (como Athapascan-1 [3] e Cilk [2]) desenvolvidas com esse propósito. O problema que se coloca é como realizar o escalonamento em nível aplicativo [19], ou seja, como permitir que a estratégia de escalonamento seja realizada de forma associada ao programa em execução de forma a obter otimizações de desempenho.

Para explorar eficientemente uma arquitetura paralela, o programa deve ser dividido em tarefas concorrentes e uma estratégia de escalonamento deve garantir eficiência no uso dos recursos de processamento. Entretanto, as ferramentas de programação clássicas exigem do programador conhecimentos específicos referentes à arquitetura sobre a qual será executado seu programa, pois é sua responsabilidade tanto codificar a aplicação, como distribuir tarefas a processadores e dados a módulos de memória [20]. Assim, além de estar programando sua aplicação, também é de sua responsabilidade introduzir instruções para realização do escalonamento do programa. Isso é ineficiente, pois além de obrigar o programador a usar uma linguagem de mais baixo nível (a do escalonamento), também faz com que a aplicação não seja portável, já que está intimamente ligada à arquitetura para a qual foi programada. A menor alteração nessa arquitetura, como, por exemplo, a incorporação de novos processadores, pode não refletir em ganho de desempenho na

aplicação. O uso dessas ferramentas gera programas não escaláveis, conflitando com a perspectiva de alta variação de configurações em aglomerados de computadores.

Existem ambientes de execução que fornecem ao programador uma interface de programação, também chamada de API (*Application Programming Interface*), dissociada do núcleo executivo. Alguns exemplos de ambientes são Athapascan-1 [3], Cilk [2], Jade [21], Nano-threads [22] e PM^2+ GTLB [5]. Os ambientes são capazes de explorar os recursos do sistema com o intuito de otimizar algum índice de desempenho nas execuções de aplicações devido ao uso de escalonamento em dois níveis, isto é, dissociam o escalonamento das tarefas paralelas da aplicação da alocação dos recursos da arquitetura. Dessa maneira o sistema operacional fica responsável pelo escalonamento da arquitetura e o ambiente de execução escolhe quais tarefas da aplicação estão aptas a usarem o recurso escalonado pelo sistema. A estratégia mais comum é utilizar grafos para representar o relacionamento entre tarefas de um programa em execução. O grafo é explorado pelo escalonador para tomadas de decisões sobre ativações de tarefas.

Uma questão a ser considerada no escalonamento dinâmico de aplicações descritas por grafos de dependência é a sobrecarga gerada pela manipulação do próprio grafo. Essa sobrecarga não é considerada em algoritmo de listas (*ex.* Graham) embora sejam consideradas na implementação de ambientes (*ex.* Cilk). Apesar de não poderem ser eliminadas, as sobrecargas devem ser evitadas para que se obtenha algum ganho na execução paralela das tarefas concorrentes.

2.2 Níveis de escalonamento

Em [19] é proposto que o escalonamento pode ser dividido em dois momentos distintos. No primeiro momento o sistema operacional escolhe qual o recurso que será alocado à aplicação. Já, no segundo momento a aplicação escolhe qual de suas tarefas concorrentes usará o recurso alocado pelo sistema operacional. Dessa maneira temos dois níveis de escalonamento, o de sistema, realizado pelo sistema operacional; e o de aplicativo, realizado pela aplicação.

O escalonamento de sistema tem por objetivo manter em uso os recursos computacionais disponíveis, não considerando características particulares das aplicações, sendo que a atenção, nesse texto, é dada ao escalonamento aplicativo.

O escalonamento em nível aplicativo [9] é feito no ambiente de execução da aplicação do usuário. Em Casavant e Kuhl [9], o escalonamento é apresentado como uma camada de decisão, situada entre a aplicação, que é geradora das necessidades de consumo de recursos, e o *hardware*, que provê os recursos de computação (Figura 2.1). Quando implementado em nível aplicativo, o escalonador é uma camada de *software* que interage tanto com a aplicação quanto com o *hardware* para efetivar a execução da aplicação segundo políticas de decisão, que podem se limitar a executa-las, pura e simplesmente, ou, considerando uma

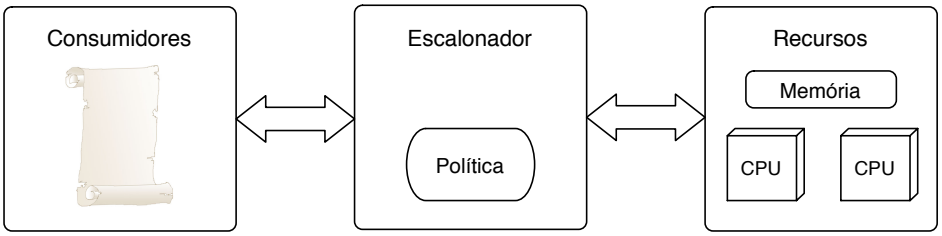

FIGURA 2.1 – Modelo de escalonamento [9]

arquitetura dotada de múltiplos recursos de processamento, realizar a execução buscando ganho em algum índice de desempenho. Assim, o escalonador em nível aplicativo pode fazer com que, quando a aplicação ganha a fatia de tempo do processador, execute a tarefa que lhe é mais vantajosa naquele instante de tempo.

Como um sistema operacional genérico não pode ser programado para escalonar uma aplicação particular, o uso de escalonamento em dois níveis tem potencial de otimizar a execução de uma aplicação e assim ganhar desempenho. O escopo deste trabalho é o escalonamento aplicativo, deixando a alocação dos recursos de processamento a cargo do sistema operacional.

2.3 Núcleo de escalonamento

Quando foi introduzido em 1945, o modelo de computador de von Neumann estabeleceu um padrão para máquinas seqüenciais e permitiu o surgimento das primeiras linguagens de programação portáveis. Entretanto, isso ainda não ocorreu para máquinas paralelas [23]. Vários modelos foram propostos, como PRAM [24], BSP [23] e LogP [25]. Todavia, nenhum obteve o sucesso equivalente ao que o modelo de von Neumann teve para máquinas seqüenciais, por isso ainda não há um modelo de máquina paralela padronizada sobre a qual criar linguagens de programação portáveis. O modelo de von Neumann formaliza que as tarefas (as instruções) de um programa são escalonadas uma a uma na ordem lexicográfica em que se encontram definidas no código. Por se tratar de um modelo de máquina que possui um único processador e um único espaço de endereçamento, essa ordem de execução garante o correto acesso aos dados residentes na memória. Atualmente quando um programador deseja utilizar-se de alguma linguagem paralela, ele deve aprender o modelo de programação dessa linguagem e organizar as dependências entre as tarefas de forma a promover a correta manipulação de dados. Isso gera uma série de dificuldades fazendo com que a programação paralela não seja a plataforma de escolha padrão para a computação, pois sem tal modelo não é possível criar programas que sejam portáveis e rodem com desempenho aceitável em aglomerados ou em redes de

computadores (*NOW/COW*).

Embora a camada de escalonamento aplicativo possa ser parte integrante de aplicações, alguns ambientes propõem dissociar o código do escalonamento do código da aplicação. Isso é visto em Athapascan-1, Cilk, Anahy, entre outros. Nesses ambientes a comunicação entre as camadas de escalonamento e aplicação é garantida por uma interface de serviços [10, 20, 11] que permitem ao escalonador ter conhecimento da evolução da execução do programa. Em muitos casos, essa interface permite a criação de grafos representando o relacionamento entre as tarefas da aplicação.

2.4 Ferramentas

Nesta sessão serão apresentadas algumas ferramentas de programação utilizadas em aglomerados de computadores, as quais possuem em comum a capacidade de dissociar a camada de aplicação do suporte executivo. Serão analisados quesitos como escalonamento, distribuição ou balanceamento de carga e métodos de comunicação entre nós.

2.4.1 DPC++

Esta ferramenta é uma extensão da linguagem C++ que permite ao usuário instanciar de objetos distribuídos. Também permite a execução transparente de aplicações, ou seja, o usuário não necessita, de forma explícita, especificar em que nó seus objetos irão executar.

DPC++ [26] é estruturado em dois níveis: o nível de linguagem e o de operação. O nível de linguagem corresponde ao uso das extensões na programação de uma aplicação. O nível de operação corresponde ao uso de pré-processadores que traduzem a sintaxe de DPC++ para C++. Também é de responsabilidade desse nível, a inserção das ferramentas de comunicação, localização de objetos e detecção de carga.

O paralelismo é expresso na forma de objetos comunicantes, sendo cada objeto composto por dados e métodos que agem sobre os dados. A comunicação pode ser feita de maneira síncrona ou assíncrona e é iniciada pela execução de um método do objeto receptor da mensagem, não havendo controle da coerência no acesso aos dados em nível de execução. As execuções dos métodos de comunicação são tratadas em ordem de chegada.

O escalonamento em DPC++ é realizado em nível de objeto e não em nível de método. Uma extensão ao modelo original introduz concorrência interna aos objetos. Todas as chamadas que resultam na criação de um objeto distribuído são enviadas a um módulo central responsável pela distribuição da carga levando em consideração uma tabela que contém todas as informações de carga de todos os nós. Após a colocação do objeto em um nó ter sido realizada, todas as chamadas a métodos desse objeto são enviadas diretamente ao nó que o contém. Caso um nó sature, ou seja, esteja sobrecarregado de

trabalho, um processo de migração de objetos é posto em prática, realizando assim, um balanceamento de carga. O escalonamento dos processos sobre os processadores do nó é de responsabilidade do sistema operacional.

A comunicação utilizada por DPC++ é a invocação remota de métodos (*RMI*), em que o objeto de origem da comunicação chama um método específico do objeto de destino, sendo os dados da comunicação passados como parâmetros dessa chamada.

2.4.2 PM^2/GTLB

GTLB [5] é um módulo que provê suporte a escalonamento para *threads* para o ambiente PM^2. A arquitetura assumida por GTLB é de uma máquina multiprocessada que possui uma memória compartilhada. Cada *thread* criada tem o início de sua execução imediata. As *threads* são executadas de maneira completamente assíncrona, tendo o seu acesso à memória compartilhada utilizando a política *first come, first service*. Não há garantia de coerência no acesso a memória compartilhada, tendo este de ser implementado pelo usuário para que a execução de sua aplicação seja correta. A última versão disponível data de 1998.

O escalonador de GTLB utiliza um algoritmo de balanceamento de carga e tem por unidade *threads* concorrentes. O algoritmo pode, a qualquer momento, interromper e migrar uma *thread* de um nó a outro para fins de balanceamento de carga. É responsabilidade de um *daemon* controlar o balanço da carga entre os nós para evitar a situação onde um deles fique sem carga. Este *daemon* é parte integrante do escalonador e roda em paralelo à aplicação do usuário.

Na estratégia de escalonamento adotada, a carga de um nó reflete a carga total do sistema. Apenas a quantidade de tarefas concorrentes é levada em consideração. A unidade de custo é o número de *threads* em execução e a política implementada busca manter balanceada a carga entre os nós. Assim, variações no número de *threads* em cada nó podem implicar uma operação de escalonamento. Essa política é implementada considerando que os processadores encontram-se organizados em um anel. Quando o escalonador é acionado, uma mensagem é enviada ao nó da esquerda do nó corrente; então, o nó toma decisão sobre a migração de carga de acordo com a informação recebida, isto é, decide se *threads* serão migradas e qual dos dois nós irá migrá-las. O processo é repetido até que a mensagem, descartada ao final, chegue ao nó o qual desencadeou esse processo.

2.4.3 Millipede

O projeto Millipede, desenvolvido pelo Instituto de Tecnologia de Israel, tem por objetivo um ambiente de alto desempenho para a execução distribuída de aplicações, mas não define uma linguagem a ser utilizada, pois apenas os ambientes de compilação e de

execução são definidos. Por intermédio do uso das interfaces aplicativas (APIs), linguagens como HPF e Java podem ser utilizadas. O modelo de programação de Millipede é baseado em tarefas que se comunicam através de uma memória compartilhada. O ambiente cria, então, uma máquina virtual composta de vários nós de execução não dedicados que implementam um mecanismo de acesso a memória global.

A expressão do paralelismo em Millipede é feita através das tarefas que são criadas e executadas de maneira concorrente e utilizam a memória global para a troca de dados. A criação de uma tarefa implica sua execução imediata, que dependendo da forma que foi criada, pode ser tanto no nó que a criou, como em outro nó remoto. Mensagens são enviadas entre as tarefas para permitir o controle da execução. A memória global é mantida sobre o sistema de paginação que se encontra distribuído sobre os módulos de memória dos nós da máquina paralela virtual, não havendo qualquer garantida de coerência na execução da aplicação, além da incluída explicitamente pelo programador.

O escalonamento do Millipede está baseado no princípio da migração de tarefas, e o principal objetivo do escalonador é controlar o seu número em execução em cada nó, considerando os custos de comunicação gerados pelo acesso à memória global. O mecanismo de controle do acesso à memória global permite ao escalonador contabilizar os acessos de cada tarefa a um dado específico. Utilizando-se dessa informação, o escalonador pode então escolher de maneira mais otimizada para onde migra uma tarefa de um nó sobrecarregado. Uma característica do escalonador de Millipede é a independência entre a política de migração de tarefas e a política da escolha de tarefas para migração. Dessa forma o programador pode definir sua própria política de migração de tarefas a qual, entretanto, é sempre realizada pelo ambiente. Isso se dá porque esta política tem de ter acesso ao gerenciador da memória global, para que tire proveito da localidade dos dados, considerando os custos gerados pela comunicação entre as tarefas comunicantes.

As informações básicas de que o ambiente trata são o número de tarefas em execução em um determinado nó e o número de acesso às páginas da memória global. A informação sobre os acessos à memória global são utilizados pelo nó para determinar qual tarefa seria melhor migrar. Já, as informações sobre o número de carga são utilizadas por cada nó para disparar uma operação de migração. Uma informação extra, adicionada em nível de escalonamento, permite detectar se um nó está sendo utilizado de forma interativa. Sendo esse o caso, o nó é marcado como não apto a receber tarefas.

2.4.4 NPM - *Nano-thread Programming Model*

NPM [22] é um modelo de programação *multithread* desenvolvido para SMP e aglomerados. Seu principal objetivo é a paralelização automática de aplicações e a eficiente execução das mesmas em ambientes multi-processados. NPM se utiliza de um compilador que extrai o paralelismo do programa do usuário, analisando a dependência de dados para criar uma representação do programa chamada de Grafo de Tarefas Hierárquico (*Hierar-*

chical Task Graph ou HTG). Essa representação da decomposição do programa pode ser variável, começando em uma única tarefa representando todo o programa até a todas as possíveis tarefas concorrentes que o programa possui. Cada vértice do grafo HTG representa uma tarefa que é executada em uma *thread*, já que NPM utiliza-se da dependência de dados para controlar a ordem de execução das tarefas, pois todos os serviços de criação e de escalonamentos das *threads* são deixados a cargo de um ambiente de execução.

2.4.5 Cid

Esta ferramenta é uma extensão do C, desenvolvida para máquinas com memória distribuída. Suas aplicações alvo são do tipo de carga recursiva que descrevem estruturas de dados do tipo árvore, listas ou grafos. Um programa Cid em execução consiste de tarefas concorrentes que tem acesso a uma memória compartilhada para o envio dos parâmetros das tarefas, assim como para coletar seus resultados.

Cid [27] utiliza-se dos mecanismos *fork/join* para expressar o paralelismo. Uma chamada *fork* faz com que a tarefa pai e a filho sejam executadas em paralelo. A primitiva *join* é opcional e faz com que o fluxo de execução da tarefa pai espere pelo término de um filho, podendo, também, executar um *join* entre vários filhos. Entre as tarefas existe a noção de uma memória compartilhada, que é mantida distribuída entre os módulos de memória privados de cada processador. Cid permite coerência no acesso à memória compartilhada utilizando-se de uma estrutura de dados própria do ambiente e de exclusão mútua no acesso à memória.

