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

Dissertação

Escalonamento dinâmico em nível aplicativo sensível à arquitetura e às dependências de dados entre as tarefas

Rodolfo Migon Favaretto

Pelotas, 2014

Rodolfo Migon Favaretto

Escalonamento dinâmico em nível aplicativo sensível à arquitetura e às dependências de dados entre as tarefas

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal de Pelotas, como requisito parcial à obtenção do título de Mestre em Ciência da Computação

Orientador: Prof. Dr. Gerson Geraldo Homrich Cavalheiro
Co-orientador: Prof. Dr. Maurício Lima Pilla

Pelotas, 2014

Universidade Federal de Pelotas / Sistema de Bibliotecas
Catalogação na Publicação

F272e Favaretto, Rodolfo Migon

Escalonamento dinâmico em nível aplicativo sensível à arquitetura e às dependências de dados entre as tarefas / Rodolfo Migon Favaretto ; Gerson Geraldo Homrich Cavalheiro, orientador ; Maurício Lima Pilla, coorientador. — Pelotas, 2014.

120 f. : il.

Dissertação (Mestrado) — Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, 2014.

1. Escalonamento em nível aplicativo. 2. Arquiteturas NUMA. 3. Escalonamento de tarefas paralelas. 4. Escalonamento de listas. I. Cavalheiro, Gerson Geraldo Homrich, orient. II. Pilla, Maurício Lima, coorient. III. Título.

CDD : 005.1

Elaborada por Aline Herbstrith Batista CRB: 10/1737

Dedico aos meus pais Hildo e Anadir e ao meu irmão Cássio pelo apoio, incentivo, amor, confiança e motivação incondicional, os quais sempre me impulsionam em direção às vitórias dos meus desafios.
Dedico também a todos aqueles que acreditaram em mim e fizeram parte do processo de desenvolvimento deste trabalho.

AGRADECIMENTOS

Cada pessoa que passa em nossa vida, passa sozinha, é porque cada pessoa é única e nenhuma substitui a outra! Cada pessoa que passa em nossa vida passa sozinha e não nos deixa só porque deixa um pouco de si e leva um pouquinho de nós. Essa é a mais bela responsabilidade da vida e a prova de que as pessoas não se encontram por acaso."

— CHARLES CHAPLIN

Durante esse dois anos só tenho a agradecer a todos que passaram pelo meu caminho e que, com certeza, deixaram um pouco de si. Os momentos de alegria serviram para me permitir acreditar na beleza da vida, e os de sofrimento, serviram para um crescimento pessoal único. É muito difícil transformar sentimentos em palavras, mas serei eternamente grato a vocês, pessoas imprescindíveis para a realização e conclusão deste trabalho.

Agradeço, primeiramente, a Deus que me concebeu a oportunidade de viver e estar dentre a parte da população que tem oportunidade de estudar e um dia vencer.

Aos meus pais Hildo e Anadir, por terem me dado educação, valores, incentivo e, mesmo tão distantes, sempre me oportunizaram momentos de plenitude e apoio familiar incondicionais. A vocês, minha eterna gratidão.

Ao meu orientador Prof. Dr. Gerson G. H. Cavalheiro e ao meu co-orientador Prof. Dr. Maurício Lima Pilla, por acreditarem na minha capacidade, pela disponibilidade, pelos conhecimentos transmitidos, pelo incentivo e pela presteza no auxílio às atividades e discussões sobre o andamento e normatização desta dissertação.

Agradeço também ao Prof. Dr. Adenauer Corrêa Yamin, o qual sempre foi um incentivador de minha vontade de pesquisar e de aprender, pelas valiosas contribuições dadas durante estes dois anos.

Aos meus amigos, em especial ao Marcelo que, mais que um colega de quarto, considero parte da família aqui em Pelotas, ao Rodrigo, grande amigo, por estar sempre presente nos momentos em que precisei, pela ajuda, apoio e incentivo no desenvolvimento deste trabalho e ao Lucas, pela ajuda no desenvolvimento dos *benchmarks*.

Gostaria de agradecer, também, ao Laércio Pilla, pela sua atenção e colaboração por permitir que eu utilizasse sua ferramenta e por todas as dicas e contribuições que veio a acrescentar no desenvolvimento deste trabalho.

Agradeço aos colegas mais próximos e todos os meus amigos do LUPS (*Laboratory of Ubiquitous and Parallel Systems*), sem citar nomes, pelo enorme

aprendizado, apoio e carinho. Com eles percebi que o aprendizado é uma construção diária cujo ingrediente principal é o afeto e a amizade. A este grupo de pesquisa que me acolheu, os meus mais sinceros agradecimentos.

Gostaria ainda de agradecer o privilégio de desfrutar de uma estrutura pública e gratuita de ensino como a da Universidade Federal de Pelotas para minha formação. Gostaria de agradecer, também, à FAPERGS, por financiar os meus estudos durante estes dois anos.

A todos que não citei, mas que de alguma forma ou de outra contribuíram para a minha formação pessoal, profissional ou acadêmica, ficam os meus sinceros agradecimentos e a certeza de que receberão em dobro tudo o que semearam no meu caminho.

A todos vocês, o meu muito obrigado.

Pode-se vencer pela inteligência,
pela habilidade ou pela sorte,
mas nunca sem trabalho.
— A. DESTOEF

RESUMO

FAVARETTO, Rodolfo Migon. **Escalonamento dinâmico em nível aplicativo sensível à arquitetura e às dependências de dados entre as tarefas**. 2014. 120 f. Dissertação (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Computação, Universidade Federal de Pelotas, Pelotas, 2014.

As arquiteturas modernas apresentam múltiplos processadores, compostos por vários núcleos e blocos de memória dedicados, caracterizando as arquiteturas NUMA (*Non-Uniform Memory Access*). NUMA têm como característica as diferentes latências no acesso aos diferentes blocos de memória. Um dos grandes desafios é o desenvolvimento de técnicas eficientes para o escalonamento das tarefas produzidas pelas aplicações paralelas entre os processadores disponíveis, considerando a heterogeneidade desses tempos de acesso à memória. Para tal, o escalonador deve tomar decisões influenciadas por diversos fatores. Um destes fatores diz respeito à questões de localidade dos dados, a decisão de alocar uma tarefa sobre um processador específico passa pela análise dos custos associados ao acesso aos seus dados na estrutura assimétrica de memória. Um outro fator está relacionado às dependências de dados utilizados pelas tarefas, onde heurísticas baseadas em algoritmos de lista podem ser utilizadas para realizar o escalonamento, em nível aplicativo, em ambientes de execução dinâmicos. Neste trabalho, foi concebida uma estratégia de escalonamento dinâmico para aplicações paralelas em arquiteturas NUMA. Esta estratégia foi validada com a realização de uma série de experimentos, onde foi possível aferir o seu desempenho comparando-a com outras ferramentas que empregam escalonamento em nível aplicativo. O objetivo da estratégia desenvolvida foi buscar reduzir o impacto, na execução de aplicações paralelas, das diferentes latências oriundas da distribuição física dos módulos de memória das arquiteturas NUMA. Este trabalho resultou em uma extensão do núcleo de execução do ambiente Anahy, o qual passou a comportar a estratégia proposta, considerando, no momento do escalonamento, as características heterogêneas da arquitetura onde a aplicação está executando. Os resultados obtidos comprovam que a qualidade do escalonamento de Anahy em arquiteturas NUMA melhorou, contribuindo com o aumento de desempenho do ambiente. A estratégia desenvolvida apresentou resultados de desempenho compatíveis com as demais ferramentas.

Palavras-chave: Escalonamento em nível aplicativo, Arquiteturas NUMA, Escalonamento de tarefas paralelas, Escalonamento de listas.

ABSTRACT

FAVARETTO, Rodolfo Migon. **Dynamic scheduling in application level aware of architecture and data dependencies between tasks**. 2014. 120 f. Dissertação (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Computação, Universidade Federal de Pelotas, Pelotas, 2014.

Modern architectures have multiple processors, each of which contains multiple cores, connected to dedicated memory blocks, featuring NUMA (Non-Uniform Memory Access) architectures. NUMA have different latencies in accessing different blocks of distributed memory. A major challenge is to develop an efficient scheduling to the tasks produced by parallel applications among the available processors by considering the heterogeneity of these memory access times. For this, the scheduler must make decisions influenced by several factors. One of these factors is related to issues of data locality, the decision of allocate a task on a specific processor is an assessment of the costs associated to the access to its data on the asymmetric memory structure. Another factor to be considered is related to the data dependences used by tasks, where heuristics based on list algorithms can be used to scheduling these tasks, in application level, in dynamic execution environments. In this work, we designed a dynamic scheduling strategy for parallel applications on NUMA architectures. This strategy was validated by a series of experiments, where it was possible to assess its performance by comparing it against other tools that employ scheduling on application level. The aim of the strategy was reduced the impact, when executing parallel aplications, of the different latencies arising from the physical distribution of memory modules in NUMA architectures. This work resulted in an extension of the Anahy execution core, which now comprise the proposed strategy, considering, at the time of scheduling, the heterogeneous characteristics of the architecture where the application is running. The results show that the proposed strategy improved the quality of Anahy scheduling on NUMA architectures, contributing to the increased performance of this environment. Compared to other tools, the proposed strategy proved compatible.

Keywords: Application level scheduling, NUMA architectures, Parallel Task Scheduling, List Scheduling.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
DAG	*Directed Acyclic Graph*
DCG	*Directed Cyclic Graph*
FSS	*FAME Scalability Switch*
GB	*Gigabyte*
GHz	*Gigahertz*
HwLoc	*Hardware Locality*
HwTopoLB	*Hardware Topology Load Balancer*
IQR	*InterQuartile Range*
KB	*Kilobyte*
LTS	*Long Term Support*
MB	*Megabyte*
NF	*NUMA Factor*
NUMA	*Non-Uniform Memory Access*
OpenMP	*Open Multi-Processing*
PV	*Processador Virtual*
RTT	*Round-Trip Time*
SMP	*Symmetric Multi-Processor*
TBB	*Threading Building Blocks*
UMA	*Uniform Memory Access*

SUMÁRIO

1 INTRODUÇÃO

As arquiteturas multiprocessadas são praticamente onipresentes nos sistemas computacionais modernos. Arquiteturas para o processamento paralelo, que até poucos anos atrás eram encontradas somente em computadores de grandes centros de pesquisa ou voltados para o processamento de alto desempenho, passaram a estar presentes em dispositivos de computação pessoal. A principal razão deste fato é a popularização, devido ao seu baixo custo, da tecnologia *multicore*.

Porém, com o constante aumento no número de núcleos de processamento das arquiteturas SMP (abreviação para *Symmetric Multi-Processor*) surge um limitador para o ganho em desempenho (KUPFERSCHMIED; STOESS; BELLOSA, 2009), uma vez que o acesso à memória tem se tornado um gargalo crucial. Arquiteturas SMPs são caracterizadas por fornecer o mesmo tempo de acesso a todas as posições de memória para todos os processadores do sistema. Esse acesso se dá por meio de um barramento, ao qual todos os processadores estão interligados. Na mesma medida em que aumenta o número de processadores, aumenta também a disputa pelo acesso ao barramento pelos processadores para realizar operações de leitura e escrita na memória.

Como alternativa a estas limitações de escalabilidade, têm-se as arquiteturas NUMA (*Non-Uniform Memory Access*) (RIBEIRO, 2011). As arquiteturas NUMA têm como característica principal os diferentes tempos de acesso à memória, para diferentes combinações de núcleos de processamento e endereços de memória, uma vez que o espaço de endereçamento encontra-se divido em diferentes módulos e que cada um destes módulos encontra-se fisicamente próximo a um determinado subconjunto de processadores.

O interesse no uso de arquiteturas NUMA é ainda maior quando considera-se que estas oferecem uma relação mais atraente entre custo e desempenho quando comparadas às arquiteturas SMPs (ANNAVARAM; SHEN, 2005; KUMAR et al., 2004). Porém, para que se consiga obter bons índices de desempenho em seus programas, é desejável que o programador conheça os detalhes da

arquitetura em que a aplicação está sendo executada e inclua em seu código instruções especificas para o mapeamento de suas atividades sobre os recursos de processamento disponíveis. Não considerar as longas latências e os limites de largura de banda provenientes das interconexões entre os processadores e os blocos de memória espalhados pela arquitetura na distribuição do trabalho pode comprometer o desempenho do programa (CALCIU et al., 2013).

Os algoritmos de escalonamento são uma peça chave para se obter bons índices de desempenho em cenários heterogêneos, como os encontrados em máquinas NUMA. A política de escalonamento determina como a memória e os recursos de processamento serão utilizados durante a execução da aplicação. Diferentes estratégias podem ser empregadas, estaticamente ou em tempo de execução, para otimizar algum índice de desempenho, como o número de acessos à memória, o consumo de energia dos processadores ou o tempo total de execução do programa. Em se tratando do problema de escalonar atividades de um programa paralelo em tempo de execução, este escalonamento deve ser realizado em nível aplicativo. (CASAVANT; KUHL, 1988) Neste caso, alguma abstração do programa em execução deve ser utilizada para que o mecanismo de escalonamento possa manipular as atividades concorrentes geradas pelo programa.

Um programa paralelo pode ser representado como um Grafo Dirigido Acíclico (do Inglês, *Directed Acyclic Graph* – DAG), onde os nós representam as tarefas produzidas por esse programa e as arestas representam as dependências de dados entre essas tarefas. Os algoritmos de listas são estratégias bastante eficientes no escalonamento de DAGs comprovadas na literatura (GRAHAM, 1976; ALBERS, 1999). Essas dependências de dados entre as tarefas da aplicação podem ser conhecidas previamente ou durante a execução do programa paralelo. Neste trabalho estamos considerando o segundo caso, onde o escalonamento é realizado em nível aplicativo de forma dinâmica, ou seja, tomando suas decisões na medida em que o programa evolui.

Um algoritmo amplamente utilizado e aceito na literatura com eficiência comprovada é o algoritmo de escalonamento por Roubo de Trabalho *(Work Stealing*) (BLUMOFE; LEISERSON, 1994). Esse algoritmo utiliza listas de tarefas distribuídas para escalonar as tarefas entre os processadores disponíveis. Cada unidade de processamento possui uma lista de tarefas para executar e, quando esta lista ficar vazia, o processador busca nas listas de outros processadores uma tarefa para roubar. Na sua forma básica, os algoritmos de lista organizam as tarefas segundo alguma política de prioridade. A decisão para o mapeamento das tarefas nos processadores depende da classificação da lista. Neste trabalho, busca-se otimizar esta estratégia de roubo, considerando as propriedades das arquitetu-

ras NUMA como critério adicional na etapa de seleção de tarefas para realizar o mapeamento sobre os processadores.

1.1 Motivação e objetivos

As arquiteturas NUMA são uma solução ao problema de disputa pelo barramento das arquiteturas simétricas, aumentando sua escalabilidade. Porém, por se tratar de um ambiente com tempos de acesso à memória bastante heterogêneo, é preciso que o programador conheça essa arquitetura e leve em consideração suas características quando do desenvolvimento de aplicações paralelas.

A motivação deste trabalho, então, parte da necessidade de se ter ferramentas de programação paralela que sejam simples de ser utilizadas, expressivas e, o mais importante, que abstraiam do programador complexidades como o escalonamento dos fluxos de execuções produzidos por um programa paralelo (*threads* criados pelo programa) sobre os recursos disponíveis, considerando as características da arquitetura de modo a obter ganhos em desempenho.

Neste trabalho buscou-se modelar e implementar uma estratégia de escalonamento para programas *multithread* em arquiteturas NUMA. Esta estratégia considera, em nível aplicativo, a localidade física dos dados e as dependências entre as tarefas em tempo de execução para realizar o escalonamento dos *threads*. O objetivo é minimizar o impacto das diferentes latências no acesso aos módulos de memória pelos processadores, proveniente das arquiteturas NUMA.

O trabalho está inserido no projeto "GREENGRID: Computação de Alto Desempenho Sustentável". O projeto envolve a participação de quatro Instituições de Ensino Superior, são elas: 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). A seguir é apresentada a metodologia adotada no desenvolvimento deste trabalho.

1.2 Metodologia de desenvolvimento

O desenvolvimento do trabalho iniciou-se com a revisão dos principais conceitos teóricos relacionados ao escalonamento de tarefas em nível aplicativo disponíveis na literatura e, também, das recentes pesquisas realizadas pela comunidade científica nesse sentido. A partir deste estudo, uma estratégia de escalonamento para arquiteturas NUMA foi especificada e implementada, considerando suas características e também as dependências de dados entre as tarefas. Inicialmente, foi realizada a especificação e parametrização da estratégia, bem como

das heurísticas utilizadas pelo escalonador.

Para extrair as características da arquitetura consideradas pelo escalonador no momento da decisão de roubo de tarefas, foi utilizada a ferramenta *HwLoc* (abreviação para *Hardware Locality*). Trata-se de uma ferramenta capaz de coletar as principais informações da topologia da máquina e também informações sobre memória e interconexões. *HwLoc* foi estendida por Laércio Pilla et al. (2012), para fornecer informações de latências de acessos à memória para cada um dos processadores e largura de banda na transmissão dos dados. Esta ferramenta está apresentada na Seção 3.2 do Capítulo 3, o qual trata do ferramental utilizado neste trabalho.

Uma vez especificada, a estratégia de escalonamento foi implementada na forma de um protótipo para avaliação. Após a implementação, foram realizados alguns testes para validá-la e realizar alguns ajustes necessários. Uma vez validada, uma nova versão foi desenvolvida para ser inserida no núcleo de execução do *Anahy*, um ambiente de programação e execução *multithread* dinâmico projetado para arquiteturas multiprocessadas, a fim de contribuir com o aumento de desempenho da ferramenta. *Anahy* está apresentado na Seção 3.3, onde está descrita sua arquitetura e interface de programação.

Para investigar o desempenho e a eficiência da estratégia desenvolvida, diversos experimentos foram realizados. Estes experimentos são compostos de aplicações capazes de explorar as características deste tipo de escalonamento, considerando diferentes estruturas de programas paralelos. As aplicações são (i) alinhamento genético com o algoritmo *Smith-Waterman*, (ii) algoritmo de ordenação *Bucket Sort*, (iii) fatoração LU de matrizes e o (iv) cálculo do n-ésimo termo da sequência de *Fibonacci*.

Para fins de avaliação do desempenho obtido com a ferramenta, os resultados obtidos com as aplicações foram comparados com os resultados obtidos pelas principais ferramentas de programação *multithread* disponíveis, são elas: *Cilk Plus*, *OpenMP* e *Threading Building Blocks* (TBB).

1.3 Contribuição e resultados

Anahy possui um núcleo de escalonamento concebido para explorar, em nível aplicativo, informações sobre a estrutura de um programa paralelo (estrutura esta que pode ser abstraída em forma de um DAG). Deste DAG são obtidas informações sobre a distância de cada tarefa em relação ao início do programa. Em uma decisão local, o processador privilegia a execução da tarefa mais recentemente criada. Já para uma decisão global, cujo resultado é a migração de uma tarefa, o escalonamento privilegia a seleção de uma tarefa mais antiga. Este trabalho se

propôs a incorporar uma estratégia complementar, contemplando a observação das diferentes latências no acesso aos módulos de memória das arquiteturas NUMA como aspecto a ser considerado no momento do escalonamento.

Esta abordagem difere da inicialmente proposta em Anahy por considerar a heterogeneidade da arquitetura em adição à heurística de detecção do caminho crítico do programa. Buscou-se, então, comprovar a possibilidade de incorporar estratégias de escalonamento complementares em Anahy, que considerem a heterogeneidade da arquitetura, e avaliar o impacto no desempenho.

1.4 Estrutura do texto

Este trabalho está organizado em 6 capítulos. O presente capítulo trouxe uma introdução ao trabalho desenvolvido, apresentando a motivação e os objetivos do trabalho, bem como a metodologia adotada. No Capítulo 2 são introduzidas duas classificações de escalonamento amplamente utilizadas na literatura. Alguns estudos de caso também são discutidos. Ainda no Capitulo 2, são apresentadas as principais ferramentas que implementam estratégias de escalonamento em nível aplicativo (Cilk Plus, Threading Building Blocks e OpenMP) e também um algoritmo de balanceamento de carga que faz uso das informações da topologia da máquina.

O Capítulo 3 é responsável por apresentar todo o ferramental utilizado no desenvolvimento deste trabalho, iniciando pela caracterização das arquiteturas NUMA, seguindo pela apresentação da ferramenta *HwLoc*, utilizada para extrair as informações da arquitetura. Na sequência é apresentado o ambiente de programação e execução *multithread* dinâmico Anahy, que teve seu núcleo de execução estendido para comportar a estratégia de escalonamento proposta neste trabalho.

No Capítulo 4 é apresentada a solução proposta, iniciando com o modelo de aplicação selecionado, seguido pelo modelo de execução adotado, depois o modelo de arquitetura, o modelo de escalonamento e, por fim, trata da implementação da estratégia proposta, apresentando seus passos para realizar o escalonamento.

O Capítulo 5 é responsável por descrever a avaliação experimental da ferramenta, apresentando as plataformas de testes utilizadas, caracterizando as aplicações que foram testadas e, por fim, apresenta e discute os estudos de caso realizados, apresentando os resultados obtidos comparando-os com os resultados obtidos pelas principais ferramentas paralelas disponíveis. O Capítulo 6 conclui o trabalho e discute os novos desafios de pesquisa.

2 TRABALHOS RELACIONADOS

Diversas ferramentas de programação suportam paralelismo de tarefas em arquiteturas *multicore*. Para tal, algumas estratégias de escalonamento são empregadas. Neste capítulo, inicialmente, vamos apresentar duas classificações de escalonamento, uma proposta por (FEITELSON; RUDOLPH, 1995), onde o escalonamento pode ser realizado em dois níveis: no nível do sistema operacional ou no nível da aplicação e a outra introduzida por (CASAVANT; KUHL, 1988), onde é definida uma taxonomia para o escalonamento.

Ainda neste capítulo, serão apresentadas algumas algumas ferramentas que fazem uso de estratégias de escalonamento em nível aplicativo, são elas: Cilk Plus (BLUMOFE et al., 1995), Threading Building Blocks (TBB) (REINDERS, 2007) e OpenMP (CHAPMAN; JOST; PAS, 2008), enfatizando suas interfaces de programação e o modelo de execução e escalonamento. Também é apresentado um algoritmo para o balanceamento de carga, HwTopoLB (abreviação para *Hardware Topology Load Balancer*) (PILLA et al., 2014) que faz uso das informações da topologia da máquina para aumentar o desempenho da aplicação.

2.1 Níveis de escalonamento

O escalonamento pode se dar em dois níveis (FEITELSON; RUDOLPH, 1995): (i) no nível do **sistema operacional** e (ii) no nível da **aplicação**. Em um primeiro nível, a alocação de processadores é realizada pelo sistema operacional. Neste nível, otimizações podem ser realizadas considerando a utilização dos recursos (processadores). Já no segundo nível, o escalonamento das partes do programa que serão executadas em cada um destes processadores é realizado pelo sistema *runtime* da linguagem de programação ou pela própria aplicação. Em nível aplicativo, otimizações podem ser obtidas considerando as propriedades do programa.

Os mecanismos e considerações do escalonamento em nível sistema são, por vezes, semelhantes às aplicadas no nível de aplicação, o que pode gerar

algumas contradições sobre quem está comandando o escalonamento. O espectro de opiniões varia entre aqueles que afirmam que a maior parte do escalonamento deve ser realizado no nível da aplicação (ANDERSON et al., 1991) e aqueles que defendem a ideia de que o sistema operacional deve fazer todo o trabalho (BLACK, 1990). A seguir são feitas algumas considerações sobre cada um destes níveis de escalonamento.

2.1.1 Escalonamento em nível aplicativo

O escalonamento de tarefas tem sido muito pesquisado nos dias atuais. Em muitos ambientes de programação é ocultado do programador os detalhes de como o programa é mapeado entre os processadores disponíveis na arquitetura da máquina. Estes detalhes são, então, tratados pelo sistema *runtime* desses ambientes (FEITELSON; RUDOLPH, 1995).

Um exemplo dessa abordagem é a utilização de *threads* (fluxos de execução de um programa paralelo) (PANCAKE, 1993). A aplicação é estruturada como um conjunto de *threads* que interagem entre si. O ambiente fornece uma série de funções para a criação, sincronização e término de *threads*. O sistema *runtime* é responsável pela implementação dessas funções quando elas são chamadas. Cada ambiente possui suas próprias estratégias de escalonamento de *threads*, na Seção 2.3, são estudadas três estratégias de escalonamento em nível aplicativo, provenientes das ferramentas Cilk Plus, Threading Building Blocks (TBB) e OpenMP.

Um outro exemplo de escalonamento em nível aplicativo é o uso da paralelização dos compiladores. A paralelização dos compiladores usualmente extrai paralelismo dos laços de repetição dos programas sequenciais. Em tempo de execução, as diferentes iterações de um laço de repetição são escalonadas para executarem em processadores distintos (LILJA, 1994). Uma abordagem comum consiste em ajustar o tamanho dos blocos de iterações (*chunks*) de modo a tentar equilibrar a carga de cada processador. Mais uma vez, o programador não precisa se preocupar em como isso é feito.

2.1.2 Escalonamento em nível de SO

Escalonamento proveniente do programador ou do sistema *runtime* tem por objetivo satisfazer as necessidades individuais do programa em questão. Porém, quando diversos programas devem co-existir no mesmo sistema, é necessário equilibrar as necessidades do grupo de processos envolvido com a oferta de recursos do *hardware*. Tais considerações e mecanismos encontram-se na esfera do sistema operacional.

Interrupções, trocas de contexto e *swap* de memória podem gerar muitos

sobre-custos (*overheads*) no momento da execução de uma aplicação paralela. Nesse sentido, algoritmos de escalonamento podem ser empregados em nível de sistema operacional para reduzir esses *overheads* e promover a exploração com maior grau de eficiência de um, ou de um conjunto de, recursos de processamento.

2.2 Taxonomia dos algoritmos de escalonamento

Com a finalidade de auxiliar na organização dos algoritmos de escalonamento e, também, facilitar o desenvolvimento de estratégias, (CASAVANT; KUHL, 1988) propuseram uma taxonomia identificando diversas estratégias de escalonamento. Esta taxonomia tem uma classificação hierárquica das técnicas, reproduzida na Figura 1, e um conjunto de cinco atributos aplicáveis às diferentes técnicas.

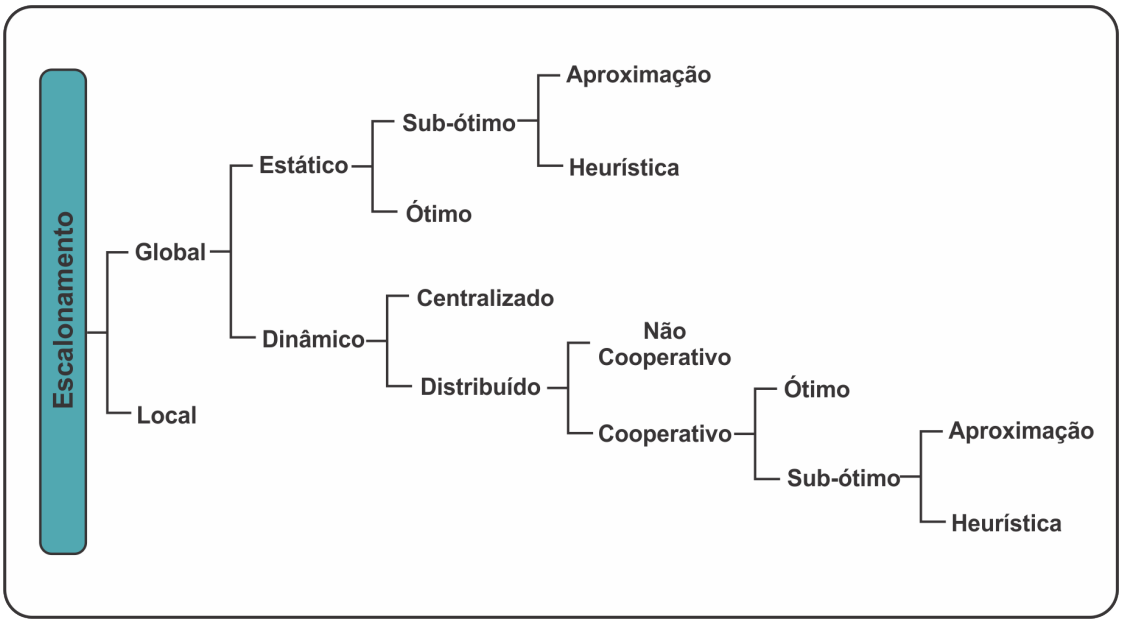


Figura 1: Taxonomia hierárquica de algoritmos de escalonamento.

Uma descrição sucinta das diferentes classes propostas por Casavant e Kuhl (1988) é apresentada na sequência.

- **Local vs. Global**: O escalonamento local determina como os processos residentes em um único processador são alocados e executados. Já o escalonamento global utiliza informações sobre o sistema para alocar processos em múltiplos processadores. Na abordagem apresentada, o escalonamento global corresponde ao escalonamento de nível aplicativo, segundo a visão de (FEITELSON; RUDOLPH, 1995) e o escalonamento local é aquele delegado às operações do escalonamento sistema. As subclasses do escalonamento local não são tratadas por Casavant e Kuhl 1988.

- **Estático vs. Dinâmico**: No escalonamento estático, as informações que descrevem as diferentes atividades concorrentes da aplicação e suas dependências estão disponíveis antes do início de sua execução. Ou seja, no escalonamento estático a atribuição de tarefas aos processadores é determinada antes do início da execução. Já no escalonamento dinâmico é realizada a alocação de tarefas durante a execução da aplicação.

- **Distribuído vs. Centralizado**: Nos escalonadores dinâmicos, as decisões de escalonamento global podem ser tomadas por um escalonador centralizado ou pode ser compartilhada por múltiplos escalonadores distribuídos. A estratégia centralizada pode ser mais simples em termos de implementação, se comparada à distribuída. Entretanto, poderá ser um gargalo em termos de desempenho, pois consiste em um único módulo responsável por reagir a cada passo da evolução do programa. Por outro lado, é de se esperar que estratégias distribuídas incluam certo *overhead*, resultado da interação dos núcleos de escalonamento operando sobre cada processador, nas tomadas de decisão.

- **Cooperativo vs. Não cooperativo**: No modo cooperativo, os diferentes núcleos de escalonamento tomam suas decisões sobre alocação das tarefas considerando as informações sobre a carga dos demais processadores. Nas estratégias não cooperativas, as decisões são tomadas sem considerar cargas dos demais processadores.

- **Ótimo vs. Sub-ótimo**: Se toda a informação do estado dos recursos e da aplicação é conhecida, o escalonamento poderá ser ótimo. Nos casos em que é computacionalmente inviável obter todo este conjunto de informações, o escalonamento alcançado é considerado sub-ótimo.

- **Aproximação vs. Heurística**: O algoritmo de aproximação procura implementar uma solução que seja suficientemente boa em relação a que foi definida como ótima. Já o algoritmo de heurística, leva em consideração alguns parâmetros que afetam o sistema de uma maneira indireta, como por exemplo, a comunicação entre processos e buscam promover o desempenho considerando este aspecto na tomada de decisões.

Os algoritmos de escalonamento descritos pelas classes identificadas na taxonomia hierárquica de Casavant e Kuhl (1988) podem ainda ter atributos que os descrevam em relação aos seguintes comportamentos:

- Potencial de adaptação da estratégia de escalonamento durante a execução do programa.

- Processo de leilão para identificação do processador mais apto para executar uma determinada tarefa.
- Uso de técnica probabilística, que considera o número de variações de escalonamentos que podem ser obtidos a partir de um conjunto de tarefas sobre uma determinada arquitetura.
- Possibilidade de reavaliar, ou não, a realocação de tarefas já mapeadas, havendo a necessidade de mecanismos de preempção, caso seja permitida a migração de tarefas já em execução.
- Balanceamento ou compartilhamento de carga entre os processadores.

Desses atributos, cabe destacar a diferença entre estratégias que produzem balanceamento de carga entre os processadores e aquelas que promovem o compartilhamento de carga. Estratégias de balanceamento procuram equalizar, considerando um desvio padrão previamente estabelecido, a carga de todos os processadores envolvidos na computação. As estratégias de compartilhamento, por sua vez, atentam para que todos os processadores possuam trabalho a realizar, não sendo considerada relevante a disparidade de carga que possa ocorrer.

A premissa é reduzir o número de interações entre os processadores para, ao mesmo tempo, reduzir a influência do tempo de operação do escalonador no tempo total de execução do programa. Outra característica relevante no escalonamento é a diferença entre as estratégias que prevêem disparo, como uma atividade em execução, automático de todo trabalho identificado como pronto das estratégias que limitam o número de atividades em execução simultânea.

2.2.1 Escalonamento *Blind* vs. *Non-blind*

Uma técnica *blind* (às cegas) de escalonamento é uma técnica que não considera características futuras da aplicação no momento de uma tomada de decisão. Por exemplo, o não conhecimento do tempo de processamento de uma tarefa não permite utilizar esta informação para definir seu mapeamento. O escalonador pode, no entanto, utilizar outros atributos desta tarefa, como por exemplo sua hora de chegada e o tempo de processamento gasto até o momento em uma heurística que lhe indique a ação a ser tomada (FENG; MISRA; RUBENSTEIN, 2007). Nestes casos, quanto mais a aplicação for regular, ou seja, quanto mais reproduzível as etapas de execução do programa se mostrarem, mais eficiente será a técnica de escalonamento obtida.

Já políticas de escalonamento *non-blind* agem de maneira oposta, ou seja, consideram as características futuras da aplicação, uma vez que têm acesso a elas. Essa técnica de escalonamento pode evitar uma série de problemas,

tais como a explosão do uso da memória por gerar muito paralelismo ou pouca exploração do potencial de execução concorrente da máquina.

2.3 Ferramentas com escalonamento em nível aplicativo

No próprio trabalho de Casavant e Kuhl (1988), diversos exemplos de estratégias de escalonamento e de ferramentas que os implementam são apresentados. Nesta seção apresentamos três ferramentas de programação *multiththread* que incluem núcleos de execução dotados de estratégias de escalonamento.

2.3.1 Intel Cilk Plus

Cilk Plus (BLUMOFE et al., 1995) é uma extensão da linguagem C++, atualmente mantida pela Intel, que dá suporte a programação *multithread* em arquiteturas *multicore*. Cilk Plus oferece recursos para criar *threads* dinamicamente e para sincronizar os *threads* criados de maneira que o *thread* que os criou aguarde o término de sua execução. A seção a seguir apresenta a interface de programação de Cilk Plus.

2.3.1.1 Interface de programação

Cilk Plus adiciona as seguintes palavras-chave à sintaxe de *C++*: `_Cilk_spawn`, `_Cilk_sync` e `_Cilk_for`. Estas palavras-chave permitem a descrição da concorrência presente em computações paralelas. 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, ou seja, a função a ser chamada será submetida como uma nova tarefa. `_Cilk_sync` indica que uma tarefa deve esperar por todas as tarefas filhas, em outras palavras, realiza a sincronização de todos os *threads*, sem a necessidade de recebimento de parâmetros.

A descrição da concorrência em Cilk ocorre de forma aninhada: um *thread* em execução pode criar diversos *threads* pelo uso da primitiva `spawn`, sedo que uma invocação à primitiva `sync` força seu bloqueio enquanto não ocorrer o término de todos os *threads* por ela criados. Já a operação `Cilk_for` consiste em uma abstração de mais alto nível para o conjunto de primitivas `spawn`/`sync`, uma vez que cada iteração do `for` resulta em um `spawn` e o término do laço em uma barreira representada por um `sync`.

A linguagem *Cilk Plus* foi concebida de tal maneira que, caso haja 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 mesmo resultado. O Código 1, a seguir, exemplifica a API (*Application program-*

ming interface) de *Cilk Plus*, aplicada à descrição da concorrência em um cálculo recursivo do enésimo número da sequência de *Fibonacci*.

Código 1: Fibonacci paralelo com Cilk Plus.

```
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 }
11 int main(){
12    long res, n = atoi(argv[1]);
13    res = fib(n);
14    printf("Resultado: %d\n", res);
15    return 0;
16 }
```

2.3.1.2 Modelo de execução e escalonamento

A implementação de *Cilk Plus* é baseada em um algoritmo de escalonamento dinâmico por meio de roubo de tarefas, descrito em (BLUMOFE; LEISERSON, 1994). *Cilk* define *threads* como funções individuais que podem submeter novos *threads* dinamicamente. A sincronização é feita permitindo que os *threads* esperem pelos *threads* filhos, ou seja, os *threads* que foram criadas pelo *threads* original.

Em *Cilk Plus* o ambiente cria um conjunto de *worker threads* (ou *workers*) que são processadores virtuais do ambiente, responsáveis por executar os *threads* criados. Estes processadores virtuais, por sua vez, são escalonados pelo sistema operacional sobre os processadores disponíveis. O escalonamento dos *threads* gerados pelo programa em execução é realizado em nível aplicativo. De maneira abstrata, 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 quando uma nova função é criada com `_Cilk_spawn`.

Os *workers* implementam a estratégia *Work-First*, 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 maior potencial de geração de paralelismo.

2.3.2 Intel Threading Building Blocks

Threading Building Blocks (TBB) (REINDERS, 2007) é uma biblioteca desenvolvida pela a Intel, escrita em *C++*, que fornece suporte a programação concorrente em arquiteturas *multicore*. 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.

2.3.2.1 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. Estes *templates* são bastante convenientes para diversos problemas e realizam a manipulação dos *threads* de maneira automática. No entanto, TBB fornece recursos para que o programador possa descrever os *threads* e as dependências de dados à sua maneira.

TBB 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* (o mesmo que processadores virtuais, como em *Cilk*). No corpo do método `execute()` podem ser criadas novas tarefas, as quais podem ser sincronizadas de diferentes maneiras.

A melhor forma é explicitando uma tarefa de continuação (modelo de programação *Continuation-Passing*), onde uma tarefa τ em execução cria tarefas filhas (*child*) e cria também uma terceira tarefa denominada continuação (*continuation*) e determina que esta será executada quando as filhas de τ terminarem sua execução e, após isto, encerra sua própria execução. Esta tarefa de continuação herda o mesmo pai da tarefa τ. Uma vez que as tarefas filhas e a continuação são criados, 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. O Código 2 apresenta um exemplo de código em TBB para efetuar o cálculo do enésimo termo da sequência de *Fibonacci*.

