UNIVERSIDADE FEDERAL DE PELOTAS
Instituto de Física e Matemática
Departamento de Informática
Bacharelado em Ciência da Computação

Trabalho Acadêmico

ApenMP: um suporte à OpenMP para o modelo Anahy

Cristian Fernando Flores Castañeda

Pelotas, 2008

Cristian Fernando Flores Castañeda

APENMP: UM SUPORTE À OPENMP PARA O MODELO ANAHY

Trabalho acadêmico apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Federal de Pelotas, como requisito parcial à obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Gerson Geraldo Homrich Cavalheiro

Pelotas, 2008

Dados de catalogação na fonte:
Ubirajara Buddin Cruz – CRB-10/901
Biblioteca de Ciência & Tecnologia - UFPel

C346a Castañeda, Cristian Fernando Flores

ApenMP : um suporte à OpenMP para o modelo Anahy / Cristian Fernando Flores Castañeda ; orientador Gerson Geraldo Homrich Cavalheiro. – Pelotas, 2008. –74f. : il. - Monografia (Conclusão de curso). Curso de Bacharelado em Ciência da Computação. Departamento de Informática. Instituto de Física e Matemática. Universidade Federal de Pelotas. Pelotas, 2008.

1.Informática. 2.Processamento de alto desempenho. 3.Multiprogramação leve. 4.OpenMP. 5.Programação paralela. 6.Algoritmos de listas. I.Cavalheiro, Gerson Geraldo Homrich. II.Título.

CDD:

005.275

Agradecimentos

Quero agradecer em primeiro lugar e acima de tudo às duas pessoas que me apoiaram durante todos estes anos: meu pai, Rufino Fernando Flores Cantillano, e minha mãe, Buby Marisol Castañeda Moreno. Sem o apoio, carinho e dedicação dessas duas pessoas extraordinárias, eu jamais teria conseguido chegar até aqui e conquistado todas as coisas que conquistei até hoje.

Também quero agradecer às minhas irmãs Daniela e Letícia e à minha namorada Médelin, por todo o apoio e suporte durante toda esta etapa.

Gostaria de agradecer também aos professores que tive contato durante todo o curso pelo comprometimento, dedicação e didática. Além do conteúdo acadêmico, aprendi muitas coisas com todos eles e sou muito grato por isso.

Também gostaria de agradecer aos meus colegas pela participação e colaboração, de alguma maneira, na realização deste trabalho de conclusão de curso. Espero que a amizade continue após a finalização do curso.

E por último, mas não menos importante, agradeço ao meu orientador Gerson Cavalheiro pela dedicação, apoio e pela orientação correta neste trabalho de conclusão de curso.

“Os conceitos e princípios fundamentais da ciência são invenções livres do espírito humano”.

Albert Einstein (1879 – 1955)

Resumo

CASTAÑEDA, Cristian Fernando Flores. **ApenMP: um suporte à OpenMP para o modelo Anahy**. 2008. 74f. Monografia – Curso de Bacharelado em Ciência da Computação. Universidade Federal de Pelotas, Pelotas.

Arquiteturas multiprocessadas são uma das opções para o processamento paralelo e de alto desempenho. Neste tipo de arquitetura, o modelo de programação que melhor aproveita as características do hardware é a multiprogramação leve, onde diversos fluxos de execução compartilham um mesmo espaço de endereçamento. Entre as diversas ferramentas de programação que implementam este modelo, OpenMP se destaca pela simplicidade de paralelização de códigos, oferecendo uma interface de programação voltada para extração de paralelismo de dados. Já em termos de escalonamento de tarefas, os algoritmos de lista são conhecidos como os mais eficientes para arquiteturas multiprocessadas, tendo sido aplicados com sucesso em Athreads. Athreads, diferentemente de OpenMP, oferece uma interface de programação com facilidades para descrição de um programa paralelo em termos de seu paralelismo de tarefas. Neste trabalho é apresentado o desenvolvimento de ApenMP, uma implementação de um modelo de execução baseado em dependências de tarefas com uma interface de programação definida segundo um subconjunto das especificações de OpenMP. ApenMP consiste em uma ferramenta de pré-processamento que converte código OpenMP em código Athreads, permitindo a execução de programas com suporte à estratégias de escalonamento de lista. O projeto de ApenMP considera o modelo de programação e execução proposto por Anahy e as especificações da interface de programação proposta pelo padrão OpenMP.

Palavras-chave: Multiprogramação leve. OpenMP. Programação paralela. Algoritmos de lista.

Abstract

CASTAÑEDA, Cristian Fernando Flores. **ApenMP: an OpenMP support for Anahy model**. 2008. 74f. Monografia – Curso de Bacharelado em Ciência da Computação. Universidade Federal de Pelotas, Pelotas.

Multiprocessors are one of the most popular architectures for parallel programming and high performance computing. For this kind of architecture, the best suitable programming model is the multithreading, which makes it possible to explore eficiently the parallel programming resources, since multiple execution flows share an addressing space as well as processors, in a multiprocessor architecture, share a global memory space. Among the differents programming tools that implement this model, OpenMP stands out by the simplicity of code parallelization, with its programming interface aimed to the extraction of data parallelism. Considering the execution support, we found in the list algorithms the scheduling strategies providing the best execution performance. We found in Athreads the application of such kind of scheduling providing good performance execution times. Nevertheless, the programming interface of Athreads offers facilities to describe the parallelism of a parallel program in terms of tasks. In this work, we present the development of ApenMP, an implementation of the execution model based on tasks dependencies providing a programming interface implementing a subset of the OpenMP standard. ApenMP is a pre-processing tool that converts OpenMP code in Athreads code, allowing the executions of programs with support of scheduling strategies. The ApenMP project considers the execution and programming model proposed by Anahy and the programming interface specifications proposed by OpenMP.

Keywords: Multithreading. OpenMP. Parallel programming. Lists Algorithms.

Lista de Figuras

Figura 2.1 - Arquitetura *multi-core*. 18

Figura 3.1 - Protótipo de uma função a ser paralelizada em Pthread. 21

Figura 3.2 - Protótipo das funções para criação, sincronização e término dos *threads*. 22

Figura 3.3 - Modelo de programação de OpenMP. 24

Figura 3.4 - Protótipo de uma região paralela em OpenMP. 25

Figura 3.5 - Exemplo de código utilizando o recurso *parallel*. 25

Figura 3.6 - Exemplo de código utilizando o recurso *for*. 26

Figura 3.7 - Funções para manipular o número de *threads* de serviço. 27

Figura 4.1 - Protótipos dos serviços de criação e sincronização de *threads* em Athreads. 30

Figura 4.2 - Grafo de dependência de tarefas em Anahy. 31

Figura 5.1 - Nodos diretivos e arestas no OMPCFG. 34

Figura 5.2 - a) Fases simples em uma região paralela. b) Fases quando há saltos. 36

Figura 5.3 - c) Fases em um laço. d) Fases em regiões paralelas aninhadas. e) Fases órfãs. 37

Figura 5.4 – Exemplo de um laço paralelo em linguagem C. 38

Figura 5.5 - Uma laço com a cláusula *schedule(runtime)*. 38

Figura 5.6 - *Overhead* no escalonamento na SUN HPC 3500. 41

Figura 5.7 - *Overhead* no escalonamento na SGI Origin 2000. 41

Figura 5.8 - *Overhead* no escalonamento na Compaq Alpha Server. 42

Figura 6.1 - Processo de compilação com ApenMP. 43

Figura 6.2 - Esquema da geração de ApenMP. 45

Figura 6.3 - Código utilizando a diretiva *parallel* e cláusula *private*.....54

Figura 6.4 - Código utilizando a diretiva *parallel* e cláusula *firstprivate*. 55

Figura 6.5 - Código utilizando a diretiva *parallel* e cláusula *reduction*.....56

Figura 6.6 - Código utilizando a diretiva *for* e cláusula *private***.** 57

Figura 6.7 - Código utilizando a diretiva *for* e cláusula *firstprivate*. 58

Figura 6.8 - Código utilizando a diretiva *for* e cláusula *lastprivate*.....59

Figura 6.9 - Código utilizando a diretiva *for* e cláusula *reduction*. 60

Figura 7.1 - Gráfico mostrando o tempo de execução dos algoritmos em OpenMP(diretiva *parallel*), ApenMP e Pthreads.....63

Figura 7.2 - Gráfico mostrando o tempo de execução dos algoritmos em OpenMP(diretiva *for*), ApenMP e Pthreads. 64

Figura 7.3 - *Overhead* no escalonamento sobre diferentes *chunks*. 65

Figura 7.4 - Gráfico do tempo de execução dos 3 casos de paralelização. 67

Lista de Tabelas

Tabela 7.1 - Tempo de execução dos algoritmos.....62
Tabela 7.2 - Tempo de execução dos algoritmos.....64
Tabela 7.3 - *Overhead* no escalonamento. 65
Tabela 7.4 - Tempo de execução dos 3 casos de paralelização.....67

Lista de Abreviaturas e Siglas

API – *Application Programming Interface*

DAG – *Directed Acyclic Graph*

DSM – *Distributed Shared Memory*

GCC – *GNU Compiler Collection*

IEEE – *Institute of Electrical and Electronics Engineers*

MIMD – *Multiple Instruction Stream, Multiple Data Stream*

MISD – *Multiple Instruction Stream, Single Data Stream*

MP – *Message Passing*

NUMA – *Non-Uniform Memory Access*

PAD – *Processamento de Alto Desempenho*

PV – *Processador Virtual*

RISC – *Reduced Instruction Set Computer*

SIMD – *Single Instruction Stream, Multiple Data Stream*

SISD – *Single Instruction Stream, Single Data Stream*

SMP – *Symmetric Multiprocessing*

TLP – *Thread Level Parallelism*

UMA – *Uniform Memory Access*

Sumário

1 Introdução 12
1.1 Motivação 13
1.2 Objetivos 14
1.3 Resultados alcançados 14
1.4 Organização do trabalho 15

2 Arquiteturas Multiprocessadas 16
2.1 Categorização das arquiteturas 16
2.2 Processadores *multi-core* 18

3 Multiprogramação Leve 20
3.1 Pthreads 21
3.1.1 Modelo de programação 21
3.1.2 Recursos de programação 21
3.1.3 Escalonamento 23
3.2 OpenMP 23
3.2.1 Modelo de programação 24
3.2.2 Recursos de programação 25
3.2.3 Escalonamento 27

4 Anahy 29
4.1 Recursos de programação 30
4.2 Escalonamento 32

5 Paralelismo no OpenMP 34
5.1 Análise da concorrência 34

5.1.1 OpenMP Control Flow Graph 34
5.1.2 OpenMP Region Tree 35
5.1.3 Fases na região paralela 36
5.2 Análise do escalonamento 37
5.2.1 *Overhead* no escalonamento 40

6 ApenMP 43
6.1 Construção do Pré-processador 43
6.1.1 Flex e Bison 44
6.1.2 Geração de código 45
6.2 Funcionamento do Pré-processador 49
6.3 Código gerado 51
6.4 Exemplos 53

7 Resultados de desempenho 61
7.1 Metodologia utilizada 61
7.2 Resultados 62
7.2.1 Paralelismo de tarefas 62
7.2.2 Paralelismo de dados 63
7.3 *Overhead* no escalonamento de Anahy 65
7.4 Outros experimentos 66

8 Conclusões e Trabalhos futuros 69

Referências 70

1 Introdução

O processamento paralelo é uma estratégia de computação utilizada com freqüência em áreas onde a demanda por recursos de processamento é alta. Recentemente os horizontes de aplicação de soluções paralelas se ampliaram com a popularização de configurações para máquinas multiprocessadas devido ao barateamento de processadores *multi-core*. Conseqüentemente, a demanda pelo processamento paralelo aumentou consideravelmente, aumentando, em conseqüência, a utilização de ferramentas de programação *multithread*.

As ferramentas comerciais mais populares para multiprogramação leve são Pthreads (DREPPER; MOLNAR, 2005) e OpenMP (DAGUM; MENON, 1998). Embora o modelo de programação *multithread* esteja implementado tanto em Pthreads como em OpenMP, cada uma destas ferramentas explora um aspecto diferente de concorrência das aplicações. Enquanto Pthreads provê uma interface de programação voltada a destacar o paralelismo de tarefas da aplicação, OpenMP provê uma interface voltada ao paralelismo de dados (AKHTER; ROBERT, 2006). No entanto, nenhuma destas ferramentas emprega estratégias de escalonamento de tarefas que permitam obter o máximo do desempenho de uma arquitetura paralela pela aplicação de algoritmos de escalonamento eficientes. Ferramentas que possibilitam integrar a multiprogramação leve e tal classe de algoritmo de escalonamento são encontradas no meio acadêmico, como Cilk (BLUMOFE et al., 1996) e Athreads (CAVALHEIRO et al., 2006).

Athreads é uma implementação do modelo proposto por Anahy. Este modelo propõe um ambiente de execução para arquiteturas paralelas no qual uma camada de software é responsável pela construção de um grafo de fluxo de dados, descrevendo as relações de dependência entre as atividades concorrentes criadas pelo programa em execução (CAVALHEIRO, 2004; LEOPOLD, 2001). Este grafo é submetido a um mecanismo de execução responsável por realizar o escalonamento das atividades sobre os recursos de processamento disponíveis. Athreads foi desenvolvida para respeitar um subconjunto de recursos de programação definidos

pelo padrão Pthreads (CAVALHEIRO et al., 2006). Experimentos realizados com Athreads (CORDEIRO et al., 2005) mostraram resultados de desempenho satisfatórios para o protótipo construído.

Este trabalho propõe a construção de ApenMP, uma ferramenta para programação em arquiteturas multiprocessadas. A interface de ApenMP é baseada no padrão OpenMP. O projeto de ApenMP considera o modelo de programação e execução proposto por Anahy e as especificações da interface de programação proposto pelo padrão OpenMP.

1.1 Motivação

Com a popularização das arquiteturas multiprocessadas, novas aplicações devem ser criadas ou reelaboradas para aproveitarem ao máximo o poder de processamento oferecido por arquiteturas paralelas com memória compartilhada. Tais aplicações devem ser construídas utilizando ferramentas que ofereçam recursos de multiprogramação leve e empreguem estratégias de escalonamento eficientes. Algumas motivações são destacadas a seguir:

- Os bons resultados de Athreads (Anahy com interface Pthreads) indicam que algoritmos de escalonamento eficientes, tais como os algoritmos de lista propostos por Graham (1969), podem ser aplicados de forma dinâmica com implicações satisfatórias em desempenho.

- OpenMP tem ocupado um papel de destaque em termos de popularidade entre os desenvolvedores de software paralelo devido a simplicidade de sua interface de programação.

Outra motivação a ser destacada é a projeção de Anahy com uma interface de programação de OpenMP. Esta interface, diferentemente de Pthreads, explora o paralelismo sem modificar a estrutura do algoritmo que irá ser paralelizado. Sendo assim, a paralelização pode ser feita de modo incremental no código da aplicação (OLIVEIRA et al., 2007). Desta forma a paralelização de um algoritmo utilizando OpenMP torna-se mais simples.

1.2 Objetivos

O objetivo principal deste trabalho foi a implementação do modelo Anahy com uma interface OpenMP, este objetivo foi atingido com o protótipo desenvolvido. É importante também destacar outros objetivos específicos atingidos com o desenvolvimento deste trabalho:

- Criação de um pré-processador para a tradução do código OpenMP em Athreads;
- Construção de funcionalidades para implementar o suporte às diretivas e às cláusulas de OpenMP;
- Avaliação dos resultados de desempenho obtidos.

1.3 Resultados alcançados

Os objetivos do trabalho foram atingidos com a implementação do processo de geração de código Athreads a partir de um código fonte OpenMP em um pré-processador denominado ApenMP. Com esta implementação foi possível mostrar que o modelo Anahy pode ser utilizado em uma interface de programação baseada na extração de paralelismo de dados.

Dada a extensão do padrão OpenMP, foi necessário delimitar o escopo de abrangência da implementação a ser realizada, sendo selecionadas as diretivas *parallel* e *for* e as cláusulas *private*, *lastprivate*, *firstprivate* e *reduction* como as relevantes como prova de conceito. As diretivas e cláusulas selecionadas permitem expressar um código paralelo em termos de dependências de dados entre tarefas. Também foi necessário analisar o impacto do uso destes recursos de programação no modelo de Anahy.

