UNIVERSIDADE FEDERAL DO PAMPA

Gustavo Paim Berned

Hórus: Reduzindo o custo de aprendizado das estratégias de otimização *offline* para Aplicações Paralelas

Alegrete
2021

Gustavo Paim Berned

Hórus: Reduzindo o custo de aprendizado das estratégias de otimização *offline* para Aplicações Paralelas

Proposta de Dissertação submetida ao Programa de Pós-graduação em Engenharia de Software da Universidade Federal do Pampa, como requisito parcial para a obtenção do título de Mestre em Engenharia de Software.

Orientador: Arthur Francisco Lorenzon

Alegrete
2021

SERVIÇO PÚBLICO FEDERAL

MINISTÉRIO DA EDUCAÇÃO

Universidade Federal do Pampa

GUSTAVO PAIM BERNED

Hórus: Reduzindo o custo de aprendizado das estratégias de otimização offline para Aplicações Paralelas

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia de Software da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Mestre em Engenharia de Software.

Dissertação defendida e aprovada em: 20, maio de 2021.

Banca examinadora:

_____

Prof. Dr. Arthur Francisco Lorenzon

Orientador

UNIPAMPA

_____

Prof. Dr. Marcelo Caggiani Luizelli

Prof. Dr. Fábio Diniz Rossi

IFFar - Alegrete

Assinado eletronicamente por **ARTHUR FRANCISCO LORENZON**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 20/05/2021, às 15:46, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **MARCELO CAGGIANI LUIZELLI**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 20/05/2021, às 16:11, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **Fábio Diniz Rossi**, **Usuário Externo**, em 20/05/2021, às 20:48, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

A autenticidade deste documento pode ser conferida no site https://sei.unipampa.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **0531266** e o código CRC **5FF7E743**.

"Eduquem as crianças, para que não
seja necessário punir os adultos"
(Pitágoras)

RESUMO

Diversas estratégias de otimização *online* e *offline* têm sido empregadas para melhorar o desempenho e consumo de energia de aplicações paralelas. Enquanto o primeiro conjunto de estratégias podem obter valores de métricas que só podem ser conhecidas em tempo de execução, o segundo (*Offline*) não impõe nenhuma sobrecarga durante a execução e pode utilizar algoritmos mais eficientes. No entanto, os algoritmos de aprendizagem das estratégias *offline* podem levar várias horas para encontrar uma solução, impedindo seu uso ou a portabilidade entre diferentes sistemas. Neste cenário, esta dissertação propõe uma metodologia para diminuir o tempo de aprendizado de estratégias *offline*, inferindo o comportamento da execução de aplicações paralelas utilizando conjuntos de entrada menores do que os utilizados pelas aplicações alvo. Duas estratégias de otimização são implementadas: SEA, onde todas as regiões paralelas de uma aplicação são executadas com o mesmo número de *threads*; e SPRA, que busca encontrar um número ideal de threads para cada região paralela de uma determinada aplicação. Com um extenso conjunto de experimentos, mostramos que as estratégias SEA e SPRA convergem para resultados próximos de uma abordagem *offline* aplicada sobre a entrada regular, mas sendo 88% e 87% mais rápidas, em média, respectivamente. Também mostrou-se que SPRA é melhor que SEA para aplicações desbalanceadas.

Palavras-chave: Computação Paralela. Otimização de Sistemas em Tempo de Execução. Paralelismo em Nível de *Threads*. *Energy-delay Product*.

ABSTRACT

Several online and offline optimization strategies have been employed to improve the performance and energy consumption of parallel applications. While the first strategy can achieve some behaviors that can only be known at run time, the last one does not impose any overhead during execution and can use more complex and efficient algorithms. However, the offline strategy learning algorithm can take several hours, preventing its use or smooth portability between different systems. In this scenario, this dissertation proposes a methodology to decrease the learning time of offline strategies, inferring the execution behavior of parallel applications using smaller input sets than those used by the target applications. Two optimization strategies are implemented: SEA, where all the parallel regions of an application are executed with the same number of threads; and SPRA, which seeks to find an ideal number of threads for each parallel region of a given application. With an extensive set of experiments, we show that the SEA and SPRA strategies converge to results close to an offline approach applied to regular entry, but being 88% and 87% faster, on average, respectively. We have also shown that SPRA is better than SEA for unbalanced applications.

