UNIVERSIDADE FEDERAL DE PELOTAS
Centro de Desenvolvimento Tecnológico
Programa de Pós-Graduação em Computação

Dissertação

Algoritmos de Escalonamento de Lista em Ambientes Multithread Dinâmicos: Análise de Estudos de Caso Teóricos e Práticos

CÍCERO AUGUSTO DE SOUZA CAMARGO

Pelotas, 2013

CÍCERO AUGUSTO DE SOUZA CAMARGO

Algoritmos de Escalonamento de Lista em Ambientes Multithread Dinâmicos: Análise de Estudos de Caso Teóricos e Práticos

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal de Pelotas, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Orientador: Prof. Dr. Gerson Geraldo Homrich Cavalheiro

Pelotas, 2013

Dados Internacionais de Publicação (CIP)

C172a Camargo, Cícero Augusto de Souza
Algoritmos de escalonamento de lista em ambientes multithread dinâmicos : análise de estudos de caso teóricos e práticos / Cícero Augusto de Souza Camargo; Gerson Geraldo Homrich Cavalheiro, orientador. – Pelotas, 2013.
112 f.; il.

Dissertação (Mestrado em Computação), Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas. Pelotas, 2013.

1.Programação multithread. 2.Ambientes dinâmicos. 3.Escalonamento de lista. 4.Arquiteturas multicore. I. Cavalheiro, Gerson Geraldo Homrich, orient. II. Título.

CDD: 005.1

Catalogação na Fonte: Leda Cristina Peres Lopes CRB:10/2064
Universidade Federal de Pelotas

Banca examinadora:

Prof. Dr. Adenauer Corrêa Yamin

Prof. Dr. Alexandre Carissimi

Prof. Dr. Maurício Lima Pilla

AGRADECIMENTOS

Primeiramente agradeço a minha família, por ter me dado toda a estrutura e condições de chegar onde cheguei, e a minha esposa, Aline, por toda sua compreensão nas minhas ausências e por todo o apoio emocional que me deu nas partes mais complicadas do trajeto, desde a graduação até aqui.

Agradeço os diversos professores que compartilharam seu conhecimento e experiência durante meu curso de mestrado, e que, de alguma forma, contribuiram para minha formação como profissional e como ser humano. Meu profundo agradecimento à preciosa orientação do professor e amigo Gerson Cavalheiro, que várias vezes me recebeu em sua casa, compartilhando, além de seu vasto conhecimento, momentos de sua família. Meu “muito obrigado” às professoras Simone André da Costa Cavalheiro e Luciana Foss, que contribuíram com seu conhecimento e um fundamento importantíssimo para este trabalho: o conhecimento sobre grafos.

Agradeço aos demais professores e colegas do LUPS (*Laboratory of Ubiquitous and Parallel Systems*) por todo o convívio e conhecimento compartilhado, em especial ao colega Alan Schlindvein de Araújo, parceiro nas sessões de programação que resultaram na eficiente implementação do ambiente Anahy3, um dos legados deste trabalho. Agradeço também os colegas da empresa Conrad Caine, com os quais evoluí muito profissionalmente, e agradeço, também, aos superiores, que foram compreensivos com minhas ausências em virtude dos compromissos acadêmicos.

Agradeço ainda, a oportunidade de ter cursado toda minha graduação e o curso de mestrado em uma ótima estrutura pública oferecida pela Universidade Federal de Pelotas. Agradeço também o importantíssimo apoio financeiro concedido pela CAPES durante o período de realização deste curso de mestrado.

Agradeço ao Grande Arquiteto do Universo, que olha por nós e faz com que o mundo conspire a nosso favor.

Eu aprendi muito cedo, mateando com meu avô,
que um homem a gente conhece nos rastros que ele deixou...
...cada consciência é um caminho que pode ou não ir além
e só cuida para onde vai, quem respeita de onde vem.
— ANGELO FRANCO

RESUMO

CAMARGO, Cícero Augusto de Souza. **Algoritmos de Escalonamento de Lista em Ambientes Multithread Dinâmicos: Análise de Estudos de Caso Teóricos e Práticos**. 2013. 112 f. Dissertação (Mestrado) – Programa de Pós-Graduação em Computação. Universidade Federal de Pelotas, Pelotas.

A popularização das arquiteturas *multicore* trouxe a capacidade de processamento paralelo para diversos dispositivos de computação pessoal, como *laptops*, *tablets* e *smartphones*. No entanto, para que uma aplicação se beneficie do hardware paralelo, precisamos do suporte de ferramentas de programação concorrente que forneçam uma interface simples e abstrata, a qual esconda do programador as complexidades do hardware e do sistema operacional.

O modelo *multithread* é o que mais se adequa ao modo de execução das arquiteturas *multicore*. Diversas ferramentas de programação concorrente amplamente utilizadas como OpenMP, Intel® Cilk Plus e Intel® Threading Building Blocks, oferecem interfaces de programação *multithread* e abstraem o escalonamento de threads em nível de aplicação, empregando estratégias baseadas em algoritmos de lista.

Uma vez que algoritmos de lista foram originalmente concebidos para o escalonamento estático de Grafos Dirigidos Acíclicos de tarefas (DAG – *Directed Acyclic Graph*), este trabalho se dedica a analisar o impacto gerado ao empregar tais algoritmos no núcleo de escalonamento de ambientes *multithread* dinâmicos. Para tanto, foram implementadas uma ferramenta de simulação e um ambiente real de execução *multithread*, este batizado de Anahy3.

Os resultados obtidos nas simulações indicam que escalonamentos de lista em ambientes *multithread* podem fornecer, para uma dada aplicação, tempos de execução muito próximos daqueles obtidos no escalonamento estático do DAG que descreve a mesma aplicação, mesmo que o grafo de threads seja gerado em tempo de execução. Nas execuções reais com Anahy3 foi possível constatar que algoritmos de lista podem gerar escalonamentos eficientes, resultando em desempenhos semelhantes àqueles fornecidos pelas principais ferramentas *multithread* da academia e da indústria. Contudo, alguns resultados demonstram que a eficiência das técnicas utilizadas na implementação do ambiente de execução é tão importante quanto a ordem de execução dos threads.

Palavras-chave: Programação multithread, ambientes dinâmicos, escalonamento de lista, arquiteturas multicore.

ABSTRACT

CAMARGO, Cícero Augusto de Souza. **Evaluating the Impacts of Applying List Scheduling Algorithms on Dynamic Multithreaded Environments in Theory and Practice**. 2013. 112 f. Dissertação (Mestrado) – Programa de Pós-Graduação em Computação. Universidade Federal de Pelotas, Pelotas.

The popularization of multicore architectures made parallel-processing available in sereval personal computing devices such as laptops, tablets and smartphones. However, for an application to benefit from the parallel hardware, we need the support of concurrent programming tools that provide a clean and abstract interface, that hides from the programmer the complexities of the underlying platform.

The multithreaded model is the most suitable for the execution mode of multicore architectures. Several widely used concurrent programming tools, such as OpenMP, Intel Cilk Plus, and Intel Threading Building Blocks, provide multithread programming interfaces and abstract thread scheduling at the application level, using strategies based on list algorithms.

Once list algorithms were originally designed to schedule DAGs (Directed Acyclic task Graphs) statically, this study aims to analyze the impact of applying list algorithms in the core of dynamic multithreaded environments. In order to this, we implemented a simulation tool and a real multithreaded environment that we called Anahy3.

Simulation results indicate that, given an application, list schedules can provide, in multithreaded environments, execution times very close to those obtained when scheduling the application DAG statically, even when the thread graph is generated at runtime. In real executions with Anahy3, we could show that list algorithms can generate efficient schedules, providing execution performances equivalent to the ones obtained using the most prominent multithreaded tools in academy and industry. However, some results showed that the an efficient implementation of the execution environment is as important as the execution order of threads.

Keywords: multithreaded programming, dynamic environments, list scheduling, multicore architectures.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

AKSSim	Anahy Kernel Schedulers Simulator
API	Application Programming Interface
DAG	Directed Acyclic Graph
DCG	Directed Cyclic Graph
FCFS	First Come, First Served
FIFO	First In, First Out
HLFET	Highest Levels First with Estimated Times
HLFNET	Highest Levels First with No Estimated Times
LIFO	Last In, First Out
LPT	Largest Processing Time first
NUMA	Non-Uniform Memory Access
PV	Processador Virtual
SCFET	Smallest Co-levels First with Estimated Times
SCFNET	Smallest Co-levels First with No Estimated Times
SMP	Symmetric Multi-Processor
SPT	Shortest Processing Time first
TBB	Intel(R) Threading Building Blocks

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento de ferramentas de programação multithread é um dos assuntos de crescente interesse na pesquisa em Computação devido à recente popularização de arquiteturas multiprocessadas, em função do surgimento da tecnologia *multicore*. Com esse tipo de arquitetura, processadores com potencial de execução paralela, que anteriormente eram encontrados somente em computadores de grandes centros de pesquisa e processamento de alto desempenho, passaram a estar presentes em dispositivos de computação pessoal, até mesmo em dispositivos móveis como *tablets* e *smartphones*. Contudo, para que as aplicações sejam beneficiadas pelo hardware paralelo, os programas devem ser desenvolvidos com o apoio de ferramentas que ofereçam recursos para a expressão da concorrência presente nestas aplicações e um ambiente de execução que realize a exploração do hardware paralelo.

O modelo multithread, amplamente difundido na academia e na indústria, é o que mais se adequa à arquitetura *multicore* (VALIANT, 1990). Neste modelo, um *thread* compõe um dos possíveis fluxos de execução concorrentes de um processo. Sistemas operacionais modernos implementam um modelo de threads que permite a execução paralela de threads de um mesmo processo, em uma arquitetura multiprocessada. Ferramentas amplamente utilizadas como Pthreads (NICHOLS; BUTTAR; FARRELL, 1998) e Solaris Threads (POWELL et al., 1991) são focadas no suporte multithread em nível de núcleo do sistema operacional. Neste nível todos os threads executáveis concorrem pelo acesso ao(s) processador(es), sendo o escalonamento gerenciado primariamente pelo próprio sistema operacional. Em geral, criar um número grande de threads nesse nível gera sobrecustos consideráveis, pois a concorrência pelos recursos físicos de processamento é muito alta e parte significativa do tempo de execução da aplicação é gasta em chaveamentos de contexto. Tal restrição faz com que o programador precise adequar a geração de concorrência na aplicação à plataforma de execução atual, resultando em um programa com escalabilidade limitada e com paralelismo de granularidade relativamente alta.

Ferramentas de programação multithread como OpenMP (CHANDRA et al., 2001), Intel® Cilk Plus (INTEL, 2012a), Intel® Threading Building Blocks (TBB) (REINDERS, 2007) e Athreads (CAVALHEIRO et al., 2007), oferecem modelos de implementação de threads onde o programador descreve a concorrência em nível aplicativo, sem levar em conta a capacidade de execução paralela do hardware disponível. Em geral, o escalonamento de threads no nível aplicativo gera sobrecustos baixos, permitindo ao programador descrever completamente a concorrência de sua aplicação, sem precisar considerar limites impostos pelo sistema operacional e pelo hardware. O ambiente de execução então adequa a criação de threads de sistema à plataforma subjacente e realiza dinamicamente o mapeamento do trabalho dos threads de usuário sobre as unidades de execução criadas no nível de sistema. A estratégia para a realização deste mapeamento e os parâmetros considerados neste processo compõem um algoritmo de escalonamento realizado em nível aplicativo (FEITELSON, 1997). Em geral, ferramentas que manipulam threads no aplicativo oferecem limitações em relação àquelas que manipulam threads em nível de sistema, como, por exemplo, a ausência de preempção ou migração. Porém, estas ferramentas permitem o desenvolvimento de aplicações com paralelismo de granularidade mais fina, aumentando o nível da concorrência na aplicação de maneira a facilitar o balanceamento de carga realizado pelo ambiente de execução e amplificar a escalabilidade da aplicação.

1.1 Motivação

Algoritmos de escalonamento de lista possuem sua eficiência comprovada (GRAHAM, 1966) quando a aplicação pode ser descrita estaticamente em termos de seu fluxo de dados (JOHNSTON; HANNA; MILLAR, 2004) em um Grafo Acíclico Dirigido (DAG – *Directed Acyclic Graph*). Nesse grafo, os vértices representam *tarefas* (trechos de código sequencial que processam dados de entrada e geram dados de saída) e as arestas representam dependências de dados entre tarefas. Considerando custos associados aos vértices e às arestas, a sequência com maior soma de custos é chamada de *caminho crítico*. Graham (1976) demonstrou que a habilidade de um algoritmo em identificar e gerenciar o escalonamento de tarefas no caminho crítico influi diretamente no tamanho do escalonamento da aplicação, isto é, no tempo total da execução sobre uma máquina com m processadores idênticos. Contudo, calcular a localização do caminho crítico exige o conhecimento do grafo completo antes do início da execução do programa, o que não é a realidade de ambientes dinâmicos, onde o grafo é construído conforme a execução do programa evolui.

Diversos ambientes modernos de execução multithread, como os que dão suporte aos ambiente OpenMP, Intel ® Cilk Plus, Intel ® Threading Building Blocks (TBB) e Athreads, empregam estratégias de escalonamento baseadas em algoritmos de lista no gerenciamento da concorrência por recursos em seu núcleo. Contudo, a unidade básica de escalonamento considerada nestes ambientes é o *thread*, que encapsula uma sequência de tarefas em uma estrutura de maior granularidade. Em geral, nestes ambientes os threads são criados dinamicamente, conforme o programa evolui. Desta forma, não é possível antecipar a estrutura completa do grafo, tampouco os custos das tarefas que serão criadas, sendo necessário o emprego de heurísticas de escalonamento em linha (*online*). Assim, os ambientes citados utilizam estratégias de escalonamento de lista em ambientes multithread dinâmicos com uma eficiência comprovada empiricamente.

Este trabalho se propõe a investigar os requisitos teóricos e práticos necessários para a aplicação de algoritmos de lista em ambientes multithread dinâmicos, e a analisar o impacto deste tipo de estratégia de escalonamento aplicada a estes ambientes. O trabalho está inserido no projeto "GREEN-GRID: Computação de Alto Desempenho Sustentável". O projeto envolve a participação de quatro Instituições de Ensino Superior, a saber: a Universidade Federal de Pelotas (UFPEL), a Universidade Federal de Santa Maria (UFSM), a Universidade Federal do Rio Grande do Sul (UFRGS) e a Pontifícia Universidade do Rio Grande do Sul (PUCRS); perdurando até o ano de 2013.

1.2 Objetivos e metodologia

Diversas ferramentas de programação multithread amplamente utilizadas na academia e na indústria realizam o escalonamento de threads em nível aplicativo; frequentemente seus ambientes de execução utilizam técnicas baseadas em algoritmos de lista em seu núcleo. O principal objetivo deste trabalho está em demonstrar que algoritmos de escalonamento de lista, aplicados a ambientes que manipulam grafos de threads gerados dinamicamente, podem fornecer um comportamento semelhante ao escalonamento estático da mesma aplicação e tempos de execução satisfatórios.

Para alcançar os objetivos propostos iremos analisar resultados de escalonamento obtidos a partir de simulações de escalonamento multithread e execuções de aplicações de teste em ambientes multithread reais.

1.2.1 Simulações

Threads encapsulam sequências de tarefas de tal forma que, a partir do grafo de um programa multithread pode-se extrair as dependências geradas pelas operações de criação e sincronização de threads e construir um DAG de tarefas, representando *a mesma aplicação*. Desta forma, por meio de simulações, podem ser obtidos os escalonamentos de ambos os tipos de grafo, considerando-se diferentes estratégias de escalonamento aplicáveis a ambientes que manipulam um ou outro tipo de grafo. Utilizando esta abordagem iremos comparar diretamente o desempenho das estratégias de escalonamento de lista aplicadas estaticamente a DAGs com o desempenho programas multithread, executando com o suporte de algoritmos de lista sobre os ambientes dinâmicos simulados. Iremos avaliar a aplicabilidade dos algoritmos de escalonamento de listas em ambientes multithread dinâmicos a partir da análise dos dados gerados pelas simulações. A partir destes dados poderemos medir a distância (em termos do tamanho do escalonamento gerado) entre escalonamentos dinâmicos de grafos threads e escalonamentos estáticos de grafos de tarefas, ambos realizados por algoritmos de lista.

1.2.2 Execuções reais

Para avaliar o impacto de algoritmos de lista em um ambiente multithread real foi necessário reimplementar parte do núcleo de execução de Anahy (CAVALHEIRO et al., 2007). Partindo da experiência da versão anterior deste ambiente e de outros ambientes similares estudados para a realização deste trabalho, foram realizadas modificações no núcleo de escalonamento. Tais mudanças objetivaram, além de otimizar o escalonamento de programas multithread em arquiteturas multiprocessadas, facilitar mudanças na estratégia de escalonamento utilizada pelo ambiente. Esta nova implementação foi batizada de *Anahy3* e será assim referenciada no restante deste trabalho.

Iremos aplicar diversas das políticas exploradas nas simulações, no contexto da implementação do ambiente Anahy3. Estando cientes de que as operações de escalonamento de threads geram sobrecustos de execução, queremos mensurar a aplicabilidade das estratégias avaliadas anteriormente pelas simulações no escalonamento de threads em um ambiente real de execução dinâmica. Este trabalho se propõe a demonstrar, por meio de avaliações práticas do ambiente Anahy3, que algoritmos de lista podem fornecer bons índices no escalonamento de aplicações multithread dinâmicas. Contudo, os resultados indicam que adaptações e técnicas adicionais na implementação do ambiente de execução são indispensáveis para um bom desempenho, principalmente quando as aplicações possuem um paralelismo de grão fino.

1.3 Resultados do trabalho

As simulações realizadas indicam que, na teoria, algoritmos de lista possuem uma grande aplicabilidade no escalonamento de aplicações multithread em ambientes dinâmicos de execução. Os resultados mostraram que os melhores escalonamentos obtidos nas simulações de ambientes multithread geram tamanhos de escalonamento não muito maiores (17%, no máximo, de acordo com os testes realizados) que os melhores escalonamentos estáticos para um DAG que representa a mesma aplicação. Deve-se ressaltar que, ao contrário do que acontece no escalonamento estático de tarefas, os ambientes multithread dinâmicos são não-clarividentes, isto é, o algoritmo de escalonamento nestes ambientes não conhece o futuro da execução de um thread, tendo que tomar decisões, a cada ponto de escalonamento, com base na porção conhecida do grafo até aquele momento. Assim, ná prática o leve aumento no tamanho de escalonamento é compensado pela flexibilidade da execução dinâmica. Ainda, os escalonamentos obtidos nas simulações multithread foram significativamente melhores que escalonamentos de lista randômicos, o que demonstra as vantages da aplicação de uma técnica elaborada de escalonamento de lista.

A execução de aplicações de teste sobre ambientes multithread reais demonstrou a eficiência da utilização dos algortimos de lista no núcleo de escalonamento do ambiente de execução Anahy3. Em comparações realizadas contra ferramentas como OpenMP, Cilk Plus e TBB foi possível perceber a competitividade de Anahy3, que em vários momentos forneceu desempenhos melhores do que os fornecidos pelas demais ferramentas citadas. Contudo, nos testes realizados sobre diversas configurações do ambiente Anahy3, foi possível observar o impacto que diferentes técnicas de implementação possuem sobre o escalonamento de threads e sobre o tempo total de execução. Alguns bons desempenhos observados por escalonamentos randômicos nestes testes também indicam que, para aplicações com estruturas regulares, a otimização das operações do ambiente de execução (e a consequente diminuição dos sobrecustos) possuem um impacto maior sobre o tempo total de escalonamento do que a ordem de execução dos threads.

Como um resultado secundário, foi realizada uma nova e eficiente implementação do modelo Anahy, batizada de *Anahy3*, a qual emprega técnicas de escalonamento de lista exploradas neste trabalho somado a uma arquitetura com listas individuais para cada processador virtual. Foram desenvolvidas estruturas de dados eficientes para o ambiente de execução, as quais podem, também, ser utilizadas para computações de propósito geral. Novos recursos também foram adicionados à ferramenta de simulação AKSSim.

1.4 Estrutura do texto

No Capítulo 2 são introduzidos conceitos relacionados a algoritmos de escalonamento de lista, bem como algumas das estratégias mais utilizadas na literatura. O Capítulo 3 introduz o modelo multithread, conceitos de programação concorrente e especifica o grafo de threads manipulado pelos algoritmos de escalonamento de ambientes multithread. O Capítulo 4 reune diversas técnicas de implementação de ambientes multithread, sendo estas técnicas oriundas de trabalhos relacionados e relatadas por meio da experiência dos autores no desenvolvimento da ferramenta Anahy3. O Capítulo 5 é destinado a detalhar os ambientes de execução de Athreads e Anahy3, ambas implementações do modelo proposto por Anahy. Nesse capítulo são detalhadas as estruturas de dados, as estratégias de escalonamento utilizadas, e as interfaces de programação oferecidas por ambas as ferramentas. Outras ferramentas relevantes de programação multithread são apresentadas no Capítulo 6, com um enfoque especial nas estratégias de escalonamento e nas técnicas empregadas na implementação de seus ambientes de execução. No Capítulo 7 são apresentados os resultados de simulações, onde o desempenho de estratégias estáticas de lista é comparado com o desempenho de ambientes multithread dinâmicos que empregam algoritmos de lista, ambos considerando uma mesma aplicação. Ainda nesse capítulo, a implementação de Anahy 3 é avaliada, por meio de aplicações de teste, e seu desempenho é comparado com o das ferramentas apresentadas no Capítulo 6. O Capítulo 8 conclui o trabalho e discute os novos desafios a serem transpostos a seguir.

2 ALGORITMOS DE ESCALONAMENTO DE LISTA

O problema do escalonamento consiste genericamente em alocar recursos a atividades concorrentes ao longo do tempo. Desde o surgimento de arquiteturas multiprocessadas, na década de 60 (CONWAY, 1963), um dos problemas mais comuns de escalonamento é o de alocar m processadores idênticos para n tarefas concorrentes com relações de precedência, com o objetivo de minimizar o tempo de execução. Algoritmos de escalonamento de lista (*List Scheduling*) (GRAHAM, 1966) resolvem esse problema atribuindo prioridades de execução às tarefas de uma aplicação e organizando-as em uma lista de prioridade. Desta forma, sempre que a execução de uma tarefa termina esta lista é percorrida de maneira que as tarefas prontas com maior prioridade de execução são escalonadas sobre os processadores disponíveis naquele momento e executadas sem preempções. Algoritmos de lista são bastante populares devido à sua simplicidade e ao fato de que escalonamentos com desempenhos satisfatórios podem ser obtidos por um algoritmo de lista, desde que a lista de prioridades seja gerada adequadamente.

Algoritmos de escalonamento de lista possuem limites teóricos de desempenho definidos quando a aplicação pode ser descrita estaticamente em um Grafo Acíclico Dirigido (DAG – *Directed Acyclic Graph*). Uma vez que este DAG é estático, podemos conhecer a estrutura completa da aplicação antes da execução do programa e determinar os custos de cada tarefa e as dependências de dados entres estas, permitindo a geração de uma lista de prioridades que leve a tomadas de decisão inteligentes durante a exeução. No entanto, quando o escalonamento é realizado *online*, pouca ou nenhuma informação sobre a aplicação é conhecida *a priori*, resultando em uma lista de prioridades construída em tempo de execução, conforme o grafo vai sendo conhecido.

O escalonamento de tarefas sobre m processadores idênticos é um problema NP-Difícil (LEUNG, 2004), mesmo em um ambiente estático, onde o algoritmo de escalonamento conhece o grafo inteiro da aplicação antes do início de sua execução. Em cenários bastante restritos, como o escalonamento de DAGs

sobre dois processadores idênticos, é possível obter sempre o escalonamento ótimo (COFFMAN; GRAHAM, 1972). Do contrário, os algoritmos de lista buscam aproximações do caso ótimo empregando diferentes heurísticas de escalonamento.

O restante deste capítulo apresenta o modelo de fluxo de dados e a estrutura de um DAG, bem como a estratégia básica de escalonamento de lista. Ainda neste capítulo serão apresentadas diversas estratégias *offline* para construir a lista de prioridades utilizada durante o escalonamento da aplicação. São apresentadas também estratégias de escalonamento *online*, onde a lista de prioridades é construída em tempo de execução, as quais oferecem uma aplicação mais direta em ambientes de execução multithread dinâmicos.

2.1 Modelo de fluxo de dados

Aplicações baseadas em tarefas e dependências entre tarefas podem ser descritas em termos de seu fluxo de dados em um DAG. No modelo de fluxo de dados (JOHNSTON; HANNA; MILLAR, 2004), cada tarefa representa uma sequência de instruções entre dois pontos de sincronização, a entrada e a saída de dados. Quaisquer outros mecanismos de sincronização, como semáforos ou barreiras, não são permitidos neste modelo. Tarefas representam funções $y = F(x)$, onde x e y são os conjuntos de dados de entrada e de saída, respectivamente, da tarefa F. Uma dada tarefa está pronta para executar quando seu conjunto de dados de entrada se encontra disponível para leitura. Quando a execução de uma tarefa termina, seus dados de saída podem ser usados como entrada para outras tarefas. O conjunto de dados de entrada de uma tarefa pode ser formado pelos dados de saída de diversas tarefas simultaneamente, bem como o conjunto de dados de saída de uma tarefa pode ser utilizado como entrada para diversas outras tarefas.

Este modelo de programação é genérico o suficiente para descrever diversas aplicações (BAKSHI et al., 2005), como: *(i)* computações que usam dados definidos em um nível mais alto de abstração, sem se preocupar em como estes dados são produzidos, e *(ii)* componentes de software altamente extensíveis e reusáveis. Outra aplicação do modelo, de suma importância para este trabalho, é a motivação original do modelo, no contexto da descrição de aplicações para a exploração de paralelismo massivo (JOHNSTON; HANNA; MILLAR, 2004).

2.2 Grafos de tarefas

Um Grafo Acíclico Dirigido (DAG) representa um programa no modelo de fluxo de dados. Em um DAG os vértices representam tarefas e as arestas representam o fluxo (ou dependência) de dados entre estas tarefas. Tanto vértices quanto arestas podem possuir custos associados, respectivamente, ao processamento e à comunicação de dados.

Definição 1 *Um DAG é uma tupla $G = (V, A)$ que representa um programa P. V corresponde ao conjunto de n vértices e A ao conjunto de arestas em G. Cada vértice em V recebe um identificador i, onde $1 \leq i \leq n$, para designar a tarefa τ_i do programa, i.e., $V = \{\tau_1, \tau_2, \ldots \tau_n\}$. Uma relação de precedência entre as tarefas τ_i e τ_j é representada por $\tau_i \prec \tau_j$ e define uma aresta dirigida do grafo: $A = \{(\tau_i \prec \tau_j) | 1 \leq i, j \leq n, i \neq j\}$. Se, no programa P, τ_i é um antecessor imediato de τ_j, então $(\tau_i \prec \tau_j) \in A$. Tal dependência significa que a tarefa τ_j não pode ser executada antes que toda tarefa τ_i tenha sido completamente executada, pois os dados de saída das tarefas τ_i são os dados de entrada de τ_j. Além disso, como G não possui ciclos, $(\tau_i \prec \cdots \prec \tau_j) \in A$ implica que $(\tau_j \prec \cdots \prec \tau_i) \notin A$. Na representação usada neste trabalho, vértices e arestas podem possuir pesos $|\tau_i|$ e $|\tau_i \prec \tau_j|$, representando, respectivamente, o custo computacional associado a τ_i e ao custo de transferência de dados entre τ_i e τ_j.*

Por definição, G pode possuir arestas transitivas: sejam τ_i, τ_j e τ_k tarefas definidas pela aplicação, e $\tau_i \prec \tau_j$, $\tau_i \prec \tau_k$ e $\tau_j \prec \tau_k$ relações de precedência existentes na aplicação, ou seja, $(\tau_i, \tau_j, \tau_k) \in V$ e $(\tau_i \prec \tau_j)$, $(\tau_i \prec \tau_k)$, $(\tau_j \prec \tau_k) \in A$. As tarefas de entrada e saída em G são definidas da seguinte forma: *tarefas de entrada* são um conjunto $V^{in} \subseteq V$ de tarefas τ_i onde não existe $(\tau_j \prec \tau_i) \in A$; *tarefas de saída* são um conjunto $V^{out} \subseteq V$ de tarefas τ_i onde não existe $(\tau_i \prec \tau_j) \in A$. As tarefas de entrada são aquelas que estão prontas para executar quando o programa inicia, enquanto as tarefas de saída são aquelas que não produzem dados para nenhuma outra tarefa no programa. A Figura 1 ilustra um DAG $G = (V, A)$ onde $V = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6, \tau_7\}$ e $A = \{(\tau_1 \prec \tau_4), (\tau_1 \prec \tau_5), (\tau_2 \prec \tau_3), (\tau_3 \prec \tau_4), (\tau_3 \prec \tau_5), (\tau_3 \prec \tau_6), (\tau_4 \prec \tau_7), (\tau_4 \prec \tau_7)\}$.

2.2.1 Caminho Crítico

O *caminho crítico* é uma propriedade do DAG que consiste na sequência de vértices e arestas com maior soma de custos (de processamento e comunicação). Identificar este caminho é importante para os algoritmos de escalonamento, uma vez que o caminho crítico define os limites superior e inferior no tempo de execução do programa descrito pelo DAG, como descrito na inequação 1.

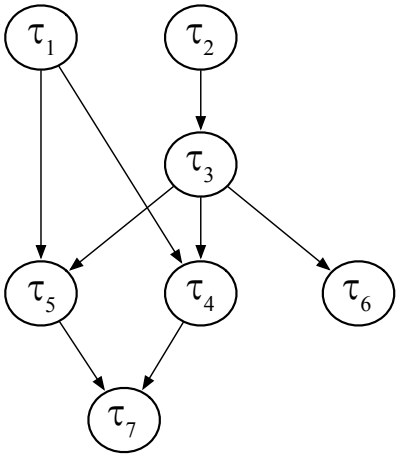


Figura 1: Exemplo de DAG.

$$\frac{T_1}{m} \leq T_m \leq \frac{T_1}{m} + T_\infty \tag{1}$$

Na inequação 1, T_m representa o tempo de execução do programa sobre m processadores. Assim, T_1 representa a soma dos custos de computação de todas tarefas e T_∞ a soma dos custos das tarefas presentes no caminho crítico, dado que a execução das tarefas neste caminho não pode ser paralelizada, mesmo que tenhamos infinitos processadores. Esta inequação dita os limites de desempenho para estratégias de escalonamento de lista de uma forma geral. Graham (1976) demonstrou que o pior escalonamento possível acontece quando priorizamos ao mínimo as tarefas do caminho crítico. Diversos algoritmos de escalonamento de lista, como o algoritmo de Hu (HU, 1961), focam na determinação do caminho crítico e priorização da execução de suas tarefas.

2.3 Estratégia básica

Algoritmos de escalonamento de lista, em geral, seguem uma estrutura básica, a qual é especializada na maneira como a *lista de prioridade* é gerada. Apresentaremos esta estratégia básica descrevendo uma ambiente hipotético de execução que utiliza escalonamento de lista. A descrição que segue é baseada no sistema descrito em (GRAHAM, 1966).

Consideremos uma arquitetura hipotética com m processadores P_i ($1 \leq i \leq m$), e um DAG $G = (V, A)$, onde V é composto por n tarefas ($V = \{\tau_1, \ldots, \tau_n\}$) que devem ser processadas pelos processadores P_i. Consideremos também o conjunto de arestas A, o qual define uma ordem parcial $\prec$ sobre V, e que cada tarefa τ_i possui um custo denotado por $|\tau_i|$. Uma vez que um processador P_i começa a execução da tarefa τ_i, o mesmo deve executá-la de maneira

atômica, i.e., sem interrupções, levando $|\tau_i|$ unidades de tempo. Caso tenhamos $(\tau_i \prec \tau_j) \in A$, a tarefa τ_j depende dos resultados de τ_i e não poderá ser executada até que τ_i tenha terminado. Um processador P_i executa uma tarefa τ_j da seguinte forma: consideremos uma ordem linear $L : (\tau_{k_1}, \ldots, \tau_{k_n})$ sobre τ, que chamaremos de *lista de prioridade*. Em cada instante de tempo t em que uma tarefa tenha sido concluída, cada processador P_i ocioso instantaneamente começa uma busca a partir do primeiro elemento de L até encontrar uma tarefa τ_j que não tenha sido executada. Se todos os antecessores de τ_j, isto é, todas as tarefas τ_i tal que $\tau_i \prec \tau_j$, tiverem sido totalmente executados no instante t, então P_i começa a execução de τ_j. Do contrário, P_i procura a próxima tarefa em L que não tenha sido executada, e assim por diante. Se P_i termina sua busca sem encontrar uma tarefa executável este se torna ocioso, até que outro processador P_j termine a execução de uma tarefa. Neste instante, P_j e qualquer outro processador ocioso realizam uma nova busca por tarefas executáveis em L. Caso dois ou mais processadores tentem executar a mesma tarefa no mesmo instante t, consideraremos que o processador com menor identificador o fará. Os demais processadores seguirão sua busca em L. Todos os processadores começam realizam uma busca em L no tempo $t = 0$ e após isso se comportam como descrito acima até o instante de tempo ω onde todas as tarefas tenham sido executadas. ω representa o tempo total de execução do programa, o *tamanho do escalonamento* gerado, também chamado de *makespan*. Os algoritmos de escalonamento de lista explorados neste trabalho visam a redução de ω, isto é, a redução do tempo total de execução dos programas paralelos.

A geração da lista de prioridades é um fator primordial para o desempenho dos algoritmos de escalonamento de lista. Graham (1976) estudou diversas anomalias que podem ocorrer no escalonamento de um DAG considerando m processadores idênticos. Em um de seus estudos o autor explora o impacto da escolha da lista de prioridades no tamanho do escalonamento. A Figura 2, extraída do estudo original, demonstra o impacto da lista de prioridades no tempo total de execução, considerando duas listas de prioridades diferentes para o mesmo DAG e o mesmo número de processadores. Na Figura 2(a) cada vértice possui um rótulo do tipo T_i/c, indicando a tarefa de índice i, a qual possui um custo de processamento de c unidades de tempo. Os escalonamentos obtidos consideram desprezíveis os custos de comunicação associados às arestas do grafo. Tais escalonamentos são apresentados nas figuras 2(b) e 2(c) pelo uso de diagramas de Gantt (CLARK; GANTT, 1935), onde o eixo horizontal representa o tempo durante a execução da aplicação e o eixo vertical os processadores utilizados.