1.4 Organização do trabalho

A seqüência desta monografia se dá em oito capítulos, como detalhado a seguir.

No Capítulo 2 são apresentados os processadores *multi-core*, descrevendo suas características e vantagens deste tipo de arquitetura.

O Capítulo 3 apresenta o modelo de multiprogramação leve e as vantagens de sua utilização em arquiteturas multiprocessadas. Também neste capítulo são apresentados dois padrões para programação *multithread*: Pthreads e OpenMP.

O Capítulo 4 descreve o modelo Anahy e apresenta Athreads, o protótipo em operação deste modelo. São apresentados o modelo de programação de Anahy, os recursos de programação de Athreads e a implementação do escalonamento de listas realizado neste ambiente.

O Capítulo 5 descreve como ocorre a análise da concorrência de um programa e os diversos escalonamentos do padrão OpenMP.

O Capítulo 6 trata exclusivamente da implementação. Neste capítulo são apresentadas as etapas de construção e de utilização desta ferramenta. Também é apresentado o funcionamento de ApenMP com exemplos de código.

O Capítulo 7 documenta, por meio de tabelas e gráficos, resultados de desempenho dos códigos mostrados no sexto capítulo.

Por fim, no oitavo capítulo, são apresentadas as conclusões dos experimentos e algumas sugestões para eventuais trabalhos futuros.

2 Arquiteturas Multiprocessadas

Nas últimas décadas o aumento da freqüência de *clock* foi um dos fatores determinantes que caracterizavam o aumento de desempenho das aplicações sobre as arquiteturas. Além disso, o paralelismo de instruções era explorado por um único *thread* (fluxo de execução) (CAVALHEIRO; SANTOS, 2007).

Desse modo, foi percebido que se tornou inviável o fato do desempenho das aplicações em si estivesse relacionado somente ao aumento da freqüência de *clock*, ou com mecanismos que explorem o paralelismo limitado a um único *thread*. Sendo assim, novas abordagens foram encaminhadas e o aparecimento das arquiteturas *multi-core* foram constatadas como alternativa para este tipo de problema. Através de sua popularidade, a tecnologia *multi-core* disponibiliza as diversas vantagens deste modelo para os usuários domésticos e até para grandes aglomerados de computadores (CAVALHEIRO; SANTOS, 2007).

2.1 Categorização das arquiteturas

A idéia de usar múltiplos processadores com a finalidade de melhorar o desempenho e/ou a disponibilidade não é recente. Flynn (FLYNN, 1966, 1972), propôs uma taxonomia simples para categorização das arquiteturas baseando-se no fluxo de instruções e dados observados. Esta taxonomia, ainda atualmente empregada, é composta por quatro categorias:

- SISD – *Single Instruction Stream, Single Data Stream*: fluxo único de instruções operando sobre um fluxo único de dados. Nesta categoria encontram-se os uniprocessadores;
- SIMD – *Single Instruction Stream, Multiple Data Stream*: uma mesma instrução pode operar sobre um conjunto de dados. Nesta categoria estão os computadores que exploram o paralelismo de dados;

- MISD – *Multiple Instruction Stream, Single Data Stream*: múltiplos fluxos de instrução operando sobre um único fluxo de dados. Não existem exemplares desta categoria no mercado (HENNESSY; PATTERSSON, 2007), no entanto alguns autores classificam arquiteturas *pipelined* ou *dataflow* como MISD;

- MIMD – *Multiple Instruction Stream, Multiple Data Stream*: cada processador executa um fluxo de instruções independente que opera sobre um conjunto de dados particular. Computadores MIMD exploram o paralelismo de *threads* (TLP – *Thread Level Parallelism*).

As arquiteturas MIMD possuem diferentes organizações e ainda podem ser classificadas em três categorias, mostradas a seguir. As subdivisões de MIMD não fazem parte da taxonomia original proposta por Flynn.

- SMP – *Symmetric Multiprocessing*: arquitetura baseada em múltiplos processadores que compartilham uma mesma memória. Também chamada de UMA (*Uniform Memory Access*) ou Multiprocessador, ou ainda MIMD Fortemente Acoplada. Nesta categoria, a memória compartilhada é centralizada;

- NUMA – *Non-Uniform Memory Access*: arquitetura baseada na existência de memórias dedicadas para cada processador ou grupo de processadores. Neste caso conectado à memória por meio de um *crossbar* ou *switch* e uma área de memória compartilhada distribuída (DSM – *Distributed Shared Memory*). A memória dedicada tem como finalidade reduzir as latências de acesso aos dados para cada processador, enquanto a memória compartilhada permite que diferentes fluxos de instruções (processadores) possam acessar uma área comum de endereçamento. Estas arquiteturas podem ser consideradas fortemente acopladas;

- *Cluster*: arquitetura baseada em múltiplos computadores, também chamada de multicomputador ou MIMD Fracamente Acoplada, que trocam informações por meio de troca de mensagens via uma rede de interconexão (MP – *Message Passing*).

Em relação às arquiteturas paralelas, é importante ressaltar que atualmente o paralelismo é explorado em diferentes níveis, em praticamente todas as arquiteturas. Assim, o paralelismo é explorado desde sua granulosidade mais fina, no nível de instrução em uniprocessadores, até a exploração do paralelismo no nível de *threads* em máquinas MIMD comunicando-se via troca de mensagens (ROSE; NAVAUX, 2003).

2.2 Processador *multi-core*

Um processador é denominado *multi-core* quando possui dois ou mais núcleos completos de processamentos idênticos instalados fisicamente no mesmo *chip* (KUMAR; TULLSEN, 2007). Cada núcleo corresponde a uma unidade de processamento completa. A arquitetura básica desse tipo de processador (Fig. 2.2) segue o modelo **SMP** (*Symmetric Multi-Processors* – Múltiplos Processadores Simétricos), cujas características básicas são: execução independente de fluxos de instrução distintos para cada processador ativo e compartilhamento de memória (CAVALHEIRO; SANTOS, 2007).

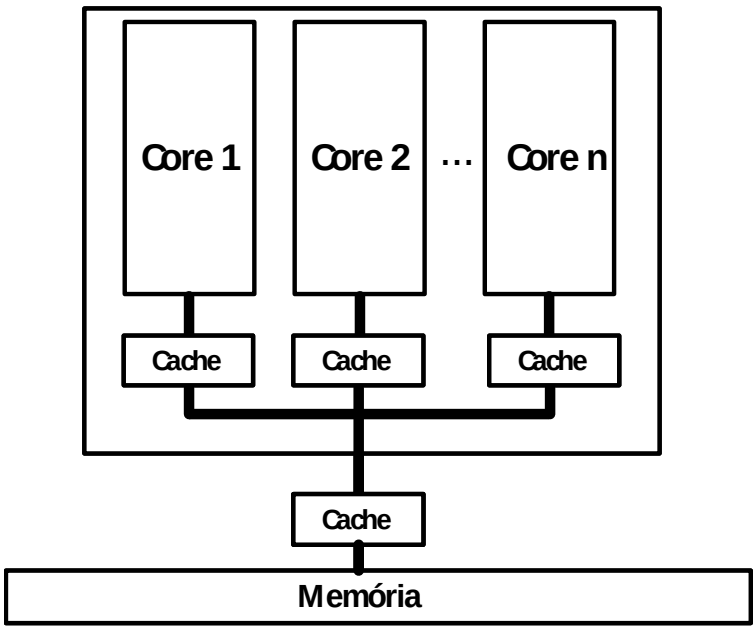

Figura 2.1 - Arquitetura *multi-core*.

Arquiteturas *multi-core* são capazes de prover maior capacidade de processamento com um custo/benefício melhor do que processadores *single-core*

(um único processador). A vantagem dos processadores *multi-core* é a possibilidade de aumentar o potencial de desempenho, com menor impacto no consumo de energia e geração de calor incorrida pelo mesmo ganho de desempenho em otimizações realizadas com a tecnologia de processadores *single-core*.

Algumas vantagens de processadores *multi-core* são:

- Maior eficácia do sistema e desempenho aprimorado de programas em computadores executando vários aplicativos simultaneamente;
- Desempenho aprimorado para aplicativos *multithreaded*;
- Redução da dissipação térmica quando comparado ao *single-core*;
- Melhora o paralelismo no nível de *threads*;
- Melhor localidade de dados;
- Melhor comunicação entre as unidades;

Também em relação aos processadores multi-core, podem ser citadas algumas desvantagens:

- A tecnologia *multi-core* é desperdiçada para programas não paralelos.
- Acontece uma queda de desempenho na inserção de novos *cores*, o motivo desta queda é a dificuldade de comunicação de banda que os *cores* encontram com a memória.

Finalmente, é possível destacar que as arquiteturas *multi-core* representam uma alternativa bastante viável em termos de aumento de desempenho e diminuição do consumo de energia. Entretanto, para uma total utilização do poder de processamento oferecido pelo *multi-core*, as aplicações devem ser escritas de modo a usar intensivamente o conceito de *threads,* ou seja, serem elaboradas ou reescritas segundo os padrões da multiprogramação leve.

Atualmente, a tecnologia *multi-core* é empregada por diversas empresas fabricantes de hardware e altamente conceituadas no mercado de computadores, como a Intel e a AMD.

3 Multiprogramação Leve

A multiprogramação leve está estreitamente relacionada com as arquiteturas *multi-core*, pois seu modelo de programação reflete este tipo de arquitetura. A multiprogramação leve oferece recursos de programação, com os quais uma aplicação pode ser descrita em termos de um programa composto de diversos fluxos de execução (*threads*) capazes de executar de forma concorrente. Assim, um programa *multithreaded* possui diversos fluxos de execução que podem estar ativos em um determinado instante de tempo e, a exemplo de programas imperativos seqüenciais, cada fluxo possui uma área de endereçamento própria. A diferença é que além desse espaço de memória privado, os *threads* compartilham acesso a um espaço de endereçamento global. A concorrência, neste caso, expressa não somente a disputa de recursos de processamento, como tempo de processador e espaço de memória, mas também a disputa dos *threads* de um programa pelo acesso aos dados armazenados em memória (CAVALHEIRO; SANTOS, 2007).

Como foi dito anteriormente, a relação entre a multiprogramação leve e a arquitetura SMP é praticamente imediata. Enquanto o modelo de arquitetura física prevê a existência de diversos processadores e uma memória comum, o modelo de programa *multithread* prevê a coexistência de diversos fluxos de execução, que podem usar um espaço de endereçamento compartilhado para comunicar dados entre si. Essa relação é explorada por diversas ferramentas de programação, as quais oferecem serviços para controle do número de *threads* ativos simultaneamente, e para sincronizar as ações destes no acesso aos dados compartilhados. Uma prática bastante comum, embora não única, é explorar essa relação de complementaridade entre a multiprogramação leve e arquiteturas multiprocessadas, para obter programas eficientes em termos de desempenho (CAVALHEIRO; SANTOS, 2007).

Algumas destas ferramentas *multithreaded* mais populares na atualidade são **Pthreads** e **OpenMP**, sendo melhor descritas nas seções subseqüentes.

3.1 Pthreads

Pthreads, ou também chamado de POSIX threads, é um padrão criado pela IEEE em 1995 (IEEE POSIX 1003.1c, 1995). Embora disponíveis na maioria dos sistemas operacionais, como no Microsoft Windows (BUTENHOF, 1997), Pthreads é amplamente popular em sistemas operacionais do tipo UNIX, como Linux e Solaris.

Pthreads oferece mecanismos de sincronização, como semáforos, mutexes e variáveis condicionais, além de mecanismos para o gerenciamento e manutenção dos diversos *threads* pertencentes a um sistema (CAVALHEIRO, 2004). Nas próximas subseções são descritas algumas funções responsáveis por estas finalidades.

3.1.1 Modelo de programação

O modelo de programação proposto por Pthreads consiste de um conjunto de *threads* que executam determinada instrução dentro de uma ou mais funções definidas pelo programador. Os *threads* co-existem num mesmo processo, compartilhando vários recursos, mas são escalonados separadamente pelo sistema operacional. Pthreads segue o modelo *fork/join* para criação e sincronização de *threads*.

3.1.2 Recursos de programação

Em Pthreads, o lançamento de um novo *thread* é associado à execução da seqüência de instruções definidas no corpo de uma função. Esta função deve receber como parâmetro um endereço para uma área de memória, correspondendo ao local onde se encontram os parâmetros de entrada da função, retornando outro endereço de memória, onde se encontram os resultados produzidos pela função. Ambos os endereços são do tipo *void*. Na Fig. 3.1 é mostrado o protótipo de uma função Pthread.

```
void *func(void *args);
```

Figura 3.1 - Protótipo de uma função a ser paralelizada em Pthread.

A criação e término dos *threads* são realizados de forma explícita. A invocação da primitiva *pthread_create* por um *thread* implica a criação de um novo fluxo. Um *thread* termina sua execução quando executar toda a seqüência de instruções definida em seu corpo, ou quando executar o comando *return* convencional de C – alternativamente, invocando o serviço *pthread_exit*. O protótipo destas funções é ilustrado na Fig. 3.2.

```
int pthread_create( pthread_t *thid, pthread_attr_t *atrib,
void *(*funcao) (void *), void *args );
int pthread_join( pthread thid, void **ret );
int pthread_exit( void *retval );
```

Figura 3.2 - Protótipo das funções para criação, sincronização e término dos *threads*.

O retorno inteiro destes serviços corresponde a um código de erro. Caso o valor zero seja retornado, a operação obteve sucesso. O mesmo é válido para os demais serviços desta interface.

Na função *pthread_create* o novo *thread* a ser criado deve executar o código especificado pela função *funcao*. Este *thread* poderá ser referenciado futuramente por meio do valor retornado no parâmetro *thid*; este valor é único por *thread* durante toda a execução do programa. O parâmetro *args* indica o endereço de memória de onde devem ser recuperados os dados passados à função. A especificação Pthread permite ainda aplicar atributos de execução ao novo *thread*. Estes atributos são passados à biblioteca pelo parâmetro *atrib*.

Uma vez executada uma operação de criação de *thread*, de forma assíncrona, o novo fluxo será criado. Em outras palavras, o padrão não define quando o novo *thread* será disparado e nem como se dará a sobreposição do tempo de uso do processador por este novo *thread* com os *threads* já em curso de execução. Por outro lado, o modelo oferece o serviço *pthread_join*, que permite a sincronização de um *thread*, aquele executando a invocação deste serviço, com outro *thread*, aquele identificado por *thid*. A sincronização por meio de *pthread_join* garante que um *thread* continue sua execução apenas quando um determinado *thread* tenha terminado sua execução – adicionalmente, o parâmetro *ret* é atualizado com o endereço de memória que contém os dados retornados pelo *thread* sincronizado quando este terminar. Caso o *thread* sincronizado já tenha terminado,

ocorre simplesmente a recuperação do retorno. Caso contrário, o *thread* que necessita a sincronização permanecerá bloqueado até que esta seja satisfeita (CAVALHEIRO; SANTOS, 2007).

Uma característica deste mecanismo de sincronização é que cada *thread* pode sofrer apenas uma operação de *join*. Caso o retorno do *thread* seja útil a dois ou mais pontos do programa, o programador deve usar estratégias de programação próprias.

3.1.3 Escalonamento

Atualmente, o padrão do escalonamento de Pthreads é realizado pelo sistema operacional e baseado em prioridades. A política de escalonamento especifica quando os processos passam do estado *run* para o estado *ready*.

O escalonamento em POSIX *threads* possui três tipos e é defino pelo serviço *setschedpolicy*:

- SCHED_RR: implementa uma política do tipo *Round-Robin;*
- SCHED_FF: implementa uma política do tipo *First-in First-out*;
- SCHED_OTHER: algoritmo de escalonamento especificado pelo programador.

Os escalonamentos *SCHED_RR* e *SCHED_FF* estão disponíveis apenas quando o processo em execução tiver privilégios de *superusuário*. Estes tipos de escalonamentos são definidos para aplicações em *tempo-real* e sua prioridade é superior à de processos com política SCHED_OTHER.