Código 2: Fibonacci paralelo com Intel TBB.

```
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    } else {
12      long x, y;
13      set_ref_count(3);
14      spawn( *new(allocate_child()) FibTask(n-1, &x) );
15      spawn_and_wait_for_all(*new(allocate_child()) FibTask(n-2, &y));
16      *sum = x+y;
17    }
18    return NULL;
19  }
20 };
21 int main (int argc, char const *argv[]) {
22   long sum;
23   FibTask& a = *new(tbb::task::allocate_root()) FibTask(n, &sum);
24   tbb::task::spawn_root_and_wait(a);
25   printf("Resultado: %d\n", sum);
26 }
```

2.3.2.2 *Modelo de execução e 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 semelhante ao de *Cilk*, com a diferença que a API de TBB permite a descrição da concorrência de suas aplicações por meio de DAGs genéricos e não puramente aninhados como em Cilk. Os `workers` de TBB implementam o modo *Help-First*, priorizando localmente as tarefas mais profundas no DAG. Caso não existam tarefas executáveis em seu *deque*, um *worker* executa um roubo de trabalho sobre a *task* menos profunda disponível no *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`. A decisão de delegar uma tarefa a lista global não é do escalonador, mas sim do programador, utilizando serviços específicos da interface de TBB.

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:

- A tarefa retornada pelo método `t.execute()` é não nula;
- O sucessor de τ, se τ foi seu último antecessor a terminar execução;
- Uma tarefa retirada do final do *deque* do próprio *worker*;
- Uma tarefa pronta com afinidade ao *worker*;
- Uma tarefa retirada do início da fila global de tarefas;
- Uma tarefa retirada do início do *deque* de outro *worker*, randomicamente escolhido.

2.3.3 OpenMP

OpenMP (*Open Multi-Processing*) (CHAPMAN; JOST; PAS, 2008) é uma ferramenta para programação paralela em *C*/*C++* e *Fortran* para arquiteturas multiprocessadas, é baseada na adição de diretivas de compilação. Essas diretivas de compilação são utilizadas para indicar a paralelização de laços de repetição ou trechos de código.

2.3.3.1 Interface de programação

A API de *OpenMP* define um conjunto de *pragmas*: diretivas de compilação, neste caso, usadas para anotações de paralelismo no código. Todos os *pragmas* de *OpenMP* possuem o formato `#pragma omp`, somado a outras diversas construções, como `parallel for`, destinado à paralelização de laços *for*. Em *OpenMP* cada trecho de código paralelo é denominado *task*, ou tarefa. Os componentes que executam essas tarefas são denominados *threads* (ou time de *threads*).

O Código 3 exemplifica o uso de alguns *pragmas* da ferramenta, aplicados em um algoritmo recursivo para calcular o enésimo termo da sequência de *Fibonacci*. A linha `#pragma omp parallel` 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 time. `#pragma omp task` provoca a criação de uma tarefa concorrente para executar o código especificado entre as chaves.

Código 3: Fibonacci paralelo com OpenMP.

```
1 long fib_parallel(long n) {
2   long x, y;
3   if (n < 2) return n;
```

```
4     #pragma omp task shared(x, n) {
5       x = fib_parallel(n-1);
6     }
7     #pragma omp task shared(y, n) {
8       y = fib_parallel(n-2);
9     }
10    #pragma omp taskwait
11    return x+y;
12  }
13  int main(int argc, char const *argv[]) {
14    long n = atol( argv[1] );
15    long res;
16    #pragma omp parallel sections shared(n, res) {
17      #pragma omp section {
18        res = fib_parallel(n);
19      }
20    }
21    printf("Resultado: %d\n", res);
22    exit(0);
23  }
```

A linha `#pragma omp taskwait`, na função `fib_parallel`, faz com que a tarefa atual fique bloqueada até que todas as tarefas criadas a partir dela tenham terminado sua execução, sendo que este 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 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. A API de *OpenMP* é bastante extensa e muitas outras construções são fornecidas para descrever paralelismo.

2.3.3.2 Modelo de execução e escalonamento

Semelhante às demais ferramentas apresentadas anteriormente, OpenMP implementa uma arquitetura virtual na qual as atividades concorrentes são escalonadas, em nível aplicativo, sobre *threads* de um "time de execução". Os *threads* deste time são escalonados, por sua vez, pelo sistema operacional sobre os processadores disponíveis. Na terminologia OpenMP, o termo *thread* designa

um *thread* do time de execução e *task* ou tarefa, uma unidade de descrição de concorrência do programa do usuário.

Um programa *OpenMP* começa sua execução em um *thread* de execução, executando uma tarefa implícita do programa (a função `main`). Quando o *thread* inicial encontra uma região paralela (definida com `#pragma omp parallel [...]`), um time de *threads* é instanciado, havendo tantos *threads* no time, em adição ao *thread* "mestre", quantos especificados previamente pelo usuário.

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. *Tasks* também podem ser criadas de forma explícita em OpenMP.

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 no momento da criação de uma nova tarefa, no término da execução de uma tarefa e na 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.

Para construir aplicações em termos de paralelismo de dados, usando *features* 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`, ou `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.

2.4 HwTopoLB

Nesta seção é apresentado um algoritmo de balanceamento de carga para máquinas *multicore*. O algoritmo, chamado *HwTopoLB* (PILLA et al., 2014) visa melhorar o desempenho da aplicação reduzindo a ociosidade dos processadores e os atrasos de comunicação. *HwTopoLB* foi concebido levando em conta as propriedades dos sistemas paralelos atuais compostos por nós de computação *multicore*, ou seja, sua interligação de rede e sua topologia complexa e hierárquica. A topologia é composta por vários níveis de cache e um subsistema de memória com design NUMA. Esses sistemas fornecem alto poder de proces-

samento em detrimento aos custos de comunicação assimétricos, o que pode dificultar o desempenho de aplicações paralelas, dependendo se seus padrões de comunicação forem ignorados.

O objetivo do algoritmo é encontrar o mapeamento ideal de um conjunto de tarefas sobre um conjunto de processadores, tal que o tempo total de execução da aplicação – *makespan* (tempo total de processamento de todas as tarefas em todos os processadores) seja minimizado. Em outras palavras, objetiva reduzir os custos com comunicações desbalanceadas e com comunicações remotas para melhorar o desempenho da aplicação. O algoritmo proposto funciona da seguinte maneira: em cada uma de suas iterações, o mapeamento atual das tarefas é modificado em um componente (processador) de acordo com três passos, a saber:

- **1**$^\circ$ **Passo**: Seleção do processador. O processador com a maior carga de processamento é selecionado com probabilidade α (na prática, o valor de α é próximo de 1). Caso contrário (com probabilidade $1 - \alpha$), é selecionado um processador aleatoriamente.

- **2**$^\circ$ **Passo**: Seleção da tarefa. A tarefa com maior custo (com maior tempo de execução) é escolhida com probabilidade β (cujo valor também é próximo a 1). Caso contrário (com probabilidade $1 - \beta$), é escolhida uma tarefa aleatoriamente entre todas as tarefas mapeadas para aquele processador.

- **3**$^\circ$ **Passo**: Mapeamento. A tarefa selecionada é movida para o processador que minimiza o custo total (tempo de execução) da aplicação com alta probabilidade, provida pela distribuição de *Gibbs*[1] (*Gibbs distribution*) com temperatura $T = 1$. A razão por trás da distribuição de *Gibbs* é que, quando a temperatura T tende a zero, a probabilidade de escolher um processador com o custo mínimo tende a 1.

O algoritmo proposto requer informações de duas naturezas distintas: (i) *modelo da topologia da máquina* e (ii) *dados da aplicação*. O modelo de topologia da máquina engloba as diferentes latências e larguras de banda provenientes da arquitetura para acessar e trafegar dados. Os dados da aplicação são compostos pelo custo de uma tarefa, pelo custo de comunicação de dados entre as tarefas e pela quantidade de *bytes* trafegados.

2.4.1 Aquisição da topologia da máquina

O modelo da topologia da máquina compreende toda a informação que pode ser extraída sobre o *hardware* que está executando a aplicação, como por exem-

[1] Maiores informações sobre a distribuição de *Gibbs* podem ser encontradas em (BREMAUD, 1999), capítulo 7.8.

plo, qual processador habita qual *socket* ou nó. Para adquirir tais informações, foi estendida a ferramenta *HwLoc* (*Portable Hardware Locality*) (BROQUEDIS et al., 2010). A ferramenta *HwLoc* é apresentada na seção 3.2 deste trabalho.

A topologia da máquina é representada como uma árvore, onde a raiz é a máquina e as folhas são as unidades de processamento (processadores). *HwLoc* foi estendida para prover informações de latências e larguras de banda e isso é realizado em dois níveis.

No primeiro nível, as estatísticas de memória são adquiridas através do *LMbench*[2] (STAELIN; PACKARD, 1996). Os valores de latência e largura de banda são mensurados para todos os níveis de memória *cache* e nós NUMA com os *benchmarks lat_men_rd* e *bw_mem*, respectivamente. *HwLoc* ajuda nesse processo provendo informações valiosas sobre a topologia da máquina, como o tamanho de cada nível de *cache* e também provendo a habilidade de vincular processos a memória. Este último é importante para avaliar os custos de acesso à memória entre pares de nós NUMA, por exemplo.

No segundo nível, estatísticas de rede são conseguidas com um *benchmark ping-pong*, escrito em *coNCePTuaL*[3] (PAKIN, 2007). Como o *benchmark* retorna o RTT (*Round-Trip Time*) entre um par qualquer de nós para diferentes tipos de mensagens, é aplicada uma regressão linear para decompor esses tempos em latências e larguras de banda. A partir disso, é possível representar a distância entre processadores em um sistema *multicore*.

2.4.2 Aquisição dos dados da aplicação

No contexto do *HwTopoLB*, dados da aplicação refere-se a todas as informações da aplicação paralela que podem ser sondados em tempo de execução, tais como os tempos de execução das tarefas, padrões de comunicação e a estratégia adotada pelo balanceador de carga em um determinado momento.

O balanceador de carga foi implementado em *Charm++* (KALE et al., 2008; Kalé; KRISHNAN, 1993), uma vez que esta ferramenta é capaz de fornecer uma série de informações sobre a aplicação em tempo de execução. *Charm++* é uma linguagem de programação paralela orientada a objetos com o objetivo de melhorar a produtividade do programador proporcionando desempenho.

Uma forte característica de *Charm++* é o seu *framework* para o balanceamento de carga, que fornece uma API simples para realizar o balanceamento de custos computacionais e custos com comunicações (ZHENG et al., 2011). Os dados são obtidos dinamicamente durante a execução da aplicação, por meio da

[2] *LMbench* é uma ferramenta para análise de desempenho. Maiores informações poderão ser obtidas em `http://lmbench.sourceforge.net/`

[3] *coNCePTuaL* é uma linguagem de teste de coretude e desempenho de redes. Maiores informações podem ser obtidas no link `http://www.ccs3.lanl.gov/~pakin/software/conceptual/`.

API de balanceamento de *Charm++*. Ao implementar *HwTopoLB* fazendo uso do *framework* de balanceamento de carga de *Charm++*, foi possível obter os parâmetros necessários para o balanceador: custo computacional das tarefas, custos de comunicação e quantidade de *bytes* trafegados.

2.5 Considerações sobre o capítulo

Este capítulo apresentou uma série de nomenclaturas e classificações de escalonamento. (FEITELSON; RUDOLPH, 1995) classificou o escalonamento em dois níveis: (i) nível de aplicação e (ii) nível de sistema. (CASAVANT; KUHL, 1988) introduziu uma taxonomia para o escalonamento amplamente adotada pela literatura atual. Existem diversas ferramentas de programação que oferecem suporte ao escalonamento em nível aplicativo, tendo sido apresentadas três delas: (i) *Cilk Plus* (BLUMOFE et al., 1995), (ii) *Threading Building Blocks* (TBB) (REINDERS, 2007) e (iii) *OpenMP* (CHAPMAN; JOST; PAS, 2008). Para cada ferramenta foi descrita a interface de programação e o modelo de execução e escalonamento.

Também foi apresentado o algoritmo *HwTopoLB* (PILLA et al., 2014). Esta ferramenta diferencia-se das demais por considerar a topologia da arquitetura para realizar o escalonamento. As ferramentas de programação *multithread* apresentadas, Cilk, TBB e OpenMP, empregam estratégia de escalonamento em dois níveis em um modelo $N \times M$. Nesta abordagem, N tarefas concorrentes geradas pelo programa são escalonadas, em nível aplicativo, sobre M processadores virtuais. Os processadores virtuais, por sua vez são mapeados, em nível sistema, sobre os recursos de processamento disponíveis.

Em Cilk e TBB a estratégia de escalonamento inclui uma heurística que busca privilegiar a execução de tarefas sobre o caminho crítico (GRAHAM, 1976; SOUZA CAMARGO, 2013), buscando também explorar a localidade de referência de dados entre *threads* com dependência mútua. OpenMP, por sua vez, mantém uma lista global entre os *threads* do time de execução mantendo o que contém o conjunto de trabalho a ser executado.

O algoritmo HwTopoLB apresentado também considera uma lista de trabalho ordenada pelo custo computacional de cada tarefa e a carga de trabalho alocada a cada processador. A decisão de alocação de uma tarefa sobre um processador considera tanto o custo individual de cada tarefa, como também o custo da migração desta tarefa para um determinado processador em uma arquitetura NUMA.

O modelo de execução $N \times M$ não é recente, tendo sido empregado de forma recorrente desde as primeiras implementações do padrão POSIX *threads* (Pthre-

ads). No entanto, a perspectiva aberta com modernas ferramentas de programação *multithread*, como OpenMP e Cilk, permitiu estender as possibilidades deste modelo com estratégias de escalonamento mais elaboradas. Embora técnicas de escalonamento que considerem arquiteturas NUMA possam ser desenvolvidas a partir das técnicas já implementadas nestas ferramentas, como em (TERBOVEN et al., 2012), a construção de ferramentas *multithread* com estratégias de escalonamento considerando arquiteturas NUMA não se popularizaram.

A ideia, neste trabalho, é estender uma estratégia de escalonamento de um ambiente de execução *multithread* baseado em *workstealing*, explorando localidade de dados e execução do caminho crítico, para contemplar a heterogeneidade, em termos de latências e largura de banda, de uma arquitetura NUMA.

Ferramentas que empregam escalonamento em nível aplicativo operam no mesmo nível do ambiente estendido e possuem uma distribuição funcional para realização de testes. Estas ferramentas, então, são consideradas como trabalhos relacionados ao proposto neste trabalho. A literatura apresenta outros trabalhos relacionados, como é o caso de (DING et al., 2014), onde foi proposto um escalonador para arquiteturas *multicore* assimétricas. O capítulo a seguir apresenta todo o ferramental utilizado no desenvolvimento da estratégia proposta neste trabalho.

3 FERRAMENTAL UTILIZADO

Este capítulo tem por finalidade apresentar o ferramental envolvido no desenvolvimento deste trabalho. Inicialmente, a Seção 3.1 apresenta todas as características e conceitos relacionados às arquiteturas NUMA. Na sequência (Seção 3.2), é apresentada a ferramenta *HwLoc*, uma ferramenta estendida por Laércio Pilla et al. (2012), capaz de coletar as principais informações da topologia da máquina. Por fim, na Seção 3.3 é apresentado o ambiente de programação *multithread* dinâmico Anahy, o qual teve seu núcleo de execução estendido para contemplar a estratégia proposta neste trabalho.

3.1 Arquitetura NUMA

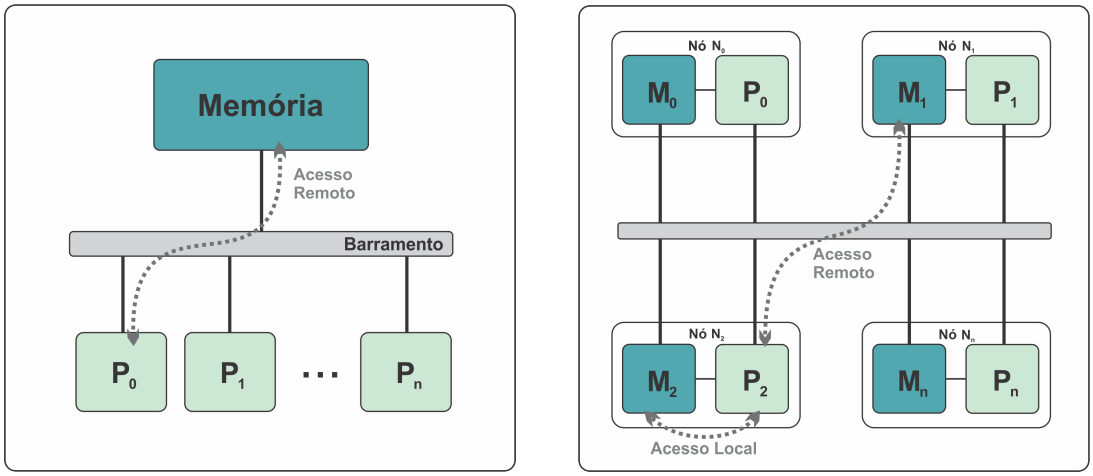
Dependendo da localização física em relação aos processadores e da quantidade de blocos de memória presentes na arquitetura de um computador, o tempo de acesso a uma posição específica desta memória pode ser uniforme ou não, dependendo do processador de partida. Têm-se então, respectivamente, as arquiteturas denominadas de UMA (*Uniform Memory Access*) e NUMA (*Non-Uniform Memory Access*) (CARíSSIMI et al., 2007).

Um computador com arquitetura UMA é caracterizado por fornecer o mesmo tempo de acesso a todas as posições de memória para todos os processadores do sistema. Essa característica é típica dos computadores Multiprocessadores Simétricos (*Symmetric Multi-Processor* – SMP) onde existe um único banco de memória que é acessado por todos os processadores do sistema de maneira uniforme.

Já em um computador com arquitetura NUMA, a memória é fisicamente composta por vários bancos de memória, podendo estar, cada um deles, vinculados a um conjunto de processadores e há um espaço de endereçamento compartilhado. Nesse caso, quando o processador acessa a memória que está vinculada a si, diz-se que houve um **acesso local**. Se o acesso for à memória de outro processador, diz-se que ocorreu um **acesso remoto**. Os acessos remotos são

mais lentos que os acessos locais, uma vez que é necessário passar pela rede de interconexão para que se consiga chegar ao dado localizado na memória remota.

Para um melhor entendimento, a Figura 2 ilustra, de maneira genérica, (a) um computador com arquitetura UMA e (b) um computador com arquitetura NUMA. Estão representados também os acessos à memória.



(a) Arquitetura UMA (SMP) (b) Arquitetura NUMA

Figura 2: Configurações das arquiteturas UMA e NUMA.

Pode-se observar que, na arquitetura UMA representada na Figura 2(a), as operações relacionadas à memória (leituras e escritas), independente da unidade de processamento ($P_0, P_1, ..., P_n$) que estão partindo, possuem o mesmo tempo de acesso, pois todos acessam a memória utilizando o mesmo caminho.

Já na Figura 2(b), existem dois possíveis e distintos caminhos para acessar a memória, dependendo de qual unidade de processamento irá partir o acesso e da localização física da memória a ser acessada: (i) Acesso Local (P_2 acessando a memória M_2) e (ii) Acesso Remoto (P_2 acessando a memória M_1). Os tempos de acessos à memória local de um processador, tanto para leitura quanto para escrita de dados e instruções, são menores do que os tempos de acessos às memórias remotas. Neste tipo de arquitetura, os acessos a memória ocorrem de maneira não uniforme, essas assimetrias no acesso caracterizam este tipo de arquitetura.

Nas arquiteturas com acessos não uniformes à memória, quando um processador for buscar um dado ou uma instrução, a memória ao qual ele irá acessar pode estar anexada a ele (nó local) ou pode estar distante (nó vizinho ou nó opositor) (BROQUEDIS et al., 2009). A Figura 3 ilustra essa relação de acessos, dependo de onde está alocada fisicamente a memória a ser acessada.

Por exemplo, quando uma computação que está executando na P_2 do nó N_2 realiza um acesso a memória M_2, esse acesso é mais rápido do que um acesso a

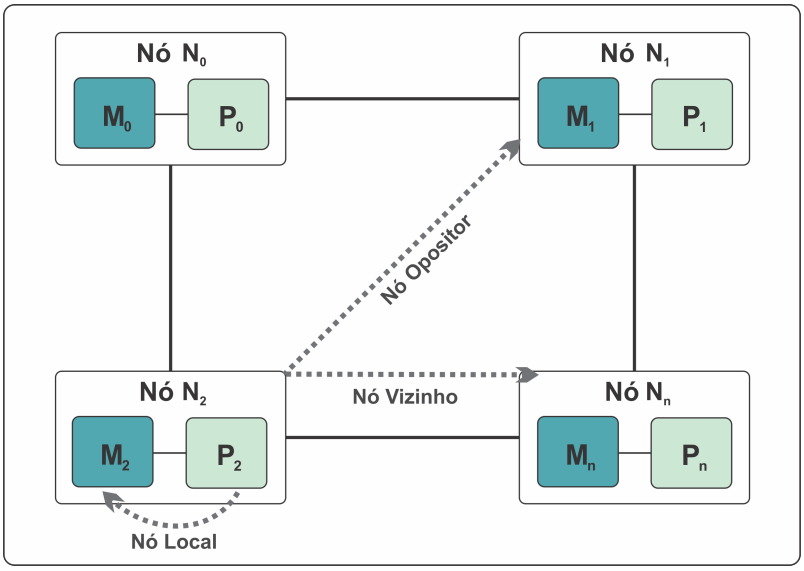


Figura 3: Tipos de acessos em arquiteturas NUMA.

memória M_3 no nó vizinho N_3, ou quando o acesso for a memória M_1, localizada no nó opositor N_1.

Uma forma bastante comum de medir a não-uniformidade de acessos à memória nas arquiteturas assimétricas, é considerar o fator NUMA – NF (*NUMA Factor*). Esse índice consiste na razão entre o acesso a uma posição de memória remota (T_{remoto}) e um acesso a uma posição de memória local(T_{local}) e está representado na Equação 1. Geralmente o fator NUMA varia de 1 até 3, dependendo da arquitetura e, portanto, possui um forte impacto no desempenho da aplicação em execução (BROQUEDIS et al., 2009).

$$NF = \frac{T_{remoto}}{T_{local}} \tag{1}$$

Os tempos de acesso à memória para realizar a leitura de dados são menores que os tempos de acesso para escrita. Quando a memória a ser acessada está fisicamente localizada em um nó vizinho ou em um nó opositor, esses tempos de acesso são ainda maiores. A Tabela 1 mostra uma relação de tempos de acessos à memória, tanto para a leitura, quanto para a escrita de dados, resultado de uma pesquisa de Broquedis et al 2009.

Tabela 1: Tempos de latência de acessos em computadores NUMA.

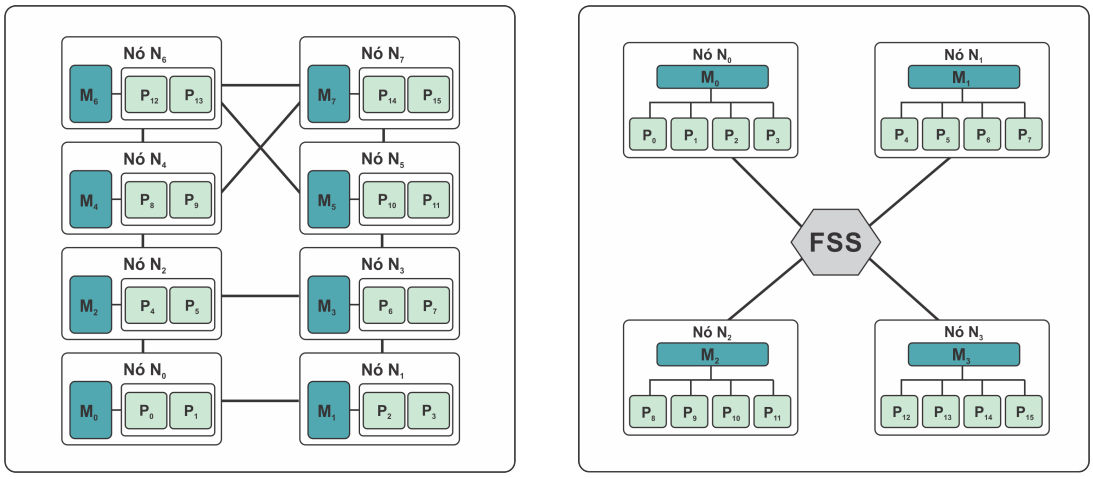
Tipo de Acesso	Acesso Local	Acesso Nó Vizinho	Acesso Nó Opositor
Leitura	83 ns	98 ns (1,18 x)	117 ns (1,41 x)
Escrita	142 ns	177 ns (1,25 x)	208 ns (1,46 x)

Os dados apresentados na Tabela 1 são oriundos de um experimento realizado em um computador NUMA com 4 nós de processamento (Processador

quad-core, 1.9 GHz OPTERON 8347HE), semelhante à arquitetura apresentada na Figura 3. Cada nó possui 2MB de memória *cache* L3 compartilhada e 8GB de memória local anexada.

O experimento comprova que os acessos à memória remota são mais lentos que um acesso local e esse tempo vai aumentando quando a distância da memória em relação ao processador que requisitou o acesso também aumenta. A latência e o fator NUMA são maiores nas escritas de dados pelo fato de haver mais tráfego de *hardware* que está sendo envolvido neste tipo de operação (BROQUEDIS et al., 2009).

Basicamente, por meio da maneira pela qual os processadores e os blocos de memória estão organizados em uma arquitetura NUMA, juntamente com o fator NUMA, é possível ter uma boa compreensão geral da arquitetura. A Figura 4 mostra duas configurações de arquitetura NUMA, cada uma com diferentes valores de latência e fator NUMA. Na Figura 4(a) está ilustrada a arquitetura *Opteron*, composta por 8 processadores dual core AMD *Opteron* de 2.2 GHz. Esta máquina possui 8 nós NUMA e um total de 32 GB de memória principal, dividida em 8 blocos de 4 GB. As disposições dos processadores e blocos de memória resultam em diferentes latências nos acessos remotos, com fator NUMA variando de 1.2 à 1.5. Esta arquitetura é caracterizada pelo baixo fator NUMA.



(a) Arquitetura *Opteron* (b) Arquitetura *Itanium*

Figura 4: Diferentes configurações de arquiteturas NUMA.

Já na Figura 4(b) está representada a arquitetura *Itanium*, composta por 16 processadores *Itanium* de 1.6 GHz. Esta máquina possui 4 nós, cada um com 4 processadores e um total de 64 GB de memória principal, distribuída em 4 blocos de 16 GB. Estes 4 nós estão conectados por um *switch* chamado FSS (*FAME Scalability Switch*). Nesta arquitetura o fator NUMA varia de 2 à 2.5, o que a caracteriza como uma arquitetura com alto fator NUMA.

É importante conhecer as características da arquitetura NUMA, principal-

mente as latências de acesso às memórias remotas. Estas informações servem de subsídio ao desenvolvedor para que este consiga definir estratégias de escalonamento de tarefas mais eficientes para aplicações que executam neste tipo de arquitetura. Para tal, são necessárias ferramentas que consigam extrair esse tipo de informações da topologia da máquina, como é o caso da ferramenta HwLoc.

3.2 HwLoc

HwLoc[1] (BROQUEDIS et al., 2010), abreviação de *Hardware Locality*, trata-se de um conjunto de ferramentas de linha de comando com uma API escrita na linguagem de programação *C*. Tem por finalidade obter o mapa hierárquico dos principais elementos presentes na arquitetura de uma máquina, como nós NUMA de memória, níveis de memória *cache*, *soquets*, núcleos de processamento e até mesmo dispositivos de I/O.

É uma ferramenta portável, suporta uma variedade de sistemas operacionais e plataformas. Além disso, pode agrupar as topologias de várias máquinas em uma única, com a finalidade de permitir que os aplicativos consigam ter uma visão geral da topologia de um conjunto de máquinas, como um *cluster* por exemplo. A topologia da máquina compreende todas as informações que podem ser coletadas sobre o *hardware* em que a aplicação está sendo executada, como a localização física de um núcleo de processamento e suas interligações dentro de um determinado processador ou nó de processamento (GOGLIN; SQUYRES; THIBAULT, 2011).

As arquiteturas estão se tornando cada vez mais complexas, com vários níveis de memória *cache*, podendo conter blocos de *cache* específicos (um para cada processador), blocos de *cache* globais (um para todos os processadores) ou ainda alguns blocos de *cache* parcialmente compartilhados (um bloco compartilhado entre alguns processadores). Para exemplificar, a Figura 5 ilustra a topologia de uma máquina baseada no processador *quad-core* XEON E5345, demonstrando como a *cache* L2 está compartilhada entre os processadores.

Estas informações da topologia da máquina precisam, de alguma forma, ser conhecidas durante o desenvolvimento da estratégia de escalonamento para que possam ser levadas em conta pelo escalonador durante o mapeamento das tarefas de um programa paralelo entre os processadores. Por exemplo, se existem duas tarefas que compartilham dados ou se comunicam, observando a arquitetura da Figura 5, uma boa estratégia seria escaloná-las nos processadores P_0 e P_4, uma vez que ambos compartilham o mesmo bloco de *cache*.

HwLoc obtém as informações da topologia e as disponibiliza ao programador

[1] Link do projeto *Hardware Locality*: http://www.open-mpi.org/projects/hwloc/

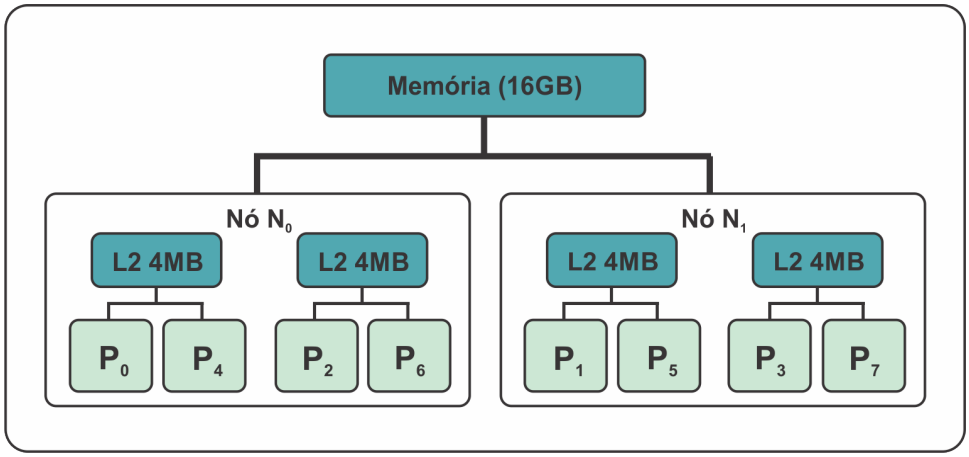


Figura 5: Arquitetura XEON E5345 com cache parcialmente compartilhada.

de diversas maneiras. A Figura 6 mostra a visão hierárquica correspondente a arquitetura XEON apresentada na Figura 5, incluindo o conhecimento dos blocos de *cache* compartilhados. Esta imagem é a representação gráfica da topologia da máquina resultante do comando *lstopo*, proveniente da ferramenta *HwLoc*, onde a topologia da máquina é representada por meio de uma árvore, sendo que a raiz é a própria máquina e as folhas são unidades de processamento.

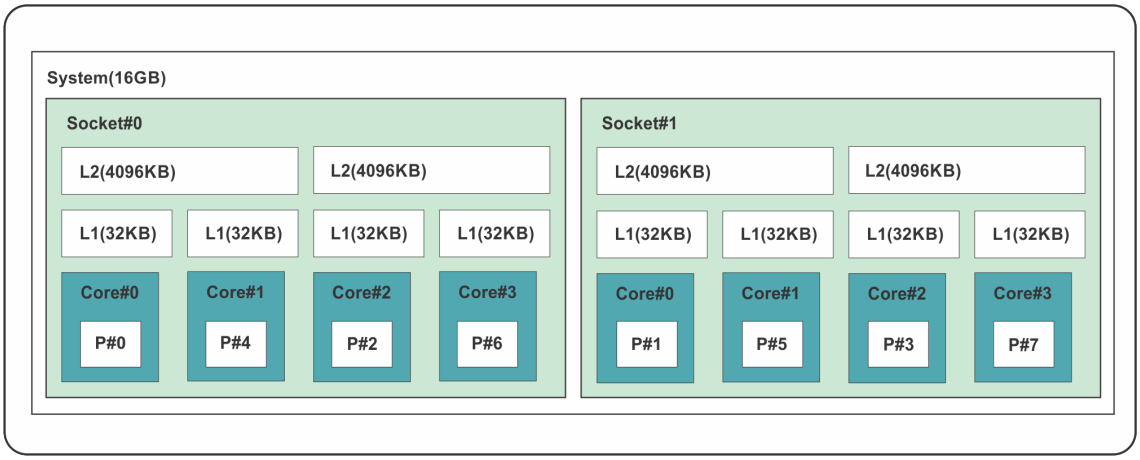


Figura 6: Representação gráfica do comando *lstopo* referente a arquitetura XEON apresentada na Figura 5.

Além da representação gráfica, a topologia da máquina pode ser armazenada em outros formatos como texto, xml, pdf, entre outros. Pode ainda ser obtida em tempo de execução com o uso da interface que a ferramenta disponibiliza (BROQUEDIS et al., 2010), como por exemplo o Código 4 a seguir, que imprime no terminal a topologia da máquina.

Código 4: Interface Hwloc para aquisição da topologia.

```
1  hwloc_obj_t obj;
2  depth = hwloc_topology_get_depth(t);
```

```
3   for (d = 0; d < depth; d++) {
4     n = hwloc_get_nbobjs_by_depth(t, d);
5     for (i = 0; i < n; i++) {
6       obj = hwloc_get_obj_by_depth(t, d, i);
7       printf("%d,%d type %d\n",
8           d, i, obj->type);
9   }
```

Para obter o máximo possível de informações sobre o hardware da máquina, a ferramenta *HwLoc* foi estendida por Pilla et al. (2013), para que fornecesse uma visão genérica e completa de qualquer arquitetura de computador. O modelo da arquitetura foi melhorado, passando a oferecer estatísticas de memória, como por exemplo os tempo de latência para buscar dados a partir de diferentes níveis de *cache* e memória compartilhada e também estatísticas das interconexões, como por exemplo a largura de banda para o tráfego dos dados. Em outras palavras, a ferramenta passou a fornecer informações da distância entre os núcleos de processamento da máquina com base nos tempos de latência nos acessos à memória e largura de banda para a transmissão de dados.

Essas informações são geradas em forma de árvore, a qual é composta pela máquina na raiz e os processadores nas folhas, contendo todos os níveis de *cache* nos níveis intermediários. Um exemplo de arquivo gerado pela ferramenta é apresentado a seguir, na Figura 7. Trata-se de uma máquina com dois processadores, cada um com dois níveis de memória *cache*.

Figura 7: Exemplo de arquivo de topologia gerado pela ferramenta *HwLoc*.

```
1  Machine#0(3954MB)
2    Socket#0
3      L2(2048KB)
4        L1d(32KB)
5          Core#0
6            PU#0
7        L1d(32KB)
8          Core#1
9            PU#1
10 Latencies:
11 0 0 1.295176
12 0 1 6.600822
13 1 0 6.600822
14 1 1 1.295176
15 Bandwidths:
16 0 0 35448.410156
17 0 1 18232.800781
18 1 0 18232.800781
19 1 1 35448.410156
```

As estatísticas de memória são adquiridas aplicando o *benchmark LMbench*[2]

[2]LMbench – http://lmbench.sourceforge.net/

(STAELIN; PACKARD, 1996). As latências (*Latencies*) e larguras de banda (*Bandwidths*) são aferidas em todos os níveis de *cache* e nós NUMA com os *benchmarks* lat_mem_rd, bw_mem e $ping - pong$, escritos em *coNCePTuaL*[3] (PAKIN, 2007). *HwLoc* ajuda nesse processo fornecendo informações sobre o tamanho de cada bloco de *cache* e a capacidade de vincular os processadores com a memória. Este último é importante para avaliar os custos de acesso à memória entre pares de nós NUMA.

A Figura 8 ilustra graficamente o modo como os parâmetros da topologia são recuperados pela ferramenta *HwLoc* em uma máquina com quatro processadores e dois níveis de *cache*.

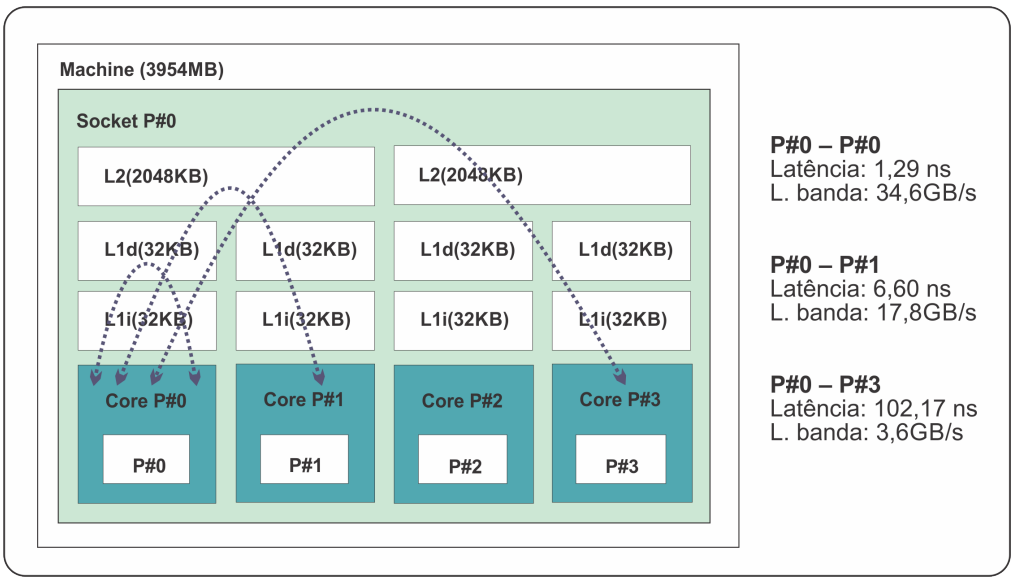


Figura 8: Exemplo de latências e larguras de banda obtidas com a ferramenta *HwLoc*.