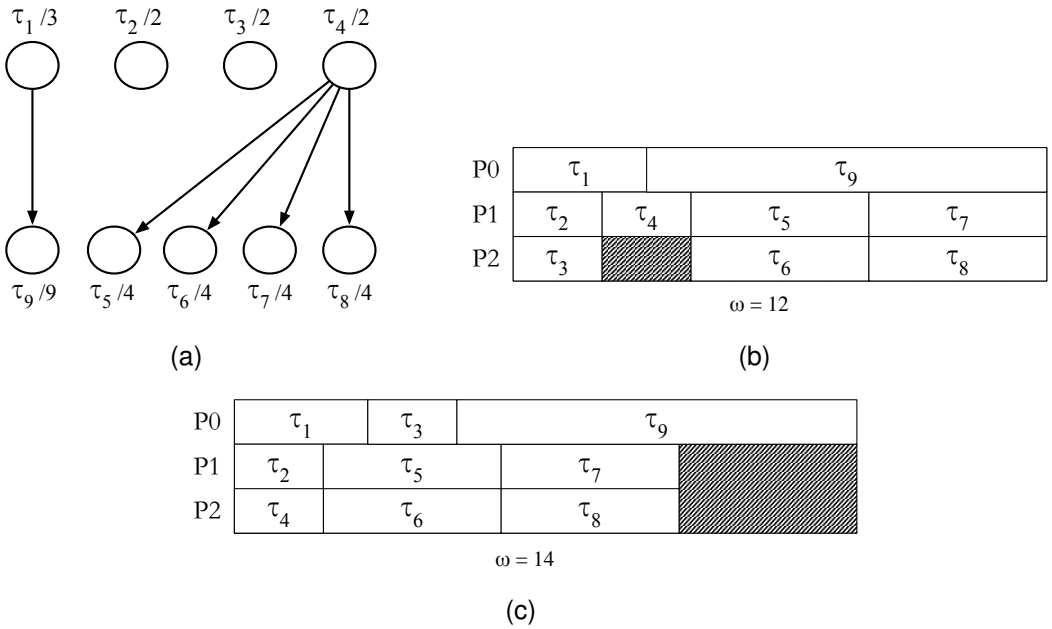


Figura 2: Um DAG (a) e dois escalonamentos considerando $m = 3$. Em (b) temos $L = (T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9)$ resultando em $\omega = 12$ (tempo ótimo). Em (c) temos $L = (T_1, T_2, T_4, T_5, T_6, T_3, T_9, T_7, T_8)$ resultando em $\omega' = 14$.

2.4 Escalonamento estático

Quando temos acesso ao grafo inteiro da aplicação antes do início da execução, podemos aplicar uma estratégia para construir a lista de prioridades estática, que norteará as decisões de escalonamento durante a execução. Quando temos todas as informações sobre as tarefas *a priori* e podemos atribuir prioridades estaticamente, dizemos, então, que este é um algoritmo de escalonamento estático, ou escalonamento determinístico *offline* (LEUNG, 2004).

2.4.1 Atributos de prioridade estáticos

Diversas heurísticas podem ser utilizadas para atribuir prioridades de execução às tarefas de um DAG de maneira estática. Tais heurísticas, quando aplicadas com o intuito de reduzir o tempo total de execução do programa, tentam fazer o melhor balanceamento de carga possível, mantendo os processadores ocupados a maior parte do tempo.

Considerando-se o comprimento de um caminho dirigido no grafo como a soma dos pesos associados a todos os vértices e arestas ao longo do caminho, incluindo os vértices inicial e final, iremos definir dois atributos de prioridade amplamente utilizados na literatura: o *nível* (*level* ou *bottom level*), e o *co-nível* (*co-level* ou *top level*). Ambos são definidos da mesma maneira que a encontrada em (COFFMAN JR; DENNING, 1973). O *nível* da(s) tarefa(s) de

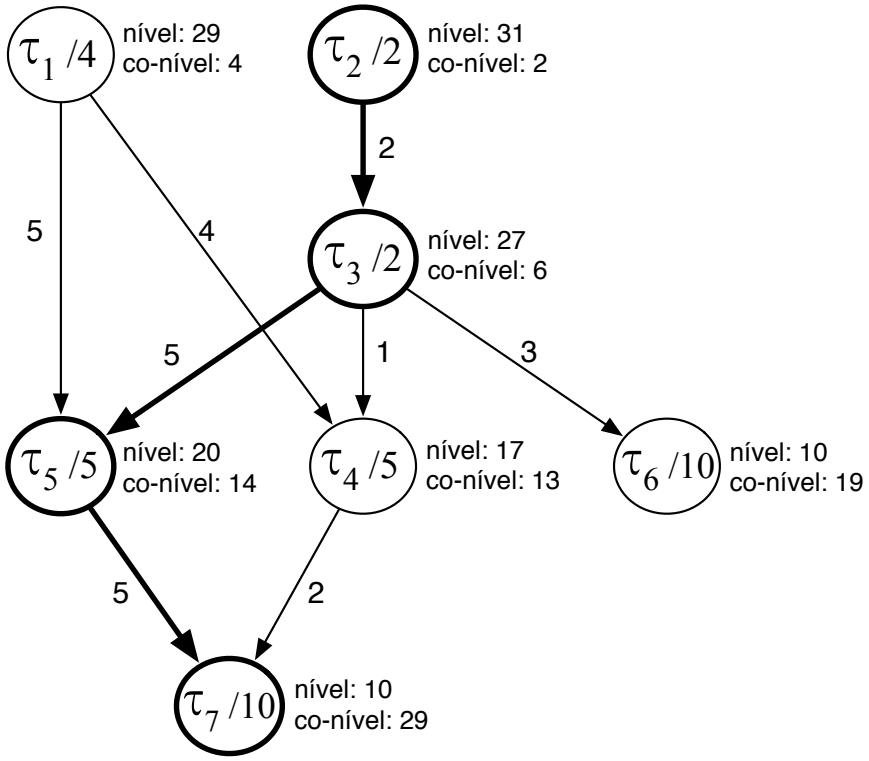


Figura 3: DAG com o nível e co-nível de cada tarefa anotado.

saída τ_i em G é $|\tau_i|$, e o *nível* das demais tarefas τ_i consiste no comprimento do caminho mais longo a partir de τ_i até uma tarefa de saída em G. O *co-nível* é definido de maneira semelhante, porém as referências são as tarefas de entrada. Assim, o *co-nível* de uma tarefa de entrada τ_i em G é $|\tau_i|$, e o *co-nível* das demais tarefas τ_i consistem no comprimento do caminho mais longo a partir de $|\tau_i|$ até uma tarefa de entrada em G. A Figura 3 mostra um DAG onde cada tarefa é anotada com um identificador e um custo de processamento (no formato $\tau_{identificador}/custo$), e também com seu *nível* e *co-nível*, ao lado direito de cada vértice. O cálculo dos atributos consideram os custos de comunicação anotados junto às arestas. O caminho crítico é destacado na Figura 3 pelos vértices e arestas com traço mais grosso.

Diversos algoritmos de escalonamento de lista são baseados nesses atributos básicos de prioridade. Os algoritmos descritos a seguir utilizam esses atributos para ordenar a lista de prioridades de maneira crescente.

- HLFET (*Highest Levels First with Estimated Times*): esse algoritmo constrói a lista de prioridades ordenando-a de forma crescente pelo *nível* das tarefas (HU, 1961). Um escalonamento gerado por esse algoritmo é chamado de *A-Schedule* (COFFMAN JR; DENNING, 1973);

- HLFNET (*Highest Levels First with No Estimated Times*): esse algoritmo considera que todas as tarefas possuem o mesmo custo de processamento. A partir daí o algoritmo funciona exatamente como o HLFET. Esse algoritmo pode ser usado onde o HLFET é desejado mas o custo das tarefas é conhecido antes do início da execução do programa.

- RANDOM: esse algoritmo, como o próprio nome sugere, atribui prioridades randômicas para as tarefas. Escalonamentos randômicos e ótimos são usados como parâmetros para medir a qualidade dos escalonamentos gerados por outros algoritmos.

- SCFET (*Smallest Co-levels First with Estimated Times*): esse algoritmo considera o valor negativo do *co-nível* da tarefa como atributo de prioridade.

- SCFNET (*Smallest Co-levels First with No Estimated Times*): esse algoritmo, de maneira similar ao HLFNET, considera que todas as tarefas tenham o mesmo custo de processamento e aplica o SCFET.

2.5 Escalonamento dinâmico

A estratégia básica de algoritmo de lista também é aplicada na literatura em ambientes dinâmicos, onde as tarefas da aplicação vão sendo conhecidas durante a execução do programa. Como o algoritmo não tem acesso ao grafo completo antes do início da aplicação, atributos como o *nível* da tarefa no grafo não podem ser aplicados nesse contexto. Além da falta de conhecimento da estrutura do grafo, muitas vezes os algoritmos *online* não possuem conhecimento dos custos das tarefas a serem escalonadas. Todas essas limitações impedem os algoritmos *online* de garantir escalonamentos eficientes.

Certos paradigmas de escalonamento *online*, como o *online-time*, obrigam o escalonador a determinar o processador em que uma tarefa irá executar, tão logo essa tarefa esteja disponível no sistema (seja porque foi criada ou porque suas dependências foram satisfeitas). Embora não seja possível determinar o momento em que novas tarefas ficarão executáveis, em algoritmos do tipo *online-time* assume-se que quando uma tarefa fica executável é possível saber seu custo de processamento. No contexto de sistemas operacionais ou em ambientes de execução *multithread*, por exemplo, frequentemente nenhuma informação sobre os custos das tarefas é dada ao algoritmo de escalonamento. Algoritmos *online* deste tipo são chamados *não-clarividentes* (*nonclairvoyant*). Em cenários de execução onde não são permitidas preempções e as tarefas possuem custos arbitrários, geralmente um exemplo trivial pode demonstrar

que algoritmos *online* do tipo *online-time* e *não-clarividentes* produzem escalonamentos distantes do ótimo (LEUNG, 2004).

2.5.1 Custo de processamento como atributo de prioridade

Diversos algoritmos, *online* e *offline*, presentes na literatura utilizam o custo de processamento das tarefas como seu atributo de prioridade. Os dois principais algoritmos dessa classe são listados a seguir.

- LPT (*Largest Processing Time*): o algoritmo LPT, introduzido por Graham (1969), o ordena a lista de prioridades utilizando como atributo de prioridade o valor negativo do custo de processamento (a lista de prioridades é ordenada de forma crescente). Assim, quanto maior o custo de processamento da tarefa, maior sua prioridade de execução.
- SPT (*Shortest Processing Time*): este algoritmo ((CONWAY; MAXWELL; MILLER, 1967)) é o oposto do algoritmo LPT; a lista de prioridade é construída considerando o tempo de processamento das tarefas como atributo de prioridade.

As estratégias citadas acima são usualmente aplicadas no escalonamento *online* de tarefas sem relações de precedência, e podem ser aplicadas tanto estática quanto dinamicamente.

2.5.2 Tempo de lançamento como atributo de prioridade

Uma vez que em diversos ambientes *online* o escalonador é não-clarividente, ou seja, não possui nenhum conhecimento prévio sobre as tarefas a serem escalonadas, um dos poucos dados disponíveis para o cálculo de prioridades é o tempo de lançamento da tarefa. Como consideraremos aqui somente escalonamentos que não utilizam preempção, o algoritmo de escalonamento mais conhecido neste cenário é o FCFS (*First Come, First Served*). Neste algoritmo a lista de prioridades simplesmente é ordenada considerando o instante de tempo em que as tarefas ficaram prontas para execução como seu atributo de prioridade. Para classificar tarefas lançadas no mesmo instante, heurísticas podem ser utilizadas de forma arbitrária. Esta estratégia é também conhecida como FIFO (*First In, First Out*) e é aplamente aplicado em sistemas simples, onde não há relações de precedência entre as tarefas.

2.6 Conclusão

Diversas estratégias podem ser aplicadas para escalonar um grafo de tarefas sobre m processadores idênticos, tanto estática quanto dinamicamente.

Dependendo da quantidade de informação que se tem sobre a estrutura do programa, o algoritmo de escalonamento pode se valer de mais ou menos dados para construir a lista de prioridades, peça fundamental nas decisões de escalonamento. Em geral, com mais dados de entrada pode-se construir listas de prioridades que forneçam escalonamentos melhores, o que neste trabalho significa minimizar o tempo total de execução da aplicação.

Ambientes multithread dinâmicos são, em geral, não-clarividentes, isto é, não possuem nenhum conhecimento prévio sobre a aplicação a ser escalonada. Neste tipo de ambiente o escalonador constrói a lista de prioridades conforme o programa evolui, utilizando um conjunto de dados bastante reduzido, composto pelos tempos de criação dos threads e pelas dependências já resolvidas no grafo, isto é, os antecessores das tarefas conhecidas até o momento. Considerando tais características podemos aplicar algoritmos de lista como o FCFS, o SCFNET e, é claro, o RANDOM no escalonamento de aplicações multithread dinâmicas. Caso seja possivel conhecer os custos de processamento das tarefas em um thread, estes custos podem ser utilizados na atribuição de prioridades. Neste caso algoritmos como o SPT, o LPT e o SCFET podem ser aplicáveis no escalonamento de programas multithread dinâmicos.

Interfaces de programação multithread, em geral, não fornecem recursos para a especificação de custos, principalmente em função da complexidade que a descrição de custos agrega à programação das aplicações. Desta forma, para poder fornecer garantias de desempenho, algumas ferramentas de programação limitam sua interface à descrição de aplicações regulares. Uma vez que o ambiente de execução pode inferir a estrutura do grafo da aplicação, heurísticas podem ser aplicadas para otimizar o escalonamento. Ambientes dinâmicos que permitem a programação de aplicações que geram não só cargas de trabalho irregulares, mas um grafo de dependências complexo, possuem uma aplicabilidade maior, porém precisam de estratégias de escalonamento mais genéricas. Ainda assim, heurísticas de escalonamento podem ser definidas de forma que o programador possa se utilizar delas para extrair um desempenho melhor do ambiente.

O capítulo seguinte introduz conceitos de ambientes de programação e execução multithread dinâmica, bem como discute os pontos de escalonamento do modelo multithread e o emprego algoritmos de escalonamento de lista nas decisões do ambiente. Como veremos, o fato de o escalonador não conhecer todo o grafo da aplicação limita as estratégias de escalonamento de lista que podem ser empregadas em ambientes multithread dinâmicos, fazendo com que técnicas adicionais precisem ser empregadas em sua implementação para reduzir os sobrecustos adicionados nas operações de escalonamento.

3 MODELO MULTITHREAD

A programação concorrente nos permite desenvolver programas compostos por mais de um fluxo de instruções. Cada fluxo de execução necessita de recursos de processamento para sua execução, como por exemplo, um processador e uma porção da memória. Logo um programa concorrente é assim chamado por sua possibilidade diversos fluxos que *concorrem* pelo acesso aos recursos de hardware. Esses fluxos concorrentes devem se comunicar ou sincronizar em algum ponto da execução. Assim, um ambiente de execução deve coordenar o acesso concorrente aos recursos de hardware e a comunicação entre os fluxos de execução, podendo incluir uma política de escalonamento que priorize a execução de certos fluxos em detrimento de outros.

O modelo *multithread* (VALIANT, 1990) define programas concorrentes em termos de fluxos de execução chamados *threads*, os quais se comunicam por meio de memória compartilhada. Uma interface de programação para um ambiente *multithread* deve prover primitivas para a criação e a sincronização dinâmica de threads, bem como maneiras de acessar a memória compartilhada de maneira segura. Considerar a memória compartilhada como meio de comunicação entre os threads implica que os custos de comunicação serão considerados desprezíveis nos algoritmos de escalonamento estudados neste trabalho.

Um thread encapsula uma sequência de tarefas do modelo de fluxo de dados (como definido na Seção 2.1), sendo que, no modelo multithread, as tarefas são delimitadas por pontos de escalonamento: operações de criação/sincronização de threads e o término de um thread. Assim como no modelo de fluxo de dados, nos pontos de escalonamento ocorrem operações de entrada e saída de dados sobre as tarefas envolvidas. Threads podem ser criados e destruídos durante a execução de um programa multithread, por meio de chamadas a primitivas do tipo *fork* e *join*, respectivamente.

No modelo multithread considerado neste trabalho, toda a sincronização entre threads acontece apenas durante as chamadas às operações *fork* e *join*. Toda

a comunicação de dados também se dá via tais primitivas, acompanhadas de parâmetros transferidos entre os threads. Monitorando as chamadas a *fork* e *join* é possível extrair um comportamento determinístico na execução de um programa *multithreaded*, o que nem sempre é possível utilizando mecanismos de sincronização como semáforos e barreiras. Por meio das dependências introduzidas nas chamadas a *fork* e *join* também é possível construir um grafo relacionando os threads criados pelo programa. Esse grafo é utilizado pelos algoritmos de escalonamento para atribuir prioridades de execução aos threads, de modo a otimizar algum índice de desempenho. A redução do tempo total de execução do programa é uma das otimizações buscadas por algoritmos de escalonamento e o objetivo principal dos ambientes de execução estudados neste trabalho.

Este capítulo tem como objetivo apresentar conceitos do modelo multithread e a forma como estes conceitos são modelados em um grafo que pode ser manipulado por algoritmos de escalonamento de lista. O restante do capítulo é estruturado como segue. A Seção 3.1 apresenta os conceitos de processo e thread no contexto de um sistema operacional, bem como os diversos tipos de implementação de threads nesses sistemas. Na Seção 3.2 é apresentada uma abstração de grafo que descreve um programa no modelo multithread, enquanto que na Seção 3.3 exploramos como as estratégias básicas de escalonamento de lista podem utilizar os dados oriundos deste grafo. Considerações finais sobre o capítulo são tecidas na Seção 3.4.

3.1 Modelos de implementação de threads

A maneira como threads são gerenciados e executados sobre o hardware subjacente são dependentes de decisões tomadas tanto na implementação da biblioteca que fornece suporte a threads em nível de usuário quanto na implementação de threads fornecida no núcleo do sistema operacional. Esta seção introduz aspectos práticos de *processos* e *threads*, bem como apresenta modelos para implementação de threads em sistemas operacionais.

3.1.1 Processos e threads

Um *processo* é uma abstração de um programa em execução, que permite a *multiprogramação* dentro e fora do núcleo de um sistema operacional. Um processo consiste basicamente do estado da unidade central de processamento (CPU – *Central Processing Unit*) a pilha do núcleo, e o *contexto* do processo, o qual consiste de um diretório de trabalho, descritores para os arquivos abertos por aquele processo, identificadores do usuário e do grupo, um mapa de

memória, entre outros sinais. O estado da CPU inclui um ponteiro para a pilha de chamadas do programa (SP – *Stack Pointer*), o contador de programa (PC – *Program Counter*) e o conteúdo dos registradores que armazenam dados locais e globais.

A concorrência em nível de sistema operacional é geralmente obtida quando temos diversos processos aptos a executar simultaneamente. O núcleo do sistema operacional é quem escalona os processos, isto é, define qual processo poderá utilizar o processador em cada fatia de tempo. Este modelo pode ser estendido a um multiprocessador, onde cada processador pode executar um processo diferente, gerando, a partir da concorrência disponível no sistema, uma execução paralela. Quando o sistema operacional retira o processador de um processo P e cede este processador para outro processo P' há um *chaveamento de contexto*, onde o estado do processador em questão é armazenado no descritor do processo P e o descritor do processo P' é carregado neste processador. Contudo, chaveamentos de contexto em processos implicam em grandes sobrecustos (*overheads*) de execução, os quais se somam o tempo de processamento real das aplicações.

Um *thread*, por outro lado, é um "processo leve", um fluxo de execução individual de um processo. Múltiplos threads podem coexistir em um mesmo processo, compartilhando seu espaço de memória, descritores de arquivos, código e dados globais. O *contexto de thread* consiste em um conjunto de dados reduzido em relação ao contexto de um processo, sendo composto apenas por um contador de programa, uma pilha de execução, um ponteiro para a pilha, registradores de propósito geral e outros sinais utilizados no gerenciamento de threads.

A programação *multihtread* consiste na descrição da concorrência presente nas aplicações pela criação de múltiplos threads ao invés de processos. O *multithreading* (VALIANT, 1990) fornece ganhos de desempenho, em geral, superiores à multiprogramação, uma vez que o escalonamento de threads é um procedimento muito menos custoso em relação aos chaveamentos de contextos de processo. Esse escalonamento de threads pode ainda ser feito somente pelo sistema operacional ou também em nível de usuário, sendo que diferentes tipos de informações são manipuladas em cada nível de escalonamento.

3.1.2 Threads de sistema

Sistemas operacionais modernos são usualmente implementados com threads de sistema (*kernel threads*), criados para executar diversas tarefas como, por exemplo, a execução de programas de usuário, o escalonamento de outros threads, o tratamento de faltas de páginas de memória, etc. Programas de

usuário que necessitem acessar algum serviço destes precisam executar uma chamada de sistema. Tais chamadas fazem com que a execução acesse o espaço de execução do kernel, realizem sua tarefa, e retornem ao espaço do usuário, procedimento que gera sobrecustos adicionais para a aplicação.

Existem implementações de threads de sistema amplamente aceitos, como Solaris Threads (POWELL et al., 1991) e PThreads (modelo POSIX[1] para threads) (NICHOLS; BUTTAR; FARRELL, 1998), que definem diversos conceitos utilizados e especializados por ferramentas que serão discutidas neste trabalho. As principais características observadas nestes modelos são:

- *Criação*: Threads podem ser criados em qualquer momento da execução de um programa. Uma função definida pelo usuário descreve um trecho de código que será executado inicialmente pelo thread. A criação de threads envolve a alocação de espaço para o descritor do thread, geralmente contendo ponteiros para o thread pai e para os filhos (entre outras informações), e para sua pilha de execução.

- *Destruição*: Após terem concluído, threads têm seus descritores desalocados e seus eventuais valores de retorno armazenados, sendo o thread pai notificado do término.

- *Independência*: Um thread pode continuar a existir mesmo após o thread que o criou terminar. Um processo com múltiplos threads acaba quando seu thread principal termina, ou somente após o término de todos seus threads, caso sejam executadas operações de *espera* pelo término de todos os threads criados durante a execução.

- *Bloqueio*: Threads podem bloquear sua execução voluntariamente ou terem sua execução bloqueada pelo escalonador a qualquer momento da sua execução. Um thread bloqueado pode “dormir” por uma certa quantidade de tempo ou esperar até que uma condição especifica seja atingida, sendo que após esta condição ser atingida os threads bloqueados podem ser reescalonados.

- *Sincronização*: Uma vez que threads podem compartilhar código e dados, o acesso simultâneo de threads a áreas de memória que podem ser alteradas pode resultar em não determinismo ou inconsistência de dados. Ferramentas devem fornecer mecanismos de sincronização como *mutexes*, semáforos, variáveis de condição, ou algum mecanismo do tipo, para que os dados compartilhados possam ser acessados de maneira ordenada.

[1] http://standards.ieee.org/develop/wg/POSIX.html

3.1.3 Threads de usuário

Threads em nível de usuário, diferente dos threads de sistema, são criados para extrair a concorrência dentro de uma única aplicação. Neste tipo de thread o escalonamento não envolve acesso ao núcleo, o que gera poucos sobrecustos de execução. Isto é especialmente interessante aplicações com paralelismo de grão fino, onde uma quantidade significativa de operações de escalonamento são realizadas durante a execução de um programa.

A maneira como o suporte *multithread* é oferecido pelo sistema operacional influi diretamente no modelo de threads oferecido ao programador em nível de aplicação e, consequentemente, na forma como threads serão escalonados em ambos os níveis (CAVALHEIRO, 2001). Quando threads são providos por bibliotecas externas, como é o caso de Athreads (CAVALHEIRO et al., 2007), o escalonamento é feito pela biblioteca em nível de usuário sobre os threads de sistema criados pelo ambiente de execução. Já quando se utiliza o suporte nativo do sistema operacional para a criação de threads, estes podem ser escalonados diretamente sobre a arquitetura física. A seguir são apresentados os modelos básicos de implementações para threads e os mesmos são ilustrados na Figura 4, onde “TU” indica um thread de usuário e “TS” indica um thread de sistema.

3.1.3.1 Modelo M:1

O modelo M:1 (muitos para um) mapeia vários threads de usuário em um thread de sistema, como mostra a Figura 4(a). Neste modelo, o escalonamento aplicativo é feito com baixo custo pela biblioteca que provê suporte a threads. Contudo, se um thread de usuário faz uma chamada de sistema bloqueante, isto causa o bloqueio do processo inteiro, pois o núcleo do sistema operacional só “enxerga” um thread por vez. Isso impede que outros threads de usuário não bloqueados usem a fatia de tempo disponível para o processo. Além disso, este modelo não permite execução paralela.

3.1.3.2 Modelo 1:1

No modelo 1:1 (um para um), cada thread de usuário é mapeado em um thread de sistema, como ilustrado na Figura 4(b). Esse modelo provê concorrência máxima, uma vez que cada thread de usuário será escalonado diretamente em um processador através do thread de sistema. Além disso, diferente do modelo M:1, uma chamada bloqueante feita por um thread de usuário não bloqueia o programa como um todo, dado que o processo ainda pode possuir threads de sistema desbloqueados. Contudo o escalonamento deste modelo é feito diretamente pelo sistema operacional, um ponto negativo quando

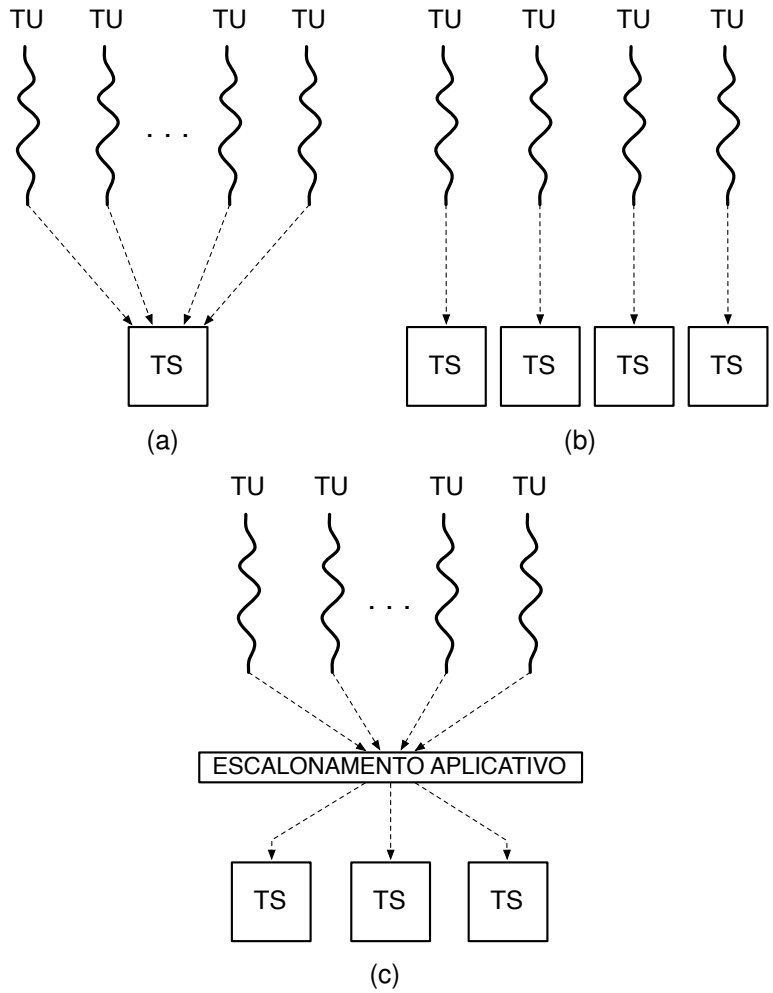


Figura 4: Modelos de implementação de threads: (a) modelo M:1; (b) modelo 1:1; (c) modelo N:M.

desejamos ter o controle sobre o escalonamento de threads dentro de uma aplicação. Além disso, a maioria dos sistemas que implementam este modelo limita o número de threads que podem ser criados, uma vez que o sobrecusto de criar e manipular threads de sistema é alto.

3.1.3.3 Modelo M:N

No modelo M:N (muitos para muitos) m threads de usuário são escalonados sobre n threads de sistema, como mostra a Figura 4(c). Neste modelo há dois níveis de escalonamento: o escalonamento que a biblioteca realiza dos m threads de usuário sobre os n threads de sistema (criados pelo ambiente de execução) e o escalonamento que o sistema operacional realiza dos n threads de sistema sobre os recursos de hardware disponíveis. Esse modelo é bastante flexível e transpõe os problemas dos outros dois, uma vez que o programador pode expressar toda a concorrência presente na lógica da aplicação e o ambiente de execução pode ajustar o número de threads de sistema de acordo com o

paralelismo que deseja extrair da arquitetura (dentro dos limites da mesma). Além disso, quando um thread de usuário realiza uma chamada bloqueante o escalonador aplicativo pode fazer o chaveamento de thread necessário no nível de usuário, sem reduzir a concorrência em nível de sistema.

Uma vez que este trabalho avalia a aplicação de algoritmos de lista no escalonamento de threads em nível aplicativo, os conceitos que serão introduzidos a seguir se referem a threads como aqueles criados pela aplicação, no nível de usuário. Neste nível, os "processadores" enxergados pelos algoritmos de escalonamento geralmente são implementados como threads de sistema. Todos os aspectos de implementação, bem como os ambientes de execução que serão apresentados no restante deste trabalho, consideram o modelo N:M descrito acima.

3.2 Grafos para a descrição de programas multithreaded

Um programa multithread pode ser descrito em termos de suas criações e sincronizações de threads por meio de um Grafo Dirigido Cíclico – DCG (*Directed Cyclic Graph*), onde os nós representam threads e as arestas representam operações de criação e sincronização de threads. O modelo de programa descrito permite a comunicação entre threads apenas no momento da criação de um novo thread ou na sincronização de um ponto da execução de um thread com o final de outro. Assumindo-se que a comunicação entre threads se dá apenas via pilha de chamada, um thread pode ser representado por uma função $y = F(x)$, de maneira semelhante à representação de uma tarefa no modelo de fluxo de dados (vide Seção 2.1). Quando F termina, seus resultados podem ser recebidos por outros threads. No entanto, diferentemente do modelo de fluxo de dados, um thread pode executar instruções do tipo *fork* em seu fluxo para criar novos threads, que estarão prontos para executar imediatamente após sua criação. Threads também podem executar instruções do tipo *join* sobre outros threads para esperar o término da execução desses threads. Assim o thread que executa o *join* pode ficar bloqueado a espera do término de outro caso este não tenha sido completamente executado.

Cada thread de um programa multithreaded define uma sequência de tarefas delimitadas pelas chamadas às operações *fork* e *join*. A Figura 5 apresenta três abstrações para um mesmo programa multithreaded. Na Figura 5(a) vemos a representação completa, onde os círculos representam tarefas e os retângulos tracejados representam os threads; uma aresta que liga uma tarefa ao topo do retângulo que representa um thread Γ_i indica uma operação *fork*, onde a tarefa em questão termina após criar o thread Γ_i; uma aresta que liga a base de um

retângulo que representa um thread Γ_i a uma tarefa em outro thread indica uma operação *join*, onde a tarefa em questão só pode ser executada no contexto de seu thread após o término do thread Γ_i. A Figura 5(b) exibe somente o DAG da aplicação, onde os círculos representam as tarefas e as arestas representam dependências de dados entre essas tarefas. Na Figura 5(c) é exibido somente o DCG, onde os retângulos representam threads, arestas sólidas representam operações *fork* e arestas pontilhadas representam operações *join*. Uma vez que a unidade de escalonamento em um ambiente multithread é o thread, somente o nível apresentado na Figura 5(c) é considerado pelos algoritmos de escalonamento desses ambientes. Com base na representação apresentada na Figura 5(c), podemos dizer que o grafo gerado por um programa *multithreaded* é dirigido e possui ciclos, pois o fluxo de execução pode sair de um thread e voltar para o mesmo.

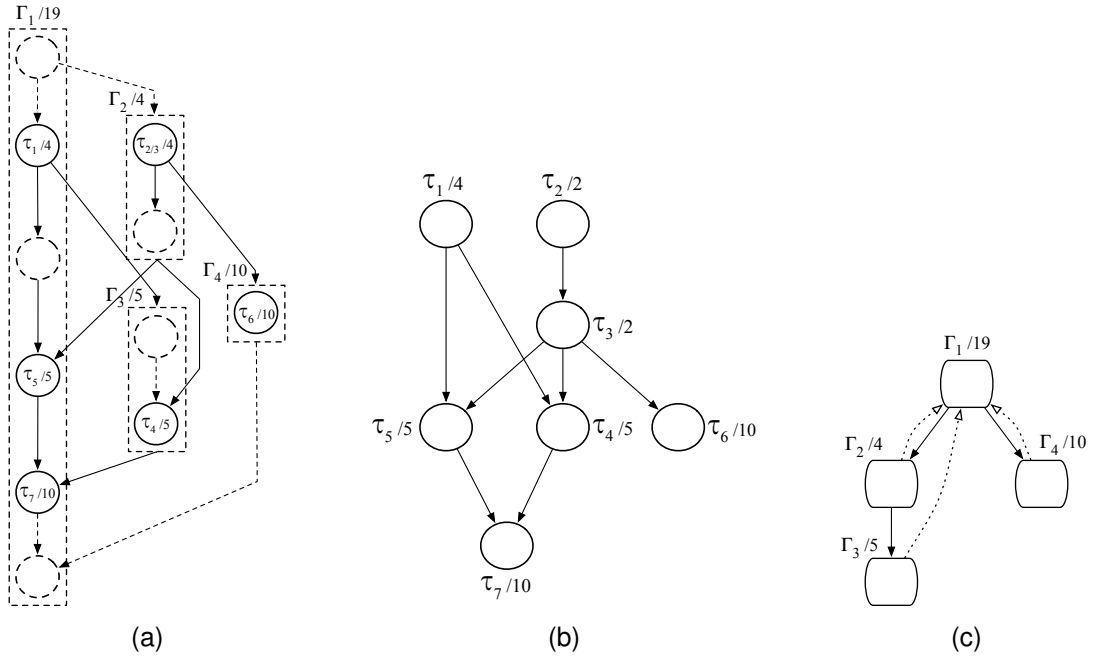


Figura 5: Três representações para um mesmo programa multithreaded: o programa completo, com tarefas e threads, o DAG de tarefas e o DCG de threads.