O escalonamento associa múltiplos fluxos de execução a um mesmo processador. Dessa forma obtém-se um mascaramento dos tempos de acesso à memória compartilhada. O escalonamento em Cid é ativado em resposta a um evento realizado pela aplicação em execução, que normalmente é a criação de uma tarefa, uma chamada do tipo *join* ou um processador que ficou ocioso, por isso Cid oferece ao programador qual o tipo de tratamento que deve ser executado quando uma tarefa é criada. Assim, quatro tipos de operação *fork* são oferecidas:

- *fork* com controle de carga: o escalonador encontra o processador com menor carga e o encarrega de executar a nova tarefa.

- *fork* com controle de migração: quando nenhum processador esta disponível, a tarefa não é criada e a execução é feita por uma simples chamada de função. Se existe um processador inativo, a tarefa é criada e imediatamente migrada para este processador.

- *fork* com controle de localização: é um *fork* onde é indicado o dado na memória compartilhada que será manipulado pela nova tarefa. Se o dado se encontra na região privada de um processador e o pai dessa tarefa também se encontra no mesmo

processador, a nova tarefa é executada no mesmo fluxo de execução do seu pai. Caso contrário um novo fluxo de execução é criado no processador o qual possui o dado.

- *fork* com controle de granulosidade: o sistema é encarregado de definir o número de tarefas que devem ser criadas para executar o tratamento de um conjunto de dados.

Quando uma tarefa executa um *join*, ela fica bloqueada esperando sincronização e, se essa for satisfeita, a tarefa é posta novamente na lista para que a retomada do seu fluxo de execução seja possível. Quando um processador fica inativo, um mecanismo permite que este obtenha trabalho. O processador ocioso então envia um pedido de trabalho a algum outro processador que deve responder enviando uma tarefa de sua lista, caso contrário é feito um *fork* com controle de migração. Não é possível realizar a preempção de tarefas e migras as que estão em execução.

2.4.6 Cilk

Cilk é uma extensão à linguagem C que provê suporte a *threads* concorrentes em arquiteturas SMP as quais são criadas explicitamente, sendo que toda a sincronização entre *threads* seja realizada através de uma memória compartilhada. A sincronização é feita de maneira igualmente explícita e permite que uma *thread* espere o término de todas as outras criadas por ela. Dessa maneira, permite ao programador o controle da coerência no acesso aos dados contidos na memória compartilhada.

A concorrência em Cilk é explicitada através da primitiva *spawn*. Uma função chamada através dessa primitiva tem sua execução concorrente com a *thread* que a chamou. Esta *thread* continua sua execução sem receber o retorno dessa função concorrente. Quando necessita utilizar o resultado, a *thread* deve, de maneira explícita, utilizar a primitiva de sincronização *sync*, fazendo com que a *thread* espere o término da execução de todas as funções concorrentes por ela chamadas antes da execução do *sync*. Dessa forma, tem-se uma expressão de paralelismo do tipo série-paralelo, havendo então, risco de concorrência ao acesso à memória compartilhada, pois as funções concorrentes criadas em teoria são independentes. A concorrência ao acesso à memória deve ser regida pelo programador através da manipulação das primitivas *spawn* e *sync*. Através da manipulação dessas primitivas, o escalonador de Cilk é capaz de gerar um grafo de precedência entre as *threads*, como também, através do uso destas primitivas, é possível definir um pedaço de código, denominado de tarefa, que uma vez iniciado é terminado sem necessitar sincronização.

O escalonador de Cilk pressupõe o uso de uma memória compartilhada acessível por todos os processadores, uma vez que a política de escalonamento é baseada em um algoritmo de *work stealing* (roubo de trabalho), realizando assim, a distribuição da carga entre os processadores. Cada processador executa suas tarefas dando prioridade a profundidade no grafo. Na prática, isso faz com que quando uma tarefa é criada, ela seja

imediatamente executada. Caso não haja processadores ociosos, a tarefa predecessora da criada é posta em espera, aguardando o término da outra recém criada. Se essa tarefa posta em espera não afeta outros processadores, então ambas serão executadas em paralelo. O programador pode definir relações de dependência entre as tarefas através do uso de primitivas de sincronização, já que a garantia da correta execução é assistida pelo processo de compilação. Quando um processador torna-se ocioso, este escolhe outro processador de maneira aleatória e executa o mecanismo de roubo de trabalho. Este algoritmo será apresentado com mais detalhes na Seção 2.7.

Cilk foi desenvolvido para arquiteturas SMP, por isso não possui um mecanismo de comunicação entre nós. Entretanto, foi proposta uma versão distribuída em [28]. Nessa versão, os princípios de roubo de trabalho, assim como o do determinismo da execução, foram mantidos. Para garantir o determinismo, é necessário manter a integridade no acesso a memória compartilhada, sendo que o mecanismo adotado para garantir essa integridade, entretanto, insere custos no processo de escalonamento. Afim de reduzir o impacto desses custos, o escalonador introduz uma noção de localidade dos dados. Dessa forma, o roubo de trabalho não é feito de maneira totalmente aleatória, visto que os processadores localizados no mesmo nó, tendo o acesso à mesma memória física, tem maior probabilidade de serem escolhidos.

2.4.7 Jade

Jade [29, 21] é uma extensão da linguagem C, feita através da noção de um bloco de instruções. Cada bloco tem anotado os dados por ele acessados e seus direitos sobre eles (leitura/escrita). O paralelismo é implícito, e durante a execução, cada entrada em um bloco é interpretada como a criação de uma tarefa, a qual tem suas entradas e saídas identificadas, possibilitando modelar a execução de um programa Jade como um fluxo de dados. A semântica de Jade dita que qualquer leitura de um dado lê a ultima escrita relativa à ordem de execução seqüencial; logo, para implantar essa semântica, o núcleo executivo de Jade gera um grafo de precedência distribuído.

Um programa Jade é um programa C onde foram inseridas diretivas de exploração do paralelismo, permitindo, assim como o Cilk, a criação de tarefas associadas a um fluxo de execução independentes. Em cada tarefa são utilizados operadores que identificam os acessos feitos à memória global. Contudo, ao contrário de Cilk, uma tarefa criada não é necessariamente uma tarefa pronta para ser executada. Ela pode ser considerada não pronta caso uma (ou várias) tarefas predecessoras no grafo não tenham sido terminadas, ou seja, os dados de entrada ainda não se encontram disponíveis. Assim, a concorrência de execução das tarefas é limitada pelos acessos aos dados.

O escalonamento em Jade é centralizado e aproveita os dados contidos no grafo de fluxo de dados para explorar a sua localidade, baseado em uma lista de tarefas escolhidas de acordo com as referências de acesso à memória global. Uma tarefa criada é

inserida nessa lista em função dos dados que ela manipula. Um processador que termina a execução de uma tarefa procura por outra dentro dessa lista que acesse o dado que acabou de ser produzido. Não existe preempção de tarefas em Jade; caso, nenhuma tarefa for encontrada, outro bloco, onde tenha ao menos uma tarefa pronta, é recuperado pelo processador para iniciar sua execução.

2.4.8 Athapascan-1

Athapascan-1 [3] é uma ferramenta para programação paralela baseada em fluxo de dados. Seu paralelismo é definido explicitamente e é expresso através de chamadas assíncronas de funções denominadas *tarefas*, que se comunicam utilizando-se de uma memória compartilhada. A semântica de Athapascan-1 se baseia no acesso aos dados presentes na memória compartilhada, assegurando que o valor lido de uma variável que esteja na memória compartilhada seja o último valor escrito de acordo com a ordem lexicográfica da aplicação. A principal vantagem desse tipo de semântica é que o fluxo de dados pode ser lido pelo programador direto do código fonte.

O paralelismo é expresso através da definição de tarefas. Uma tarefa tem como entrada as variáveis que vai utilizar da memória compartilhada. O tipo de acesso a estas variáveis deve ser especificado. A tarefa só é disparada quando chamada através da primitiva *fork* que é assíncrona, portanto, a execução do programa segue a próxima instrução na ordem lexicográfica do ponto onde foi chamado o *fork*. A tarefa criada pelo *fork* é considerada apta para execução quando todas as variáveis que por ela serão lidas, encontram-se disponíveis na memória compartilhada.

O escalonamento em Athapascan-1 é baseado na detecção do paralelismo e no controle da evolução dos dados presentes na memória compartilhada, sendo que a detecção do paralelismo é feita através do tipo de acesso que uma tarefa faz a um determinado dado, ou seja, quais dados ela irá ler e escrever na memória compartilhada. Dessa forma, é possível para o ambiente determinar a precedência entre as tarefas, pois, apenas quando todas as variáveis forem lidas por uma tarefa forem consideradas prontas é que a tarefa é considerada pronta para execução. Uma variável é considerada pronta se todas as possíveis escritas diretas e indiretas sobre ela já foram resolvidas. O controle é realizado sobre um grafo de dependência de tarefas. Toda tarefa que é criada é inserida nesse grafo, assim como as variáveis por ela modificadas.

A execução de Athapascan-1 em um ambiente de memória distribuída compartilhada é possível através do uso do Athapascan-0. Por esta razão, uma máquina virtual é criada, através da especificação dos nós, entre os quais a comunicação é realizada através do envio de mensagens. O grafo da execução da aplicação é distribuído. Sua manutenção é local e apenas as transições das variáveis que são comuns entre os nós são replicadas. Dessa forma, toda as tarefas tem acesso local aos dados por elas manipulados.

2.5 Grafos como base para escalonamento

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 então criar um grafo orientado (*DAG*) da execução da aplicação. Esse grafo pode ser considerado uma interface entre o programa em execução e o escalonador [20, 10].

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 [30].

Podem-se inferir dados importantes sobre uma aplicação que é descrita na forma de grafo de precedência de tarefas, tais como, qual o grau de paralelismo que pode ser atingido e qual o seu caminho crítico, que é um dos dados mais importantes que se pode obter de um grafo, pois no caso de aplicações paralelas, ele indica o caminho no qual não se obtém ganho com paralelismo devido à dependência de dados entre as tarefas desse caminho. Dependendo do tipo de estratégia utilizada, pode-se agregar maiores informações ao grafo. Por exemplo, pesos podem ser associados a cada arco, indicando a quantidade de dados que é comunicada, e a cada vértice, indicando o tempo de cálculo da tarefa. Caso esses pesos sejam omitidos, então o grafo fica sendo apenas de dependências entre tarefas.

Na Figura 2.2 é apresentado um exemplo de grafo de dependência entre tarefas de um programa concorrente hipotético. Nele pode-se observar que o caminho crítico dessa aplicação seria formado pelas tarefas 1, 2, 5, 6, 7, 9 e 10 supondo que o custo de comunicação e o custo de execução seja unitário. Isso poderia guiar um algoritmo de escalonamento de maneira a pôr todas as tarefas pertencentes ao caminho crítico ao mesmo processador, para minimizar os custos de comunicação.

Uma aplicação pode não instanciar todas as suas tarefas concorrentes em tempo de carga. Dessa maneira, tarefas são criadas e removidas à medida que outras são executadas. Portanto, não há como saber de antemão qual é o grafo da aplicação. Sendo assim necessário o uso de heurísticas para se utilizar o grafo parcial a fim de se chegar a uma solução, considerando situações onde a aplicação seja regular ou, pelo menos, semi-regular [12].

2.5.1 Atributos de grafos

Quando um grafo representa uma aplicação paralela, ele possui características que podem ser utilizadas para saber de ante-mão como tal aplicação se comporta. Sendo um

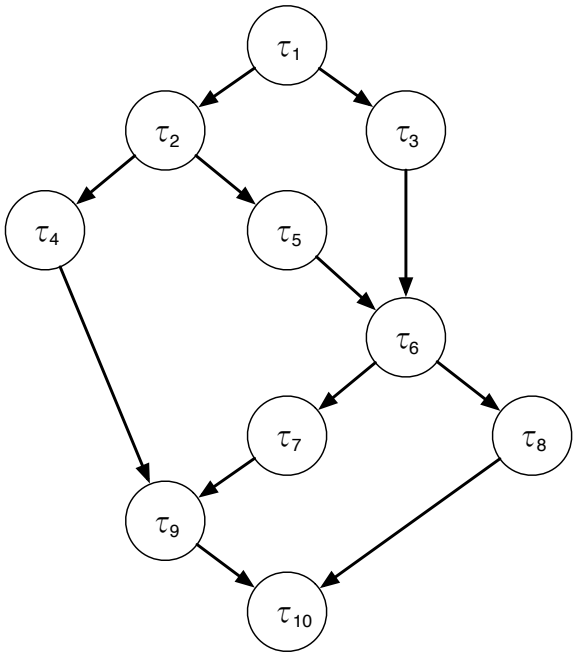

FIGURA 2.2 – Exemplo de um grafo orientado acíclico

grafo $\mathcal{G}$, o grafo que representa uma aplicação tem-se os seguintes atributos:

- T_s - tempo da execução seqüencial de $\mathcal{G}$. Este tempo corresponde ao tempo de execução obtido pela melhor implementação seqüencial do algoritmo
- T_1 - tempo da execução paralela de $\mathcal{G}$ em uma arquitetura dotada apenas de 1 processador. Este tempo corresponde ao trabalho total executado pela implementação concorrente do algoritmo.
- T_p - tempo da execução paralela de $\mathcal{G}$ em uma arquitetura dotada de p processadores. Este tempo corresponde ao tempo decorrido entre o lançamento e o término do programa em uma arquitetura paralela.
- T_∞ - tempo da execução paralela de $\mathcal{G}$ em uma arquitetura dotada de infinitos processadores. Medida de tempo para aferição de desempenho.

Um grafo anotado é um grafo em que o peso dos vértices indica o custo de execução da tarefa e o peso da aresta o custo de comunicação dos dados produzidos. Podem-se ver algumas das relações apresentadas no grafo anotado da Figura 2.3a, onde está anotado o custo de execução de cada tarefa. Neste exemplo tem-se T_s como o somatório de todos os pesos presentes, sendo portanto 28 unidades de tempo. Se a arquitetura possuir um número de processadores igual ou maior do que a concorrência da aplicação para que pudesse executar paralelamente todas as tarefas sem relação de dependência, o tempo de execução da aplicação seria T_∞, que nesta aplicação é 17 unidades de tempo

e é atingido através da execução do caminho crítico formado pelas tarefas 1, 2, 5, 6, 7, 9 e 10. Um *speedup* de aproximadamente 1,65 pode ser alcançado com essa aplicação. Entretanto, devido aos custos associados ao controle de execução, este limite é apenas teórico. Assim $T_1 = T_s + \theta$, onde θ representa toda sobrecarga associada à execução do programa concorrente. Nessa análise foram considerado somente os custos inerentes ao acesso a uma memória compartilhada, porém não distribuída.