As principais vantagens da utilização deste modelo como descrição da topologia da máquina são: (i) modelo genérico, pode ser facilmente calculado por diferentes máquinas e bibliotecas de programação, (ii) agrega as diferentes características de máquinas *multicore*. Além disso, as informações de hierarquia de *cache* e latências de acesso a memória podem ser pré-computadas e armazenadas antes da execução da aplicação, o que reduz o *overhead*, uma vez que elas não precisam ser calculadas novamente toda vez que uma aplicação for executada (PILLA et al., 2012, 2013).

Essas informações servem de base para gerar uma tabela de latências entre todos os núcleos de processamento disponíveis na máquina. Essa tabela de latências é utilizada pelo escalonador no momento da tomada de decisões para alocar, de maneira mais eficiente, as tarefas produzidas pelo programa paralelo em execução.

[3]coNCePTuaL – http://www.ccs3.lanl.gov/ pakin/software/conceptual/

3.3 Anahy

O modelo Anahy (CAVALHEIRO et al., 2007) define um ambiente de execução *multithread* e uma interface de programação para este ambiente. A interface de programação é minimalista, basicamente composta pelas primitivas $init$ e $terminate$, para inicializar e terminar o ambiente, respectivamente, 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. As unidades responsáveis por realizar a execução do programa paralelo são compostas por um conjunto de objetos de ambiente chamados Processadores Virtuais (PVs). Estes objetos contêm em sua estrutura uma pequena unidade de execução implementada como um *thread* de sistema, esta unidade é responsável por executar o função contida no *job*.

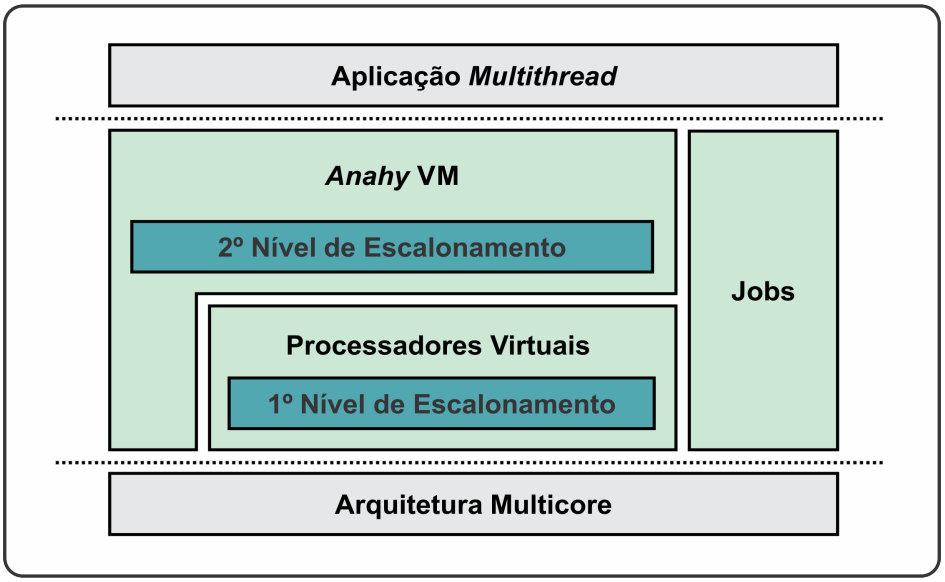


Figura 9: Arquitetura do ambiente Anahy.

A Figura 9 apresenta a arquitetura em camadas do Anahy. O primeiro nível é definido por um escalonador interno a cada processador virtual. Neste nível, cada escalonador gerencia os *jobs* locais de seu PV. O princípio desta distribuição é de que o próprio PV pode implementar a sua estratégia de escalonamento, permitindo que se implemente um escalonador especifico para cada tipo de *job*.

O segundo nível de escalonamento diz respeito à criação de *threads* na camada de aplicação, sendo de responsabilidade do gerente do ambiente distribuir estes *threads* descritos no código do usuário entres os PVs da arquitetura. A seção a seguir apresenta a interface de programação para utilizar Anahy.

3.3.1 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 *jobs*, e lançar à execução um *job*, para mais adiante sincronizá-lo. O Código 5 traz essas interfaces.

Código 5: Interface para tarefas básicas em Anahy.

```
1  /* Inicializa o ambiente de execução */
2  static void AnahyVM::init(int argc, char **argv);
3
4  /* Finaliza o ambiente de execução */
5  static void AnahyVM::terminate();
6
7  /* Cria um job */
8  AnahyJob* job = new AnahyJob(foo, NULL);
9
10 /* Lança jobs para execução */
11 static void AnahyVM::fork(AnahyJob* job);
12
13 /* Sincroniza jobs */
14 static void AnahyVM::join(AnahyJob* job, void** result);
```

O Código 5 mostra os métodos principais da API da Máquina Virtual de Anahy. 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.

Quando um *job* é criado, ocorre apenas o encapsulamento, em um objeto, das referências para uma função e seus argumentos (entre outros valores). No entanto, para que fique explícito ao ambiente de execução que este *job* pode ser executado de maneira concorrente aos demais *jobs*, deve-se chamar o método *fork* da máquina virtual.

`AnahyVM::fork` recebe um ponteiro para o *job* como parâmetro e delega o *fork* ao PV atual, que o insere em sua própria lista para que, futuramente, seja executado por um PV ocioso. Da mesma forma, quando chama-se `AnahyVM::join`, é passado 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.

Antes de um *job* ser passado como argumento para `AnahyVM::fork`, o programador deve criar um objeto do tipo *AnahyJob* e inicializá-lo de maneira apropriada. Uma das maneiras de fazer isso está descrita na linha 8 do código apresentado. Na linha 8, um novo *job* é alocado por meio da palavra-chave **new** de *C++*. Objetos criados desta maneira, explicitamente alocados pelo programa-

dor, devem ser desalocados pelo próprio programador em um momento futuro da execução(*delete job;*).

3.3.2 Modelo de execução e escalonamento

A estratégia de escalonamento do modelo Anahy é baseada em lista de tarefas, onde estas tarefas são encapsuladas no contexto de *threads*, em Anahy conhecidos como *Jobs*. Uma camada intermediária entre a interface de programação e o ambiente de execução é responsável por identificar a concorrência em tarefas e criar um grafo acíclico direcionado (DAG) para representar as dependências entre estas tarefas. O grafo é explorado por algoritmos de escalonamento de listas. Neste DAG, cada vértice corresponde a uma tarefa a ser executada pelo programa e cada aresta dirigida uma comunicação de dados entre as tarefas.

Os PVs são os consumidores do trabalho descrito pelos *jobs*, onde este trabalho inclui também as atividades de escalonamento geradas pela execução dos *jobs*. O escalonamento se dá pelo roubo de trabalho e ocorre da seguinte maneira: um PV ocioso busca um *job* localmente em sua lista; caso não obtenha nenhum, o PV ocioso escolhe um outro PV, aleatoriamente, e tenta roubar um *job* de sua lista. Quando todos os PVs estiverem ociosos, ou seja, tentando roubar trabalho, significa que todos os *jobs* da aplicação já foram consumidos e a aplicação é finalizada.

3.4 Considerações sobre o capítulo

Neste capítulo foi apresentado todo o ferramental utilizado no desenvolvimento deste trabalho, inicialmente apresentou e caracterizou as arquiteturas com acesso não uniforme à memória, as arquiteturas NUMA. Onde, para se conseguir bons índices de desempenho neste tipo de arquitetura, é preciso considerar as diferentes latências de acesso aos dados nos blocos de memória presentes na máquina.

Para se extrair as informações da arquitetura foi utilizada a ferramenta *HwLoc* que, conforme apresentado na Seção 3.2, é capaz de obter uma série de atributos da arquitetura, entre estes, as latências de acessos aos dados a partir de cada um dos processadores.

Por fim, foi apresentado o ambiente Anhay, o qual recebeu a estratégia de escalonamento proposta neste trabalho, a fim de contribuir com o aumento de desempenho deste ambiente. O próximo capítulo trata da modelagem da estratégia proposta, apresentando o modelo de aplicação, o modelo de execução, o modelo de arquitetura e o modelo de escalonamento.

4 SOLUÇÃO PROPOSTA

Este capítulo trata das principais questões envolvendo a estratégia desenvolvida. A Seção 4.1 apresenta o modelo de aplicação, o qual mostra como as tarefas produzidas por uma aplicação paralela compõem um *thread* e a representação das dependências entre *threads*. Na sequência, na Seção 4.2, é descrito o modelo de execução, discutindo o modelo de execução adotado.

Na Seção 4.3 é descrito o modelo de arquitetura, onde são apresentadas as características das arquiteturas NUMA, as quais devem ser consideradas no momento do escalonamento para que consiga melhorar os índices de desempenho. Na Seção 4.4, é apresentado o modelo de escalonamento, o qual se dá em dois níveis. Este modelo de escalonamento considera as características da arquitetura.

A Seção 4.5 trata das características do modelo de escalonamento proposto, segundo (CASAVANT; KUHL, 1988). Por fim, a Seção 4.6 aborda questões referentes ao processo de escalonamento da estratégia proposta, mostrando os principais passos realizados durante a execução da aplicação.

4.1 Modelo de aplicação

Trata-se de uma aplicação *multithread*, ou seja, uma aplicação que contém diversos fluxos de execução executando tarefas distintas. A Figura 10 ilustra a execução de uma aplicação *multithread* convencional em termos de criação e execução de *threads*.

Como pode-se observar na Figura 10, cada *thread* Γ_i é composto por um conjunto de k tarefas $\tau_{i,j}$ que são executadas de maneira sequencial. Cada tarefa consiste em um conjunto de instruções executadas de forma sequencial. As tarefas são delimitadas por operações de escalonamento. A tarefa $\tau_{i,1}$ é criada simultaneamente a criação do *thread* Γ_i e o término da tarefa $\tau_{i,k}$ o finaliza. As demais tarefas $i, j | 1 \leq j \leq k$ são iniciadas imediatamente a invocação de primitivas de criação de novos *threads* ou de sincronização.

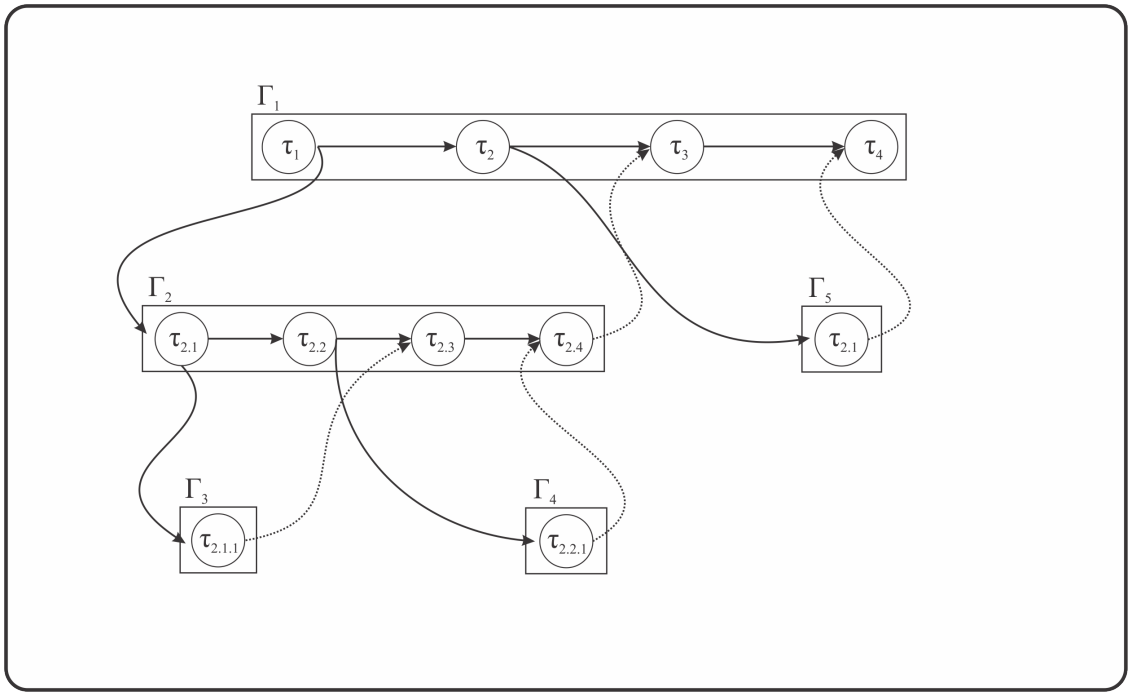


Figura 10: Execução de uma aplicação multithread.

Na figura apresentada, as arestas entre uma *thread* `i` e uma *thread* `j`, tal que `i < j`, representam uma criação de *thread* e uma aresta tal que `i > j`, uma operação de sincronização. As arestas direcionadas nas tarefas dentro de um *thread* determinam a ordem de execução destas tarefas. De uma maneira mais formal, estas arestas são chamadas arestas de dependência e definem a execução sequencial das tarefas localizadas dentro do escopo do *thread* ao qual elas pertencem.

Neste modelo de aplicação, as primitivas de exclusão mútua como semáforos e mutexes, não estão disponíveis. Cada uma das instruções (tarefas) são executadas respeitando-se a ordem de dependência. No exemplo apresentado na Figura 10, a tarefa τ_4 não pode ser executada até que a tarefa τ_3 tenha terminado. Conforme a execução da aplicação *multithread* vai avançando, os processadores virtuais criam seus próprios *threads* locais e os executam. É possível estruturar as dependências entre os *threads* de cada processador por meio de uma árvore de dependências entre *threads*.

Na abstração proposta para um programa *multithread*, cada *thread* possui um identificador único, gerado na sua criação. Este identificador único pode ser utilizado para sincronizar, com uma operação de *join*, dois ou mais *threads*, desde que o *thread* que desejar realizar a sincronização tenha acesso ao identificador. Desta forma, a estrutura do programa é livre, oferecendo uma maior liberdade ao programador, não restringindo o programa a uma estrutura pré-definida, como, por exemplo em Cilk e OpenMP, com uma estrutura *fork*/*join* aninhada. A aplica-

ção *multithread*, como descrita neste modelo, quando terminar de executar, pode ser representada por um DAG, representando as tarefas e as dependências de dados entre elas. A seção a seguir trata desta abstração.

4.1.1 Representação de aplicações *multithread*

Uma aplicação *multithread* pode ser descrita em termos de suas criações e sincronizações de *threads*, por meio de um Grafo Dirigido Cíclico – DCG (*Directed Cyclic Graph*), onde os nós representam os *threads* e as arestas representam operações de criação e sincronização entre esses *threads*. Esse modelo de aplicação permite a comunicação entre os *threads* apenas quando um novo *thread* é criado ou no momento da sincronização de um ponto de execução de um *thread* com o final da execução de outro (SOUZA CAMARGO, 2013).

Como foi apresentado anteriormente, o modelo de programação adotado por Anahy é baseado nas operações *Fork*/*Join*. Basicamente, uma operação *Fork* cria um novo *thread* para ser executado e uma operação *Join* realiza a sincronização com *threads* anteriormente criados para obter seus resultados. Isso permite construir um grafo de precedência entre os *threads* do programa. Para exemplificar, o Código 6, representa um exemplo simples de aplicação *multithread* que faz uso das primitivas *Fork* e *Join*.

Código 6: Exemplo de código de uma aplicação *multithread*.

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

A Figura 11 ilustra a criação do grafo no decorrer da execução do programa *multithread* apresentado no Código 6. Em cada uma das subfiguras (a, b e c) para cada *thread* (Γ_1 e Γ_2) a tarefa em execução no momento está em destaque.

O programa inicia executando τ_A, a primeira tarefa do *thread* inicial (*main*) do programa, representado na Figura 11(a) por Γ_1. A Figura 11(b) representa o estado do grafo logo após a execução do comando *fork(t)*, no programa apre-

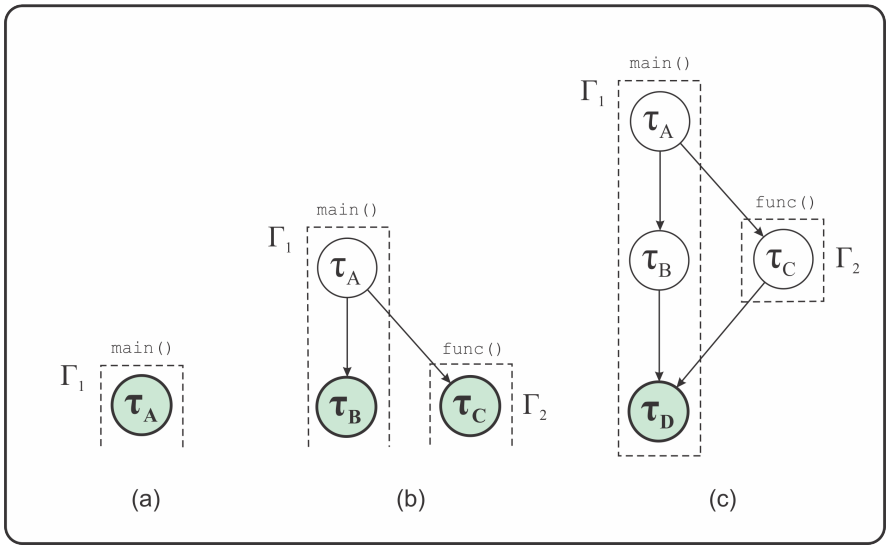


Figura 11: Grafo gerado durante a execução do programa *multithread* apresentado no Código 6.

sentado pelo Código 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. No entanto, quando o comando *join(t)* é executado, a tarefa τ_D só é criada depois que Γ_2 for completamente executado, o que já ocorreu no momento ilustrado pela Figura 11(c). Neste instante, τ_D passa a ser a tarefa atual de Γ_1.

Abstraindo-se o nível de tarefas e considerando apenas o nível de *threads*, o possível DCG gerado ao final da aplicação *multithread* apresentada no Código 6 é grafo apresentado na Figura 12(a) a seguir. Por se tratar de uma aplicação bastante simples, com apenas 2 *threads*, o DCG obtido ao final da execução deste código ilustra esses 2 *threads* e a relação de dependência entre eles. A seta com linha contínua representa a criação de um *thread*, já a seta com linha pontilhada indica uma sincronização entre dois *threads*. Na Figura 12(b), está representado o DCG de uma aplicação *multithread* qualquer, com 4 *threads*.

Na próxima seção (Seção 4.2) é apresentada a maneira como esses *threads* são mantidos e executados. Esta seção apresenta as premissas de escalonamento empregadas pelos processadores virtuais (PVs) do ambiente de execução de Anahy.

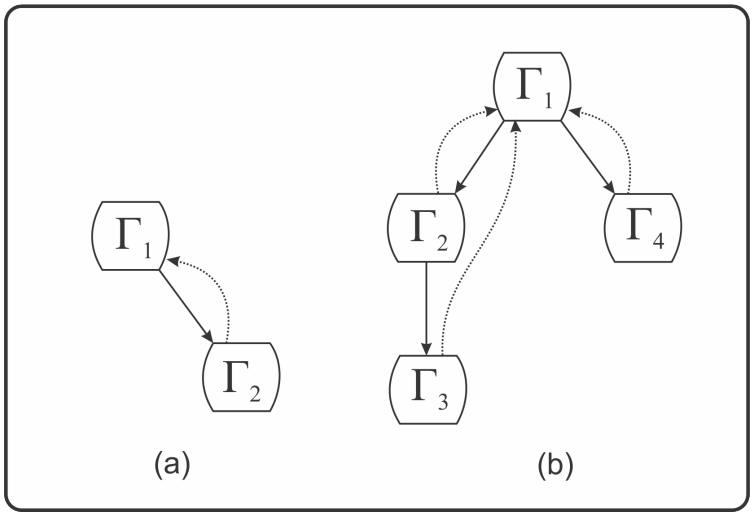


Figura 12: Exemplos de DCGs: (a) DCG gerado a partir do Código 6 e (b) DCG de uma aplicação *multithread* qualquer.

4.2 Modelo de execução

O núcleo de execução do Anahy é uma camada de software que oferece uma abstração de arquitetura multiprocessadas e escalona os *threads* gerados pelo programa em execução sobre os recursos computacionais desta arquitetura. Este núcleo de escalonamento é composto por 3 partes básicas, a saber: (i) *Jobs* (tarefas), (ii) Processadores Virtuais e a (iii) Máquina Virtual, apresentados nas subseções a seguir.

4.2.1 Jobs

O *Job* é a unidade de escalonamento do ambiente Anahy, uma porção de trabalho que deve ser executado por algum dos PVs. Um *job* descreve um *thread* no nível de aplicação e sua estrutura de dados mantém o endereço estático da função que inicia um *thread*, um ponteiro para os argumentos que essa função deve receber e um ponteiro para a área de memória na qual os resultados dessa função devem ser escritos.

No início da execução de um programa é lançada a execução do *job* raiz, representando um *thread* criado de forma implícita para executar a função `main`. A partir da execução deste *job* raiz, outros *jobs* podem ser criados e sincronizados de forma explícita.

Cada *job* criado durante a execução da aplicação mantém uma referência para o *job* que o criou, ou seja, o *job* "pai". A partir dessas referências é possível criar uma DCG com as relações de precedências entre os *jobs*. As dependências de espera (quando um *job* precisa esperar pelos resultados produzidos por outro *job*) também são inseridas neste DCG. O ambiente de execução escalona todos os *jobs* buscando reduzir o tempo total de execução do programa, respeitando

todas as dependências de dados entre os *jobs*.

Um *job*, quando criado e estiver pronto para ser executado, é escalonado para ser executado no próprio processador que o criou. Seu lançamento não é imediato, sendo inserido em uma estrutura de dados (uma fila com duplo fim). Eventualmente o escalonador seleciona um *job* para ser executado. Neste momento, um *job* é lançado e será executado sobre o processador que o lançou sem que seja preemptado ou migrado para outro processador. A única condição para que um *job* seja interrompido é quando ele próprio executar uma operação de sincronização (*join*) e o *job* solicitado não estiver, neste momento, já concluído.

4.2.2 Processadores Virtuais

Os processadores virtuais consomem o trabalho descrito pelos *jobs*. Cada PV é responsável por criar e sincronizar seus próprios *jobs*. Para isso, são utilizadas duas estruturas de dados. Uma fila de *jobs* prontos para execução e uma pilha de *jobs* bloqueados, utilizada para salvar o contexto do *job* atual de um PV durante a execução de uma operação de sincronização.

A fila de *jobs* prontos consiste em uma fila com duplo fim. Localmente o PV insere e retira *jobs* do início da fila, privilegiando uma execução que explore tanto a localidade de referência implícita aos *threads* criados pelo programa, como uma execução em profundidade do programa. A seleção de um *thread* para migração recai sobre os *jobs* na cauda desta fila, de forma a não comprometer as heurísticas do escalonamento local, conforme estratégia adotada em (BLUMOFE et al., 1995). O tamanho desta fila também é utilizado para determinar o quão sobrecarregado de *jobs* o PV está. A Figura 13 ilustra essa ideia.

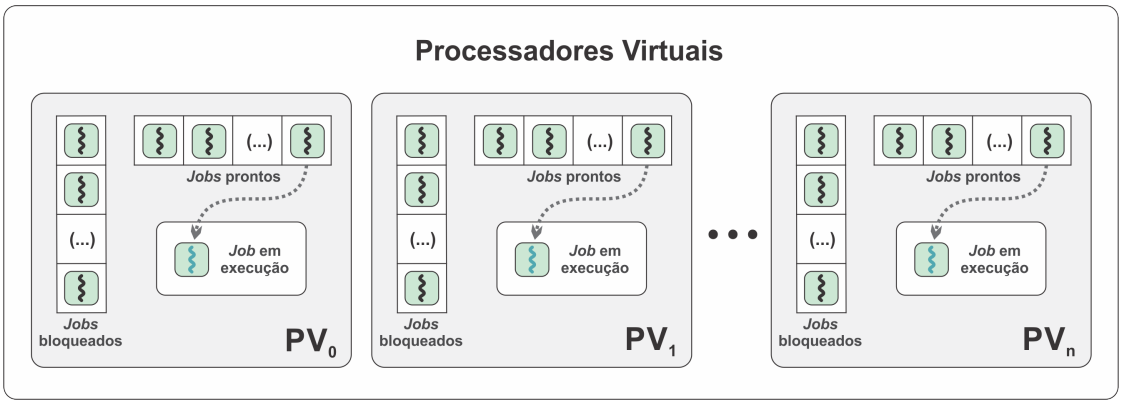


Figura 13: Arquitetura do ambiente de execução do Anahy.

Cada PV executa de maneira sequencial as tarefas que a ele forem alocadas, não preemptando *jobs* em sua execução. O princípio de desempenho do ambiente parte da premissa de que PVs não podem ficar ociosos. Para que isso seja possível, cada PV, durante seu tempo de vida, cria seus *jobs* de maneira

independente. Desta forma, o grafo de precedência de *jobs* é distribuído entre as unidades de execução do ambiente, sendo que os pontos de ligação entre as partes estão relacionados com as migrações de *jobs* efetivadas.

Quando um PV não possui *jobs* em sua lista local, ele realiza um roubo de trabalho em outro PV. Portanto, um processador sem trabalho assume o estado de *idle*, e então percorre a lista local dos outros PVs, selecionados um a um randomicamente (KARP; ZHANG, 1993), em busca de trabalho. Os *jobs* são atribuídos aos PVs por meio do gerenciador *AnahyVM*, o qual é descrito a seguir.

4.2.3 Máquina Virtual

AnahyVM é a máquina virtual do ambiente *Anahy*, é responsável por criar e gerenciar os PVs. A máquina virtual é composta por dois *arrays*, um *array* de `n` PVs e um outro array de `n - 1` *threads* de sistema (*threads* POSIX) que executam a lógica de cada PV em paralelo. Este segundo *array* possui um elemento a menos porque um PV é executado sobre o *thread* principal do programa (*thread* `main`).

A máquina virtual *AnahyVM* permite inicializar (por meio da primitiva `AnahyVM::init`) e terminar (por meio da primitiva `AnahyVM::terminate`) o ambiente de execução, criar e configurar *jobs*, e colocar um *job* em execução (com `AnahyVM::fork`), para depois sincronizá-lo (com `AnahyVM::join`). *AnahyVM* 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.

Um dos parâmetros passado é o número de PVs a serem utilizados. Apesar de os PVs serem criados por *AnahyVM*, o escalonamento destes é realizado pelo sistema operacional. Pode-se atribuir afinidades aos *threads* de sistema a processadores, para determinar que um PV seja escalonado pelo sistema operacional sempre sobre o mesmo processador físico.

A estratégia de escalonamento é realizada sobre esta máquina virtual, resultando da colaboração da lógica distribuída entre os PVs. Desta forma, em um mesmo programa não devem coexistir duas máquinas virtuais, sobre o risco das decisões de escalonamento tomadas em cada máquina virtual serem incompatíveis entre si. A consequência, para o programador, é que ele não pode fazer uso de ligação dinâmica de programas na atual versão de Anahy (Athreads).

4.3 Modelo de arquitetura

As arquiteturas alvo para o desenvolvimento da estratégia foram as arquiteturas NUMA. Conforme apresentado na Seção 3.1, um computador com arqui-

tetura NUMA apresenta memória distribuída, ou seja, nestas máquinas a memória é fisicamente composta por vários blocos, podendo estar, cada um deles, vinculados a um determinado processador ou um determinado conjunto de processadores. Para exemplificar, a Figura 14 ilustra uma arquitetura NUMA, a qual servirá de base para a apresentação e discussão de alguns conceitos.

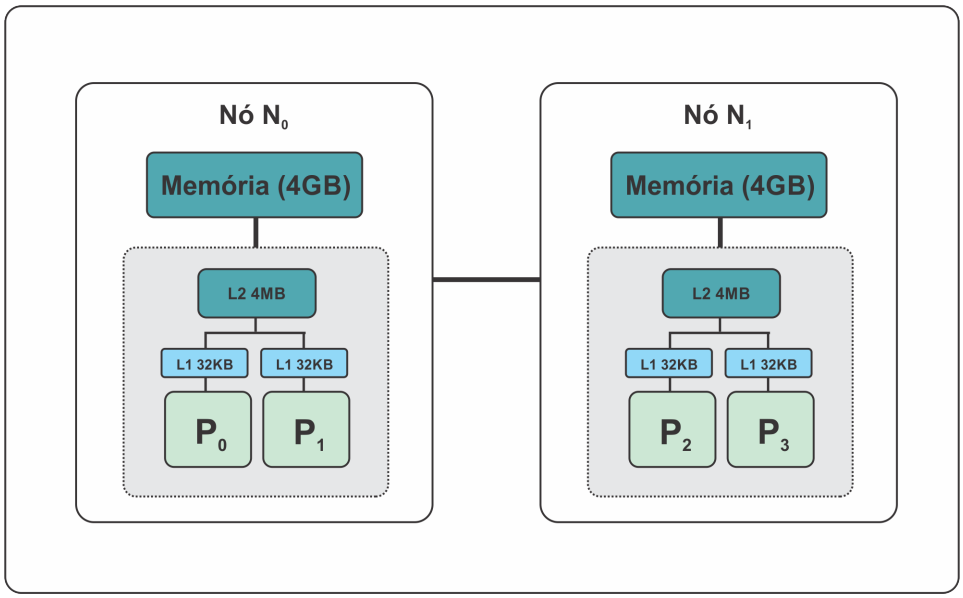


Figura 14: Exemplo de uma arquitetura com acesso não uniforme à memória (NUMA).

Na Figura 14 está representada uma arquitetura NUMA composta por dois processadores. Cada um destes processadores possui dois núcleos (*cores*), com dois níveis de cache (`L1` de 32KB e `L2` de 4MB). Quanto à memória, esta é dividida em dois blocos de 4GB, onde cada bloco está vinculado a um dos processadores. Nesta arquitetura, existem diferentes latências para acessar a memória dependendo de onde partir o acesso.

Por exemplo, se o *core* P_0 estiver executando uma tarefa cujos dados de entrada para esta tarefa estão armazenados em sua *cache* `L1`, o tempo necessário para acessar estes dados será menor do que se os dados estivessem armazenados na *cache* `L2` do *core* P_2. O mesmo ocorre se o *core* P_3 precisar fazer um acesso à memória do nó N_0 ao invés de acessar a memória do seu próprio nó.

Em se tratando de roubo de tarefas, têm-se o mesmo princípio. Se um processador virtual estiver alocado no *core* P_0 localizado no nó N_0 e precisar realizar um roubo, se este roubo for realizado dentro do mesmo nó (por exemplo o processador P_1, também dentro do nó N_0) o tempo necessário para buscar trabalho será menor do que buscar trabalho nos processadores P_2 ou P_3 no Nó N_1. Cada roubo de trabalho que acontecer em um nó que não seja o local, acarretará em uma maior latência, prejudicando certamente o desempenho do programa. Algu-

mas vezes, estas características das arquiteturas NUMA não são consideradas pelo programador no desenvolvimento de aplicações paralelas, sendo estas tratadas como arquiteturas SMP (com memória centralizada e os tempos de acesso à memória se dão de maneira uniforme).

Ignorar as diferentes latências provenientes da arquitetura NUMA, na maioria das vezes, há um impacto muito grande na execução da aplicação fazendo com que bons índices de desempenho não sejam alcançados. A forma com que os processadores e blocos de memória estão organizados em uma arquitetura NUMA e as diferentes latências de acesso aos dados permitem uma boa compreensão geral da arquitetura. Além disso, essas informações servem de subsídios ao desenvolvedor para que ele possa definir suas estratégias de paralelização de aplicações neste tipo de arquitetura.

Portanto, o conhecimento prévio das características da arquitetura precisam ser considerados no momento do escalonamento. A seguir, é descrita maneira como as tarefas produzidas por uma aplicação paralela são escalonadas entre os processadores disponíveis. Este escalonamento é realizado em dois níveis, onde as tarefas são escalonadas entre os processadores virtuais do ambiente de execução e estes, por sua vez, são escalonados entre os processadores físicos da arquitetura.

4.4 Modelo de escalonamento

A Figura 15 ilustra o modelo de escalonamento da estratégia proposta. Como está representado, o escalonamento acontece em dois momentos, no nível da aplicação, onde as tarefas produzidas pela aplicação paralela são distribuídas entre os PVs e, no nível de sistema operacional, onde estes PVs são distribuídos entre os processadores físicos presentes na arquitetura. Em nível aplicativo, a estratégia de escalonamento é baseada em lista de tarefas, onde o escalonamento acontece por meio de roubo de trabalho nessa lista.

Um escalonamento baseado em roubo de trabalho mantém uma lista de trabalhos aptos à serem executados. Neste caso, trata-se de uma lista global mantida de maneira distribuída, ou seja, no escopo de cada processador virtual e tem como princípio que, roubos e operações de inserção e remoção acontecem em diferentes extremos. Um PV insere tarefas aptas na frente da sua lista, e também as retira na mesma extremidade, caracterizando uma pilha.

Sempre que um PV for executar um trabalho, ele busca por um *job* em sua lista de *jobs* local e, ao escolher um *job*, o eleito é o mais recente criado, o que é ideal para o modelo *fork*/*join* aninhado. O trabalho mais recentemente criado está no caminho crítico da aplicação, uma vez que quem o criou precisa de seu

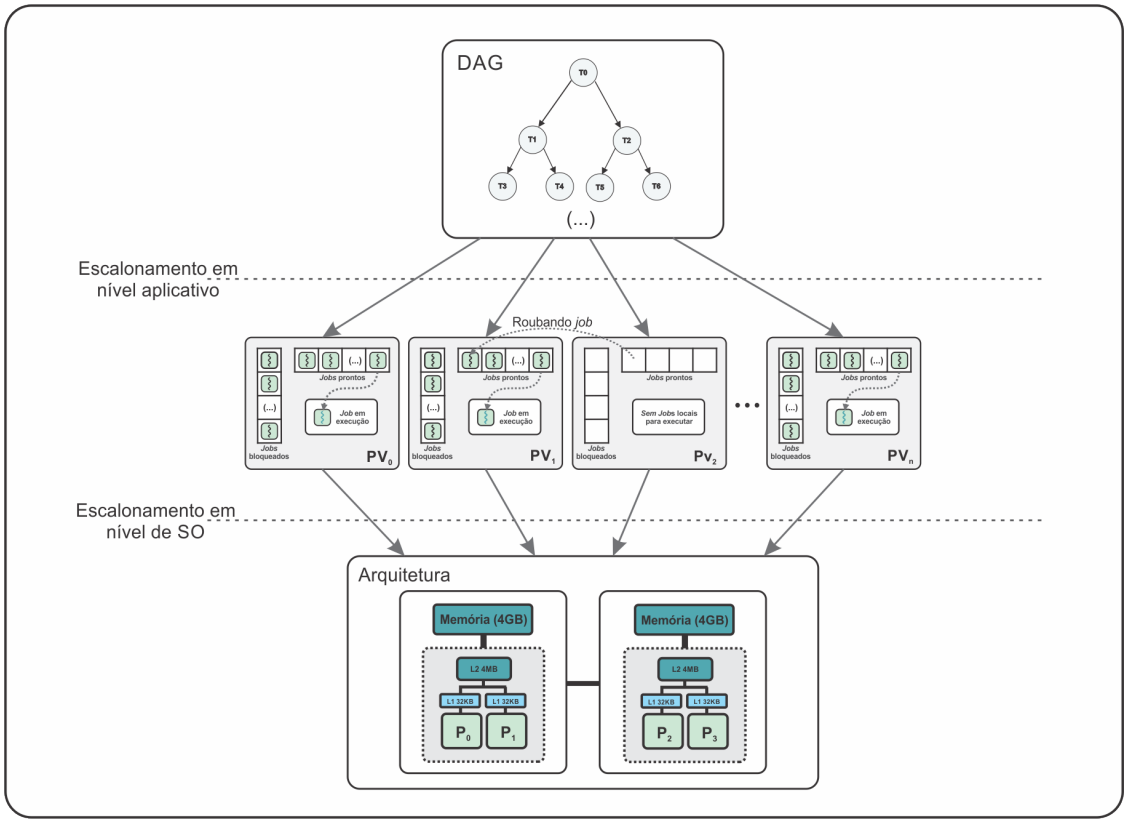


Figura 15: Ilustração do modelo de escalonamento em dois níveis (Aplicativo e Sistema Operacional).

resultado para prosseguir com a execução.

Quando esta lista local ficar vazia, o PV precisa buscar trabalho em outros PVs, tornando-se um ladrão e, então, escolhe um outro VP (vítima) para realizar o roubo de trabalho. A tarefa roubada é retirada do final da lista local do PV vítima, sem disputar tarefas com o PV detentor desta lista de *jobs*.

Para escolher uma vítima no momento do roubo de trabalho, um VP consulta sua lista de prioridades, a qual mantém, de maneira ordenada da menor para a maior latência de acesso, todos os PVs que podem ser roubados. Para criar essa lista de prioridades para roubo são consideradas as informações providas pela ferramenta *HwLoc*, apresentada na Seção 3.2. Para exemplificar, vamos considerar a Figura 16, a qual ilustra essa ideia.

A Figura 16 mostra seis PVs executando sobre uma arquitetura NUMA composta por quatro processadores. As setas pontilhadas indicam as afinidades de cada PV, ou seja, o processador no qual o PV está executando. Os PVs PV_0 e PV_4 executam no processador P_0, os PVs PV_1 e PV_5 executam no processador P_1, o PV PV_2 executa no processador P_2 e o PV_3 executa no processador P_3.