Definição 2 *Um DCG é uma tupla $C = (N, E)$ que representa um programa P. Nessa tupla, N corresponde ao conjunto de n nós e E ao conjunto de arestas em C. Cada nó em N recebe uma identificação i, onde $1 \leq i \leq n$, representando o thread Γ_i no grafo: $N = \{\Gamma_1, \Gamma_2, \ldots \Gamma_n\}$. Uma ordem de precedência entre os threads Γ_i e Γ_j é representada por $\Gamma_i \rightarrow \Gamma_j$, sendo que $E = \{\Gamma_i \rightarrow \Gamma_j | 1 \leq i, j \leq n, i \neq j\}$. Uma aresta dirigida de Γ_i para Γ_j existe se o programa P descreve que Γ_i cria Γ_j ou que Γ_j sincroniza com o término de Γ_i. Tal aresta representa que, ou Γ_j pode ser lançado depois de ser criado por Γ_i ou em algum ponto*

da execução de Γ_j precisa-se que Γ_i tenha terminado. Nesta representação, nós e arestas podem conter pesos: $|\Gamma_i|$ e $|\Gamma_i \rightarrow \Gamma_j|$ correspondem ao custo computacional associado à execução do thread Γ_i e ao custo de transferência de dados associado à $\Gamma_i \rightarrow \Gamma_j$, respectivamente.

Por definição, um DCG pode conter, ao mesmo tempo, $\Gamma_i \rightarrow \Gamma_j$ e $\Gamma_i \rightarrow \Gamma_k$, representado por $\Gamma_i \rightarrow \Gamma_{j,k}$, bem como $\Gamma_i \rightarrow \Gamma_j$ e $\Gamma_k \rightarrow \Gamma_j$, representado por $\Gamma_{i,k} \rightarrow \Gamma_j$. O primeiro ($\Gamma_i \rightarrow \Gamma_{j,k}$) representa uma operação de criação de múltiplos threads, enquanto o segundo ($\Gamma_{i,k} \rightarrow \Gamma_j$), uma operação de sincronização de múltiplos threads em um mesmo ponto da execução. Outra característica do modelo explorado neste trabalho é que o programa começa e termina pela execução do thread Γ_1, que representa o fluxo de execução principal do programa, que começa pela execução da função `main` em um programa C/C++, por exemplo.

3.2.1 Semântica das primitivas *fork* e *join*

Cada interface de programação possui sua própria nomenclatura para as operações de criação e sincronização de threads. Neste trabalho nos referiremos à operação de criação de threads como *fork* e à operação sincronização de threads como *join*. Para entender o significado destas operações em um ambiente multithread, consideremos um processador P executando uma tarefa τ_a em um thread Γ_i. Quando o fluxo de instruções do thread Γ_i encontra uma chamada do tipo **fork**(Γ_j) a tarefa τ_a termina e agora Γ_j pode ser executada concorrentemente com Γ_i. Nesse momento o ambiente de execução deve escolher se P deverá continuar a execução de Γ_i ou iniciar a execução de Γ_j. Suponhamos que P continue a execução de Γ_i: essa continuação partirá de uma tarefa τ_b, sucessora de τ_a neste thread. Suponhamos agora que nas instruções da tarefa τ_b encontremos uma chamada do tipo **join**(Γ_j). Neste instante τ_b termina e Γ_j pode ou não ter sido completamente executada. No primeiro caso uma nova tarefa τ_c, sucessora de τ_b no thread Γ_i, é criada para processar os eventuais dados produzidos durante a execução de τ_b e do thread Γ_j (isto é, pela última tarefa no fluxo de Γ_j). Caso Γ_j não tenha terminado no momento do *join*, Γ_i é bloqueada e o processador P é liberado para que outro thread possa ser executado. Eventualmente o thread Γ_j terá sido completamente executado e a execução do thread Γ_i será continuada sobre o processador P ou algum outro processador disponível, caso o ambiente de execução possibilite a migração de threads entre os processadores.

3.3 Escalonamento de lista em ambientes multithread

Como visto na Seção 2.4, algoritmos de lista podem ser aplicados em ambientes dinâmicos, com a penalidade de terem que atribuir prioridades de execução a uma certa tarefa sem possuir o conhecimento das tarefas que utilizarão seus resultados. O mesmo acontece quando aplicamos o escalonamento de lista em ambientes multithread dinâmicos. Neste tipo de ambiente cada thread possui uma sequência de tarefas de forma que, caso este thread esteja apto a ser executado ou esteja em execução, somente uma de suas tarefas estará ativa. Chamaremos esta tarefa de *tarefa atual* do thread.

Para que possamos aplicar uma estratégia básica de escalonamento de lista (como a apresentada na Seção 2.3) precisamos de uma política para atribuição de prioridades para os threads. Contudo, os dados que possuímos para o cálculo de atributos de prioridade em uma execução dinâmica são limitados. Por exemplo, consideremos o pseudo-programa multithreaded presente na Figura 6. A Figura 7 mostra os estados do grafo durante a execução do programa da Figura 6 a cada ponto de escalonamento: em cada subfigura, para cada thread apto a executar, a sua tarefa atual está em destaque, desenhada com uma linha mais grossa; cada tarefa é indicada na figura da seguinte forma τ_{id}/n, onde id é um identificador único para a tarefa e n é igual ao *co-nível* da tarefa; todas as tarefas possuem custo 1 (um). O programa inicia executando τ_a, a primeira tarefa do thread inicial (raiz) do programa, indicado na Figura 7(a) por Γ_1. A Figura 7(a) representa o estado do grafo logo após a execução do comando `fork(`Γ_2`)`, no programa da Figura 6. Neste momento, a tarefa atual de Γ_1 passa a ser τ_b, e τ_c passa a ser a tarefa atual de Γ_2. Como estas duas tarefas possuem todas as suas dependências satisfeitas neste momento, é possível calcular seu *co-nível*. Quando o comando `join(`Γ_2`)` é executado, a tarefa τ_d só é criada depois que Γ_2 foi completamente executado, o que já ocorreu no momento ilustrado pela Figura 7(c). Neste momento τ_d é a tarefa atual de Γ_1 e seu *co-nível* pode ser calculado, uma vez que todos os seus antecessores já foram executados.

Como pode ser observado nesse exemplo, o co-nível da tarefa atual é um dos atributos de prioridade de execução para threads que pode ser derivado de um grafo construído em tempo de execução. Uma vez que queremos atribuir prioridades aos threads em um DCG e não a tarefas, podemos considerar o *co-nível* de um thread como sendo igual ao *co-nível* de sua tarefa atual. Desta forma podemos aplicar um algoritmo simples como o SCFNET no nível de threads. Podemos utilizar também a profundidade do thread no DCG como atributo de prioridade. A profundidade de um thread Γ pode ser definida pelo número de *forks* no caminho do thread principal até Γ.

```
1  void* foo(void* inputData) {
2    /* Tarefa C */
3  }
4
5  int main() {
6    /* Tarefa A */
7    Thread t;
8    t.startRoutine = foo;
9    t.inputData = /* some data */
10   fork(t);
11   /* Tarefa B */
12   join(t);
13   /* Tarefa D */
14 }
```

Figura 6: Exemplo básico de programa um programa multithread.

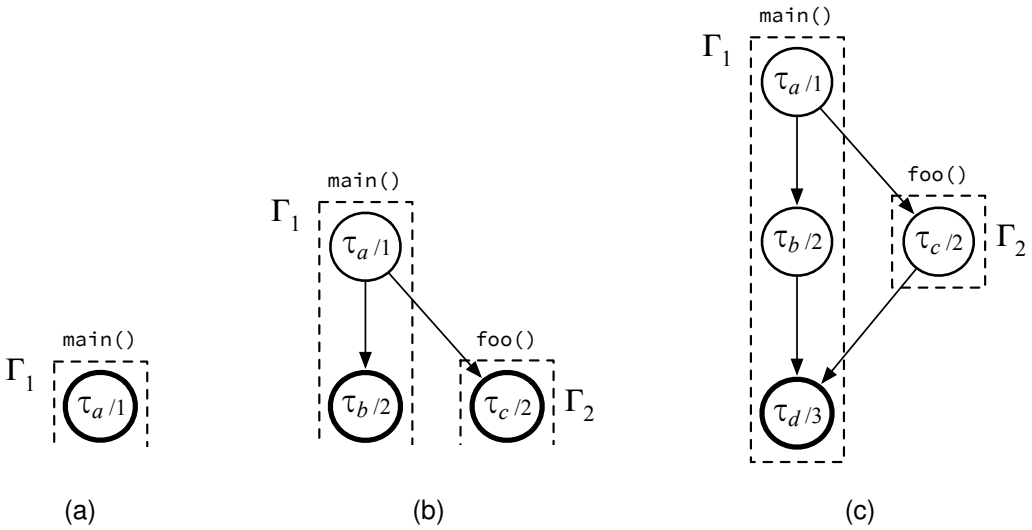


Figura 7: Estados do grafo durante a execução do programa da Figura 6.

Quando é possível saber os custos de cada thread do programa, podemos utilizar tais custos como atributos de prioridade para aplicar algoritmos como o LPT ou o SPT. Também podemos considerar o *co-nível* da tarefa atual como atributo de prioridade do thread e aplicar o algoritmo SCFET. Esses custos podem ser obtidos com uma pré-execução, com uma análise do código em tempo de compilação, ou mesmo por meio de interfaces de programação, onde o programador especificará o custo de cada trecho de código. Contudo, uma vez que as ferramentas de programação multithread visam simplificar ao máximo a programação com threads, não é comum encontrarmos APIs para a descrição de custos de tarefas. Desta forma, a maioria das ferramentas multithread aplica algoritmos de escalonamento considerando custos unitário para as unidades de trabalho.

Independente de conhecermos a estrutura do programa ou os custos das tarefas de um thread, podemos utilizar o algoritmo FIFO para escalonar os threads por ordem de chegada na lista. Além do algoritmo FIFO, podemos utilizar o algoritmo RANDOM, que atribui prioridades randômicas aos threads.

3.4 Conclusão

Este capítulo apresentou diversos conceitos a respeito do suporte multithread presente nos sistemas operacionais modernos e dos ambientes dinâmicos de execução multithread que executam em nível de aplicação. O modelo M:N estudado possibilita a implementação de ferramentas de programação, nas quais as aplicações simplesmente descrevem a concorrência das aplicações e o ambiente de execução fica responsável por explorar o paralelismo da arquietura subjacente, por meio de recursos do sistema operacional.

Introduzimos neste capítulo os DCGs, que descrevem as interações entre threads em um programa multithread, e que definem a representação manipulada pelos algoritmos de escalonamento em ambientes multithread. A dinamicidade de aplicações multithread nos permite extrair menos informações de um DCG em relação ao escalonamento *offline* de DAGs. Contudo, vimos que ainda assim é possível aplicar diversas estratégias de escalonamento de lista em ambientes multithread dinâmicos.

Em ambientes reais de execução multithread, as operações de escalonamento possuem custos que devem ser considerados e minimizados ao máximo na implementação desses ambientes. O capítulo seguinte aborda tais questões práticas e apresenta técnicas de implementação que incluem diferentes organizações arquitetônicas para as estruturas de dados do ambiente e diferentes implementações das operações *fork* e *join*.

4 IMPLEMENTAÇÃO DE AMBIENTES DE EXECUÇÃO MULTITHREAD

Em um ambiente multithread dinâmico, o algoritmo de escalonamento é ativado a cada ponto de escalonamento. Os pontos de escalonamento são momentos da execução de um programa multithreaded onde o ambiente atualiza o grafo de threads e decisões devem ser tomadas. Estes momentos acontecem quando threads são criados, bloqueados, ou terminam. Os sobrecustos de execução (*overheads*) em um ambiente multithread são os custos associados às computações realizadas por este ambiente nos momentos de sua inicialização e término, e em cada ponto de escalonamento. Os sobrecustos do ambiente de execução não dizem respeito à manipulação dos dados do programa concorrente que este executa, portanto é de extrema importância que as operações de escalonamento sejam executadas da maneira mais eficiente possível.

Por exemplo, consideremos o tempo de execução t_s que um programa sequencial P_s, o qual resolve o problema *x*, leva para executar em uma máquina m com um processador. Consideremos agora um programa multithread P_c, que também resolve *x*, sendo que P_c basicamente adiciona chamadas a P_s para expressar a concorrência da aplicação. P_c executará todas as mesmas computações que P_s para resolver o problema, mas também executará a inicialização do ambiente de execução multithread, as operações de escalonamento desse ambiente (o algoritmo de lista, as atualizações do grafo), e a rotina de finalização do ambiente de execução. Digamos que o tempo de execução de P_c sobre a mesma máquina m seja t_1. Podemos facilmente supor que o tempo t_1 será maior do que t_s. A diferença $t_1 - t_s$ representa o sobrecusto adicionado pelo escalonamento de threads em tempo de execução.

Além dos sobrecustos do ambiente de execução, diversos fatores podem influenciar no tempo total de execução de um programa multithreaded. Um destes fatores é a política usada para atribuir prioridades aos threads: diferentes políticas resultam em listas de prioridades diferentes, o que, em uma execução paralela, influencia diretamente no *makespan* gerado pelo

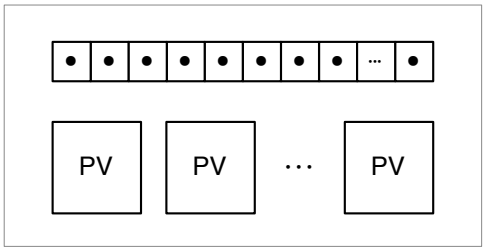
escalonamento (GRAHAM, 1976). Contudo, diferentes técnicas podem ser empregadas, tanto na arquitetura do ambiente de execução quanto na implementação das operações de escalonamento, resultando em mais ou menos sobrecustos adicionados à execução do programa. Esta seção se dedica a discutir alguns conceitos e estratégias de implementação de ambientes multithread, baseados tanto nas documentações das principais ferramentas da literatura quanto na experiência dos autores nas implementações do ambiente Anahy (CAVALHEIRO et al., 2007).

Neste capítulo assumimos um modelo de execução M:N (vide Seção 3.1.3.3), onde o escalonamento de M threads da aplicação é realizado sobre N threads no nível de sistema operacional, os quais chamaremos de Processadores Virtuais (PVs). O restante do capítulo está organizado como segue. Na Seção 4.1 discutiremos os impactos causados nas operações de escalonamento ao adotarmos diferentes distribuições para os PVs e lista(s) de threads, bem como alguns requisitos práticos de implementação para que cada arquitetura funcione adequadamente. Na Seção 4.2 estudaremos situações que ocorrem em cada ponto de escalonamento e as diferentes estratégias que podem ser tomadas. A Seção 4.3 aborda a questão do roubo de trabalho e o funcionamento básico dos algoritmos de lista sobre as configurações apresentadas na Seção 4.1. Na Seção 4.4 abordaremos a questão da granularidade de paralelismo, seu impacto no projeto de aplicações concorrentes, mas principalmente seu impacto sobre o ambiente de execução multithread. A Seção 4.5 conclui o capítulo.

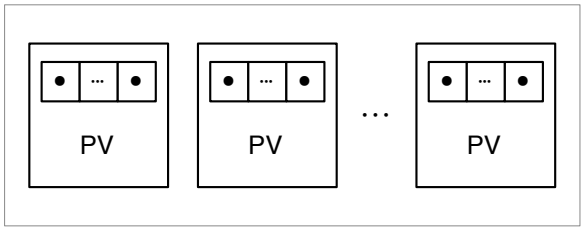
4.1 Arquitetura do núcleo

Diversas configurações podem ser adotadas na organização da arquitetura do núcleo de um ambiente de execução multithread. A Figura 8 apresenta três organizações básicas que podem ser observadas nas ferramentas mais proeminentes na indústria e na academia: implementação centralizada, distribuída e híbrida. Em todas as subfiguras, os retângulos menores, com um ponto no centro, representam threads prontos para a execução, enquanto os retângulos maiores, com a inscrição “PV”, representam processadores virtuais.

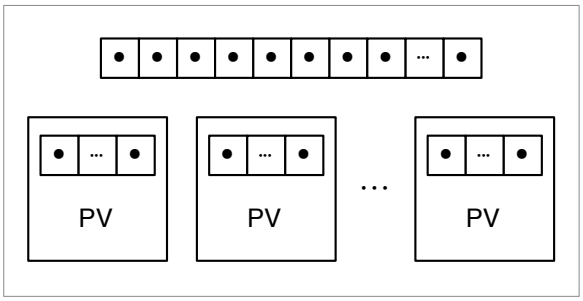
Na Figura 8(a) temos o modelo centralizado, no qual há apenas uma lista onde threads aptos a executar são inseridos e retirados. Para que seja mantida a integridade dessa lista, cada PV acessa a mesma em regime de exclusão mútua, ou seja, existe uma variável `v` do tipo `mutex` compartilhada entre todos os PVs, de forma que o PV que deseja acessar a lista deve primeiro conseguir executar um `lock` sobre `v`. Assim, é possível deduzir que se houverem muitos acessos concorrentes haverá muita contenção, isto é, como os acessos



(a)



(b)



(c)

Figura 8: Arquiteturas básicas de ambientes multithread: com lista centralizada (a); com listas distribídas (b); híbrido (c).

à lista devem ser ordenados, os PVs passarão uma quantia significativa do tempo total de execução esperando por outro PV que está acessando a lista. Athreads (CAVALHEIRO et al., 2007), uma implementação do modelo Anahy com API ao estilo POSIX, utiliza este modelo arquitetônico em seu núcleo.

A Figura 8(b) apresenta uma arquitetura distribuída[1] onde temos uma lista individual por PV, sendo que para cada lista existe uma variável do tipo `mutex` para controlar os acessos concorrentes dos PVs. Geralmente, em sistemas com este tipo de arquitetura cada PV cria threads apenas em sua lista, e só busca threads nas listas de outros PVs quando fica sem trabalho em sua própria lista. Em gera,l nessa técnica, chamada de *roubo de trabalho* (*work*

[1]O termo “arquitetura distribuída” é geralmente usado para designar um multicomputador, composto por várias máquinas interconectadas por meio de uma rede de comunicação. No entanto, neste trabalho o termo se refere a uma distribuição lógica de componentes sobre os processadores de uma única máquina.

stealing) (BLUMOFE; LEISERSON, 1994), a escolha do PV que sofrerá o roubo é feita randomicamente. Uma vez que somente o roubo de trabalho gera a possibilidade real de concorrência no acesso às listas, os threads roubados devem ser preferencialmente aqueles com maior probabilidade de gerar trabalho na lista do "ladrão", de forma a minimizar o número de roubos. A ferramenta Cilk (BLUMOFE; AL., 1995), hoje mantida pela Intel ® sob o nome de Cilk Plus (INTEL, 2012a), emprega este modelo arquitetônico com alta eficiência, porém sua eficiência se baseia em restrições severas no modelo de programa permitido pela ferramenta.

Quando consideramos esqueletos genéricos de paralelismo, certos aspectos do modelo arquitetônico distribuído podem complicar o balanceamento de carga entre os PVs. Por exemplo, consideremos o pseudo-programa da Figura 9. O paralelismo da aplicação em questão é baseado no dados do vetor `data`. Na prática, isto representa que em um programa sequencial a função `func` seria chamada diretamente no laço `for`, mas em um programa concorrente criaremos um thread para executar cada iteração do laço. Agora consideremos que este programa executará sobre um ambiente que implementa a arquitetura distribuída da Figura 8(b), onde cada PV deve criar threads apenas em sua própia lista. Neste caso, o PV que estiver executando o thread principal irá inserir todos os threads do vetor `threads` em sua própria lista, por meio das chamadas a `fork`. Assim a distribuição será prejudicada, pois muitos roubos de trabalho deverão ocorrer, e os trabalhos roubados não irão gerar filhos na lista do PV que os executa. Além disso, como somente um PV tem trabalho em sua lista, os PVs ociosos terão grande probabilidade de escolher outro PV ocioso para tentativas de roubo de trabalho.

Para transpor o cenário descrito no parágrafo anterior, algumas ferramentas, como o Intel ® Threading Building Blocks (TBB) (REINDERS, 2007), empregam um modelo arquitetônico híbrido, tal como ilustrado na Figura 8(c). Esse modelo é composto por listas individuais para cada PV, nas quais somente o dono pode inserir threads (podendo haver roubo), e uma lista compartilhada, onde todos os PVs podem inserir ou obter threads executáveis. Nesse tipo de ambiente a API deve também ser estendida para diferenciar a criação de threads na lista local da criação de threads na lista compartilhada.

4.2 Pontos de escalonamento

No modelo multithread considerado neste trabalho, tarefas são elementos atômicos: uma vez que um PV começa a executar uma tarefa τ no contexto de um thread, esse PV não poderá executar nenhuma tarefa de outro thread

```
1  void* func(void* inputData) {
2    /* process input data and return result */
3  }
4
5  int main() {
6    Thread threads[n];
7    void* data[n] = /* some data */
8
9    for (int i = 0; i < n; i++) {
10     /* set the thread up and fork it */
11     threads[i].startRoutine = func;
12     threads[i].inputData = data[i];
13     fork(threads[i]);
14   }
15   joinAll(threads);
16
17   /* process threads' output data */
18 }
```

Figura 9: Exemplo básico de um programa multithread.

até que τ termine. Contudo, no nível de threads, chaveamentos de contexto podem acontecer durante operações de escalonamento. Assim, em ambientes multithread dinâmicos as decisões devem ser tomadas a cada ponto de escalonamento. Estes pontos ocorrem em três momentos da execução: quando um thread cria outro, por meio da primitiva *fork*, quando um thread sincroniza com o fim de um outro, por meio da primitiva *join*, ou quando a execução de um thread acaba. Diferentes estratégias podem ser aplicadas em cada ponto de escalonamento, influenciando diretamente no escalonamento dos threads e, por consequência, no tempo total de execução das aplicações. A seguir iremos explorar cada ponto de escalonamento e apresentar algumas das estratégias cabíveis.

4.2.1 Criação de threads

No capítulo anterior introduzimos o conceito da *tarefa atual* de um thread, que consiste na tarefa que deve ser executada no thread que está pronto para execução ou a tarefa que está sendo executada quando o thread já está em execução. Quando um fluxo de execução se divide em dois (em uma operação *fork*), a tarefa atual do thread "pai" acaba. O thread "filho" é criado, e temos um nova tarefa para executar na continuação do pai e uma nova tarefa para executar no início do thread filho. A estratégia de escalonamento deve, neste momento, escolher qual thread/tarefa será executada após o *fork* pelo PV atual e qual será disponibilizada para que algum PV possa executar futuramente. Uma alternativa é comparar as prioridades dos threads pai e filho e escolher o de maior prioridade. Porém, é comum que os processadores possuam decisões estáticas

nesse ponto de escalonamento. Os dois modos de execução possíveis nesse caso são chamados de *Help-First* e *Work-First* (GUO et al., 2009). No modo *Help-First* o PV opta sempre por dar continuidade à execução do thread atual, o thread pai, e disponibilizar o thread criado na lista, seja ela global ou local. No modo *Work-First* o oposto ocorre: quando o PV encontra uma chamada do tipo *fork*(Γ), ele começa a executar o thread Γ imediatamente e disponibiliza o thread pai, parcialmente executado, para os outros PVs. O modo *Work-First* possui uma implementação um pouco mais complexa, dado que o estado do thread atual, incluindo seus resultados parciais, devem ser salvos no momento do *fork* e copiados para outro PV quando acontece um roubo de trabalho.

Assumindo um ambiente com listas distribuídas, além da escolha do thread que deve executado após um *fork*, o PV pode prever uma distribuição de trabalho já no momento da criação de threads, isto é, dependendo da carga de trabalho na lista local, escolher alguma outra lista para colocar o thread que não será executado imediatamente. O comum na maioria dos ambientes é, para cada primitiva de criação de threads, ter um lugar definido para inserir um thread. É o que acontece na ferramenta TBB, por exemplo, onde chamadas a `spawn` inserem o thread criado na lista local ao PV e chamadas a `enqueue` inserem o thread na fila global. Em OpenMP (CHANDRA et al., 2001), por outro lado, diferentes parâmetros passados para o pragma `parallel for`, por exemplo, fazem com que os threads gerados a partir de cada iteração (cada *chunk*, no dialeto de OpenMP) sejam criados em listas diferentes.

4.2.2 Sincronização e término de threads

Quando um PV está executando um thread Γ_i e encontra uma chamada a *join* sobre um thread não terminado Γ_j, a execução de Γ_i deve ser bloqueada até que Γ_j tenha terminado. Quando Γ_i é bloqueado o PV fica ocioso e deve executar o procedimento de busca por trabalho. Neste cenário observamos duas estratégias básicas geralmente adotadas pelas ferramentas: realizar uma busca normal, como a busca que o PV executa logo que é criado pelo ambiente de execução, ou realizar uma busca parametrizada pelo thread que causou o bloqueio. Nesse segundo caso, a política de prioridade do PV é alterada temporariamente de forma que a busca por trabalho irá priorizar threads que colaborem para o término de Γ_j (o PV possivelmente tentará executar o próprio Γ_j ou algum de seus descendentes). O intuito dessa estratégia é desbloquear Γ_i o quanto antes. A busca de forma parametrizada, em geral, pode ser mais custosa e, no caso de um ambiente com listas distribuídas, mais difícil de implementar de forma eficiente, pois as referências para os threads se encontram em listas diferentes e não são todas protegidas pelo mesmo *mutex*.

Independente do tipo de busca realizada após um *join* não satisfeito, iremos assumir que cada PV possui uma pilha de threads bloqueados de modo que, no momento em que o thread Γ_i foi bloqueado ao executar um *join* sobre Γ_j ele é empilhado pelo PV que o executava. Quando um PV termina a execução de um thread Γ qualquer, o estado de outros threads bloqueados previamente ao executar *join* sobre Γ deve ser alterado novamente para *executável*. Neste momento, o PV que terminou a execução de Γ pode ou não possuir um thread bloqueado no topo de sua pilha. Caso possua, a busca por trabalho pode ser, novamente, parametrizada por este thread. O PV pode, também, encontrar o thread no topo de sua pilha no estado executável e dar continuidade à execução desse thread. Quando a pilha está vazia, o PV busca trabalho da mesma maneira que buscou logo que fora criado.

4.2.3 Migração de threads

Em certos ambientes de programação, uma vez que um PV começa a executar um thread, somente o mesmo PV pode terminar de executar esse thread. Nesses ambientes dizemos que não há *migração de threads*. Em outros ambientes não existe essa ligação forte entre um thread em um PV, e threads podem começar a serem executados em um PV e terminar sua execução em outro. Nesse caso dizemos que a migração de threads é permitida.

Em PVs que implementam o modo *Work-First*, como em Cilk, assume-se que há migração, pois toda vez que é executado um *fork* o PV passa a executar o thread criado e libera o thread *parcialmente executado* para que outros possam executar. Já em PVs que implementam o modo *Help-First* pode ou não haver migração. A implicação disso se dá em um *join* não satisfeito: quando não há migração o PV possui uma pilha individual onde empilha as referências para os threads bloqueados; quando há migração, um thread que se torna executável novamente após ter sido bloqueado em um *join* deve ser escalonável sobre qualquer PV.

Quando existe migração, dados devem ser copiados de um PV para outro, gerando sobrecustos de execução e, geralmente, uma implementação mais complexa. Por outro lado, quando a migração é permitida o escalonamento fica mais flexível, pois qualquer thread, recém criado ou parcialmente executado, pode ser escalonado sobre qualquer PV. Na especificação do modelo Anahy, por exemplo, PVs devem implementar a política *Help-First* sem possibilidade de migração.

4.3 Estratégias de escalonamento

Como discutido na Seção 3.3, diversos dados podem ser derivados do grafo da aplicação para atribuir prioridade aos threads de um DCG, de maneira que eles possam ser organizados dinamicamente em uma lista para seja aplicado um algoritmo de escalonamento de lista. Contudo, quando estamos lidando com mais de uma lista de threads, como nos ambientes ilustrados nas figuras 8(b) e 8(c), necessitamos de alguma técnica como o *roubo de trabalho* para o balanceamento de carga. Quando um PV busca trabalho em uma lista que não a sua, as prioridades dos threads podem, assim como na busca parametrizada (vide seção anterior), ser tratadas de maneira diferente.

Blumofe e Leiserson (1994) introduziram a técnica de *roubo de trabalho* para realizar o balanceamento de carga em um ambiente com listas de threads distribuídas pelos PVs. Essa técnica nasceu durante o desenvolvimento do ambiente de execução da ferramenta Cilk (BLUMOFE; AL., 1995), onde não existem listas globais, cada PV possui sua própria lista e somente cria threads nela. Sempre que um processador fica ocioso ele busca em sua lista o thread criado há menos tempo. Caso não encontre nenhum, ele escolhe randomicamente uma vítima entre os demais PVs e tenta roubar um thread da lista do escolhido. Quando o ladrão acessa a lista de uma vítima ele sempre busca o thread criado há mais tempo, isto é, a prioridade dos threads é invertida durante o roubo. Essa inversão de prioridades é baseada na premissa de que um thread mais antigo manterá o PV que efetuou o roubo mais tempo ocupado, pois o thread roubado estará mais próximo à raiz do grafo e tem mais chances de criar filhos na lista local do PV "ladrão". No caso de Cilk, na maior parte das vezes esta premissa se confirma, pois a interface de programação limita o programador a descrever computações *fully-strict* (BLUMOFE; LEISERSON, 1994), modelo que privilegia a estratégia de roubo.

Quando se trata de um ambiente híbrido, onde temos uma lista global e listas locais aos PVs, o algoritmo de escalonamento deve prever ainda mais uma etapa na busca por trabalho: a busca na lista global. Nessa lista, o ambiente pode aplicar uma política de prioridade diferente das demais. A ferramenta TBB, por exemplo, emprega um modelo arquitetônico híbrido, onde os PVs aplicam uma política de prioridade semelhante à de Cilk ao buscar trabalho na lista local e ao roubar trabalho de outros PVs. Contudo, a lista global é escalonada por ordem de chegada dos threads (política FIFO/FCFS).

4.4 Granularidade do paralelismo

A decomposição do paralelismo é um fator que gera impactos sérios no desempenho de programas paralelos. Diversas estratégias podem ser tomadas no momento da divisão de tarefas em um programa paralelo. Por exemplo, em um ambiente de execução com p PVs, pode-se dividir o trabalho (quando possível) em p unidades e criar um thread para executar cada unidade. A quantidade de computação acumulada em cada unidade de trabalho é chamada de *granularidade*.

Consideremos uma máquina m com p processadores físicos executando um ambiente de execução multithread com p PVs. A Figura 10 demonstra duas estratégias opostas na divisão de trabalho em um programa que realiza a soma paralela dos valores dos vetores A e B, e armazena os resultado em um terceiro vetor C. Na Figura 10(a) podemos observar uma divisão de trabalho com a maior (mais grossa) granularidade possível. Em tal divisão um número mínimo de threads é gerado, isto é, a concorrência gerada é igual ao paralelismo do ambiente, o que resulta em um mínimo de sobrecustos adicionados pelo ambiente de execução. Contudo, esta abordagem não é nada flexível, pois a divisão de tarefas é feita para uma arquitetura específica: embora o desempenho seja o melhor possível com p processadores, a mesma aplicação não ganhará desempenho uma máquina com $p + 1$ processadores, por exemplo.

Considerando ainda a divisão de trabalho apresentada na Figura 10(a), suponhamos que cada unidade de trabalho gere uma carga diferente em sua execução (em função de uma instrução condicional, por exemplo) ou que a arquitetura possua processadores operando em diferentes velocidades. Nestes casos a distribuição da carga ficará desbalanceada em tempo de execução, e se considerarmos nossa máquina m, por exemplo, um processador poderia ficar boa parte do tempo ocioso enquanto o outro executa uma carga que poderia ter sido dividida. Esse cenário onde um processador espera pelo outro pode, ainda, ocorrer caso o sistema operacional ceda, temporariamente, um processador para outra aplicação. Assim, quando o PV x, que ficou parado por falta de processador, ganhar novamente uma fatia de tempo para utilizar o processador, o outro PV possivelmente já terá terminado sua parte do trabalho e ficará ocioso enquanto o PV x executará sozinho uma carga que poderia ser dividida.

Na Figura 10(b), ilustramos a divisão de trabalho com a granularidade mais fina possível. Neste cenário temos diversas unidades independentes de trabalho que executam pouca computação. Essa estratégia de divisão costuma gerar um sobrecusto de execução muito maior em relação à divisão com granularidade grossa, pois a concorrência é, em geral, muito maior do que o paralelismo

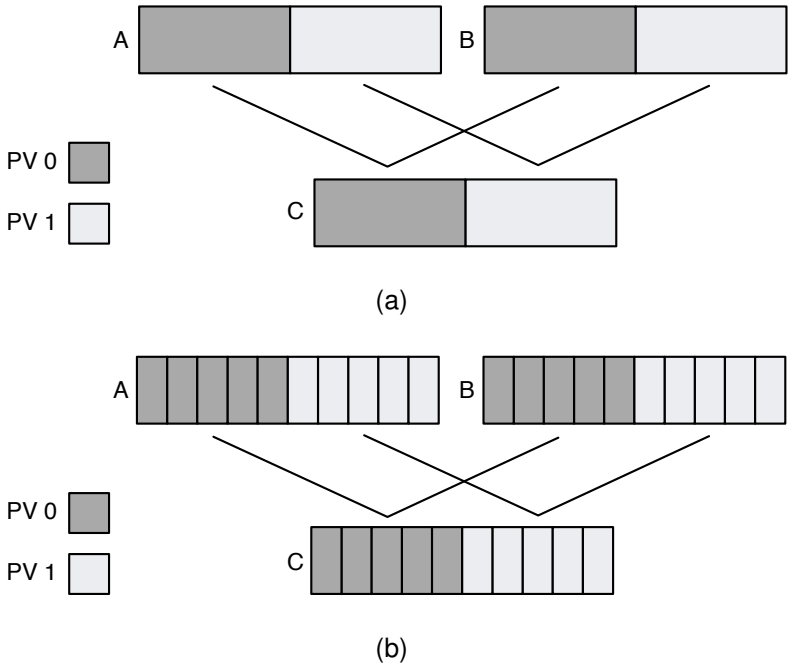


Figura 10: Três vetores são usados para calcular $C = A + B$ paralelamente em uma arquitetura com 2 PVs. Na Figura (a) a divisão do trabalho é feita com uma granularidade mais grossa possível, onde cada thread calcula metade do vetor C. Na Figura (b) temos uma divisão com a granularidade mais fina possível, onde cada elemento de C é calculado por um thread diferente.

disponível no ambiente. No entanto, a programação torna-se mais simples, pois a divisão de trabalho baseia-se no paralelismo oferecido pela aplicação e não se limita ao paralelismo oferecido pela arquitetura. Esses fatores contribuem tanto para a escalabilidade da aplicação quanto para o balanceamento de carga entre os PVs, pricipalmente em situações adversas como as descritas no parágrafo anterior.