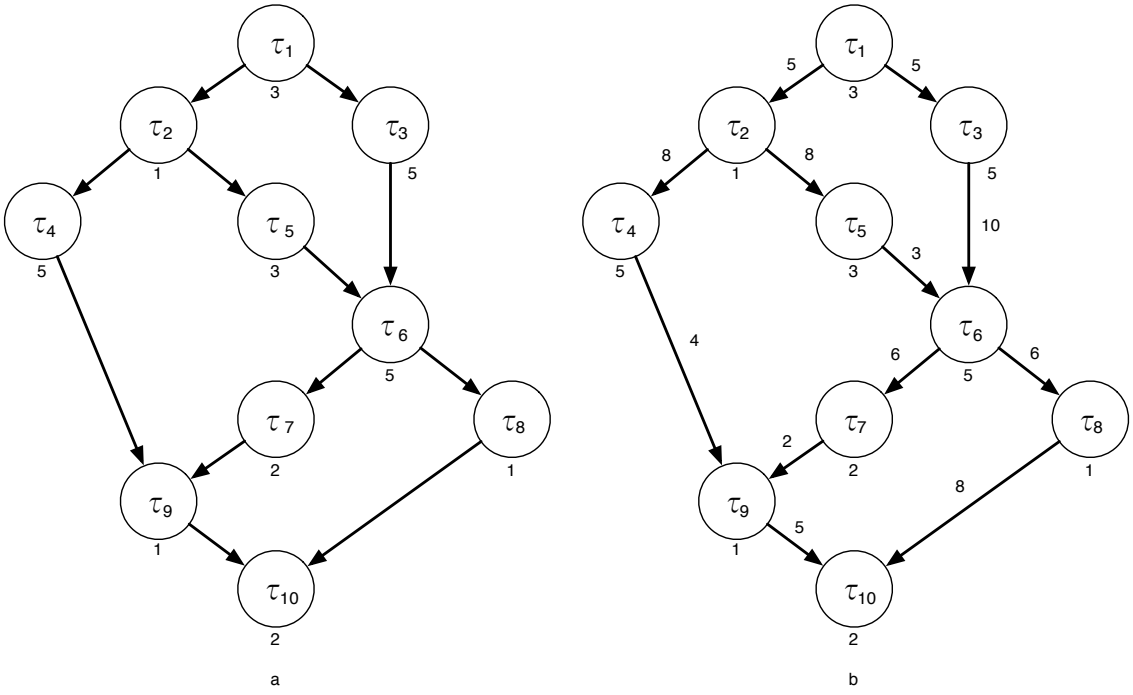

FIGURA 2.3 – Exemplo de grafo anotado

Em se tratando de memória distribuída compartilhada, existem atributos análogos aos apresentados acima, porém, referentes à memória. Dessa forma, podem ser citados:

- S_1 - memória consumida durante a execução de $\mathcal{G}$ em uma arquitetura dotada apenas de 1 processador;
- S_p - memória consumida durante a execução de $\mathcal{G}$ em uma arquitetura dotada de p processadores;
- C_1 - quantidade de comunicação necessária para a execução de $\mathcal{G}$ em um processador com módulo de memória distribuído;
- C_∞ - quantidade de comunicação no caminho crítico. É o maior volume de comunicação realizado entre tarefas de acordo com as relações de precedência.

Na Figura 2.3b temos o grafo, agora, anotado dos custos de comunicação entre as tarefas. De posse desses custos pode-se, por exemplo, verificar se a execução de uma

tarefa em outro nó acarreta em maiores custos de comunicação do que se ganharia em cálculo efetivo.

Um conceito importante em grafos é o da granularidade [31]. Uma granularidade ρ de um grafo $\mathcal{G}$ é a razão entre o menor peso de um vértice e o maior peso de um arco em $\mathcal{G}$. Caso ρ seja menor do que 1 então $\mathcal{G}$ é chamado de grão fino, senão é chamado de grão grosso. Essa razão é utilizada para saber a relação entre cálculo efetivo e comunicação, pois quanto maior o ρ, maior é o tempo de cálculo em relação ao de comunicação e, portanto, esse tempo de comunicação pode ser sobreposto com o de cálculo.

Por fim, trabalhando-se com arquiteturas distribuídas, é necessário levar em consideração tanto os custos de comunicação de dados, quanto o de execução da tarefa. Dessa forma pode-se obter uma estimativa de quão custoso é a execução de uma tarefa em um nó, e se há inserção de custos extras ao caminho crítico à execução da mesma em outro nó.

2.5.2 Heurísticas

Existem várias heurísticas que podem ser utilizadas para escalonamento de grafos em arquiteturas com memória distribuída, entre estas, podem ser citadas:

- BNP (*Bounded Number of Processors*): esta heurística escalona o grafo a um número limitado de processadores. É assumido que esses processadores são totalmente interconectados e não é levado em consideração possíveis contenções nos canais de comunicação.

- UNC (*Unbounded Number of Clusters*): esta heurística escalona o grafo a um número ilimitado de grupos de processadores. Nada é determinado sobre o escalonamento dentro do grupo, portanto é necessário que um escalonamento em nível de grupo seja feito.

- APN (*Arbitrary Processor Network*): esta heurística leva em consideração a topologia da rede de interconexão dos processadores presentes na arquitetura para realizar o escalonamento e o mapeamento das tarefas. Ele necessita que os custos de comunicação sobre os nós e os canais de comunicação sejam explicitados. Assim, ele é capaz de minimizar o tempo de execução através de critérios de proximidade entre processos que se comunicam entre si. Ele também tenta evitar problemas de contenção durante a troca de mensagens entre processadores.

- TDB (*Task Duplication Based*): esta heurística duplica processos com o objetivo de minimizar os custos de comunicação das sincronizações. Assim, ele cria uma duplicata do processo em todo nó que irá necessitar de comunicação com o mesmo.

Como o tempo de comunicação em sistemas de memória distribuída não é desprezível, convém, na hora do escalonamento, distribuir as tarefas de forma a minimizar o tempo

de comunicação entre processadores. Para isso pode-se distribuir ramos do grafo. Assim, tem-se que todos os dados para a execução daquele determinado ramo se encontrará no mesmo processador, eliminando comunicações desnecessárias. Caso o ramo crie outras sub-árvores, essas poderão ser distribuídas para outros processadores, caso estejam disponíveis. Essas heurísticas podem ser facilmente empregadas para aplicações cujos grafos sejam conhecidos *a priori*. Em aplicações dinâmicas regulares também é possível obter boas estratégias de escalonamento pelo conhecimento da estrutura do grafo gerado.

2.6 Algoritmo de Graham

Graham [32] prova que um algoritmo de listas é capaz de realizar o escalonamento de forma ótima. Esse algoritmo, porém, não é completo, pois não leva em consideração o tempo necessário para se comunicar os dados necessários entre as tarefas e que todas as tarefas de uma aplicação já se encontram prontas para executar.

Quando um problema é particionado em atividades concorrentes e sua execução é feita de forma paralela pode-se obter um ganho de desempenho. Entretanto, uma grande gama de aplicações define uma seqüência de tarefas que não podem ser paralelizadas, ou seja, possuem uma parte de seu código que descreve um fluxo de dados. Essa seqüência é, portanto, o limite de desempenho da aplicação, sendo denominada, neste trabalho, de caminho crítico. Porém, os ambientes de execução necessitam de estruturas de controle para garantir que a execução da aplicação seja feita de forma coerente, ou seja, respeitando a dependência de dados existentes entre as tarefas concorrentes. A manipulação dessas estruturas gera uma sobrecarga, também chamada na literatura de *overhead*, a qual degrada o desempenho durante a execução. O modelo de Graham não leva em consideração tais sobrecargas, por conseqüência ele é ineficaz para representar o desempenho de um determinado ambiente de execução de forma a refletir sua implementação.

O modelo de Graham é utilizado para obter os limites teóricos de desempenho para a computação paralela de uma aplicação. O limite, o qual nenhum ambiente de execução pode mudar, é a cadeia de tarefas as quais não podem ser processadas em paralelo devido à dependência de dados entre elas. Esse limite prova-se da seguinte forma [33]: uma aplicação paralela qualquer, executando em uma máquina paralela com mais de uma unidade de processamento, termina em um instante T_p quando a tarefa τ_n terminou de executar. Analisando o estado da máquina paralela no instante de tempo σ anterior a T_p, duas situações podem acontecer: há unidades de processamento ociosas disponíveis para iniciar a execução de τ_n ou não há unidades de processamento ociosas. Caso não haja nenhuma unidade de processamento disponível para começar a executar τ_n neste instante, então, nada se pode concluir. No entanto, se no momento houver ao menos um processador disponível, então há alguma condição que impede que τ_n seja executada antes do instante t_σ. Continuando a análise, considerando o caso em que havia pelo menos um

processador ocioso no instante t_σ, então tem que existir uma tarefa τ_{n-1} que termina no instante de tempo t_σ e essa tarefa produz dados que são necessários para a execução de τ_n. Portanto, obtém-se uma relação de dependência entre essas tarefas, representada por $\tau_{n-1} \prec \tau_n$. A análise pode ser feita recursivamente até o instante inicial da aplicação, obtendo assim $\tau_1 \prec \tau_2 \prec \tau_3 \ldots \prec \tau_{n-2} \prec \tau_{n-1} \prec \tau_n$. A cadeia de dependências mostra as tarefas da aplicação que não podem ser paralelizadas. A relação de dependência entre as tarefas fica bem clara quando posta em forma de grafo, pois assim vê-se o seu caminho crítico.

Dessa forma, definindo T_1 como o tempo necessário para um algoritmo ser executado de forma seqüencial, T_p para o tempo necessário para uma máquina com p processadores e T_∞ para uma máquina com infinitos processadores, tem-se a Equação 2.1 que dá o limite do tempo máximo levado pelo algoritmo para ser executado.

$$T_p \leq \frac{T_1}{p} + \left(1 - \frac{1}{p}\right) T_\infty \tag{2.1}$$

No caso de infinitos processadores, o limite de tempo mínimo é somente o tempo necessário para executar o caminho crítico, pois qualquer tarefa fora do caminho crítico é imediatamente executada por um dos infinitos processadores. Na Equação 2.2 tem-se o somatório de todos os tempos das tarefas pertencentes ao caminho crítico, que por definição e por conveniência chama-se de T_∞. Voltando à Figura 2.2, pode-se ver T_1 como sendo o somatório da execução de todos os vértices presentes e T_∞ como o caminho crítico que é formado pelos vértices 1, 2, 5, 6, 7, 9 e 10.

$$T_\infty = \sum_{i=1}^{n} |T_i| \tag{2.2}$$

Em [33] tem-se um trabalho que comprova que não é possível melhorar esse limite, além de mostrar que um algoritmo de listas não depende das tarefas que ele controla. Porém, tanto em Graham quanto em Shmoys [33] os custos de comunicação entre processadores e nós não são levados em consideração. Isso os torna incompletos no que tange os modelos de algoritmos de escalonamento, já que os custos não são desprezíveis.

2.7 Algoritmos para escalonamento

Na literatura encontram-se vários algoritmos de escalonamento que consideram grafos como entrada, cada qual tendo seus pontos fracos e fortes. Dentre esses algoritmos, encontram-se os seguintes:

- *Earliest Task First* [34]: é uma estratégia estática que consiste em pegar a primeira tarefa que estiver pronta para executar e alocar para o primeiro processador disponível. Se tratando de grafos, essa estratégia sempre pega a tarefa pronta do nível

mais baixo;

- *Least Loaded*: esta estratégia pode ser utilizada em sistemas de escalonamento dinâmico e consiste em alocar a tarefa que está sendo criada e/ou está apta a ser executada no momento ao processador que estiver com a menor carga computacional. Há variações desse algoritmo que levam em consideração o custo de comunicação entre os nós envolvidos na migração da tarefa;

- *Work Stealing* [35]: esta estratégia consiste em um processador que está oscioso roubar trabalho de outro processador que tenha trabalho esperando para ser executado. O roubo ocorre no nível mais baixo da lista de tarefas prontas a serem executadas, ou seja, a tarefa mais antiga presente nessa lista. Isso é possível devido ao fato de que cada tarefa presente na lista, além de seus dados tradicionais, contém também o seu nível na árvore de dependências. Esse roubo se dá no nível mais próximo à raiz para minimizar o custo de comunicação, pois quanto mais próximo a raiz da árvore se encontra, maior a possibilidade dessa tarefa criar outras, gerando assim mais trabalho, e diminuindo a necessidade de outros roubos de trabalho e os conseqüentes custos de comunicação.

- *Dominant Sequence Clustering* (DSC) [13]: este algoritmo analisa o *DAG* de forma direta e de forma invertida e escolhe o menor escalonamento entre os dois como resultado. A análise, de forma invertida, é obtida invertendo-se todos os arcos presentes no grafo, realizando o algoritmo do DSC e, então, esse resultado tem seus arcos invertidos novamente. A análise de forma invertida é feita, pois nem sempre a direta obtém um resultado ótimo quando a invertida pode obter [13]. O algoritmo leva em consideração que o custo de comunicação de tarefas executadas pelo mesmo processador é zero. O primeiro passo que toma é identificar o caminho crítico, chamado por ele de seqüência dominante (*Dominant Sequence*), e , assim, tentar reduzir essa seqüência. Isso se dá através do agrupamento (*clustering*) das tarefas pertencentes à seqüência dominante em um mesmo processador, fazendo assim que o custo de comunicação entre essas tarefas seja zero. Para isso, o custo de comunicação das tarefas que estão sendo agrupadas, e portanto zeradas, é comparado com o custo que será agregado para a comunicação com tarefas que estão fora desse agrupamento. Se o custo de comunicação for maior do que o valor que esta sendo minimizado, este agrupamento não é considerado válido, pois na verdade insere maiores custos do que realmente minimiza.

2.8 Sumário

Como visto, cada algoritmo de escalonamento funciona para um determinado tipo de sistema e/ou aplicações. Portanto, para obter melhor desempenho, o algoritmo de escalonamento deve saber como a aplicação se comporta de maneira a melhor mapear suas tarefas aos recursos disponíveis. Em ambientes com escalonamento em dois níveis, o sistema operacional se encarrega de alocar os recursos para o ambiente de execução, restando a este último apenas mapear as tarefas da aplicação aos recursos previamente alocados. Se o ambiente de execução fornece ao programador alguma forma de controlar o mapeamento das tarefas feito pelo ambiente de execução, então, o mesmo pode ser adaptado pelo programador de forma que ele atenda as necessidades específicas de cada aplicação. Como, por exemplo, se uma aplicação possui um grafo de dependências regular, o programador pode criar uma heurística específica para essa regularidade, fazendo assim com que o sistema possa distribuir a carga entre os nós do sistema de forma mais otimizada.

Analisando as ferramentas para programação paralela e distribuída, é possível notar que em algumas, como por exemplo Cilk, há uma preocupação em não inserir custos no processo de escalonamento. A abordagem adotada por cada ferramenta reflete as premissas adotadas no seu desenvolvimento. Particularmente nota-se que em Athapascan-1 o grafo de dependência é criado toda vez que um dado precisa ser acessado. Isso gera um custo grande devido a constante manipulação desse grafo, isto é, devido ao fato que Athapascan-1 obtém o grafo a partir do acesso aos dados. Em contrapartida, Cilk cria o mesmo grafo sob demanda, ou seja, apenas quando a inserção de tarefas no grafo é necessária para manter o controle de acesso aos dados pois o mesmo é obtido a partir do controle da execução da aplicação. Avaliando as diferenças entre as ferramentas, tem-se que Athapascan-1 é mais eficiente em ambientes distribuídos, pois possui um grafo de dependência mais detalhado que pode ser usado para minimizar o custo de comunicação entre os nós da arquitetura. Já Cilk é mais eficiente em ambientes SMP porque ele minimiza a sobrecarga de custos na manipulação do grafo.

Capítulo 3

Anahy

Para executar uma aplicação em arquiteturas paralelas é necessário utilizar alguma ferramenta de programação que explore tais ambientes. Anahy propõe recursos para tal, provendo uma API para descrição de programas concorrentes e um núcleo de escalonamento de tarefas. O ambiente de programação de Anahy é descrito neste capítulo. O ambiente de execução encontra-se em desenvolvimento e o objetivo desse trabalho é provê-lo com suporte à utilização em ambientes com memória distribuída.

O modelo de programação utilizado por Anahy [36] é baseado em operações do tipo *fork/join*, onde um *fork* necessariamente cria um novo fluxo de execução que eventualmente será sincronizado com outro através do *join*. O controle de execução de Anahy utiliza-se do grafo de precedência de tarefas que é construído a partir das chamadas primitivas *fork/join*.

3.1 Arquitetura alvo

Anahy disponibiliza um ambiente para a exploração do processamento de alto desempenho sobre arquiteturas do tipo aglomerado de computadores, em que cada nó pode vir a ser um multiprocessador com memória compartilhada; no entanto, essa arquitetura é considerada apenas para a implementação do ambiente. O programador tem a visão de uma arquitetura virtual multiprocessada dotada de memória compartilhada, cujas visões são ilustradas na Figura 3.1.

Como destaca a Figura 3.1, a arquitetura real é composta por um conjunto de nós de processamento, dotados de memória local e de unidades de processamento (CPUs). A arquitetura virtual é composta por um conjunto de processadores virtuais (PVs) alocados sobre os nós e por uma memória compartilhada pelos PVs cujo número e o tamanho da memória compartilhada são limitados em função da capacidade dos recursos da arquitetura real. No entanto, a capacidade de processamento e de armazenamento virtuais não alteram o modelo.