Para exemplificar, vamos considerar que o PV_4 não possui mais nenhuma tarefa pronta pra ser executada em sua lista local. Com base nas informações fornecidas pela ferramenta *Hwloc* (na Figura 16 representadas pelo retângulo

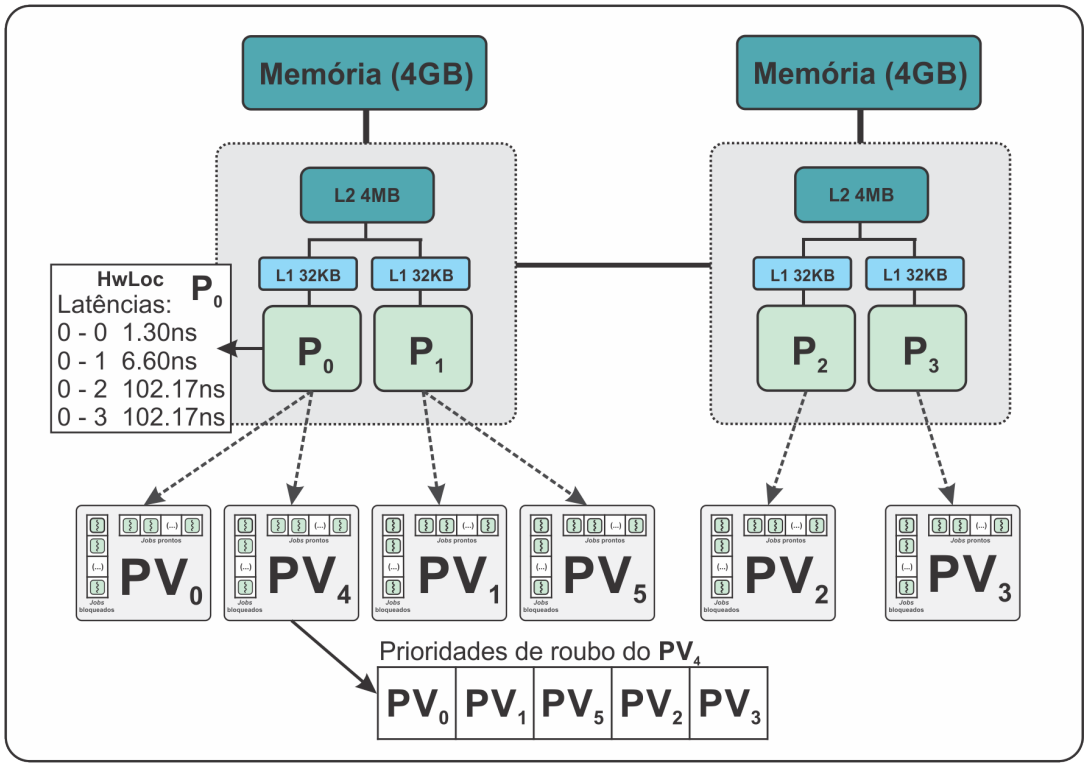


Figura 16: Exemplo de como é criada a lista de prioridades para roubo de um PV.

branco vinculado ao processador P_0, contendo as latências de P_0 aos demais processadores) é possível definir uma lista de prioridades para roubo, considerando como prioritários os PVs que possuem menor latência na busca de trabalho.

Como PV_4 está executando em P_0, segundo a tabela de latências fornecida para este processador, a menor latência está no acesso à memória *cache* dele mesmo (1.30ns), logo o PV_4 deverá, inicialmente, procurar trabalho na lista de tarefas do PV_0, que também executa neste processador. Caso não consiga roubar tarefas do PV_0, os próximos PVs da lista de prioridade são os PVs PV_1 e PV_5, executando no processador P_1, cuja latência para acesso é 6.60ns. Por fim, com a maior latência de acesso (102.17ns) estão PV_2 e PV_3. E assim, com base nas latências fornecidas pela ferramenta *HwLoc* as prioridade de roubo de cada PV são definidas.

Em se tratando de operações de roubo de trabalho, para escolher entre um *job* da lista de *jobs* do PV vítima, a política de escolha é inversa a política de escolha da execução local, o *job* a ser roubado é o mais antigo criado, por possuir uma maior probabilidade de gerar mais trabalho, o que deixará o PV ladrão ocupado por mais tempo, tendendo a diminuir a quantidade de roubos. Isso justifica-se por dois motivos importantes (FRIGO; LEISERSON; RANDALL, 1998):

- Somente há contenção entres os PVs quando o número tarefas for menor ou igual ao número de PVs competindo por elas. Também, as operações de inserção e remoção de *threads* acontecem no mesmo extremo da pilha, como estes *threads* são concorrentes entre si, o princípio de execução do PV não é prejudicado. Isso contribui com a exploração do paralelismo no outro extremo, onde os roubos de trabalho acontecem.

- Este tipo de roubo alcança altos índices de desempenho para algoritmos que seguem o princípio de divisão e conquista, pelo fato de que este tipo de algoritmo possui uma granularidade fina e o roubo ajuda na execução de trabalhos mais antigos criados.

Operações de roubo de trabalho são mais frequentes em dois momentos da execução: (i) quando o programa é iniciado e as listas dos PVs estão vazias, onde apenas o PV *main* executa o programa e gera trabalho e (ii) próximo ao final da execução, quando as listas dos PVs voltam a ficar com poucas tarefas, pois a execução do programa está chegando ao seu final.

Em nível de sistema operacional, estes PVs precisam ser distribuídos entre os processadores físicos da arquitetura, para tal, utiliza-se uma política de distribuição circular. Esta política circular faz com que haja um balanceamento do número de PVs entre os processadores reais da arquitetura. Este balanceamento tem por objetivo não deixar ociosos os recursos da máquina, distribuindo os PVs entre todos os processadores disponíveis, uma vez que o número de PVs pode ser maior que o número de processadores físicos presentes na arquitetura. A Figura 17 ilustra um exemplo de distribuição de seis processadores virtuais nos quatro processadores físicos presentes na arquitetura.

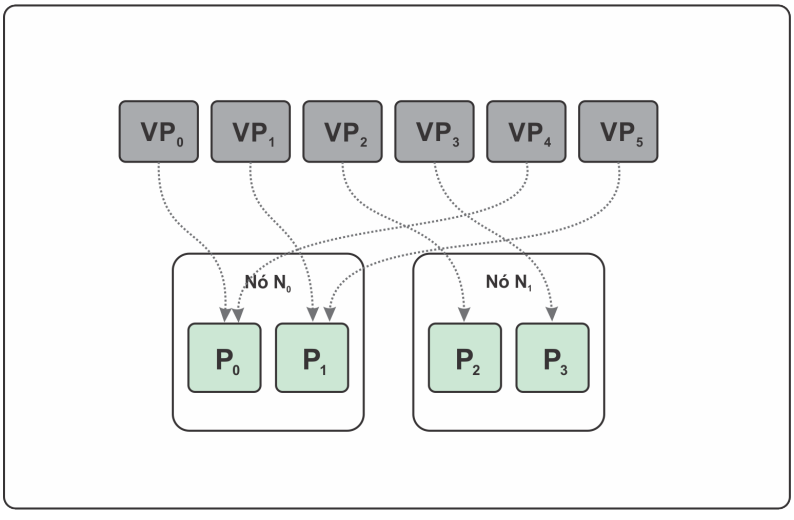


Figura 17: Política de distribuição dos PVs.

4.5 Características do modelo de escalonamento proposto

Nesta seção, a estratégia de escalonamento é classificada segundo (CASAVANT; KUHL, 1988) apresentada na Seção 2.2. O modelo de escalonamento proposto neste trabalho é realizado em nível aplicativo, onde é possível explorar atributos do programa em tempo de execução. Trata-se, então, de uma abordagem de escalonamento dinâmica, ou seja, todas as informações que descrevem as atividades concorrentes da aplicação e suas dependências não são conhecidas, ou tratadas quando disponíveis, previamente. A alocação das tarefas é realizada com base em informações adquiridas durante a execução da aplicação.

Para evitar um gargalo em termos de desempenho, as decisões de escalonamento são tomadas de maneira distribuída, baseadas em roubo de trabalho. O roubo de trabalho pelos PVs ocorre de maneira não cooperativa, ou seja, as decisões de roubo de trabalho de cada PV são tomadas sem considerar as cargas de trabalho dos demais PVs. Com esta estratégia não é esperado que a carga esteja distribuída de forma balanceada entre os processadores, mas sim que todos estejam ativos.

No modelo proposto, uma tarefa, depois de iniciada as sua execução, não pode ser interrompida e retomada posteriormente, uma vez que a execução da tarefa é iniciada, ela executará até a sua conclusão, caracterizando um escalonamento não preemptivo. Buscando explorar o potencial de execução concorrente de maneira mais eficiente, este modelo de escalonamento segue uma política *non-blind*, na medida em que aplica uma heurística que considera que os *threads* mais antigos são os que geram maior carga computacional e os mais recentes referenciam dados em posições próximas de memória.

4.6 O processo de escalonamento

O processo de escalonamento da estratégia proposta pode ser dividido em cinco etapas principais, são elas:

- **Configuração do ambiente**: Trata-se do primeiro passo da estratégia, o qual é responsável por inicializar o ambiente informando a quantidade de PVs que executarão as tarefas produzidas pela aplicação. A configuração do ambiente de execução dá-se a partir da primitiva `void init(int argc, char **argv)`, a qual coleta dados provenientes da linha de comando e configura o ambiente para executar a aplicação paralela. Essa primitiva é chamada logo após a `main` e antes da criação e distribuição dos *threads*. O dado informado a ela e que corresponde à configuração do ambiente

consiste no número de processadores virtuais (`-v (int)`). Este argumento é opcional e, caso o usuário não especificar nenhum valor, a aplicação será executada com apenas um um processador virtual.

- **Aquisição da topologia**: Neste passo, a quantidade de processadores presentes na máquina, as latências de acesso aos blocos de memória distribuídos na arquitetura e outras informações provenientes da topologia da máquina são lidos por meio de um arquivo de texto. Este arquivo é gerado pela ferramenta *HwLoc*, apresentada na Seção 2.4.1. No momento da compilação do ambiente, esse arquivo é lido em um diretório específico dentro dos diretórios de instalação. Caso este arquivo não seja encontrado dentro do diretório específico, o ambiente assume que se trata de uma arquitetura SMP, com os tempos de acesso à memória simétricos, neste caso, não é criada uma lista de prioridades para roubo e estes acontecem de maneira aleatória, conforme previsto no modelo original.

- **Criação das prioridades de roubo**: Uma vez atribuída a afinidade de cada PV e conhecidas as latências de acessos à memória partindo de cada processador presente na arquitetura, é criada uma lista de prioridade de roubo para cada um dos PVs. Uma vez que se conhece as latências entre todos os processadores físicos presentes na arquitetura da máquina, com base na afinidade de cada PV a um destes processadores, é possível que cada PV mantenha a sua lista de prioridades para roubo.

- **Criação e distribuição dos PVs**: Após lida a topologia da máquina, os processadores virtuais são criados. No momento da criação, é definido um processador físico no qual esse PV irá executar, ou seja, é definida uma afinidade para este PV (KAZEMPOUR; FEDOROVA; ALAGHEBAND, 2008). Uma vez que esta máscara é atribuída a um *thread*, todos em sua hierarquia a terão. Desta forma, todos os *threads* criados a partir deste *thread* executarão sobre o mesmo processador, priorizando assim a localidade de dados na memória *cache*. Conforme visto na Seção 4.4, esta distribuição segue uma política circular, o que busca manter balanceados os recursos de processamento. Para atribuir a afinidade dos PVs, utiliza-se as primitivas segundo o padrão *PThreads* (NICHOLS; BUTTLAR; FARRELL, 1996).

- **Realização do escalonamento**: Neste ponto o escalonamento começa a ser realizado. Os PVs consomem as tarefas produzidas pela aplicação paralela em execução e, quando um deles estiver sem tarefas em sua lista local, procura um PV para roubar, com base na lista de prioridades criada no passo anterior. O procedimento de escalonamento ocorre da seguinte

maneira: inicialmente, o PV busca uma tarefa em sua lista local e, caso encontre, começa a executá-la. Caso contrário, o PV busca uma tarefa na lista de outros PVs e, quando encontrar um PV que contenha tarefas prontas para serem executadas em sua lista, rouba a tarefa que for mais antiga, ou seja, a tarefa que estiver no final da fila. O motivo pelo qual um PV rouba a tarefa mais antiga criada na fila do VP vítima está relacionado à heurística que considera que as tarefas mais próximas ao início da execução possuem maior probabilidade de gerar maiores cargas de trabalho, o que irá manter o PV ladrão ocupado por maior tempo. A ideia é tentar roubar, primeiramente, dos PVs que estão fisicamente mais próximos, o que reduz o tempo gasto com a operação de roubo. Se o programa tiver uma estrutura aninhada de paralelismo, é o caso extremo que ilustra bem o caso de sucesso desta estratégia, pois, o último *thread* a ser criado é o próximo a ser sincronizado, o que justifica um PV executar o *thread* mais recente criado em sua lista local.

O detalhamento de cada uma destas etapas, bem como a apresentação dos códigos implementados encontra-se no Apêndice A deste volume.

4.7 Considerações sobre o capítulo

Este capítulo apresentou a modelagem da estratégia de escalonamento proposta neste trabalho, ou seja, mostrou como acontece a execução de uma aplicação paralela e que essa aplicação, ao final da execução, pode ser representada por um DCG. Mostrou também as principais características das arquiteturas NUMA, características estas que devem ser consideradas na concepção de uma estratégia de escalonamento, principalmente as latências nos acessos aos dados.

Por fim, apresentou o modelo de escalonamento, o qual é realizado em dois níveis. Desta estratégia, destaca-se a alteração, em relação a proposta original de Anahy, da heurística de seleção do PV a ser roubado. O modelo original propõe que o PV seja selecionado de forma randômica, o que é adequado quando a natureza da aplicação é recursiva e aninhada, como na implementação realizada em Cilk, e considerando uma arquitetura multiprocessada homogênea. No entanto, ambas características são relaxadas na modelagem da estratégia proposta.

Embora o programador, com conhecimento da estratégia de escalonamento, possa organizar seu código de forma a expressar relações de dependências entre *threads* refletindo as heurísticas de escalonamento aplicadas, existe a liberdade de construir estruturas concorrentes não uniformes. O modelo de progra-

mação de Anahy prevê que *threads* sejam identificados individualmente, então um *thread* pode ser sincronizado por qualquer outro *thread* que tenha acesso a seu identificador.

Adicionalmente, um mecanismo de controle de referências permite que um mesmo *thread* seja sincronizado por mais de uma vez. Como consequência, a relação de custos dos *threads* não prevê que, sistematicamente, o *thread* mais antigo embuta maior quantidade de trabalho, podendo haver desbalanceamento consequentemente.

Outra perspectiva da corrente proposta é que a execução se dá em uma arquitetura NUMA, onde os custos de acesso à memória influenciam no tempo total de execução. A seleção passa a incluir uma variável de prioridade na escolha dos PVs vítimas em função dos tempos de latências ao acesso à memória.

No próximo capítulo é apresentada uma avaliação de Anahy-N, onde foram realizados uma série de experimentos com diversas aplicações. Há também uma comparação entre a versão original de Anahy e a versão voltada para as arquiteturas assimétricas Anahy-N.

5 EXPERIMENTAÇÃO

Este capítulo apresenta uma avaliação experimental da estratégia de escalonamento proposta. Inicialmente, na Seção 5.1 é apresentada a metodologia adotada para a realização dos experimentos. Na Seção 5.2 são apresentadas as plataformas computacionais onde foram executados os experimentos e na Seção 5.3 as aplicações que compõem os casos de estudo. Na Seção 5.4, são apresentados os resultados de desempenho obtidos pelo escalonamento proposto e por outras ferramentas de programação *multithread* que dispõem de mecanismos de escalonamento em nível aplicativo. Por fim, a Seção 5.5 avalia o impacto de desempenho da adoção da nova proposta de escalonamento em ambiente NUMA, em relação ao mecanismo original de Anahy.

5.1 Metodologia adotada

Os estudos de caso foram realizados em máquinas dedicadas aos experimentos, sem outros usuários ou processos aplicativos em paralelo. O tempo anotado, em segundos, corresponde ao tempo decorrido entre o início e o final da porção paralela dos programas, não sendo consideradas as porções sequenciais de inicialização e término dos programas. Todas as aplicações foram compiladas com o compilador `icpc` (Intel C++ Composer XE 2013 Compiler).

Para mostrar os dados oriundos dos experimentos optou-se por utilizar o gráfico *Box Plot* (também conhecido como gráfico de caixas). Esse tipo de gráfico, criado por John Tukey (1977), tornou-se um método bastante eficiente para mostrar os valores que sumarizam qualquer conjunto de dados. Ele resume as seguintes medidas estatísticas: (i) mediana, (ii) quartis superior (`Q3`) e inferior (`Q1`), (iii) valores máximos e mínimos não discrepantes e (iv) valores discrepantes superiores e inferiores. Nos resultados apresentados, cada gráfico de caixa contém dados de 30 execuções para cada estudo de caso.

Para auxiliar a compreensão deste recurso para apresentação dos resultados de desempenho, a Figura 18 ilustra a distribuição das medidas estatísticas

no gráfico *Box Plot*. A caixa (*box*) contém a metade (50%) dos dados da amostra. O limite superior da caixa indica o 3° quartil (percentil de 75% do total de dados da amostra) e o limite inferior da caixa indica o 1° quartil (percentil de 25% dos dados). A distância entre esses dois quartis é conhecida como intervalo interquartil – `IQR` (do inglês, *InterQuartile Range*).

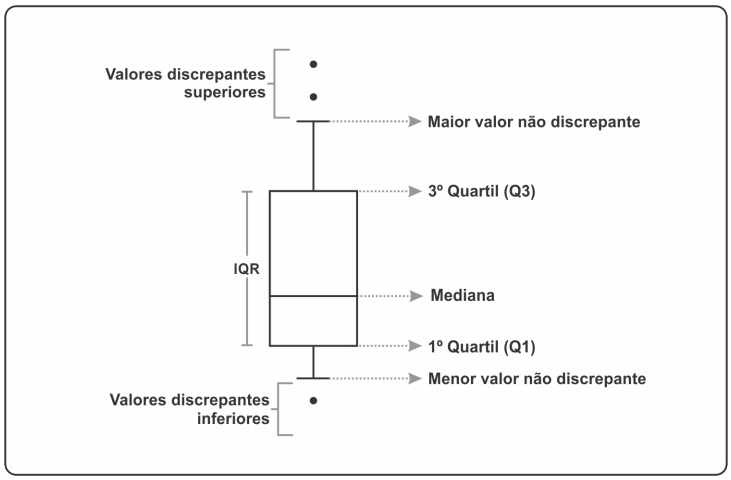


Figura 18: Dados estatísticos presentes no gráfico *Box Plot*.

A linha na caixa indica o valor da mediana dos dados amostrais. Se a linha da mediana dentro da caixa não for equidistante dos extremos, diz-se então que os dados são assimétricos. Por meio dos bigodes (também conhecidos como *whiskers*) são representados os maiores e menores valores não discrepantes. Os bigodes se estendem até o máximo de `1.5` vezes da distância `IQR`, ou seja, quaisquer valores acima de `Q3` ou abaixo de `Q1` por mais de `1.5 * IQR` são considerados valores atípicos. O valores fora desse intervalo são, então, os valores discrepantes (*outliers*).

A análise de um grafo de caixa permite identificar a qualidade dos resultados obtidos. Por exemplo, na Figura 18 verificamos que os dados são assimétricos, uma vez que a distância entre a mediana e o `Q3` é maior que a distância entre a mediana e `Q1`, tendo em vista que a quantidade de valores em cada segmento é o mesmo. Também observamos a existência de bigodes e três amostras discrepantes. Quanto menores forem os bigodes e quanto menos amostras discrepantes forem observadas, mais estável podem ser considerados os resultados. Outro elemento visual que auxilia na identificação da estabilidade dos dados é o tamanho da caixa, quanto menor a caixa, mais estável são os dados amostrados.

Ao analisar os gráficos, é importante que o leitor observe que, nos valores obtidos com a estratégia proposta neste trabalho, o valor da mediana encontra-se, na maioria das vezes, dentro do 3° quartil (`Q3`) das demais ferramentas. Consideramos, nestes casos, que os resultados obtidos com a estratégia proposta são equivalentes aos resultados das demais ferramentas.

5.2 Arquiteturas de testes

Os experimentos foram realizados em duas arquiteturas NUMA, com oito e 64 *cores*, respectivamente. Esta seção se propõe a caracterizar essas máquinas, descrevendo suas principais características e apresentando suas topologias.

5.2.1 Arquitetura GoodTwin

A máquina **GoodTwin**, nome pelo qual esta máquina será referenciada deste ponto em diante, é composta por dois nós de processamento Intel® *Quadcore* Xeon® E5620 de 2,66GHz, porém conta com três níveis de *cache*. A Figura 19 ilustra a topologia da máquina GoodTwin.

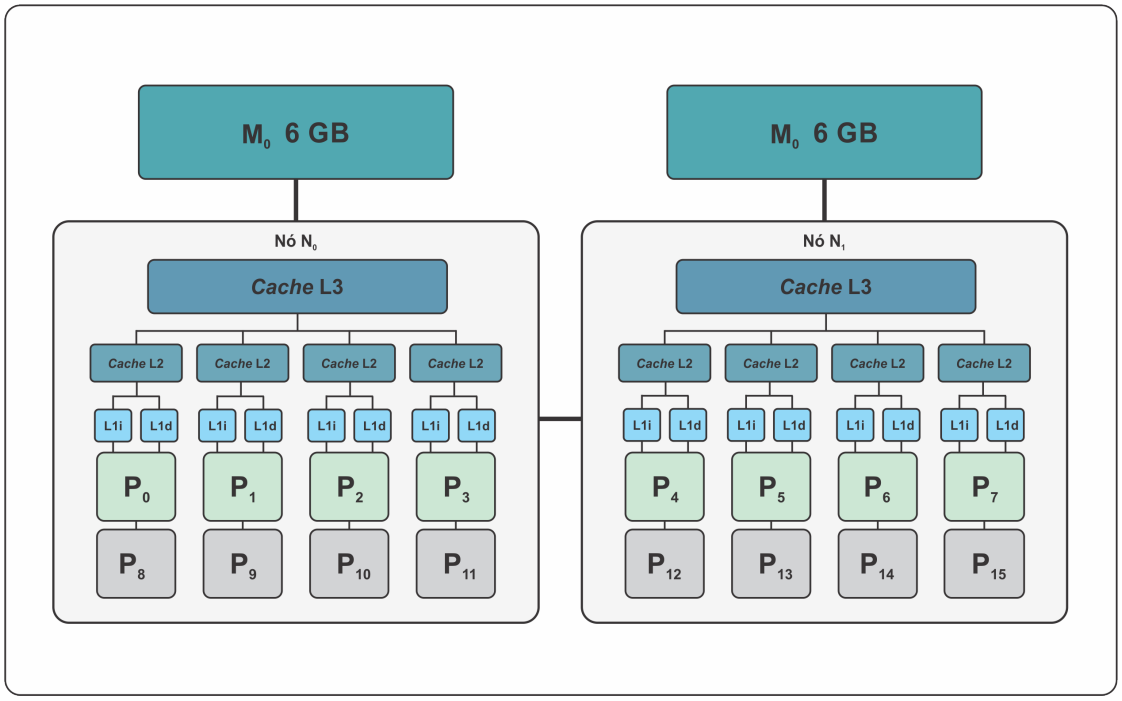


Figura 19: Topologia da máquina GoodTwin.

Cada nó de processamento desta máquina é composto por 4 *cores* físicos com possibilidade de ativar *Hyper-threading* (*cores* em cinza, quando o *Hyper-threading* encontra-se ativado) e possui um total de 12GB de memória principal. Esta memória é dividida em dois blocos (6GB de memória em cada nó). Quanto aos níveis de *cache*, a GoodTwin apresenta 12MB de *L3*, 256KB de *L2* e 32KB de *L1*, sendo *L1* e *L2* únicos para cada processador e *L3* compartilhado entre todos os processadores do nó. O sistema operacional é o Ubuntu Server 12.4 LTS de 64 *bits*.

5.2.2 Arquitetura Hydra

A segunda arquitetura utilizada neste trabalho é composta por quatro nós de processamento AMD® Opteron 6276 *series* com 2,3GHz e três níveis de *cache* (*L3* de 6MB, *L2* de 2MB e *L1* de 64KB). Cada nó de processamento contém 16 *cores*, somando um total de 64. Esta máquina possui 128GB de memória principal, sendo 32GB por nó separados em blocos de 16GB. A Figura 20 ilustra a topologia desta máquina, **Hydra**, como será referenciada.

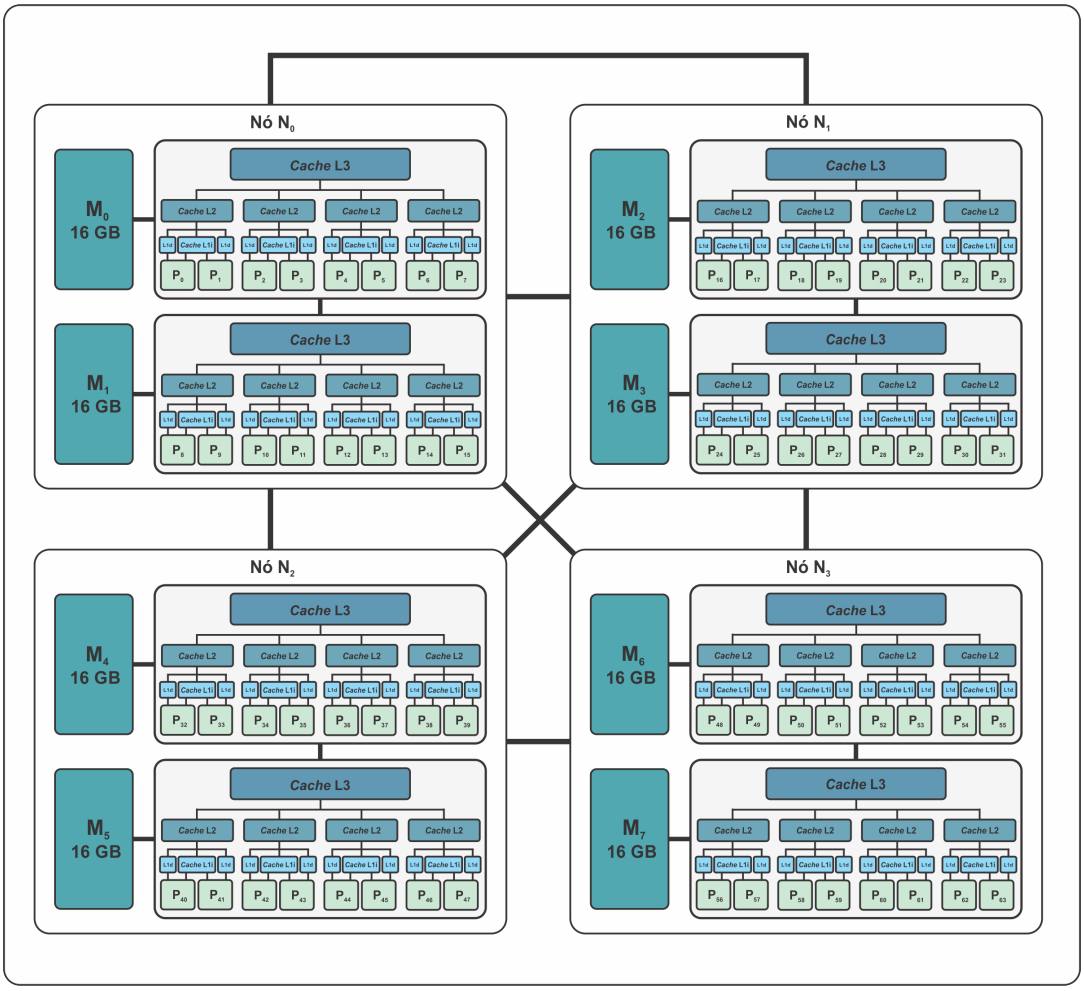


Figura 20: Topologia da máquina Hydra

Como pode-se observar na topologia apresentada na Figura 20, esta máquina possui *cache L1* separada para dados e instruções, sendo a *L1* de dados única para cada processador e a *L1* de instruções compartilhada entre dois processadores. Esta máquina, da mesma forma que a GoodTwin, possui o sistema operacional Ubuntu Server 12.04 LTS de 64 *bits*. A seguir, é apresentado um sumário com as principais características de cada uma das arquiteturas.

5.2.3 Sumário das arquiteturas

Após apresentadas as duas arquiteturas que compõem os experimentos, com a finalidade de facilitar a visualização e mostrar as diferenças entre suas características, a Tabela 2 contém um resumo das principais configurações de cada uma delas.

Tabela 2: Principais características das máquinas de testes.

	GoodTwin	**Hydra**
Processador	Intel® *Quadcore* Xeon®E5620 2,66GHz	AMD® Opteron 6276 2,3GHz
Nro. Processadores	2	4
Hyper-threading	Sim	Não
Nro. núcleos físicos	8	64
Nro. núcleos lógicos	8	–
Mem. Principal	12GB	128GB
Mem./processador	6GB	32GB
Tam. *Cache L1*	32KB	64KB
Tam. *Cache L2*	256KB	2MB
Tam. *Cache L3*	12MB	6MB
Sistema operacional	Ubuntu Server 12.04 64*bits*	Ubuntu Server 12.04 64*bits*

Procuramos utilizar o mesmo sistema operacional nas duas plataformas, a máquina *GoodTwin* possui *Hyper-threading*, a qual será avaliada tanto com *Hyper-threading* ativado quanto desativado. Com estas plataformas tem-se um leque que abrange diferentes possibilidades de uso, podendo inclusive ser verificada a escalabilidade em termos de exploração do paralelismo (uso dos processadores).

5.3 Aplicações teste

Foram realizados testes de desempenho com quatro aplicações: (i) alinhamento genético com o algoritmo de *Smith-Waterman*, (ii) algoritmo de ordenação *Bucket Sort*, (iii) fatoração LU de matrizes e o (iv) cálculo do n-ésimo termo da sequência de *Fibonacci*. Cada uma dessas aplicações é detalhada a seguir, onde são apresentadas as suas características e seus grafos de execução. Todos os experimentos, para todas a ferramentas, foram realizados com os mesmos conjuntos de dados de entrada.

5.3.1 Alinhamento genético com Smith-Waterman

O Algoritmo de *Smith-Waterman* (SMITH; WATERMAN, 1981) é utilizado para realizar alinhamentos de sequências genéticas, isto é, é elaborado para determinar as regiões similares entre duas sequências de nucleotídeos ou proteínas. Este algoritmo compara segmentos de todos os comprimentos possíveis e otimiza a medida de similaridade.

O algoritmo faz uso de uma estratégia de programação dinâmica, na qual uma matriz M de tamanho $n * m$ guarda informações dos melhores sub-alinhamentos das sequências genéticas de tamanhos n e m. Cada elemento $M(i, j)$ desta matriz é calculado a partir das seguintes equações recursivas:

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

O melhor alinhamento é obtido à partir do maior valor da matriz M, este valor representa o *score*, ou seja, o melhor alinhamento possível das duas sequências comparadas. Pode-se rastrear a origem deste valor (se veio da célula da esquerda, de cima ou da diagonal superior esquerda) e, com o trajeto inverso, imprimir o melhor alinhamento entre as sequências. A Figura 21 mostra o grafo de dependências desse algoritmo.

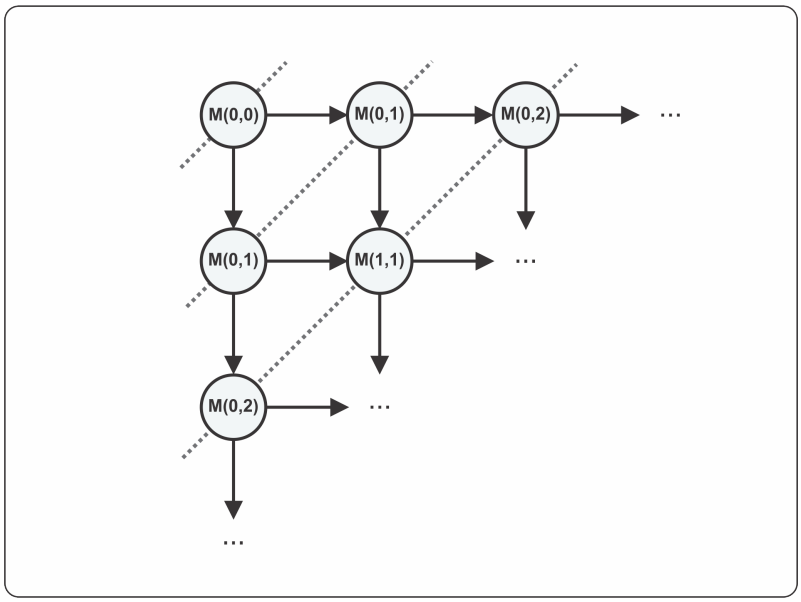


Figura 21: Grafo de dependências da matriz do algoritmo de Smith-Waterman.

Embora o valor da célula $M(i, j)$ dependa de toda linha i à esquerda e de toda

a coluna j acima da célula, é possível eliminar a dependências transitivas e criar um grafo de tarefas para calcular a matriz de modo que a célula $M(i, j)$ dependa somente das células $M(i-1, j)$ e $M(i, j-1)$, como mostrado na Figura 21. Este arranjo de dependências permite o cálculo dos elementos das antidiagonais da matriz M indicadas pelas linhas tracejadas. Este cálculo é realizado de maneira paralela a partir da estratégia chamada *wavefront*.

Uma vez que realizar o cálculo de cada elemento da matriz como uma tarefa independente gera uma granularidade muito fina e, assim, muito *overhead* no ambiente de execução, aplica-se a mesma estratégia *wavefront* em nível de bloco de elementos da matriz. Para fazer os experimentos, foi utilizada uma matriz de sequências de tamanho 2.000, com blocos de tamanho 50x50.

5.3.2 Ordenação com Bucket Sort

O algoritmo *Bucket Sort* (também chamado de *Bin Sort*) é um algoritmo de ordenação que particiona os elementos de um vetor em elementos denominados *buckets* (baldes), e ordena cada um destes *buckets* com algum outro método de ordenação ou usando o *Bucket Sort* recursivamente.

Cada *bucket* é um *conteiner* de inteiros que contém números de um intervalo do vetor original. O intervalo de cada *bucket* é estabelecido assumindo-se as seguintes condições:

- O vetor a ser ordenado possui tamanho n e seus elementos contidos no intervalo $[0, n)$;
- Para um número b de *buckets*, cada *bucket* armazenará elementos de um sub-intervalo de $[0, n)$ de tamanho n/b.

Para exemplificar, vamos considerar um vetor com 16 elementos e 4 buckets. Para a ordenação podemos ter a seguinte configuração:

- Vetor a ser ordenado: $[3, 9, 1, 12, 6, 6, 0, 11, 14, 8, 11, 7, 10, 8, 6, 4]$;
- Conteúdo do *bucket*(0): $[3, 1, 0]$;
- Conteúdo do *bucket*(1): $[6, 6, 7, 6, 4]$;
- Conteúdo do *bucket*(2): $[9, 11, 8, 11, 10, 8]$;
- Conteúdo do *bucket*(3): $[12, 14]$.

Neste trabalho, o vetor a ser ordenado é gerado por um outro programa e possui 10.000.000 de elementos. Este vetor contém tanto elementos repetidos quanto elementos ausentes, ainda assim mantendo o número n de elementos,

todos contidos no intervalo $[0, n)$. Desta forma, os *buckets* que serão ordenados e colocados de volta no vetor original de forma paralela possuem tamanhos diferentes, ou seja, cargas de trabalho diferentes para cada *thread* que ordenará um determinado *bucket*.

O algoritmo usado para ordenar cada *bucket* foi o *qsort*, implementação do *Quick Sort* da biblioteca padrão de *C/C++ stdlib*. A Figura 22 ilustra o grafo de dependências do algoritmo Bucket-Sort.

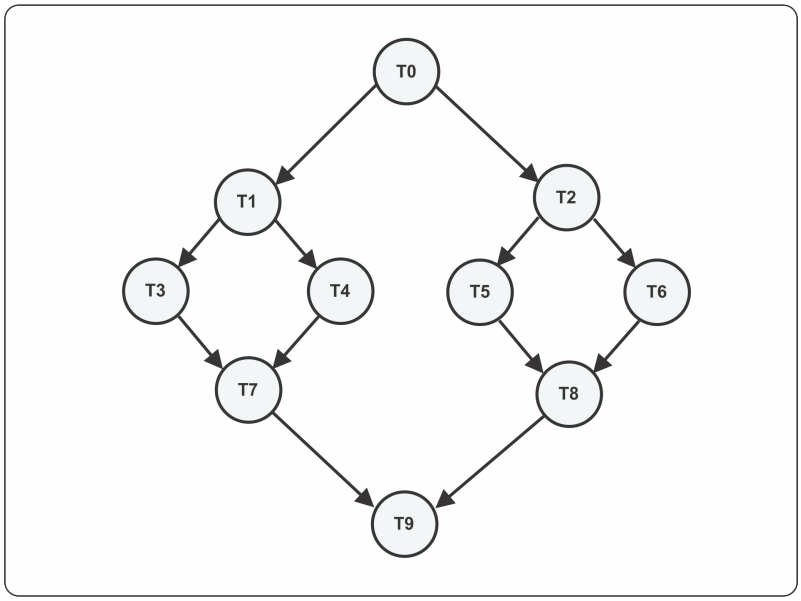


Figura 22: Grafo de dependências do algoritmo Bucket-Sort.

5.3.3 Fatoração LU de matrizes

Em álgebra linear, a fatoração LU (em que LU vem do inglês *lower* e *upper*) é uma forma de fatoração de uma matriz não singular como o produto de uma matriz triangular inferior (*lower*) e uma matriz triangular superior (*upper*). Esta decomposição é utilizada em análise numérica para resolver sistemas de equações ou encontrar matrizes inversas.

A fatoração LU de uma determinada matriz surge da necessidade de resolver um conjunto de sistemas do tipo $Mx = b1, Mx = b2, Mx = b3$ e assim por diante. Pode-se resolver cada um desses sistemas escalonando a matriz completa $[Mb]$ correspondente até sua forma reduzida mas, computacionalmente, o escalonamento é um processo muito demorado e, em determinados casos, é possível haver centenas ou milhares de bs para resolver.

A fatoração LU surge para resolver esse problema. Basicamente, supondo-se que M seja uma matriz de tamanho $m \times n$ que pode ser escalonada até sua forma reduzida sem trocar linhas, a Álgebra Linear nos diz que ela pode ser escrita na forma $M = L \times U$, onde L é a matriz triangular inferior $m \times n$ cuja

diagonal principal possui apenas 1s e U é a matriz triangular superior também de tamanho $m \times n$.