Contudo, as vantagens da divisão de trabalho com granularidade fina trazem consequências: (1) os custos de criação, escalonamento e término de threads são muito maiores se comparados a uma chamada de função simples; (2) quando os threads acessam dados compartilhados são necessários códigos e técnicas adicionais; (3) quando paralelizamos laços, as otimizações de compilador podem ser perdidas, uma vez que os threads podem ser escalonados em qualquer ordem. Esses custos devem ser gerenciados no ambiente de execução das ferramentas de programação e minimizados tanto quanto for possível, de forma que ainda se possa se obter ganhos de desempenho com um grão de paralelismo muito fino.

4.5 Conclusão

Diversos aspectos devem ser observados na hora de implementar um modelo multithread em nível aplicativo. Os modelos arquitetônicos que utilizam componentes distribuídos conseguem diminuir a contenção experimentada no modelo centralizado. Contudo, a busca de um PV por trabalho torna-se mais complexa, tornando a técnica de roubo de trabalho uma necessidade, e as políticas de prioridades devem ser revistas para que o balanceamento de carga funcione corretamente. De qualquer forma, os pontos de escalonamento devem ser executados da maneira mais eficiente possível se quisermos nos valer da flexibilidade de aplicações com granularidade fina.

No próximo capítulo discutiremos em detalhe o ambiente Anahy, um ambiente minimalista, que emprega algoritmos de escalonamento de lista e utiliza PVs com política *Work-First*, sem migração. Apresentaremos duas implementações do modelo proposto por Anahy, uma delas realizada especialmente para as necessidades deste trabalho. Iremos descrever ambas as versões da ferramenta, detalhando tanto aspectos de implementação quanto de interface de programação.

5 O AMBIENTE ANAHY

O modelo Anahy (CAVALHEIRO et al., 2007) define um ambiente de execução multithread e uma interface de programação para esse ambiente. A interface de programação é minimalista, basicamente composta pelas primitvas *init*, para inicializar, e *terminate*, para encerrar o ambiente, e pelas primitivas *fork* e *join* para criar e sincronizar threads, respectivamente. No modelo Anahy toda a sincronização e comunicação de dados entre os threads se dá durante as chamadas a *fork* e *join*, não sendo permitidos outros mecanismos de sincronização tais como semáforos e barreiras. Anahy emprega algoritmos de lista no escalonamento de threads em seu núcleo, e os processadores virtuais (PVs) empregam uma política *Help-First* sem migração de threads.

Anahy possui atualmente duas implementações distintas, as quais serão detalhadas no restante deste capítulo. Na Seção 5.1 é apresentada a ferramenta *Athreads*, que implementa o modelo Anahy sobre uma arquitetura centralizada, fornecendo uma interface de programação baseada no modelo POSIX[1] para threads. Na Seção 5.2 é detalhada a ferramenta *Anahy3*, a qual implementa o modelo de Anahy com uma arquitetura de software com listas distribuídas e com uma API orientada a objetos. A Seção 5.3 tece considerações sobre o capítulo.

5.1 Athreads

Athreads é uma biblioteca escrita em C que dá suporte a programação *multithread* com uma API baseada na de Pthreads (NICHOLS; BUTTAR; FARRELL, 1998). Sua primeira versão data de 2006 e é mantida atualmente pelo grupo de pesquisa LUPS (*Laboratory of Ubiquitous and Parallel Systems*[2]) na Universidade Federal de Pelotas. Athreads[3] fornece recursos para programação em arquiteturas multiprocessadas, com uma interface focada na descrição da concorrência da aplicação desvinculada ao paralelismo do *hardware*.

[1] Padrão IEEE P1003.c.

[2] `lups.ufpel.edu.br`

[3] Disponível para download em `http://anahy.org/Downloads.html`.

5.1.1 Interface de programação

Athreads possui uma interface de programação minimalista baseada, essencialmente, nas operações `pthread_create` e `pthread_join` do modelo POSIX para *threads*. As funções básicas de criação e sincronização em Athreads são `athread_create` e `athread_join`, as quais também recebem argumentos semelhantes às operações correspondentes no modelo POSIX. Sua sintaxe exata é apresentada na Figura 11 (juntamente com outras duas funções da API, `aInit` e `aTerminate`, que serão detalhadas posteriormente).

```
1 int aInit(int* argc, char*** argv);
2 int aTerminate(void);
3 int athread_create(athread_t *th, athread_att_t *atrib,
4     void* (*func)(void*), void * in);
5 int athread_join(athread_t th, void **res);
```

Figura 11: Declarações das operações de Athreads.

Uma chamada à função `athread_create` cria um novo thread cuja execução começará pela função apontada por `func`. Essa função deve receber argumentos e retornar resultados por meio de ponteiros do tipo `void*`. Os dados de entrada do thread são os argumentos que `func` recebe, apontados por `in`. `th` aponta para um descritor de thread do tipo `athread_t`. Esse servirá posteriormente para sincronizar o thread criado. O argumento `atrib`, quando não nulo, aponta para uma estrutura de dados que contém atributos definidos pelo programador para a execução do thread, como por exemplo afinidade do thread a um PV. Uma chamada à função `athread_join` sincroniza a execução do thread que chama essa função com o final do thread indicado pelo descritor `th`. Quando um thread retorna de uma chamada a `athread_join`, eventuais dados retornados pelo thread `th` sincronizado serão apontados pelo ponteiro `res`.

A Figura 12 exibe o código para o cálculo recursivo do número de Fibonacci. Note que as chamadas a `athread_create` não especificam nenhum atributo especial aos threads criados, bem como as chamadas a `athread_join` não passam endereços para o recebimento de resultados, uma vez que a função `fib` usa a mesma variável de entrada para escrever o resultado. As funções `aInit` e `aTerminate` servem para as respectivas inicialização e desligamento da arquitetura virtual do ambiente. `aInit` recebe os endereços dos argumentos recebidos pela função `main` via linha de comando, por onde podemos definir, por exemplo, um número específico de PVs a serem usados pelo ambiente.

```
1  void *fib(void *arg){
2    long* n = (long*)arg;
3    long m1, m2;
4    athread_t th1, th2;
5
6    if (*n == 0 || *n == 1) return;
7
8    m1 = *n - 1;
9    m2 = *n - 2;
10
11   athread_create(&th1, NULL, fib, (void*)&m1);
12   athread_create(&th2, NULL, fib, (void*)&m2);
13
14   athread_join(th1, NULL);
15   athread_join(th2, NULL);
16
17   *n = m1 + m2;
18 }
19
20 int main (int argc, char *argv[]) {
21   long n = atol( argv[1] );
22
23   aInit (&argc, &argv);
24   fib(&n);
25   aTerminate ();
26
27   return 0;
28 }
```

Figura 12: Cálculo paralelo da sequência de Fibonacci em Athreads.

Em Athreads, descritores do tipo `athread_t` podem ser recebidos ou passados como parâmetros da função de início de um thread. Desta forma Ahtreads permite o desenvolvimentos de programas com padrões irregulares de paralelismo. Além disso, através do segundo argumento de `athread_create` pode-se especificar uma criação (e posterior sincronização) múltipla de threads. Athreads suporta a criação e sincronização múltipla de threads em Athreads através do modelo *split-compute-merge* (CAVALHEIRO; VIÇOSA, 2007). Na implementação, a função `func`, passada como parâmetro para `athread_create`, é aplicada sobre um vetor de dados apontados por `in`. Os atributos que o programador deve incluir na estrutura `atrib` devem fornecer o número de fluxos de execução que devem ser criados, o tamanho do vetor com os dados de saída, e as funções para o particionamento dos dados de entrada entre os threads e posterior união dos dados de saída.

5.1.2 Implementação de Athreads

Athreads possui uma arquitetura centralizada de software, onde só existe uma lista de threads prontos que todos os PVs acessam. Os PVs funcionam com política *Help-First* sem migração, assim cada PV possui sua própria pilha de threads bloqueados. Estes PVs, implementados como *threads* de sistema (utilizando Pthreads), podem ser vinculados, através de parâmetros passados para `aInit`, aos processadores físicos de forma a explorar a localidade de dados

e otimizar a utilização dos recursos de memória.

Athreads emprega uma estratégia de escalonamento de lista em seu núcleo utilizando como prioridade o nível do thread (na árvore de criações), isto é, a prioridade do thread é igual à distância deste thread até Γ_1, o thread inicial do programa. Para compreender melhor o algoritmo de escalonamento, considere uma arquitetura monoprocessada e o estado inicial do grafo onde existe apenas Γ_1 e a lista de threads prontos é composta, também, apenas por Γ_1. Essa lista é ordenada de forma crescente pelo nível dos threads de forma que um PV ocioso retira o primeiro elemento da lista para executar. Γ_1 é então retirado da lista e a tarefa $\tau_{1,1}$ (tarefa 1 do thread 1) é iniciada sobre o PV. Ao ser executada uma operação de criação do thread Γ_2, um novo thread é inserido no grafo e sua referência é igualmente armazenada na lista de threads prontos. PV segue executando Γ_1, agora com a tarefa $\tau_{2,1}$. Na sequência, uma operação de sincronização de Γ_1 com Γ_2 faz com que $\tau_{2,1}$ termine e Γ_1 seja bloqueado e inserido na pilha do PV. Neste momento o PV realiza uma busca por trabalho de forma parametrizada: como Γ_2 causou o bloqueio do thread atual o PV percorre o grafo e tenta executar Γ_2 ou um de seus descendentes. Caso não seja possível, ele realiza uma busca normal. No nosso cenário o PV em questão obtém sucesso na busca parametrizada e Γ_2 é escalonado, iniciando a execução de $\tau_{1,2}$ sobre o PV. O ciclo pode prosseguir neste mesmo padrão de forma recursiva, de forma que ao final da execução da última tarefa de Γ_2, o PV retoma a execução do thread Γ_1, retirando-o de sua pilha de threads bloqueados.

No caso de uma arquitetura multiprocessada, dois ou mais PVs executam o mesmo algoritmo descrito acima, assim dois ou mais threads podem executados paralelamente.

5.2 Anahy3

Anahy3 implementa o modelo Anahy na linguagem C++, sobre uma arquitetura de software distribuída, isto é, composta por PVs com listas individuais de trabalho, sem uma lista global, implicando na necessidade de roubo de trabalho. A concorrência desses PVs em nível de sistema é provida por threads POSIX criadas pelo ambiente. A interface de programação oferecida por Anahy3 permite com que o programador descreva toda a concorrência de suas aplicações por meio de primitivas `fork` e `join`. O ambiente de execução se encarrega de, em tempo de execução, realizar o balanceamento de carga da aplicação sobre as listas dos PVs criados, por meio do roubo de trabalho.

A interface de programação define objetos que funcionam como descritores de threads e possuem métodos para configurar a execução do thread a ser

lançado. A API fornece primitivas `fork` e `join` para a criação e sincronização dinâmica de threads, de forma que esses threads serão distribuídos nas listas dos PVs e o futuramente os dados produzidos por eles podem ser lidos de forma segura.

Para que o ambiente funcione de maneira correta e eficiente, ferramentas e técnicas específicas foram utilizadas nesse projeto. As seções seguintes apresentam em detalhe a arquiteura do ambiente Anahy 3 e os componentes envolvidos nas atividades de escalonamento, além de uma documentação detalhada das interfaces de programação disponíveis.

5.2.1 Arquitetura do núcleo

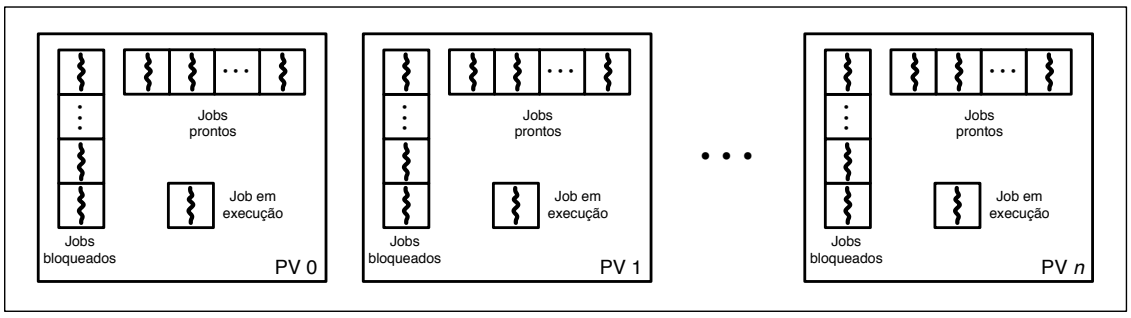


Figura 13: Arquitetura do ambiente Anahy 3.

Anahy implementa uma arquitetura de listas distribuídas pelos PVs. Como não é permitida migração, cada PV possui uma pilha onde armazena referências para os threads bloqueados por consequência de um `join` não satisfeito. A Figura 13 ilustra a arquitetura do ambiente de execução. O retângulo externo representa o encapsulamento da *máquina virtual*, composta por objetos de PVs e threads de sistema para executar a lógica desses PVs. Os retângulos no próximo nível interno representam os PVs. Estes PVs são compostos por estruturas que armazenam os *jobs* (nomenclatura usada em Anahy3 para designar threads criados pela aplicação): uma lista para os jobs prontos e uma pilha para os jobs bloqueados. Os jobs, por sua vez, são ilustrados pelas linhas sinuosas contidas nas divisões das estruturas de cada PV. O *job* é a unidade de escalonamento do ambiente, trabalho que deve ser executado por algum dos PVs. Estes três componentes básicos, jobs, processadores virtuais e a Máquina Virtual, compõem o núcleo de execução do ambiente Anahy.

5.2.1.1 Job

O *job* é a unidade básica de trabalho no ambiente Anahy. Um job é o descritor de um thread em nível aplicativo. Basicamente, a estrutura de dados de um job encapsula o endereço estático da função de início do thread, um ponteiro para os

argumentos que essa função deve receber e um ponteiro para a área de memória na qual os resultados da função devem ser escritos.

Toda e qualquer função em um programa que executa sobre *Anahy3* executa no contexto de um job. No início da execução do programa existe o job raiz, o qual executa a função `main`. A partir da execução deste job, novos jobs podem ser criados e sincronizados dinamicamente, da mesma forma que as funções executadas por esses jobs também podem criar e sincronizar jobs. Cada job criado durante a execução da aplicação armazena uma referência para seu "pai" criando um DCG com relações de precedência entre os jobs. Da mesma maneira, dependências são inseridas neste DCG quando um job deve esperar pelos resultados produzidos por um outro. O ambiente de execução realiza o escalonamento dos jobs buscando a redução no tempo total de execução do programa, respeitando as dependências de dados entre jobs.

5.2.1.2 Processadores virtuais e estratégia de escalonamento

Os processadores virtuais (PVs) são os consumidores do trabalho descrito pelos jobs, onde esse trabalho inclui as atividades de escalonamento geradas pela execução dos jobs. Os PVs utilizam uma política *Help-First* sem migração. O atributo de prioridade para os jobs em Anahy3 é o *co-nível* de sua tarefa atual, calculado considerando custos unitários para todas as tarefas do programa. Cada PV executa executa o algoritmo de escalonamento em paralelo com outros PVs da seguinte forma: um PV ocioso busca o job com maior *co-nível* em sua lista; caso não obtenha nenhum, o PV ocioso escolhe um outro PV, aleatoriamente, e tenta roubar o job com maior co-nível de sua lista. Quando um PV termina a execução de um job (ou não obtém sucesso nas buscas por jobs prontos) e possui um thread no topo de sua pilha, o PV tenta continuar a execução desse job.

5.2.1.3 Máquina Virtual

A máquina virtual de Anahy3 é basicamente composta por um vetor de n processadores virtuais e um outro vetor de $n - 1$ threads de sistema (threads POSIX) para executar a lógica de cada PV em paralelo. Este segundo vetor possui um elemento a menos porque um PV é executado sobre o thread principal do programa. A interação do programador com o ambiente de execução se dá essencialmente por meio da interface de programação da máquina virtual, a qual será detalhada a seguir.

5.2.2 Interface de Programação

Anahy fornece interfaces orientadas a objeto para que o programador possa realizar três tarefas básicas: inicializar/terminar o ambiente de execução, criar e configurar jobs, e lançar à execução um job, para mais adiante sincronizá-lo.

5.2.2.1 Inicializando/terminando o ambiente de execução

A Figura 14 mostra quatro métodos principais da API da Máquina Virtual de Anahy3. Dois deles server para inicializar (`init`) e terminar (`terminate`) o ambiente de execução. `AnahyVM::init` recebe os mesmos parâmetros da função `main` do programa, pois os parâmetros para máquina virtual devem ser passados por linha de comando, no momento do lançamento da aplicação. Estes parâmetros definem certos comportamentos do ambiente de execução como o número de PVs a serem utilizados e atributos de afinidade de threads de sistema a processadores, para determinar que um PV seja escalonado pelo sistema operacional sempre sobre o mesmo processador físico. Enquanto `AnahyVM::init` cria e inicializa os PVs e os threads de sistema que suportam a execução paralela do PVs, `AnahyVM::terminate` sincroniza os threads de sistema e libera a memória ocupada pelos objetos do ambiente de execução.

```
1 static void AnahyVM::init(int argc, char **argv);
2 static void AnahyVM::terminate();
3 static void AnahyVM::fork(AnahyJob* job);
4 static void AnahyVM::join(AnahyJob* job, void** result);
```

Figura 14: Subconjunto básico da API da Máquina Virtual de Anahy 3.

5.2.2.2 Executando e sincronizando jobs

Quando criamos um job apenas encapsulamos em um objeto as referências para uma função e seus argumentos (entre outros valores). No entanto, para que deixemos explícito ao ambiente de execução que este job pode ser executado concorrentemente aos demais jobs executáveis no momento, devemos chamar o método *fork* da máquina virtual. `AnahyVM::fork` recebe um ponteiro para o job como parâmetro e delega o *fork* ao PV atual, de forma que este PV calcula a prioridade do job e o insere em sua própria lista em uma posição adequada, para que ele possa ser futuramente executado por um PV ocioso. Da mesma forma, quando chamamos `AnahyVM::join` passamos um ponteiro para o job a ser sincronizado e a máquina virtual delega o *join* ao PV atual. Após o PV terminar a operação é garantido que os resultados do job sincronizado estão prontos, assim estes dados são lidos seguramente e `result` (segundo parâmetro de `AnahyVM::join`) passa a apontar para estes resultados.

5.2.2.3 Criando e configurando jobs

```
1  void* bar(void* input) { return NULL; }
2
3  void* baz(void* input) {
4
5    AnahyJob* job = (AnahyJob*)input;
6    void* result;
7    AnahyVM::join(job, &result);
8    delete job;
9
10   /* processa result e retorna algum valor */
11 }
12
13 void* foo(void* input) {
14
15   AnahyJob autoreleased_job(bar, NULL);
16   AnahyVM::fork(&autoreleased_job);
17
18   AnahyJob* allocated_job = new AnahyJob(bar, NULL);
19   AnahyVM::fork(allocated_job);
20
21   AnahyJob* smart_job = AnahyJob::new_smart_job(baz, (void*)allocated_job);
22   smart_job->set_join_counter(1);
23   AnahyVM::fork(smart_job);
24
25   AnahyVM::join(autoreleased_job, NULL);
26
27   return (void*)smart_job;
28 }
29
30 int main(int argc, char** argv) {
31   AnahyVM::init(argc, argv);
32
33   AnahyJob job(foo, NULL);
34   AnahyVM::fork(&job);
35
36   AnahyVM::join(job, NULL);
37   AnahyJob* smart_job = job.result(); /* outra maneira de ler resultados de um job */
38
39   AnahyVM::join(smart_job, NULL);
40
41   AnahyVM::terminate();
42   return 0;
43 }
```

Figura 15: Exemplo de programa irregular em Anahy3.

Antes de passarmos um job como argumento para `AnahyVM::fork`, o programador deve criar um objeto do tipo `AnahyJob` e inicializá-lo de maneira apropriada. Existem três maneiras de criar um objeto da classe `AnahyJob`, as quais são demonstradas na Figura 15: *(i)* na pilha da função, *(ii)* no *heap* (área de memória alocada dinamicamente pelo programa em execução) usando `new` e *(iii)* no *heap* usando `AnahyJob::new_smart_job`. A primeira opção é demonstrada na linha 15 e oferece menos sobrecustos de execução, pois o objeto `autoreleased_job` é criado na pilha de execução do PV. Essa opção deve ser empregada quando o job não precisa ser válido após o final da função onde ele foi criado, o que, de fato, acontece no exemplo da figura, visto que na linha 25 é realizado o `join` do job descrito pelo objeto `autoreleased_job`. Na linha 18 um novo job é alocado por meio da palavra-chave `new` de C++. Objetos

criados desta maneira, explicitamente alocados pelo programador, devem ser desalocados pelo próprio programador em um momento futuro da execução, como demonstrado na linha 8. O objeto `allocated_job` é criado no *heap* e continuará válido mesmo após o final da função `foo`, porém este tipo de criação gera mais sobrecustos, visto que o *heap* do programa em execução é uma área de memória compartilhada por todos os PVs, protegida por um *mutex* implícito. Na linha 21 criamos um novo objeto do tipo `AnahyJob` através do método de classe `AnahyJob::new_smart_job`. Objetos obtidos desta maneira são alocados no *heap* e destruídos automaticamente pelo ambiente de execução de Anahy3. O ambiente sabe que deve destruir um *smart job* quando este job já recebeu um número específico de *joins*. Esse número deve ser determinado pelo programador e, caso ele não o faça, o ambiente destrói o job após o primeiro *join*. A linha 22 mostra como o número de operações *join* que um job deve receber pode ser especificado (neste caso, o exemplo é redundante pois 1 é o valor padrão). A utilização de *smart jobs* gera sobrecustos ligeiramente maiores do que a criação de jobs por meio de chamadas a `new`, porém facilita a programação de aplicações concorrentes complexas em Anahy, uma vez que o gerenciamento da memória ocupada pelos descritores é feito pelo ambiente de execução.

O exemplo apresentado na Figura 15 nos permite demonstrar a expressividade do modelo de programação de Anahy. Ponteiros para objetos do tipo `AnahyJob` podem ser recebidos como argumentos pela função inicial de um job ou retornado por essa função. Isto permite o desenvolvimento de aplicações com padrões de paralelismo altamente irregulares, como podemos observar no grafo gerado pelo programa da Figura 15 ilustrado na Figura 16. Nesta figura retângulos tracejados representam jobs (a função inicial é indicada no topo do retângulo), círculos com linha sólida representam tarefas normais, círculos pontilhados representam tarefas vazias nas quais nenhuma computação acontece entre operacões *fork* e *join*, e arestas representam relações de precedência (uma aresta ligando um job a uma tarefa representa um *fork* ou um *join*).

5.2.3 Interface ao estilo POSIX

Anany3 oferece compatibilidade com programas escritos com a interface de Athreads, mapeando as chamadas à API *POSIX-like* de Athreads em métodos da API de Anahy3.

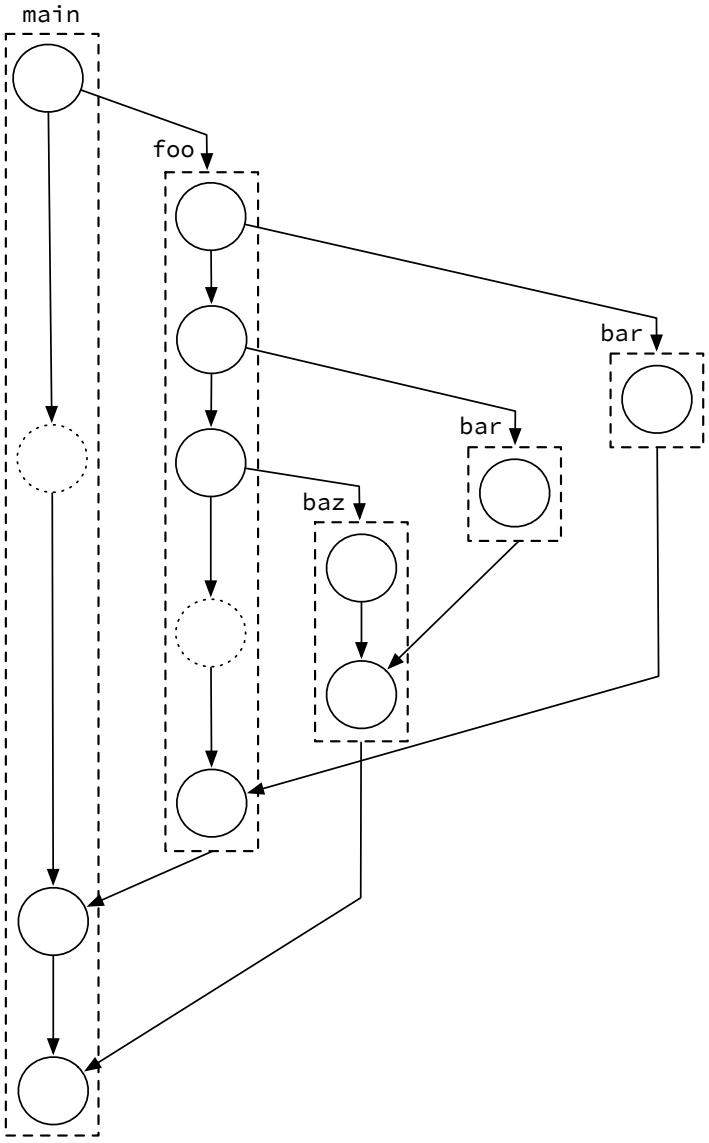


Figura 16: Grafo gerado pelo programa da Figura 15.

5.3 Conclusão

O modelo Anahy define um ambiente de execução multithread que emprega processadores virtuais com política *Help-First* sem migração. Embora esta escolha limite certas decisões de escalonamento, o roubo de trabalho é realizado com maior eficiência, uma vez que somente jobs não executados são roubados e não é necessário salvar e copiar o histórico de um job para outro PV no momento do roubo. Além disso, o fato de não haver migração pode otimizar a utilização da memória cache dos processadores, caso haja uma afinidade estabelecida entre os processadores virtuais e os núcleos de processamento da arquitetura física.

Athreads é uma implementação do modelo Anahy que utiliza uma arquitetura de software centralizada, implementada na linguagem C e utiliza threads POSIX no suporte à execução paralela dos processadores virtuais. Anahy3 utiliza, igualmente, threads POSIX, porém implementa uma arquietetura de software distribuída utilizando a linguagem C++. O código de Anahy3 deve ser compilado com o compilador `g++` ou `icpc`, uma vez que o ambiente faz uso de instruções atômicas implementadas por estes compiladores[4]. Athreads fornece uma API simples, baseada no modelo POSIX, o que permite, com pouco esforço de programação, traduzir programas escritos com Pthreads para programas em Athreads. Anahy3, por sua vez, mantém a compatibilidade com a interface de Athreads e adiciona uma API orientada a objetos, introduzindo o conceito de *smart job*, que automatiza o gerenciamento da memória ocupada pelos descritores de threads. A possibilidade de especificar o thread que irá receber o *join* e de passar descritores de threads como argumentos no momento da criação de um thread, tanto em Athreads quanto em Anahy3, permitem ao programador descrever complexas relações de dependências entre os threads, o que torna ambas as ferramentas altamente expressivas.

Algumas das ideias implementadas em Anahy3 foram inspiradas em conceitos introduzidos pelas principais ferramentas de programação multithread disponíveis na indústria e na academia. O capítulo seguinte se dedica a apresentar tais ferramentas, com ênfase na implementação de seus ambientes de execução e estratégias de escalonamento baseadas em algoritmos de lista.

[4] http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html

6 FERRAMENTAS DISPONÍVEIS COMERCIALMENTE

Diversas ferramentas estão disponíveis comercialmente para dar suporte a programação multithread em nível aplicativo. Ferramentas proeminentes como Intel® Cilk Plus (INTEL, 2012a) e OpenMP (CHANDRA et al., 2001) restringem o modelo de programa *multithread* oferecido pela API para adequá-lo à política de escalonamento empregada pelo ambiente de execução. Intel® TBB (REINDERS, 2007), por outro lado, oferece alta eficiência, com um modelo de programação que permite a descrição de DAGs genéricos. Tais ferramentas empregam estratégias de escalonamento altamente baseadas em algoritmos de lista e introduzem conceitos de implementação peculiares. As ferramentas citadas serão abordadas no decorrer deste capítulo de maneira sistemática, apresentando sua interface, aspectos de implementação (como o modelo de execução e as estratégias de escalonamento empregadas) e considerações do autor sobre as ferramentas. Na Seção 6.4 são tecidos comentários gerais sobre as ferramentas apresentadas.

6.1 OpenMP

OpenMP (*Open Multi-Processing*) é uma ferramenta para programação paralela em C/C++ e Fortran em arquiteturas multiprocessadas com memória compartilhada. OpenMP é mantido por um comitê internacional chamado OpenMP ARB (OpenMP *Architecture Review Board*), formado por alguns do pricipais produtores mundiais de *hardware* e *software* para alto desempenho em nível mundial[1]. A primeira especificação de OpenMP foi estabelecida na década de 1990 para padronizar a descrição de paralelismo em C/C++ e Fortran, uma vez que cada ferramenta de programação e cada compilador introduziam APIs diferentes. Até sua versão 2.5, OpenMP era baseado apenas em primitivas para descrição de paralelismo de dados, isto é, as diretivas eram basicamente dedicadas à paralelização de *loops*. A partir de 2008, na

[1] Mais informações em `http://openmp.org/wp/about-openmp/`.

versão 3.0, foi adicionado a OpenMP o suporte ao paralelismo de tarefas, que permitiu o desenvolvimento de aplicações mais dinâmicas, com padrões de paralelismo mais complexos. Atualmente OpenMP encontra-se na versão 4.0 e sua especificação pode ser encontrada em `http://www.openmp.org/mp-documents/OpenMP4.0RC1_final.pdf`.

6.1.1 Interface de programação

A API de OpenMP define um conjunto de *pragmas*, diretivas de compilação usadas, neste caso, para anotações de paralelismo no código. Nos limitaremos a apresentar alguns desses *pragmas*, aplicados na paralelização de programas escritos em C/C++. Todos os *pragmas* de OpenMP possuem o formato `#pragma omp`, somado a outras diversas construções, como `parallel for`, destinado à paralelização de laços *for*.

Diferente da nomenclatura que usamos nos últimos capítulos, em OpenMP os componentes que chamávamos de *Processador Virtual* são chamados de *threads* (ou time de *threads*), e o que chamávamos de *thread* de aplicação ou *job*, no caso de Anahy3, em OpenMP é chamado de *task* ou tarefa.

A Figura 17 exemplifica o uso de alguns *pragmas* da ferramenta, aplicados em um algoritmo recursivo para calcular números de Fibonacci. O trecho `#pragma omp parallel` (linha 23) indica que naquele ponto do programa será criado um time de threads que irão executar a região paralela seguinte, delimitada pelas chaves. A palavra-chave `sections`, no mesmo *pragma*, indica que a região paralela que está sendo definida é composta por seções concorrentes, onde cada seção é executada por apenas um dos threads do grupo. Essa região paralela composta por seções, no entanto, possui apenas uma seção, para que somente um dos threads execute a tarefa raiz do grafo de dependências que será gerado a partir da função `fib_parallel`. `#pragma omp task` provoca a criação de uma tarefa concorrente para executar o código especificado entre as chaves. A linha `#pragma omp taskwait`, na função `fib_parallel`, faz com que a tarefa atual[2] fique bloqueada até que todas as tarefas criadas a partir dela tenham terminado sua execução, sendo que esse padrão pode se repetir de maneira recursiva, aninhada em múltiplos níveis de profundidade.

Um *pragma* pode possuir notações adicionais para especificar o compartilhamento de variáveis definidas no escopo onde o *pragma* aparece, com as tarefas criadas dentro deste *pragma*. Por exemplo, as variáveis indicadas entre os parênteses em `shared(`*`var_list`*`)` referenciarão as mesmas áreas de memória do escopo onde o *pragma* foi definido, ou seja as variáveis da

[2] A partir do OpenMP 3.0, toda unidade de trabalho concorrente executada por um *thread* é uma tarefa implícita, como no caso da seção especificada com `#pragma omp section`.

lista possuem acesso compartilhado pelas tarefas concorrentes criadas neste *pragma*. `private`, `firstprivate` (padrão), `lastprivate` e `reduction` são outras anotações deste tipo, as quais, diferente de `shared`, criam cópias das variáveis indicadas de diversas maneiras, dentro da nova região paralela ou da tarefa definida pelo *pragma* onde este tipo de anotação aparece. Observa-se que estas anotações, quando feitas em um certo *pragma* são herdadas por *pragmas* aninhados neste, a menos que novas anotações nos *pragmas* internos sobrescrevam as definições do *pragma* externo.

```
1  long fib_parallel(long n){
2    long x, y;
3
4    if (n < 2) return n;
5
6    #pragma omp task shared(x, n)
7    {
8      x = fib_parallel(n-1);
9    }
10   #pragma omp task shared(y, n)
11   {
12     y = fib_parallel(n-2);
13   }
14   #pragma omp taskwait
15
16   return x+y;
17 }
18
19 int main(int argc, char const *argv[]) {
20   long n = atol( argv[1] );
21   long res;
22
23   #pragma omp parallel sections shared(n, res)
24   {
25     #pragma omp section
26     {
27       res = fib_parallel(n);
28     }
29   }
30   exit(0);
31 }
```

Figura 17: Cálculo recursivo paralelo do número de Fibonacci com OpenMP.