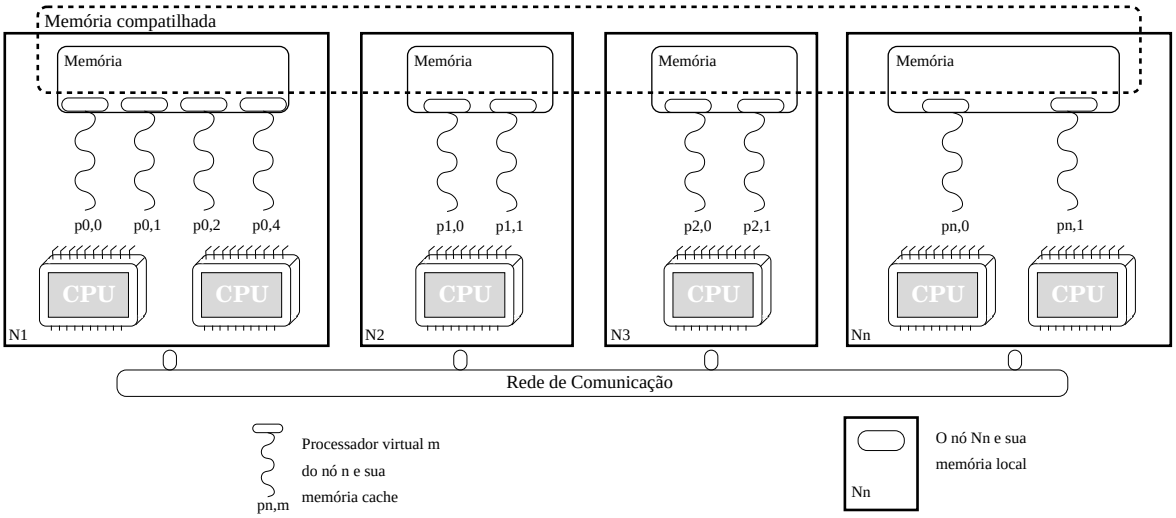

FIGURA 3.1 – Modelo da arquitetura de Anahy

Cada um dos PVs possui a capacidade de executar seqüencialmente as tarefas que a ele forem submetidas: enquanto um PV estiver executando uma atividade, nenhuma outra sinalização será tratada por ele. Quando ocioso, ou seja, não executando nenhuma tarefa do usuário, o PV pode ser despertado ao existir uma nova atividade apta a ser executada. Além das instruções convencionais (aritméticas, lógicas, de controle de fluxo, etc.), e essa arquitetura conta com duas novas instruções para descrição da concorrência da aplicação, permitindo a criação de uma nova atividade e a sincronização de uma atividade com o final de outra, e instruções de alocação, de deleção, de leitura e de escrita na memória compartilhada. Nenhum sinal é previsto para ser enviado entre PVs, estejam esses no mesmo nó ou não. Cada PV conta ainda com um espaço de memória próprio, utilizado para armazenar dados locais às atividades do usuário que serão executadas.

A comunicação entre os PVs se dá através da memória compartilhada, acessada pelas instruções introduzidas pela arquitetura virtual, a qual não suporta nenhum mecanismo de sincronização: todo controle ao acesso aos dados compartilhados deve ser feito através das instruções de controle de concorrência.

3.2 Comunicação e sincronização entre tarefas

Assim como nos programas seqüenciais, programas concorrentes produzem resultados através de transformações de dados recebidos em entrada. No entanto, a programação concorrente (paralela ou distribuída) implica a divisão do trabalho total da aplicação em atividades concorrentes, as quais são denominadas *tarefas*, neste trabalho. Inevitavelmente essas tarefas necessitam trocar dados entre si de forma a fazer com que o programa evolua.

Desse modo, as interfaces para programação concorrente introduzem mecanismos de

comunicação de dados e de sincronização às tarefas [37]. Os mecanismos de comunicação permitem que dados produzidos por alguma tarefa sejam, de alguma forma, colocados à disposição de uma outra. Os mecanismos de sincronização permitem a uma tarefa informar a outra que um dado encontra-se disponível ou verificar a disponibilidade de um determinado dado. Com os mecanismos de sincronização é possível controlar o avanço da execução do programa, não permitindo que tarefas sejam executadas antes que seus dados de entrada estejam disponíveis.

É importante observar que, no contexto de Anahy, a função da sincronização é de conciliar as datas de execução das tarefas em relação à produção/consumo de dados. Muitas ferramentas de programação, no entanto, oferecem mecanismos de sincronização que não garantem uma ordem na execução das atividades, garantindo apenas que uma atividade tenha conhecimento do estado de uma outra; um exemplo clássico é o uso de mutex para controle de execução de seções críticas. O uso desse recurso de sincronização, embora fundamental para diversas aplicações, introduz um nível de indeterminismo na execução que não permite que seja garantido um determinado resultado para todas as execuções de um programa, considerando um determinado conjunto de dados de entrada. Como esse tipo de sincronização não permite controle da comunicação de resultados de tarefas, ela não está sendo considerada.

Anahy utiliza-se do grafo de dependência de tarefas para controlar a execução. Dessa forma, aquelas que possuem os dados de entrada já disponíveis na memória compartilhada são executadas. Ao terminar, elas produzem resultados que poderão viabilizar a execução de outras tarefas.

3.3 Interface de programação

Um dos maiores problemas ligados ao desenvolvimento de programas concorrentes advém do alto grau de liberdade de ação que o programador passa a ter: decomposição da sua aplicação em atividades concorrentes e escalonamento dessas atividades sobre as unidades de cálculo da arquitetura, entre outros. Anahy automatiza o processo de escalonamento, facilitando o processo de desenvolvimento de aplicações. No entanto, restringe o número de primitivas para descrição de concorrência e sincronização ao necessário para criação de um grafo de dependência.

Nesta seção são apresentados os recursos de programação oferecidos por Anahy para decomposição de um programa concorrente. Também é apresentado como o ambiente cria, a partir dos serviços, a representação do programa em termos de grafo de dependências.

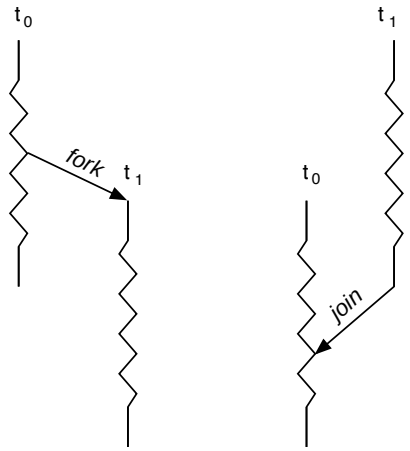

FIGURA 3.2 – Exemplo das operações *fork* e *join*

3.3.1 Serviços oferecidos

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. Dessa maneira, permite sincronizar as tarefas concorrentes da aplicação realizando troca de dados implícitas entre elas. Esses serviços podem ser representados através das operações *fork/join*, disponibilizando ao programador uma interface de programação bastante próxima ao modelo oferecido pela multiprogramação baseada em processos leves (no que diz respeito à criação e à sincronização com o término de fluxos de execução). Essa abstração permite a descrição de atividades sem que o programador identifique explicitamente quais dessas atividades são concorrentes na sua aplicação. Na Figura 3.2 é mostrado um exemplo de como as operações *fork/join* funcionam em Anahy.

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 a sua execução. O programador não possui nenhuma hipótese sobre o momento em que este fluxo será disparado, todavia, sabe-se que após seu término, um resultado será produzido e armazenado na memória compartilhada.

A sincronização com o término da execução de um fluxo é realizada através da operação *join*, identificando o fluxo a ser sincronizado. Essa operação permite que um fluxo bloqueie, aguardando o término de outro fluxo, 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

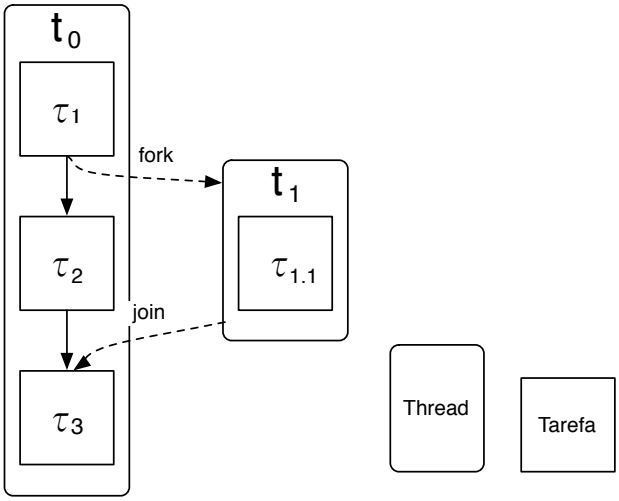

FIGURA 3.3 – Exemplo de sincronização entre tarefas usando *fork* e *join*

forma concorrente. Essas tarefas são definidas implicitamente:

- no momento do *fork*: o novo fluxo de execução inicia executando uma nova tarefa que possui como dados de entrada os argumentos da própria função;
- no momento de um *join*: o fluxo de execução termina a execução de uma tarefa e cria uma nova a partir da instrução que sucede (na ordem lexicográfica) o operador *join*. Essa nova tarefa tem como dados de entrada a memória local do fluxo de execução (atualizada até o momento que precedeu a realização do *join*) e os resultados retornados pela função executada pelo fluxo sincronizado; e,
- no fim da função executada por um fluxo de execução.

Observe que o acesso à memória compartilhada é realizado implicitamente pelos operadores *fork* e *join*. Na Figura 3.3 é visto como as operações de sincronização delimitam as tarefas concorrentes.

Outro aspecto a observar é a capacidade de execução seqüencial do programa quando eliminados os operadores de sincronização. Em outras palavras: a execução concorrente da aplicação produz o mesmo resultado que ofereceria a execução seqüencial do mesmo programa, o que facilita o desenvolvimento do programa e sua depuração.

3.4 Sintaxe de Anahy

Anahy está sendo desenvolvido de forma a permitir compatibilidade com o padrão POSIX para *threads* (IEEE P1003.c). Dessa forma, as primitivas e estruturas ofereci-

das são um subconjunto dos serviços oferecidos por esse padrão. Os atuais esforços de implementação estão concentrados em uma interface de serviços para programas C/C++.

O corpo de um fluxo de execução é definido como uma função C convencional, como representado na Figura 3.4.

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

FIGURA 3.4 – Exemplo de código para um fluxo de execução em Anahy

Nesse exemplo, a função `func` pode ser instanciada em um fluxo de execução próprio. O argumento `in` corresponde ao endereço de memória (na memória compartilhada) onde se encontram os dados de entrada da função. A operação de retorno (`return out`) foi colocada apenas para explicitar que, ao término da execução de `func`, o endereço de um dado, na memória compartilhada, deve ser retornado pela função. Esse endereço contém o resultado produzido pela tarefa. As sintaxes das operações *fork* e *join* correspondem as operações de criação e de espera por término de *thread* em POSIX: `pthread_create` e `pthread_join`. As sintaxes destes operadores são apresentadas na Figura 3.5.

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

FIGURA 3.5 – Sintaxe para os operadores *fork*/*join* em Anahy

Nessa sintaxe, `athread_create` cria um novo fluxo de execução para a função `func`; a entrada dessa função está presente no endereço de memória `in`. O novo fluxo criado poderá ser referenciado posteriormente através do valor `th`, o qual consiste em um identificador único. Os valores fornecidos por `atrib` definem atributos com quais o programador informa características do novo fluxo de execução no que diz respeito a sua execução (por exemplo, necessidades de memória). Na operação `athread_join` é identificado o fluxo com o qual se quer realizar a sincronização e `res` identifica um endereço de memória (compartilhada) para os dados de retorno do fluxo. Ambos operadores retornam zero em caso de sucesso ou um código de erro.

```
void* T_0( void * in ) {
    Tarefa τ1
    athread_create( t_1, thread_attr, T_1, void * in );
    Tarefa τ2
    athread_join( t_1, resultado)
    Tarefa τ3
    return out;
}
```

FIGURA 3.6 – Exemplo de programa destacando as tarefas

Na Figura 3.6 temos um programa de exemplo, o mesmo ilustrado na Figura 3.3. Nele encontram-se destacadas como as tarefas são delimitadas pelas primitivas *fork* e *join*.

3.5 Núcleo executivo

Anahy prevê a execução de programas concorrentes tanto sobre aglomerados de computadores como sobre arquiteturas SMP. O ambiente garante transparência no acesso aos recursos de processamento da máquina. Como resultado, o uso de Anahy como ambiente de programação/execução permite que o programador codifique apenas sua aplicação, livrando-o de especificar o escalonamento das tarefas nos processadores (ou dos dados nos módulos de memória). O núcleo executivo foi igualmente concebido de forma a suportar a introdução de mecanismos de balanceamento de carga.

3.5.1 Algoritmo de escalonamento

O algoritmo de escalonamento pressupõe a arquitetura descrita na Seção 3.1 e tem as tarefas como unidade de manipulação. Uma tarefa é uma unidade de trabalho, definida pelo programa em execução, composta por uma seqüência de instruções capaz de ser executada em tempo finito: uma tarefa não possui nenhuma dependência externa (tal uma sincronização), nem pode entrar em nenhuma situação de errônea, tal um laço sem fim. Dentre as instruções executadas por uma tarefa, podem existir operações de criação de novas tarefas. Como visto na Seção 3.3.1, uma tarefa termina ao executar uma operação de sincronização com outra tarefa.

O algoritmo gerencia quatro listas de tarefas: a primeira contém as tarefas *prontas* (aptas a serem lançadas), a segunda, as tarefas *terminadas* cujos resultados ainda não foram solicitados (a operação de *join* sobre estas tarefas ainda não foi realizada). A terceira e a quarta lista contêm tarefas *bloqueadas* e *desbloquadas*, respectivamente.

Anahy utiliza-se do algoritmo de Graham, apresentado na Seção 2.6 para nortear seu escalonador. Dessa forma, tem-se as garantias de desempenho provadas por Graham, assim como também a existência do caminho crítico.

Considerando $t_{(\tau)}$ o tempo necessário para executar a tarefa τ tem-se que o algoritmo de Graham permite obter o tempo de término T_{τ_k} de uma tarefa τ_k definida em um grafo de precedência $\tau_1 \prec \tau_2 \prec \ldots \prec \tau_k$

$$T_{\tau_k} \geq t_{(\tau_k)} + \sum_{i=1}^{k-1} t_{(\tau_i)} \tag{3.1}$$

Ou seja, o tempo mínimo necessário para executar uma tarefa τ_k é o custo da execução de suas instruções elementares, mais o custo associado à criação das $k-1$ tarefas que a precedem neste caminho crítico. A data de término de uma tarefa τ_k, portanto, deve considerar o tempo de execução das $k-1$ tarefas que a precedem. Esses tempos colocam em evidência que uma tarefa não pode ser iniciada antes que todas as tarefas que produzam dados necessários à sua computação não estejam concluídas.

Considerando que a criação e o término de uma tarefa geram custos associados à manipulação do grafo e considerando que esses custos sejam idênticos e identificados por σ, temos para cada tarefa uma sobrecarga equivalente a 2σ, refletindo sua inserção no grafo e a alteração de seu estado para terminada. Então:

$$T'_{(\tau_k)} \geq t_{(\tau_k)} + \sum_{i=1}^{k-1} t_{(\tau_i)} + 2k\sigma \tag{3.2}$$

O caminho crítico norteia o modelo do escalonador de Anahy: todo custo adicional à execução de tarefas deve ser evitado e, durante toda execução do programa, ao menos um dos processadores deve estar ativo executando uma tarefa desse caminho.