A importância de escrevermos uma matriz como o produto de duas outras matrizes triangulares, uma inferior e a outra superior, se dá pelo fato de que, ao assumir que $M = L * U$, pode-se reescrever a equação $Mx = b$ como $LUx = b$ ou, pela propriedade associativa, $L(Ux) = b$. Tomando-se $Ux = y$, tem-se $Ly = b$ e $Ux = y$. Resolvendo-se a primeira equação para y e a segunda para x, obtêm-se a resposta procurada de uma maneira bem mais simples do que fazer pelo método tradicional, visto que as matrizes são triangulares.

Para calcular as matrizes é necessário escalonar a matriz M com operações de substituição de linhas. A matriz resultante será a matriz U. A matriz L é composta pelas colunas pivô de cada passo do escalonamento, com os elementos de cada coluna divididos pelo pivô correspondente. A Figura 23 mostra parte do grafo de dependências da fatoração LU de uma matriz M qualquer.

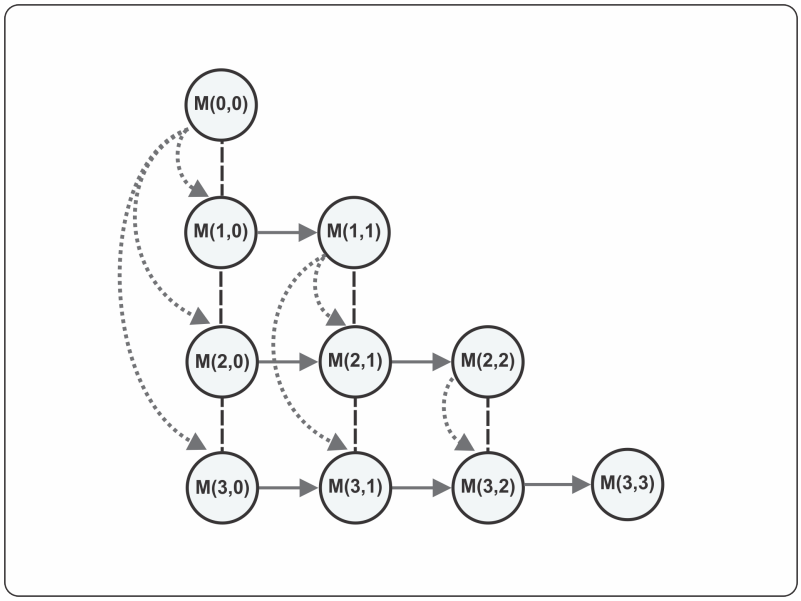


Figura 23: Grafo de dependências da Fatoração LU de matrizes.

Para exemplificar, vamos ver como fica a matriz M de tamanho 3×3 escrita na sua forma fatorada ($M = L \times U$):

$$M = \begin{bmatrix} 3 & -1 & 2 \\ -3 & -2 & 10 \\ 9 & -5 & 6 \end{bmatrix}$$

Como a matriz M possui três linhas, a matriz L possuirá o tamanho 3×3. A matriz M não precisa, necessariamente, ser quadrada. Realizando o escalonamento de M pelas operações elementares, obtêm-se:

$$M = \begin{bmatrix} 3 & -1 & 2 \\ -3 & -2 & 10 \\ 9 & -5 & 6 \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} 3 & -1 & 2 \\ 0 & -3 & 12 \\ 0 & -2 & 0 \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} 3 & -1 & 2 \\ 0 & -3 & 12 \\ 0 & 0 & -8 \end{bmatrix}$$

Neste momento, sabe-se que a última matriz obtida é a matriz U, ou seja,

$$U = \begin{bmatrix} 3 & -1 & 2 \\ 0 & -3 & 12 \\ 0 & 0 & -8 \end{bmatrix}$$

. Pode-se perceber que U é, de fato, uma matriz triangular superior, todos os elementos abaixo da diagonal principal estão zerados. Sabe-se que L é a matriz triangular inferior, os elementos de suas colunas, então, serão os elementos das colunas pivô de cada estágio do escalonamento de M, abaixo do pivô daquela coluna, divididos pelo mesmo, a fim de formar a diagonal principal só de 1s.

A primeira coluna da matriz L será composta pelos elementos da primeira coluna de M divididos pelo pivô que, neste caso, é 3. Assim, como a primeira coluna de M é $(3, -3, 9)$, a primeira coluna de L será $(1, -1, 3)$, pois esses são os elementos da primeira coluna de M divididos pelo pivô 3. O pivô da segunda coluna é -3 e, abaixo dele, está o -2. Dividindo esses elementos por -3 obtêm-se $(0, 1, \frac{2}{3})$. O pivô da terceira coluna é -8, sem elementos abaixo dele. Dividindo, obtêm-se a terceira coluna, $(0, 0.1)$. Dessa forma, a matriz L será:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 3 & \frac{2}{3} & 1 \end{bmatrix}$$

Dessa forma obtemos a fatoração da matriz M, ou seja, as matrizes triangulares L e U. Se multiplicarmos essas duas matrizes, obteremos a matriz M novamente.

5.3.4 Sequência de Fibonacci

A sequência de *Fibonacci* é uma sequência de números naturais, na qual os primeiros dois termos são 0 e 1 e cada termo subsequente corresponde à soma dos dois termos precedentes. A sequência tem o nome do matemático pisano do século XIII Leonardo de Pisa, conhecido como Leonardo Fibonacci, e os termos da sequência são chamados números de *Fibonacci*. A sequência de *Fibonacci* é definida recursivamente, como mostrado a seguir:

$$f(n) = \begin{cases} 0 & \text{Se } n = 0 \\ 1 & \text{Se } n = 1 \\ f(n-1) + f(n-2) & \text{Se } n \geqslant 2 \end{cases}$$

A paralelização do algoritmo é uma tarefa trivial, uma vez que a chamada recursiva das funções f , $f(n-1)$ e $f(n-2)$ podem ser realizadas concorrentemente por um *thread* independente. Caso não seja imposto um limite na criação de tarefas paralelas, um número significativamente grande de *threads* pode coexistir durante a execução para realizar computações de finíssima granularidade.

Por exemplo, $f(40)$ faz com que a função f seja chamada 331.160.280 vezes, sendo que metade destas chamadas executam um teste e retornam um resultado (0 ou 1). As demais chamadas executam o mesmo teste, criam outros dois *threads* e ficam esperando seus resultados para realizar a soma de seus números precedentes.

A escolha desta aplicação para os experimentos deu-se com o intuito de analisar o alto grau de paralelismo existente em sua estrutura regular. Durante a execução do programa a expansão dos nodos do grafo gera um desbalanceamento à esquerda da árvore de execução, conforme pode-se observar na Figura 24. A Figura 24 exibe o grafo simplificado da execução de um algoritmo *multithread* para o calculo do 4° número da sequência de *Fibonacci*.

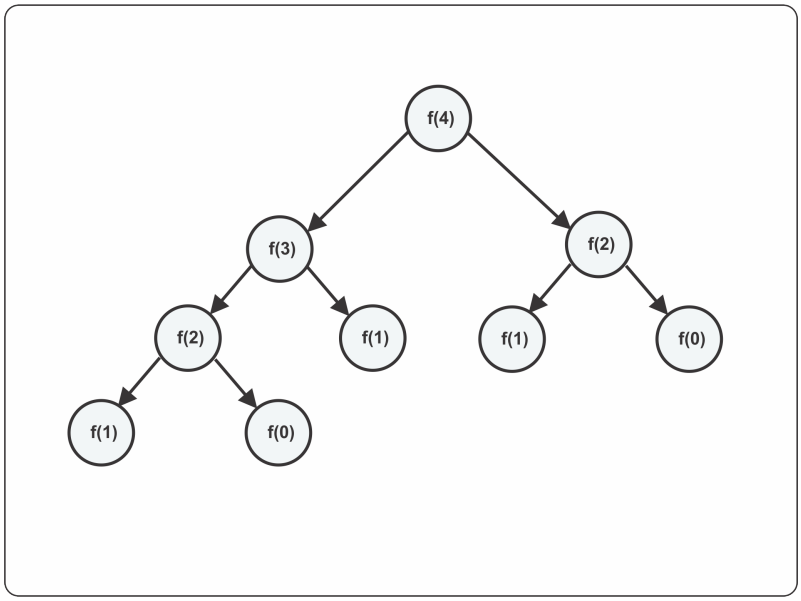


Figura 24: Grafo de dependências do algoritmo de Fibonacci

5.3.5 Sumário das aplicações

Procuramos utilizar aplicações com diferentes estruturas de grafos porém, todas apresentam grafos regulares e geram diferentes níveis de paralelismo. A

aplicação *Smith-Waterman* apresenta um modelo de programação dinâmica com altas dependências. A implementação do método de ordenação *Bucket Sort* é de paralelização trivial e apresenta compartilhamento de dados em leitura. A fatoração LU de matrizes, possui granularidade fina e assim como *Smith-Waterman*, apresenta um alto grau de dependências. *Fibonacci* é uma aplicação geradora de uma grande quantidade de atividades concorrentes de forma recursiva, também com granularidade fina.

Um aspecto a ressaltar destas aplicações é que, embora elas descrevam grafos regulares, a distribuição da carga computacional nos diferentes ramos não é constante. No caso do algoritmo de *Smith-Waterman* o custo de cada tarefa está associado ao comprimento das antidiagonais e, de forma semelhantes na fatoração LU o custo está associado ao comprimento da linha e da coluna associado ao pivô. No caso do *bucket sort*, a irregularidade está associada a natureza aleatória de valores presentes no vetor inicial, que pode formar baldes com diferentes números de elementos. A seguir são apresentados os resultados obtidos com cada uma destas aplicações.

5.4 Estudos de casos

Nesta seção são apresentados os resultados obtidos na realização dos experimentos. Nos gráficos apresentados, o número de *threads* informado representa o número de *threads* sistema (processadores virtuais) com o qual o ambiente de execução das diferentes ferramentas foi instanciado. Este número de *threads* não representa, portanto, o número de *threads* usuário, geradas pelo programa em execução. Ilustra-se que o número de *threads* usuário é bastante superior ao número de *threads* sistema, citando que na aplicação de *Fibonacci* são geradas aproximadamente 1,5 milhões de *threads* no decorrer da execução do programa.

5.4.1 Resultados Smith-Waterman

Os experimentos envolvendo esta aplicação foram realizados com uma matriz de sequências de tamanho 2.000 dividida em blocos de tamanho 50x50. As Figuras 25 e 26 apresentam os gráficos de caixa com os resultados obtidos na execução desta aplicação, na arquitetura GoodTwin com *Hyper-threading* ativado e desativado, respectivamente.

Pode-se observar que, nas execuções com apenas um (1) *thread*, a estratégia proposta neste trabalho obteve um bom desempenho, introduzindo pouco *overhead* em relação à Cilk e TBB. Como foi visto no Capítulo 2, as ferramentas Cilk e TBB possuem um princípio de escalonamento similar, mas o comportamento da aplicação *Smith-Waterman* não é bem representado no modelo de

escalonamento restrito (*Fork*/*Join* aninhado de Cilk).

Até oito *threads*, momento em que se chega ao limite de processadores reais na máquina, a estratégia proposta continua obtendo bons índices de desempenho em relação à Cilk e TBB, com a vantagem de os dados serem mais estáveis, considerando que as caixas no gráfico são mais estreitas. De maneira geral, os resultados obtidos com a estratégia proposta não variaram muito em ambos os casos (com *Hyper-threading* ativado e desativado) enquanto o número de *threads* não ultrapassou o número de *cores* físicos disponíveis. Nos experimentos com 16 e 32 *threads*, o desempenho da estratégia proposta foi decaindo quando o *Hyper-threading* foi desativado, o que é justificado se considerarmos que a estratégia proposta explora a localidade de dados em cada *thread*, logo o *Hyper-threading* ativado contribui no aumento de desempenho.

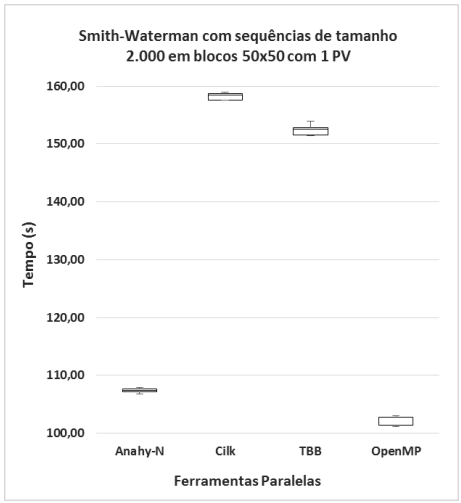
A Figura 27 mostra os gráficos de *speedup* obtidos em relação à execução de *Smith-Waterman* na arquitetura GoodTwin (a) com *Hyper-threading* e (b) sem *Hyper-threading*. O *speedup* foi computado considerando a execução sequencial do problema e a média dos dados apresentados nas figuras 25 e 26.

Analisando os gráficos de *speedup* na Figura 27, observa-se que a estratégia proposta neste trabalho obteve bons índices de desempenho nesta aplicação, sendo suplantada apenas por OpenMP. Pode-se observar, também, que a ferramenta TBB foi a que menos perdeu desempenho na medida que o número de *threads* ultrapassou o número de *cores* físicos da arquitetura. Cilk foi a ferramenta que, de modo geral, obteve o pior desempenho na execução desta aplicação, o que pode ser justificado pelo seu modelo de escalonamento restrito (*Fork*/*Join*).

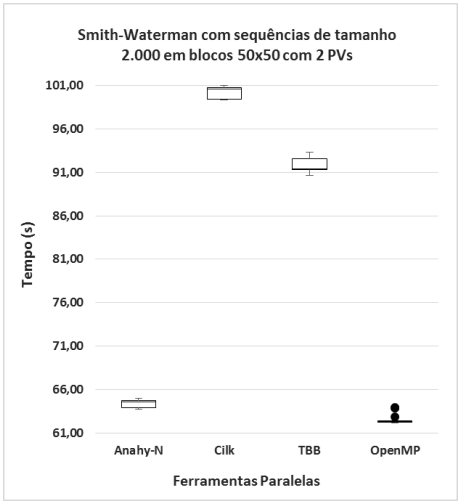
O desempenho desta mesma aplicação na arquitetura Hydra é apresentado nas Figuras 28 e 29 mostrando, respectivamente, os gráficos de caixas e o gráfico de *speedups*. Nesta arquitetura, este estudo de caso se mostrou uma computação que pode ser considerada leve, pois, utilizando todo o seu potencial de execução paralela (ou seja, com um número de *threads* elevado em relação ao número de cores disponíveis), os tempos médios observados para as ferramentas aumentou devido ao *overhead* na gestão da concorrência.

Nos demais casos, a estrutura regular da aplicação, que consiste em laços aninhados, foi mais eficientemente explorado por OpenMP e Cilk. Analisando em conjunto, os resultados obtidos em ambas as arquiteturas, Hydra e GoodTwin, a estratégia proposta neste trabalho obteve desempenhos compatíveis com as demais ferramentas enquanto a carga de trabalho da aplicação estava compatível com a capacidade explorada da arquitetura, na medida em que a aplicação se torna mais leve que o suporte da arquitetura, o *overhead* de execução da versão de Anahy com a estratégia proposta é maior que nas demais ferramentas.

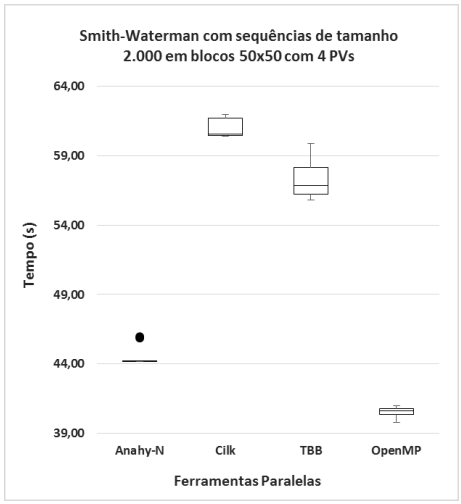
O maior *overhead* em Anahy-N é esperado e pode se explicar em função da



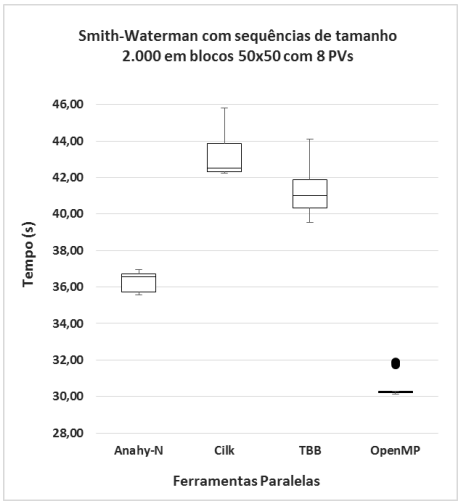
(a) 1 Thread



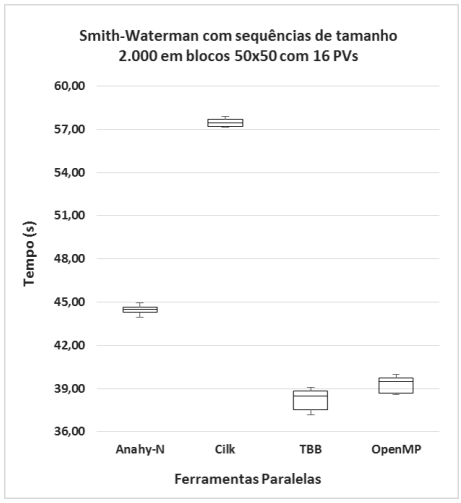
(b) 2 Threads



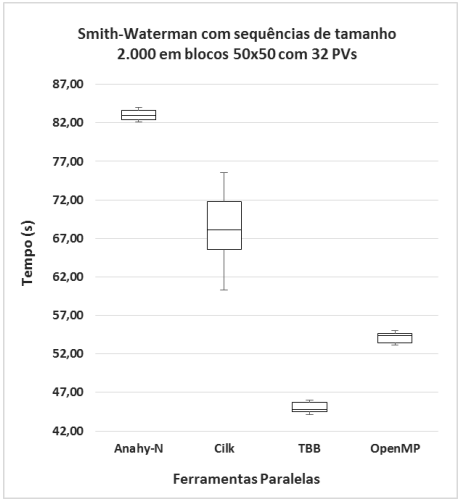
(c) 4 Threads



(d) 8 Threads

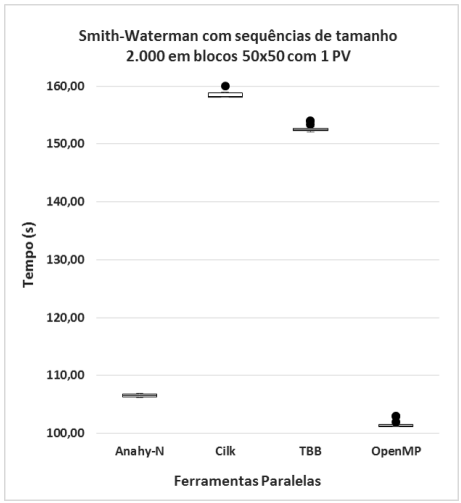


(e) 16 Threads

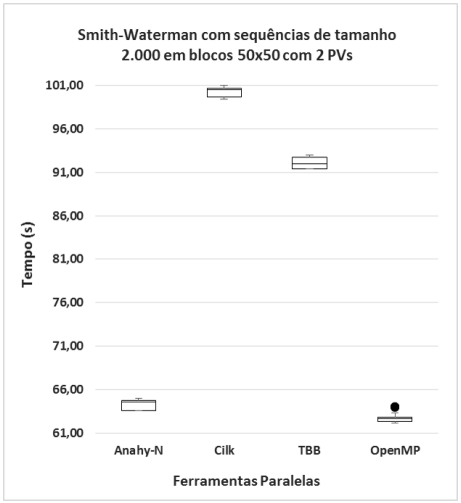


(f) 32 Threads

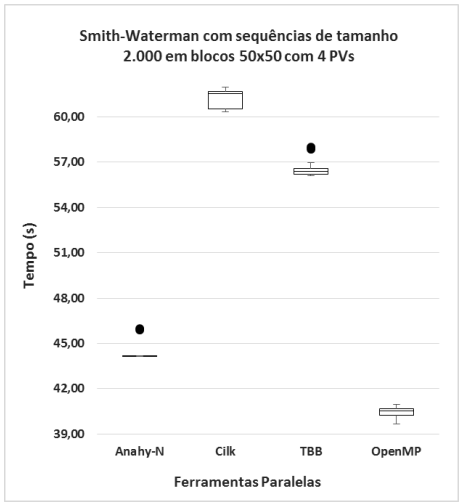
Figura 25: Resultados obtidos com Smith-Waterman em sequências de tamanho 2000 com blocos 50x50 na arquitetura GoodTwin com *Hyper-threading*.



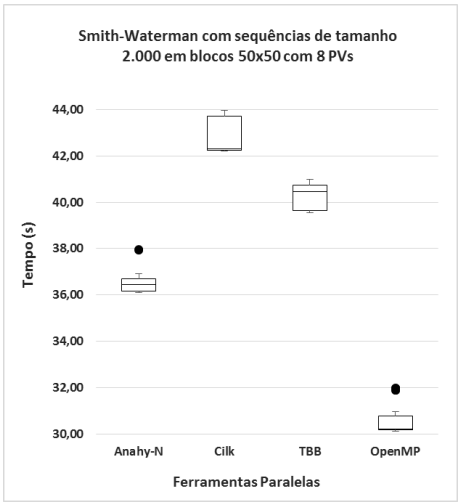
(a) 1 Thread



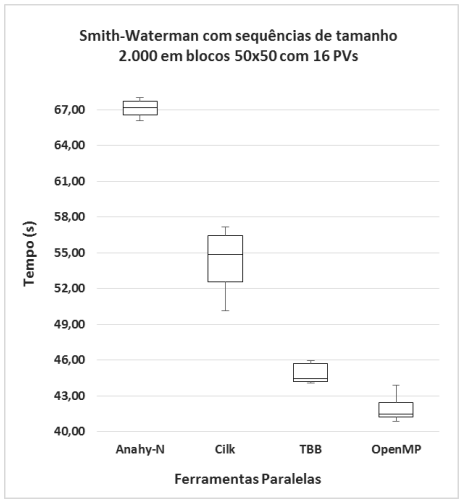
(b) 2 Threads



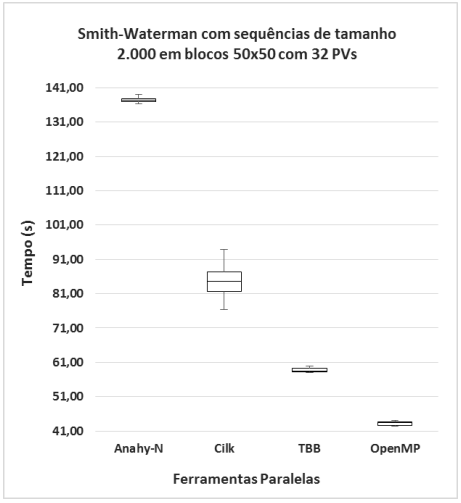
(c) 4 Threads



(d) 8 Threads

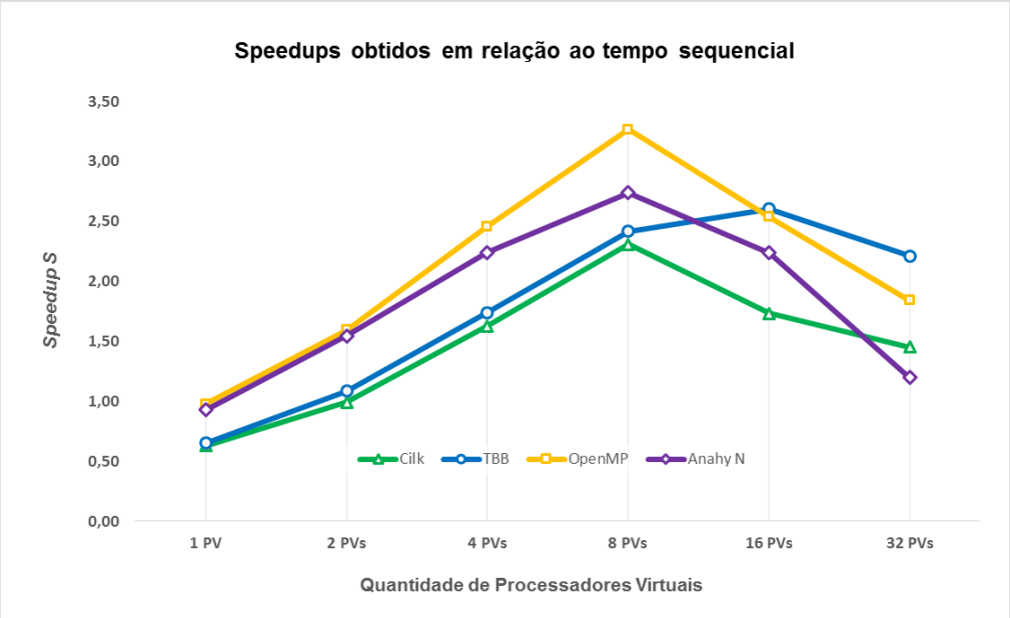


(e) 16 Threads

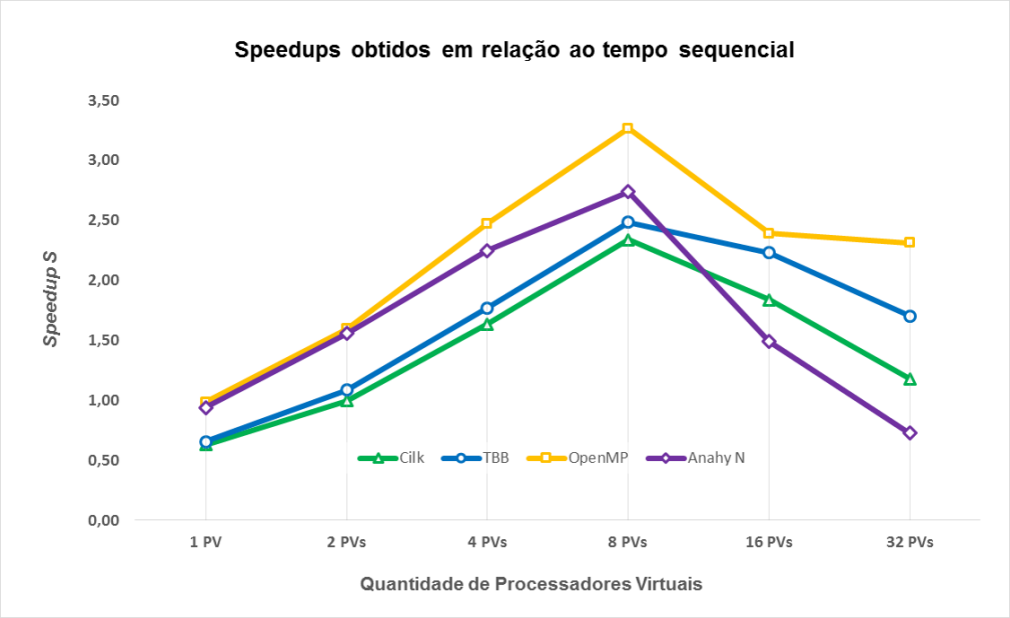


(f) 32 Threads

Figura 26: Resultados obtidos com Smith-Waterman em sequências de tamanho 2000 com blocos 50x50 na arquitetura GoodTwin sem *Hyper-threading*.

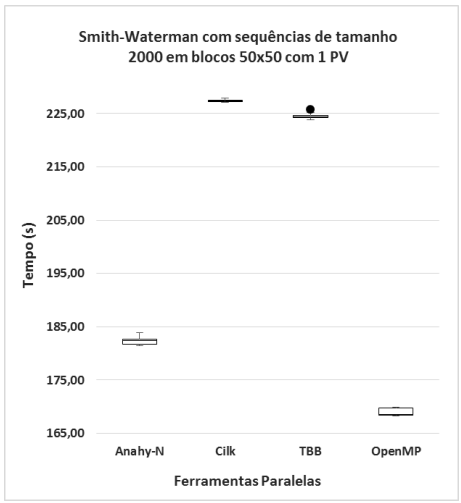


(a) Com Hyper-threading

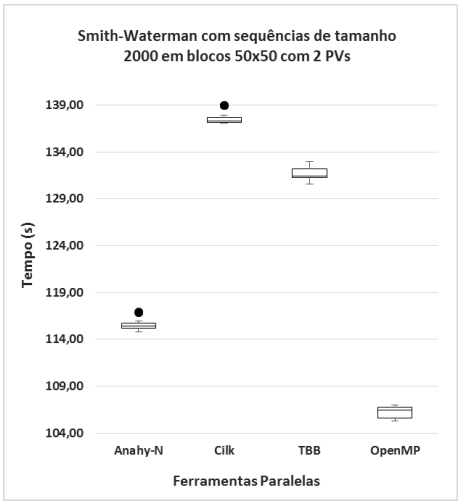


(b) Sem Hyper-threading

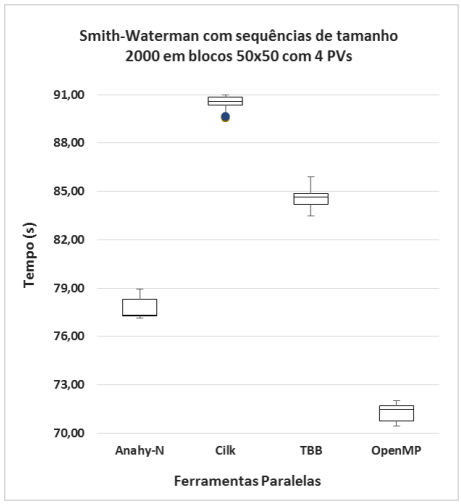
Figura 27: Gráfico dos *Speedups* obtidos com a aplicação Smith-Waterman em relação ao tempo de execução sequencial na arquitetura GoodTwin: (a) com *Hyper-threading* e (b) sem *Hyper-threading*.



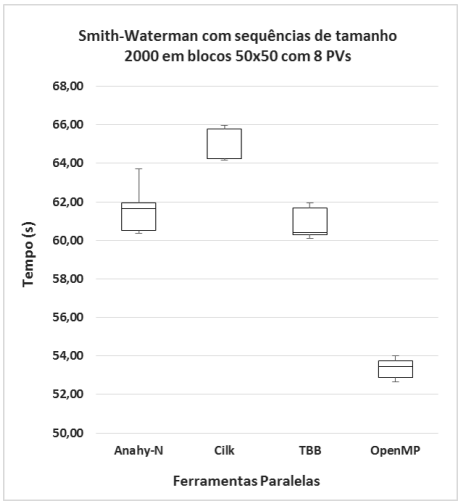
(a) 1 Thread



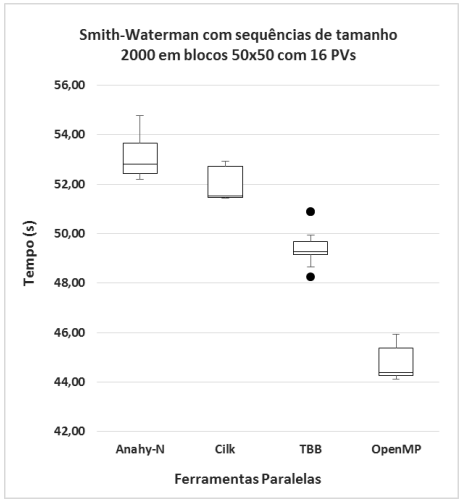
(b) 2 Threads



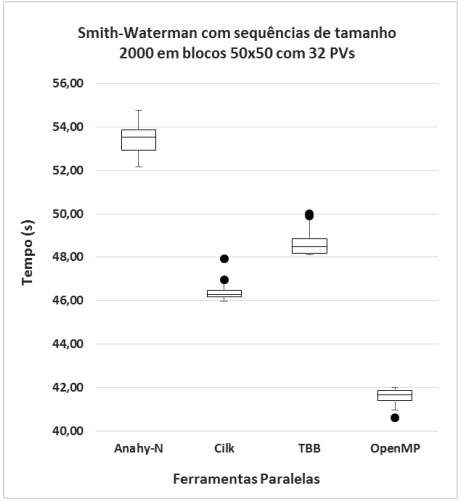
(c) 4 Threads



(d) 8 Threads

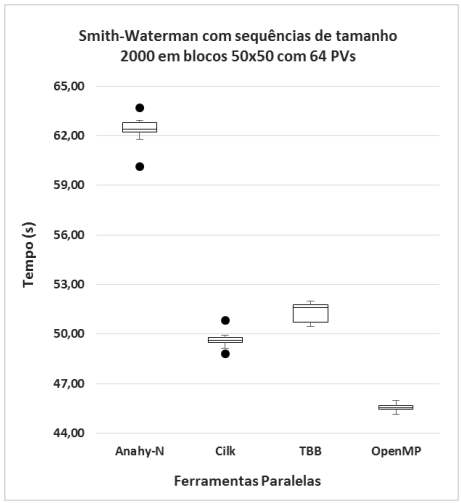


(e) 16 Threads

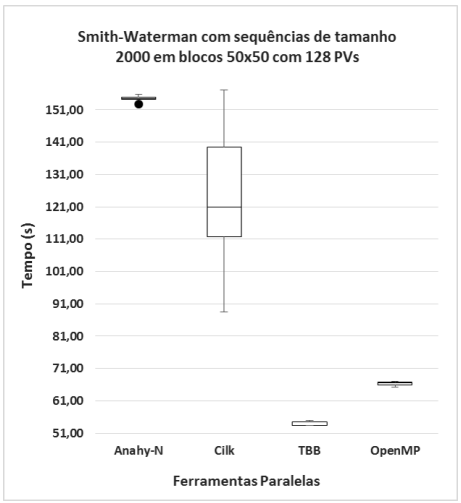


(f) 32 Threads

Figura 28: Resultados obtidos com Smith-Waterman em sequências de tamanho 2000 com blocos 50x50 na arquitetura Hydra. *(cont.)*



(g) 64 Threads



(h) 128 Threads

Figura 28: Resultados obtidos com Smith-Waterman em sequências de tamanho 2000 com blocos 50x50 na arquitetura Hydra.

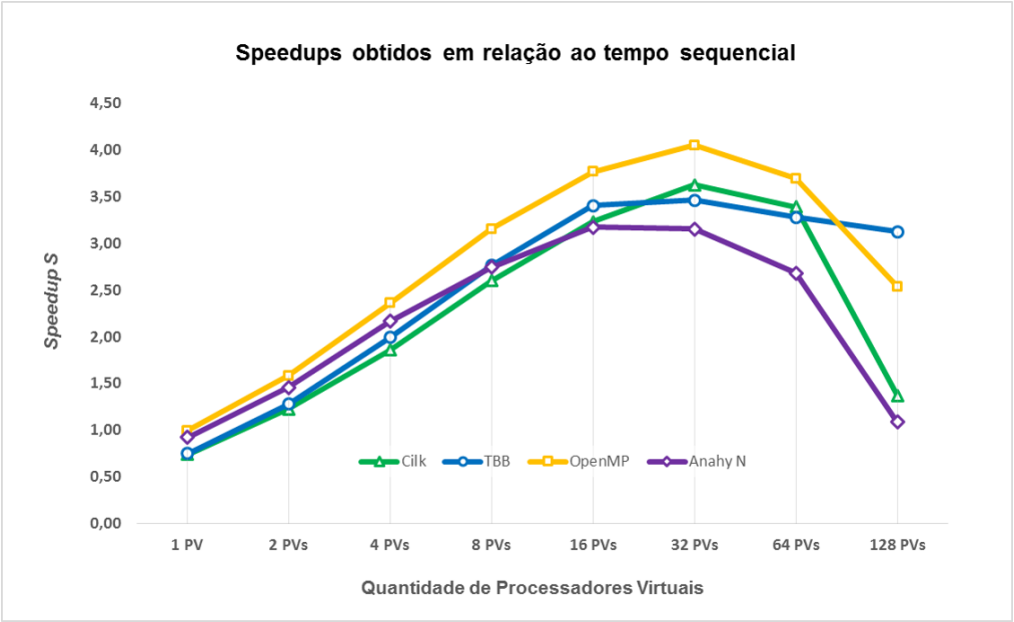


Figura 29: Gráfico dos *Speedups* obtidos com a aplicação Smith-Waterman em relação ao tempo de execução sequencial, na arquitetura Hydra.

grande atividade do escalonador durante a execução de um programa escrito em Anahy-N. O mesmo não ocorre nas outras ferramentas por diversas razões. Em *Cilk*, boa parte das ações de escalonamento são resolvidas em tempo de compilação e o núcleo de escalonamento altamente especializado em estruturas aninhadas de paralelismo não considera muitas variantes de escalonamento. De forma semelhante *OpenMP* tem uma estratégia de escalonamento que não possui margem para alternativas nas decisões. Já em *TBB*, o que se observa é que uma grande quantidade de informações sobre a estrutura do programa é passada para o núcleo de escalonamento, estas informações permitem tomadas de decisões efetivas.

5.4.2 Resultados Bucket-Sort

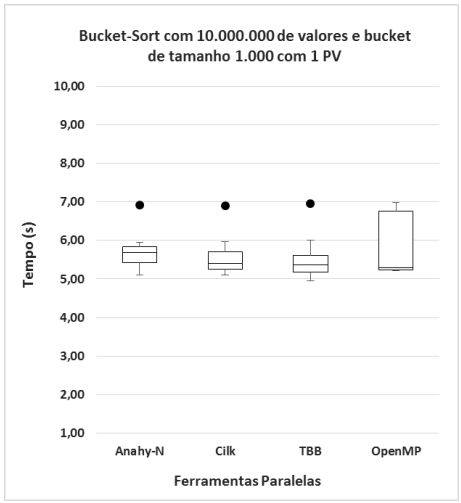
As Figuras 30 e 31 apresentam, respectivamente, os resultados obtidos na execução da aplicação *Bucket Sort* na ordenação de 10.000.000 de valores com e sem *Hyper-threading*, na arquitetura GoodTwin.