A API de OpenMP é bastante extensa e muitas outras construções são fornecidas para descrever paralelismo, todos relacionando tarefas concorrentes em um modelo *fork/join* de execução paralela.

6.1.2 Modelo de execução e estratégias de escalonamento

Em OpenMP as tarefas (threads em nível aplicativo) são criadas em uma estrutura de *forks*/*joins* aninhados. Múltiplos *threads* (criados no nível

de sistema) consomem as tarefas criadas implicitamente pelo ambiente de execução ou explicitamente pelo programador, por meio do *pragma* `task`. A API de OpenMP foi projetada para que um programa produza os mesmos resultados em execuções paralelas e sequenciais (ignorando os *pragmas* de OpenMP no momento da compilação), embora o uso de certas anotações possa levar o programa a apresentar diferentes resultados em execuções paralelas e sequenciais.

Um programa OpenMP começa sua execução em um thread de execução, executando uma tarefa implícita, definida pela função `main`. Quando o thread inicial encontra uma região paralela (anotada com `#pragma omp parallel [...]`) um time de threads é criado, composto por ele (o thread "mestre" do time) e zero ou mais threads trabalhadores, sendo que o número de threads do time é especificado pelo programador via variáveis de ambiente ou via funções da biblioteca de OpenMP. Cada thread do time executa, como uma tarefa implícita, o bloco de código definido pela região paralela. Há também uma barreira implícita ao final do bloco de código que define a região paralela, sendo que somente o thread mestre seguirá a execução do programa e os demais serão destruídos quando todos alcançarem esta barreira.

Quando um thread encontra um *pragma* `task`, uma nova tarefa explícita é criada e, eventualmente, será escalonada sobre um dos threads do time atual. Threads podem suspender a execução de uma tarefa em qualquer *ponto de escalonamento* para executar uma outra tarefa. Os *pontos de escalonamento* são os momentos em que o escalonador é ativado e acontecem quando da criação de uma nova tarefa, do término da execução de uma tarefa e da execução de operações de sincronização entre tarefas (*pragmas* `taskyield`, `taskwait` e barreiras implícitas ou explícitas, pelo uso do *pragma* `barrier`), embora possam haver outros pontos de escalonamento específicos da implementação.

O comportamento padrão quando uma tarefa é criada é que esta seja associada a um thread (`tied`), isto é, uma vez que a tarefa seja escalonada em um thread, somente este thread poderá continuar sua execução posteriormente, isto é, não haverá migração desta tarefa. Outro comportamento padrão é o de que a nova tarefa criada seja escalonada posteriormente (`deferred`) e que, após sua criação, o thread continue a execução da tarefa que a criou. Contudo, o programador pode alterar esses comportamentos adicionando anotações a `#pragma omp task`: `undeferred` força com que o thread execute imediatamente a tarefa criada e suspenda a execução da tarefa criadora; `untied` indica que qualquer thread do time poderá retomar a execução da tarefa em questão, caso essa seja fique bloqueada em um determinado thread por algum motivo. Em suma, OpenMP implementa uma estratégia *Help-First* sem

migração por padrão, porém este comportamento pode ser alterado como o programador especificar.

O término da execução de tarefas *implícitas*, associadas diretamente a uma região paralela é garantido quando o thread "mestre" do time retomar a execução após a barreira implícita que existe ao final da região paralela. O término do conjunto de um conjunto de tarefas *explícitas*, criadas por meio do *pragma* `task`, deve ser garantido com *pragmas* de sincronização de tarefas.

Para construir aplicações em termos de paralelismo de dados, usando construções como o *parallel for*, OpenMP é bastante flexível e apropriado. O ambiente de execução distribui unidades de trabalho entre os threads usando uma das seguintes estratégias: `static`, `dynamic`, `guided`, or `auto`. A primeira distribui cargas iguais de trabalho entre os processadores. As estratégias `dynamic` e `guided` podem ser usadas para ajustar de forma mais fina o balanceamento de carga do sistema. `auto` deixa a escolha da política de escalonamento a cargo da implementação, seja ela definida pelo compilador ou pelo ambiente de execução. Para cada política dessas, o programador pode especificar um valor para `chunk_size`, variável global do OpenMP que indica quantas iterações de um laço *parallel for*, por exemplo, serão agrupadas em uma única tarefa. Contudo, essas estratégias de escalonamento são aplicáveis somente a construções baseadas em paralelismo de dados, como o *parallel for*. Embora a especificação de OpenMP defina certos comportamentos esperados para o ambiente na execução de tarefas, os detalhes do ambiente de execução são específicos de cada implementação do padrão OpenMP e não pode-se fazer maiores hipóteses sobre a organização e o gerenciamento das listas de tarefas.

6.1.3 Considerações sobre a ferramenta

OpenMP é um padrão de indústria, suportado pelos principais compiladores e, portanto, de fácil acesso aos programadores. As construções baseadas em paralelismo de dados possuem uma interface bastante flexível e intuitiva. Tais construções resultam em implementações altamente escaláveis, em geral, fornecendo ótimos *speedups*. OpenMP também fornece diversos mecanismos de controle de seções críticas, o que aumenta a aplicabilidade da ferramenta, porém o uso desses mecanismos não é recomendado no código de tarefas explícitas, uma vez que gera complexidades de controle que podem facilmente resultar em *deadlocks* (AYGUADÉ et al., 2009).

OpenMP possui uma interface bastante extensa e o comportamento dos algoritmos de escalonamento sobre os programas baseados em tarefas explícitas é altamente dependente da implementação do compilador e do ambiente de execução.

6.2 Intel Cilk Plus

Cilk Plus é uma extensão da linguagem C++ que dá suporte a programação multithread em nível aplicativo em arquiteturas *multicore*. Basicamente, Cilk Plus adiciona três palavras-chave à sintaxe de C++, as quais permitem ao programador criar e sincronizar *threads* dinamicamente, em um padrão *fully-strict*, com *forks/joins* aninhados, e paralelizar laços *for* simples, eventualmente executando operações de redução sobre o vetor de dados percorrido nas iterações.

Cilk Plus é um produto da Intel® cuja origem vem do projeto Cilk (BLUMOFE; AL., 1995). Em 2009 a Cilk Arts, que mantinha o projeto Cilk++, evolução sobre o Cilk original, foi incorporada ao conjunto de ferramentas da Intel. Cilk Plus é um projeto de código aberto, disponível em diversos *frameworks* da Intel e atualmente está presente como um *branch* de desenvolvimento do compilador `gcc`[3].

6.2.1 Interface de programação

Cilk Plus adiciona três palavras-chave à sintaxe de C++: `_Cilk_spawn`, `_Cilk_sync` e `_Cilk_for`. Essas palavras-chave permitem a descrição da concorrência presente em computações paralelas do tipo *fully-strict*. A linguagem também oferece outras extensões, como notações de *array* e programação de instruções vetoriais. No entanto esta seção se limita a explorar apenas a expressividade das três palavras-chave.

A presença da palavra-chave `_Cilk_spawn` antes do nome de uma função `func` em um programa Cilk Plus indica que aquela função pode ser executada em paralelo com a função atual, isto é, possui o mesmo papel da primitiva *fork*, introduzida no Capítulo 3. `_Cilk_sync` executa um papel semelhante ao da primitiva *join* (introduzida, também, no Capítulo 3), porém não recebe nenhum parâmetro, devido ao modelo de programa forçado por Cilk Plus: uma função que cria threads por meio de `_Cilk_spawn` deve sincronizar todos os threads criados de uma só vem por meio de `_Cilk_sync`. Caso o programador não execute o comando `_Cilk_sync` em uma função que executa chamadas com `_Cilk_spawn`, o ambiente de execução executa `_Cilk_sync` implicitamente ao final desta função. A palavra-chave `_Cilk_for` paraleliza a execução de laços do tipo `for`.

A linguagem Cilk Plus foi concebida para que a remoção das palavras-chave `_Cilk_spawn` e `_Cilk_sync`, e a substituição de `_Cilk_for` por `for`, resulte em um programa C++ válido, cuja execução sequencial produz o

[3] http://software.intel.com/en-us/articles/intel-cilk-plus-open-source/

mesmo resultado. Ainda assim a execução paralela com acessos a variáveis globais pode causar condições de corrida e o uso de expressões disruptivas, como o `goto`, pode gerar comportamento indefinido, entre outras situações que representam erro de programação ao não respeitar o modelo de execução da ferramenta. A Figura 18 exemplifica a API de Cilk Plus aplicada à descrição da concorrência em um cálculo recursivo da sequência de Fibonacci.

```
1  long fib(long n) {
2    if (n < 2)
3      return 1;
4    else {
5      long x = _Cilk_spawn fib(n-1);
6      long y = _Cilk_spawn fib(n-2);
7      _Cilk_sync;
8      return (x+y);
9    }
10 }
```

Figura 18: Cálculo recursivo paralelo do número de Fibonacci em Cilk Plus.

6.2.2 Modelo de execução e estratégia de escalonamento

No vocabulário de Cilk Plus, um thread em nível de aplicação é composto por uma sequência de *strands*. Um *strand* é uma sequência de instruções sem nenhum ponto de criação ou sincronização de threads, ou seja, um *strand* é uma tarefa no modelo multithread, como definido no Capítulo 3. Quando a execução do *strand* inicial alcança um `_Cilk_spawn` este *strand* termina e dois outros nascem, um que começa pela execução da função “spawned” e outro que é a continuação do *strand* inicial. Caso esse *strand* de continuação encontre um `_Cilk_sync` ele encerra. Neste caso, após todos os *strands* vivos criados a partir da função atual terminarem, é criado um novo *strand* com a continuação do *strand* que terminou na operação `_Cilk_sync`.

Um *spawning block* é um bloco de código com um siginificado especial em Cilk Plus. Funções invocadas com a palavra-chave `_Cilk_spawn` antes são *spawning blocks*. As operações de controle de um laço `_Cilk_for` formam um *spawning block* que gera *spawning blocks* de uma maneira especial com o código do corpo do laço. Ao final de cada *spawning block* há uma chamada implícita a `_Cilk_sync`, o que força a semântica *fork/join* aninhada de Cilk Plus. Chamadas implícitas ou explícitas a `_Cilk_sync` irão sincronizar todas as chamadas a `_Cilk_spawn` realizadas naquele *spawning block* até o momento.

Em Cilk Plus, o ambiente (opcionalmente, o programador, por meio de chamadas do ambiente) cria um conjunto de *worker threads* (ou *workers*) que são processadores virtuais do ambiente, reponsáveis por executar os

threads de usuário, criados por chamadas a `_Cilk_spawn` e `_Cilk_for`. Logicamente, o ambiente de execução de Cilk Plus mantém um grafo com as dependências entre os threads, sendo que cada *worker* possui uma pilha *cactus* (SARDESAI; MCLAUGHLIN; DASGUPTA, 1998) onde empilha o estado da função em execução (*frame* da função) quando uma nova função é criada com `_Cilk_spawn`. Os *workers* implementam a estratégia *Work-First*, i. e., ao encontrar `_Cilk_spawn` um *worker* empilha o contexto da função atual e começa a executar a função criada com `_Cilk_spawn`. Se um outro *worker* efetuar um roubo de trabalho em sua pilha, o ladrão roubará o *frame* mais antigo, com mais potencial de geração de paralelismo. Quando um ladrão continua a execução de um thread roubado, este executa o chamado *clone lento* da função, código gerado pelo compilador de Cilk Plus, que executa chamadas para realizar a sincronização entre threads entre diferentes workers. Exceto em ocasiões de roubo de trabalho, um *worker* executa thread criado por si próprio, executando o *clone rápido*, código gerado pelo compilador, o qual dispensa os mecanismos de sincronização.

Uma vez que o modelo de execução e o algoritmo de escalonamento de Cilk Plus são otimizados para (e limitados a) computações *fully-strict*, o ambiente de execução converte os laços `_Cilk_for` em um percurso eficiente de *divisão e conquista* sobre as iterações do laço. A Figura 19 mostra um DAG representando como *strands* são criados em um laço `_Cilk_for` com 8 iterações. Neste DAG, diferente da representação comumente utilizada, as arestas representam *strands* enquanto os vértices representam pontos de criação e sincronização de *threads*.

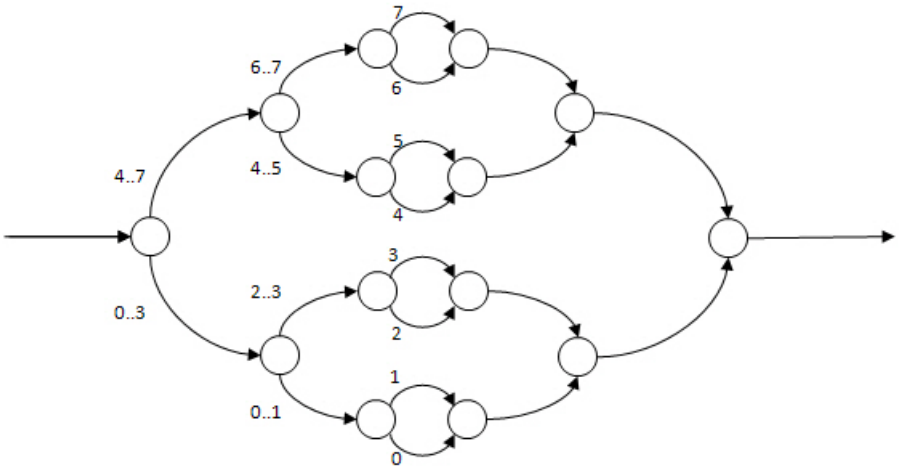


Figura 19: DAG representando um percurso de divisão e conquista sobre as iterações de um laço `_Cilk_for` com 8 iterações. Fonte: Intel (2012a).

6.2.3 Considerações sobre a ferramenta

Cilk limita o programador a escrever programas com estrutura paralela de *forks*/*joins* aninhados, assim seu modelo de programação é bastante simples e conciso. Este fator também é benéfico para o ambiente de execução, pois o algoritmo de escalonamento conhece a estrutura do grafo que irá escalonar, e, podendo assumir certos comportamentos sobre o programa, é possível aplicar otimizações como os clones rápidos e a pilha cactus, a qual permite que a maioria dos acessos concorrentes não necessitem de *locks*. Esta abordagem, no entanto, só garante bons ganhos de desempenho quando o programa pode ser resolvido com paralelismo de *forks/joins* aninhados ou na paralelização de laços de execução por meio da função `_Cilk_for`. A descrição de programas paralelos com grafo irregular, como o da Figura 15 por exemplo, não é possível em Cilk Plus, o que limita a aplicabilidade da ferramenta em aplicações com certos padrões de paralelismo. Além disso, os únicos pontos de sincronização em programas Cilk Plus são as criações e sincronizações de *threads*, não existem *locks* ou mecanismos de controle de concorrência semelhantes.

6.3 Intel Threading Building Blocks

Threading Building Blocks (TBB) (REINDERS, 2007) é uma biblioteca desenvolvida pela Intel®, escrita em C++, que fornece suporte a programação concorrente em arquiteturas *multicore*. Tal biblioteca fornece *templates* que possibilitam o desenvolvimento rápido de aplicações com diversos padrões de paralelismo em construções de alto nível, em sua maioria baseados no paralelismo de dados. TBB fornece uma interface de programação expressiva e um ambiente de execução multithread dinâmico com arquitetura híbrida e escalonamento de tarefas baseado em algoritmos de lista.

6.3.1 Interface de programação

A API de TBB oferece diversos *templates*, focados na descrição de padrões de paralelismo bastante específicos como é o caso do `tbb::parallel_for`, que cria threads para computar cada iteração de um laço `for` de tamanho estático, e do `tbb::parallel_do`, que paraleliza a enumeração sobre uma coleção de dados cujo tamanho pode aumentar conforme alguns itens são computados. Esses *templates* são bastante convenientes para diversos problemas e realizam a manipulação de threads de maneira automática. No entanto, TBB fornece recursos para que o programador possa descrever threads e dependências de dados ao seu modo. Diversos outros esqueletos de paralelismo podem ser criados com a interface de tarefas de TBB como, por

exemplo, o *fork/join* recursivo, presente em algoritmos como o do cálculo do número de Fibonacci.

TBB, assim como OpenMP, chama os threads criados em nível aplicativo de *tasks* (ou tarefas). A classe `tbb::task` deve ser especializada na aplicação com a adição de atributos e métodos, porém o programador deve implementar o método `execute()`, o qual é executado quando a tarefa em questão é escalonada sobre um dos *workers* (mesma terminologia de Cilk Plus, o mesmo que *processadores virtuais*). No corpo do método `execute()` podem ser criadas novas tarefas, as quais podem ser sincronizadas de diferentes maneiras. A forma mais eficiente é explicitando uma tarefa de continuação (modelo de programação *Continuation-Passing*), onde uma tarefa τ em execução cria tarefas filhas, cria uma terceira tarefa e determina que esta será executada quando as filhas de τ terminarem sua execução, e após isto encerra sua própria execução. Essa tarefa de continuação herda o mesmo pai da tarefa τ. Como pode-se observar na sequência de tarefas exibida na Figura 20, uma vez que as tarefas filhas (*child*) e a continuação (*continuation*) são criadas, a tarefa τ pode terminar sua execução e ter seu espaço liberado na pilha do *worker* que a executava. Uma vez que as tarefas filhas (ou suas continuações) tenham terminado sua execução, a tarefa de continuação pode ser escalonada. A Figura 20 ilustra uma sequência de momentos do grafo, onde as tarefas em exeução estão realçadas.

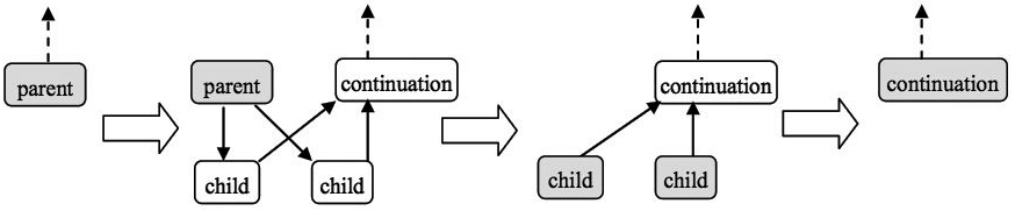


Figura 20: Porção de um grafo de tarefas em TBB para o padrão *fork/join* com tarefa de continuação, não bloqueante. Fonte: (INTEL, 2012b).

A segunda forma de sincronizar tarefas criadas dinamicamente é bloqueando a tarefa que depende do resultado das tarefas filhas que ela criou. A Figura 21 ilustra a sequência de execução e as relações de dependência entre as tarefas nesse modelo. Note que existem arestas da tarefa mãe (*parent*) para suas filhas e vice-versa, gerando ciclos no grafo, fazendo com que o algoritmo de escalonamento precise lidar com esse tipo de dependência cíclica. As tarefas em execução em cada momento da sequência estão realçadas na A Figura 21.

Embora simplifique a programação, o segundo modelo de sincronização é menos eficiente, pois quando uma tarefa fica bloqueada a espera do resultado de suas filhas, seu estado precisa ficar armazenado na pilha do *worker* para posterior recuperação. Esta estratégia poderia levar a um crescimento ilimitado

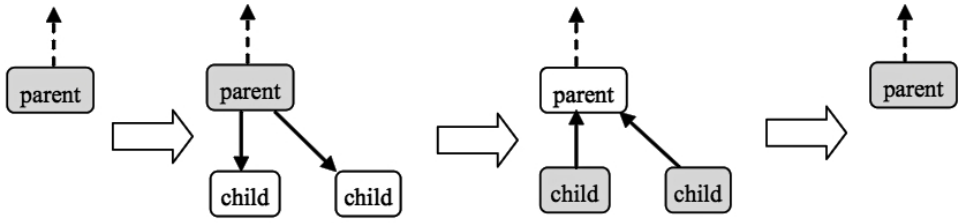


Figura 21: Porção de um grafo de tarefas em TBB para o padrão *fork/join* com bloqueio. Fonte: (INTEL, 2012b).

no tamanho da pilha e abortar a execução do programa. Contudo, o escalonador de TBB impede que um *worker* execute uma tarefa com profundidade menor do que a da tarefa bloqueada no topo de sua pilha (se houverem tarefas bloqueadas). Contudo, esta estratégia pode afetar no desempenho de certos programas, uma vez que limita a quantidade de paralelismo disponível.

A Figura 22 apresenta o código para o cálculo do número de Fibonacci, escrito em C++ com o uso de TBB, no estilo *fork/join* recursivo com tarefa bloqueante. Ao invés de termos uma função que deve ser invocada recursivamente para calcular o número de Fibonacci, cada cálculo parcial do número de Fibonacci é executado em objeto da classe `FibTask` que especializa `tbb:task`. No método `execute`, caso $n \geq 2$, uma tarefa `FibTask` é criada para calcular $n-1$ com o método `spawn` e uma segunda tarefa `FibTask` é criada com o método `spawn_and_wait_for_all` para calcular $n-2$. O segundo método cria, implicitamente, uma tarefa para esperar as outras duas, por isso a chamada do método `set_ref_count` recebe o valor 3 como parâmetro. TBB exige a chamada a `set_ref_count` antes de chamadas a métodos do tipo *spawn* por motivos de desempenho, mas também para realizar a contagem de referências, a fim de gerenciar a memória do ambiente de execução automaticamente (perceba que o método `new` é sobrecarregado por TBB – o programador não deve executar `delete` sobre um objeto do tipo `tbb::task` ou classes derivadas). Um método estático da classe `tbb::task`, `spawn_root_and_wait`, lança a execução da tarefa raiz do grafo e bloqueia a tarefa atual[4] para esperar pelo resultado das computações.

Como vimos, o código da Figura 22 não é o mais otimizado possível, pois pode-se utilizar a técnica de continuação não bloqueante. Caso usemos essa técnica uma outra otimização pode ser feita: o retorno do método `execute` retorna um ponteiro para um objeto do tipo `tbb::task`; a tarefa retornada, caso não seja nula, é executada imediatamente após o término do método `execute()`, não gerando quaisquer operações de escalonamento.

[4]assim como em OpenMP todo trecho de código executa em uma tarefa, implícita ou explicita

```
1  class FibTask: public tbb::task {
2  public:
3    const long n;
4    long* const sum;
5
6    FibTask( long n_, long* sum_ ) : n(n_), sum(sum_) {}
7
8    tbb::task* execute() {
9      if( n < 2 ) {
10       *sum = n;
11     }
12     else {
13       long x, y;
14       set_ref_count(3);
15       spawn( *new(allocate_child()) FibTask(n-1, &x) );
16       spawn_and_wait_for_all( *new(allocate_child()) FibTask(n-2, &y) );
17       *sum = x+y;
18     }
19     return NULL;
20   }
21 };
22
23 int main (int argc, char const *argv[]) {
24   ...
25   long sum;
26   FibTask& a = *new(tbb::task::allocate_root()) FibTask(n, &sum);
27   tbb::task::spawn_root_and_wait(a);
28   ...
29 }
```

Figura 22: Código para o cálculo recursivo do número de Fibonacci em C++ com tarefas de TBB.

6.3.2 Modelo de execução e estratégia de escalonamento

TBB implementa uma arquitetura híbrida, onde cada *worker* mantém uma *deque* (*double ended queue*) onde insere as referências para *tasks* executáveis. O ambiente de execução funciona de uma maneira muito semelhante ao de Cilk Plus, com a diferença que a API de TBB permite a construção de DAGs genéricos para a descrição da concorrência de suas aplicações. Os `workers` de TBB implementam o modo *Help-First*, priorizando localmente o tarefas mais profundas no DAG. Caso não existam tarefas executáveis em sua *deque*, um *worker* executa um *roubo de trabalho* sobre a *task* menos profunda disponível na *deque* de uma vítima. Uma vez que uma *task* é bloqueada na pilha de um *worker*, sua execução só pode ser completada por aquele *worker*, ou seja, não há migração de tarefas bloqueadas. Existe ainda uma fila global de tarefas, compartilhada por todos os *workers*, consumida por ordem de chegada das tarefas enfileiradas por meio do método `enqueue`.

Após completar a execução de uma tarefa τ, um *worker* escolhe a próxima tarefa a ser executada de acordo com a primeira das seguintes regras que for aplicável:

1. A tarefa retornada pelo método `t.execute()` é **não** nula.

2. O sucessor de τ, se τ foi seu último antecessor a terminar execução.

3. Uma tarefa retirada do final da *deque* do próprio *worker*.

4. Uma tarefa pronta com afinidade ao *worker*.

5. Uma tarefa retirada do início da fila global de tarefas.

6. Uma tarefa retirada do início da *deque* de outro *worker*, randomicamente escolhido.

A menos que haja uma razão clara para enfileirar tarefas, deve-se usar a primitiva `spawn` que faz com que o escalonador explore melhor a localidade de dados, o uso de memória do ambiente e o balanceamento de carga.

6.3.3 Considerações sobre a ferramenta

TBB é uma biblioteca muito vasta, possuindo uma gama enorme de *templates* de alto nível para padrões de paralelismo diversos. A implementação das tarefas de TBB fornece recursos que permitem ajustes finos e até mesmo a diminuição de sobrecustos ao intervir no escalonamento. Tais recursos aumentam a aplicabilidade de TBB em aplicações com paralelismo de grão fino. Além disso, TBB fornece o *template* `atomic`, um mecanismo de controle para pequenas manipulações de variáveis críticas, o que aumenta significativamente o poder de expressão de concorrência da ferramenta. Contudo, a interface de programação de TBB não é tão simples e intuitiva. A maioria das construções exige do programador o conhecimento de conceitos avançados da linguagem C++, como sobrecarga de operadores e *templates*, e conceitos de programação funcional como *functors* e expressões *lambda*.

6.4 Conclusão

Cada uma das ferramentas apresentadas neste capítulo introduzem diferentes nomenclaturas para os conceitos de PVs e threads que apresentamos no Capítulo 3. Os PVs são chamados de *threads* em OpenMP e *workers* (ou *worker threads*) em Cilk Plus e TBB. O thread, unidade básica de escalonamento em um ambiente *multithreaded*, é chamado de *task* em OpenMP e TBB e de thread em Cilk Plus, onde os segmentos que compõem um thread também têm o nome especial de *strand*.

Diversas ferramentas comerciais fornecem APIs para a programação de aplicações multithread e ambientes de execução que fazem o escalonamento dinâmico da concorrência gerada na aplicação sobre os recursos disponíveis em nível de sistema operacional. Todas as ferramentas apresentadas empregam

estratégias de escalonamento baseadas em algoritmo de lista e a técnica de roubo de trabalho para realizar o balanceamento de carga entre os PVs.

Das três ferramentas apresentadas, Cilk Plus é a mais restrita, onde os *workers* funcionam sempre da mesma maneira e os programas devem seguir sempre uma mesma estrutura de criação e sincronização de threads. Contudo, essas restrições permitem com que o ambiente de execução seja extremamente especializado e eficiente. Em OpenMP, por outro lado, parâmetros passados para `#pragma omp task` podem fazer com que os threads (PVs) se comportem de uma maneira *Help-First* ou *Work-First* e podem permitir ou proibir a migração de *tasks*, porém o modelo de programa também é restrito. TBB, por sua vez, fornece diferentes recursos para a sincronização de *tasks*, os quais influem nos sobrecustos gerados pelo ambiente de execução, e oferece uma API que permite a descrição de programas com um grafo de dependências genérico. Além disso, TBB introduz uma arquitetura mais complexa em seu ambiente de execução, fornecendo ao programador um controle mais fino sobre o escalonamento do ambiente.

Diversos recursos das ferramentas apresentadas neste capítulo inspiraram o desenvolvimento do ambiente de execução de Anahy3, como as listas de trabalho distribuídas, o roubo de trabalho e o gerenciamento automático da memória ocupada pelos descritores de threads. No capítulo seguinte diversos testes serão realizados, analisando tanto em simulações quanto em execuções reais o impacto de estratégias de implementação, como as adotadas pelas ferramentas aqui apresentadas, sobre o escalonamento de lista em ambientes multithread. As ferramentas apresentadas neste capítulo também serão diretamente comparadas com Anahy3 por meio de aplicações de teste, implementadas em cada ferramenta estudada.

7 RESULTADOS

Neste capítulo apresentamos diferentes experimentos buscando analisar o impacto dos algoritmos de lista no escalonamento de aplicações multithread dinâmicas. A aplicação desta classe de escalonador em ambientes multithread dinâmicos depende de adaptações nos algoritmos para a manipulação de uma unidade de escalonamento diferente, o thread, e seus possíveis estados durante uma execução dinâmica. O impacto destas adaptações em termos do *makespan* gerado será analisado na Seção 7.1, onde por meio de uma ferramenta de simulação, iremos obter os escalonamentos de uma determinada aplicação sobre diversas configurações de ambientes multithread dinâmicos e compará-los contra os escalonamentos estáticos de tarefa para as mesmas aplicações.

Como discutimos no Capítulo 4, aspectos práticos da implementação de ambientes de execução multithread influenciam diretamente no escalonamento realizado e nos sobrecustos gerados pelas operações de escalonamento. Assim implementamos o modelo proposto por Anahy sobre duas arquiteturas de *software*, uma distribuída e uma centralizada, cada uma empregando diferentes estratégias de escalonamento de lista. O desempenho de tais configurações é avaliado por meio da execução das aplicações de teste descritas na Seção 7.2, sendo que os resultados das execuções são apresentados e discutidos na Seção 7.3. Na Seção 7.3.1 as mesmas aplicações de teste são executadas sobre Intel ® TBB, Intel ® Cilk Plus e OpenMP e comparadas com os melhores resultados obtidos por Anahy.

7.1 Simulações

Por meio de simulações é possível analisar o impacto de estratégias de lista no escalonamento de aplicações multithread dinâmicas sem levar em conta aspectos que influenciam nos tempos de execuções reais como, por exemplo, a arquitetura da máquina de testes, os compiladores e seus parâmetros de otimização, custos de processamento das operações de escalonamento etc.

As simulações realizadas neste trabalho consideram a ferramenta de simulação AKSSim, descrita a seguir.

7.1.1 A ferramenta AKSSim

A**nahy **K**ernel **S**chedulers **Sim**ulator* (CAMARGO; ARAUJO; CAVALHEIRO, 2011) – AKSSim – é uma ferramenta de simulação idealizada e implementada pelo autor e seu orientador, a qual realiza o escalonamento de grafos que representam programas concorrentes considerando diferentes algoritmos de lista e diversas arquiteturas simuladas de execução. A entrada de grafos na ferramenta pode ser feita de duas maneiras: (*i*) informando-se dois arquivos no formato GXL[1], os quais devem descrever a estrutura de um DAG e de um DCG que descreva o mesmo programa sob uma interface multithread, ou (*ii*) por parâmetros passados por linha de comando, para que AKSSim gere os grafos programaticamente. No primeiro caso o usuário pode descrever graficamente um DAG na ferramenta GROOVE[2] (GR**aphs for **O**bject-**O**riented **VE**rification*) e obter um DCG equivalente aplicando uma gramática de grafos (CAMARGO et al., 2013). No segundo caso, um DCG é gerado em uma estrutura recursiva, com *forks/joins* aninhados e, a partir das dependências entre as tarefas internas aos threads desse DCG, AKSSim extrai o DAG para realizar escalonamentos considerando somente o nível de tarefas da aplicação. De ambas as maneiras AKSSim obtém duas representações para uma mesma aplicação concorrente de forma que os resultados de escalonamento considerando um ou outro grafo podem ser comparados diretamente. Neste trabalho utilizaremos a segunda opção de entrada, a geração parametrizada de grafos. Os parâmetros aceitos por AKSSim via linha de comando são os seguintes:

- **Custo**: o custo (em unidades de tempo) das tarefas do grafo é gerado por meio do valor inteiro positivo que segue o parâmetro `-c`. Adicionalmente pode-se concatenar a letra “v” ao valor passado (e.g. `-c 10v`) de forma que o custo irá variar de 1 até o valor informado (inclusive);

- **Profundidade**: uma vez que o grafo é gerado com uma estrutura de *forks/joins* aninhados em níveis, threads no nível i irão criar threads no nível $i + 1$ e executar o *join* sobre os threads criados, exceto quando i é igual à profundidade informada. No último caso os threads executam apenas uma tarefa e terminam, sem criar ou sincronizar outros threads. Para especificar a profundidade da geração do grafo é utilizado o parâmetro `-d` seguido de um valor inteiro positivo, o qual pode, também, ser pode ser seguido da

[1] http://www.gupro.de/GXL/
[2] http://groove.cs.utwente.nl/

letra “v” para informar uma profundidade variável para os threads do último nível (e.g. `-d 10v`).

- **Largura**: a largura indica quantos threads um thread do nível i (diferente da profundidade máxima) irá criar no nível $i + 1$. A largura é informada por meio do parâmetro `-w` seguido de um valor inteiro positivo.

- **Processadores**: o número de processadores virtuais a serem utilizados nas arqueiteturas simuladas é informado por meio do parâmetro `-p` seguido de um valor inteiro positivo.

Uma vez que AKSSim possui o DAG e o DCG que descrevem de maneira equivalente a aplicação a ser escalonada, são criadas as arquiteturas de processadores virtuais sobre as quais os grafos serão escalonados. Estas arquiteturas simulam somente a comunicação esperada dos diferentes tipos PVs com o escalonador, no momento das operações de escalonamento, não sendo considerados fatores como acesso à memória ou sobrecustos de operações de escalonamento, ou seja, AKSSim simula o primeiro nível de escalonamento de Anahy, do grafo da aplicação sobre os processadores virtuais, e não considera o escalonamento de segundo nível, dos processadores virtuais sobre os núcleos físicos. Neste trabalho foram adicionados novos algoritmos e tipos de PVs à ferramenta AKSSim. Os PVs que chamaremos de “Task Processors” executam atomicamente as tarefas de um DAG e informam eventos de escalonamento a um escalonador que calcula as prioridades das tarefas *estaticamente* (antes do início da simulação) utilizando um dos seguintes algoritmos: HLFET, HLFNET, RANDOM, SCFET ou SCFNET[3]. Os PVs que chamaremos de “WF”, “HF” e “HF (no migration)” simulam processadores virtuais de ambientes multithread que empregam os modo de execução *Work-First*, *Help-First* com e sem migração, respectivamente. Todos esses PVs simulam a execução de threads e informam eventos de escalonamento a um escalonador que (re)calcula *dinamicamente* as prioridades dos threads executáveis, utilizando um dos seguintes algoritmos: FIFO, LIFO, RANDOM, SCFET ou SCFNET. Para os dois últimos algoritmos o atributo dinâmico do thread é igual ao atributo estático de sua tarefa ativa (ou em curso de execução, mesmo que bloqueada).