3.5.2 Implementação

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 3.7 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. Assim temos que o tempo de término de uma tarefa τ_k em Anahy é dado por:

$$T''_{(\tau_k)} \geq t_{(\tau_k)} + \sum_{i=1}^{k} t_{(\tau_i)} + k\sigma \tag{3.3}$$

Essa otimização acontece, pois quando uma *thread* é criada, ela é posta em uma lista de *threads* prontas a serem executadas, correspondendo, então, a uma inserção no grafo de dependência de dados. No momento do *join*, a lista de *threads* prontas é percorrida, à procura da *thread* da qual pretende se obter os resultados. Por se tratar apenas de uma busca em uma lista, então não há custo associado à manipulação do grafo. Dessa forma

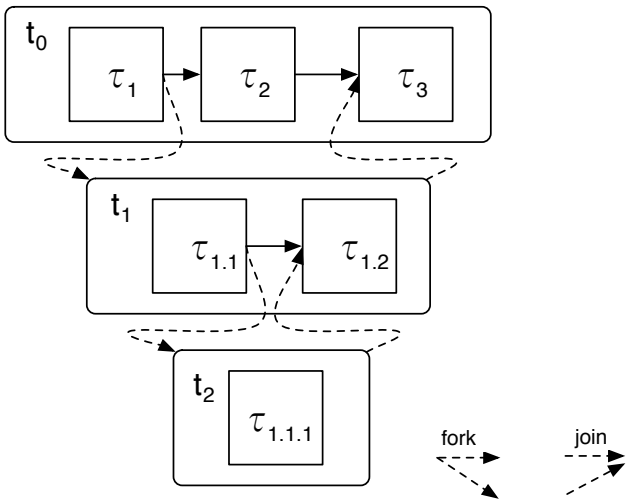

FIGURA 3.7 – Exemplo de relação tarefa × *thread*

a implementação do algoritmo evita a inserção de custos na execução das tarefas.

3.6 Escalonamento multinível

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 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 [32] 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, o escalonador 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 de maneira a 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 tomada sempre que uma nova tarefa é buscada, isto é, ela é calculada sempre que uma busca à lista de tarefas prontas é realizada.

3.7 Sumário

Para que Anahy possa ser utilizado em ambientes distribuídos, deve-se inserir um nível ao escalonamento. Este terceiro nível deve ser encarregado da distribuição da carga computacional gerada entre os nós que compõem a arquitetura utilizada. Nessa distribuição podem ser considerados diversos fatores, entre eles o custo computacional das tarefas e a localidade física dos dados. Essa última pode ser obtida através de uma análise da dependência dos dados de entrada e saída das tarefas do usuário.

Por fim, analisando o comportamento do escalonador e da lógica de controle de Anahy, é possível determinar que o funcionamento de Anahy pode ser decomposto em alguns serviços básicos como mostra a Tabela 3.1. Estes serviços devem ser também implementados no ambiente distribuído, de maneira a haver uma comunicação transparente entre um nó e outro da arquitetura.

Identificando os custos associados na relação *thread* × tarefa é visto que Anahy otimiza essa relação através de menor quantidade de manipulações no grafo em sua implementação, evitando assim, a inserção de custos durante a execução.

Serviço	Funcionalidade esperada
Requisição de trabalho	Utilizado pelo PV quando está ocioso. Procura na lista uma tarefa para execução, caso não haja nenhum na sua lista, rouba a tarefa mais próxima da raiz da lista de outro PV, escolhido de forma aleatória.
Criação de *thread*	Utilizado pelo sistema na criação de um novo fluxo de execução
Término de *thread*	Utilizado pelo sistema quando uma *thread* termina e tem seu resultado obtido para escrita em memória
Leitura de um dado na memória compartilhada	Utilizado pelo sistema para obter um dado de entrada que está contido na memória compartilhada.
Escrita de um dado na memória compartilhada	Utilizado pelo sistema para se escrever um dado de saída de uma tarefa na memória compartilhada

TABELA 3.1 – Serviços detectados em Anahy

Capítulo 4

Modelo de Escalonamento Distribuído

Neste capítulo apresenta-se o modelo da arquitetura de Anahy-DVM, um módulo para o ambiente Anahy. Este módulo permite que o ambiente seja capaz de executar aplicações em ambientes dotados de memória compartilhada distribuída, como por exemplo em aglomerados de computadores.

4.1 Arquitetura distribuída para Anahy

A Figura 4.1 apresenta a arquitetura modular do ambiente Anahy, e reproduz a Figura 1.1 para facilitar a leitura desta seção. Na figura é destacado que o mecanismo de comunicação implementado é baseado em Mensagens Ativas [6, 38]. Anahy já possui primitivas de comunicação entre nós que possibilita a distribuição de carga de uma aplicação. No entanto, um núcleo de escalonamento de uso geral não se encontra disponível.

Em [6] foi identificado que, a fim de ocorrer a correta comunicação entre nós Anahy, são necessários alguns serviços para realização do escalonamento distribuído e manutenção do grafo. Esses serviços foram detalhados no curso deste trabalho e encontram-se identificados na Seção 3.7. É de responsabilidade desses serviços realizar a requisição e o envio de tarefas, assim como também o envio e o recebimento de dados de entrada e resultados das tarefas. Para que não se altere a arquitetura do Anahy-SMP, foi incluso um novo elemento, denominado de *daemon* de comunicação (Figura 4.2), o qual é responsável por prover os serviços supracitados.

Deve-se enfatizar que na Figura 4.1 vê-se uma sub-divisão no processo de escalonamento em global e local. Essa sub-divisão corresponde à concorrência entre-nós e intra-nó, respectivamente. A concorrência intra-nó refere-se à exploração dos recursos computacionais inerentes a um nó e é realizada pela versão atual de Anahy-SMP. A concorrência entre-nós refere-se à exploração dos recursos computacionais de dois ou mais nós os quais

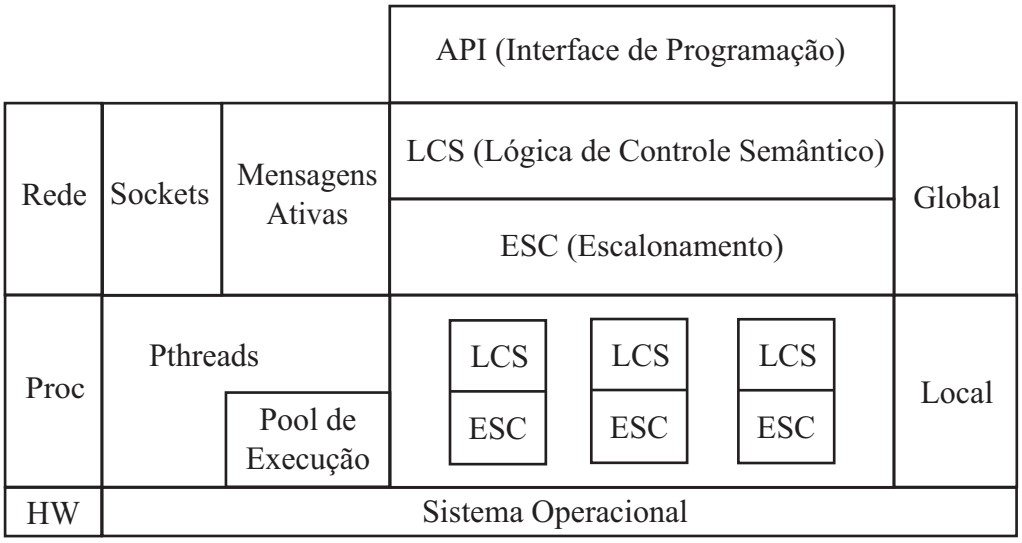

FIGURA 4.1 – Modelo em Camadas de Anahy

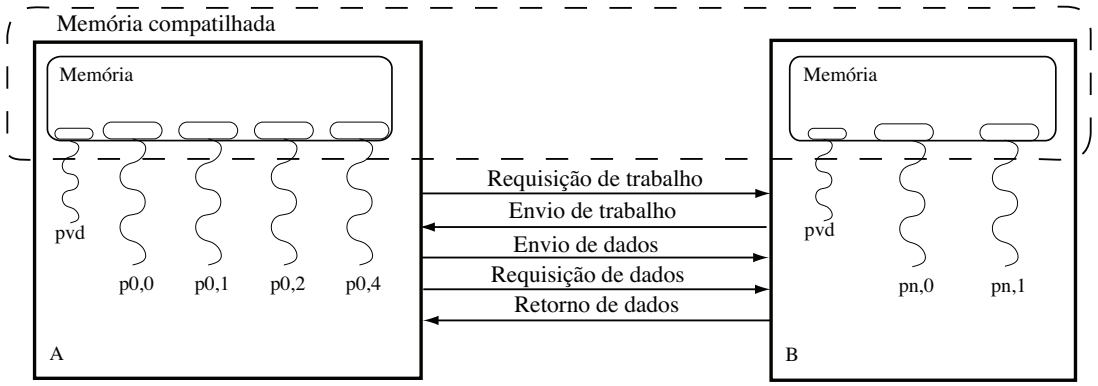

FIGURA 4.2 – Suporte a comunicação em Anahy[6]

estão interligados por algum tipo de rede. Para isso é necessário um conjunto de estruturas e primitivas responsáveis pelo controle da execução nesse ambiente. No caso do ambiente intra-nó, Anahy-SMP oferece um conjunto de funcionalidades para controle de lista de tarefas, criação, sincronização e escalonamento delas. Esse mesmo conjunto de funcionalidades deve estar presente no contexto distribuído para que o escalonamento e manutenção do grafo de tarefas em nível global seja possível.

4.2 Serviços de comunicação

Os serviços de comunicação devem prover o suporte ao balanceamento de carga, assim como para a migração de dados entre os nós da arquitetura. A Figura 4.2 identifica os serviços de suporte à operação de Anahy através das possíveis interações entre dois nós

- A e B - da arquitetura distribuída. Esses serviços são discutidos na seqüência:

- Requisição de trabalho: é chamado quando um nó não possui nenhum trabalho em sua lista local. Assim, para que o paralelismo da arquitetura seja aproveitado, este nó sinaliza a outro nó que está ocioso e pode receber tarefas. Na Figura 4.2 é representado o pedido de trabalho do nó A para o B.

- Envio de trabalho: é a resposta a uma requisição de trabalho. Quando um nó possui *threads* que podem ser migradas, este as envia para os nós que sinalizarem que estão ociosos. Consiste em uma mensagem que envia a descrição da *thread* a ser executada e os dados que esta tarefa manipula. Em caso negativo, ou seja, que nenhuma *thread* possa ser migrada, também é enviada uma mensagem ao nó requisitante, porém esta apenas contém um código significando que o nó não possui nenhuma *thread* que possa ser enviada. Na Figura 4.2 esta representado o envio de trabalho de B para A, em resposta ao pedido previamente feito.

- Requisição de dados: este serviço é chamado por um PV quando necessita de dados produzidos por outro PV, estando este presente no mesmo nó ou não. O serviço visa a dar a primitiva *join* transparência de localização. Assim o programador precisa apenas explicitar qual *thread* produziu os dados necessários, sem se preocupar com o local onde ocorreu a efetiva computação dessa *thread*. Na Figura 4.2 é representada esta requisição sendo feita pelo nó A para o nó B.

- Retorno de dados: este serviço é a resposta a alguma requisição. O nó que recebeu a requisição determina se a *thread* existe em seu conjunto local de dados, se os dados já se encontram disponíveis na memória e, em caso positivo, os envia ao nó requisitante. Em caso negativo o nó aguarda o término da execução da *thread* para, então, enviar os dados produzidos por ela. Na Figura 4.2 é representado pelo envio de dados de B para A.

- Envio de dados: este serviço tem por finalidade antecipar possíveis requisições de dados. Quando um nó termina a execução de uma *thread* que foi migrada, possivelmente o nó de origem dessa *thread* vai necessitar do dado em algum momento. Portanto o nó que a executou envia os resultados da computação ao nó de origem mesmo que este não os requisite. Isso ocorre para otimizar desempenho, pois quando necessários, os dados já estarão no nó e não será preciso esperar a sua comunicação. Na Figura 4.2 é representado pelo envio de dados de A para B.

Além dessas funções é necessário prover a arquitetura virtual de Anahy de primitivas que permitam a migração transparente de *threads* entre os nós. Por essa razão foi proposto em [6] a utilização de funções para empacotamento e desempacotamento de dados. Assim, quando uma *thread* necessita de receber algum dado, as funções são utilizadas. São quatro

as primitivas propostas: `packInFunc`, `unpackInFunc`, `packOutFunc` e `unpackOutFunc`. As primeiras são responsáveis por empacotar e desempacotar os dados de entrada e as 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 tem por objetivo associar custos as 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.

Ao haver uma migração de *thread*, pode ocorrer que o custo de comunicação dos dados seja superior ao custo de execução da tarefa, nesse caso, ocorre também a inserção de custos na execução da aplicação. Para que o ambiente seja capaz de detectar essas situações é necessário anotar no grafo de dependência de dados os custos associados à migração dos dados e o custo estimado de execução, porque o sistema não tem como saber *a priori* qual o custo de uma *thread*, ficando, então, de responsabilidade do programador estimar qual seria o custo de execução, assim como também determinar o custo de comunicação dos dados de uma *thread*, informações de vital importância para o escalonador. Portanto, também foi necessário estender as primitivas de criação de tarefas Anahy para que o programador possa associar esses custos às *threads*.

O escalonador utiliza informações para permitir ao ambiente criar um grafo anotado sobre o qual serão feitos 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 *Daemon* de comunicação

Foi criado em Anahy-DVM um PV dedicado ao processamento da comunicação entre os nós da arquitetura identificado por PV_d, visto esse elemento ser baseado no algoritmo das Mensagens Ativas, as quais quando chegam no nó de destino, executam um serviço associado à mensagem. O serviço é chamado utilizando os dados passados na mensagem como dados de entrada. Dessa forma, para que houvesse a comunicação transparente entre os nós e esta comunicação não acarretasse ao sistema uma sobrecarga de execução, foi escolhido um processador dedicado para executar esses serviços.

É função do *daemon* processar todos os serviços apresentados na Seção 4.2. Por conseguinte, no momento de sua instância na memória, todos os serviços de comunicação do sistema serão associados com ele.

Os PVs de Anahy, portanto, ficam dedicados ao processamento das tarefas da aplicação, obtendo-se uma sobreposição de cálculo efetivo com comunicação para fins de ganho de desempenho [23]. Entretanto, para que haja este ganho, os serviços executados pelo *daemon* têm de serem rápidos e não bloqueantes.

4.4 Funcionamento do escalonador

O mecanismo de escalonamento implementado em Anahy-SMP obedece ao modelo de execução proposto por Graham, cuja versão distribuída deve obedecer ao mesmo conjunto de regras. Portanto, o núcleo com suporte a ambientes distribuídos utilizará, em sua grande parte, o algoritmo de roubo de trabalho, descrito na Seção 2.7. Como o objetivo de Anahy é minimizar os custos associados à execução do caminho crítico, o mesmo comportamento deve ser adotado na sua operação distribuída, sendo, assim, 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 na Seção 4.2.

Existem estratégias de manipulação de dados distribuídos que podem explorar informações sobre tarefas e dados, ou seja, utilizam-se dos custos associados a essas tarefas e a quantidade de dados com que elas se comunicam. Tais estratégias então tiram proveito de grafos anotados. Portanto, para dotar Anahy-DVM da capacidade de utilizar tais estratégias, é possível, ao definir a tarefa concorrente, fornecer ao ambiente de execução informações relativas ao tempo de processamento esperado da *thread* e o custo de sua comunicação. Esse mecanismo pode ser automatizado, porém encontra-se fora do escopo deste trabalho. Utilizando-se dessas anotações, o núcleo executivo pode inferir quando uma tarefa pode migrar do nó que a gerou para outro, ou se essa migração acarretará em mais custos de comunicação do que se a mesma permanecesse no nó de origem. Na análise feita a seguir não se utiliza das anotações no grafo e se tem como premissa que a tarefa mais próxima a raiz potencialmente possui mais trabalho.

Inicialmente, quando a máquina virtual Anahy está sendo inicializada, apenas o nó que começou a executar o programa possui trabalho. Depois de criada a máquina virtual, o roubo de trabalho será iniciado sempre pelos nós ociosos. Um exemplo de máquina virtual Anahy pode ser visto na Figura 4.3.