O que se pode ver é que, tanto nas execuções com *Hyper-threading* ativado, quanto nas execuções em que ele esteve desativado, os resultados são bastante semelhantes. Todas as ferramentas se mantiveram equivalentes, com exceção da estratégia proposta neste trabalho, que ao atingir o limite de *cores* reais da arquitetura, começou a perder desempenho em relação às demais ferramentas.

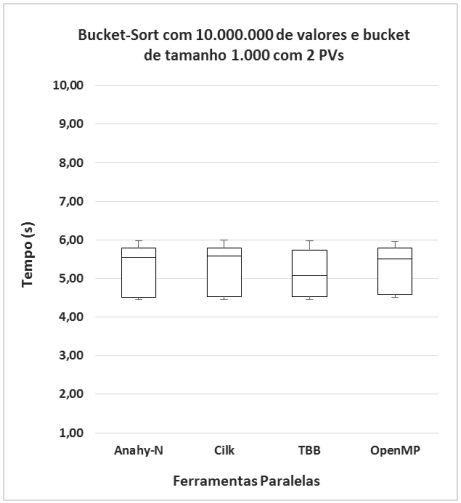
Este fato pode ser entendido em função da natureza da aplicação, a qual é refletida em um paralelismo de simples particionamento de um vetor em segmentos atribuídos a *threads* distintos. Este tipo de paralelismo não é beneficiado pela estratégia de escalonamento proposto em que a localidade de dados é um aspecto relevante nas decisões de mapeamento dos *threads*.

Isso fica mais evidente ao se analisar os gráficos dos *speedups* obtidos, representados nas Figuras 32(a) para a versão com *Hyper-threading* ativado e 32(b) para a versão com *Hyper-threading* desativado.

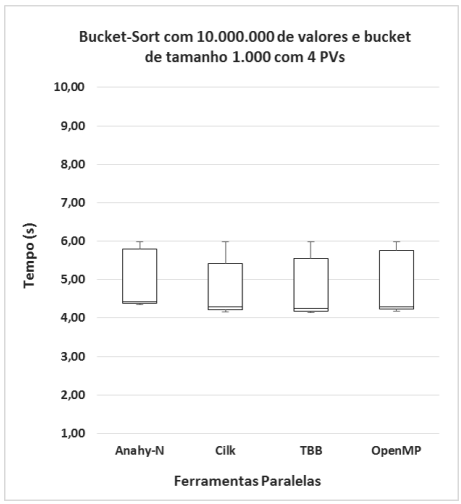
Já nos experimentos realizados na arquitetura Hydra, os resultados obtidos a partir da estratégia proposta neste trabalho, com esta aplicação, não se mostraram equivalentes aos resultados das demais ferramentas, como mostram as figuras 33 e 34, com, respectivamente, os tempos anotados da execução do problema e o *speedup* obtido em relação à execução sequencial. Estes resultados negativos apenas ressaltam a não vocação do escalonador proposto para esta classe de problemas.



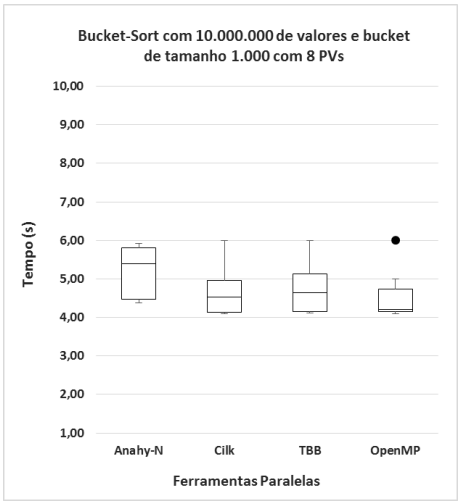
(a) 1 Thread



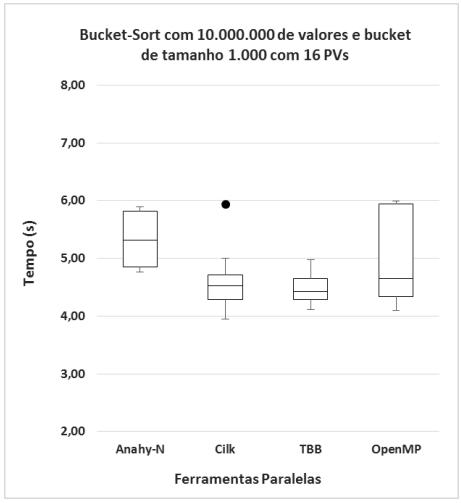
(b) 2 Threads



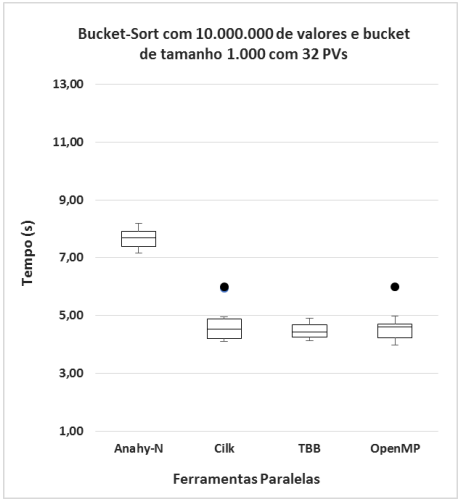
(c) 4 Threads



(d) 8 Threads

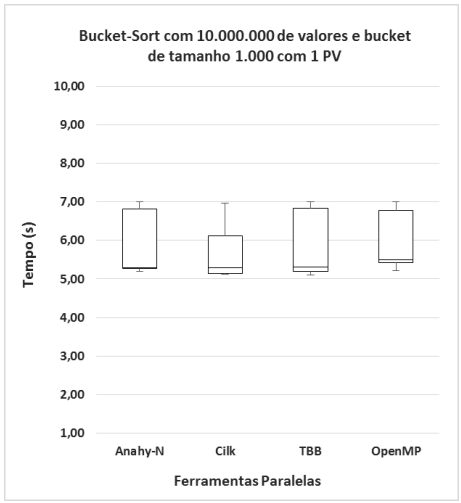


(e) 16 Threads

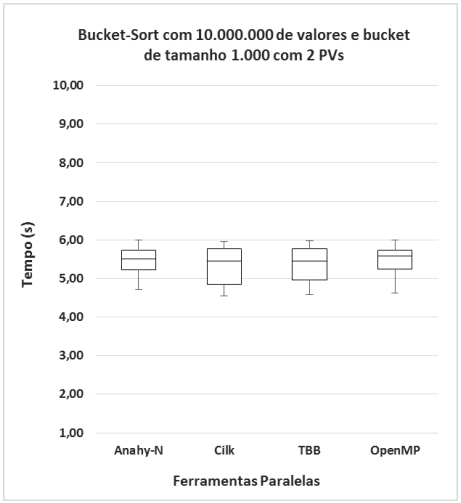


(f) 32 Threads

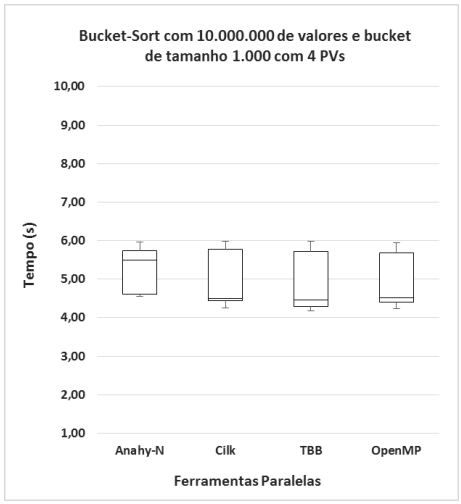
Figura 30: Resultados obtidos com Bucket Sort na ordenação de 10.000.000 valores na arquitetura GoodTwin com *Hyper-threading*.



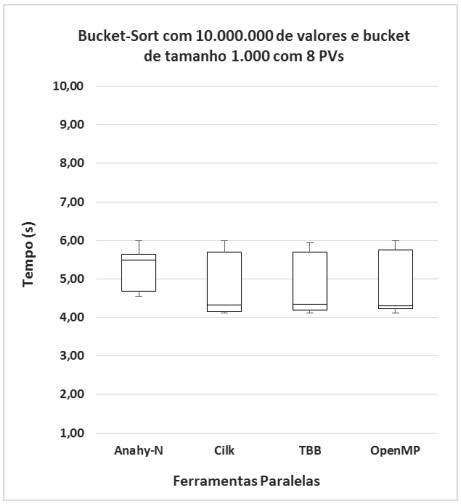
(a) 1 Thread



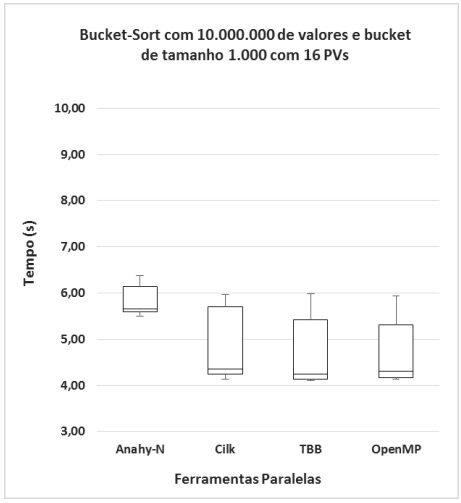
(b) 2 Threads



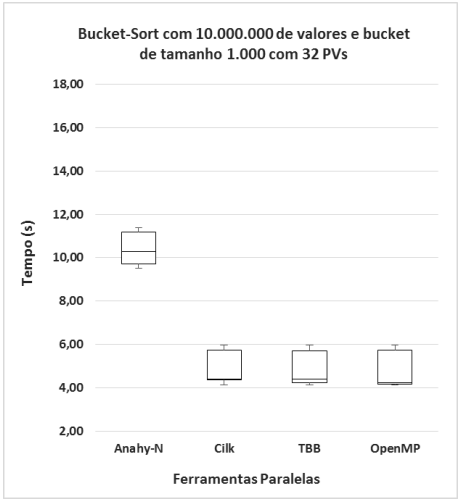
(c) 4 Threads



(d) 8 Threads

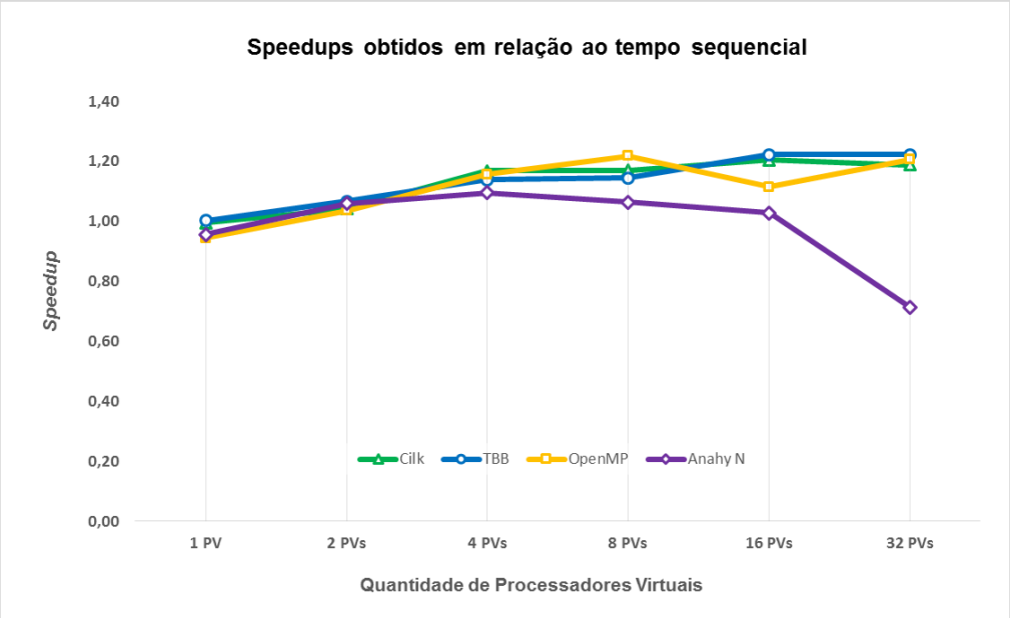


(e) 16 Threads

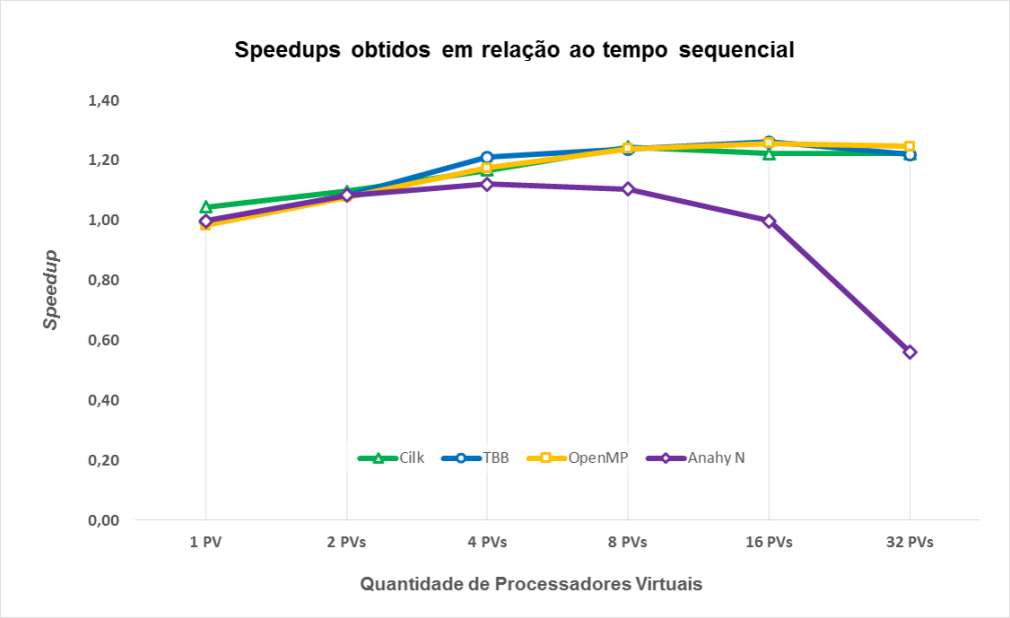


(f) 32 Threads

Figura 31: Resultados obtidos com Bucket-Sort na ordenação de 10.000.000 valores na arquitetura GoodTwin sem *Hyper-threading*.

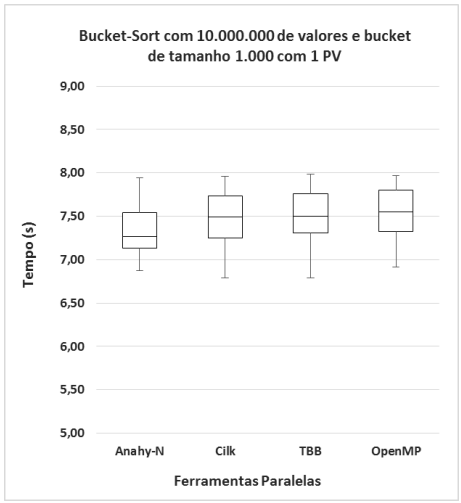


(a) Com Hyper-threading

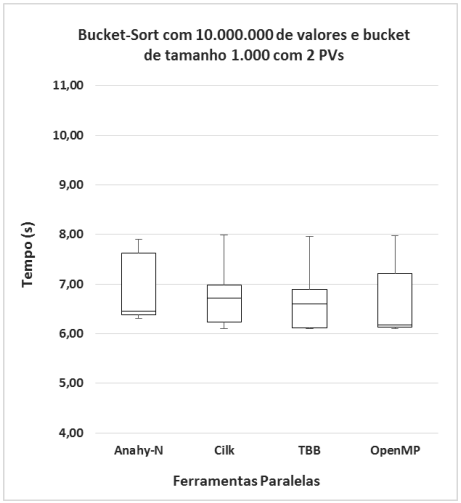


(b) Sem Hyper-threading

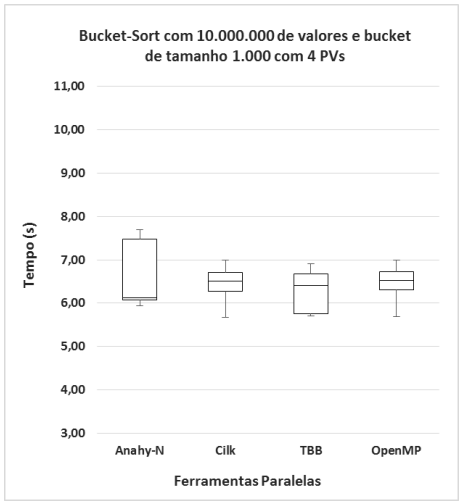
Figura 32: Gráfico dos *Speedups* obtidos com a aplicação Bucket-Sort em relação ao tempo de execução sequencial na arquitetura GoodTwin: (a) com *Hyper-threading* e (b) sem *Hyper-threading*.



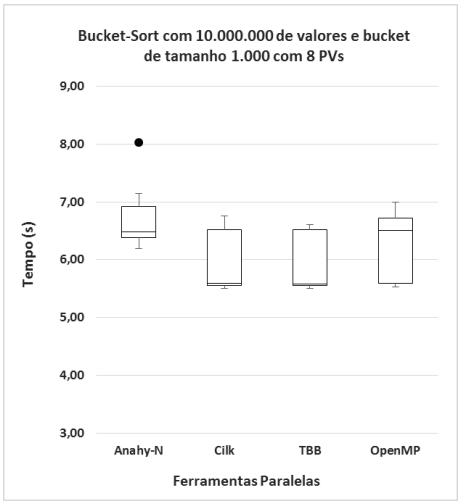
(a) 1 Thread



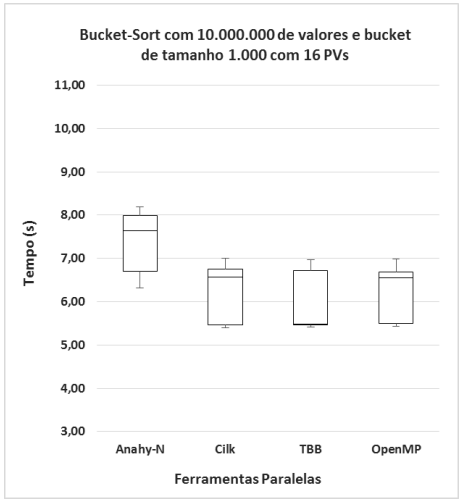
(b) 2 Threads



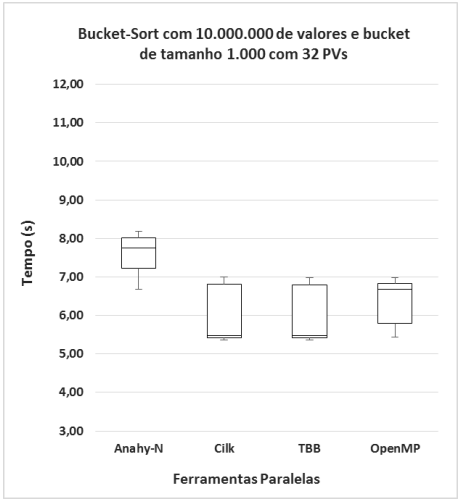
(c) 4 Threads



(d) 8 Threads

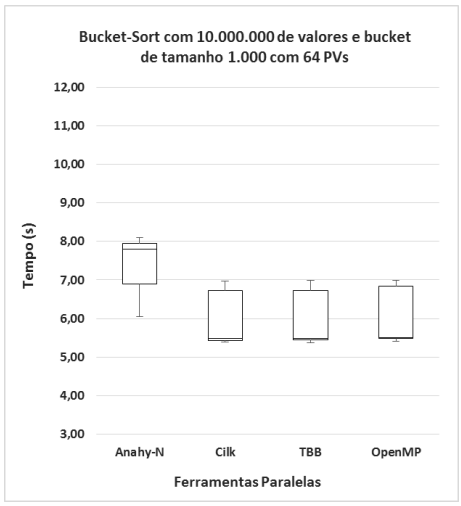


(e) 16 Threads

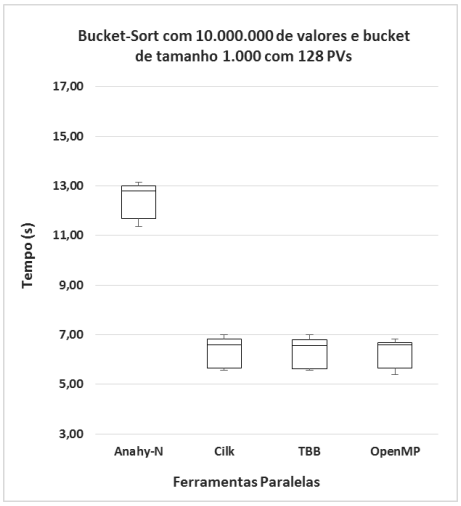


(f) 32 Threads

Figura 33: Resultados obtidos com Bucket-Sort na ordenação de 10.000.000 valores na arquitetura Hydra. *(cont.)*



(g) 64 Threads



(h) 128 Threads

Figura 33: Resultados obtidos com Bucket-Sort na ordenação de 10.000.000 valores na arquitetura Hydra.

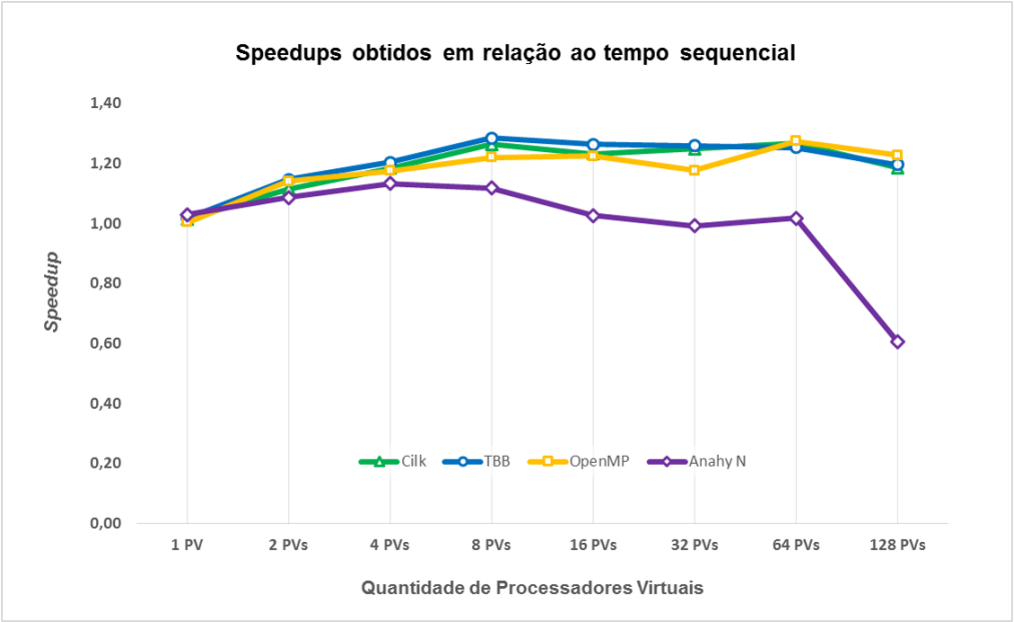


Figura 34: Gráfico dos *Speedups* obtidos com a aplicação Bucket-Sort em relação ao tempo de execução sequencial, na arquitetura Hydra.

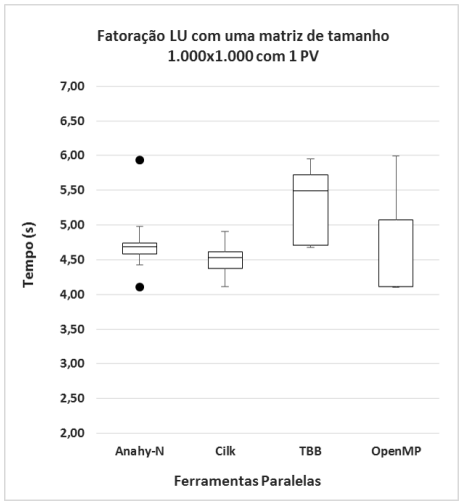
5.4.3 Resultados Fatoração LU

As Figuras 35 e 36 representam, respectivamente, os tempos de execução da aplicação Fatoração LU com uma matriz de tamanho 1.000x1.000, com e sem *Hyper-threading*, na arquitetura GoodTwin. Nos resultados desta aplicação, a estratégia proposta neste trabalho obteve bons índices de desempenho, demonstrando que ela é escalável. O ápice de desempenho se deu no momento em que atingiu o número de processadores reais da arquitetura.

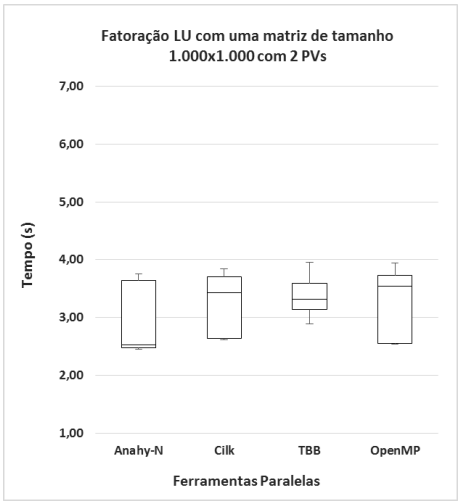
Os gráficos de *speedup* ilustrados na Figura 37(a), com *Hyper-threading* e na Figura 37(b), sem *Hyper-threading*, mostram exatamente isso, tanto no *speedup* do experimento com *Hyper-threading*, quanto no *speedup* do experimento sem *Hyper-threading*, ao chegar no limite de cores da arquitetura (oito *cores* reais), o *speedup* se manteve ou teve uma leve melhoria.

O bom índice de desempenho obtido pela ferramenta OpenMP se deve ao fato de que a implementação da aplicação foi otimizada, utilizando a operação atômica `compare_and_swap` para sincronizar o cálculo das tarefas indicadas na Figura 23. Desta forma, a implementação realizada não é composta por dois *loops* aninhados (o laço externo percorrendo a diagonal e o laço interno percorrendo as colunas), mas sim, pela criação de lotes de tarefas (*tasks*) responsáveis pelo cálculo de cada posição da matriz. O bom desempenho de OpenMP se deve, então, a essa otimização em nível de *tasks*, como é o caso da estratégia proposta neste trabalho.

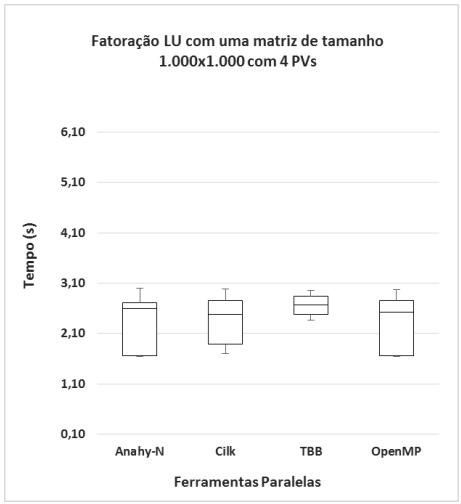
O mesmo desempenho obtido, pela estratégia desenvolvida, na arquitetura GoodTwin, é refletido na arquitetura Hydra. As Figuras 38 e 39 mostram os gráficos de caixa que ilustram os tempos de execução para esta aplicação e os *speedups* obtidos, também em relação ao tempo sequencial, na arquitetura Hydra.



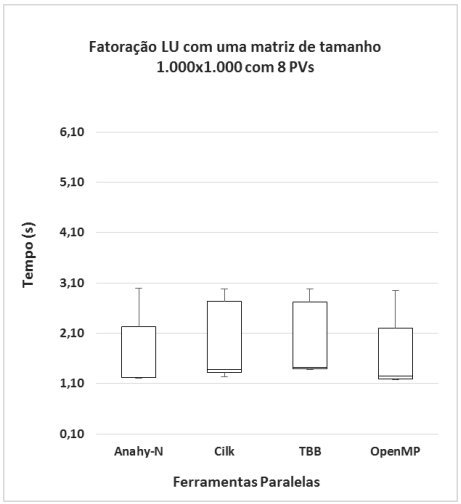
(a) 1 Thread



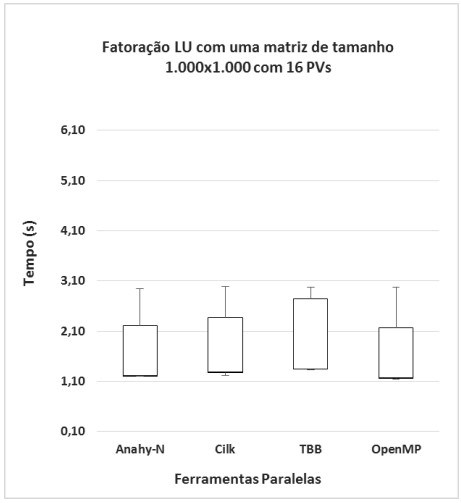
(b) 2 Threads



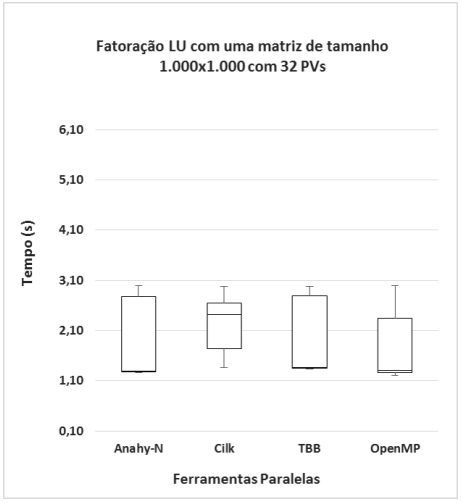
(c) 4 Threads



(d) 8 Threads

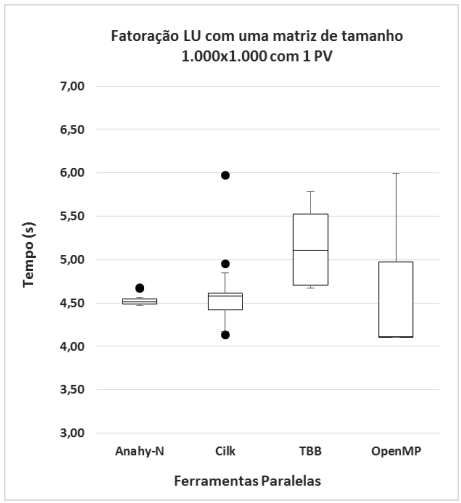


(e) 16 Threads

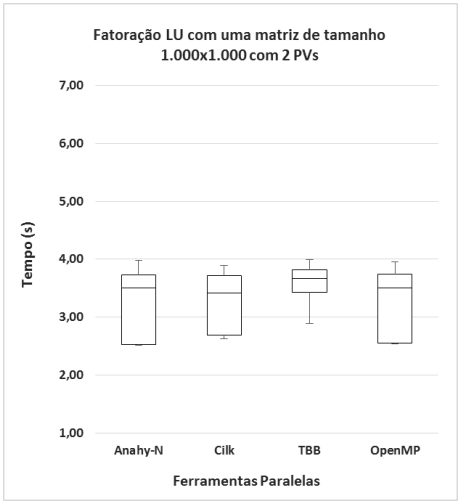


(f) 32 Threads

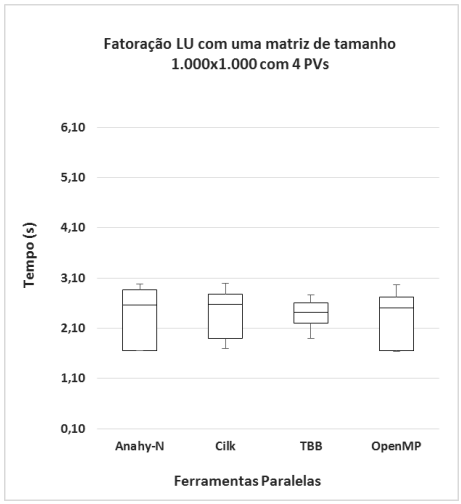
Figura 35: Resultados obtidos com a aplicação Fatoração LU de uma matriz de tamanho 1.000x1.000 na arquitetura GoodTwin com *Hyper-threading*.



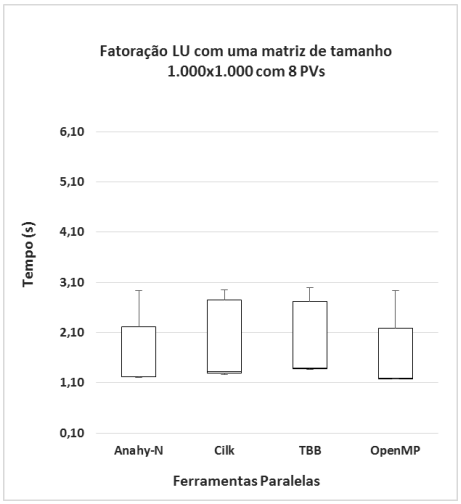
(a) 1 Thread



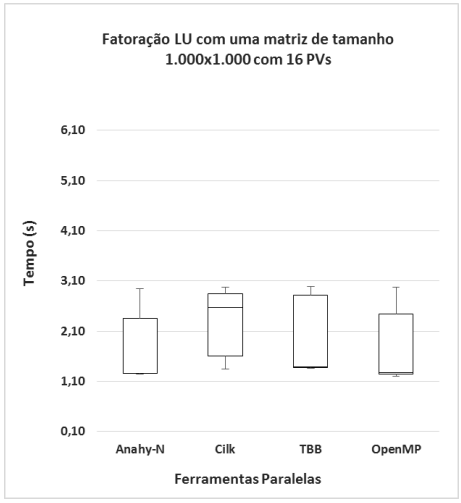
(b) 2 Threads



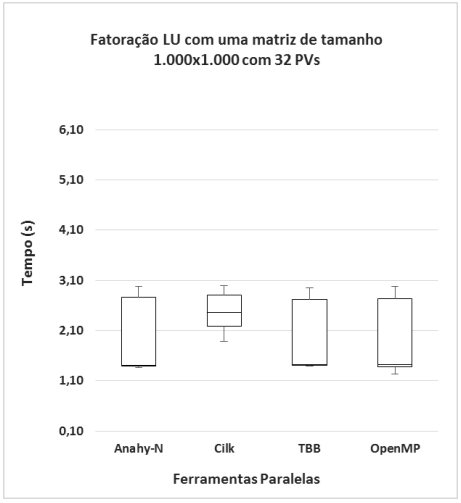
(c) 4 Threads



(d) 8 Threads

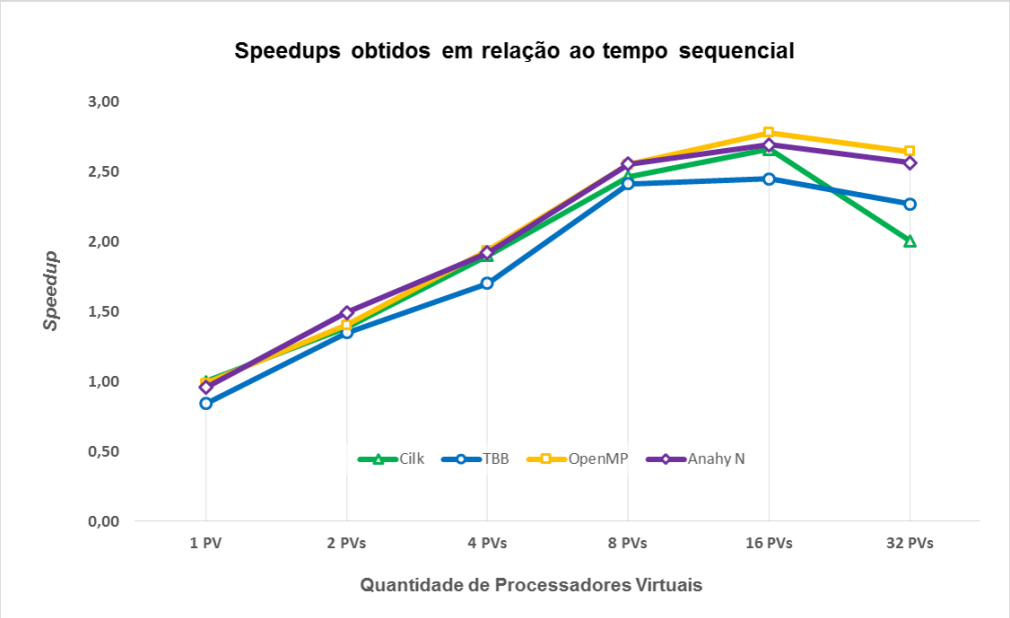


(e) 16 Threads

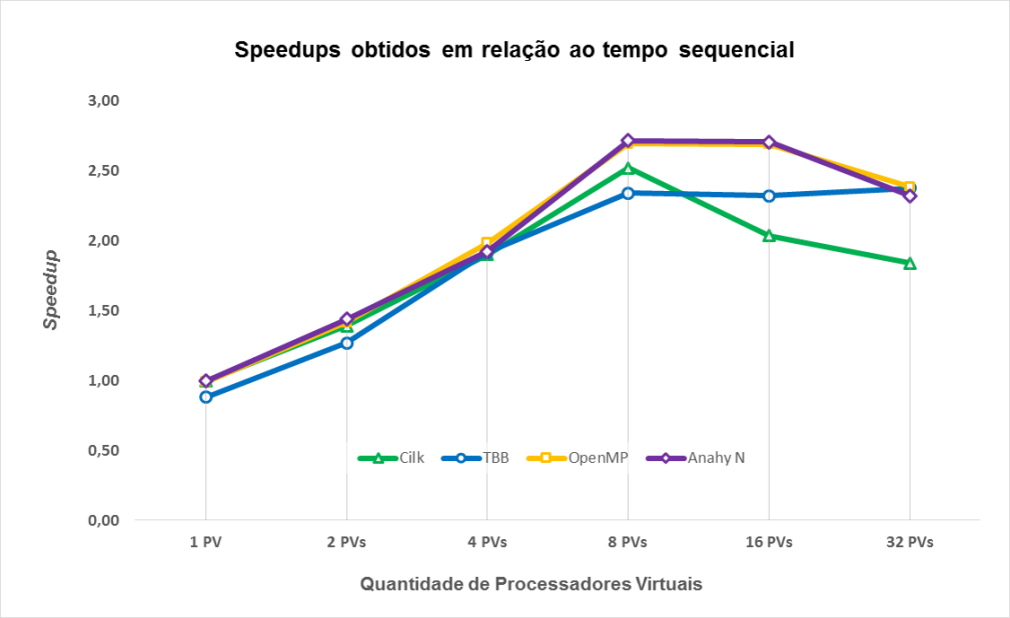


(f) 32 Threads

Figura 36: Resultados obtidos com a aplicação Fatoração LU de uma matriz de tamanho 1.000x1.000 na arquitetura GoodTwin sem *Hyper-threading*.