A Figura 23 mostra parte de um exemplo de saída gerada por AKSSim considerando os parâmetros de *profundidade* igual a 3, *largura* igual a 2 e *custos* varindo de 1 a 10, e 3 PVs para os escalonamentos. AKSSim imprime a estrutura do grafo expondo as tarefas que compõem cada thread, indicando (nesta ordem) seu identificador único, seu custo, seus antecessores e sucessores, seus

[3] A descrição dos algoritmos de escalonamento encontra-se na Seção 2.4.1.

atributos de prioridade e, opcionalmente, se a tarefa sendo descrita termina executando *fork* ou *join* sobre outro thread. Para cada par possível de algoritmo de escalonamento e arquitetura, AKSSim simula o escalonamento do grafo adequado e informa o *makespan* gerado.

Enquanto o grafo descrito por AKSSim na Figura 23 possui uma estrutura totalmente balanceada e custos variáveis nas tarefas, a Figura 24 mostra um exemplo de DCG com custos variáveis, mas estrutura desbalanceada, onde os threads mais profundos do grafo (os que não criam ou sincronizam outros threads) possuem profundidades diferentes. Na Figura 24 os retângulos tracejados representam threads, os círculos representam tarefas com custos indicados em seu interior, uma seta ligando um círculo a parte superior de um retângulo representa que a tarefa termina ao executar um *fork* e uma seta ligando a parte inferior de um retângulo a um círculo representa que a tarefa inicia após a execução de um *join*. Estruturas desbalanceadas como a do DCG da Figura 24 podem ser produzidas em AKSSim informando-se uma profundidade variável para a geração do grafo.

7.1.2 Resultados

Testaremos a eficiência dos algoritmos de lista no contexto de diferentes arquiteturas simuladas gerados grafos a serem escalonados. Faremos a geração de 4 tipos de grafos, explorando as combinações de custos fixos/variáveis contra estruturas balanceadas/desbalanceadas.

Os grafos gerados nos testes a seguir são escalonados sobre arquiteturas de PVs do tipo “Task Processors”, WF, HF e HF (*no migration*). Como já dito, o primeiro considera algoritmos que realizam o escalonamento de lista estático no nível de tarefas; os demais consideram o escalonamento de lista multithread, simulando uma execução dinâmica, isto é, mesmo que todo o grafo seja inteiramente conhecido antes do início da simulação, somente a porção que estaria disponível em um ambiente dinâmico é revelada para que o escalonador atribua prioridades aos threads e os organize na lista. Diversas simulações foram executadas considerando-se 1, 2, 4, 6, 8, 10, 12, 14 e 16 PVs. Para cada número de PVs considerado, foram executadas 100 simulações e calculadas as médias dos *makespans* obtidos por cada par arquitetura-algoritmo. Este número de execuções foi aplicado até mesmo para os testes onde o grafo gerado é sempre o mesmo, caso cujo resultado é apresentado na Figura 25. Neste caso as múltiplas execuções buscam estabilizar os escalonamentos randômicos, os quais variam em cada simulação. Nas figuras 25, 26, 27 e 28 as linhas apresentam o melhor *makespan* médio entre os algoritmos de escalonamento explorados em cada arquitetura.

```
Graph:

 Thread 0:
   Task 0 (cost 10; pred: []; suc: [1, 8]; level: -51; co-level: 10) --> forks Th1
   Task 8 (cost 9; pred: [0]; suc: [9, 16]; level: -41; co-level: 19) --> forks Th4
   Task 16 (cost 5; pred: [8]; suc: [17]; level: -12; co-level: 24) <-- joins Th4
   Task 17 (cost 6; pred: [15, 16]; suc: [18]; level: -7; co-level: 50) <-- joins Th1
   Task 18 (cost 1; pred: [7, 17]; suc: []; level: -1; co-level: 51)

 Thread 1:
   Task 1 (cost 2; pred: [0]; suc: [2, 3]; level: -28; co-level: 12) --> forks Th2
   Task 3 (cost 3; pred: [1]; suc: [4, 5]; level: -26; co-level: 15) --> forks Th3
   Task 5 (cost 6; pred: [3]; suc: [6]; level: -23; co-level: 21) <-- joins Th3
   Task 6 (cost 9; pred: [4, 5]; suc: [7]; level: -17; co-level: 30) <-- joins Th2
   Task 7 (cost 7; pred: [2, 6]; suc: [18]; level: -8; co-level: 37)

 Thread 2:
   Task 2 (cost 10; pred: [1]; suc: [7]; level: -18; co-level: 22)

 Thread 3:
   Task 4 (cost 2; pred: [3]; suc: [6]; level: -19; co-level: 17)

 Thread 4:
   Task 9 (cost 1; pred: [8]; suc: [10, 11]; level: -32; co-level: 20) --> forks Th5
   Task 11 (cost 6; pred: [9]; suc: [12, 13]; level: -31; co-level: 26) --> forks Th6
   Task 13 (cost 2; pred: [11]; suc: [14]; level: -21; co-level: 28) <-- joins Th6
   Task 14 (cost 7; pred: [12, 13]; suc: [15]; level: -19; co-level: 39) <-- joins Th5
   Task 15 (cost 5; pred: [10, 14]; suc: [17]; level: -12; co-level: 44)

 Thread 5:
   Task 10 (cost 10; pred: [9]; suc: [15]; level: -22; co-level: 30)

 Thread 6:
   Task 12 (cost 6; pred: [11]; suc: [14]; level: -25; co-level: 32)

GanntChart for HLFET on 'Task Processors': (makespan = 52 u.t.)
  Processor 00: ************************************---************
  Processor 01: ----------************************************------------
  Processor 02: ------------************************************------------

GanntChart for SCFNET on 'HF Processors without migration': (makespan = 56 u.t.)
  Processor 00: ********************************---------------*******
  Processor 01: ----------********************************-----------------
  Processor 02: ------------****************************************-------

(...)
```

Figura 23: Exemplo de saída de AKSSim; parâmetros considerados: 3 processadores, profundidade igual a 3, largura igual a 2 e custos das tarefas variando randomicamente entre 1 a 10.

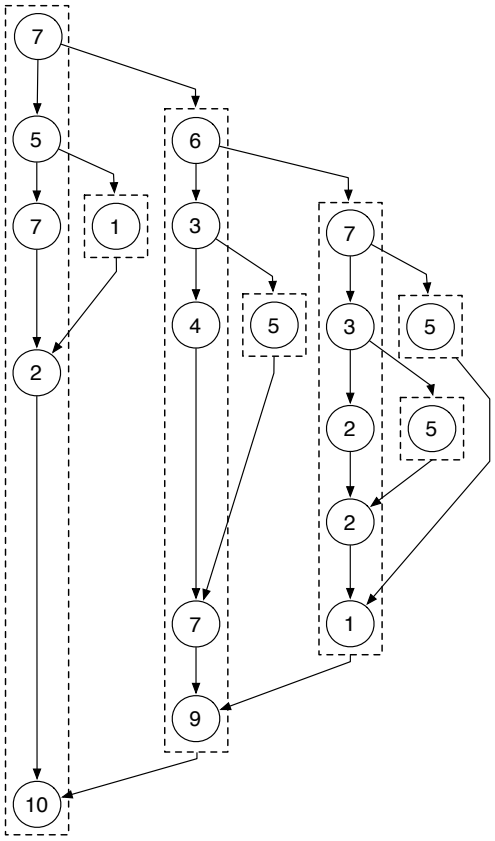


Figura 24: Representação de um exemplo de DCG gerado em AKSSim com custos variáveis e estrutura desbalanceada.

Os resultados das figuras 25 e 26 consideram um grafo com uma estrutura totalmente uniforme, sendo que no primeiro os custos são iguais para todas tarefas e no segundo os custos são gerados randomicamente dentro de um intervalo. O parâmetros de profundidade fixa = 5 e largura = 2 resultam em um grafo com 63 threads / 187 tarefas.

Na Figura 25 onde são considerados custos unitários para todas as tarefas desse grafo, observa-se que a arquitetura “Task Processors” obtém todos os melhores desempenhos nos testes que consideram 4 PVs ou mais. Este resultado é o esperado, uma vez que nesta arquitetura o escalonamento é realizado estaticamente. Além disso, todos os melhores resultados de “Task Processors” são fornecidos pelo algoritmo HLFET, o qual fornece escalonamentos ótimos quando o DAG possui estrutura aninhada e todas as tarefas possuem custos iguais (HU, 1961). Já os resultados obtidos nas arquiteturas WF, HF e HF (*no migration*) são ótimos para 2 PVs, e muito próximos dos resultados ótimos fornecidos pelo algoritmo HLFET sobre “Task Processors”. 70% dos melhores escalonamentos das simulações sobre ambientes multithread são fornecidos pelo algoritmo SCFNET, enquanto os outros 30% são fornecidos pelo algoritmo FIFO. Em relação aos escalonamentos randômicos, as arquiteturas WF e HF são 3% mais eficientes, a arquitetura HF (*no migration*), 6,7%, e HLFET, 7,8% mais eficiente, percentuais de ganho médio. Os melhores resultados obtidos nas simulações multithread geraram um *makespan*, no máximo, 13% maior do que os melhor escalonamentos estáticos.

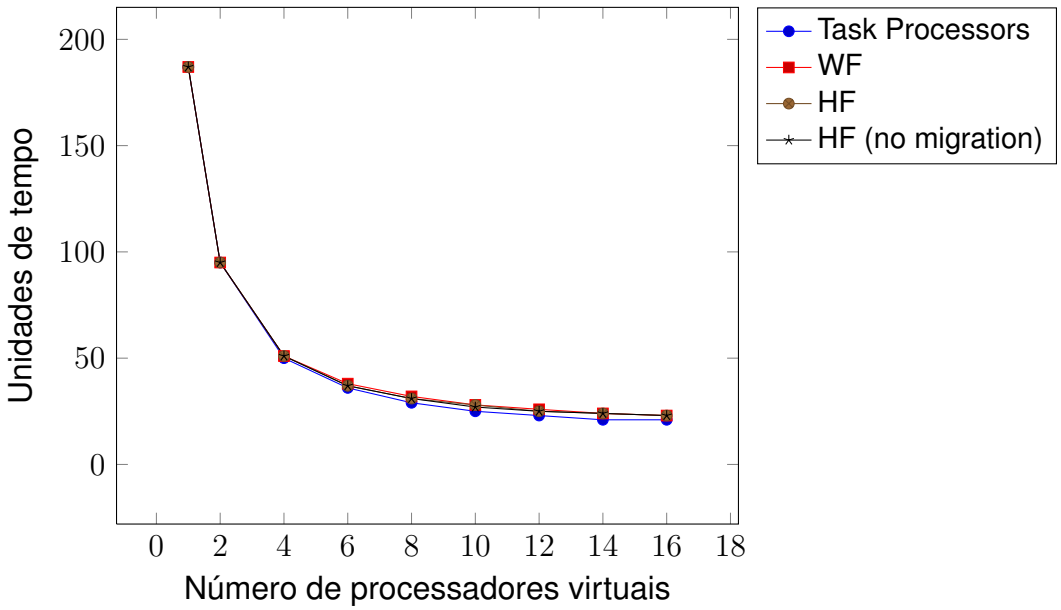


Figura 25: Simulações considerando custo unitário para as tarefas, profundidade = 5 e largura = 2.

Na Figura 26 temos resultados bastante semelhantes aos da Figura 25,

contudo podemos observar que as linhas que representam os melhores escalonamentos obtidos nos ambientes multithread dinâmicos se distanciam ligeiramente da linha que representa os melhores escalonamentos estáticos, todos, novamente fornecidos pelo algoritmo HLFET. Para os resultados das simulações multithread o algoritmo FIFO foi o que forneceu o maior percentual de melhores escalonamentos, 56% deles; o restante dos melhores resultados provieram dos algoritmos SCFET (18%) e SCFNET (26%). Estes resultados demonstram que, para uma aplicação com este tipo de DCG, os melhores escalonamentos são obtidos quando o grafo é executado em largura. Também é interessante notar que o algoritmo SCFNET obteve melhores resultados do que o algoritmo SCFET, o qual, ao contrário do primeiro, considera os tempos de processamento das tarefas para calcular seus atributos de prioridade, transportados dinamicamente para o nível de threads. Sobre a eficiência em relação aos escalonamentos randômicos, os resultados obtidos na arquitetura HF (*no migration*) foram os mais eficientes produzindo *makespans* 9,3% menores, em média, do que o algoritmo RANDOM. Os escalonamentos estáticos fornecidos pelo algoritmo HLFET foram, em média 8,5% melhores do que os escalonamentos randômicos sobre “Task Processors”, enquanto os melhores escalonamentos sobre WF e HF foram no máximo 3% melhores que RANDOM. Os melhores resultados obtidos nestas simulações multithread geraram um *makespan*, no máximo, 17% maior do que os melhores escalonamentos estáticos.

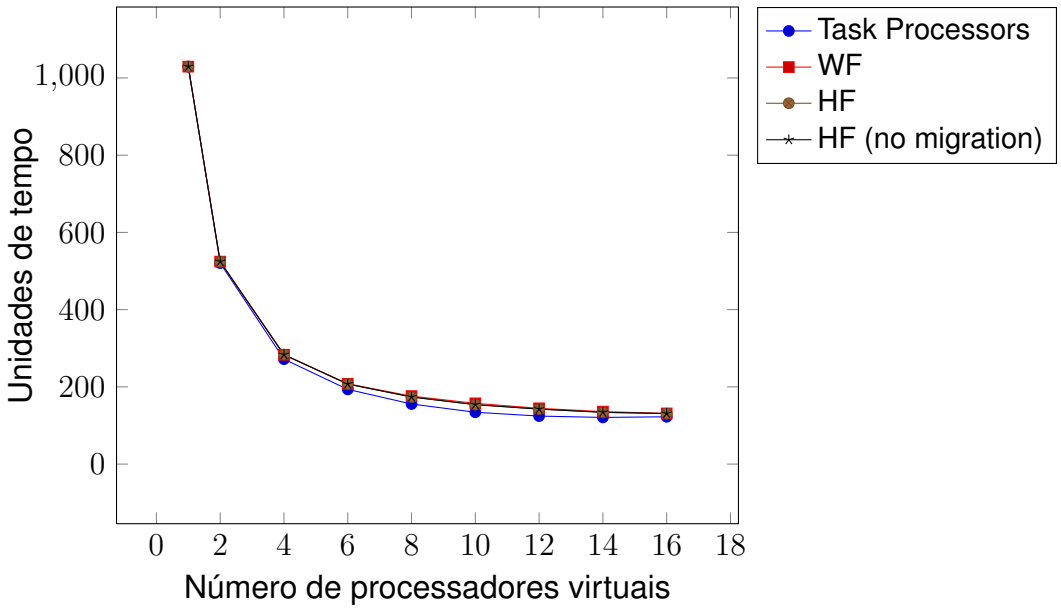


Figura 26: Simulações considerando custos randômicos entre 1 e 10 para as tarefas, profundidade = 5 e largura = 2.

Nas figuras 27 e 28 foram gerados grafos com estrutura desbalanceadas

considerando custos unitários e variáveis para as tarefas, respectivamente. Em ambos os estudos de caso os melhores escalonamentos estáticos sobre “Task Processors” foram fornecidos pelo algoritmo HLFET, e, como pode-se observar, não há uma diferença significativa entre estes escalonamentos e os melhores obtidos nas simulações de ambientes multithread. Considerando custos unitários para as tarefas o algoritmo SCFET apresentou 74% dos melhores escalonamentos, e a estratégia FIFO, 26%. Considerando custos variáveis, FIFO gerou 52% dos melhores escalonamentos e SCFNET, 33%; nesse caso o algoritmo SCFET apresentou os melhores resultados em apenas 15% dos testes. Nesses mesmos testes, considerando custos unitários para as tarefas, os resultados obtidos na arquietura HF (*no migration*) foram até 12% mais eficientes que os escalonamentos randômicos, enquanto em WF e HF os melhores escalonamentos foram, no máximo, 3% melhores do que os randômicos. Já considerando custos variáveis para as tarefas, os resultados obtidos na arquitetura “HF (*no migration*)” foram até 14,5% mais eficientes que os escalonamentos randômicos, enquanto em “WF” e “HF” os melhores escalonamentos não foram 2% melhores do que os randômicos. Em ambos os testes com estruturas debalanceadas, os melhores resultados obtidos nestas simulações multithread geraram um *makespan*, no máximo, 11% maior do que os melhores escalonamentos estáticos.

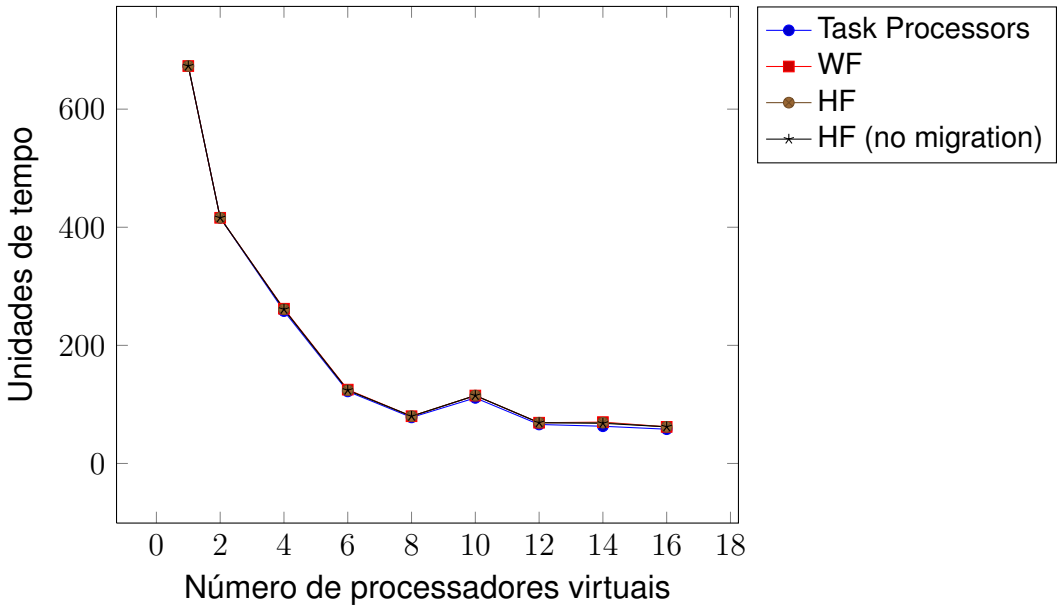


Figura 27: Simulações considerando custo unitário para as tarefas, profundidade variando entre 5 e 10 em cada thread do último nível e largura = 2.

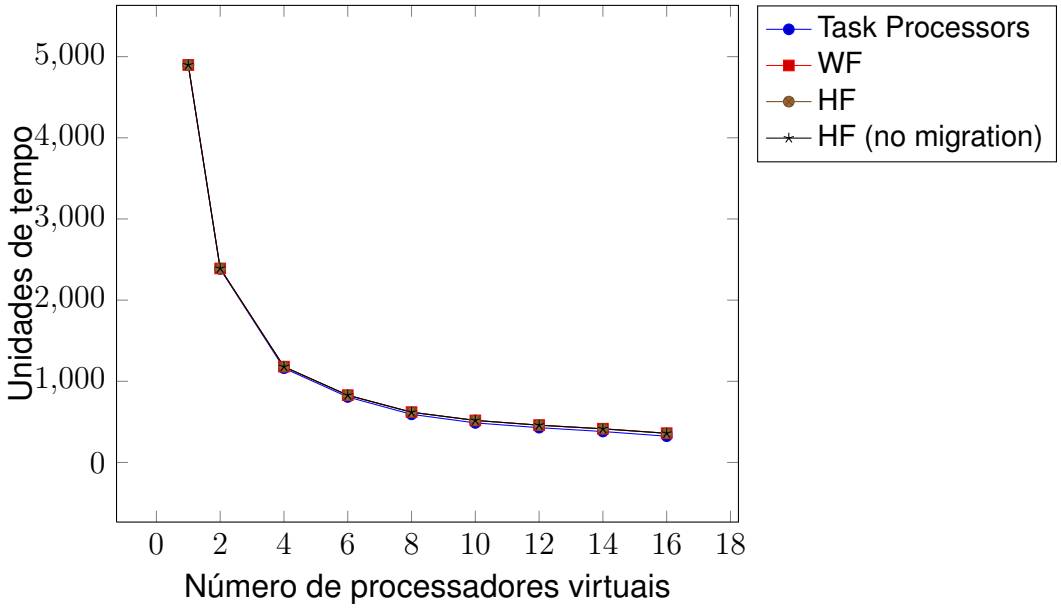


Figura 28: Simulações considerando custos entre 1 e 10 para as tarefas, profundidade variando entre 5 e 10 em cada nos threads do último nível e largura = 2.

7.2 Aplicações de teste

Nesta seção serão apresentadas quatro aplicações de teste, as quais serão implementadas e executadas sobre as versões do ambiente Anahy3, bem como sobre as ferramentas Cilk Plus, TBB e OpenMP.

7.2.1 Quick Sort

Quick Sort (HOARE, 1961) é um algoritmo de ordenação por comparação que possui complexidade $O(n \log n)$ para o caso médio e, em sua implementação mais simples (utilizada neste trabalho), não é estável, isto é, registros com o mesmo valor não necessariamente permanecem na mesma ordem após a ordenação. O *Quick Sort* emprega um procedimento de partição do vetor a ser ordenado, de forma que em cada chamada da função o vetor é partido em dois e o *Quick Sort* é aplicado novamente sobre os dois vetores resultantes. Em geral, o *Quick Sort* é chamado recursivamente até que o vetor passado como parâmetro possua um ou nenhum elemento. No entanto, na implementação realizada neste trabalho, foi utilizado um *threshold* para determinar o tamanho mínimo do vetor a ser ordenado nos threads da base da recursão, com o objetivo de aumentar a granularidade do cáculo – quando o vetor é menor que o *threshold*, é aplicado o método *Selection Sort* (KNUTH, 1997). A Figura 29 apresenta o código em Anahy3 para a computação paralela do *Quick Sort*.

```
1  void* quickSort(void* args) {
2    QSortArgs* _args = (QSortArgs *)args;
3
4    if (_args->last - _args->first < THRESHOLD) {
5      selectionSort(_args->array, _args->first, _args->last);
6    } else {
7      int r = partition(_args->array, _args->first, _args->last);
8      AnahyJob left(quickSort, new QSortArgs(_args->array, _args->first, r-1), NULL);
9      Anahy::fork(&left);
10     AnahyJob right(quickSort, new QSortArgs(_args->array, r+1, _args->last), NULL);
11     Anahy::fork(&right);
12     Anahy::join(&right, NULL);
13     Anahy::join(&left, NULL);
14   }
15   return NULL;
16 }
```

Figura 29: Paralelização do *Quick Sort* em Anahy3.

7.2.2 Alinhamento de sequências genéticas com o algoritmo de Smith-Waterman

O algoritmo de Smith-Waterman (SMITH; WATERMAN, 1990) é empregado para realizar o alinhamento de sequências genéticas. A solução empregada neste trabalho utiliza uma estratégia de programação dinâmica, onde uma matriz M de tamanho $n*m$ armazena informações sobre os melhores sub-alinhamentos entre duas sequências genéticas de tamanho n e m. Cada elemento $M(i,j)$ dessa matriz é calculado da seguinte forma:

$$
\begin{aligned}
E(i,j) &= \max_{k \in \{0,1,\ldots,i-1\}} M(k,j) + \gamma(i-k) \ ; \\
F(i,j) &= \max_{k \in \{0,1,\ldots,j-1\}} M(i,k) + \gamma(j-k) \ ; \\
M(i,j) &= \max \begin{cases} M(i-1,j-1) + s(i,j) \ , \\ E(i,j) \ , \\ F(i,j) \ . \end{cases}
\end{aligned}
\tag{2}
$$

O elemento de maior valor na matriz M representa a pontuação obtida pelo melhor alinhamento possível entre as duas sequências testadas. Embora, de acordo com a Equação 2, o cálculo da célula $M(i,j)$ dependa de todas as células da linha i que estão à esquerda de $M(i,j)$ e todas as células da coluna j que estão acima de $M(i,j)$, é possível eliminar as dependências transitivas e criar um grafo de tarefas para calcular a matriz, de modo que o cálculo do valor da célula $M(i,j)$ dependa somente dos valores das células $M(i-1,j)$ e $M(i,j-1)$, como ilustrado na Figura 30. Esse arranjo de dependências permite o cálculo paralelo dos elementos das antidiagonais da matriz M (indicadas na Figura 30 pelas linhas pontilhadas), padrão de paralelismo conhecido como *wavefront*. As ferramentas utilizadas neste trabalho permitem obter os grafos gerados na Figura 31. Nessa figura os retângulos pontilhados representam threads e os

círculos representam tarefas que calculam células da matriz M (indicadas na figura pelos índices da célula) ou executam uma operação de escalonamento (indicadas na figura pelas palavras *fork*, *join* ou *end*); arestas tracejadas representam operações do tipo *fork* enquanto arestas sólidas representam operações do tipo *join*.

Cilk Plus e OpenMP implementam a estratégia *wavefront* executando, em sequência, primitivas do tipo `parallel_for` para cada antidiagonal da matriz M. Essa estratégia gera um grafo semelhante ao da Figura 31(a).

Anahy3 e TBB fornecem APIs que possibilitam a criação de dependências arbitrárias entre os threads, de modo que elementos de antidiagonais diferentes podem ser calculados em paralelo, conforme suas dependências são satisfeitas. O DCG gerado por Anahy3 e TBB se assemelha ao apresentado na Figura 31(b). Note que nesse DCG, após conhecer o tamanho do problema, o thread principal Γ_1 cria descritores de thread para calcular cada célula da matriz M, chama o *fork* para o thread que irá calcular a célula $M(0,0)$ e faz *join* em todos os threads; cada um dos outros threads calcula a célula $M(i,j)$ de sua responsabilidade e chama *fork* nos threads que irão calcular células que dependam de $M(i,j)$. Um thread em Anahy3 somente torna-se escalonável após receber um número específico de operações *fork*, onde esse número pode ser especificado nos atributos do descritor do thread `(fork_counter)`, sendo que o valor padrão é 1 (um). Esse modelo é inspirado na interface dos objetos `tbb::task` introduzidos por TBB.

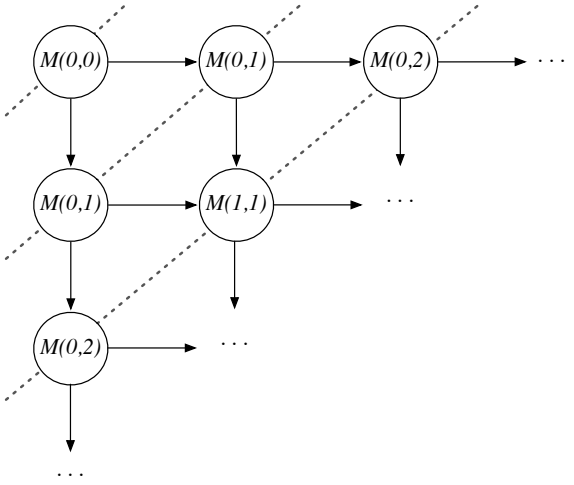


Figura 30: Grafo de dependências da matriz do algoritmo de Smith-Waterman.

Ao paralelizar o algoritmo de Smith-Waterman de forma que cada célula da matriz M é calculada em um thread separado, obtém-se um número significativo de threads com granularidade fina.

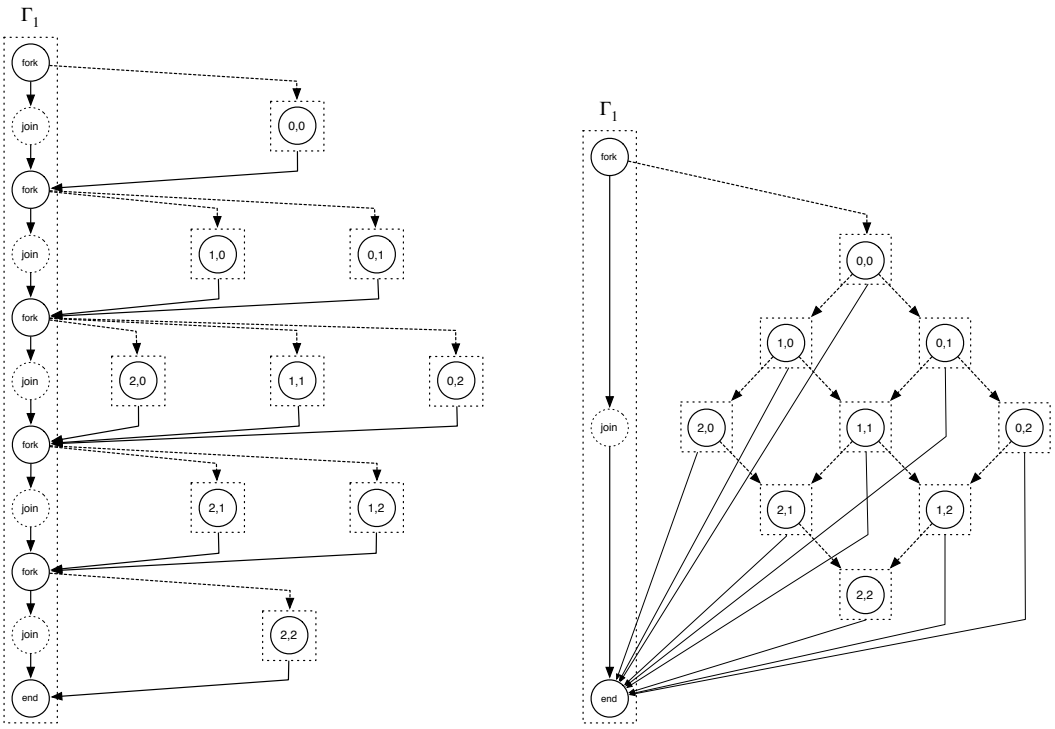


Figura 31: DCGs gerados pela execução do algoritmo paralelo de Smith-Waterman (a) em Cilk Plus e OpenMP, e (b) em Anahy3 e TBB.

7.2.3 Multiplicação de matrizes

A multiplicação de matrizes é uma situação recorrente em diversos problemas que envolvem cálculos matemáticos. Em uma multiplicação do tipo $C = A \times B$ cada célula $C(i, j)$ depende da linha i da matriz A e da coluna j da matriz B. Uma vez que há somente acesso de leitura nas matrizes A e B para os cálculos da matriz C podemos paralelizar o problema em diversos níveis com diferentes estratégias. Neste trabalho iremos analisar o desempenho de dois métodos de decomposição paralela do problema: o cálculo de cada linha i da matriz C em um thread diferente e o cálculo de cada célula $C(i, j)$ em um thread diferente. Enquanto o primeiro método gera menos concorrência, em apenas um nível de criação de threads, o segundo gera mais concorrência, criando threads no DCG em 2 níveis de profundidade. Assim iremos analisar os impactos de cada método sobre os ambientes testados.

7.3 Escalonamentos em Anahy3

Nesta seção iremos analisar o desempenho do ambiente de execução de Anahy3 o qual implementa um modelo de execução *Help-First* sem migração de threads. Duas implementações de Anahy3 foram realizadas neste trabalho, uma

empregando uma arquitetura centralizada e outra empregando uma arquitetura distribuída, como descrito na Seção 4.1. Para o ambiente *centralizado* consideramos as seguintes heurísticas de escalonamento de lista:

- FIFO;
- LIFO;
- RANDOM;
- maior prioridade para o thread com menor *co-nível* (considerando tempos de processamento iguais para todas as tarefas);
- maior prioridade para o thread com menor profundidade na árvore de *forks*.

Essas heurísticas aplicadas ao ambiente centralizado são referenciadas nas figuras 32 a 36 respectivamente por “FIFO (C)”, “LIFO (C)”, “RANDOM (C)”, “SCFNET (C)” e “SFORK (C)”.

No ambiente *distribuído*, onde há roubo de trabalho e inversão de prioridades no roubo, consideramos como os seguintes atributos de prioridade para um dado thread:

- Ordem de criação na lista do PV que o criou;
- Profundidade do thread na árvore de *forks*;
- Número randômico;
- *co-nível* da tarefa atual (considerando tempos de processamento iguais para todas as tarefas).

Esses atributos, somados ao funcionamento do ambiente de execução geram heurísticas que serão são referenciadas nas figuras 32 a 36 respectivamente por “LIFO/FIFO (D)”, “FORK (D)”, “RANDOM (D)” e “H/SCFNET (D)”. A exemplo do que fora realizado nas simulações, o atributo de prioridade do thread corresponde ao atributo de prioridade da tarefa em curso de execução no algoritmo “H/SCFNET”. O “H” foi adicionado ao nome original do algoritmo, devido à inversão de prioridades que utilizamos no roubo de trabalho, pois localmente um PV irá buscar o thread com maior (***H**ighest*) *co-nível* em sua lista, e no momento do roubo irá buscar um thread com menor (***S**mallest*) *co-nível* possível na lista de algum outro PV.