Cada nó tem uma lista de todos os outros que compõe a máquina virtual Anahy e, ao acaso, escolhe outro nó 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

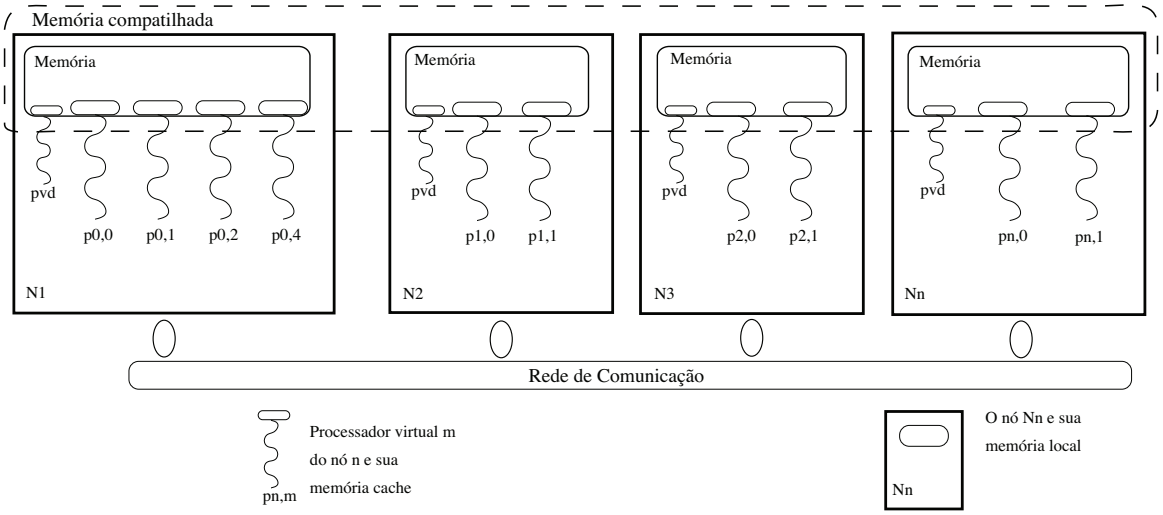

FIGURA 4.3 – Exemplo de máquina virtual Anahy

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 de Anahy, apresentado na Seção 3.5.2, evitando assim diferenças nas políticas de escalonamento global e local. Logo, a Equação 3.3 (página 31) é também válida para Anahy-DVM; entretanto, o valor da sobrecarga σ varia de acordo com o nível de escalonamento que se está tratando, ou seja, se é escalonamento global possui um valor, se é escalonamento local possui outro.

4.5 Desenvolvimento de estratégias

O algoritmo de escalonamento de Anahy-DVM é modular. O usuário pode escolher qual estratégia de escalonamento lhe é mais conveniente para a aplicação que está criando, sendo que, no momento, foram criadas duas formas de decidir qual *thread* escolher para execução. A primeira é realizada considerando a posição da *thread* no grafo de precedência. A segunda é feita considerando os custos de comunicação e de execução da tarefa. Mais estratégias poderão ser introduzidas no futuro.

4.6 Sumário

A operação distribuída de Anahy não deve divergir da sua operação em arquiteturas não distribuídas. Dessa forma, o escalonador global tem de seguir os mesmos princípios do

escalonador local. Também é necessário ao escalonador global ter o suporte à manutenção de um espaço de endereçamento compartilhado, que deve também prover primitivas que façam a manutenção do grafo de dependência de dados em ambientes com memória distribuída.

Concebido de forma modular, o escalonador distribuído de Anahy-DVM permite que a estratégia de escalonamento possa ser adequada à aplicação.

As primitivas para anotar o grafo permitem que o usuário adicione maiores informações que o escalonador pode utilizar de maneira a melhorar a tomada de decisão no momento de uma migração.

O uso do *daemon* permite ao ambiente realizar comunicação enquanto executa o cálculo efetivo da aplicação, obtendo assim a possibilidade de ganho de desempenho.

Capítulo 5

Implementação

A implementação de Anahy-DVM se deu em duas etapas. A primeira composta da adequação da ferramenta de Mensagens Ativas 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 Mensagens Ativas com a finalidade de permitir a esse mecanismo lidar com os tipo de dados utilizados no núcleo executivo de Anahy. Isso permitiu as Mensagens Ativas [38] manipularem tais estruturas de dados para fins de migração de tarefas e de dados entre nós da máquina virtual Anahy-DVM.

Após, foram criadas as funções de tratamento, assim chamadas quando uma Mensagem Ativa chega. Essas funções têm por finalidade efetivar os serviços mostrados na Figura 4.2. 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 Mensagens Ativas 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, necessitando 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, cujas primitivas necessárias à sua manipulação são citadas a seguir:

- `athread_msg_init( int tamanho)`: esta primitiva inicializa o pacote, alocando o espaço necessário na memória. Retorna o ponteiro para o pacote criado.
- `athread_pack( athread_msg_t *pacote, int deslocamento, void *buffer, int tamanho)`: esta função realiza a cópia da quantidade de dados indicada no *buffer* para dentro do pacote. É possível indicar o deslocamento dentro dele para fins de permitir o usuário escolher o local onde serão copiados os dados.
- `athread_unpack( athread_msg_t *pacote, int deslocamento, void *buffer, int tamanho)`: função responsável pelo acesso de leitura dentro do pacote. Com ela 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 teve de ser estendida para dar suporte aos serviços necessários à distribuição de tarefas e dados como proposto em [6]. 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

O ambiente de execução Anahy, assim como o padrão POSIX para *threads*, permite ao usuário utilizar quaisquer tipos de dados em suas aplicações através de ponteiros sem tipo definido. No entanto, isso faz com que o ambiente não saiba como tratar os dados utilizados, sendo que, apenas o usuário programador sabe como tratar os dados que a aplicação utiliza.

Isso se torna um problema para o núcleo executivo quando executando em um ambiente de memória distribuída compartilhada, já que o ambiente não sabe como tratar os dados apontados de maneira que possam ser migrados.

Para tornar a migração dos dados possível, escolheu-se o programador que fornece ao ambiente um conjunto de rotinas para tratar os dados de maneira a serem empacotadas e então migradas 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. Também foi necessário criar quatro primitivas para se poder atribuir os valores a essas extensões, denominadas de: `pack_in_fun`, `unpack_in_fun`, `pack_out_fun` e `unpack_out_fun`. Respectivamente, elas armazenam os ponteiros para as funções responsáveis pelo empacotamento dos dados de entrada, o desempacotamento dos dados de entrada, o empacotamento dos dados de saída e o desempacotamento dos dados de saída de uma determinada tarefa.

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, portanto, os atributos `execution_cost` e `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. Na Figura 5.1 pode-se ver o resultado final das extensões dos atributos.

```
struct athread_attr_t {
    unsigned char max_joins;
    char detach_state;
    char initialized;
    char force_remote;
    pfunc in_pack_func;
    pfunc in_unpack_func;
    pfunc out_pack_func;
    pfunc out_unpack_func;
    int execution_cost;
    int communication_cost;
    long input_len;
    long output_len;
}
```

FIGURA 5.1 – `athread_attr_t` após as extensões

5.3.2 Interface de programação

Para que o programador tenha acesso às extensões feitas no núcleo executivo, novas funções na API tiveram de ser implementadas. Elas podem ser utilizadas para permitir que a aplicação se utilize de Anahy-DVM para ser executada em ambientes de memória distribuída compartilhada. Pontualmente, as funções criadas foram:

- `athread_attr_pack_in_func( athread_attr_t *attr, void *func)`: esta primitiva insere na estrutura de atributos de *thread* apontada por `attr` o ponteiro para a função `func` passada. Esta função será utilizada pelo ambiente na necessidade de empacotar dados de entrada desta *thread*.

- `athread_attr_pack_out_func( athread_attr_t *attr, void *func)`: esta primitiva insere na estrutura de atributos de *thread* apontada por `attr` o ponteiro para a função `func` passada. Esta função será utilizada pelo ambiente na necessidade de empacotar dados de saída desta *thread*.

- `athread_attr_unpack_in_func( athread_attr_t *attr, void *func)`: esta primitiva insere na estrutura de atributos de *thread* apontada por `attr` o ponteiro para a função `func` passada. Esta função será utilizada pelo ambiente na necessidade de desempacotar dados de entrada desta *thread*.

- `athread_attr_unpack_out_func( athread_attr_t *attr, void *func)`: esta primitiva insere na estrutura de atributos de *thread* apontada por `attr` o ponteiro para a função `func` passada. Esta função será utilizada pelo ambiente na necessidade de desempacotar dados de saída desta *thread*.

- `athread_attr_set_communication_cost( athread_attr_t *attr, int custo)`: insere o custo estimado de comunicação da *thread*.

- `athread_attr_set_execution_cost( athread_attr_t *attr, int custo)`: insere o custo estimado de execução da *thread*.

5.3.3 Serviços

Para que as Mensagens Ativas 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. Aqui são descritos os serviços criados, assim como detalhes de sua implementação.

Os serviços são divididos em duas categorias: os que trabalham com o roubo de tarefas e os que organizam a sincronização de tarefas. Os serviços que são responsáveis pelo roubo de tarefas são as seguintes:

- *steal_job*: serviço responsável pela escolha de um nó para enviar uma mensagem de roubo de trabalho. Esta escolha do nó é realizada de maneira aleatória. Após o envio da mensagem de roubo, o serviço para em um semáforo esperando uma resposta do nó requisitado. Caso a resposta seja uma tarefa válida, esta é entregue ao escalonador através do serviço *deliver_job_service*. Caso contrário, a tarefa recebida é descartada e um novo roubo de trabalho é executado.

- *deliver_job_service*: este serviço pega uma tarefa válida recebida e a entrega ao escalonador. Isto é feito desempacotando o descritor da tarefa da Mensagem Ativa recebida. Após isso, é executada a função estabelecida pelo usuário para o desempacotamento dos dados de entrada da tarefa que são desempacotados por esta função e um ponteiro para eles é atualizado no descritor da tarefa. Por último, o descritor da tarefa é inserido na lista de tarefas prontas e o escalonador é sinalizado da chegada da tarefa.

- *steal_job_service*: este serviço é chamado quando chega uma mensagem de roubo de trabalho. Ele procura uma tarefa possível de ser migrada, ou seja, que possui os atributos das funções de usuário não nulos. Com a tarefa encontrada, o empacotamento do descritor dessa tarefa é realizado e, após isso, a função que empacota os dados de entrada da tarefa é chamada para que estes sejam inclusos no pacote a ser enviado ao nó requisitante.

Os serviços responsáveis pela sincronização de dados entre nós são os seguintes:

- *athread_join_remote*: serviço chamado quando uma tarefa requer um dado que é resultado de uma outra tarefa a qual foi executada remotamente. Envia uma

mensagem requisitando os dados de saída para o nó onde a tarefa foi efetivamente executada.

- *reply_join_service*: serviço responsável pela entrega dos dados após o término da computação. Procura a tarefa na lista de prontas, caso a encontre, empacota os dados de saída da tarefa, utilizando-se da função estabelecida pelo usuário, e as envia para o nó que requisitou esses dados, chamando a função *rcv_job_back_service* na sua chegada. Caso não encontre na lista de prontas, o algoritmo de escalonamento do Anahy-SMP é utilizado, como descrito na Seção 3.5.1.

- *rcv_job_back_service*: serviço o qual entrega à memória compartilhada os dados de saída recebidos de um nó remoto. Realiza o desempacotamento deles chamando a função do usuário e atualiza no descritor da tarefa o ponteiro para a área da memória compartilhada onde estes dados foram desempacotados.

5.4 Sumário

Nesse capítulo se detalhou a implementação de Anahy-DVM. Como foi modificado o mecanismo de Mensagens Ativas, assim como foram operacionalizadas as primitivas utilizadas no uso de memória compartilhada distribuída, também foi visto como o usuário deve criar suas funções que serão utilizadas na hora do empacotamento e desempacotamento dos dados a serem migrados entre nós. Por fim, descreveram-se os serviços criados para o tratamento das Mensagens Ativas e como estes tornam possível a operação de migração de tarefas.

Capítulo 6

Resultados Obtidos

Neste capítulo são apresentados os testes realizados para avaliação do escalonador distribuído. Os experimentos realizados para obter os resultados foram feitos utilizando-se uma aplicação sintética, criada para gerar uma grande quantidade de tarefas concorrentes assim como, distribuí-las entre os nós de um aglomerado de computadores. Os resultados obtidos tem por objetivo avaliar o funcionamento do escalonador distribuído frente a execução seqüencial e SMP do mesmo.

6.1 Aplicação sintética

O cálculo do número de Fibonacci, quando executado de forma recursiva gera um fluxo de execução como pode ser visto na Figura 6.1. Dessa forma, esta aplicação tem um potencial de paralelismo que pode ser explorado pelo ambiente de execução. Assim, quando programado para Anahy, o cálculo é realizado gerando uma *thread* para calcular cada nó do grafo. Portanto, fica a cargo do ambiente de execução o escalonamento das *threads*, assim como, a migração delas em caso de roubo de tarefas. Uma visão de como as *threads* interagem pode ser vista na Figura 6.2.

A aplicação de avaliação implementada foi o cálculo de Fibonacci com o acréscimo de uma carga arbitrária de dados a serem comunicados a cada experimento. A razão do acréscimo é possibilitar variar a quantidade de dados a serem comunicados em caso de migração para medir o impacto dessa comunicação na execução da aplicação. Os custos de execução da aplicação são dois. O primeiro é referente ao custo de execução da tarefa no processador virtual, o qual pode ser visto na Figura 6.1 como sendo o nó do grafo. O segundo custo é o de comunicação entre as tarefas da aplicação, que também pode ser visto na Figura 6.1 como sendo a aresta que conecta dois nós. Para avaliar o impacto da comunicação no tempo de execução da aplicação, o peso da aresta foi variado, sendo também variado o número de tarefas totais da aplicação. Para isto, basta aumentar o número de Fibonacci que se quer calcular. No Apêndice A, o código fonte da aplicação é

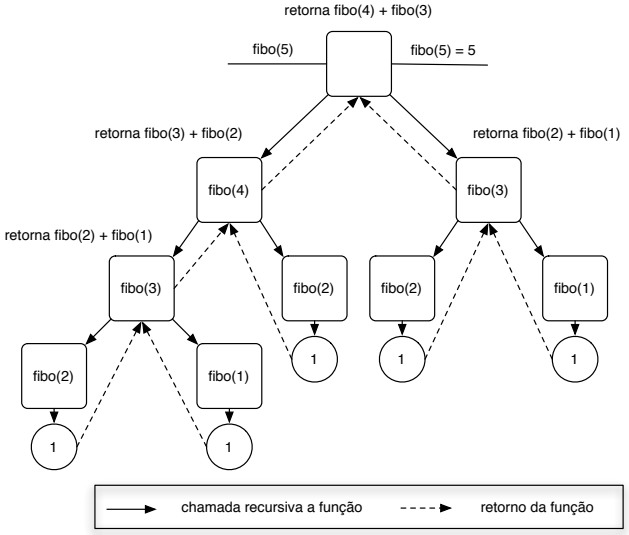

FIGURA 6.1 – Fluxo de execução recursiva de Fibonacci.

mostrado para exemplificar o código completo de uma aplicação de Anahy.

6.2 Experimentação

Os experimentos foram realizados em um aglomerado de computadores, considerando dois casos. O primeiro tem por objetivo testar o comportamento do escalonador quando a aplicação sintética não possui nenhuma carga extra por experimento. Dessa forma, testa-se se o comportamento é consistente com a versão SMP. No segundo caso, uma carga extra, que tem por objetivo testar o impacto da comunicação no tempo de execução da aplicação, é adicionada à comunicação das tarefas. Em outras palavras, testa-se a influência do *daemon* de comunicação no tempo de execução da aplicação.