3.2 OpenMP

OpenMP é um acrônimo para Open Multi-processing. OpenMP basicamente é uma interface de programação proposta em 1997 que oferece diversos recursos para a criação, desenvolvimento e execução de programas *multithread*. O suporte a esta API encontra-se implementado em C/C++ e Fortran sobre diferentes sistemas

computacionais, incluindo plataformas Unix e Microsoft Windows. O modelo OpenMP propicia aos programadores uma interface eficiente e flexível para desenvolver aplicações paralelas diversas, desde *desktops* até supercomputadores. É composta por uma série de recursos, tais como: diretivas de compilação, chamadas a funções de biblioteca e variáveis de ambiente.

3.2.1 Modelo de programação

OpenMP baseia-se no modelo *fork/join*, para criação e sincronização de *threads* (Fig. 3.3). Há um fluxo de execução principal (*thread mestre*) e, quando necessário, computações concorrentes são disparadas para dividir a carga total de trabalho em uma seção paralela. Por fim, ao término de uma seção paralela, é realizada uma operação *join*, retornando o fluxo de execução para o fluxo mestre.

Na nomenclatura OpenMP, o termo *threads* refere-se aos fluxos de execução que suportam a execução do programa. O programa, quando em execução, dispara a criação de trechos de código que podem ser executados como regiões paralelas. Os *threads* de suporte à execução são reunidos em um núcleo denominado *thread-pool*. O número de *threads* neste núcleo de execução pode ser alterado dinamicamente (CHANDRA et al., 2001).

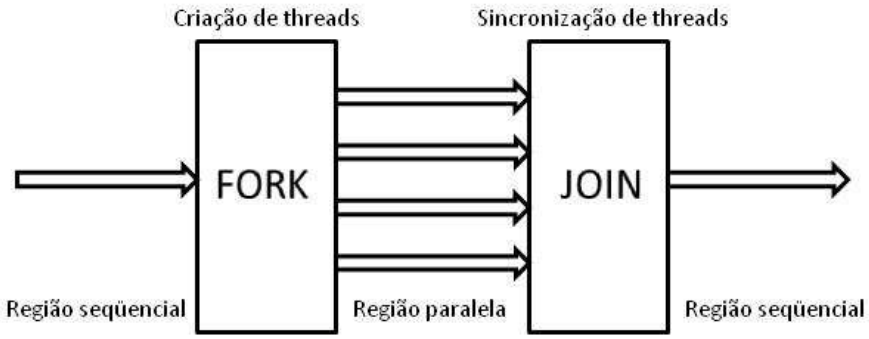

Figura 3.3 - Modelo de programação de OpenMP.

Um recurso interessante de OpenMP é que a sincronização entre os *threads* quase sempre ocorre de maneira implícita. Isso faz com que sua utilização seja simples. Outro aspecto importante é que os processos de criação e inicialização de regiões paralelas, assim como a divisão de trabalho realizada sobre um arranjo ou

vetor, não são visíveis ao programador. Isso garante a transparência na utilização pelo usuário (CHANDRA et al., 2001).

3.2.2 Recursos de programação

A interface de programação de OpenMP emprega diretivas de compilação (*pragmas*) como abstração para as operações *fork/join*. Chamadas a serviços de biblioteca permitem interação do programa em execução com o ambiente de execução. As diretivas de compilação OpenMP são compostas por sentinelas, diretivas e cláusulas e o protótipo da estrutura básica é exemplificado na Fig. 3.4.

```
#pragma omp diretiva [cláusula]
  Corpo
```

Figura 3.4 - Protótipo de uma região paralela em OpenMP.

Nesta sintaxe, *#pragma omp* é chamado sentinela, sendo dependente da linguagem de programação utilizada. A diretiva do OpenMP é identificada em *diretiva*, opcionalmente acompanhada de uma ou mais *cláusulas*. O corpo para os *threads* é definido em *Corpo*, composto por um bloco de instruções ou por um comando de iteração por bloco. As operações *fork* e *join* encontram-se implícitas nesta estrutura: o *fork* é especificado na própria linha onde o *pragma* se encontra e o *join* no final do bloco que especifica o corpo das atividades concorrentes. Uma diretiva importante para expressar paralelismo é *parallel*. Esta diretiva (Fig. 3.5) implica a criação de *thread*s, que deverão ser executados pelos *threads* de serviço do ambiente de execução (CAVALHEIRO; SANTOS, 2007).

```
int main() {
  int x = 100, y = 200;
  #pragma omp parallel private(x) {
      x = y + 1;
  }
  return 0;
}
```

Figura 3.5 - Exemplo de código utilizando o recurso *parallel*.

Outro importante recurso disponível em OpenMP é a possibilidade de realizar a paralelização de laços, que permite o compartilhamento de trabalho

gerado pela execução iterativa de um laço. A diretiva empregada para este tipo de paralelização é *for*. O código na Fig. 3.6 apresenta o uso deste recurso:

```
int main(int argc, char** argv) {
     int vet[50], i, somatorio = 0;
     #pragma omp for
       for( i = 0 ; i < 50 ; i++ ) vet[i] = i;
     #pragma omp for reduction(+:somatorio)
       for( i = 0 ; i < 50 ; i++ ) somatorio += vet[i];
     printf("Somatorio = %d\n", somatorio);
     return 0;
}
```

Figura 3.6 - Exemplo de código utilizando o recurso *for*.

Algumas regras devem ser seguidas para construção do laço que irá ser paralelizado. A variável de iteração necessita ser do tipo inteiro com sinal. Da mesma forma, o valor de teste da condição de término do laço e o valor a ser adicionado ou subtraído da variável de iteração a cada iteração realizada devem ser valores inteiros com sinal. Estes dois valores não podem variar durante toda execução do laço. Outro aspecto importante é que o corpo do laço possui apenas uma entrada e uma saída, ou seja, comandos como a instrução *break* de C/C++ não podem, portanto ser utilizados (CAVALHEIRO; SANTOS, 2007).

Embora de aparência e utilização simples, na prática a paralelização de laços não é uma tarefa trivial. Um conjunto de regras deve ser seguido para eficiência na utilização deste recurso. O programador deve evitar situações nas quais a passagem por uma iteração qualquer dependa da execução de uma passagem anterior, como no caso onde a computação da passagem *i* depende dos resultados da computação da passagem *i−1* (CAVALHEIRO; SANTOS, 2007).

Em relação às cláusulas, estas servem para instanciar a comunicação de dados entre os *threads*. O comportamento padrão implica que todos os dados locais ao *thread* original sejam acessíveis aos *threads* criados (*shared*). A cláusula *private* indica que cópias locais devem ser instanciadas para uso local a cada *thread*. Por outro lado, utilizando a cláusula *firstprivate* é possível herdar o último valor de uma variável antes do disparo dos *threads*.

Outras cláusulas como a *lastprivate* faz com que o valor computado pelo último *thread* do programa seja o valor utilizado pelo *thread* original após a

sincronização. Por outro lado, a cláusula *reduction* é mais elaborada, permitindo a combinação de resultados parciais dos *threads* ao final da região paralela.

Outro aspecto importante é a possibilidade de alterar dinamicamente o número de *threads* de serviço invocando as seguintes funcionalidades oferecidas em tempo de execução pela biblioteca OpenMP (Fig. 3.7):

```
int omp_get_num_threads();
void omp_set_num_threads(int);
int omp_get_thread_num();
```

Figura 3.7 - Funções para manipular o número de *threads* de serviço.

Em OpenMP o número de *threads* de serviço corresponde ao valor associado à variável de ambiente *OMP_NUM_THREADS*, entre outras palavras, o número de *threads* em execução concorrente será dependente deste número. As primitivas *get_/set_num_threads* permitem, respectivamente, obter o número de *threads* criados e alterar o valor associado à *OMP_NUM_THREADS* no contexto do programa para as próximas criações de *threads*. A primitiva *omp_get_thread_num* informa a um *thread* seu número de identificação dentre os *threads* do *thread-pool*. É importante destacar que o valor retornado estará, necessariamente, entre 0 e o *OMP_NUM_THREADS - 1.*

3.2.3 Escalonamento

Em OpenMP o escalonamento das regiões paralelas obedece a dependência definida pelo fluxo de execução do programa. Em OpenMP é possível o uso de estratégias para ajustar o particionamento e distribuição das regiões paralelas entre os *threads* de serviço. No entanto, OpenMP não aplica um mecanismo baseado em algoritmos de listas para controlar a execução de programas. No OpenMP existem quatro recursos para indicar o tipo de escalonamento que irá ser utilizado no programa. Estas cláusulas indicam a estratégia que deve ser utilizada para distribuir o trabalho no *thread-pool*. Estes recursos são descritos a seguir:

- Static: Implica na divisão do trabalho por igual entre cada *thread* do *thread-pool*. É indicado quando a quantidade de trabalho em cada grupo de instruções é a mesma;

- Dynamic: Cada *thread* do *thread-pool* busca uma nova quantidade de trabalho quando terminar uma etapa de processamento. Indicado quando a quantidade de trabalho em cada grupo de instruções não é a mesma;

- Guided: Semelhante ao *dynamic*, mas a carga de trabalho inicia grande e se torna menor exponencialmente. A carga de trabalho é obtida através da divisão do número de iterações restantes pelo número de *threads* no *thread-pool*. Indicado quando existe execução assíncrona.

- Runtime: O escalonamento é deixado para ser determinado para a execução, sendo determinado pelo valor da variável de ambiente *OMP_SCHEDULE*.

4 Anahy

O projeto Anahy apresenta o modelo de um ambiente de execução para arquiteturas paralelas. É composto por uma interface de programação e de um núcleo executivo capaz de controlar a execução de tarefas geradas pelo programa em execução de forma a garantir uma semântica de execução coerente com a definida pelo programador. O projeto Anahy tem como foco a questão do desempenho, permitindo que o desempenho de execução de um programa seja escalável conforme a configuração do hardware disponível.

O enfoque dado para o modelo da implementação do ambiente Anahy tem como premissa básica a dissociação da descrição da concorrência da aplicação do paralelismo real disponível na arquitetura. Assim, o programador pode definir o número de atividades concorrentes de sua aplicação desconsiderando os recursos de processamento disponíveis. A exploração da arquitetura é realizada por um núcleo executivo responsável pelo escalonamento. Este núcleo tem por função adaptar o número de atividades realizadas em paralelo de acordo com a capacidade de processamento disponível (CAVALHEIRO et al., 2006).

Para obter tal modelo, foi definida uma abstração de tarefa concorrente em nível usuário. Nesta abstração, um *thread* consiste em um fluxo de execução não bloqueante que recebe parâmetros de entrada e que retorna resultados de seu processamento. Um *thread* em Anahy, portanto, não realiza nenhuma operação de sincronização, exceto criação de novos *threads* e obtenção de resultados de retorno (*join*) de outros *threads*. Um dos motivos que os *threads* não podem ter no código instruções bloqueantes é devido ao fato que situações de *deadlock* podem ocorrer.

Para permitir a portabilidade de desempenho, o núcleo executivo foi modelado de forma a suportar a implantação de diferentes algoritmos de escalonamento baseados em algoritmos de lista (GRAHAM, 1969) para suportar a execução paralela de programas. A idéia é adicionar ao ambiente a possibilidade de introduzir técnicas de escalonamento pela adaptação do núcleo executivo para responder de forma adequada a diferentes critérios de regulação de carga (tempo de

execução, consumo de memória etc.) conforme as características da aplicação e da arquitetura. No modelo Anahy, a questão clássica da portabilidade de código também foi considerada. Para a implementação dos módulos dependentes de arquitetura optou-se pelo uso de ferramentas de programação que possam ser facilmente encontradas nas mais diferentes configurações de agregados: POSIX threads e MPI/sockets. Essas ferramentas foram selecionadas por possibilitarem a exploração dos dois níveis de paralelismo de um agregado: intra e entre - nodos. Nota-se que este aspecto não envolve somente a linguagem de programação, mas também bibliotecas que permitam manipular efetivamente os recursos de uma arquitetura, como um agregado (CAVALHEIRO; DENNEULIN; ROCH, 1998; CAVALHEIRO, 2001).

4.1 Recursos de programação

O modelo Anahy encontra-se implementado na ferramenta Athreads. A interface de programação de Athreads é baseada no modelo *fork/join* para descrever a concorrência em termos de *threads*. Athreads permite criação e sincronização de *threads* e compartilhamento de um espaço de endereçamento e foi implementada como uma biblioteca para programas em C/C++. Os principais recursos de programação oferecidos são *athread_create* e *athread_join,* para criação e sincronização de *athreads*. Os protótipos destas funções são apresentados na Fig. 4.1.

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

Figura 4.1 - Protótipos dos serviços de criação e sincronização de *threads* em Athreads.

Os parâmetros destas funções possuem a mesma semântica definida pelo padrão POSIX threads: *func* é a função que deve ser executada pelo *thread*; *attr* especifica os atributos do *thread* que devem ser aplicados ao novo *thread*; *th* é um valor que será atualizado para identificar o *thread* criado; *in* é o endereço onde os dados de entrada da função definida por *func* irão ser armazenados. No serviço *athread_join*: *th* indica o *thread* no qual a sincronização irá ser realizada; *res* irá ser

atualizada para apontar à posição na memória compartilhada onde os resultados da função executada pelo *thread th* podem ser encontrados. A exemplo do que ocorre com o uso de ferramentas baseadas em Pthreads, a concorrência de um programa pode ser descrita por um grafo não estruturado de *threads* (Fig. 4.2), uma vez que qualquer *thread* pode sincronizar (realizar uma operação de *join*) sobre qualquer outro *thread* por ele conhecido. No entanto, Athreads extende a especificação original permitindo que *threads* sofram múltiplas sincronizações por *join*.

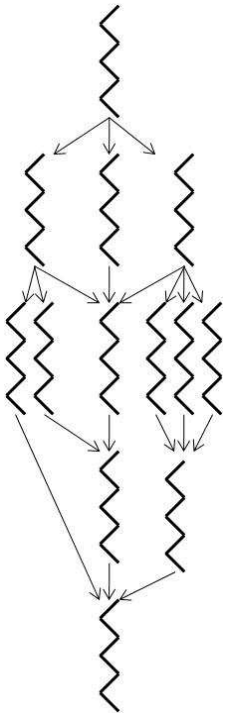

Figura 4.2 - Grafo de dependência de tarefas em Anahy.

Outra extensão é a implementação do esqueleto *split*/*compute*/*merge*. A operação *split* permite criar múltiplos fluxos de execução (réplicas) associados a um mesmo trecho de código. Analogamente, a operação *merge* permite sincronizar o término de todas as réplicas criadas. O *compute*, é de fato, o código do usuário a ser executado. As operações *split* e *merge* são encapsulados pelos serviços *athread_create* e *athread_join*. Atributos do *thread* permitem especificar o número de réplicas a serem criadas (*splitfactor*) e também como o dado de entrada informado no *athread_create* deve ser particionado entre as réplicas e, de forma simétrica, como os dados de retorno das réplicas devem compor o resultado final à ocasião do *athread_join*.

4.2 Escalonamento

A estratégia de escalonamento do modelo Anahy é baseada em lista de tarefas. Na ferramenta Athreads, as tarefas são encapsuladas no contexto de *threads* (fluxo de controle e pilha de execução). Uma camada intermediária entre a interface de programação e o ambiente de execução é responsável por identificar a concorrência em tarefas e criar um grafo acíclico direcionado (DAG) para representar as dependências entre estas tarefas. O grafo é explorado por algoritmos de escalonamento de listas. Neste DAG cada vértice corresponde a uma tarefa a ser executada pelo programa e cada aresta dirigida uma comunicação dos dados entre as tarefas. No grafo existe o chamado *caminho crítico* o qual representa o tempo limite de execução do programa em uma arquitetura paralela (GRAHAM, 1969), sendo este o tempo necessário à execução da maior seqüência de dependências entre tarefas.

As tarefas do DAG são definidas por conjunto de instruções que devem ser executadas de forma seqüencial, recebendo como entrada um conjunto de dados e fornecendo outro conjunto como saída. Uma tarefa está pronta para ser executada no momento que sua dependência estiver satisfeita, ou seja, quando o conjunto de dados de entrada estiver disponível para leitura.