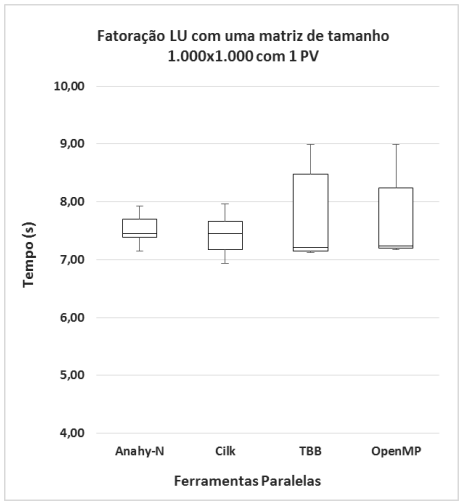


(a) Com Hyper-threading

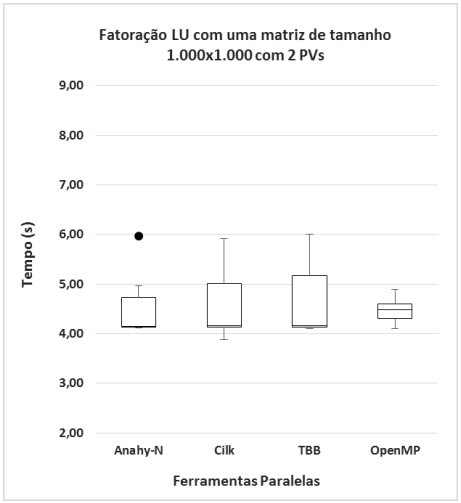


(b) Sem Hyper-threading

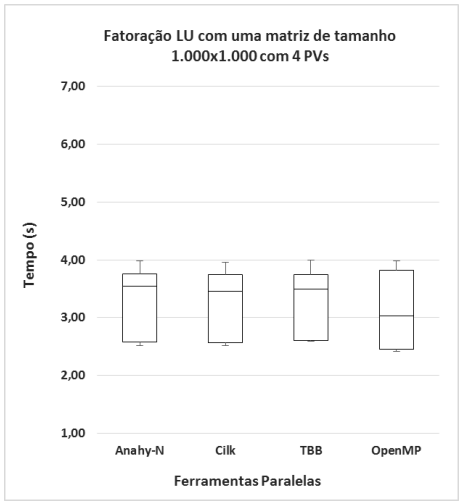
Figura 37: Gráfico dos *Speedups* obtidos com a aplicação Fatoração LU em relação ao tempo de execução sequencial na arquitetura GoodTwin: (a) com *Hyper-threading* e (b) sem *Hyper-threading*.



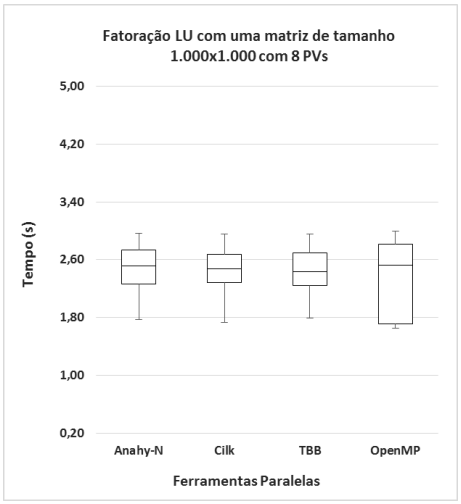
(a) 1 Thread



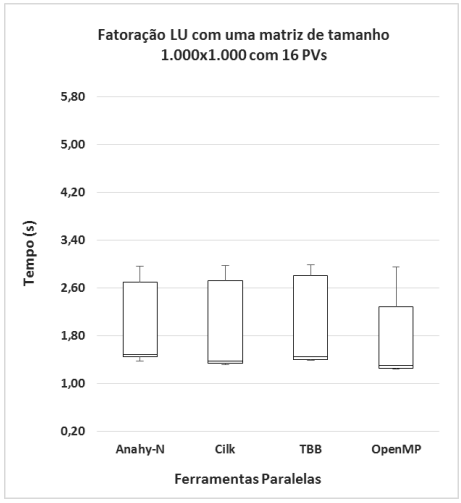
(b) 2 Threads



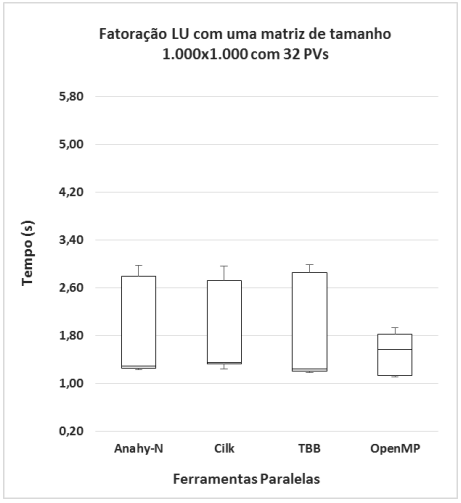
(c) 4 Threads



(d) 8 Threads

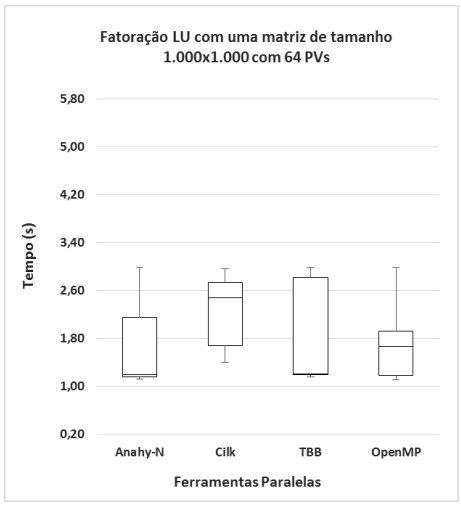


(e) 16 Threads

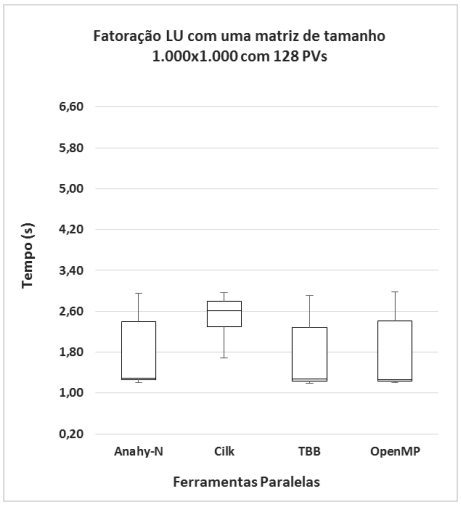


(f) 32 Threads

Figura 38: Resultados obtidos com a aplicação Fatoração LU de uma matriz de tamanho 1.000x1.000 na arquitetura Hydra. *(cont.)*



(g) 64 Threads



(h) 128 Threads

Figura 38: Resultados obtidos com a aplicação Fatoração LU de uma matriz de tamanho 1.000x1.000 na arquitetura Hydra.

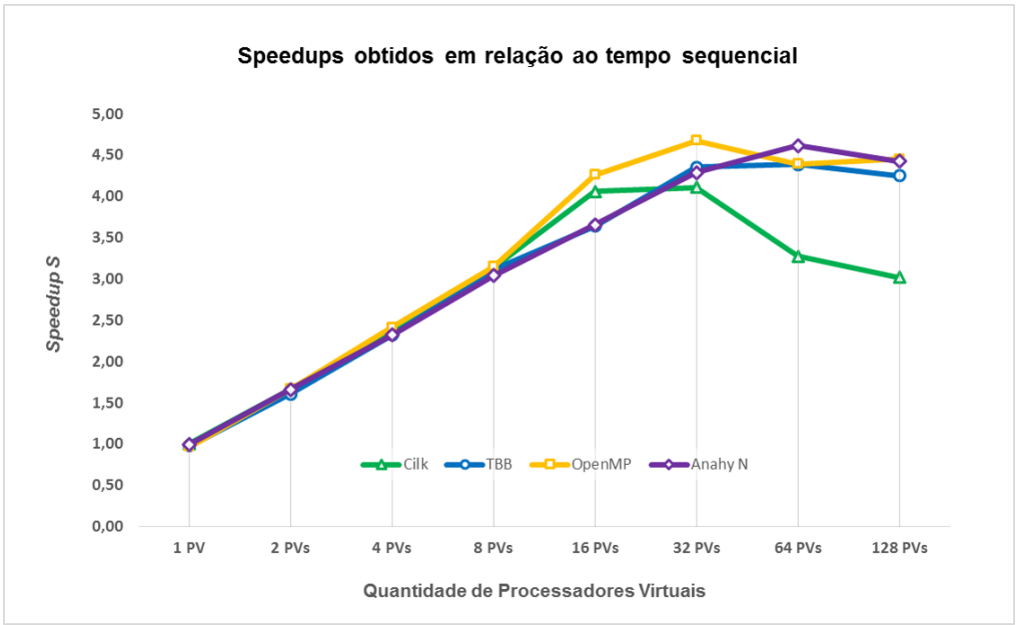


Figura 39: Gráfico dos *Speedups* obtidos com a aplicação Fatoração LU em relação ao tempo de execução sequencial, na arquitetura Hydra.

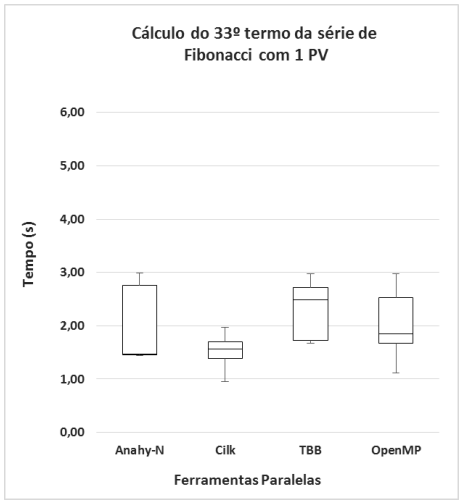
5.4.4 Resultados Fibonacci

Esta aplicação é importante pois ela reflete, exatamente, o problema à estratégia de escalonamento da versão original de Anahy. As Figuras 40 e 41 apresentam os tempos tempos de execução obtidos com esta aplicação, na arquitetura GoodTwin com e sem *Hyper-threading*, nesta ordem. Assim como nos experimentos realizados com a Fatoração LU, a estratégia proposta neste trabalho obteve bons índices de desempenho, perdendo apenas pela ferramenta Cilk, a qual é especializada nesse tipo de aplicação e conta, ainda, com otimizações na compilação.

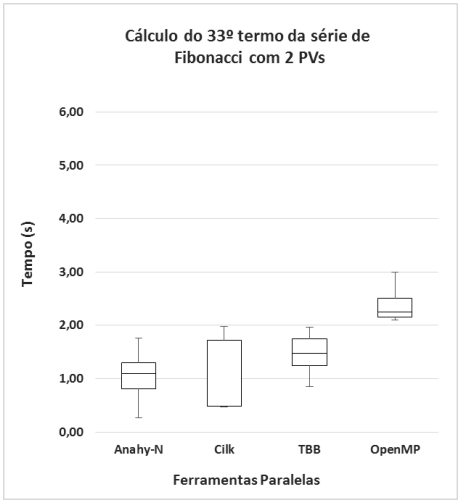
Os gráficos de *speedup*, mostrados na Figura 42(a), com *Hyper-threading* e na Figura 42(b), sem *Hyper-threading*, confirmam o desempenho obtido pela estratégia desenvolvida neste trabalho. Nos experimentos realizados na arquitetura GoodTwin, a ferramenta OpenMP foi a ferramenta que obteve os piores *speedups*, para todos os casos de teste, seguida pela ferramenta TBB, que foi o segundo pior caso.

Nos experimentos realizados na arquitetura Hydra, cujos resultados dos tempos de execução são mostrados na Figura 43 e os *speedups* na Figura 44, a estratégia proposta obteve um bom desempenho e se mostrou equivalente as demais ferramentas enquanto o número de processadores virtuais era menor ou igual ao número de processadores reais da arquitetura.

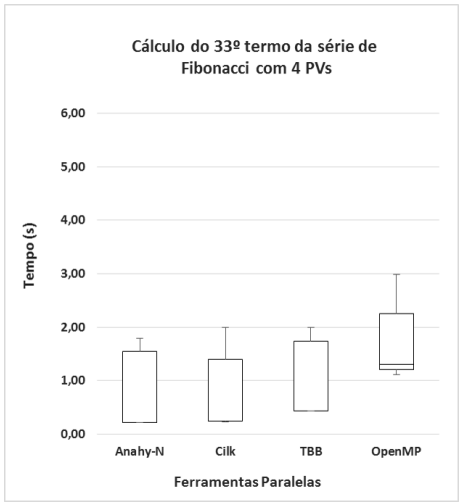
O gráfico dos *speedups*, apresentado na Figura 44, mostra o desempenho obtido por cada ferramenta em relação a média dos tempos de execução da versão sequencial. A ferramenta TBB foi a ferramenta que obteve o melhor desempenho, seguida pela estratégia proposta (Anahy-N) e Cilk. Anahy-N e Cilk, na medida em que o número de processadores virtuais foi aumentando, tiveram seu desempenho comprometido, devido aos *overheads* para gerenciar os *threads*. OpenMP foi a ferramenta que obteve o pior desempenho.



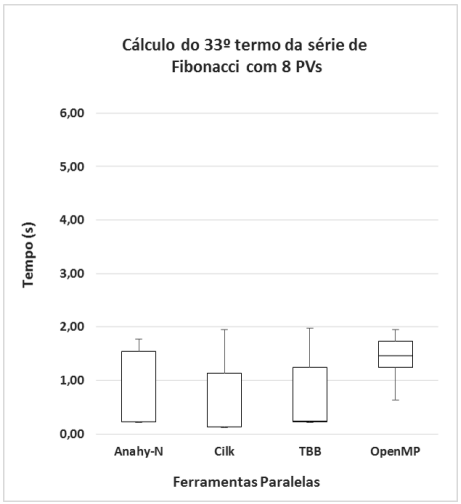
(a) 1 Thread



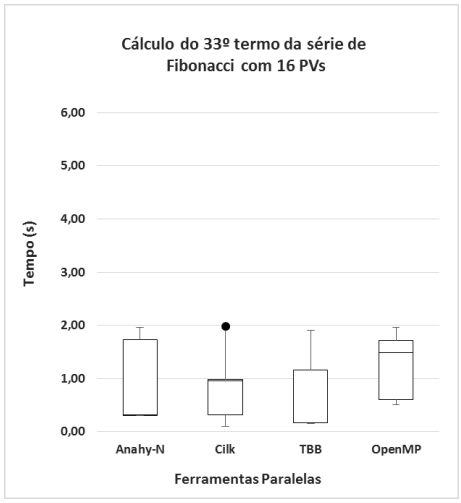
(b) 2 Threads



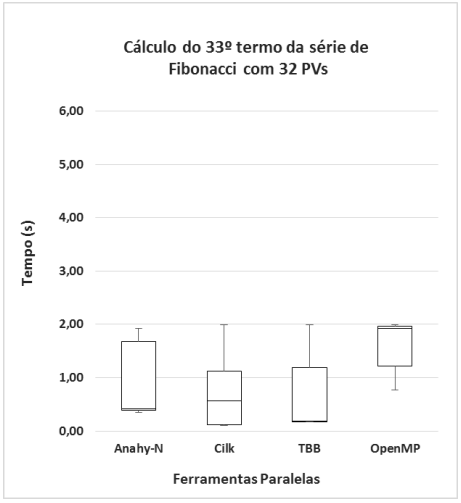
(c) 4 Threads



(d) 8 Threads

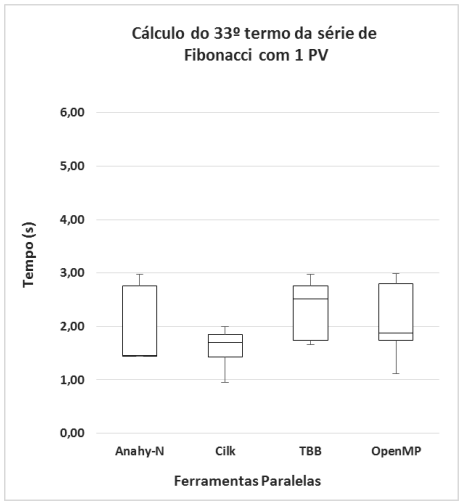


(e) 16 Threads

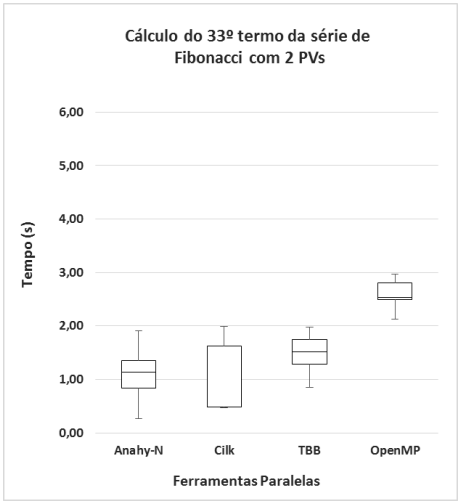


(f) 32 Threads

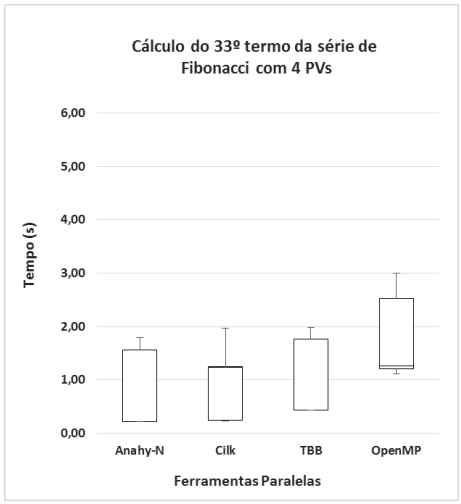
Figura 40: Resultados obtidos com a aplicação Fibonacci para o cálculo do 33° número da série na arquitetura GoodTwin com *Hyper-threading*.



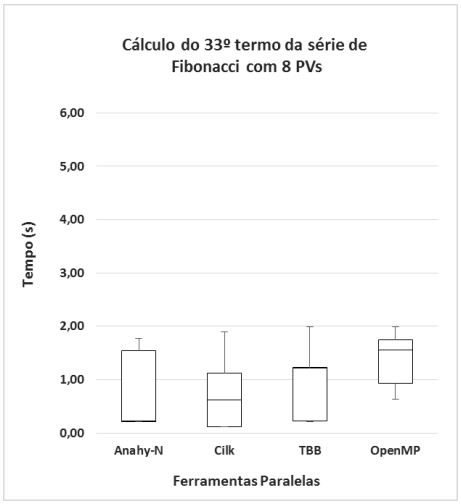
(a) 1 Thread



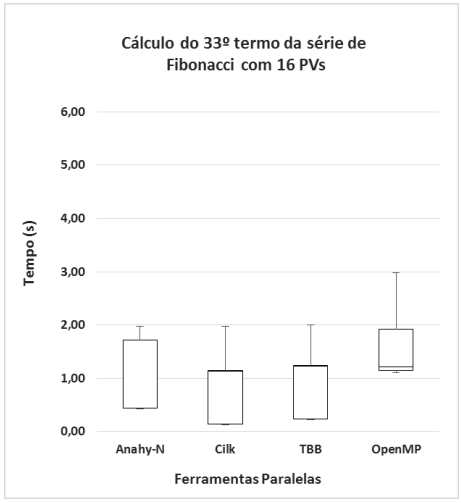
(b) 2 Threads



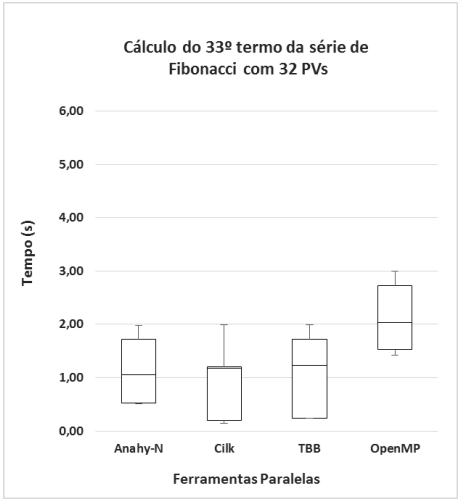
(c) 4 Threads



(d) 8 Threads

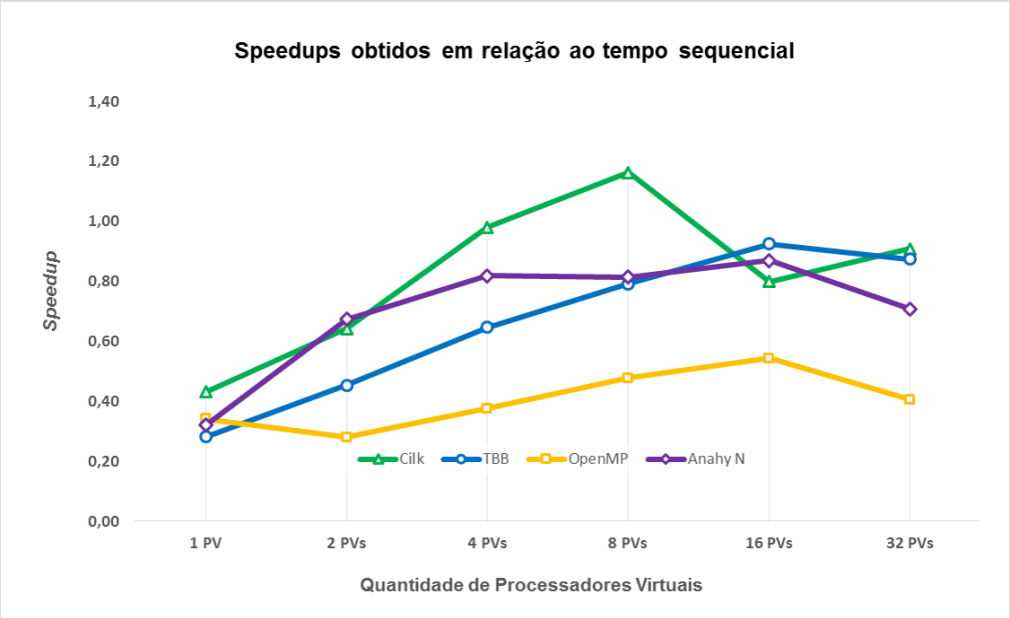


(e) 16 Threads

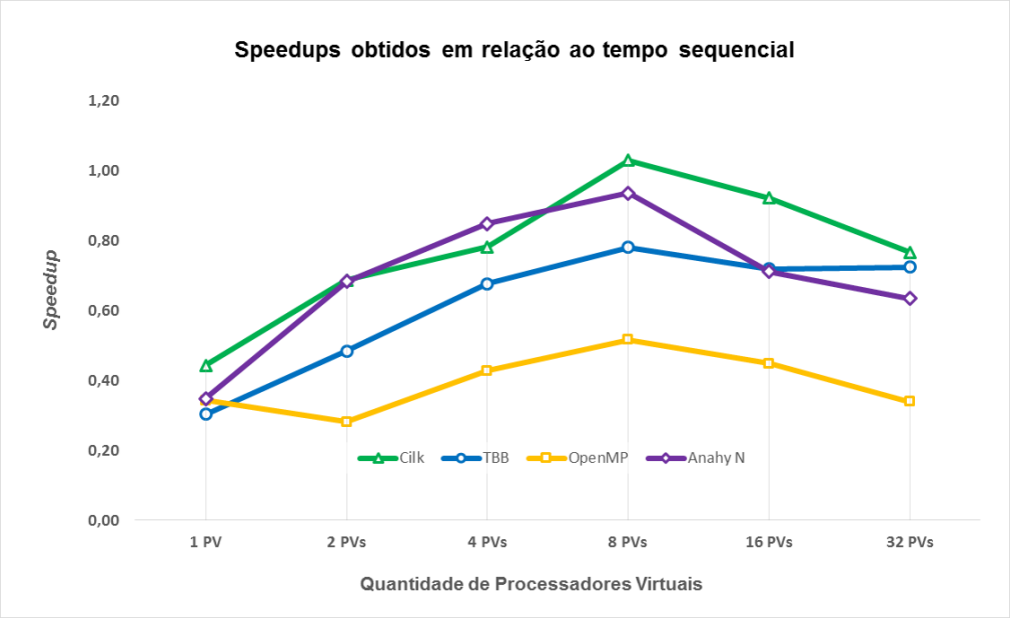


(f) 32 Threads

Figura 41: Resultados obtidos com a aplicação Fibonacci para o cálculo do 33° número da série na arquitetura GoodTwin sem *Hyper-threading*.

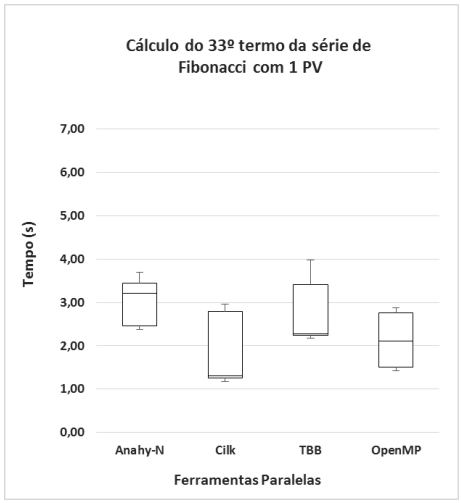


(a) Com Hyper-threading

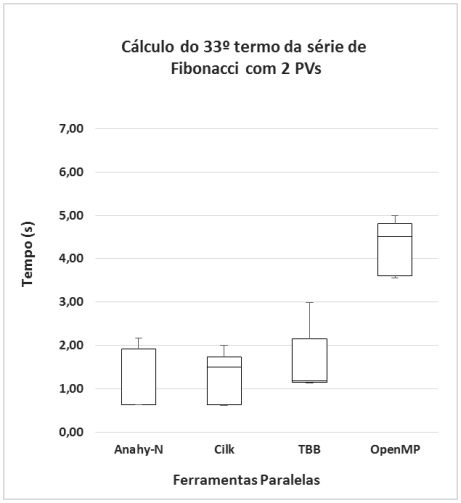


(b) Sem Hyper-threading

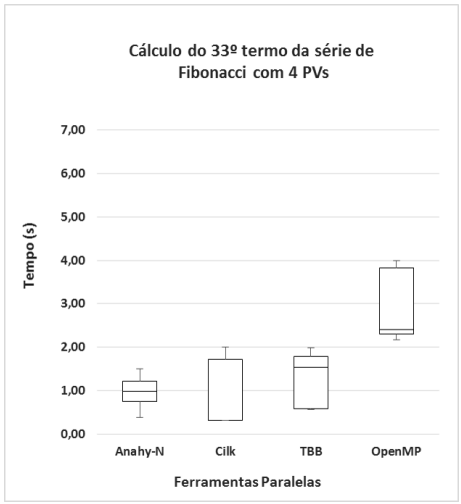
Figura 42: Gráfico dos *Speedups* obtidos com a aplicação Fibonacci para o cálculo do 33° número da série em relação ao tempo de execução sequencial na arquitetura GoodTwin: (a) com *Hyper-threading* e (b) sem *Hyper-threading*.



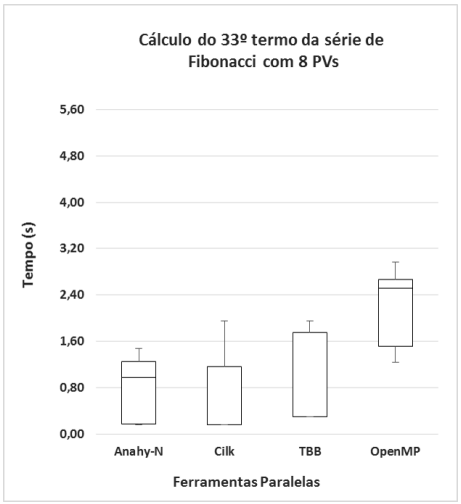
(a) 1 Thread



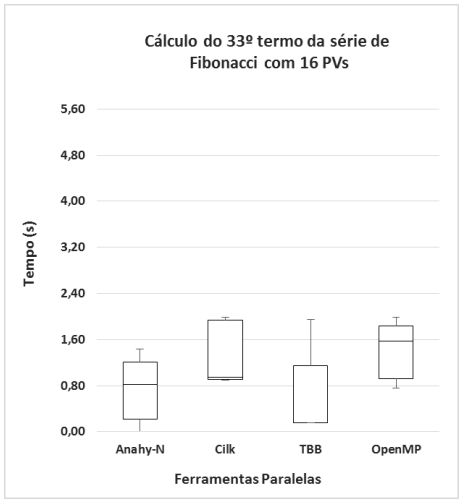
(b) 2 Threads



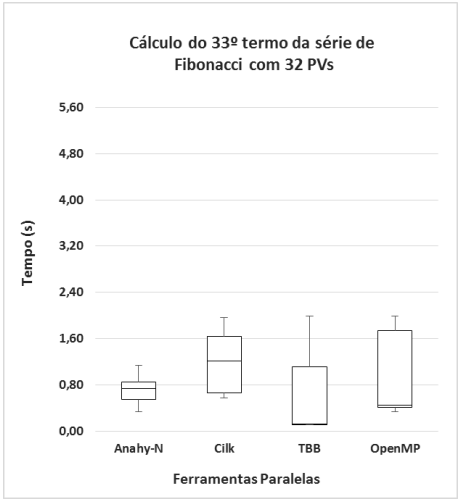
(c) 4 Threads



(d) 8 Threads

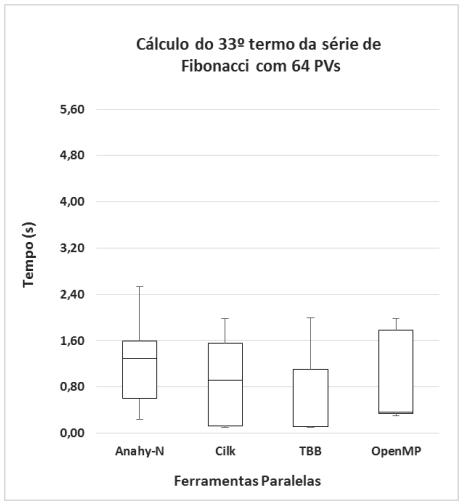


(e) 16 Threads

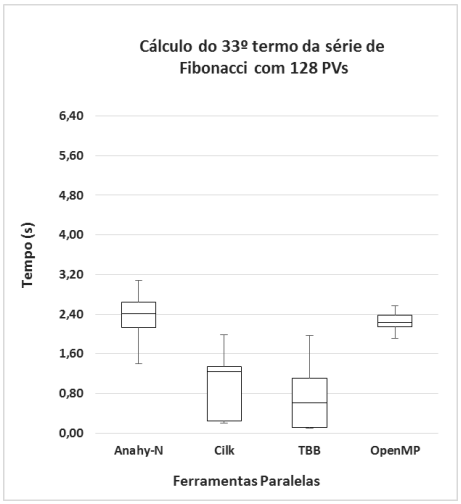


(f) 32 Threads

Figura 43: Resultados obtidos com a aplicação Fibonacci para o cálculo do 33° número da série na arquitetura Hydra. *(cont.)*



(g) 64 Threads



(h) 128 Threads

Figura 43: Resultados obtidos com a aplicação Fibonacci para o cálculo do 33° número da série na arquitetura Hydra.

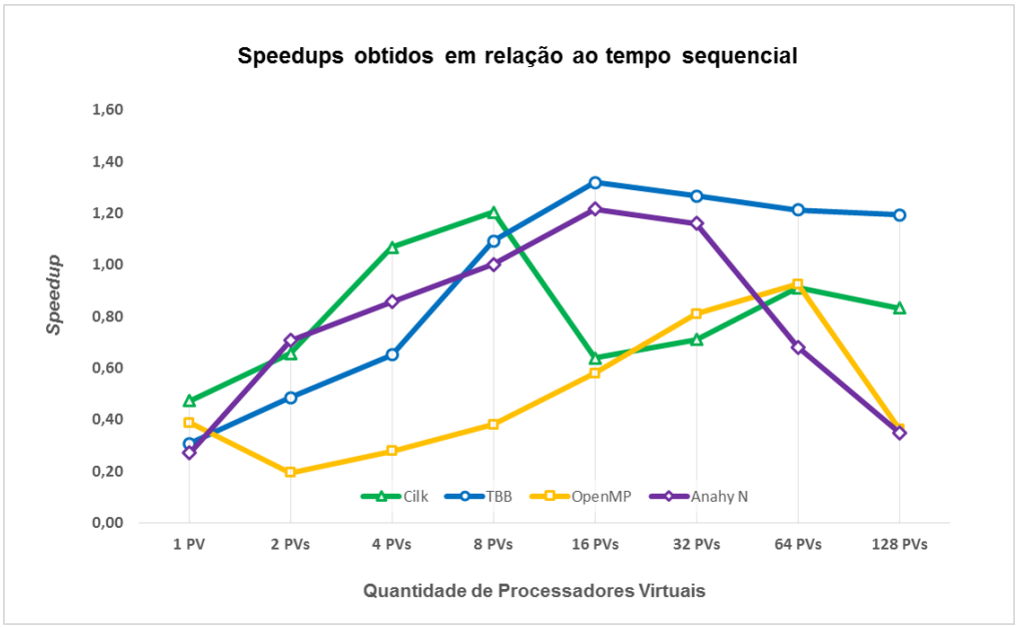


Figura 44: Gráfico dos *Speedups* obtidos com a aplicação Fibonacci para o cálculo do 33° número da série em relação ao tempo de execução sequencial, na arquitetura Hydra.

5.5 Anahy-N vs. Anahy

Nesta seção é apresentada uma comparação entre a versão original de Anahy e a versão que faz uso da estratégia proposta neste trabalho (Anahy-N), com a finalidade de investigar se a estratégia desenvolvida neste trabalho contribuiu, ou não, para o aumento de desempenho da versão original de Anahy.

5.5.1 Aplicação testada

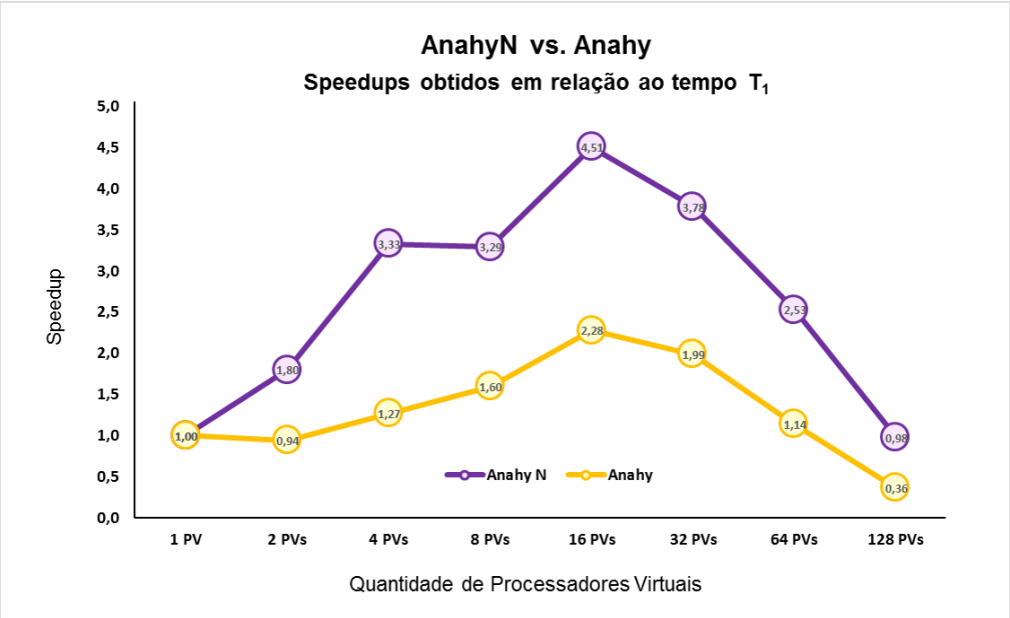
Para fins de comparação de desempenho entre as duas versões da ferramenta, optou-se pelo uso da aplicação Fibonacci. Este *benchmark* é importante porque ele reflete, naturalmente, no problema à estratégia de escalonamento da versão original de Anahy. Sua principal característica é gerar um grafo regular, desbalanceado, uma vez que a tarefa responsável por calcular a posição $n-1$ possui maior quantidade de trabalho que aquela que for calcular a posição $n-2$.

5.5.2 Resultados obtidos

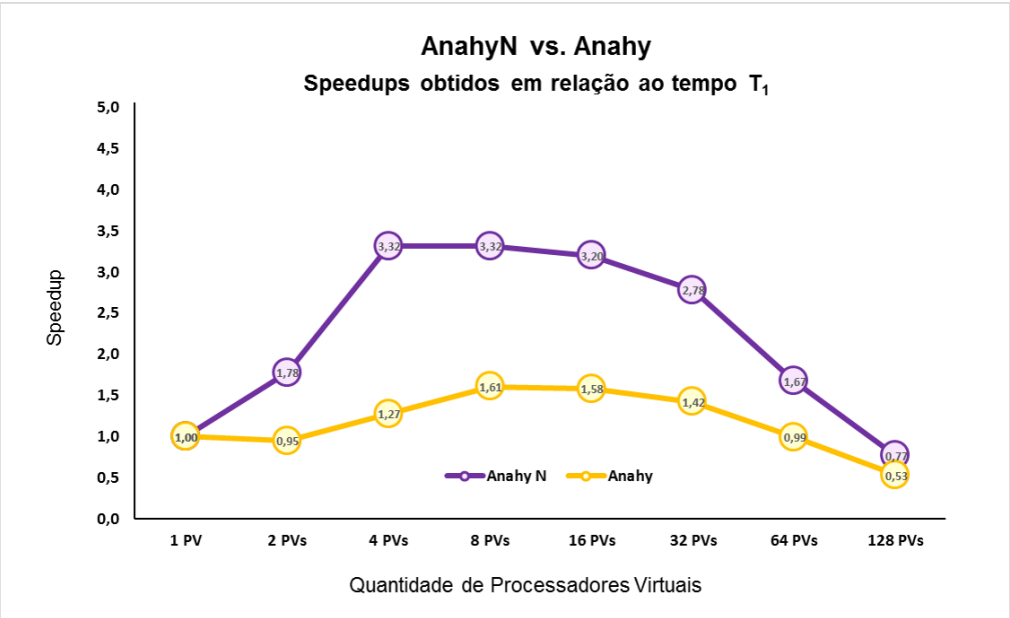
As Figuras 45 e 46 apresentam os *speedups* obtidos nas arquiteturas GoodTwin e Hydra, respectivamente. Estes *speedups* foram calculados em relação ao tempo T_1, ou seja, o tempo de execução com um (1) *thread*. Sendo que, da mesma forma como os demais experimentos, na arquitetura GoodTwin avaliamos com *Hyper-threading* ativado (Figura 45(a)) e desativado (Figura 45(b)).