6.2.1 Arquitetura utilizada no experimento

Os experimentos foram conduzidos em um aglomerado de computadores composto por oito nós. Cada nó é bi-processado e é composto de dois processadores Xeon de 2.8 Ghz. Cada nó possui 2 GB de memória RAM, um disco rígido de 80 GB de capacidade de armazenamento e rede gigabit ethernet. O sistema operacional utilizado foi o Linux cuja distribuição chama-se Gentoo. O kernel instalado nos nós é a versão 2.6.8-smp.

6.2.2 Metodologia aplicada

Os experimentos foram repetidos vinte vezes para obtenção de uma média e um desvio padrão. Durante cada experimento, nenhuma outra aplicação de usuário estava

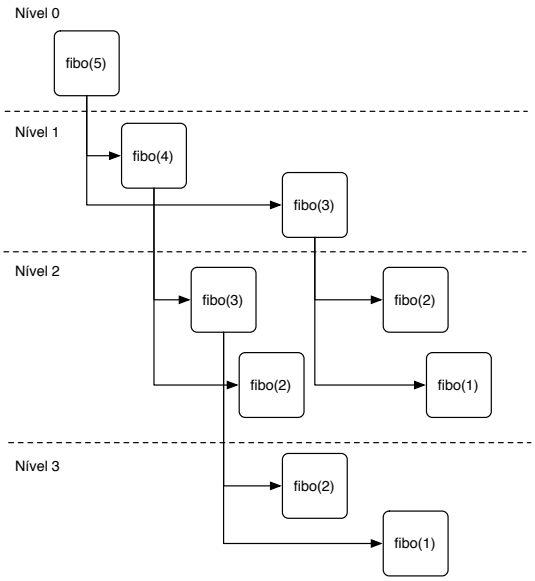

FIGURA 6.2 – Relação entre as *threads* na execução recursiva de Fibonacci.

executando nos nós envolvidos. Os tempos foram coletados depois que o núcleo executivo de Anahy foi carregado na memória, até o momento em que ele retorna o valor final da computação da aplicação, com o propósito de remover o tempo de carga da biblioteca na memória dos resultados coletados.

6.3 Desempenho coletado

Nesta seção apresentam-se os resultados coletados com a experimentação descrita na Seção 6.2. Os números de Fibonacci escolhidos para os experimentos foram de 10, 15 e 20, pois estes números representam uma quantidade pequena, média e grande de tarefas geradas pela aplicação de teste descrita na Seção 6.1.

6.3.1 Caso 1

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. Obtêm-se assim os resultados mostrados nas tabelas 6.1 e 6.2, com núcleo de execução configurado, respectivamente, com 1 e 2 PVs. Na Tabela 6.1 pode-se ver 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 6.3 complementa a avaliação de desempenho do Caso 1 apresentando o *speed up* obtido pelas execuções paralelas. Por questões de espaço, é apresentado apenas o *speed up* obtido para o cálculo de Fibonacci de 20. Neste gráfico, foi considerado referência para o cálculo o tempo T_1, ou seja, o tempo obtido para a execução da aplicação em 1 nó com 1 PV. Neste gráfico também observa-se que o núcleo de escalonamento reproduz o comportamento da execução variando o suporte de concorrência representado pelos PVs.

TABELA 6.1 – Resultados obtidos no caso 1 com 1 PV.

Nós	Número Fibonacci	Média (s)	Desvio padrão
1	10	2,67	0,003
1	15	30,08	0,020
1	20	541,25	0,187
2	10	1,75	0,002
2	15	19,52	0,014
2	20	292,86	0,116
4	10	0,94	0,001
4	15	10,55	0,008
4	20	158,15	0,066
8	10	0,50	0,001
8	15	5,54	0,005
8	20	83,03	0,038

TABELA 6.2 – Resultados obtidos no caso 1 com 2 PVs.

Nós	Número Fibonacci	Média (s)	Desvio padrão
1	10	1,65	0,026
1	15	18,59	0,032
1	20	334,60	0,087
2	10	0,95	0,016
2	15	10,69	0,020
2	20	171,02	0,048
4	10	0,55	0,010
4	15	6,20	0,012
4	20	99,19	0,029
8	10	0,29	0,006
8	15	3,29	0,007
8	20	52,67	0,016

Na Figura 6.3 também pode-se ver 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. Esse mecanismo, por escolher o nó de quem se vai roubar trabalho de forma aleatória, permite, então, que no início da computação os nós sem

trabalho fiquem tentando 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 máquina virtual, esse problema se potencializa, explicando assim os ganhos atingidos.

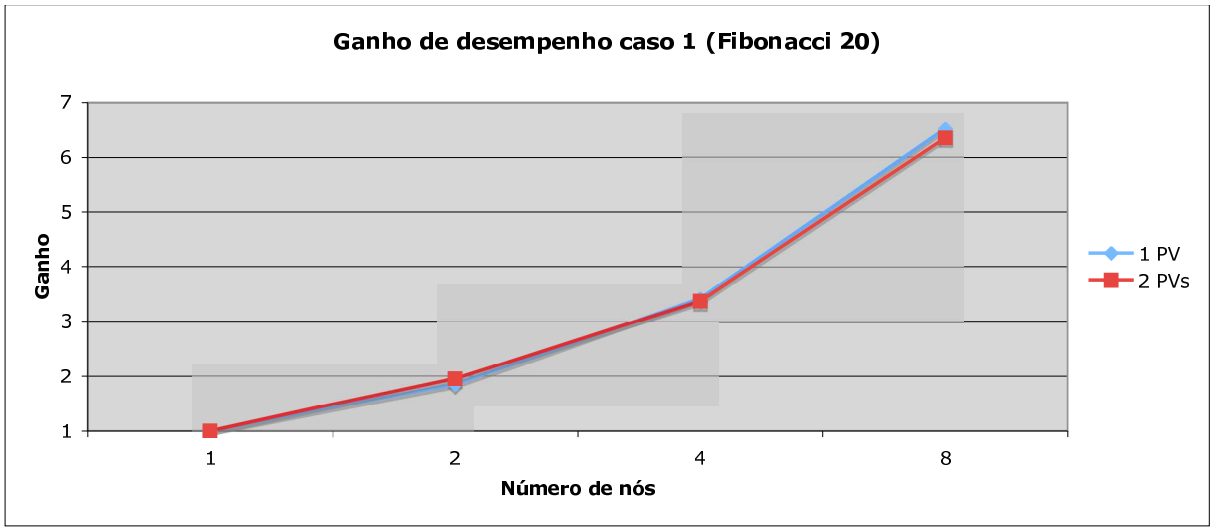

FIGURA 6.3 – Ganhos de desempenho obtidos no caso 1.

6.3.2 Caso 2

No segundo caso, varia-se uma carga de comunicação extra a ser comunicada a cada tarefa. O Caso 1 corresponde ao experimento com custo de comunicação mínimo (4 bytes). No experimento do Caso 2, foi avaliado o comportamento de execução com duas cargas sintéticas, com 512 e 4096 bytes. Assim obtiveram-se os resultados mostrados pelas tabelas 6.3 e 6.4, respectivamente com 1 PV e 2 PVs. Pode-se notar que não houve impacto significativo no tempo de execução da aplicação, devido à sobreposição de comunicação com cálculo efetivo, devido ao *daemon* de comunicação ser na verdade um PV dedicado. No entanto, destaca-se que o comportamento da execução mantém-se estável, independente do número de tarefas criadas.

Nas figuras 6.4 e 6.5 mostram-se graficamente os dados contidos nas tabelas 6.3 e 6.4. Nelas, podemos 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. Pode-se notar que quando há poucos nós na arquitetura virtual, o peso da comunicação pode causar um aumento do tempo de execução da aplicação. Isto é devido ao fato que há poucos PVs na arquitetura e quando um ou mais PVs bloqueiam esperando a sincronização dos dados, isto causa 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

TABELA 6.3 – Resultados obtidos no caso 2 com 1 PV.

Nós	Peso (Bytes)	Média (s)	Desvio padrão
2	4	292,86	0,116
2	512	299,01	0,116
2	4096	309,85	0,115
4	4	158,15	0,066
4	512	162,57	0,066
4	4096	168,27	0,066
8	4	83,03	0,038
8	512	85,68	0,038
8	4096	88,84	0,038

TABELA 6.4 – Resultados obtidos no caso 2 com 2 PVs.

Nós	Peso (bytes)	Média (s)	Desvio padrão
2	4	171,02	0,048
2	512	174,61	0,049
2	4096	180,94	0,048
4	4	99,19	0,029
4	512	101,97	0,029
4	4096	105,54	0,030
8	4	52,67	0,016
8	512	54,36	0,017
8	4096	56,36	0,017

a aplicação, cairá e, portanto, o impacto da comunicação será menor nesse caso. Cabe ressaltar que as curvas presentes nas figuras 6.4 e 6.5 possuem o mesmo comportamento, mostrando assim um impacto homogêneo do *daemon* de comunicação na execução da aplicação.

6.4 Análise Geral

Os resultados de desempenho coletados puderam mostrar que o núcleo executivo implementado funciona dentro das restrições impostas pelas premissas. Também foi mostrado que o núcleo provê meios para manter um comportamento de execução estável mesmo que os custos de comunicação sejam alterados. Os dados coletados também mostraram como as informações de tempo podem ser utilizadas para a análise de sobrecarga de escalonamento. Por fim, os dados mostraram 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 feitas com esses resultados.

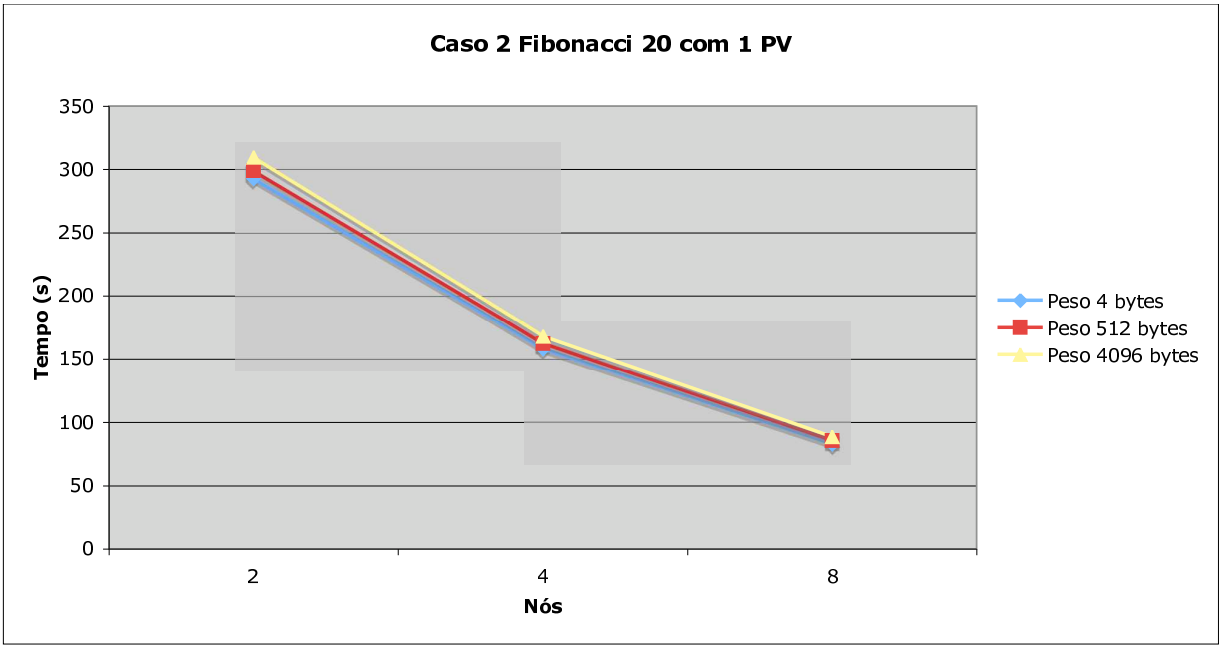

FIGURA 6.4 – Resultados obtidos no caso 2 com 1 PV.

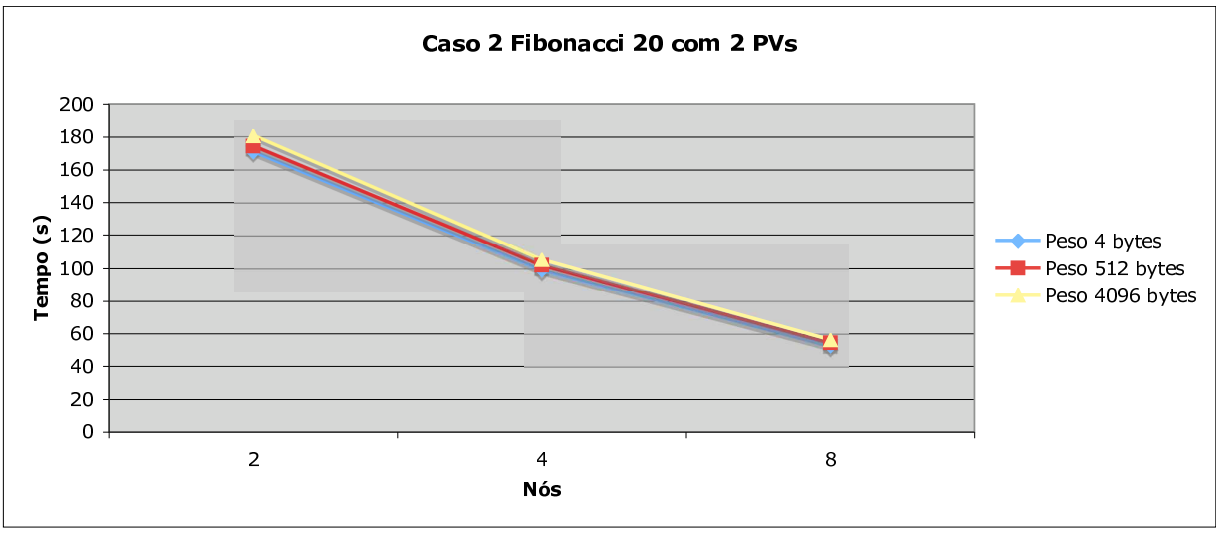

FIGURA 6.5 – Resultados obtidos no caso 2 com 2 PVs.

Capítulo 7

Conclusões

Com a evolução do processamento de alto desempenho e o advento dos aglomerados de computadores, surgiu a necessidade de métodos de exploração eficiente dos ambientes. Entretanto, essa exploração eficiente provou-se ser não trivial, pois surgiram vários ambientes de programação, dotados ou não de mecanismos de escalonamento, os quais têm por objetivo auxiliar o programador a obter desempenho.

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 arquitetura utilizada deve manter em execução simultânea.

Para transpor essas dificuldades, foram desenvolvidas ferramentas as quais tiram do programador a responsabilidade de realizar o mapeamento da concorrência da aplicação para o paralelismo real da arquitetura: uma dessas ferramentas é o Anahy. Entretanto, como ainda é uma ferramenta em desenvolvimento, Anahy não era dotado de um escalonador para ambientes de memória distribuída compartilhada, tais como aglomerados de computadores, sendo, por isso, esse escalonador desenvolvido e implementado de maneira a funcionar de forma idêntica ao escalonador SMP já existente em Anahy.

Para este fim, foi necessário estender o núcleo executivo de Anahy para suportar ambas as arquiteturas. Foram criadas e implementadas novas chamadas de API que permitem ao programador desenvolver aplicações para ambientes de memória distribuída compartilhada, sem tornar essa aplicação incompatível com a execução em ambientes SMP. Em outras palavras, uma aplicação desenvolvida para aglomerados de computadores rodando em Anahy pode ser executada em uma arquitetura SMP sem qualquer modificação do código fonte dessa aplicação.

Foi necessário, portanto, criar e implementar um escalonador para ambientes distribuídos o qual utilizasse um algoritmo de listas em seu núcleo. Dessa forma, a compatibi-

lidade com o algoritmo utilizado na versão SMP de Anahy fica garantida. O escalonador foi implementado seguindo o modelo proposto por Graham, mas não levando em consideração as sobrecargas associadas à manipulação das estruturas necessárias para realizar o escalonamento. O escalonador implementado, entretanto, leva em consideração que os custos existem e que podem ser utilizados para melhor realizar o escalonamento. Por essa razão, foi implementado em Anahy a habilidade de anotar no grafo custos de execução e de comunicação. Porém, uma estratégia de escalonamento que se utilizasse dessas anotações no grafo não foi implementada, ficando como trabalho futuro.