Em um ambiente de execução paralela, o escalonamento é realizado em dois níveis distintos: sistema e aplicativo. Enquanto o escalonamento sistema busca realizar a ocupação dos recursos computacionais de uma arquitetura para realizar a execução de um programa, o escalonamento aplicativo tem como preocupação a distribuição da carga computacional gerada pelo programa sobre os diferentes recursos de execução da arquitetura. Anahy é um ambiente de execução que desenvolve o escalonamento aplicativo (DALL'AGNOL et al., 2003).

O núcleo executivo do modelo Anahy limita o número de atividades concorrentes da aplicação em execução simultânea. Isto significa que, em um dado momento, o número máximo de atividades em execução é limitado em função dos recursos computacionais disponíveis. Este limite é dado pelo número de processadores virtuais, ou PVs, ativos no núcleo. Assim, o programador pode definir

um número de atividades concorrentes que ultrapasse as capacidades (DALL'AGNOL et al., 2003).

No modelo Anahy as políticas de escalonamento devem explorar as características do programa a ser executado e da arquitetura destino sobre a qual a execução é realizada. Desta forma, um algoritmo de escalonamento é composto por dois subsistemas internos: um de manipulação de informação, responsável por observar a carga do sistema, oferecendo subsídios para um segundo subsistema, e outro de decisão, responsável por manipular as tarefas do programa e sua execução sobre a máquina.

A solução para garantir que Anahy seja um ambiente de execução que assegure a portabilidade de desempenho dos programas em execução passa pela definição de um módulo de escalonamento que seja independente da aplicação. Desta forma, alterar o algoritmo de escalonamento, adequando a política de tratamento das tarefas às características da aplicação e da máquina destino, é uma operação que pode ser feita sem a necessidade de modificar o código do programa submetido.

5 Paralelismo no OpenMP

5.1 Análise da concorrência

No OpenMP, o OpenMP Control Flow Graph (OMPCFG) modela o controle de fluxo em programas escritos com OpenMP e a representação de uma árvore (OpenMP *Region Tree*) para modelar a estrutura hierárquica de laços e construções. Estas duas representações servem como base para a análise da concorrência do código (LIN, 2005).

5.1.1 OpenMP Control Flow Graph

O OpenMP Control Flow Graph (OMPCFG) provê a representação de um grafo para modelar o controle de fluxo das instruções de um programa em OpenMP. As instruções de uma subrotina em OpenMP são particionadas em blocos básicos e cada diretiva do OpenMP é colocada em um bloco individual. Cada bloco se transforma em um nodo no OMPCFG. Os nodos que representam os blocos básicos são chamados de nodos básicos, e os nodos representando os blocos diretivos são chamados nodos diretivos. Na Fig. 5.1 são mostrados os nodos diretivos e as correspondentes arestas no OpenMP (ZHANG et al., 2007).

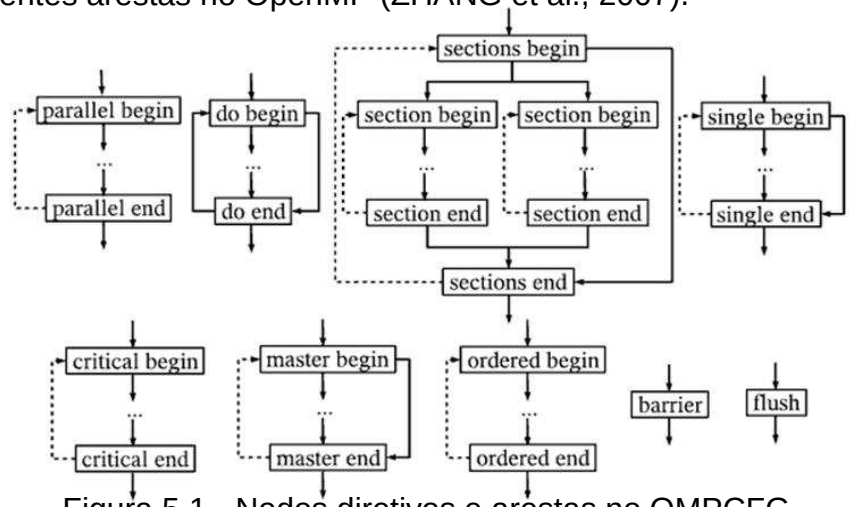

Figura 5.1 - Nodos diretivos e arestas no OMPCFG.
Fonte: LIN, 2005.

Em um OMPCFG, para facilitar a análise do compilador, barreiras implícitas são transformadas em explícitas, e cada construção paralela *work-shared*, como a construção *sections*, é separada em uma construção *work-shared* aninhada na região paralela (BASUMALLIK; EIGENMANN, 2008). Os nodos diretivos do *parallel begin* e *parallel end* são considerados nodos barreiras na região paralela definida por esses dois nodos diretivos. Uma aresta no OMPCFG representa uma possível transferência de um controle de fluxo executado por um *thread*. As arestas entre nodos básicos são criadas de maneira similar que em programas seqüenciais. As arestas entre nodos básicos e nodos diretivos, e as entre nodos diretivos são criadas de acordo com as semânticas de OpenMP.

O código de uma construção de OpenMP forma uma região de entrada/saída simples. Como pode ser visto na próxima seção na Fig. 5.2, para cada construção em OpenMP, uma aresta é criada pelo nodo diretivo inicial para o nodo de entrada da região da construção, e uma aresta é criada do nodo de saída da região para o nodo diretivo final. As arestas para os nodos, ou vindas dos nodos *barrier* e *flush* são criados como se fossem nodos básicos (ZHANG et al., 2007).

5.1.2 OpenMP Region Tree

Nos programas em OpenMP é utilizado a Region Tree para modelar a estrutura hierárquica do laço e a estrutura da construção do OpenMP.

No OMPCFG para cada construção do OpenMP é adicionado uma aresta do nodo diretivo do final da construção para o nodo diretivo inicial da construção. Esta aresta é chamada de aresta de construção e é representada na Fig. 5.1 por uma linha pontilhada. Uma aresta de construção não representa nenhum controle de fluxo, sendo assim pode-se dizer que uma construção forma um ciclo no OMPCFG. Portanto, o algoritmo de detecção de laços para programas seqüenciais pode ser utilizado para encontrar os laços e as regiões de construção de OpenMP no OMPCF (LIN, 2005).

Dessa maneira, devido ao fato das instruções de uma construção formarem uma região de entrada/saída simples, as construções de OpenMP na subrotina então são propriamente aninhadas. Se toda a subrotina fosse tratada como uma

construção raiz de uma árvore, então todas as construções de OpenMP formariam uma árvore. As construções de OpenMP são propriamente aninhadas com laços na subrotina. No momento que é combinada a árvore do laço com a árvore de uma construção de OpenMP é obtido o OpenMP Region Tree. Cada nodo no OpenMP Region Tree representa um laço ou uma construção do OpenMP (LIN, 2005).

5.1.3 Fases na região paralela

As barreiras são freqüentemente utilizadas como métodos de sincronizações em OpenMP. As barreiras podem ser inseridas utilizando a diretiva *barrier,* ou implicitamente no fim de construções compartilhadas ou em construções paralelas (LIN, 2005). No padrão de OpenMP as diretivas *barrier* devem ser encontradas por todos os *threads* no *thread-pool*, e elas devem ser encontradas na mesma ordem por todos os *threads* do *thread-pool*.

A restrição do padrão de OpenMP que impõe no uso de barreiras essencialmente particiona a execução de uma região paralela em um conjunto distinto de fases de execução não sobrepostas. Na Fig. 5.2 e Fig. 5.3 são ilustrados o uso de barreiras. Por exemplo: no caso (a) da Fig. 5.2 as barreiras são colocadas com os nodos não-barreiras em três fases: o nodo N_2, depois com os nodos N_4 e N_5, e por fim com o nodo N_7.

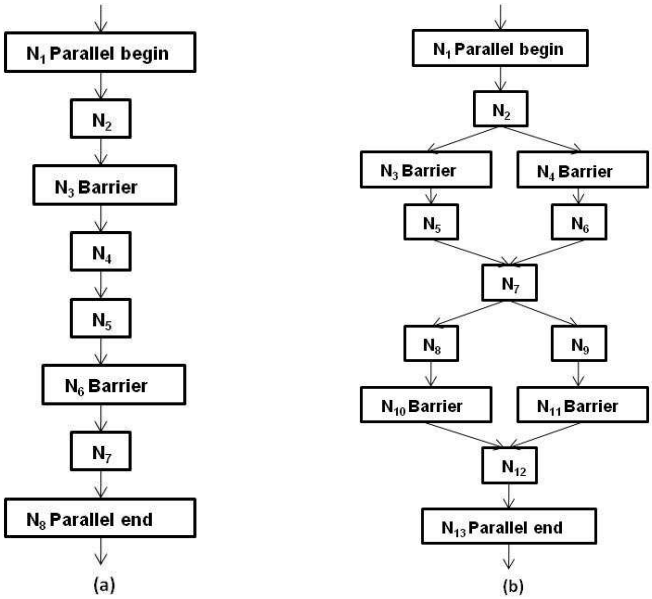

Figura 5.2 - a) Fases simples em uma região paralela. b) Fases quando há saltos.

Fonte: LIN, 2005.

As instruções em N_2 e as instruções em N_4 não serão executadas concorrentemente pelos diferentes *threads thread-pool*. É importante destacar que a imposição que o padrão OpenMP realiza em relação às barreiras não são aplicadas aos *threads* de diferentes *thread-pool*.

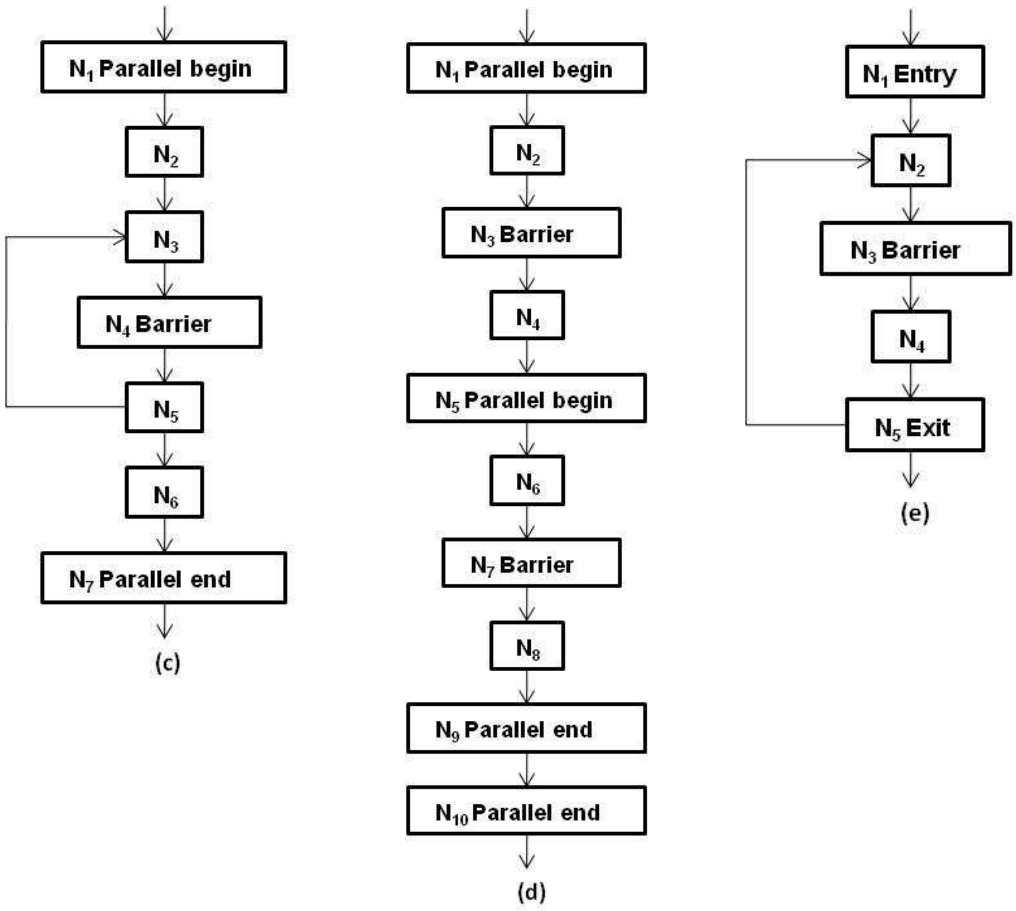

Figura 5.3 - c) Fases em um laço. d) Fases em regiões paralelas aninhadas. e) Fases órfãs.
Fonte: LIN, 2005.

5.2 Análise do escalonamento

A interface de programação de aplicativos do OpenMP suporta regiões paralelas, seções paralelas e laços paralelos. Um laço paralelo, definido com uso das diretivas *parallel* e *for*, é uma das maiores fontes de paralelismo em programas do tipo *fork/join*. Dessa forma, é importante o escalonamento das iterações de laços paralelos nos processadores seja eficiente (BERRENDORF; NIEKEN, 1999). Um laço paralelo pode ser descrito utilizando a construção *omp for* em um programa

C/C++. Na Fig. 5.4 é possível ver a utilização de um laço paralelo escrito em OpenMP em um código C. Neste exemplo, a cláusula *schedule(runtime)* especifica que o escalonador a ser utilizado é aquele indicado pelo usuário na execução em uma variável de ambiente inicializada antes da execução do programa.

```
#pragma omp parallel for private(i,j) schedule(runtime)
 for(i = 0 ; i < 100 ; i++) {
   for(j = 0 ; j < 100 ; j++) {
     a[i][j] = a[i]][j] + b[i][j];
   }
 }
```

Figura 5.4 – Exemplo de um laço paralelo em linguagem C.

O compilador do OpenMP converte esses laços em códigos baseados em *threads* com chamadas à biblioteca de execução do OpenMP para realizar a sincronização e escalonamento. O código gerado pelo compilador OpenMP para a entrada apresentada na Fig. 5.4 encontra-se na Fig. 5.5. Neste código gerado, a chamada para *_ompc_runtime_sched_init* inicializa o escalonamento *runtime* e cada chamada à *_ompc_runtime_sched_next* retorna um *chunk* de iterações para os *threads* *_p_i0*, *_p_i1* e *_p_i2* do pool de execução trabalharem.

```
_ompc_runtime_sched_init(_p_i0, _p_i1, _p_i2);
while(_ompc_runtime_sched_next(&_p_i0, &_p_i1)) {
  for( i = _p_i0 ; i < _p_i1 ; i+ = _p_i2) {
    for( j = 0; j < 100 ; j++) {
      a[i][j] = a[i][j] + b[i][j];
    }
  }
}
```

Figura 5.5 - Uma laço com a cláusula *schedule(runtime)*.

O laço mostrado na Fig. 5.4 foi delineado em uma subrotina pelo compilador Omni. No local do código onde o laço é para ser executado, uma chamada ao escalador do compilador Omni (*_ompc_do_parallel*) é realizada com um ponteiro para a subrotina delineada como um argumento. O sistema de execução cria um *thread-pool* para executar o laço e passa para cada um deles para executar o laço e uma cópia para o ponteiro da função. Cada *thread* que executa o laço faz uma chamada à função *_ompc_runtime_sched_init* que inicializa o escalonador selecionado pelo usuário. Na Fig. 5.5 em cada iteração do laço *while*, a chamada à função *_ompc_runtime_sched_next* retorna o chunk de iterações que a *thread* deve

realizar. Cada *thread* continua executando o laço *while* até *_ompc_runtime_sched_next* retornar zero (ZHANG et al., 2004).

Em relação aos tipos de escalonamento, a cláusula *schedule(type,chunk_size)* é utilizada para a especificação do escalonamento do laço, onde *type* pode ser *dynamic*, *guided*, *static*, ou *runtime*, e *chunk_size* é opcional e deve ser do tipo inteiro positivo especificando o número contínuo de iterações designadas aos *threads*.

No escalonamento *static*, ou seja, quando a cláusula *schedule* for *schedule(static,chunk_size)* as iterações são divididas em *chunks* de tamanho *chunk_size*, e os *chunks* são designados para o grupo dos *threads* segundo o modelo *round-robin* na ordem do número do *thread*.

Quando *chunk_size* não é especificado, o espaço de iteração é dividido em *chunks* que são aproximadamente iguais em tamanho e pelo menos um *chunk* é distribuído para cada *thread*. É importante destacar que neste caso os tamanhos dos *chunks* não são especificados.