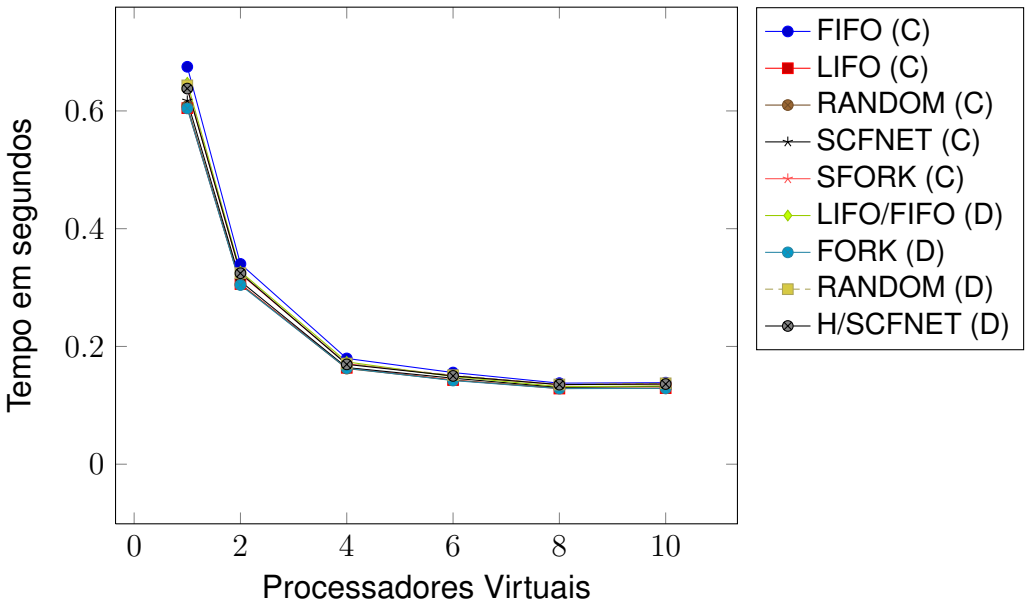
Em todos os testes realizados nesta seção, cada instância (cada combinação de aplicação, parâmetros da aplicação, ambiente de execução e estratégia de escalonamento) foi executada 10 vezes e os números apresentados representam

a média dos tempos de execução em cada instância de execução. As implementações de Anahy3 bem como todas as aplicações foram compiladas com o compilador `icpc` (versão 13.0 presente na suíte Intel®C++ Composer XE 2013) utilizando o parâmetro de otimização `-O2`. As execuções foram realizadas em uma máquina com processador Intel®Corei7-2600 (4 *cores* com *Hyper-Threading* habilitado, *clock* de 3.40GHz e 8 MB de cache L2), memória RAM de 8 GB 1333MHz DDR3 e sistema operacional Linux (*Kernel* 3.0.0-14-generic). Assim, mesmo que o paralelismo real da máquina de teste seja igual 4 núcleos físicos, os testes que seguem consideram 1, 2, 4, 6, 8, e 10 processadores virtuais.

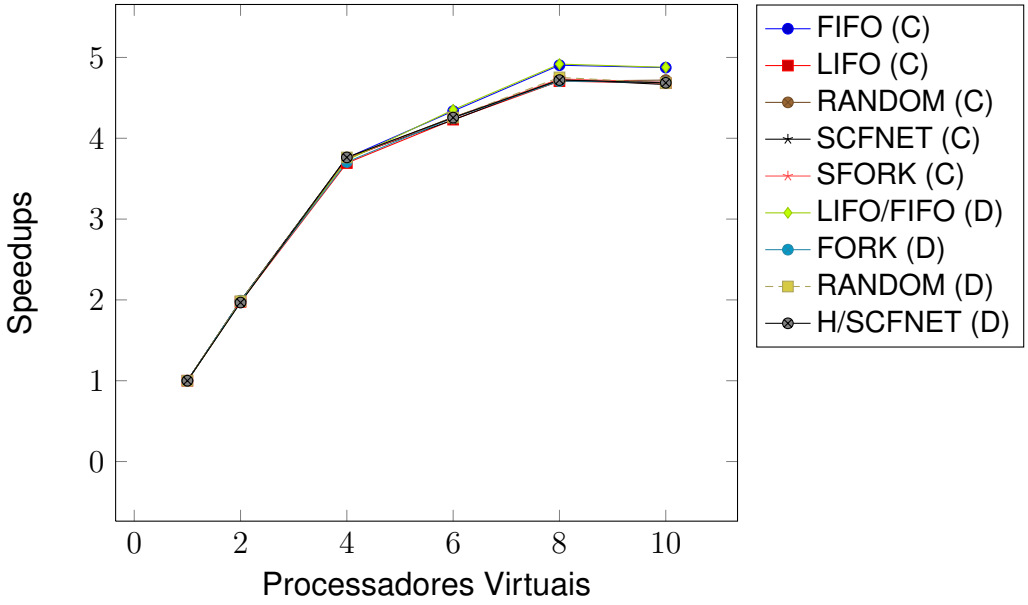
A Figura 32 apresenta os resultados de execução do algoritmo *Quick Sort*, considerando-se vetores gerados randomicamente com 1.000.000 elementos e um *threshold* igual a 1000, isto é, quando uma chamada de *Quick Sort* recebe um vetor de tamanho menor ou igual a 1000 o algoritmo *Selection Sort* é executado em seu lugar, sequencialmente. Observando-se os gráficos não percebe-se diferenças significativas nos tempos de execução e nos *speedups* obtidos, tanto entre os ambientes centralizados e distribuídos entre as estratégias empregadas em cada ambiente. O motivo dos resultados observados é que, embora algoritmos recursivos em geral beneficiem ambientes de execução com arquietura distribuída, o *Quick Sort* realiza uma quantidade considerável de computação em cada nível da recursão. Assim, existe pouca concorrência nas listas de threads do ambiente, pois uma vez que um processador consegue trabalho ele passará bastante tempo ocupado, sem tentar acessar a lista de jobs, seja ela local ou compartilhada.

As figuras 33 e 34 apresentam os tempos de execução e os *speedups* obtidos na execução do algoritmo de Smith-Waterman para realizar 1000 alinhamentos de sequências de tamanho 1000, calculando-se blocos de 10$\times$10 e de 20$\times$20 células, respectivamente, da matriz M (vide Seção 7.2.2) em cada thread. Como pode-se perceber, as execuções sobre a implementação distribuída fornecem os menores tempos de execução e os melhores *speedups*, mesmo quando o algoritmo RANDOM é utilizado, demonstrando a maior eficiência desta implementação na minimização dos sobrecustos. Contudo, conforme aumenta-se o tamanho dos blocos de células, a granularidade dos threads aumenta e o sobrecusto relativo diminui, de forma que a implementação centralizada se aproxima do desempenho fornecido pelo ambiente distribuído quando considera-se blocos de 20$\times$20 células.

As figuras 35 e 36 apresentam os resultados obtidos na multiplicação $C = A \times B$ de matrizes quadradas de 500$\times$500 números de ponto flutuante. Na Figura 35, onde o algoritmo considerado cria um thread para calcular cada

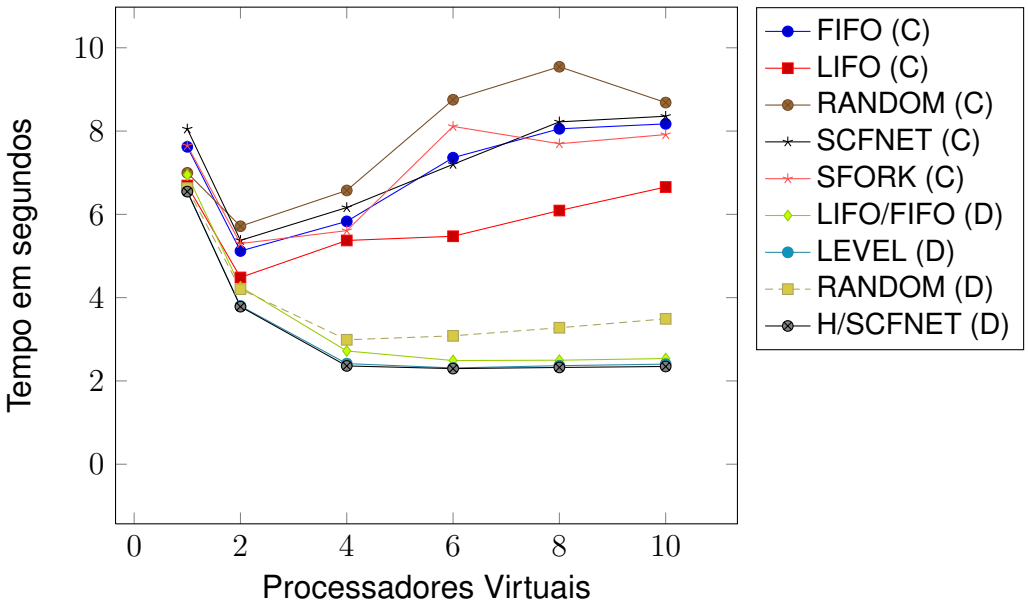


(a) Tempos de execução sobre 1, 2, 4, 6, 8, e 10 processadores virtuais.

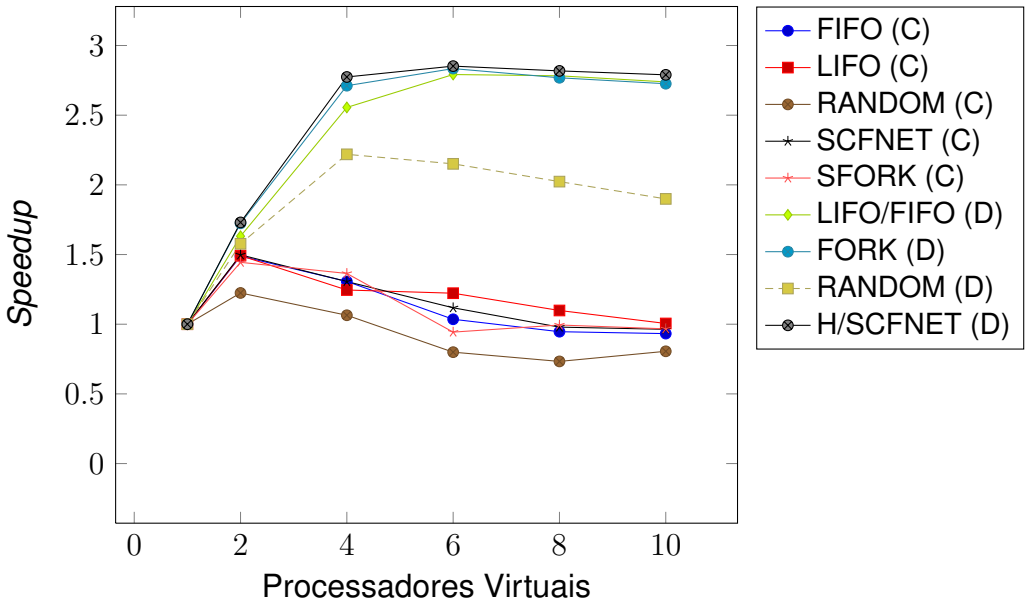


(b) Speedups relativos aos tempos de execução com 1 processador virtual.

Figura 32: Ordenação de 1.000.000 de elementos com o algoritmo *Quick Sort* (*threshold* = 1000) sobre as implementações de Anahy3.

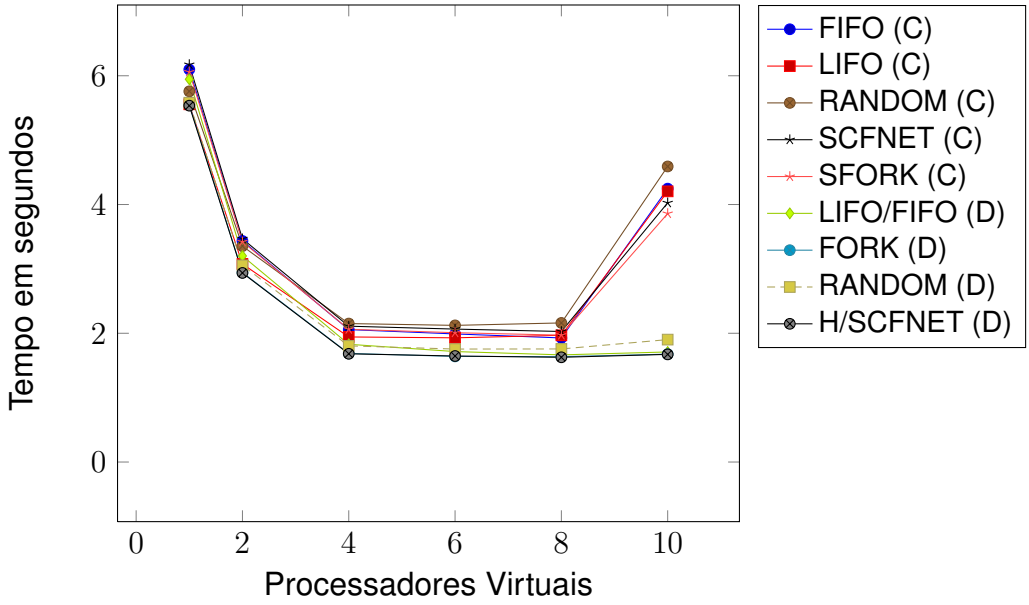


(a) Tempos de execução sobre 1, 2, 4, 6, 8, e 10 processadores virtuais.

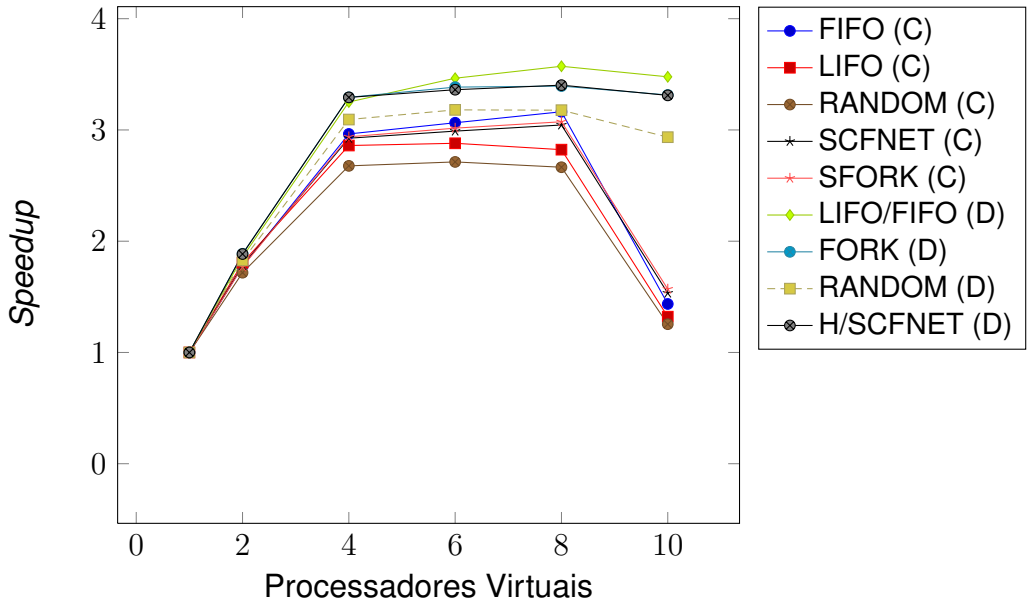


(b) Speedups relativos aos tempos de execução com 1 processador virtual.

Figura 33: Execução do Algoritmo de Smith-Waterman com sequências de tamanho 1000 e blocos de tamanho 10×10 sobre as implementações de Anahy3.



(a) Tempos de execução sobre 1, 2, 4, 6, 8, e 10 processadores virtuais.



(b) Speedups relativos aos tempos de execução com 1 processador virtual.

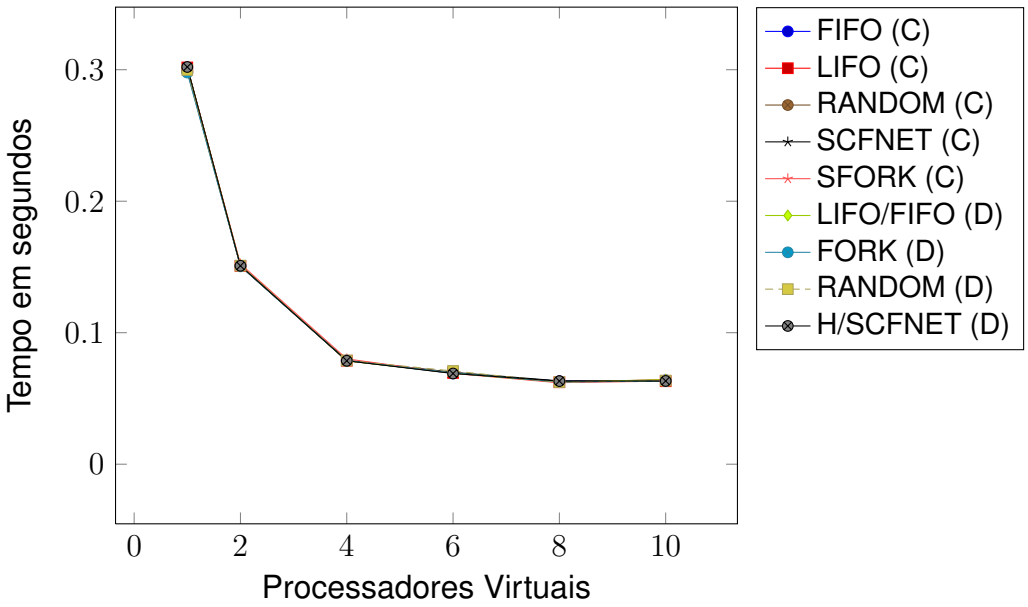
Figura 34: Execução do Algoritmo de Smith-Waterman com sequências de tamanho 1000 e blocos de tamanho 20×20 sobre as implementações de Anahy3.

linha da matriz C, não é possível observar diferenças significativas nos tempos de execução ou nos *speedups* obtidos pelos ambientes/algoritmos utilizados, dado que a maneira com que os threads são criados gera baixa concorrência pelo acessos na(s) lista(s) de threads. Contudo, na Figura 36 consideramos uma implementação que cria um thread para calcular cada elemento da matriz resultante da seguinte forma: o thread *main* cria e sincroniza um thread para cada linha da matriz C, e cada thread desses simplesmente cria e sincroniza um thread para calcular cada elemento daquela linha. Dada a maneira com que o problema é paralelizado neste segundo caso, toda a computação do problema se concentra nos threads mais profundos no grafo, de forma que os threads menos profundos servem simplesmente para quebrar e distribuir o problema. O ambiente distribuído se beneficia desta estratégia, como observado nos tempos apresentados pelos algoritmos “FORK (D)” e “H/SCFNET (D)”. Esses algoritmos ainda fornecem os melhores *speedups* dentre as combinações de ambientes e algoritmos experimentadas. A combinação que chamamos de “LIFO/FIFO (D)”, no entanto, um desempenho ruim nesse segundo caso, uma vez que seu desempenho depende da ordem de execução dos threads e esta implementação do problema não beneficiou essa estratégia sobre o ambiente distribuído.

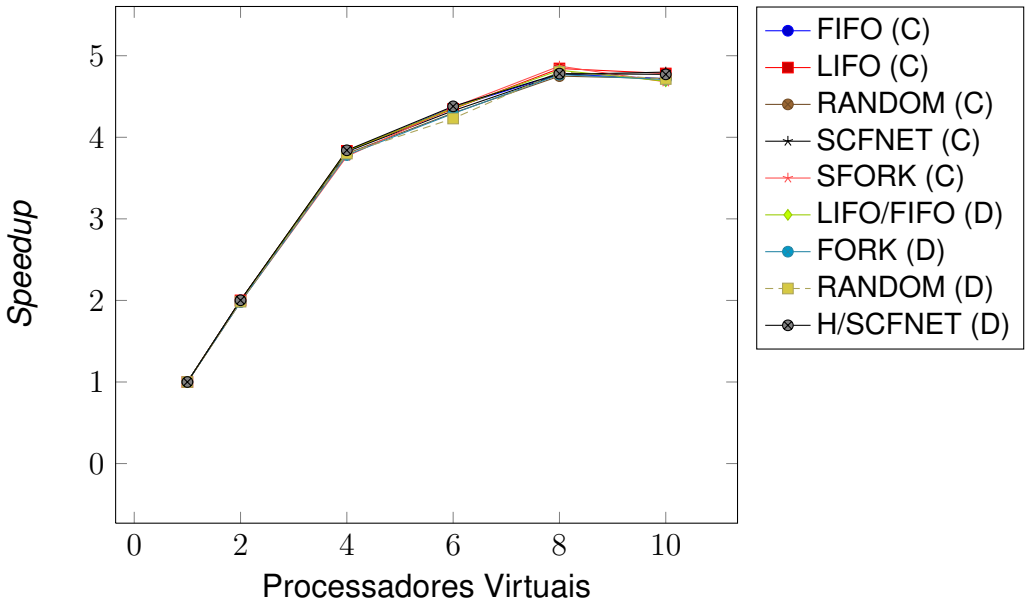
7.3.1 Comparativo com outros ambientes

Nesta seção iremos comparar o desempenho das mesmas aplicações descritas na Seção 7.2 executadas sobre o ambiente de execução de Anahy3, com a sua execução sob os mesmos parâmetros, sobre os ambientes de OpenMP, Cilk Plus e Threading Building Blocks, as duas últimas ferramentas mantidas pela Intel®. Dos resultados obtidos em Anahy3 iremos utilizar a implementação distribuída somada a estratégia de escalonamento que chamamos de H/SCFNET, a qual, além de ser uma aplicação quase que direta do algoritmo empregado no escalonamento de DAGs, forneceu, nos testes anteriores, alguns dos melhores resultados de escalonamento. A título de comparação iremos também incluir os resultados do escalonamento randômico sobre a implementação distribuída de Anahy3. Assim, quando nos referirmos aos resultados de Anahy3 estaremos falando dos resultados da estratégia que chamamos de H/SCFNET, do contrário enfatizaremos que o resultado provém do escalonamento randômico. Para todas as ferramentas, em todos os testes que seguem, foram considerados os mesmos números de processadores virtuais utilizados nos testes de Anahy3: 1, 2, 4, 6, 8 e 10 PVs. Todos os programas, bem como o próprio Anahy3, foram compilados com o compilador *icpc* 13.0.

A Figura 37 apresenta os resultados de execução do algoritmo *Quick Sort*, considerando-se vetores gerados randomicamente com 1.000.000 elementos e

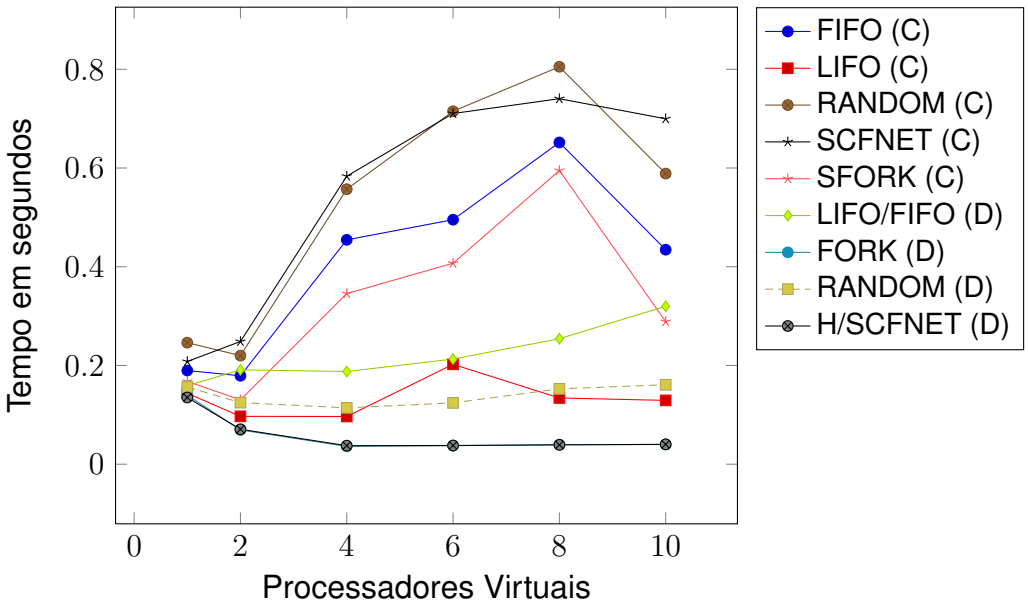


(a) Tempos de execução sobre 1, 2, 4, 6, 8, e 10 processadores virtuais.

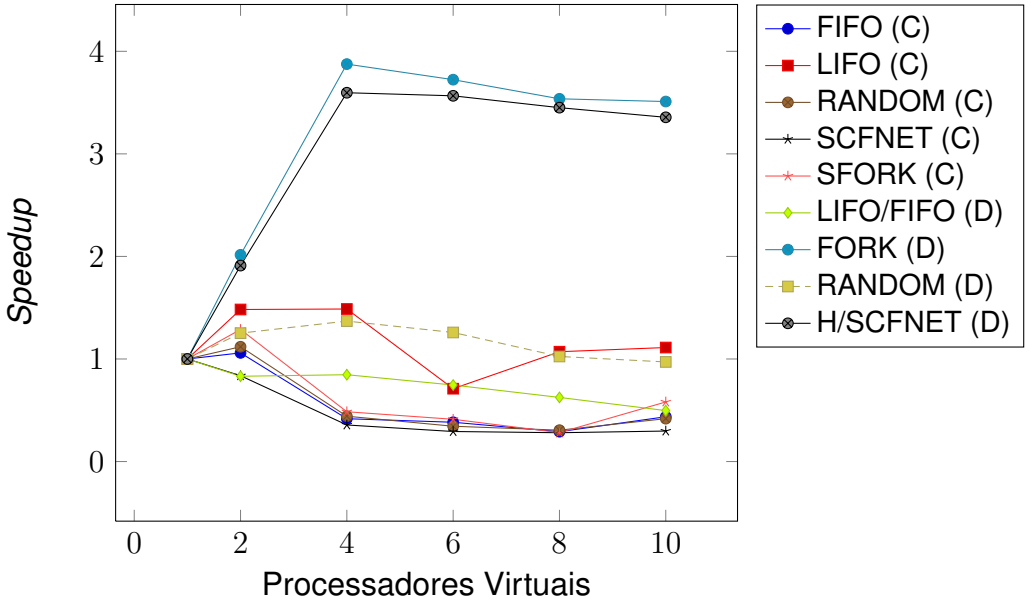


(b) Speedups relativos aos tempos de execução com 1 processador virtual.

Figura 35: Multiplicação de matrizes 500×500 com o cálculo paralelo de *cada linha* da matriz resultante sobre as implementações de Anahy3.



(a) Tempos de execução sobre 1, 2, 4, 6, 8, e 10 processadores virtuais.



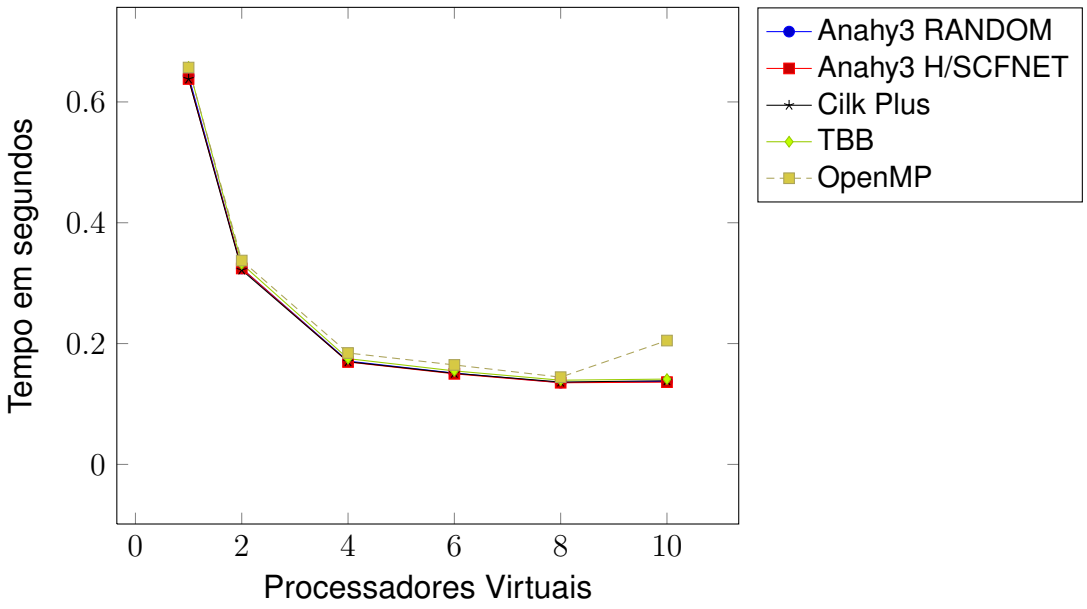
(b) Speedups relativos aos tempos de execução com 1 processador virtual.

Figura 36: Multiplicação de matrizes 500×500 com o cálculo paralelo de *cada elemento* da matriz resultante sobre as implementações de Anahy3.

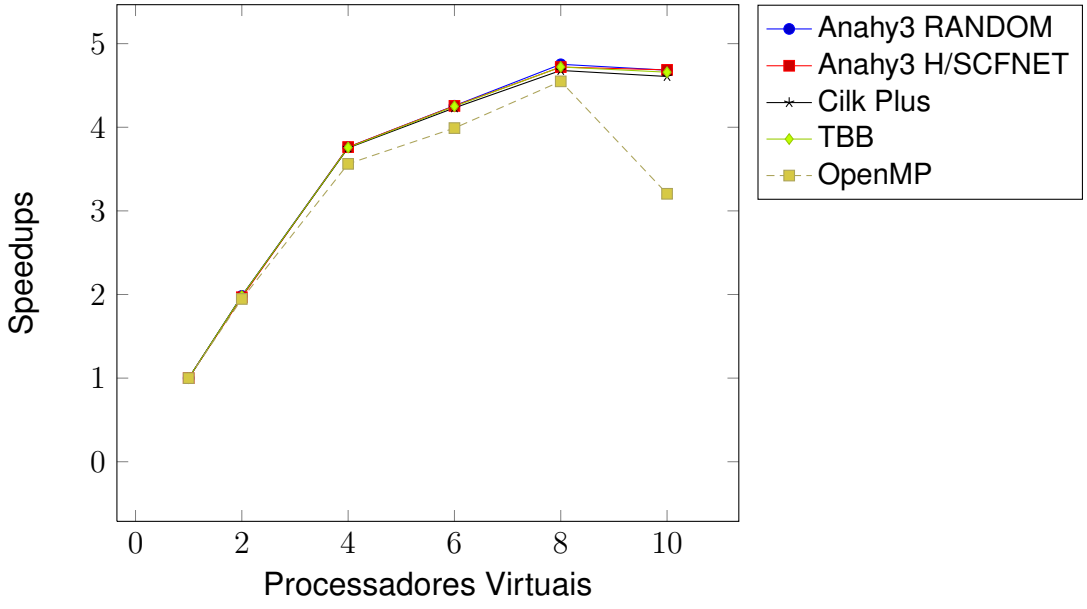
um *threshold* igual a 1000. Como podemos observar, OpenMP fica ligeiramente atrás das demais ferramentas com uma perda significativa de desempenho quando são considerados 10 PVs. Não é possível perceber diferenças significativas entre os desempenhos das demais ferramentas. O desempenho do escalonamento randômico sobre Anahy3 indica que, para esta instância do problema, sobre estes ambientes de execução, a ordem de execução dos threads não é significativa.

As figuras 38 e 39 apresentam os tempos de execução e os *speedups* obtidos na execução do algoritmo de Smith-Waterman para realizar 1000 alinhamentos de sequências de tamanho 1000, com o cálculo concorrente de blocos de tamanho 10×10 e 20×20. Nos testes que utilizam blocos de dimensão 10×10 podemos notar que OpenMP fornece os melhores resultados até 4 PVs e após isso seu desempenho é degradado, enquanto TBB, fornece os tempos de execução e speedups mais estáveis, resultado seguido de perto por Anahy3. Cilk Plus fornece o pior *speedup*, menos estável do que o fornecido pelo escalonamento randômico sobre Anahy neste caso. Quando consideramos blocos de dimensão 20×20 temos tempos de execução bastante semelhantes até 4 PVs, sendo que para números maiores de PVs, OpenMP e Cilk Plus perdem desempenho em relação às demais ferramentas. Nesse segundo caso Anahy3 forneceu os melhores tempos de execução para 4 e 6 VPs seguindo com tempos muito próximos a TBB, ferramenta que oferece os melhores índices de *speedup*, em virtude da maneira semelhante como a concorrência pode ser descrita nestas ferramentas. é Novamente o escalonamento randômico sobre Anahy3 obteve resultados expressivos, demonstrando que para esta aplicação a ordem de execução dos blocos é menos importante do que manutenção constante do nível de concorrência que é realizada no próprio Anahy3 e em TBB.

As figuras 40 e 41 apresentam os resultados obtidos na multiplicação $C = A \times B$ de matrizes quadradas de 500×500 números de ponto flutuante. Novamente, na Figura 40 é criado apenas um nível de threads para calcular cada linha da matriz C concorrentemente, enquanto que na Figura 41 são criados 2 níveis de threads, o primeiro para distribuir o problema e o segundo, mais profundo, para calcular cada elemento da matriz C em um thread separado. No primeiro caso (Figura 40) podemos observar que as ferramentas Cilk Plus e OpenMP, as quais possuem implementações altamente eficientes de primitivas do tipo *parallel for*, fornecem os melhores tempos de execução. Apesar disso Anahy3 fornece os melhores *speedups*, inclusive quando é considerado o escalonamento randômico; TBB fornece resultados semelhantes aos de Anahy3. Quando consideramos o caso da Figura 41, com dois níveis de threads, as execuções sobre OpenMP resultam nos melhores tempos de execução, porém fornecem

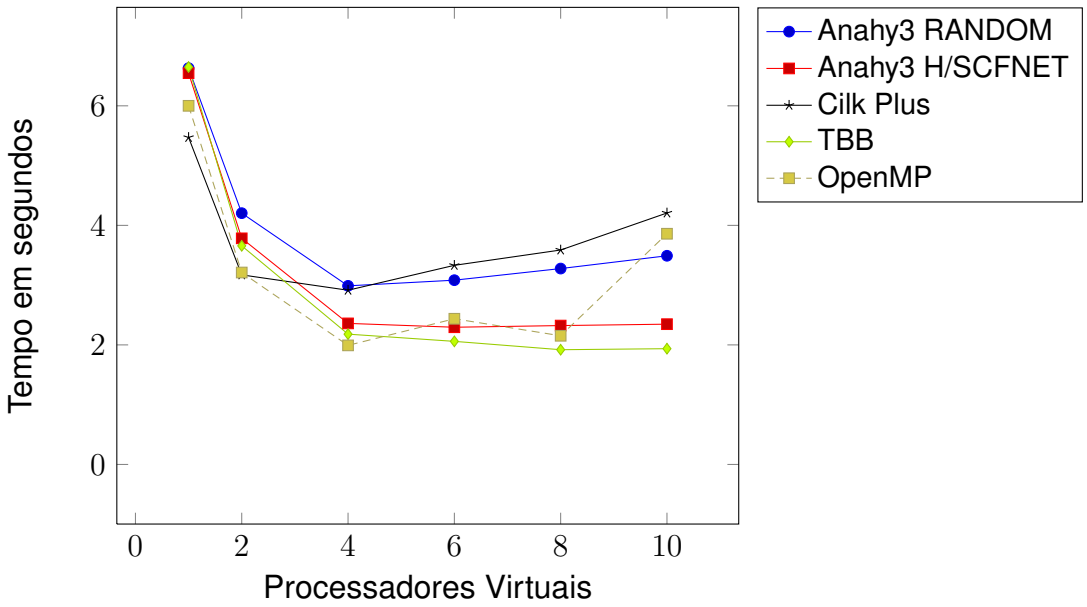


(a) Tempos de execução sobre 1, 2, 4, 6, 8, e 10 processadores virtuais.