Na arquitetura GoodTwin, como era de se esperar, a estratégia proposta neste trabalho obteve ótimos resultados, exceto quanto executado com 128 *threads*, onde o potencial de paralelismo da nova versão de Anahy se esgota, chegando muito próximo do desempenho obtido pela versão original de Anahy.

Para todos os casos, em todas as arquiteturas testadas, pode-se observar que Anahy-N aumentou, significativamente, o desempenho da versão original de Anahy. Contudo, quando a execução chega no limite dos recursos disponíveis na máquina, a tendência é que o *overhead* destas etapas adicionais de escalonamento, presentes em Anahy-N, comece a refletir em uma perda de desempenho mais acentuada em relação a versão original de Anahy.



(a) Com Hyper-threading



(b) Sem Hyper-threading

Figura 45: Gráfico dos *Speedups* obtidos com a aplicação Fibonacci em relação ao tempo de execução T_1 na arquitetura GoodTwin: (a) com *Hyper-threading* e (b) sem *Hyper-threading*.

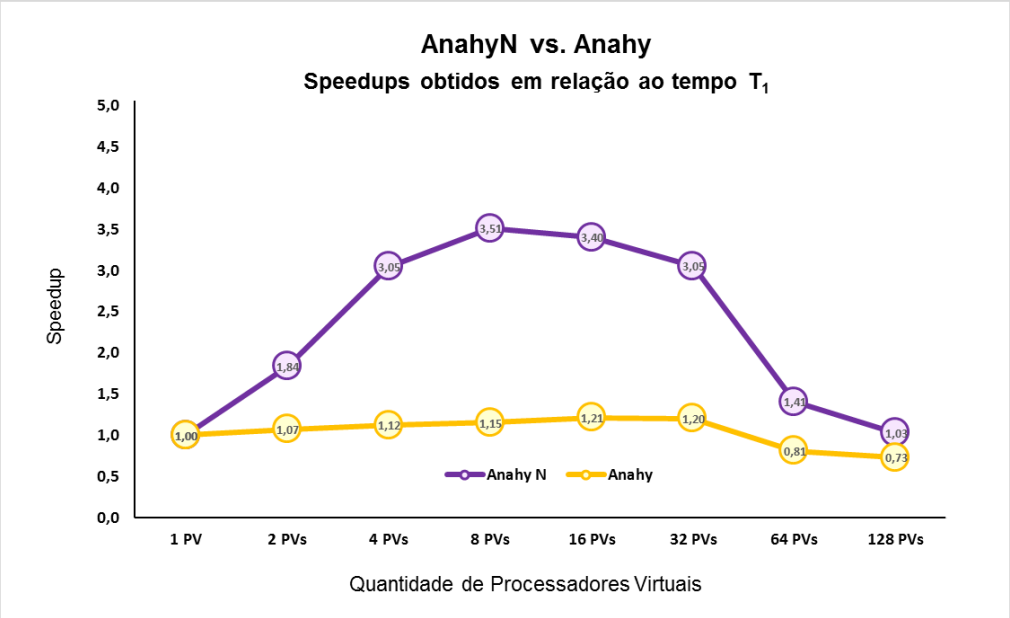


Figura 46: Gráfico dos *Speedups* obtidos com a aplicação Fibonacci em relação ao tempo de execução T_1 na arquitetura Hydra.

5.6 Considerações sobre o capítulo

As ferramentas Cilk e TBB possuem um modelo de escalonamento muito semelhante. Uma diferença entre elas é que a ferramenta TBB permite ao programador, no momento do desenvolvimento da aplicação, especificar que determinadas tarefas, que geram maior computação, sejam colocadas em uma lista global, a parte das demais tarefas para que, no momento em que um *worker* for buscar trabalho, essas tarefas sejam executadas prioritariamente.

Outra diferença entre estas ferramentas, é que em Cilk, ao ser criado um novo *thread*, automaticamente, esse novo *thread* passa a ser executado, deixando o *thread* que o criou em espera. Este ponto em espera é representado pelas versões rápidas e lentas do *thread*. Caso a continuação seja migrada, a versão lenta, que inclui interações com o núcleo de escalonamento, é executada. Caso não ocorra migração, a execução é continuada no mesmo fluxo sem onerar a execução. Em TBB, o programador pode, ao criar uma tarefa, explicitar que as tarefas a serem criadas na sequência, são uma continuação desta tarefa criada anteriormente. A ferramenta TBB oferece recursos em sua interface de programação para reproduzir, em tempo de execução, as ações tomadas em Cilk durante o processo de compilação.

Observando os gráficos de *speedup* apresentados, nota-se que nenhuma aplicação possibilitou utilizar a pleno os recursos da arquitetura, resultando em *speedups* relativamente baixos para o número de processadores disponíveis.

De maneira geral, nas comparações da estratégia proposta neste trabalho

com as demais ferramentas, foi possível constatar que a estratégia desenvolvida foi capaz de oferecer resultados equivalentes, em algumas vezes até melhores que os resultados fornecidos por OpenMP, Cilk e TBB, consideradas as principais ferramentas de programação paralela com escalonamento em nível aplicativo disponíveis.

6 CONSIDERAÇÕES FINAIS

Neste trabalho, foi modelada e implementada uma estratégia de escalonamento, em nível aplicativo, para programas *multithread* em arquiteturas NUMA. A validação desta estratégia se deu pela sua implementação no núcleo de execução do ambiente Anahy e com a realização de um conjunto de experimentos visando aferir seu desempenho. O objetivo desta estratégia foi minimizar o impacto das diferentes latências, provenientes da distribuição física dos módulos de memória em arquiteturas NUMA, na execução de aplicações paralelas. A abordagem adotada considera os custos da migração de *threads* considerando a localidade de seus dados.

No Capítulo 2 foram estudadas as técnicas de escalonamento utilizadas na literatura e as principais ferramentas que utilizam estratégias de escalonamento em nível aplicativo, Cilk Plus (BLUMOFE et al., 1995), Threading Building Blocks (TBB) (REINDERS, 2007) e OpenMP (CHAPMAN; JOST; PAS, 2008), bem como o algoritmo *HwTopoLB* (PILLA et al., 2014) que, diferente das demais ferramentas, considera a topologia da arquitetura para realizar o escalonamento.

O Capítulo 3 trouxe todo o ferramental envolvido no desenvolvimento da estratégia proposta. Caracterizou as arquiteturas NUMA onde, para se conseguir bons índices de desempenho, deve-se considerar a heterogeneidade das latências de acesso aos dados nos blocos distribuídos de memória. Mostrou como essas latências podem ser mesuradas a partir da extensão, por Pilla et al. (2013), da ferramenta *HwLoc* (BROQUEDIS et al., 2010). Apresentou também o ambiente Anahy, o qual teve seu núcleo de execução alterado para comportar a estratégia proposta.

O Capítulo 4 tratou da apresentação da estratégia proposta, contemplando os modelos de aplicação, execução, arquitetura e escalonamento. Neste capítulo, foi visto como ocorre a execução de uma aplicação paralela e que, ao final de sua execução, essa aplicação pode ser desenhada como um DCG. Mostrou, também, as características da arquitetura NUMA que foram consideradas na concepção da estratégia de escalonamento proposta neste trabalho. Apresentou o

modelo de escalonamento, o qual altera a política original de *Anahy*, passando a considerar, as assimetrias de acesso aos dados na memória provenientes da arquitetura, no momento do escalonamento. Por fim, mostrou os principais passos realizados durante o processo de escalonamento.

O Capítulo 5 apresentou a avaliação experimental da estratégia proposta neste trabalho, iniciando pela metodologia adotada, apresentando as plataformas onde foram realizados os experimentos e suas principais características, bem como as aplicações que compõem os estudos de caso. Por fim, apresenta e discute os resultados obtidos com os estudos de caso, bem como uma comparação entre a versão original de Anahy e a versão estendida.

Os experimentos realizados no Capítulo 5 nos fornecem uma série de resultados que, de maneira geral, servem para comprovar que a estratégia proposta neste trabalho permite obter bons índices de desempenho em arquiteturas assimétricas, como é o caso das arquiteturas NUMA, quando comparada a outras ferramentas e, em particular, quando comparada à versão anterior de Anahy, onde, na aplicação testada, o *speedup* aumentou consideravelmente em ambas as plataformas testadas.

É importante ressaltar que OpenMP é uma ferramenta altamente especializada na paralelização de regiões paralelas de forma aninhada. Cilk tem apoio em processos de compilação que gera clones rápidos e lentos para a execução dos *threads*, conforme o caso, se eles são migráveis ou não. TBB possui, na sua interface, um extenso conjunto de primitivas que permitem ao programador sinalizar características de sua aplicação para auxiliar no processo de escalonamento. Já a estratégia proposta se baseia, unicamente, na heurística adotada no escalonamento, a interface de programação não permite oferecer nenhuma informação extra ao escalonamento e tudo é realizado em tempo de execução. O benefício é dispor de um mecanismo de um ambiente de execução competitivo em termos de eficiência e, ao mesmo tempo, não limitado em relação ao modelo de paralelismo, como Cilk e OpenMP, nem prolífero em primitivas em sua interface de programação, como TBB.

Além de comprovar, experimentalmente, que a estratégia de escalonamento proposta contribui para o aumento de desempenho do escalonamento de aplicações paralelas em arquiteturas NUMA, este trabalho resultou em uma nova implementação do ambiente Anahy, a qual nos referimos como Anahy-N, por considerar as características das arquiteturas NUMA durante o escalonamento. Anahy-N resulta de uma série de mudanças na estrutura original do ambiente Anahy, o qual foi modificado para comportar a estratégia de escalonamento proposta neste trabalho.

A implementação de Anahy-N, resultante do desenvolvimento deste trabalho,

bem como as aplicações de testes utilizadas no capítulo anterior estão disponíveis para download em: `https://github.com/rmfavaretto/Anahy-N`. A seguir, são definidas algumas continuidades deste trabalho.

6.1 Trabalhos futuros

Futuramente, desejamos realizar mais testes com as aplicações exploradas neste trabalho, buscando variar alguns de seus parâmetros, modificando a granularidade das tarefas produzidas e avaliando o impacto no desempenho, bem como, avaliar a estratégia proposta com outras aplicações que exijam um maior poder computacional.

Pretende-se ainda validar a ferramenta Anahy-N em relação à versão original de Anahy com uma aplicação que utilize intensivamente os processadores da arquitetura *Hydra*, configurando a afinidade dos processadores virtuais da versão original de Anahy manualmente, mas não considerando as prioridades de roubo, ou seja, mantendo a heurística original aleatória.

Como outro trabalho futuro, desejamos evoluir a ferramenta Anahy-N, fazendo com que esta considere, em sua estratégia de escalonamento, informações sobre a taxa de utilização de cada processador e seus *cores*, para decidir entre migrar, ou não, um determinado PV, alterando sua afinidade.

REFERÊNCIAS

ALBERS, S. Better Bounds for Online Scheduling. **SIAM J. Comput.**, Philadelphia, PA, USA, v.29, n.2, p.459–473, Oct. 1999.

ANDERSON, T. E.; BERSHAD, B. N.; LAZOWSKA, E. D.; LEVY, H. M. Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism. In: THIRTEENTH ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 1991, New York, NY, USA. **Proceedings...** ACM, 1991. p.95–109. (SOSP '91).

ANNAVARAM, M.; SHEN, J. Mitigating Amdahl's law through EPI throttling. In: IN PROCEEDINGS OF INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 2005. **Anais...** [S.l.: s.n.], 2005. p.298–309.

BLACK, D. L. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. **Computer**, Los Alamitos, CA, USA, v.23, n.5, p.35–43, May 1990.

BLUMOFE, R. D.; JOERG, C. F.; KUSZMAUL, B. C.; LEISERSON, C. E.; RANDALL, K. H.; ZHOU, Y. Cilk: An Efficient Multithreaded Runtime System. **SIGPLAN Not.**, New York, NY, USA, v.30, n.8, p.207–216, 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.

BREMAUD, P. **Markov Chains**: Gibbs Fields, Monte Carlo Simulation, and Queues. [S.l.]: Springer, 1999. (Texts in Applied Mathematics).

BROQUEDIS, F.; CLET-ORTEGA, J.; MOREAUD, S.; FURMENTO, N.; GOGLIN, B.; MERCIER, G.; THIBAULT, S.; NAMYST, R. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In: EUROMICRO CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCES-

SING, 2010., 2010, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2010. p.180–186. (PDP '10).

BROQUEDIS, F.; FURMENTO, N.; GOGLIN, B.; NAMYST, R.; WACRENIER, P.-A. Dynamic Task and Data Placement over NUMA Architectures: An OpenMP Runtime Perspective. In: INTERNATIONAL WORKSHOP ON OPENMP: EVOLVING OPENMP IN AN AGE OF EXTREME PARALLELISM, 5., 2009, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2009. p.79–92. (IWOMP '09).

CALCIU, I.; DICE, D.; LEV, Y.; LUCHANGCO, V.; MARATHE, V. J.; SHAVIT, N. NUMA-aware Reader-writer Locks. **SIGPLAN Not.**, New York, NY, USA, v.48, n.8, p.157–166, Feb. 2013.

CARíSSIMI, A.; DUPROS, F.; MéHAUT, J.-F.; POLANCZYK, R. V. Aspectos de Programação Paralela em arquiteturas NUMA. In: MINICURSOS DO VIII WSCAD, 2007. **Anais...** [S.l.: s.n.], 2007.

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

CAVALHEIRO, G.; GASPARY, L.; CARDOZO, M.; CORDEIRO, O. Anahy: A Programming Environment for Cluster Computing. In: DAYDé, M.; PALMA, J.; COUTINHO, I.; PACITTI, E.; LOPES, J. (Ed.). **High Performance Computing for Computational Science - VECPAR 2006**. [S.l.]: Springer Berlin Heidelberg, 2007. p.198–211. (Lecture Notes in Computer Science, v.4395).

CHAPMAN, B.; JOST, G.; PAS, R. van der. **Using OpenMP**: Portable Shared Memory Parallel Programming. [S.l.]: MIT Press, 2008. n.v. 10. (Scientific Computation Series).

DING, J.-H.; CHANG, Y.-T.; GUO, Z. dong; LI, K.-C.; CHUN, Y.-C. An efficient and comprehensive scheduler on Asymmetric Multicore Architecture systems. **Journal of Systems Architecture**, [S.l.], v.60, n.3, p.305 – 314, 2014. Real-Time Embedded Software for Multi-Core Platforms.

FEITELSON, D. G.; RUDOLPH, L. Parallel Job Scheduling: Issues and Approaches. In: WORKSHOP ON JOB SCHEDULING STRATEGIES FOR PARALLEL PROCESSING, 1995, London, UK, UK. **Proceedings...** Springer-Verlag, 1995. p.1–18. (IPPS '95).

FENG, H.; MISRA, V.; RUBENSTEIN, D. PBS: A Unified Priority-based Scheduler. **SIGMETRICS Perform. Eval. Rev.**, New York, NY, USA, v.35, n.1, p.203–214, June 2007.

FRIGO, M.; LEISERSON, C. E.; RANDALL, K. H. The Implementation of the Cilk-5 Multithreaded Language. **SIGPLAN Not.**, New York, NY, USA, v.33, n.5, p.212–223, May 1998.

GOGLIN, B.; SQUYRES, J.; THIBAULT, S. Hardware Locality: Peering under the hood of your server. **Linux Pro Magazine**, [S.l.], v.128, p.28–33, July 2011.

GRAHAM, R. **Computer and Job-Shop Scheduling Theory**. [S.l.: s.n.], 1976. p.165?227.

Kalé, L.; KRISHNAN, S. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In: Proceedings of OOPSLA'93, 1993. **Anais...** ACM Press, 1993. p.91–108.

KALE, L. V.; BOHM, E.; MENDES, C. L.; WILMARTH, T.; ZHENG, G. Programming Petascale Applications with Charm++ and AMPI. In: BADER, D. (Ed.). **Petascale Computing**: Algorithms and Applications. [S.l.]: Chapman & Hall / CRC Press, 2008. p.421–441.

KARP, R. M.; ZHANG, Y. Randomized Parallel Algorithms for Backtrack Search and Branch-and-bound Computation. **J. ACM**, New York, NY, USA, v.40, n.3, p.765–789, July 1993.

KAZEMPOUR, V.; FEDOROVA, A.; ALAGHEBAND, P. Performance Implications of Cache Affinity on Multicore Processors. In: EURO-PAR CONFERENCE ON PARALLEL PROCESSING, 14., 2008, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2008. p.151–161. (Euro-Par '08).

KUMAR, R.; TULLSEN, D. M.; RANGANATHAN, P.; JOUPPI, N. P.; FARKAS, K. I. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In: AISCA, 2004., 31., 2004. **Proceedings...** [S.l.: s.n.], 2004. p.64 – 75.

KUPFERSCHMIED, P.; STOESS, J.; BELLOSA, F. **NUMA-aware User-Level Memory Management for Microkernel-Based Operating Systems**. Nuremberg, Germany: [s.n.], 2009.

LILJA, D. J. Exploiting the Parallelism Available in Loops. **Computer**, Los Alamitos, CA, USA, v.27, n.2, p.13–26, Feb. 1994.

NICHOLS, B.; BUTTLAR, D.; FARRELL, J. P. **Pthreads Programming**. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.

PAKIN, S. The Design and Implementation of a Domain-Specific Language for Network Performance Testing. **Parallel and Distributed Systems, IEEE Transactions on**, [S.l.], v.18, n.10, p.1436–1449, 2007.

PANCAKE, C. M. Multithreaded languages for scientific and technical computing. **Proceedings of the IEEE**, [S.l.], v.81, n.2, p.288–304, 1993.

PILLA, L. L.; RIBEIRO, C. P.; COUCHENEY, P.; BROQUEDIS, F.; GAUJAL, B.; NAVAUX, P. O.; MéHAUT, J.-F. A topology-aware load balancing algorithm for clustered hierarchical multi-core machines. **Future Generation Computer Systems**, [S.l.], p.–, 2013.

PILLA, L. L.; RIBEIRO, C. P.; COUCHENEY, P.; BROQUEDIS, F.; GAUJAL, B.; NAVAUX, P. O.; MéHAUT, J.-F. A topology-aware load balancing algorithm for clustered hierarchical multi-core machines. **Future Generation Computer Systems**, [S.l.], v.30, p.191 – 201, 2014. Special Issue on Extreme Scale Parallel Architectures and Systems, Cryptography in Cloud Computing and Recent Advances in Parallel and Distributed Systems, {ICPADS} 2012 Selected Papers.

PILLA, L.; NAVAUX, P.; RIBEIRO, C.; COUCHENEY, P.; BROQUEDIS, F.; GAUJAL, B.; MEHAUT, J. Asymptotically Optimal Load Balancing for Hierarchical Multi-Core Systems. In: PARALLEL AND DISTRIBUTED SYSTEMS (ICPADS), 2012 IEEE 18TH INTERNATIONAL CONFERENCE ON, 2012. **Anais...** [S.l.: s.n.], 2012. p.236–243.

REINDERS, J. **Intel Threading Building Blocks**: Outfitting C++ for Multi-core Processor Parallelism. [S.l.]: O'Reilly Media, 2007.

RIBEIRO, N. S. **Explorando programação Híbrida no contexto de Clusters de Máquina NUMA**. 2011. Dissertação (Mestrado em Ciência da Computação) — (Dissertação), PPG em Ciência da Computação, Pontífica Universidade Católica do Rio Grande do Sul – PUCRS, Porto Alegre, RS.

SARDESAI, S.; MCLAUGHLIN, D.; DASGUPTA, P. **Distributed Cactus Stacks**: Runtime Stack-Sharing Support for Distributed Parallel Programs.

SMITH, T.; WATERMAN, M. Identification of common molecular subsequences. **Journal of Molecular Biology**, [S.l.], v.147, n.1, p.195 – 197, 1981.

SOUZA CAMARGO, C. A. de. **Algoritmos de Escalonamento de Lista em Ambientes Multithread Dinâmicos**: Análise de Estudos de Caso Teóricos e Práticos. 2013. Dissertação (Mestrado em Ciência da Computação) — (Dissertação), Programa de Pós-Graduação em Computação, Universidade Federal de Pelotas – UFPel, Pelotas, RS.

STAELIN, C.; PACKARD, H. lmbench: Portable Tools for Performance Analysis. In: IN USENIX ANNUAL TECHNICAL CONFERENCE, 1996. **Anais...** [S.l.: s.n.], 1996. p.279–294.

TERBOVEN, C.; SCHMIDL, D.; CRAMER, T.; MEY, D. an. Assessing OpenMP Tasking Implementations on NUMA Architectures. In: INTERNATIONAL CONFERENCE ON OPENMP IN A HETEROGENEOUS WORLD, 8., 2012, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2012. p.182–195. (IWOMP'12).

TUKEY, J. **Exploratory Data Analysis**. [S.l.]: Addison-Wesley Publishing Company, 1977. (Addison-Wesley series in behavioral sciences).

ZHENG, G.; BHATELÉ, A.; MENESES, E.; KALÉ, L. V. Periodic Hierarchical Load Balancing for Large Supercomputers. **Int. J. High Perform. Comput. Appl.**, Thousand Oaks, CA, USA, v.25, n.4, p.371–385, Nov. 2011.

APÊNDICE A IMPLEMENTAÇÃO DA ESTRATÉGIA

A seguir são apresentados os códigos fonte da implementação das principais funcionalidade da estratégia proposta.

A.1 Criando as prioridades para roubo

A seguir, no Código 7 é mostrado o código que lê o arquivo da topologia da máquina e cria a lista de prioridades de cada PV.

Código 7: Código que lê o arquivo da topologia gerada por HwLoc.

```
1  ifstream file (TOPOLOGY);
2  if (file) {
3    /* Define o primeiro processador de origem */
4    int proc_origem = 0;
5
6    /* s1: origem, s2: destino e s3: latência */
7    while (file >> s1 >> s2 >> s3) {
8
9      /* Se s1 for diferente de proc_origem, todas as latências partindo
10        deste processador já foram lidas */
11     if (atoi(s1.c_str()) != proc_origem) {
12
13       /* As latências são ordenadas por sort_by_priority */
14       sort(cpus.begin(), cpus.end(), sort_by_priority);
15
16       /* a lista de prioridade deste processador é definida */
17       for (int i=0; i<cpus.size(); i++) {
18         priority.push_back(cpus[i].getIdDestino());
19       }
20
21       /* a lista de prioridades à partir deste processador é criada */
22       proc_priority.push_back(priority);
23
24       /* O processador de origem (s1)  é incrementado */
25       proc_origem++;
```

```
26
27        /* As listas temporárias são limpas */
28        cpus.clear();
29        priority.clear();
30      }
31
32      /* Lista temporária para as latências de cada processador */
33      cpus.push_back(Topology(atoi(s2.c_str()), atof(s3.c_str())));
34    }
35  }
```

O Código 7 está mostrando como o arquivo da topologia da máquina é lido e como as prioridades para roubo são definidas. Em outras palavras, a execução deste código cria uma lista que contém, para cada processador, a identificação dos processadores reais da máquina ordenados da menor para a maior latência de acesso para, posteriormente, quando um PV é criado, poder definir uma lista de prioridades de roubo para este PV, de acordo com os processadores onde os demais PVs estão alocados.

Como se conhece as latências entre todos os processadores presentes na arquitetura da máquina, como base na afinidade de cada PV a um destes processadores, é possível que cada PV mantenha a sua lista de prioridades para roubo. A criação e atribuição desta lista de prioridades é realizada segundo é mostrado no Código 8, a seguir.

Código 8: Criação da lista de prioridade de roubo de cada PV.

```
1  /* Vetor temporário para as prioridades de cada PV */
2  vector<int> temp_priority;
3
4  /* Variável que controla os processadores */
5  int id_proc=0;
6
7  /* Laço que cria as prioridades de cada PV */
8  for (int i=0; i<num_vps; i++) {
9
10   /* Laço que percorre as prioridades dos processadores */
11   for (int k=0; k<priority.size(); k++) {
12
13     /* Laço que compara o ID do processador onde cada PV está
14        executando com as prioridades de roubo de cada processador
15        e cria a lista de prioridades de cada PV */
16     for(int r=num_vps-1; r>=0; r--) {
17       if(vp_array[r]->get_cpu_id() == proc_priority[id_proc][k]) {
18         temp_priority.push_back(r);
19       }
```

```
20      }
21    }
22    id_proc++;
23
24    /* Variável que controla os processadores é reiniciada */
25    if (id_proc == num_cpus)
26        id_proc=0;
27
28    /* Atribui ao PV sua lista de prioridades para roubo,
29       nesta lista estão todos os demais PVs ordenados
30       pelo menor custo para efetuar o roubo */
31    vp_array[i]->set_vps_to_steal(temp_priority);
32
33    /* A lista temporária é limpa para a próxima iteração do laço */
34    temp_priority.clear();
35  }
```

Quando um PV é criado e posto para executar, todas as operações de roubo de trabalho são realizadas observando-se a ordem de prioridades definida na lista `vps_to_steal` de cada PV. A seção a seguir mostra como são criados os PVs e como é atribuída a afinidade de cada um deles.

No Código 9 é apresentada a estrutura que define um processador virtual. A variável `cpu_id` armazena o ID do processador físico, o qual é utilizado para setar a afinidade do PV. O vetor `vps_to_steal` mantém a lista de PVs ordenados pela prioridade para roubo de cada PV. O acesso de um PV aos demais processadores virtuais é realizado a partir da lista `vp_list`.

Código 9: Estrutura que define um Processador Virtual.

```
1  class VirtualProcessor {
2    int id;
3    long job_counter;
4
5    /* ID do processador onde o PV está executando */
6    int cpu_id;
7
8    /* Lista de prioridades para roubo */
9    vector<int> vps_to_steal;
10
11   AnahySmartHeap<AnahyJob*> job_list;
12   AnahyJob* current_job;
13   AnahySmartStack<AnahyJob*> context_stack;
14
15   /* Lista de VPs - Somente leitura */
16   static list<VirtualProcessor*> vp_list;
17
```

```
18    /* Quantidade de PVs sem trabalho */
19    static int idle_vps;
20
21    /* Sub-rotina própria de execução do PV */
22    static void* call_vp_run(void *vp_obj);
23  }
```

Uma vez criados, os PVs são distribuídos sobre os processadores físicos da arquitetura da máquina e, conforme visto na Seção 4.4, esta distribuição segue uma política circular, o que busca manter balanceados os recursos de processamento. Para distribuir os PVs entre os processadores físicos, é necessário informar a este PV sobre qual ou quais processadores ele pode executar. Isso é realizado por meio de máscaras de afinidade. A seguir é descrito como os PVs são criados e distribuídos entre os processadores da arquitetura.

A.1.1 Criando os PVs

A quantidade de PVs que irão executar a aplicação deve ser informada pelo usuário no momento em que for executar a aplicação (`-v (int)`). Este argumento é opcional e, caso não seja passado, apenas um (1) PV será criado para executar a aplicação. O Código 10 apresenta como está implementada a rotina que cria os PVs.

Código 10: Criando um array de Processadores Virtuais.

```
1  /* Variável que controla a distribuição dos PVs */
2  affinity_control = 0;
3
4  /* Alocando um array de PVs */
5  vp_array=(VirtualProcessor**)malloc(num_vps*sizeof(VirtualProcessor*));
6
7  for(int i = 0; i < num_vps; i++) {
8
9    /* Cria um objeto PV */
10   vp_array[i] = new VirtualProcessor();
11
12   /* Informa ao PV sua afinidade */
13   vp_array[i]->set_cpu_id(affinity_control);
14
15   /* Incrementa o controle de afinidades */
16   affinity_control++;
17
18   /* Caso hajam mais PVs que processadores */
19   if (affinity_control == num_cpus) {
20     affinity_control = 0;
```

```
21    }
22  }
23
24  /* Aloca um array de threads que executarão os PVs */
25  thread_array = (pthread_t*)malloc((num_vps-1)*sizeof(pthread_t));
26
27  /* Cria e lança os PVs para execução */
28  for (int i = 0; i < num_vps; i++) {
29    pthread_create(&thread_array[i], NULL, call_vp_run, vp_array[i]);
30  }
```

Um PV tem sua execução iniciada logo após sua criação, ou seja, a execução do programa paralelo tem início antes mesmo da criação de todos os VPs. Para tal é chamado o método `call_vp_run`, o qual também seta a afinidade do PV, conforme é descrito a seguir.

A.1.2 Setando a afinidade

Muitas vezes, quando se deseja balancear a quantidade de trabalho entre os processadores, o sistema operacional pode migrar *threads* de um processador para outro e, se a decisão de quando migrar não for bem concebida, ocorrerão muitas migrações, o que pode se tornar um gargalo de desempenho no sistema, pois a migração de *threads* gera trocas de contexto nos processadores, isso significa, parar a execução de um *thread* e dar início à execução de outro. Nesta operação é gasto tempo de processamento efetivo, o qual é relevante à evolução do programa.

É possível indicar a afinidade do PV à arquitetura física, permitindo que cada PV possa ser atribuído a um processador específico na arquitetura da máquina. Atribuir afinidade a um processador consiste em atribuir uma máscara de mapeamento a um *thread*. Essa máscara é composta por um conjunto de *bits* que indicam sobre qual(is) processador(es) um determinado *thread* pode executar. Uma vez que esta máscara é atribuída a um *thread*, todos em sua hierarquia a terão. Desta forma, todos os *threads* criados a partir deste *thread* executarão sobre o mesmo processador, priorizando assim a localidade de dados na memória *cache*.

Para atribuir a afinidade dos PVs, utiliza-se as primitivas segundo o padrão *PThreads* (NICHOLS; BUTTLAR; FARRELL, 1996), conforme mostrado no Código 11, a seguir.

Código 11: Setando a afinidade dos processadores.

```
1  void* AnahyVM::call_vp_run(void* vp_obj) {
2    /* Variável que contém a máscaro do thread */
```

```
3     cpu_set_t mask;
4
5     /* Limpa a afinidade padrão do thread */
6     CPU_ZERO(&mask);
7
8     /* Obtém a afinidade do PV e a define na máscara */
9     CPU_SET(((VirtualProcessor*)vp_obj)->get_cpu_id(), &mask);
10
11    /* Seta a afinidade do VP */
12    pthread_setaffinity_np(pthread_self(), sizeof(&mask), &mask);
13
14    /* Atribui uma chave ao thread */
15    pthread_setspecific(key, vp_obj);
16
17    /* Lança o PV para execução */
18    ((VirtualProcessor*)vp_obj)->run();
19    return NULL;
20 }
```

A função `pthread_setaffinity_np` gera uma chamada de sistema que atualiza a afinidade atual do *thread* em `pthread_self()` para a nova máscara de afinidade contida em `mask`. Quando esta primitiva não é especificada, o padrão de afinidade é atribuído na criação do *thread* e determina que este possa ser executado em qualquer core. Isso deixa a cargo do sistema operacional decidir sobre qual processador o *thread* irá executar.

A.2 Realizando o escalonamento

O escalonamento realizado nesta estratégia é baseado no roubo de tarefas (*work-stealing*), onde há uma lista global de trabalhos para serem executados. Na implementação realizada, a lista é mantida de maneira distribuída, ou seja, cada PV possui uma parte desta lista localmente e possui acesso as listas dos outros PVs.

A lista é implementada como uma fila de duplo fim, sendo o topo manipulado pelo PV que detêm a propriedade da fila, para fins de inserção para novos *threads* ou retirada para execução local, e eventuais roubos são realizados na cauda. Entretanto, quando esta fila local ficar vazia, o PV torna-se um ladrão e, então, escolhe um outro PV (vítima) para realizar um roubo de trabalho. Essa ideia está representada na Figura 47.

O procedimento de escalonamento ocorre da seguinte maneira: inicialmente, o PV busca uma tarefa em sua lista local e, caso encontre, começa a executá-la. Caso contrário, o PV busca uma tarefa na lista de outros PVs e, quando encontrar um PV que contenha tarefas prontas para serem executadas em sua lista, rouba

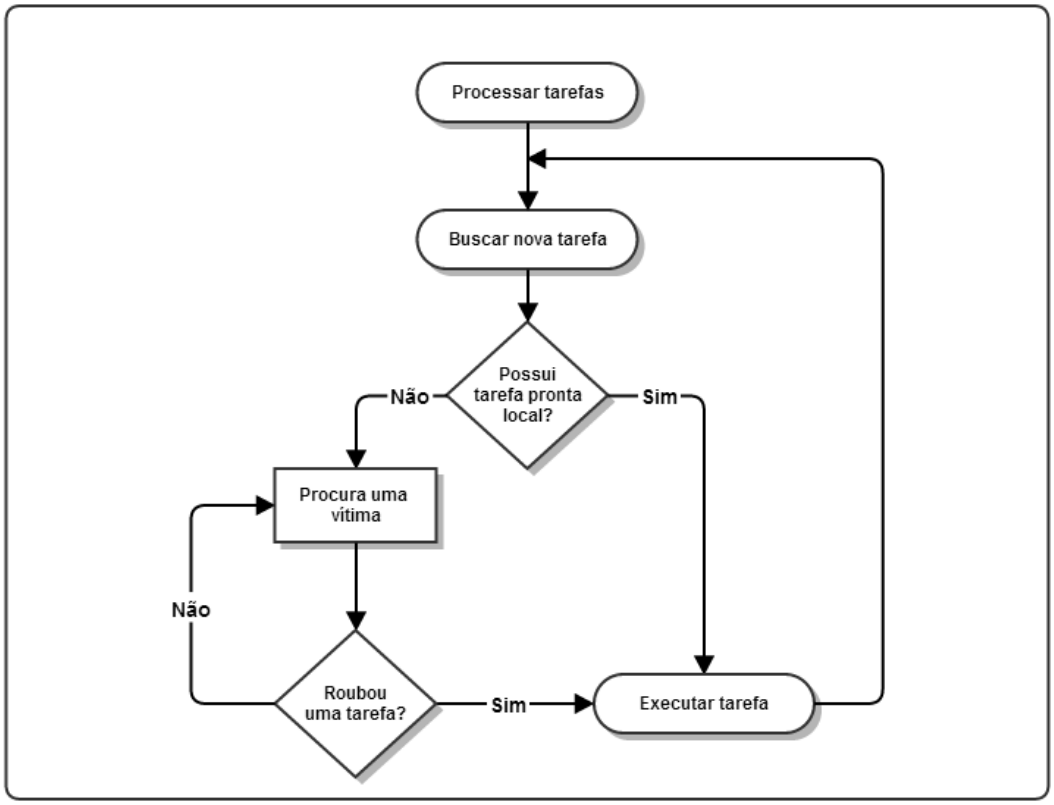


Figura 47: Fluxograma de execução das tarefas.

a tarefa que for mais antiga, ou seja, a tarefa que estiver no final da fila.

O motivo pelo qual um PV rouba a tarefa mais antiga criada na fila do VP vítima está relacionado à heurística que considera que as tarefas mais próximas ao início da execução possuem maior probabilidade de gerar maiores cargas de trabalho, o que irá manter o PV ladrão ocupado por maior tempo. Se o programa tiver uma estrutura aninhada de paralelismo, é o caso extremo que ilustra bem o caso de sucesso desta estratégia, pois, o último *thread* a ser criado é o próximo a ser sincronizado, o que justifica um PV executar o *thread* mais recente criado em sua lista local.

A escolha do PV vítima, se dá pela lista de prioridades de roubo de cada PV, apresentada anteriormente. A ideia é tentar roubar, primeiramente, dos PVs que estão fisicamente mais próximos, o que reduz o tempo gasto com a operação de roubo. O Código 12 define o núcleo de execução de um PV. Como cada PV possui um *thread* associado, a busca por um novo trabalho fica dentro de um laço (*loop*). Desta maneira, o PV fica constantemente tentando obter um trabalho para executar.

Código 12: Laço principal responsável por executar as tarefas.

```
1 while(1) {
2   /* Busca tarefas na lista local */
3   current_job = this->get_job();
```

```
4
5     /* Caso não tiver tarefas na lista local do PV
6        o PV deve, então, tentar roubar trabalho de outros PVs */
7     if (!current_job) {
8
9       /* Incrementa atomicamente o número de PVs ladrões */
10      int __idle_vps = __sync_add_and_fetch(&idle_vps, 1);
11
12      /* Condição de parada do programa */
13      if (__idle_vps == num_vps) {
14        break;
15      }
16
17      /* Realiza o roubo de trabalho */
18      for (it = vps_to_steal.begin(); it != vps_to_steal.end(); ++it) {
19        if (*it != this) {
20          current_job = (*it)->steal_job();
21          if (current_job) {
22            break;
23          }
24        }
25      }
26
27      /* Decrementa atomicamente o número de PVs ladrões */
28      __sync_sub_and_fetch(&idle_vps, 1);
29    }
30
31    /* Executa se há tarefa disponível */
32    if (current_job) {
33      current_job->run();
34    }
35  }
```

Operações atômicas são utilizadas para incrementar (`__sync_add_and_fetch`, linha 9) e decrementar (`__sync_sub_and_fetch`, linha 27) o número de PVs ladrões. Isso evita o uso de *locks*, variáveis de condição ou sinalizações para controlar a condição de parada dos PVs do ambiente.