No escalonamento *dynamic*, quando *schedule* for *schedule(dynamic,chunk_size)*, as iterações são distribuídas no grupo dos *threads* em *chunks* na medida que os *threads* as requerem. Cada *thread* executa um *chunk* de iterações, e então requisita outro *chunk* até não existirem *chunks* para serem distribuídos. A política de distribuição de *chunks* segue o modelo *first-in first-out*.

Cada *chunk* contém um número de *chunk_size* iterações, exceto para o último *chunk* a ser distribuído que deve ter menos iterações. Quando nenhum *chunk_size* for especificado, por padrão o valor passa a ser um.

No escalonamento *Guided*, quando a cláusula *schedule* for *schedule(guided,chunk_size)*, as iterações são designadas nos grupos dos *threads* em *chunks* de modo quando os *threads* em execução as requerem. Cada *thread* executa um número *chunk* de iterações, logo requisita outro *chunk*, até os *chunks* acabarem.

O tamanho de cada *chunk* é proporcional ao número de iterações restantes dividido pelo número de *threads* no grupo, decrementado por um. Para um

chunk_size de valor K (maior que 1), o tamanho de cada *chunk* é determinado na mesma maneira, com a restrição que os *chunks* não contenham menos que K iterações (exceto para o último *chunk* a ser designado, que deve conter menos que K iterações). Quando nenhum *chunk_size* for especificado o padrão passa a ser um.

Por último, quando a cláusula for *schedule(runtime)*, o escalonamento *runtime* indica que o tipo de escalonamento e os *chunks* são especificados pela variável de ambiente *OMP_SCHEDULE*.

5.2.1 *Overhead* no escalonamento

Em programas de aplicação real programas sendo executados em grandes quantidades de processadores, os acessos a dados e as localidades dos processadores são fatores dominantes no desempenho das aplicações. Porém, o *overhead* dos laços paralelos e a estabilidade da arquitetura são fatores consideráveis no desempenho (BERRENDORF; NIEKEN, 1999).

A análise do overhead nas estratégias de escalonamento foi realizada utilizando a máquina HPC 3500 (BULL, 1999). A Fig. 5.6 reproduz um dos resultados obtidos neste trabalho, mostrando que o escalonamento de um bloco cíclico *scheduling(static,n)* possui o mesmo custo que o mesmo bloco escalonado utilizando *chunks* grandes, porém o custo aumenta rapidamente na medida em que o tamanho do *chunk* diminui. Para *chunks* de tamanho pequeno o *overhead* é aproximadamente linear nos números de *chunks* por *thread*. O escalonamento *dynamic* possui um compartamento similar, mas é cinco vezes mais custoso que o bloco cíclico. O escalonamento *guided* tem um custo similar ao *dynamic* com grandes *chunks*, e o custo aumenta lentamente à medida que o tamanho do *chunk* dimuniu. Isto é esperado, como o tamanho do *chunk* é o menor tamanho de um *chunk*, a maioria das iterações é executada em *chunks* grandes (BULL, 1999).

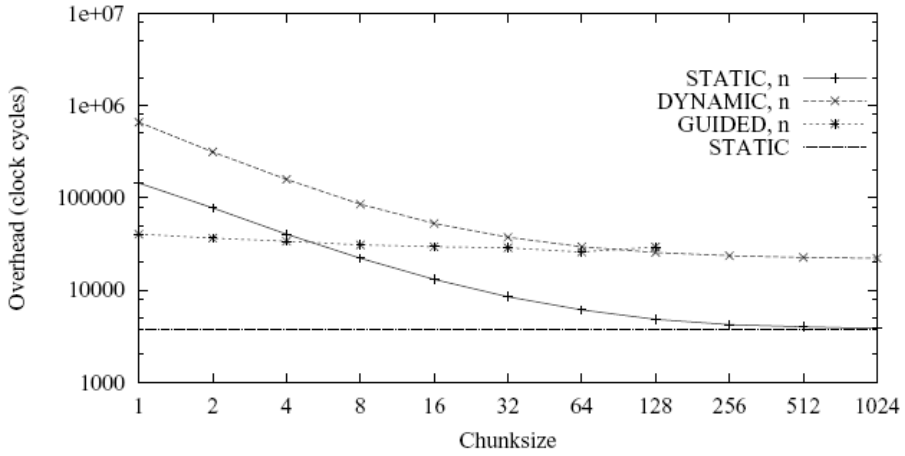

Figura 5.6 - *Overhead* no escalonamento na SUN HPC 3500.
Fonte: BULL, 1999.

Outros testes foram realizados na máquina Origin 2000 (Fig. 5.7) no qual o *overhead* do escalonamento de um bloco cíclico não converge para o bloco escalonado tanto quanto o número de *chunks* por *thread* diminui. O custo do escalonamento *dynamic* aumenta rapidamente à medida que aumenta o número de *chunks*, então com um tamanho de *chunk* 1, o custo está relacionado à magnitude mais do que na máquina HPC 3500. O custo do escalonamento *guided* tambem aumenta notavelmente tanto quanto o tamanho do *chunk* diminui (BULL, 1999).

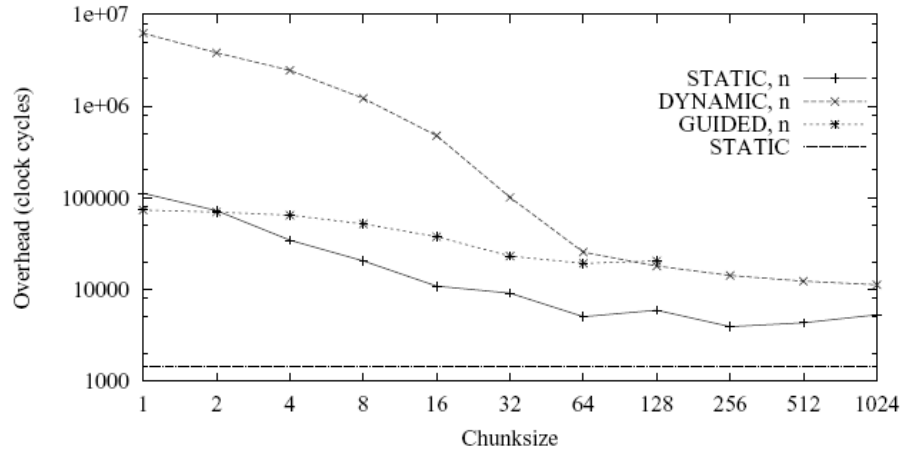

Figura 5.7 - *Overhead* no escalonamento na SGI Origin 2000.
Fonte: BULL, 1999.

No servidor Alpha (Fig. 5.8), o escalonamento do bloco cíclico é cerca de 30 vezes mais custoso que o bloco escalonado com tamanhos de *chunks* grandes, e suavemente mais caro que o escalonamento *guided*, entretanto para *chunks*

pequenos o overhead é comparável com os outros dois sistemas citados acima (BULL, 1999).

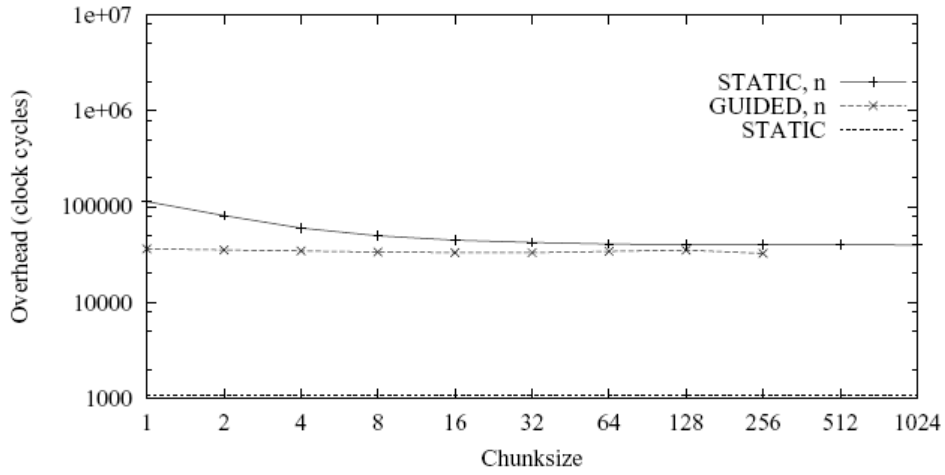

Figura 5.8 - *Overhead* no escalonamento na Compaq Alpha Server.
Fonte: BULL, 1999.

A principal observação de esses resultados é que o escalonamento com *chunks* de tamanho pequeno introduz um substancioso *overhead* nos três sistemas. Provavelmente, isto resulta porque cada *chunk* realiza uma chamada de função, desde que a implementação seja baseada na criação de uma rotina contendo o corpo do laço e passando ela para a rotina escalonadora da biblioteca do OpenMP. Para escalonamentos de bloco cíclicos, isto pode ser evitado tendo o compilador a responsabilidade de gerar as chamadas dos laços requeridos, ao invés de ser dependente da biblioteca de execução do OpenMP para realizá-la (BULL, 1999).

6 ApenMP

O protótipo de ApenMP consiste em uma ferramenta de pré-processamento. Esta ferramenta aplica técnicas de compilação para traduzir diretivas e cláusulas de OpenMP em código Athread. O código gerado pelo pré-processador possui o mesmo valor semântico do código fonte apresentado, respeitando as dependências entre as instruções expressas no programa do usuário. Este capítulo apresenta a implementação do pré-processador ApenMP, destacando a documentação das suas etapas de análise léxica,sintática e geração de código.

6.1 Construção do Pré-processador

ApenMP é um pré-processador responsável pela análise de código OpenMP e geração de código Athreads. O processo de geração de código executável utilizando esta ferramenta é apresentado na Fig. 6.1. O primeiro protótipo de ApenMP foi desenvolvido na linguagem C com o compilador GCC 4.2.4 sobre ambiente GNU-Linux.

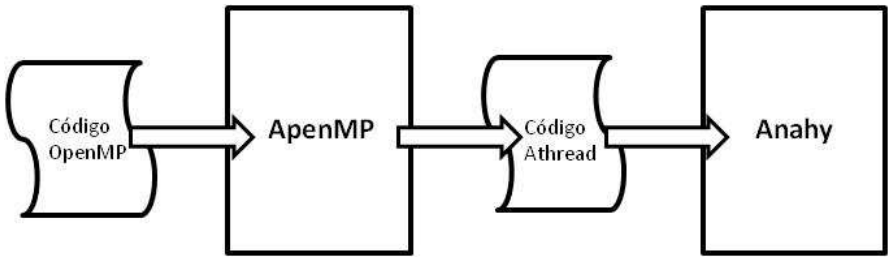

Figura 6.1 - Processo de compilação com ApenMP.

Para a construção de ApenMP, foram utilizadas duas ferramentas que auxiliam no processo de criação e desenvolvimento de compiladores: **Flex** e **Bison**[1].

[1] Estas ferramentas são versões livres do analisador léxico Lex e do analisador sintático YACC.

Elas servem de auxílio na escrita de programas que promovem transformações sobre entradas estruturadas. Também podem tornar-se válidas para outras aplicações, tais como detecção de padrões em arquivos de dados, linguagens de comandos, entre outras. Flex e Bison permitem um rápido desenvolvimento de protótipos e uma manutenção simples do software, em comparação com ferramentas alternativas.

6.1.1 Flex e Bison

O programa Flex (LEVINE; MASON; BROWN, 1995), produz automaticamente um analisador léxico a partir de especificações de expressões regulares, passíveis de representação por meio de autômatos finitos. As expressões regulares reconhecidas produzem *tokens* como saída (AHO; SETHI; ULLMAN, 1995). Na implementação do analisador léxico de ApenMP são retornados *tokens* para construtores da linguagem C e de OpenMP. Estes *tokens* servem como entrada para execução do analisador sintático gerado com o Bison.

O Bison (LEVINE; MASON; BROWN, 1995) é uma ferramenta para geração de analisadores sintáticos. Enquanto o Flex gera uma função *yylex()*, que retorna o identificador de um item léxico reconhecido (*token*), o Bison gera a função *yyparse()*, que analisa os itens léxicos e decide se eles formam ou não uma sentença válida. As regras da análise sintática devem ser fornecidas ao Bison para que este gere o analisador sintático. Uma vez geradas as funções *yylex()* e *yyparse()*, estas devem ser compiladas e ligadas em um executável único. Na implementação realizada também foi incluído o gerador de código Athreads, ativado, conforme a necessidade, no reconhecimento de itens léxicos.

A Fig. 6.2 ilustra o processo de criação de ApenMP com o auxílio das duas ferramentas, Flex e Bison, juntamente com as bibliotecas de apoio e do compilador gcc em questão. Os arquivos *apenmp.l* e *apenmp.y* representam, respectivamente, as regras para as análises léxico e sintático.

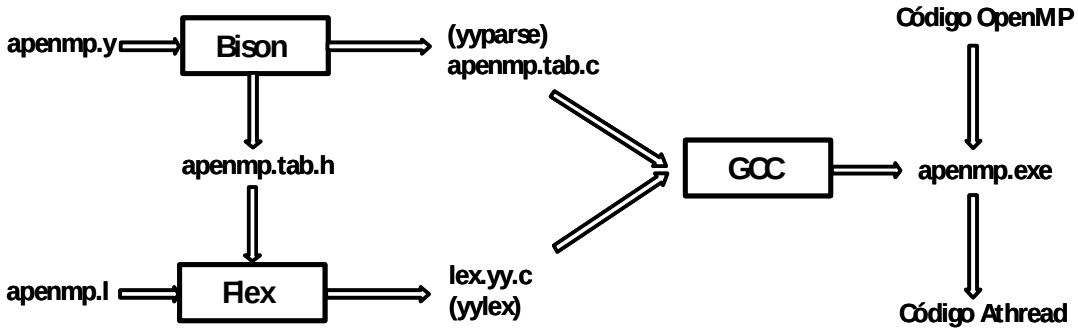

Figura 6.2 - Esquema da geração de ApenMP.

Na Fig. 6.2 é possível observar a criação de arquivos auxiliares manipulados no processo de compilação, como o *apenmp.tab.h*, o qual estabelece a dependência simbólica entre o analisador léxico e o sintático. Também é válido citar que o arquivo *lex.yy.c* contém a função *yylex()* e o arquivo *apenmp.tab.c* contém a função *yyparse()*.

Em relação a implementação do pré-processador, as ferramentas Flex e Bison foram utilizadas na construção por serem software livre e atenderem aos propósitos do trabalho de análise e geração de código. Além das vantagens citadas, o desempenho destas ferramentas é alto, pois a utilização de um analisador léxico desenvolvido utilizando o Flex é relativamente simples de implementar e quase sempre mais rápido do que um analisador léxico escrito diretamente em C (LEVINE; MASON; BROWN, 1995).

Por outro lado, um analisador sintático desenvolvido utilizando Bison é geralmente mais lento do que um analisador sintático escrito diretamente em C, porém o ganho no desenvolvimento e manutenção do programa é enorme (LEVINE; MASON; BROWN, 1995).

6.1.2 Geração de código

No protótipo de ApenMP, uma etapa fundamental é a geração de código Athreads. O código é gerado com apoio de um conjunto de serviços que utilizam os resultados do pré-processamento (identificação de diretivas, cláusulas e dados manipulados) para produzir o código Athreads correspondente. O código gerado

ocupa a porção de código original contendo os *pragmas* com as diretivas *parallel* e *for*. Estes serviços são descritos na seqüência em ordem alfabética pelo nome da função.

- **void atualizaVar(struct tabela *vetor, int aux, int varcount);**

  Aplicação: Quando do uso das cláusulas *lastprivate* e *reduction*.

  Funcionalidade: Garante que os dados correspondentes ao indicado nas cláusulas *lastprivate* e *reduction* em cada região paralela sejam unificados no *thread master*.

- **void armazenaClausulaVar(int clausula, struct tabela *vetor, char *str, int varcount);**

  Aplicação: Utilizada no momento que as cláusulas são declaradas para a região paralela.

  Funcionalidade: Armazenar o tipo de cláusula da variável.

- **void armazenaPragmaVar(struct tabela *vetor, char *str, int count, int varcount);**

  Aplicação: A toda variável manipulada em um *pragma*.