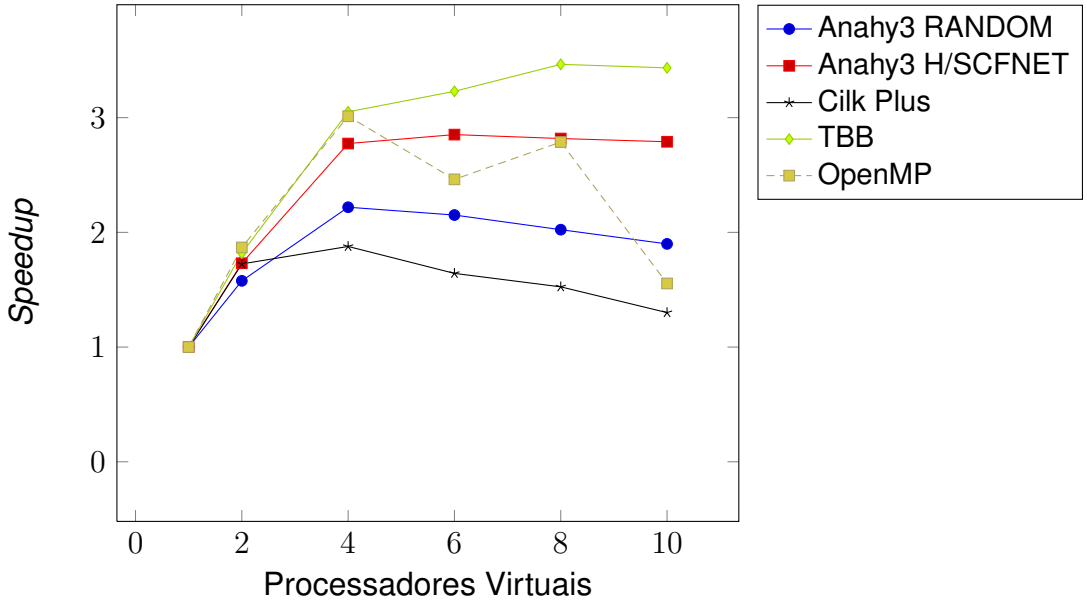


(b) Speedups relativos aos tempos de execução com 1 processador virtual.

Figura 37: Ordenação de 1.000.000 de elementos com o algoritmo *Quick Sort* (*threshold* = 1000) sobre diversas ferramentas.

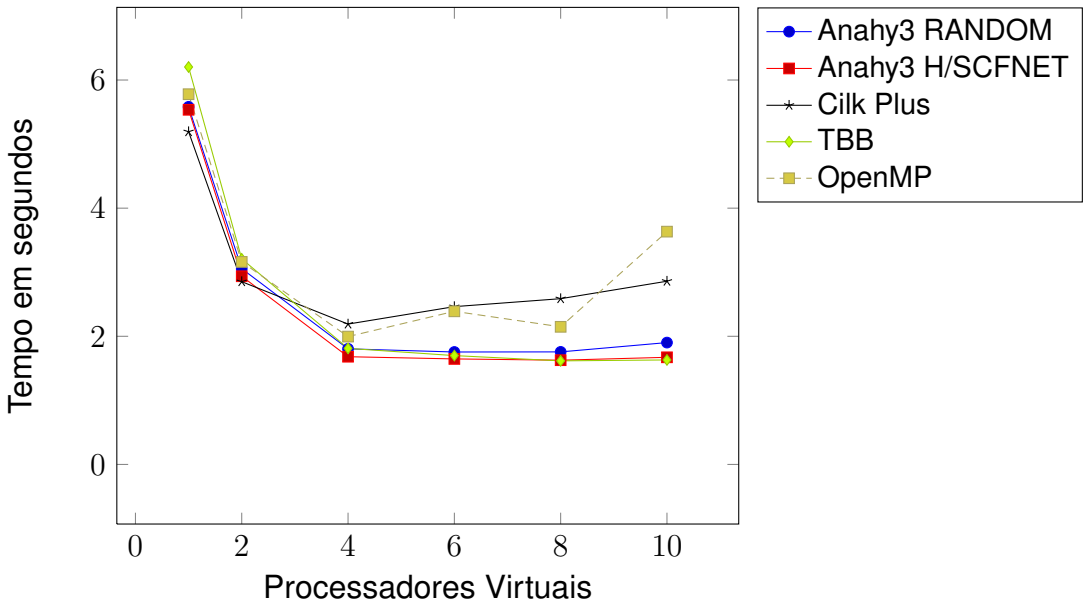


(a) Tempos de execução sobre 1, 2, 4, 6, 8, e 10 processadores virtuais.

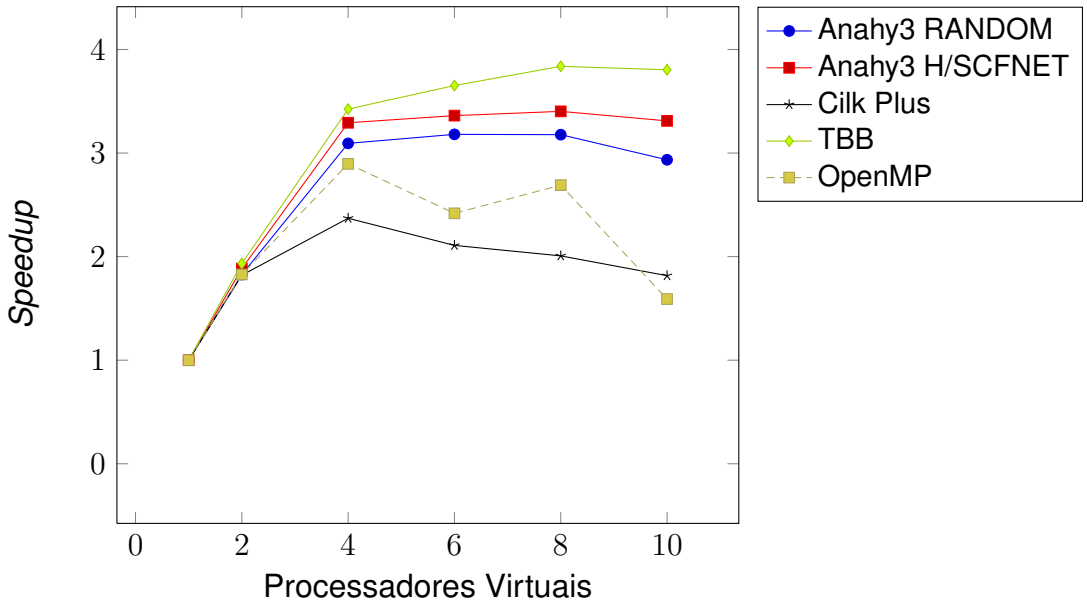


(b) Speedups relativos aos tempos de execução com 1 processador virtual.

Figura 38: Execução do Algoritmo de Smith-Waterman com sequências de tamanho 1000 e blocos de tamanho 10×10 sobre diversas ferramentas.



(a) Tempos de execução sobre 1, 2, 4, 6, 8, e 10 processadores virtuais.



(b) Speedups relativos aos tempos de execução com 1 processador virtual.

Figura 39: Execução do Algoritmo de Smith-Waterman com sequências de tamanho 1000 e blocos de tamanho 20×20 sobre diversas ferramentas.

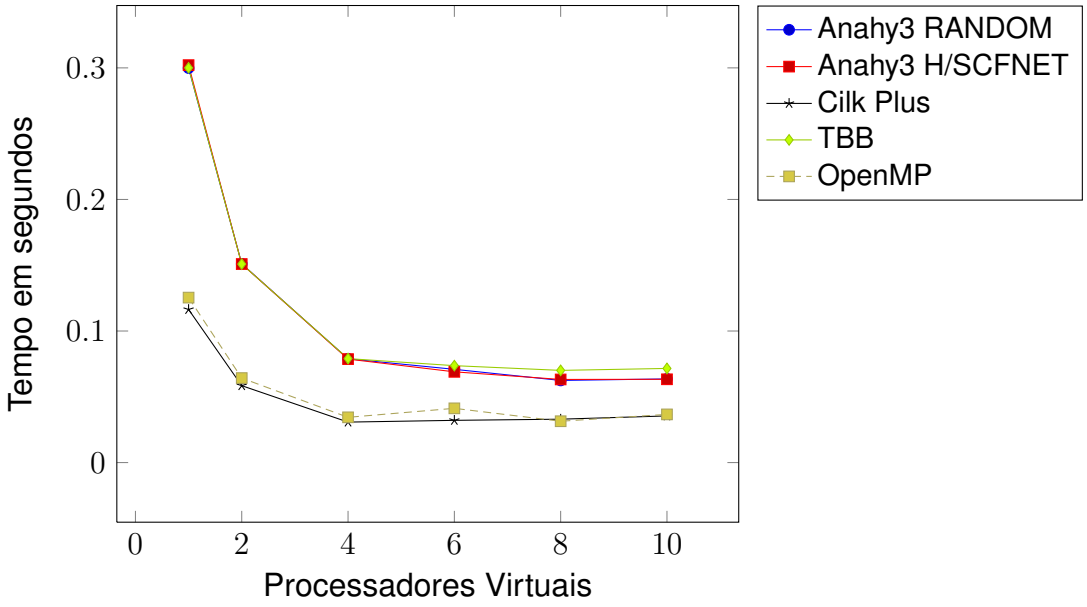
speedups bastante inferiores aos das demais ferramentas. Por outro lado, TBB fornece tempos de execução maiores que os das demais ferramentas, mas oferece os melhores speedups para este estudo de caso. Anahy fornece resultados muito parecidos, ligeiramente inferiores, aos de Cilk Plus. Já os escalonamentos randômicos sobre Anahy3 fornecem tempos de execução e *speedups* ruins, indicando que, para este estudo de caso, o emprego de uma estratégia de escalonamento consciente da estrutura da aplicação resulta em execuções mais eficientes.

Os *speedups* superlineares observados nos resultados das execuções podem ser justificados pela melhora na utilização da memória *cache* dos processadores físicos conforme aumentamos o número de processadores virtuais, bem como pela utilização do *Hyper-Threading* habilitado nos processadores da máquina utilizada nos testes práticos.

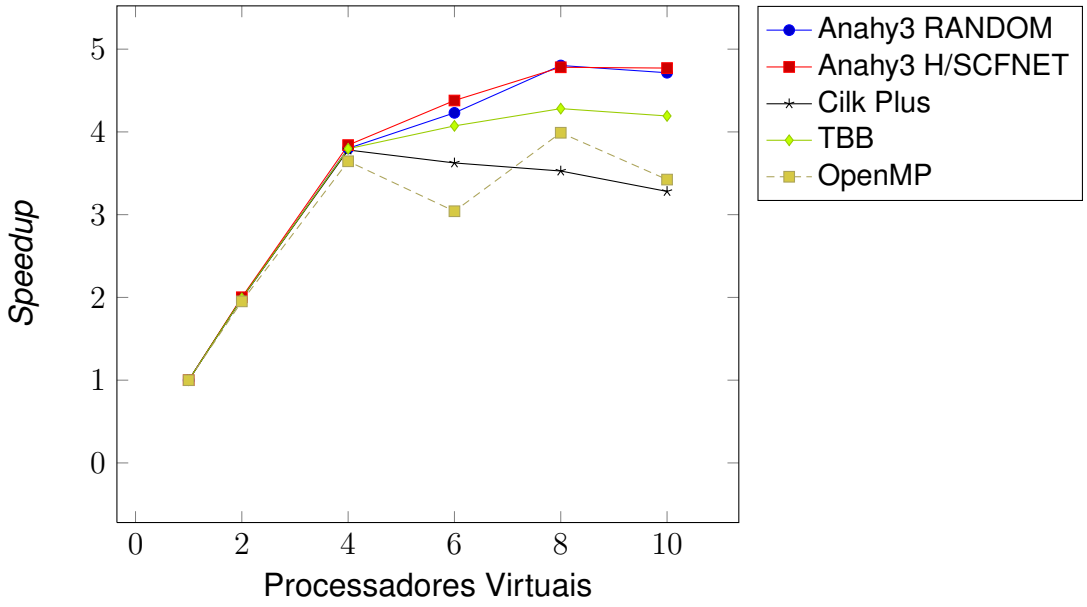
7.4 Conclusões

Neste capítulo foram realizados diversos estudos de caso baseados tanto em simulações de ambientes de execução multithread quanto na execução de aplicações de teste sobre ambientes reais. Foi possível avaliar o desempenho de diferentes estratégias de escalonamento no núcleo de execução de Anahy3, nova implementação do modelo Anahy (CAVALHEIRO et al., 2007) realizada para a execução deste trabalho. Foi também possível analisar a competitividade de Anahy3 comparando seus resultados nas aplicações de teste com os resultados das mesmas aplicações executadas sobre ambientes proeminentes na academia e na indústria como OpenMP, Cilk Plus e TBB.

Por meio das simulações pudemos avaliar o impacto teórico da aplicação estratégias de escalonamento de lista em ambientes multithread com diferentes modos de execução (*Work-First* e *Help-First* com ou sem migração de threads). Nestas simulações não observamos diferenças significativas entre os melhores escalonamentos estáticos de DAGs e os melhores escalonamentos dinâmicos de DCGs, o que reforça, experimentalmente, a aplicabilidade e a eficiência de algoritmos de lista em ambientes multithread dinâmicos. Além disso, pudemos medir a eficiência de certos algoritmos de escalonamento comparando seus desempenhos com o obtido por escalonamentos onde prioridades são atribuídas a threads de um DCG de forma randômica. Desta forma, pudemos perceber que ao considerarmos um ambiente de execução multithread como o de Anahy3, com PVs *Help-First*, sem migração, os tempos de execução gerados por estratégias de escalonamento como FIFO e SCFNET (as quais aplicamos, de fato, em Anahy3) são significativamente menores que os gerados pelo algoritmo

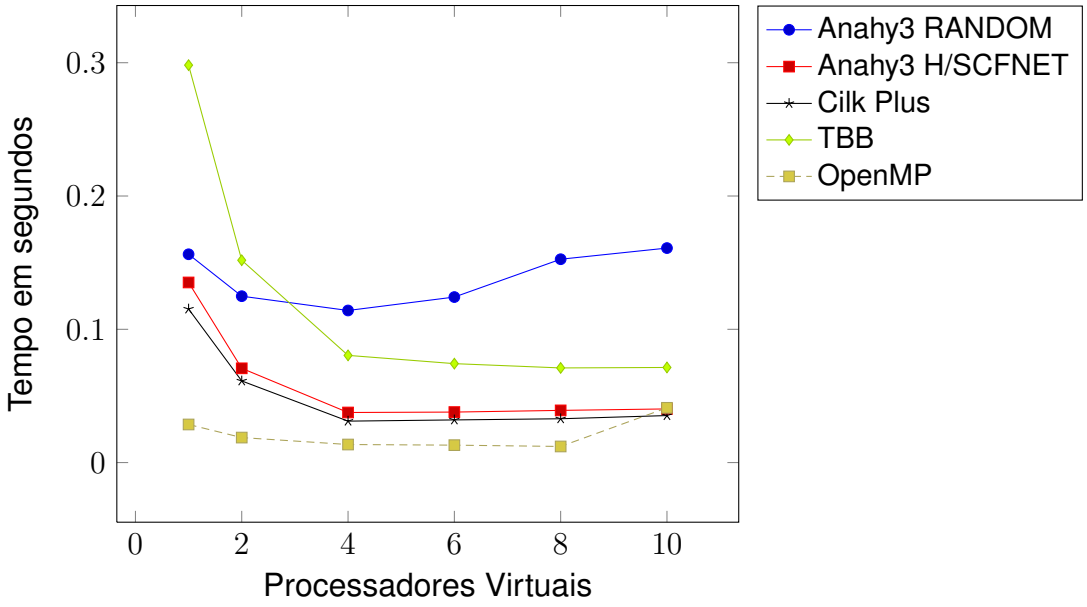


(a) Tempos de execução sobre 1, 2, 4, 6, 8, e 10 processadores virtuais.

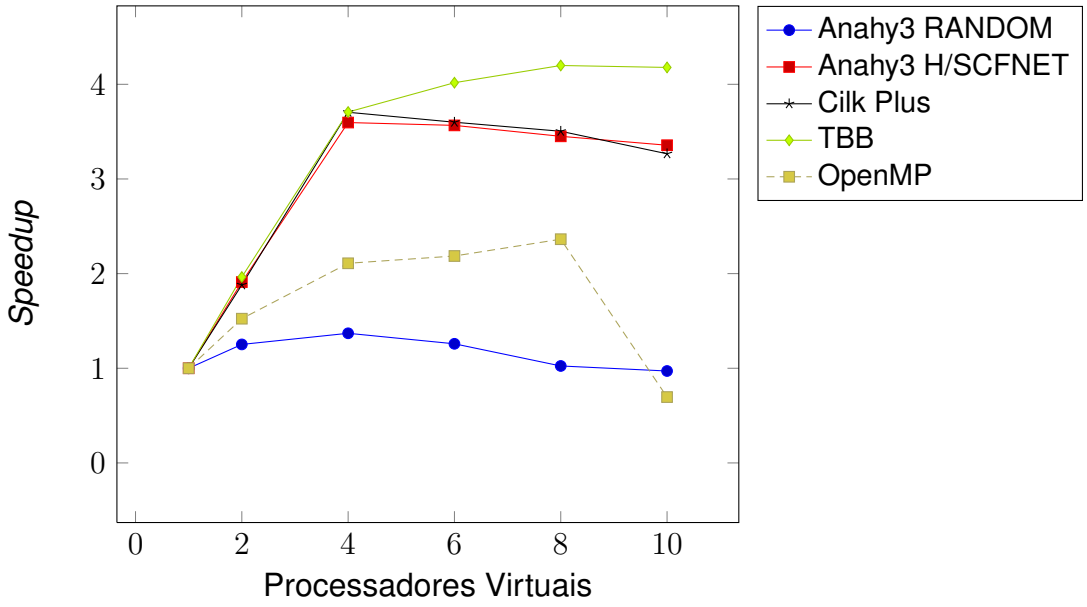


(b) Speedups relativos aos tempos de execução com 1 processador virtual.

Figura 40: Multiplicação de matrizes 500×500 com o cálculo paralelo de *cada linha* da matriz resultante sobre diversas ferramentas.



(a) Tempos de execução sobre 1, 2, 4, 6, 8, e 10 processadores virtuais.



(b) Speedups relativos aos tempos de execução com 1 processador virtual.

Figura 41: Multiplicação de matrizes 500×500 com o cálculo paralelo de *cada elemento* da matriz resultante sobre diversas ferramentas.

randômico. Estes resultados reforçam a importância de elaborar uma estratégia de escalonamento, ainda mais quando o modo de execução do ambiente impõe restrições como a impossibilidade de migração.

Nas execuções das aplicações de teste pudemos observar o impacto de diferentes estratégias de escalonamento sobre diferentes arquiteturas de software implementadas no ambiente de execução de Anahy3. Os resultados demonstram que, no universo dos experimentos realizados, algoritmos de lista podem fornecer bons resultados e *speedups* eficientes em ambientes multithread dinâmicos, independentemente da organização interna das estruturas do ambiente. Contudo, em certos cenários uma lista centralizada de threads representa um gargalo de desempenho. Os diferentes tempos obtidos para estratégias de escalonamento similares sobre as diferentes implementações de Anahy realizadas (com arquitetura centralizada e distribuída) indicam que as técnicas empregadas na implementação do ambiente de execução geram grandes impactos nos escalonamentos e no tempo total de execução das aplicações de teste. Diversos resultados apresentados demonstram que a implementação com arquitetura distribuída consegue gerenciar de maneira mais eficiente os sobrecustos do ambiente de execução, o que traz benefícios maiores em relação à implementação centralizada conforme diminui-se a granularidade do paralelismo das aplicações. Ainda assim, em arquiteturas físicas com um grau de paralelismo maior do que o oferecido pela máquina de teste utilizada neste trabalho, certas aplicações que não privilegiem a arquitetura de *software* distribuída podem gerar uma quantidade de sobrecustos alta em função do alto número de tentativas mal-sucedidas de roubo de trabalho, o que justificaria a implementação de uma arquitetura híbrida de software em Anahy3.

Nas comparações de Anahy3 com outras ferramentas foi possível constatar que Anahy3 foi capaz de oferecer resultados competitivos, muitas vezes, melhores que os resultados fornecidos por OpenMP, Cilk Plus e TBB, algumas das principais ferramentas na indústria e da academia. Estes resultados provêm de uma implementação simples e eficiente do modelo Anahy, com estruturas internas (`AnahySmartHeap`/`AnahySmartStack`) altamente otimizadas. Além de bons tempos de execução, Anahy3 também apresentou *speedups* expressivos, bastante semelhantes aos fornecidos por TBB (em certos casos, até melhores), o que destaca a escalabilidade de Anahy3, uma característica de suma importância dada a tendência de termos máquinas com cada vez mais *cores* dentro de um mesmo *chip*.

8 CONCLUSÃO

Este trabalho apresentou aspectos teóricos e práticos da aplicação de algoritmos de lista em ambientes multithread dinâmicos. Estudamos o funcionamento básico da estratégia de escalonamento de lista e diversos algoritmos de prioridade no Capítulo 2. No Capítulo 3 vimos que a aplicação destas estratégias deve contemplar a adaptação do núcleo de escalonamento em função da unidade de escalonamento manipulada no grafo de uma aplicação multithread, o thread. Vimos também que ambientes em multithread dinâmicos os algoritmos de escalonamento devem empregar heurísticas para calcular a prioridade dos threads em tempo de execução, durante as operações de escalonamento, considerando apenas a porção conhecida do grafo naquele momento. Estudamos no Capítulo 4 diversas decisões que podem ser tomadas na implementação de ambientes multithread e que estas decisões impactam de diferentes formas tanto no escalonamento das aplicações quanto nos sobrecustos gerados pelo ambiente em tempo de execução. Uma nova implementação para o ambiente Anahy foi introduzida no Capítulo 5 enquanto outras ferramentas importantes de programação multithread foram apresentadas no Capítulo 6.

Os testes realizados no Capítulo 7 nos forneceram diversos resultados importantes. Observamos, nas simulações realizadas, que os algoritmos de lista oferecem tempos de escalonamento em ambientes multithread dinâmicos que são, *no máximo*, 17% maiores que os *melhores* escalonamentos estático de um DAG que representa a mesma aplicação. Estes resultados são satisfatórios, uma vez que o escalonamento multithread é não-clarividente, isto é, toma decisões sem conhecer o futuro da aplicação, assim o aumento no tempo de execução é compensado por não existir custo de leitura e montagem prévia do grafo da aplicação. Além disso, mbientes multithread dinâmicos temos a vantagem de podermos executar, com relativa eficiência, aplicações altamente dinâmicas, onde o número de threads a serem criados e as relações de precendência entre estes threads são conhecidos em tempo de execução do programa.

Por meio dos escalonamentos randômicos pudemos constatar que, tanto na teoria quando na prática, isto é, tanto nas simulações quanto na execução de aplicações multithread reais, é importante que o ambiente de execução empregue uma estratégia de escalonamento elaborada, consciente da estrutura do grafo da aplicação. Isso é especialmente importante em ambientes multithread com modos de execução mais restritivos, como é o caso de Anahy3 onde não há migração de threads parcialmente executados. Por outro lado, executamos diversas aplicações de teste sobre Anahy3, e em várias execuções reais foi possível constatar que em certas aplicações a ordem de execução de atividades concorrentes não é tão importante quanto a eficiência da implementação do ambiente. Ainda no Capítulo 7 pudemos verificar a eficiência e a escalabilidade fornecida por Anahy3 para as aplicações testadas, tanto sobre uma implementação com lista de threads centralizada quanto com listas dsitribuídas. Também pudemos constatar a competitividade e, em alguns momentos, a superioridade de Anahy3 contra OpenMP, Cilk Plus e TBB, ferramentas de programação multithread consagradas na indústria e na academia.

8.1 Contribuições

Diversas análises foram feitas com o objetivo de mensurar a eficiência de algoritmos de lista no escalonamento não-clarividente de aplicações multithread. Todos o resultados teóricos e práticos apresentados neste trabalho reafirmam, acima de tudo, a aplicabilidade de algoritmos de lista e, mais do que isso, sua eficiência no escalonamento de aplicações multithread em ambientes dinâmicos.

Além de comprovar experimentalmente que algoritmos de lista pode ser eficientes em ambientes multithread dinâmicos, este trabalho deixa como legado uma implementação nova e mais eficiente do modelo Anahy, a ferramenta Anahy3. Além disso as estruturas de pilha e *heap* binário criadas especialmente para o núcleo de Anahy3 tiveram sua eficiência comprovada e podem ser utilizadas para propósito geral. Novas arquiteturas e novos algoritmos de escalonamento foram adicionados à ferramenta de simulação AKSSim, que também teve sua API incrementada.

8.2 Trabalhos Futuros

Futuramente desejamos realizar mais testes com as aplicações de teste exploradas neste trabalho, bem como outras aplicações interessantes, sobre máquinas com um número maior de processadores do que a máquina utilizada

neste trabalho. Acreditamos que, conforme aumentamos o nível de paralelismo, para certos padrões de paralelismo (como aplicações baseadas em paralelismo de dados, por exemplo) onde a estrutura do grafo não colabora com a distribuição de threads em um ambiente com múltiplas listas, a utilização de uma lista centralizada fornecerá menos sobrecustos de execução, o que justificaria a implementação de uma arquitetura híbrida como a de TBB no núcleo de Anahy3. Um cenário como este também justificaria a implementação de uma primitiva do tipo *parallel for* que inclua recursos para melhorar o balanceamento de carga, como acontece na primitiva `cilk_for` da ferramenta Cilk Plus.

Como outro trabalho futuro, desejamos evoluir a ferramenta AKSSim de modo a incluir a simulação de arquiteturas com acesso não uniforme à memória (NUMA – *Non-Uniform Memory Access*). Para tanto, devemos estender, também, a interface da ferramenta, de modo a possibilitar a associação de custos às arestas dos DCGs gerados de forma parametrizada. Também desejamos testar o ambiente Anahy3 sobre arquiteturas NUMA. Neste sentido, o fato da ferramenta aplicar algoritmos de lista derivados do escalonamento de DAGs, algoritmos estes que já consideram os custos associados às arestas do grafo de dependências para o cálculo de atributos de prioridade, facilita a otimização de Anahy3 para arquiteturas NUMA.

Um trabalho futuro que já está em curso é a documentação apropriada da ferramenta Anahy3 e seu licenciamento para um futuro lançamento oficial. No presente momento, todas as implementações de Anahy3 utilizadas neste trabalho, bem como as aplicações de teste utilizadas no capítulo anterior e o código da ferramenta AKSSim estão disponíveis para download em: `https://github.com/cicerocamargo/FinalDissertation`.

REFERÊNCIAS

AYGUADÉ, E.; COPTY, N.; DURAN, A.; HOEFLINGER, J.; LIN, Y.; MASSAIOLI, F.; TERUEL, X.; UNNIKRISHNAN, P.; ZHANG, G. The design of openmp tasks. **Parallel and Distributed Systems, IEEE Transactions on**, [S.l.], v.20, n.3, p.404–418, 2009.

BAKSHI, A.; PRASANNA, V.; REICH, J.; LARNER, D. The Abstract Task Graph: A Methodology for Architecture-Independent Programming of Networked Sensor Systema. In: WORKSHOP ON END-TO-ED, SENSE-AND-RESPOND SYSTEMS, APPLICATIONS, AND SERVICES, 2005. **Anais...** [S.l.: s.n.], 2005.

BLUMOFE, R. D.; AL. et. Cilk: an efficient multithreaded runtime system. **ACM SIGPLAN Not.**, [S.l.], v.30, n.8, Aug. 1995.

BLUMOFE, R.; LEISERSON, C. Scheduling multithreaded computations by work stealing. In: FOUNDATIONS OF COMPUTER SCIENCE, 1994 PROCEEDINGS., 35TH ANNUAL SYMPOSIUM ON, 1994. **Anais...** [S.l.: s.n.], 1994. p.356–368.

CAMARGO, C. A. S.; ARAUJO, A. S. de; CAVALHEIRO, G. G. H. AKSSim: uma ferramenta para a análise de algoritmos de lista em ambientes multithread dinâmicos. In: ESCOLA REGIONAL DE ALTO DESEMPENHO. ERAD 2011. ANAIS DA, 11., 2011. **Anais...** [S.l.: s.n.], 2011.

CAMARGO, C. A. S.; CAVALHEIRO, S. A. d. C.; FOSS, L.; CAVALHEIRO, G. G. H. A Graph Grammar to Transform a Dataflow Graph into a Multithread Graph and its Application in Task Scheduling. **Revista de Informática Teórica e Aplicada**, [S.l.], v.20, 2013.

CAVALHEIRO, G. G. H. Introdução a Programação Paralela e Distribuída. In: I ESCOLA REGIONAL DE ALTO DESEMPENHO, 2001, Gramado – RS. **Anais...** [S.l.: s.n.], 2001.

CAVALHEIRO, G. G. H.; GASPARY, L. P.; CARDOZO, M. A.; CORDEIRO, O. C. Anahy: A Programming Environment for Cluster Computing. In: VII

HIGH PERFORMANCE COMPUTING FOR COMPUTATIONAL SCIENCE, 2007, Berlin. **Anais...** Springer-Verlag, 2007. (LNCS 4395).

CAVALHEIRO, G. G. H.; VIÇOSA, E. Athreads: a dataflow programming interface for multiprocessors. In: LTPD 2007, 2007, Gramado-RS. **Anais...** [S.l.: s.n.], 2007.

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

CLARK, W.; GANTT, H. **The Gantt Chart**: A Working Tool of Management. [S.l.]: Pitman, 1935.

COFFMAN, E.; GRAHAM, R. Optimal scheduling for two-processor systems. **Acta Informatica**, [S.l.], v.1, n.3, p.200–213, 1972.

COFFMAN JR, E.; DENNING, P. **Operating systems theory**. [S.l.]: Prentice Hall Professional Technical Reference, 1973.

CONWAY, M. A multiprocessor system design. In: NOVEMBER 12-14, 1963, FALL JOINT COMPUTER CONFERENCE, 1963. **Proceedings...** [S.l.: s.n.], 1963. p.139–146.

CONWAY, R. W.; MAXWELL, W. L.; MILLER, L. **Theory of Scheduling**. 1. ed..ed. [S.l.]: Addison-Wesley Publishing Company, 1967.

FEITELSON, D. Job scheduling in multiprogrammed parallel systems. **IBM Research Report**, [S.l.], v.19790, 1997.

GRAHAM, R. Bounds on multiprocessing timing anomalies. **SIAM Journal on Applied Mathematics**, [S.l.], v.17, n.2, p.416–429, 1969.

GRAHAM, R. Bounds on the performance of scheduling algorithms. **Computer and Job-Shop Scheduling Theory**, [S.l.], p.165–227, 1976.

GRAHAM, R. L. Bounds for Certain Multiprocessor Anomalies. **The Bell Syst. Tech. J.**, [S.l.], v.CLV, n.9, 1966.

GUO, Y.; BARIK, R.; RAMAN, R.; SARKAR, V. Work-first and help-first scheduling policies for async-finish task parallelism. In: PARALLEL & DISTRIBUTED PROCESSING, 2009. IPDPS 2009. IEEE INTERNATIONAL SYMPOSIUM ON, 2009. **Anais...** [S.l.: s.n.], 2009. p.1–12.

HOARE, C. Quicksort: Algorithm 64. **Comm. ACM**, [S.l.], v.4, n.7, p.321–322, 1961.

HU, T. Parallel sequencing and assembly line problems. **Journal of Operations Research**, [S.l.], v.9, p.841–848, November-December 1961.

INTEL, C. **Using Intel(R) Cilk(TM) Plus**. Acessado em Janeiro/2012, http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/cpp/mac/cref_cls/common/cilk_bk_using_cilk.htm.

INTEL, C. **Intel(R) Threading Building Blocks**: Reference Manual. Acessado em Janeiro/2012, http://software.intel.com/sites/products/documentation/hpc/tbb/referencev2.pdf.

JOHNSTON, W. M.; HANNA, J. R. P.; MILLAR, R. J. Advances in dataflow programming languages. **ACM Comput. Surv.**, New York, NY, USA, v.36, n.1, p.1–34, 2004.

KNUTH, D. Sorting by Selection. **The art of computer programming**, [S.l.], v.3, p.138–141, 1997.

LEUNG, J. **Handbook of scheduling**: algorithms, models, and performance analysis. [S.l.]: Chapman & Hall/CRC, 2004. v.1.

NICHOLS, B.; BUTTAR, D.; FARRELL, J. P. **Pthreads Programming**. 2.ed. Cambridge: O'Reilly, 1998.

POWELL, M. L.; KLEIMAN, S. R.; BARTON, S.; SHAH, D.; STEIN, D.; WEEKS, M. SunOS Multi-thread Architecture. In: IN PROCEEDINGS OF THE WINTER 1991 USENIX CONFERENCE, 1991. **Anais...** [S.l.: s.n.], 1991. p.65–80.

REINDERS, J. **Intel threading building blocks**. 1.ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

SARDESAI, S.; MCLAUGHLIN, D.; DASGUPTA, P. Distributed cactus stacks: Runtime stack-sharing support for distributed parallel programs. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, 1998. **Proceedings...** [S.l.: s.n.], 1998.

SMITH, T. F.; WATERMAN, M. S. Identification of Common Molecular Subsequences. **Journal of Molecular Biology**, [S.l.], v.215, p.403–410, 1990.

VALIANT, L. G. A bridging model for parallel computation. **Communications of ACM**, [S.l.], v.33, n.8, 1990.