Também foi necessário criar um mecanismo de comunicação entre os nós presentes no aglomerado de computadores, para que eles possam realizar todas as operações presentes em Anahy SMP. Assim, foi implementado um *daemon* de comunicação o qual é responsável por toda troca de mensagens realizada entre os nós, assim como pela migração de tarefas e dados entre eles. O *daemon* foi implementado de maneira a ser um PV dedicado apenas à comunicação, de forma a não modificar o comportamento de Anahy quando rodando sobre aglomerado de computadores.

No decorrer desse trabalho foram publicados alguns artigos em eventos como o Workshop em Sistemas Computacionais de Alto Desempenho (WSCAD) [38], VecPar (*International Meeting on High Performance Computing for Computational Science*) [39], Workshop em Sistemas Computacionais e de Comunicação (WPerformance) [40].

Por fim, alguns trabalhos futuros a serem tomados são: a remoção de todos os *bugs* presentes na solução de maneira a tornar Anahy uma ferramenta operacional, tomada de medidas de desempenho mais completas, para fins de testar o uso da solução com aplicações cujo grafo é irregular, realizar análise das sobrecargas presentes no escalonador distribuído para fins de otimização e desenvolvimento de novas estratégias de escalonamento que levem em consideração os custos anotados no grafo.

Apêndice A

Código Fonte da Aplicação Sintética

```
/*
 * carga.c
 * ------
 *
 * Copyright (C) 2006 by the Anahy Project
 *
 * Anahy is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as published
 * by the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * Anahy is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public
 * License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with Anahy; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "athread.h"

struct data {
   int n;
```

```
   char *dados;
   int carga;
};

void *fibo(void *);
void *packfun(void *);
void *unpackfun(void *);

int main (int argc, char **argv) {
   athread_t thread;
   athread_attr_t attr;
   char *string, seed;
   struct data *entrada;
   int ret, n, load, size, interator;
   void *get;

   /* Inicializa o núcleo de Anahy */
   aInit(&argc, &argv);

   if (argc != 4) {
      printf ("syntax: %s [number] [load]  [stringsize]\n", argv[0]);
      exit (EXIT_FAILURE);
   }
   /* Obtém dados passados pela linha de comando */
   n = atoi (argv[1]);
   load = atoi(argv[2]);
   size = atoi(argv[3]);
   /* Inicializa os dados passados */
   entrada = (struct data *)malloc(sizeof(struct data));
   string = (char *) malloc(sizeof(char)*(size+1));
   seed = 'a';
   for(interator = 0 ; interator < size; interator++){
      strncat(string,a,1);
   }
   /*Insere os dados inicializados na estrutura usada pela aplicação */
   entrada->n = n;
   entrada->carga = load;
   entrada->dados = string;
  /* Seta os atributos da thread a serem utilizados, também seta as funções pack/unpac
```

```
   athread_attr_init(&attr);
   athread_attr_setinputlen(&attr, sizeof(struct data));
   athread_attr_pack_in_func(&attr, packfun);
   athread_attr_unpack_in_func(&attr, unpackfun);
   athread_attr_pack_out_func(&attr, packfun);
   athread_attr_unpack_out_func(&attr, unpackfun);
   /* Começa contagem de tempo e inicia a execução */
   initial_time();
   athread_create(&thread, &attr, fibo, (void *) entrada);
   athread_join(thread, &get);
   final_time();
   /* Desaloca dados não mais utilizados */
   free(entrada);
   /* Converte dados recebidos pelo join para seu tipo original */
   entrada = (struct data *) get;
   printf("Resultado: %d\nDados: %s\n", entrada->n, entrada->dados);
   /* Termina a execução do núcleo executivo */
   ret = aTerminate();
   exit (0);
}

int toma_tempo(int x) {
   int i,j;
   float a;

   for (j=0;j<x;j++){
      for (i=0; i<200000; ++i)
         a += sin(sin(cos(i)));
   }
   return (int) a;
}

void *fibo (void *void_data) {
   athread_t thread, thread2;
   athread_attr_t attr, attr2;
   struct data *parametros, *direita, *esquerda, *resposta;
   struct data *ret, *ret2;
   int nada, load, n, calc, tamanho;
   void *sync, *sync2;
```

```
/* Converte os dados de volta ao tipo original */
parametros = (struct data *) void_data;
/* Extrai os dados que interessam para a computação */
n = parametros->n;
load = parametros->carga;
tamanho = strlen(parametros->dados);
tamanho++;
/* Aloca espaço na heap para os dados */
resposta = malloc(sizeof(struct data));
resposta->dados = (char *) malloc(sizeof(char)*tamanho);
memcpy(resposta->dados,parametros->dados, tamanho);

if (n <= 2) {
   resposta->n = 1;
   return((void *) resposta);
} else {
   /* Cria dados de entrada para próximo nó */
   direita = malloc(sizeof(struct data));
   direita->n = n-1;
   direita->carga = parametros->carga;
   direita->dados = (char *) malloc(sizeof(char)*tamanho);
   memcpy(direita->dados, parametros->dados, tamanho);
   athread_attr_init(&attr);
   athread_attr_setinputlen(&attr, (3*sizeof(int))+tamanho);
   athread_attr_pack_in_func(&attr, packfun);
   athread_attr_unpack_in_func(&attr, unpackfun);
   athread_attr_pack_out_func(&attr, packfun);
   athread_attr_unpack_out_func(&attr, unpackfun);

   esquerda = malloc(sizeof(struct data));
   esquerda->n = n-2;
   esquerda->carga = parametros->carga;
   esquerda->dados = (char *) malloc(sizeof(char)*tamanho);
   memcpy(esquerda->dados, parametros->dados, tamanho);
   athread_attr_init(&attr2);
   athread_attr_setinputlen(&attr2, (3*sizeof(int))+tamanho);
   athread_attr_pack_in_func(&attr2, packfun);
   athread_attr_unpack_in_func(&attr2, unpackfun);
```

```
      athread_attr_pack_out_func(&attr2, packfun);
      athread_attr_unpack_out_func(&attr2, unpackfun);
      /* Cria as threads que irão calcular os próximos nós */
      athread_create(&thread, &attr, fibo, (void *)direita);
      athread_create(&thread2, &attr2, fibo, (void *)esquerda);
   }
   nada = toma_tempo(load);
   /* Sincroniza dados */
   athread_join(thread, &sync);
   athread_join(thread2, &sync2);

   ret = (struct data *) sync;
   ret2 = (struct data *) sync2;

   calc = ret->n + ret2->n;

   /* Limpa dados não mais utilizados */
   free(ret);
   free(ret2);

   resposta->n = calc;

   return((void *) resposta);
}

void *packfun(void *dados) {

   athread_msg_t *msg;
   struct data *interno;
   int cursor, tamanho;
   long total;

   interno = (struct data *) dados;
   cursor = 0;
   tamanho = strlen(interno->dados);
   tamanho++;
   total = (3*sizeof(int)) + (tamanho*sizeof(char));
   /* Cria buffer do pacote a ser enviado */
   msg = athread_msg_init(total);
```

```
   /* Insere dados no pacote */
   athread_msg_pack(msg, cursor, &(interno->n), sizeof(int));
   cursor += sizeof(int);
   athread_msg_pack(msg, cursor, &(interno->carga), sizeof(int));
   cursor += sizeof(int);
   athread_msg_pack(msg, cursor, &tamanho, sizeof(int));
   cursor += sizeof(int);
   athread_msg_pack(msg, cursor, interno->dados, tamanho);
   return (void *) msg;
};

void *unpackfun(void *msg) {

   struct data *dados;
   int cursor, tamanho;
   athread_msg_t *msg_interna;

   cursor = 0;
   dados = (struct data *) malloc(sizeof(struct data));
   /* Recupera pacote passado */
   msg_interna = (athread_msg_t *) msg;
   /* Retira do pacote os dados inseridos */
   athread_msg_unpack(msg_interna,cursor,&(dados->n),sizeof(int));
   cursor += sizeof(int);
   athread_msg_unpack(msg_interna,cursor,&(dados->carga),sizeof(int));
   cursor += sizeof(int);
   athread_msg_unpack(msg_interna,cursor,&tamanho,sizeof(int));
   dados->dados = (char *) malloc(tamanho*sizeof(char));
   cursor += sizeof(int);
   athread_msg_unpack(msg_interna,cursor,dados->dados,tamanho);
   return (void *) dados;
};
```

Bibliografia

[1] ALVERSON, G. A. et al. Abstractions for portable, scalable parallel programming. *IEEE Trans. on Parallel and Distributed Systems*, v. 9, n. 1, p. 71–86, jan. 1998.

[2] BLUMOFE, R. D. et al. Cilk: an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, v. 30, n. 8, p. 207–216, ago. 1995.

[3] GALILÉE, F. et al. Athapascan-1: on-line building data flow graph in a parallel language. *IEEE Annual Conference on Parallel Architectures and Compilation Techniques*, Paris, out. 1998.

[4] CAVALHEIRO, G. G. H.; REAL, L. C. V. Uma biblioteca de processos leves para a implementação de aplicações altamente paralelas. *WSCAD - Workshop em Sistemas Computacionais de Alto Desempenho, São Paulo*, p. 117–124, 2003.

[5] DENNEULIN, Y.; NAMYST, R.; MÉHAUT, J. F. Architecture virtualization with mobile threads. In: *Proc. of ParCo 97*. Amsterdam: Elsevier, 1998. (Advances in Parallel Computing, v. 12).

[6] PERANCONI, D. S. *Alinhamento de Seqüências Biológicas em Arquiteturas com Memória Distribuída*. Dissertação (Mestrado) — Universidade do Vale do Rio dos Sinos, 2005.

[7] SYSTEMS, O. S. for T.-P. Parallel sequencing and assembly line problems. *Acta Informatica*, v. 1, p. 200–213, 1972.

[8] HU, T. Parallel sequencing and assembly line problems. *Operations Research*, v. 19, n. 6, p. 841–848, 1961.

[9] CASAVANT, T. L.; KUHL, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, v. 14, n. 2, p. 141–154, Fevereiro 1988.

[10] CAVALHEIRO, G. G. H.; DENNEULIN, Y.; ROCH, J.-L. A general modular specification for distributed schedulers. *Proc. of Europar'98*, Southampton, set. 1998.

[11] TAERNVIK, E. Dynamo - a portable tool for dynamic load balancing on distributed memory multicomputers. In: *CONPAR '92/ VAPP V: Proceedings of the Second Joint International Conference on Vector and Parallel Processing*. London, UK: Springer-Verlag, 1992. p. 485–490. ISBN 3-540-55895-0.

[12] KONIG, J.-C.; ROCH, J.-L. Machines virtuelles et techniques d'ordonnancement. In: AL, D. B. et (Ed.). *ICaRE'97: Conception et mise en oeuvre d'applications parallèles irrégulières de grande taille*. Aussois: CNRS, 1997.

[13] YANG, T.; GERASOULIS, A. DSC: scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, v. 5, n. 9, p. 951–967, Sept 1994.

[14] IVERSON, M. A.; ÖZGÜNER, F. Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment. *Heterogeneous Computing Workshop*, p. 70–78, 1998.

[15] SINNEN, O.; SOUSA, L. List scheduling: extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. *Parallel Computing*, v. 30, n. 1, p. 81–101, 2004.

[16] SAKELLARIOU, R.; ZHAO, H. A hybrid heuristic for DAG scheduling on heterogeneous systems. *Proceedings of the IEEE Heterogeneous Computing Workshop*, 2004.

[17] BUYYA, R. *High Performance Cluster Computing: Architectures and Systems*. Indianapolis: Prentice Hall PTR, 1999.

[18] FEITELSON, D. G.; RUDOLPH, L. Parallel job scheduling: Issues and approaches. In: FEITELSON, D. G.; RUDOLPH, L. (Ed.). *Job Scheduling Strategies for Parallel Processing – IPPS'95 Workshop*. Santa Barbara: Springer, 1995. v. 949, p. 1–18.

[19] FEITELSON, D. G. *Job Scheduling in Multiprogrammed Parallel Systems*. IBM T. J. Watson Research Center, 1997.

[20] CAVALHEIRO, G. G. H. A general scheduling framework for parallel execution environments. *Proceedings of SLAB'01*, Brisbane, Australia, maio 2001.

[21] RINARD, M. C.; LAM, M. S. The design, implementation, and evaluation of jade. *ACM Transactions on Programming Languages and Systems*, v. 20, n. 3, p. 483 – 545, May 1998.

[22] HADJIDOUKAS, P.; POLYCHRONOPOULOS, E.; PAPATHEODOROU, T. Runtime support for multigrain and multiparadigm parallelism. *HIPC 2002*, Bangalore, 2002.

[23] VALIANT, L. G. A bridging model for parallel computation. *Communications of the ACM*, v. 33, n. 8, p. 103–111, Agosto 1990.

[24] FORTUNE, S.; WYLLIE, J. Parallelism in random access machines. *Proceedings of the tenth annual ACM symposium on Theory of computing*, p. 114–118, 1978.

[25] CULLER, D. E. et al. LogP: Towards a realistic model of parallel computation. *Principles Practice of Parallel Programming*, p. 1–12, 1993.

[26] CAVALHEIRO, G. G. H. et al. DPC++ an object-oriented distributed language. *XV International Conference of the Chilean Computer Science Society*, Arica, 1995.

[27] NIKHIL, R. S. Parallel symbolic computing in cid. In: *PSLS '95: Proceedings of the International Workshop on Parallel Symbolic Languages and Systems*. London, UK: Springer-Verlag, 1996. p. 217–242. ISBN 3-540-61143-6.

[28] RANDALL, K. *Cilk: Efficient Multithreaded Computing*. MIT/LCS/TR-749, 1998. 179 p.

[29] RINARD, M. C.; SCALES, D. J.; LAM, M. S. Jade: A high-level machine-independent language for parallel programming. *Computer*, v. 26, n. 6, p. 28–38, 1993.

[30] GOLDMAN, A. Modelos para computação paralela. *Escola Regional de Alto Desempenho*, Santa Maria, p. 35–66, 2003.

[31] YAMIN, A. C. Escalonamento em sistemas paralelos e distribuídos. *Escola Regional de Alto Desempenho*, Gramado, p. 73–126, 2001.

[32] GRAHAM, R. L. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, v. 17, n. 2, p. 416–429, mar. 1969. ISSN 0036-1399 (print), 1095-712X (electronic).

[33] SHMOYS, D. B.; WEIN, J.; WILLIAMSON, D. P. Scheduling parallel machines on-line. *SIAM Journal on Computing*, v. 24, n. 6, p. 1313–1331, December 1995.

[34] HWANG, J.-J. et al. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, v. 18, p. 244–257, Abril 1989.

[35] BLUMOFE, R.; LEISERSON, C. Scheduling multithreaded computations by work stealing. *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, Santa Fe, New Mexico, p. 356–368, November 1994.

[36] CORDEIRO, O. C. et al. Exploiting multithreaded programming on cluster architectures. In: *HPCS '05: Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications (HPCS'05)*. Washington, DC, USA: IEEE Computer Society, 2005. p. 90–96. ISBN 0-7695-2343-9.

[37] GHEZZI, C.; JAZAYERI, M. *Programming Language Concepts*. 3. ed. New York: John Wilei & Sons, 1998.

[38] DALL'AGNOL, E. C. et al. Construção de um mecanismo de comunicação para ambientes de processamento de alto desempenho. *Workshop em Sistemas Computacionais de Alto Desempenho*, 2004.

[39] CAVALHEIRO, G. G. H. et al. Anahy: a programming environment for cluster computing. *7th International Meeting on High Performance Computing for Computational Science*, 2006. A ser publicado.

[40] BENITEZ, E. D. et al. Avaliação de desempenho de anahy em aplicações paralelas. *XXIV Congresso da Sociedade Brasileira de Computação*, 2004.