  Funcionalidade: Cria uma associação entre uma variável e o *pragma* ao qual ela esta sendo utilizada; pode ocorrer que uma variável seja utilizada por mais de um *pragma*.

- **void armazenaTamVetor(struct tabela *vetor, int indice, char *str);**

  Aplicação: No momento de declaração de um vetor.

  Funcionalidade: Armazenar o tamanho de um vetor na tabela de variáveis.

- **void chamaBiblioteca();**

  Aplicação: É utilizada no início da geração de código.

  Funcionalidade: Criar a chamada *athread.h* de Anahy.

- **void corpoThread(struct omp *amp, struct tabela *var, int count, int varcount);**

  Aplicação: É utilizada no final da geração de código.

  Funcionalidade: Esta função tem a finalidade de criar os corpos dos *threads* do programa.

- **void criaThread(struct omp *vetor, struct tabela *vet, int varcount, int count, int flag);**

  Aplicação: Quando uma região paralela é encontrada.

  Funcionalidade: Esta função tem a finalidade de gerar as funções *athread_create* e *athread_join*.

- **void declaraPrototipo(int flag);**

  Aplicação: Utilizada no início da geração de código.

  Funcionalidade: Esta função tem o objetivo de gerar no código Athread a declaração dos protótipos das funções e dos *threads*.

- **void funcoesAmp();**

  Aplicação: Esta função é utilizada no final da geração de código.

  Funcionalidade: Criar as funções *ampAdiciona*, *ampAtualiza* e *ampVetor*. Estas funções serão geradas no código Athread.

- **void imprimeTabVar(struct tabela *vetor, int varcount);**

  Funcionalidade: Imprimir na tela a tabela de variáveis.

- **void indicaVetor(struct tabela *vetor, int indice);**

  Aplicação: Quando um vetor é declarado.

  Funcionalidade: Indicar na tabela de variáveis se uma determinada variável é um vetor.

- **void inicializaTabVar(struct tabela *vetor);**

  Aplicação: Esta função é executada antes do *parser* ser chamado.

  Funcionalidade: Inicializar a tabela de variáveis.

- **void initAnahy();**

  Aplicação: Utilizada antes da criação dos *threads*.

  Funcionalidade: Criar a chamada *aInit(argc,argv)* de Anahy.

- **void insereAtributo(struct tabela *vetor, char *string, int indice);**

  Aplicação: Quando uma variável é declarada com algum valor.

  Funcionalidade: Inserir atributos de uma variável na tabela de variáveis.

- **void insereTabVar(struct tabela *vetor, char *string, int aux, int *indice, int *varcount);**

  Aplicação: Esta função é utilizada no momento de declaração das variáveis do programa.

  Funcionalidade: Armazenar uma variável na tabela de variáveis.

- **void preparaDados(struct tabela *vetor, int varcount, int count);**

  Aplicação: Quando os dados são *firstprivate* e *reduction*.

  Funcionalidade: Construir de forma correta as variáveis que são passadas para os *threads*.

- **void preparaThreadVar(struct tabela *vetor, int varcount, int numThreads);**

  Aplicação: Utilizada logo após a declaração das variáveis.

  Funcionalidade: Nesta função serão gerados no código Athread as declarações dos *threads*, o número de *threads* e preparar as variáveis da função *main*.

- **void recuperaVarThread(struct tabela *var, int i, int varcount);**

  Aplicação: Quando os dados são *firstprivate* e *reduction*.

  Funcionalidade: Recuperar dentro dos *threads* os dados que foram passados para os *threads*.

- **int verificaDeclaracao(struct tabela *vetor, char *string, int varcount);**

  Aplicação: Utilizada pelas variáveis dentro dos *threads*.

  Funcionalidade: Verificar durante a geração das variáveis se uma determinada variável já foi declarada.

6.2 Funcionamento do Pré-processador

Durante a análise do arquivo de entrada, diversas ações devem ser tomadas para a geração do código final. Na geração do código em Athreads, as ações semânticas devem ser executadas dependendo das regras sintáticas do código. A seguir, estas ações estão descritas:

- Todo código que não expresse paralelismo não é alterado.
- Todo código que expresse paralelismo com o uso das diretivas consideradas no protótipo (*parallel* e *for*) em *#pragmas* deve ser traduzido em código correspondente Athreads.
- *Pragmas* contendo a diretiva *parallel* são traduzidos para chamadas à função *athread_create* e todo código dentro desta diretiva pertence ao corpo do *thread*.
- O código fonte explicitado em *pragmas* contendo a diretiva *for* é traduzido em corpo de funções lançadas com chamadas à *athread_create*. O laço orginal é particionado em *chunks* entre os *threads* lançados.
- Toda variável que não tenha sido especificada com alguma cláusula será tratada como *private* dentro do *thread*. Esta decisão foi escolhida por respeitar o modelo Anahy.

- Os dados *private* são declarados dentro dos *threads*, pois eles serão privados para cada *thread*.

- Os dados tipos *firstprivate* serão armazenados em uma lista encadeada e passados para os *threads*. Desse modo os *threads* serão inicializados com o valor deles no *thread master*.

- Os dados *lastprivate* serão retornados para o *thread master* e o valor do dado manipulado pelo último *thread* será recuperado no momento da sincronização com o *thread master*.

- Os dados *reduction* são passados para os *threads* e manipulados dentro dos *threads*. No momento da sincronização estes valores são combinados de acordo com o resultado que cada *thread* produziu.

Outro aspecto a ser considerado é o tratamento padrão aplicado para as variáveis declaradas no *thread master*. Em OpenMP, caso nenhuma cláusula seja especificada para uma determinada variável declarada no *thread master*, ela é tratada como *shared* nas regiões paralelas. A decisão desta implementação pode ser explicada pelo melhor desempenho de execução a ser obtido em OpenMP, pois as omissões do usuário não geram nenhum sobre-custo de execução. Já o modelo Anahy não contempla este tipo de compartilhamento de dado, embora ApenMP suporta o uso da cláusula *shared* entre as regiões paralelas para manter a compatibilidade com OpenMP. Em ApenMP, variáveis *shared* geram custo adicional de execução, pois requerem manipulação com tratamento de seção crítica, em regime de exclusão mútua. Para evitar que a simples omissão do programador incorra em custos adicionais de processamento, ApenMP considera a cláusula *private* (e não *shared*) como a *default* para os dados.

6.3 Código gerado

No código gerado é importante destacar que as bibliotecas, funções e variáveis, descritas a seguir, serão criadas para qualquer regra sintática válida de OpenMP. Porém, a manipulação dos dados incorpora técnicas mais elaboradas, pois o código em relação às variáveis em Athread irá depender do tipo de cláusula do dado em OpenMP. Abaixo o funcionamento e o protótipo delas são descritos:

- **athread.h**

  Funcionalidade: Este é o cabeçalho da biblioteca utilizada para realizar a invocação dos serviços de Athreads.

- **void amp1 (void *dados);**

  Funcionalidade: Este protótipo corresponde ao cabeçalho para as funções a serem geradas com o corpo dos *thread* que corresponderão as regiões paralelas. É importante destacar que para cada corpo do *thread* encontrado será incrementado o número que está no fim do nome do corpo do *thread*. Exemplo: amp1, amp2, amp3, etc.

- **struct valores{**
  void *valor;
  struct valores *prox.
  };

  Funcionalidade: Nesta estrutura são armazenadas as variáveis declaradas como *firstprivate* e *reduction*. Elas são passadas como parâmetros de entrada para o *thread* correspondente.

- **int numThreads;**

  Funcionalidade: Esta variável contém o número de *threads* da aplicação.

- **athread_t ampth[numThreads];**

  Funcionalidade: Neste vetor está contido o número de *threads* da aplicação.

- **int ampAux;**

  Funcionalidade: Esta variável é auxiliar para a criação de um número determinado de *threads* definida pelo programador.

- **void *retorno;**

  Funcionalidade: Este ponteiro do tipo *void* conterá o endereço dos valores retornado pelos *threads*.

- **struct valores *head;**

  Funcionalidade: Neste ponteiro está contido o endereço onde se encontram as variáveis passadas para os *threads*.

- **struct valores *temp;**

  Funcionalidade: Ponteiro utilizado para armazenar no *thread master* o endereço das variáveis retornadas dos *threads*.

- **struct valores *res;**

  Funcionalidade: Ponteiro utilizado para armazenar dentro dos *threads* as variáveis que são retornadas para o *thread master*.

- **struct valores *aux;**

  Funcionalidade: Ponteiro utilizado para recuperar dentro dos *threads* as variáveis vindas do *thread master*.

- **void ampAdiciona(struct valores **listas, void *fonte,size_t tam);**

  Funcionalidade: Esta função é utilizada para criar a lista encadeada das variáveis que serão passadas para os *threads*. O primeiro parâmetro receberá o endereço da primeira variável da lista encadeada. O ponteiro *fonte* receberá o *cast* da variável fonte. No último parâmetro encontra-se o tamanho do tipo da variável *fonte*.

- **void ampAtualiza(&head);**

  Funcionalidade: Este ponteiro percorre a lista encadeada das variáveis *firstprivate* e *reduction*.

- **aInit(argc,argv);**

  Funcionalidade: Inicializador das funções de Anahy.

- **athread_create(&ampth[ampAux],NULL,amp1,head)**

  Funcionalidade: Lançamento dos *threads*. A operação desta função está descrita no Capítulo 5.

- **athread_join(ampth[ampAux],&head);**

  Funcionalidade: Sincronização dos *threads*. Assim como a função anterior, esta se encontra descrita no Capítulo 5.

- **aTerminate();**

  Funcionalidade: Função para finalizar os serviços de Athreads.

6.4 Exemplos

Nesta seção são mostrados exemplos de códigos em OpenMP e Athread. Por motivos de espaço não serão mostradas as diversas combinações que podem ocorrer entre as cláusulas e as diretivas. O código fonte encontra-se na parte esquerda dos exemplos apresentados e o código gerado à direita. Para facilitar a leitura, os *pragmas* tratados no pré-processador encontram-se em negrito e o código gerado pelo ApenMP encontra-se em itálico.

- **#pragma omp parallel private(lista_de_variáveis)**

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[]){
   int total, y, x;
   #pragma omp parallel
private(total,y,x)
     if( y > x)
        total = y + x;
     else
        total = x – y;
   return 0;
}
```

```
#include <stdio.h>
#include “athread.h”

void amp1(void *dados);

struct valores{
    void *valor;
    struct valores *prox;
};

int main(int argc, char *argv[]){
   int *total, *y, *x;
   int numThreads;
   int ampAux;
   athread_t ampth[numThreads];
   struct valores *temp, *head;
   void *retorno;
   total = malloc(sizeof(int));
   y = malloc(sizeof(int));
   x = malloc(sizeof(int));
   aInit(&argc,&argv);
   for(ampAux=0;ampAux<numThreads;ampAux++)
   athread_create(&ampth[ampAux],NULL,amp1,
head);
   for(ampAux=0;ampAux<numThreads;ampAux++)
   athread_join(ampth[ampAux],&retorno);
   aTerminate();
   return 0;
}

void *amp1(void *dados){
   struct valores *aux, *res;
   int total, y, x;
   if(y > x)
      total = y + x;
   else
      total = x – y;
   return (void *) res;
}

/* Função ampAdiciona
   Função ampAtualiza
   Função ampCombinaRes
*/
```

Figura 6.3 - Código utilizando a diretiva *parallel* e cláusula *private*.

Como podem ser observado na Fig. 6.3, as variáveis ***total***, ***y*** e ***x*** foram instanciadas locais nos *threads*, sendo privadas para cada *thread*.

- **#pragma omp parallel firstprivate(lista_de_variáveis)**

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[]){
   int total, y, x;
   x = 100, y = 200;
   #pragma omp parallel
firstprivate(y,x)
      if(y>x)
         total = y + x;
      else
         total = x – y;
   return 0;
}
```

```
#include <stdio.h>
#include “athread.h”

void amp1(void *dados);

struct valores{
    void *valor;
    struct valores *prox;
};

int main(int argc, char *argv[]){
   int *total, *y, *x;
   int numThreads;
   int ampAux;
   athread_t ampth[numThreads];
   struct valrores *temp, *head;
   void *retorno;
   total = malloc(sizeof(int));
   y = malloc(sizeof(int));
   x = malloc(sizeof(int));
   *x = 100;
   *y = 200;
   aInit(&argc,&argv);
   ampAdiciona(&head,(void*)y,sizeof(int));
   ampAdiciona(&head,(void*)x,sizeof(int));
   for(ampAux=0;ampAux<numThreads;ampAux++)
   athread_create(&ampth[ampAux],NULL,amp1,
head);
   for(ampAux=0;ampAux<numThreads;ampAux++)
   athread_join(ampth[ampAux],&retorno);
   aTerminate();
   return 0;
}

void *amp1(void *dados){
   struct valores *aux, *res;
   aux = (struct valores *) dados;
   int total, y, x;
   y = (int) aux->valor;
   ampAtualiza(&aux);
   x = aux->valor;
   if(y > x)
      total = y + x;
   else
      total = x – y;
   return (void *) res;
}

/* Função ampAdiciona
   Função ampAtualiza
   Função ampCombinaRes
*/
```

Figura 6.4 - Código utilizando a diretiva *parallel* e cláusula *firstprivate*.

Neste exemplo na Fig. 6.4, as variáveis ***x*** e ***y*** são preparadas e passadas no último parâmetro de *athread_create* para os *threads*. Dentro dos *threads* elas são recuperadas e são passadas para as variáveis locais.

- **#pragma omp parallel reduction(lista_de_variáveis)**

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[]){
   int total, y, x;
   x = 100, y = 200 ;
   #pragma omp parallel
reduction(total)
      if(y > x)
         total = total + 1;
      else
         total = total – 1;
   return 0;
}
```

```
#include <stdio.h>
#include “athread.h”

void amp1(void *dados);

struct valores{
    void *valor;
    struct valores *prox;
};

int main(int argc, char *argv[]){
   int *total, *y, *x;
   int numThreads;
   int ampAux;
   athread_t ampth[numThreads];
   struct valrores *temp, *head;
   void *retorno;
   total = malloc(sizeof(int));
   y = malloc(sizeof(int));
   x = malloc(sizeof(int));
   *x = 100;
   *y = 200;
   aInit(&argc,&argv);
   ampAdiciona(&head,(void*)total,sizeof(in
t));
   for(ampAux=0;ampAux<numThreads;ampAux++)
   athread_create(&ampth[ampAux],NULL,amp1,
head);
   for(ampAux=0;ampAux<numThreads;ampAux++)
   athread_join(ampth[ampAux],&retorno);
   ampCombinaRes(&retorno);
   aTerminate();
   return 0;
}

void *amp1(void *dados){
   struct valores *aux, *res;
   aux = (struct valores *) dados;
   int total, y, x;
   total = (int) aux->valor;
   if(y > x)
      total = total + 1;
   else
      total = total – 1;
   ampAdiciona(&res,(void*)total,sizeof(int
));
   return (void *) res;
}

/* Função ampAdiciona
   Função ampAtualiza
   Função ampCombinaRes
*/
```

Figura 6.5 - Código utilizando a diretiva *parallel* e cláusula *reduction*.

Neste exemplo na Fig. 6.5, a variável ***total*** é preparada e passada para os *threads*. Dentro dos *threads* ela é computada e na sincronização com o *thread master* os resultados desta variável são combinados.

- **#pragma omp for private(lista_de_variáveis)**

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[]){
   int i, total, v[100];
   total = 100;
   #pragma omp for private(total)
      for(i=0;i<100;i++){
         v[i] = i;
         total = i;
      }
   return 0;
}
```

```
#include <stdio.h>
#include “athread.h”

void amp1(void *dados);

struct valores{
    void *valor;
    struct valores *prox;
};

int main(int argc, char *argv[]){
   int *i, *total, *v;
   int numThreads;
   int ampAux;
   athread_t ampth[numThreads];
   struct valrores *temp, *head;
   void *retorno;
   i = malloc(sizeof(int));
   total = malloc(sizeof(int));
   v = malloc(sizeof(int)*100);
   *total = 100;
   aInit(&argc,&argv);
   for(ampAux=0;ampAux<numThreads;ampAux++)
   athread_create(&ampth[ampAux],NULL,amp1,
head);
   for(ampAux=0;ampAux<numThreads;ampAux++)
   athread_join(ampth[ampAux],&retorno);
   aTerminate();
   return 0;
}

void *amp1(void *dados){
   struct valores *aux, *res;
   int i, total, *v;
   v = malloc(sizeof(int)*100);
   for(i=0;i<100;i++){
        v[i] = i;
        total = i;
   }
   return (void *) res;
}

/* Função ampAdiciona
   Função ampAtualiza
   Função ampCombinaRes
*/
```

Figura 6.6 - Código utilizando a diretiva *for* e cláusula *private**.***

Neste exemplo na Fig. 6.6, a variável ***total*** é declarada dentro do *thread*, pois ela é privada para todos os *threads*.

- **#pragma omp for firstprivate(lista_de_variáveis)**

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[]){
   int i, total, v[100];
   total = 100;
   #pragma omp for
firstprivate(total)
      for(i=0;i<100;i++){
         v[i] = i;
         total = i;
      }
   return 0;
}
```

```
#include <stdio.h>
#include “athread.h”

void amp1(void *dados);

struct valores{
    void *valor;
    struct valores *prox;
};

int main(int argc, char *argv[]){
   int *i, *total, *v;
   int numThreads;
   int ampAux;
   athread_t ampth[numThreads];
   struct valrores *temp, *head;
   void *retorno;
   i = malloc(sizeof(int));
   total = malloc(sizeof(int));
   v = malloc(sizeof(int)*100);
   *total = 100;
   aInit(&argc,&argv);
   ampAdiciona(&head,(void*)total,sizeof(in
t);
   for(ampAux=0;ampAux<numThreads;ampAux++)
   athread_create(&ampth[ampAux],NULL,amp1,
head);
   for(ampAux=0;ampAux<numThreads;ampAux++)
   athread_join(ampth[ampAux],&retorno);
   aTerminate();
   return 0;
}

void *amp1(void *dados){
   struct valores *aux, *res;
   int i, total, *v;
   v = malloc(sizeof(int)*100);
   aux = (struct valores *) dados;
   total = (int) aux->valor;
   for(i=0;i<100;i++){
        v[i] = i;
        total = i;
   }
   return (void *) res;
}

/* Função ampAdiciona
   Função ampAtualiza
   Função ampCombinaRes
*/
```

Figura 6.7 - Código utilizando a diretiva *for* e cláusula *firstprivate*.

Neste exemplo, o conteúdo da variável ***total*** é passado para dentro dos *threads* (Fig. 6.7).

- **#pragma omp for lastprivate(lista_de_variáveis)**

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char
*argv[]){
   int i, total, v[100];
   total = 100;
   #pragma omp for
lastprivate(total)
      for(i=0;i<100;i++){
         v[i] = i;
         total = i;
      }
   return 0;
}
```

```
#include <stdio.h>
#include “athread.h”

void amp1(void *dados);

struct valores{
    void *valor;
    struct valores *prox;
};

int main(int argc, char *argv[]){
   int *i, *total, *v;
   int numThreads;
   int ampAux;
   athread_t ampth[numThreads];
   struct valrores *temp, *head;
   void *retorno;
   i = malloc(sizeof(int));
   total = malloc(sizeof(int));
   v = malloc(sizeof(int)*100);
   *total = 100;
   aInit(&argc,&argv);
   for(ampAux=0;ampAux<numThreads;ampAux++)
   athread_create(&ampth[ampAux],NULL,amp1,
head);
   for(ampAux=0;ampAux<numThreads;ampAux++)
   athread_join(ampth[ampAux],&retorno);
   temp = (struct valores *) retorno;
   *total = temp->valor;
   aTerminate();
   return 0;
}

void *amp1(void *dados){
   struct valores *aux, *res;
   int i, total, *v;
   v = malloc(sizeof(int)*100);
   for(i=0;i<100;i++){
         v[i] = i;
         total = i;
   }
   ampAdiciona(&res,(void*)total,sizeof(int
));
   return (void *) res;
}

/* Função ampAdiciona
   Função ampAtualiza
   Função ampCombinaRes
*/
```

Figura 6.8 - Código utilizando a diretiva *for* e cláusula *lastprivate*.

Neste exemplo, o conteúdo da variável ***total*** do último *thread* que executa o *join* é passado para o *thread master* (Fig. 6.8).

- **#pragma omp for reduction(lista_de_variáveis)**

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[]){
   int i, total, v[100];
   total = 100;
   #pragma omp for
reduction(total)
      for(i=0;i<100;i++){
         v[i] = i;
         total = i;
      }
   return 0;
}
```

```
#include <stdio.h>
#include “athread.h”

void amp1(void *dados);

struct valores{
    void *valor;
    struct valores *prox;
};

int main(int argc, char *argv[]){
   int *i, *total, *v;
   int numThreads;
   int ampAux;
   athread_t ampth[numThreads];
   struct valrores *temp, *head;
   void *retorno;
   i = malloc(sizeof(int));
   total = malloc(sizeof(int));
   v = malloc(sizeof(int)*100);
   *total = 100;
   aInit(&argc,&argv);
   ampAdiciona(&head,(void*)total,sizeof(in
t);
   for(ampAux=0;ampAux<numThreads;ampAux++)
   athread_create(&ampth[ampAux],NULL,amp1,
head);
   for(ampAux=0;ampAux<numThreads;ampAux++)
   athread_join(ampth[ampAux],&retorno);
   ampCombinaRes(&retorno);
   aTerminate();
   return 0;
}

void *amp1(void *dados){
   struct valores *aux, *res;
   int i, total, *v;
   v = malloc(sizeof(int)*100);
   for(i=0;i<100;i++){
         v[i] = i;
         total = i;
   }
   ampAdiciona(&res,(void*)total,sizeof(int
));
   return (void *) res;
}

/* Função ampAdiciona
   Função ampAtualiza
   Função ampCombinaRes
*/
```

Figura 6.9 - Código utilizando a diretiva *for* e cláusula *reduction*.

Neste exemplo, o conteúdo da variável ***total*** dos *threads* é combinado no momento da sincronização (Fig. 6.9).

7 Resultados de desempenho

Para a obtenção dos resultados de desempenho foram realizados testes com os algoritmos descritos no capítulo anterior. Estes testes foram aplicados utilizando um computador com:

- Processador Intel Core 2 Duo 1,86 GHz com 4MB Cache, memória RAM de 2 GB, disco rígido de 250GB.
- Sistema Operacional GNU-Linux, kernel 2.6.24.
- Compilador GCC 4.2.4, pois a última versão deste compilador provê suporte total à OpenMP, alem de já fornecer suporte à Pthreads e Athreads (Anahy).

7.1 Metodologia utilizada

A metodologia de avaliação de desempenho contabiliza o tempo de execução de um programa em segundos. Este tempo é tomado no momento do começo da região paralela (criação dos *threads*) até o fim desta região (sincronização dos *threads*). É importante destacar que o tempo de execução das regiões seqüenciais é desconsiderado, pois não altera o desempenho das tarefas. Os tempos apresentados representam uma média de, pelo menos, 15 execuções, sendo apresentados os desvios padrões observados. Os testes foram realizados utilizando 2, 4 e 8 *threads* no pool de execução de Athreads (Processadores Virtuais - PVs).

Os códigos utilizados para avaliação de desempenho são os apresentados como exemplo na seção anterior. Foram escolhidas duas abordagens de paralelismo, tarefas e dados, para avaliar o comportamento de execução final. O motivo desta escolha é permitir a avaliação do desempenho de Anahy em relação à OpenMP e Pthreads nas duas abordagens citadas.

7.2 Resultados

Os resultados apresentados nesta seção correspondem à execução dos programas indicados, considerando que:

- Os tempos associados à OpenMP correspondem ao código compilado com o compilador GCC 4.2.4 com –fopenmp ativada;
- Os tempos associados à Athreads correspondem ao código Athreads gerado com o pré-processador ApenMP desenvolvido;
- Os tempos associados à Pthreads correspondem ao código gerado com ApenMP, sendo substituídas, manualmente, as invocações aos serviços Athreads por invocações aos serviços equivalentes de Pthreads nativo.

7.2.1 Paralelismo de tarefas

Neste experimento foi utilizado o algoritmo da Figura 6.3 do Capítulo 6. Para este tipo de paralelismo foi utilizada a diretiva ***parallel***, pois ela permite que uma seqüência de instruções seja paralelizada. Deste modo, foi utilizado o algoritmo em OpenMP utilizando a diretiva *parallel* do capítulo anterior e a respectiva tradução em Athreads através de ApenMP e Pthreads. Neste código, a diretiva *parallel* lança a execução de tantas regiões paralelas quanto for o número de *threads* de suporte à execução.

Na Tab. 7.1 e na Fig. 7.1 é apresentada a média e o desvio padrão em segundos do tempo de execução dos algoritmos através do número de *threads*.

Tabela 7.1 - Tempo de execução dos algoritmos.

	OpenMP			**ApenMP**			**ApenMP ⟶ Pthreads**		
Threads	**2**	**4**	**8**	**2**	**4**	**8**	**2**	**4**	**8**
Média	**0,159**	**0,262**	**0,441**	**0,141**	**0,248**	**0,410**	**0,167**	**0,268**	**0,574**
Desvio padrão	**0,011**	**0,002**	**0,137**	**0,001**	**0,008**	**0,122**	**0,002**	**0,022**	**0,089**

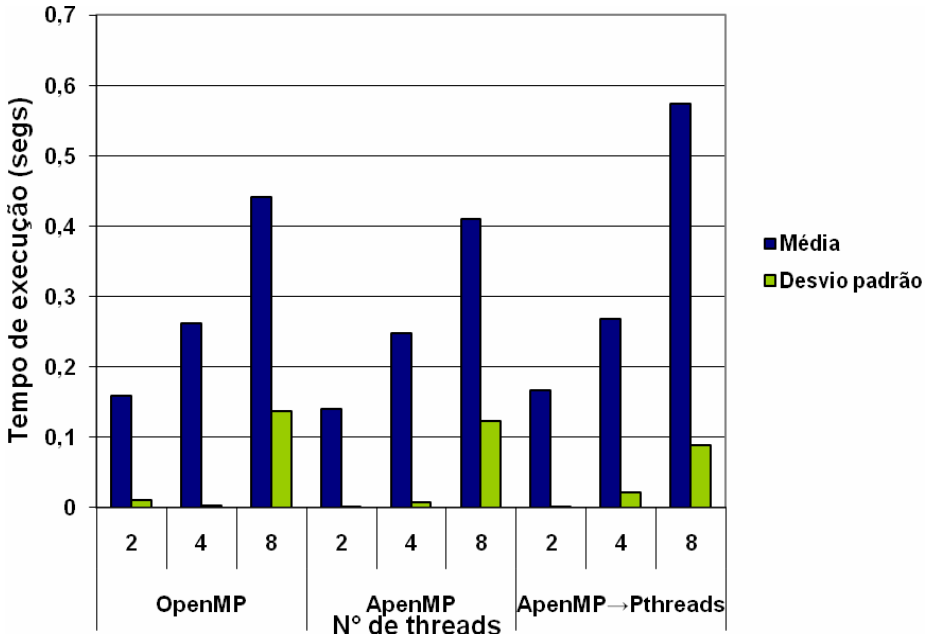

Figura 7.1 - Gráfico mostrando o tempo de execução dos algoritmos em OpenMP(diretiva *parallel*), ApenMP e Pthreads.

Os resultados de desempenho permitem observar que a diferença para as versões dos programas desenvolvidos através de ApenMP e Pthreads são coerentes com outros resultados de desempenho documentados em Cordeiro et al. (2005) e Benitez et al. (2004). Em relação à comparação dos desempenhos observado para OpenMP e ApenMP, observa-se que este último produziu um tempo de execução inferior. Este resultado é bastante satisfatório para os resultados esperados, mas relembra-se que a aplicação reflete uma natureza de paralelismo de tarefa, não sendo este o nicho onde se espera o maior desempenho de OpenMP.

7.2.2 Paralelismo de dados

Neste experimento foi utilizado o algoritmo da Fig. 6.6 do Capítulo 6, onde a diretiva ***for*** expressa o paralelismo de dados em um laço *for*. Neste algoritmo, as iterações do laço são divididas em tantas regiões paralelas quanto forem o número de *threads* no pool de execução.

Na Tab. 7.2 e na Fig. 7.2 são apresentados a média e o desvio padrão em segundos do tempo de execução dos algoritmos de acordo com o número de *threads*.

Tabela 7.2 - Tempo de execução dos algoritmos.

	OpenMP			ApenMP			ApenMP ⟶ Pthreads		
Threads	**2**	**4**	**8**	**2**	**4**	**8**	**2**	**4**	**8**
Média	**0,101**	**0,105**	**0,115**	**0,187**	**0,211**	**0,356**	**0,196**	**0,236**	**0,369**
Desvio padrão	**0,002**	**0,006**	**0,011**	**0,056**	**0,012**	**0,013**	**0,031**	**0,004**	**0,006**

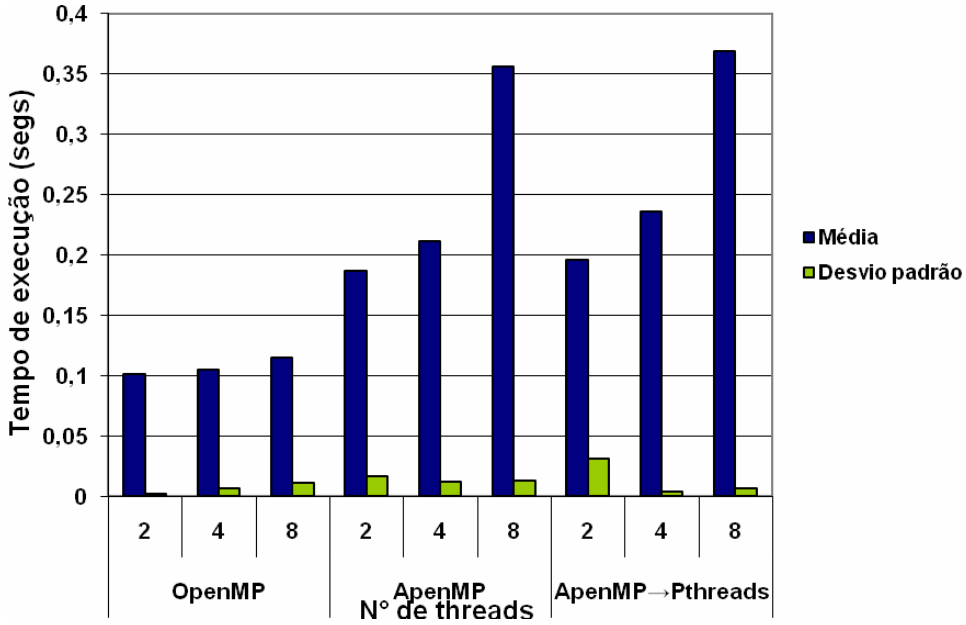

Figura 7.2 - Gráfico mostrando o tempo de execução dos algoritmos em OpenMP(diretiva *for*), ApenMP e Pthreads.

Como esperado, quando a abordagem está relacionada ao paralelismo de dados, a execução do código original em OpenMP leva vantagem em relação à versão do código traduzido com ApenMP e da versão em Pthreads, uma vez que este tipo de paralelismo é o nativo de OpenMP e onde existe a expectativa de melhor desempenho. Em relação ao tempos de execução das versões de ApenMP e Pthreads é verificado que para a aplicação em questão, o desempenho de ApenMP também se mantém superior.

7.3 *Overhead* no escalonamento de Anahy

Como estudos de pesquisa também foram realizados testes sobre o mesmo algoritmo descrito na Fig. 5.4 para analisar o *overhead* na utilização de diferentes tamanhos de *chunks* nos PVs de Anahy.

A metodologia utilizada foi definir um número de PVs fixo, neste caso foram utilizados 2 PVs (o computador utilizado no experimento possui dois *cores),* sendo utilizados *chunks* de tamanho 2, 4, 8, e 16. Os resultados em segundos na Tab. 7.3 e na Fig. 7.3 apresentam uma média e desvio padrão de 15 execuções sobre cada tamanho de *chunk* correspondente.

Tabela 7.3 - *Overhead* no escalonamento.

Chunks	**2**	**4**	**8**	**10**
Média	**0,187**	**0,211**	**0,356**	**0,486**
Desvio padrão	**0,016**	**0,012**	**0,013**	**0,003**

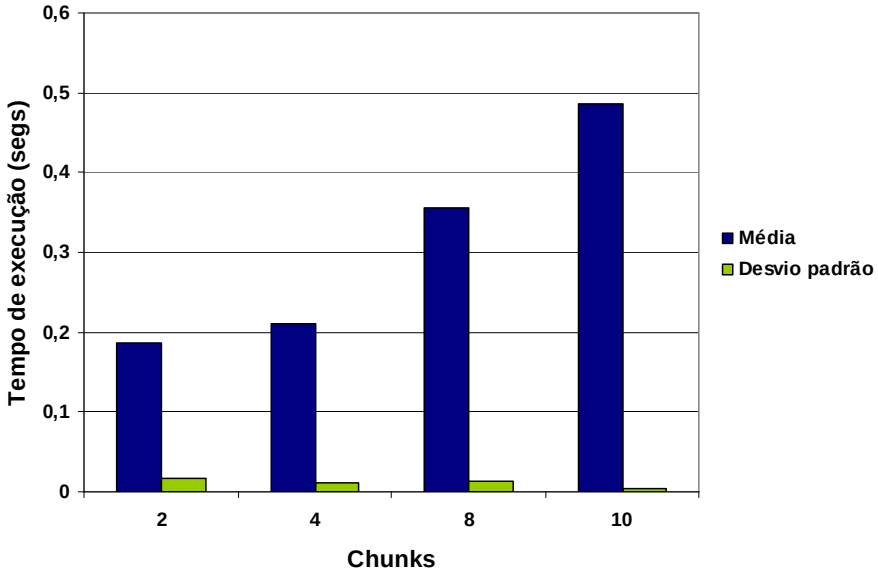

Figura 7.3 - *Overhead* no escalonamento sobre diferentes *chunks*.

Tomando os resultados da Tab. 7.3, assumimos que o custo computacional da aplicação teste é de 0,187 * 2 = 0,374 segundos, onde cada um dos PVs do *pool* de execução é responsável pela execução de um *chunk*. Este custo despreza tanto a influência da fração serial do programa, uma vez que ela é mínima, quanto o

overhead do próprio sistema, pois é considerado o tempo total de execução e assume-se que este overhead é igualmente distribuído entre o tempo gasto para o processamento de cada *chunk*. Assim, com quatro *chunks*, tem-se que cada um possui um custo computacional de 0,374 / 4 = 0,0935 segundos. Como espera-se que cada PV execute a carga associada a dois *chunks*, o menor tempo de execução é duas vezes o custo computacional de cada um dos quatro *chunks*, ou seja, o mesmo tempo final de execução 0,187 segundos era o esperado. Como o tempo obtido foi, na realidade, de 0,211 segundos, tem-se que 0,211 – 0,187 = 0,024 segundos é o overhead para geração de dois *chunks* extras, ou seja, 12% de *overhead* na execução. Aplicando o mesmo raciocínio para os casos com oito e dez *chunks* tem-se os resultados de *overhead* na geração de *chunks*, respectivamente 0,169 (90% de *overhead* na execução) e 0,30 segundos (259 % de *overhead* na execução).

Embora os resultados não sejam conclusivos, observa-se que o *overhead* de execução está associado ao custo de criação de novos *threads*. Athreads sendo uma biblioteca de *threads* executada em nível usuário, este tempo é sabidamente baixo, como pode ser verificado nos resultados de desempenho apresentado nas seções anteriores onde o tempo de processamento de aplicações com Athreads foi inferior ao tempo apresentado na execução das versões Pthreads.

7.4 Outros experimentos

Também foram realizados experimentos com um programa mesclando o uso das diretivas *omp parallel* da Fig. 6.3 e *omp for* da Fig. 6.6. Neste programa existem duas seções paralelas aninhadas, a mais externa definida pelo *parallel* e a mais interna pelo *for*. Nos experimentos realizados foram utilizados 2, 4 e 8 PVs e 2 *threads* em cada região paralela e o computador em questão para a realização dos testes foi o mesmo descrito no início deste capítulo. A medida de desempenho tomada foi o tempo total de execução, considerando três possibilidades de execução paralela:

1. Criação de *athreads* para o *parallel* e para o *for*, tantos forem os pvs utilizados - se forem 2 PVs, 2 pvs executando o parallel e outros dois executando o for

2. Criação de *athreads* somente para o *parallel* (tantas quanto forem os PVs).

3. Criação de *athreads* somente para o *for*.

Na Tab. 7.4 e Fig. 7.4 é possível verificar a média e desvio padrão de 15 execuções para os 3 casos descritos anteriormente.

Tabela 7.4 - Tempo de execução dos 3 casos de paralelização.

Casos	**1**			**2**			**3**		
PVs	**2**	**4**	**8**	**2**	**4**	**8**	**2**	**4**	**8**
Média	**0,334**	**0,335**	**0,341**	**0,255**	**0,257**	**0,259**	**0,265**	**0,266**	**0,269**
Desvio padrão	**0,019**	**0,018**	**0,015**	**0,015**	**0,017**	**0,026**	**0,013**	**0,021**	**0,019**

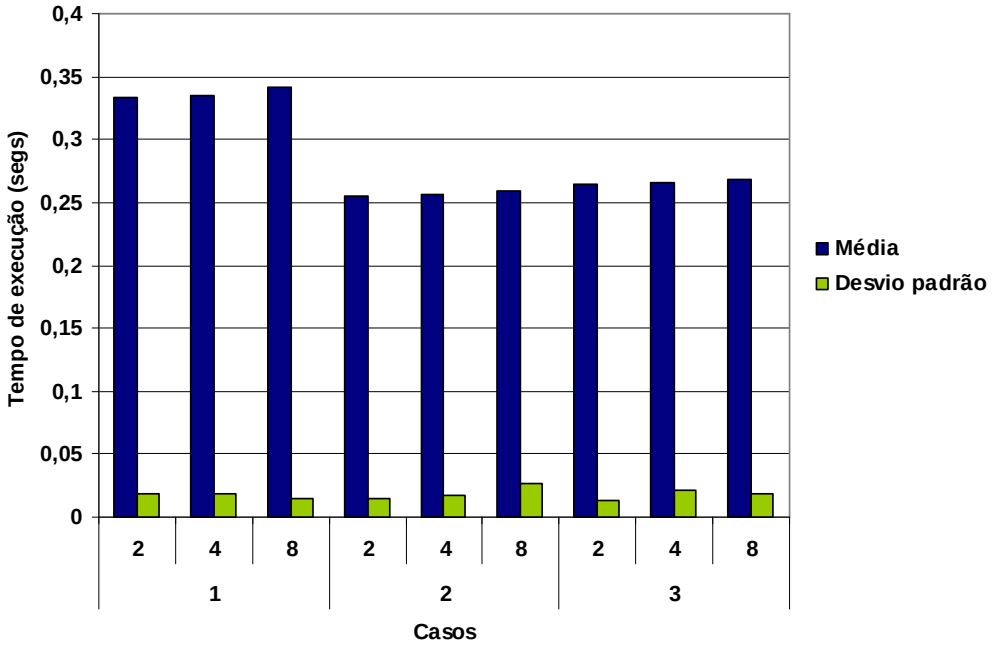

Figura 7.4 - Gráfico do tempo de execução dos 3 casos de paralelização.

Como podem ser observados os resultados da Tab. 7.4, o primeiro caso na criação de *athreads* para as diretivas *parallel* e *for* foram criados em total

6 *athreads*, o tempo de execução deste caso foi o pior em relação aos demais. Em relação ao segundo caso, foram criados 2 *athreads* para a diretiva *parallel* e a diretiva *for* foi tratada pelo OpenMP, neste caso onde o *parallel* gera *athreads* e o *for* é executado pelo OpenMP o tempo de execução foi o melhor. Por outro lado, no terceiro caso foram criados *athreads* somente para a diretiva *for* e a diretiva *parallel* foi tratada pelo OpenMP, neste caso foram gerados 4 *athreads* e o tempo de execução esteve muito próximo ao segundo caso. Em relação aos números de PVs utilizados, estes não afetaram os tempos de execução finais.

8 Conclusões e Trabalhos futuros

O principal objetivo do trabalho realizado foi avaliar o emprego de um mecanismo de escalonamento de lista dinâmico em programas paralelos implementados com uma interface de programação baseada na extração de paralelismo de dados. Os resultados alcançados, em função das avaliações de desempenho realizadas, indicam que é possível obter um ambiente de programação paralela com tais características.

Este trabalho especificou, implementou e validou por meio de experimentos a ferramenta ApenMP. ApenMP consistiu em um protótipo de uma ferramenta de programação paralela oferecendo recursos de programação definidos segundo um subconjunto da especificação do padrão OpenMP e um núcleo de execução baseado no modelo proposto por Anahy, implementado em Athreads.

Com uso de ApenMP, o programador pode se beneficiar das vantagens da simplicidade de paralelização de um programa escrito em OpenMP e obter um desempenho eficiente pela utilização do núcleo de execução provido por Athreads, implementando mecanismos de escalonamento baseados algoritmos de listas dinâmicos.

Em relação aos resultados de desempenho, estes se mostraram satisfatórios, refletindo a natureza esperada para os casos estudados. Observou-se que os experimentos realizados com programas destacando o paralelismo de tarefas, o tempo de execução das versões traduzidas para Athreads foi inferior aqueles apresentados com a compilação do caso em OpenMP. Em contrapartida, ao se tratar de códigos que expressem o paralelismo de dados, OpenMP levou vantagem em relação à versão gerada para Athreads.

A continuidade deste trabalho pode se dar pela introdução do conjunto completo da especificação de OpenMP no pré-processador ApenMP e pela implementação de um núcleo de execução próprio, eliminando a necessidade da geração de código intermediário Athreads.

Referências

AHO, A.V.; SETHI, R.; ULLMAN, J. D. **Compiladores Princípios, Técnicas e Ferramentas**. Rio de Janeiro : Editora Guanabara Koogan S.A, 1995.

AKHTER, S.; ROBERT, J. **Multi-core Programming: Increasing Performance through Software Multithreading**. Hillsboro : Intel Press, 2006.

BACKUS, J. **Can programming be liberted from the von-Neumann style? A functional style and its algebra of programs**. Em: Communications of the ACM**,** v.21, n.8, p.613-641, Ago. 1978.

BASUMALLIK, A.; EIGENMANN R. **Incorporation of OpenMP Memory Consistency into Conventional Dataflow Analysis**. 71-82. Em: IWOMP (International WorkShop on OpenMP). Dresden, Germany, 2008.

BENITEZ, E. D.; VILLA REAL, L. C.; CARDOZO JÚNIOR, M. A.; CAVALHEIRO, G. G. H. **Avaliação de Desempenho de Anahy em Aplicações Paralelas**. Em: WPerformance – III Workshop em Desempenho de Sistemas Computacionais e de Comunicação, 2004, Salvador.

BERRENDORF, R.; NIEKEN, G., **Performance characteristics for OpenMP constructs on different parallel computer architectures**. Em: Concurrency: Practice and Experience, 12(12):1261-1273. 2000.

BLUMOFE, R.; JOERG, C.; KUSZMAUL, B.; LEISERSON, C.; RANDALL, K.; ZHOU, Y. **Cilk: An Efficient Multithreaded Runtime System**. Em: Journal of Parallel and Distributed Computing, v. 37, i. 1, 55-69, Agosto, 1996.

BULL. J. M., **Measuring Synchronization and Scheduling Overheads in OpenMP**. Em: Proc. of the 1st European Workshop on OpenMP (EWOMP ’99), Lund, Sweden, Setembro 1999.

BUTENHOF, D. R. **Programming with POSIX threads**. Addison-Wesley Professional Computing Series, 1997.

CAVALHEIRO, G. G. H. **A general scheduling framework for parallel execution enviroments**. Em: Proceedings of SLAB’01, Brisbane, Australia, Maio 2001.

CAVALHEIRO, G. G. H. **Princípios da programação concorrente**. Em: ERAD 2004.

CAVALHEIRO, G. G. H.; DENNEULIN, Y.; ROCH, J.-L. **A general modular specification for distributed schedulers**. Em: LNCS 980 Springer Verlag, editor, Proceedings of Europar'98, Southampton, England, Setembro. 1998.

CAVALHEIRO, G. G. H.; GASPARY, L. P.; CARDOZO, M. A.; CORDEIRO, O. C. **Anahy: A programming environment for cluster computing.** Em: VECPAR, 2006.

CAVALHEIRO, G. G. H.; SANTOS, R. R. **Multiprogramação leve em arquiteturas multi-core**. Em: Atualizações em Informática. Cap 7. Rio de Janeiro: PUC-RJ. 2007.

CHANDRA, R.; DAGUM, L.; KOHR, D.; MAYDAN, D.; MCDONALD, J.; MENON, R. **Parallel Programming in OpenMP**. Em: Morgan Kaufmann, San Francisco, 2001.

CORDEIRO, O. C.; PERANCONI, D. S.; VILLA REAL, L. C.; DALL'AGNOL, E. C.; CAVALHEIRO, G. G. H. **Exploiting Multithreaded Programming on Cluster Architectures**. Em: HPCS 2005.

DAGUM, L.; MENON, R. **Openmp: an industry standard api for sharedmemory programming.** Em: Computational Science and Engineering, IEEE, 5(1):46–55, 1998.

DALL'AGNOL, E. C.; VILLA REAL, L. C.; BENITEZ E. D.; CAVALHEIRO, G. G. H. **Portabilidade na programação para o processamento de Alto Desempenho**. Em: WSCAD - Workshop em Sistemas Computacionais de Alto Desempenho, 2003, São Paulo, 2003.

DREPPER, U.; MOLNAR, I. **The native POSIX thread library for Linux**. 2005. Disponível em: <http://people.redhat.com/drepper/nptl-design.pdf>. Acesso: 24 abr. 2008.

FLYNN, M. J. **Very high-speed computing systems**. IEEE, 54(12):1901–1909, 1966.

FLYNN, M. J. **Some computer organizations and their effectiveness**, 1972.

GRAHAM, L. R. **Bounds on multiprocessing timing anomalies**. Em: SIAM Journal of Applied Mathematics, v.(17):2, 1969.

HENNESSY, J. L.; PATTERSSON, D. A. **Computer architecture (4th ed.): a quantitative approach**. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

IEEE Computer Society. A Backgrounder on IEEE Std. 1003.1, 1995. Disponível em: <http://www.opengroup.org/austin/papers/backgrounder.html> Acesso: 19 abr. 2008.

KUMAR, R; TULLSEN, D. M. The **Architecture of efficient multi-core processors: a holistic approach**. Em: Advances in Computers, V. 69. ZELKOWITZ, M. V., ed. Amsterdam: Elsevier. 2007.

LEOPOLD, C. **Parallel and distributed computing**. John Wiley & Sons, England, 2001.

LEVINE, John R.; MASON, T.; BROWN, D. **Lex & Yacc**. Beijing: O'Reilly, 1995.

LIN, Y. **Static Nonconcurrency Analysis of OpenMP Programs**. Em: International Workshop on OpenMP, 2005.

OLIVEIRA, R. L. de; REIS, T. A.; VIEIRA, M. A.; CHARÃO, A. S. **Avaliação do Suporte à Programação Multithread com OpenMP no compilador GCC**. VIII Workshop de Software Livre, Porto Alegre / RS, 2007.

PATTERSON, D. A.; HENNESSY J. L.**Computer Organization and Design: the hardware/software interface**. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

ROSE, C. de; NAVAUX, P. O. A. **Arquiteturas Paralelas**. Editora Sagra Luzzato, 2003.

VON NEUMANN, J. **First Draft of a Report on the EDVAC**. Moore School of Electrical Engineering, University of Pennsylvania, 1945.

ZHANG, Y.; BURCEA, M.; CHENG, V.; HO, R.; VOSS, M. **An Adaptive OpenMP Loop Scheduler for Hyperthreaded SMPs**. Anais do PDCS-2004: International Conference on Parallel and Distributed Systems, San Francisco, CA, September 2004.

ZHANG, Y.; DUESTERWALD, E.; GAO, G. R. **Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers**. Anais do LCPC 2007: 95-109. 2007.