Key-words: Parallel Computing. Runtime Optimization Systems. Thread-level Parallelism Exploitation. Energy-delay Product.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

CPU *Central Process Unit*

DVFS *Dynamic Voltage and Frequency Scaling*

EDP *Energy Delay Product*

FFT *Fast Fourier Transform*

HPC *High Performance Computing*

ILP *Instruction Level Parallelism*

IPC Instruções por Ciclo

IPP Interface de Programação Paralela

MPI *Message Passing Interface*

RAPL *Running Average Power Limit*

SMT *Simultaneous Multithreading*

TDP *Thermal Design Power*

TLP *Thread Level Parallelism*

ULA Unidade Lógica Aritmética

SUMÁRIO

1 INTRODUÇÃO

Nas últimas décadas, computadores compostos de múltiplas unidades de processamento (e.g., núcleos) têm se tornado onipresentes em sistemas de propósito geral (*desktops* e *laptops*), embarcados (*smartphones* e *tablets*) e em servidores de alto desempenho (STALLINGS, 2018). Tais sistemas multiprocessados (ou simplesmente processadores *multicore*) surgiram em meados de 2001 devido a uma limitação térmica dos processadores com um único núcleo de processamento (*single core*) (STALLINGS, 2018). A indústria então, alterou o modo como os processadores estavam sendo produzidos através da incorporação de mais núcleos de processamento em apenas um chip e da diminuição da frequência de operação em cada um destes núcleos.

Neste sentido, diferentes abordagens têm sido adotadas para melhorar o uso dos recursos computacionais (e.g., cache, *Central Process Unit* (CPU)), tais como, exploração do paralelismo no nível de instrução (*Instruction Level Parallelism* (ILP)) e no nível de *threads* (*Thread Level Parallelism* (TLP)). Um exemplo típico de ILP são as arquiteturas superescalares, na qual vários *pipelines* de instruções independentes são utilizados, em que cada instrução (e.g., *load*, *store*) de um programa pode ser executada simultaneamente (STALLINGS, 2018). Já o paralelismo ao nível de *threads*, explora o paralelismo no fluxo de um conjunto de instruções. Isso significa ser possível executar mais de uma *thread* no mesmo instante de tempo (PAS et al., 2017).

Portanto, desenvolvedores de *software* passaram a explorar o paralelismo no nível de *threads* para melhorar o desempenho (e.g., diminuir o tempo de execução da aplicação) de aplicações de diferentes domínios, como, por exemplo, medicina, previsão do tempo, dinâmica de fluidos computacional, *machine learning*, entre outros. Porém, apesar da exploração de TLP oferecer um aumento significativo no desempenho, o consumo de energia se tornou uma questão urgente: enquanto se espera que o consumo de energia dos sistemas de *High Performance Computing* (HPC)(Computação de Alto Desempenho) cresça significativamente (até 100 MW nos próximos anos (DUTOT et al., 2017)), processadores de uso geral estão tendo o desempenho limitado pelo *Thermal Design Power* (TDP). Tendo em vista este alto consumo de energia no qual os processadores podem gerar, é preciso também atender a questões legais para se evitar o alto consumo de energia de sistemas de HPC, tais como, o protocolo de Kyoto (2005) e o acordo de Paris (2015). Portanto, ao executar aplicações paralelas que realizam computação sobre quantidades massivas de dados, o objetivo não é apenas melhorar o desempenho, mas fazer isso com o menor consumo de energia possível.

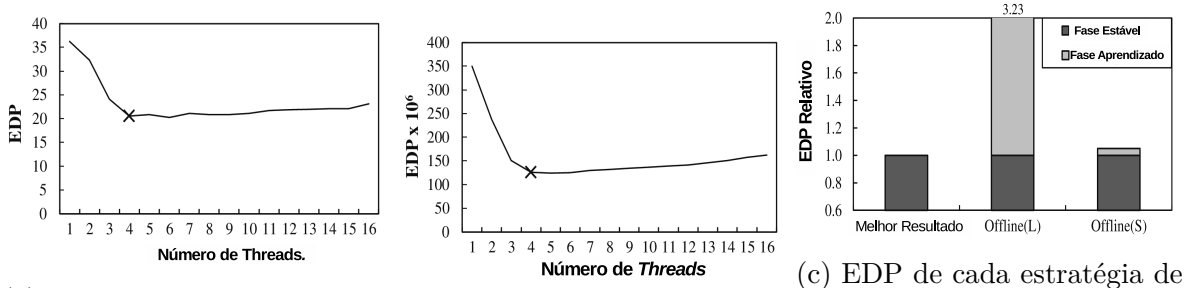
Quando TLP é explorado, a forma padrão de executar uma aplicação paralela consiste em utilizar o máximo dos recursos disponíveis no sistema (e.g., núcleos de processamento e caches). Embora esta estratégia possa fornecer o melhor desempenho, executar uma aplicação com o número de *threads* igual ao número de núcleos disponíveis nem sempre resulta no melhor custo benefício entre tempo de execução e o consumo de ener-

gia, representado pelo *Energy Delay Product* (EDP) (BERNED et al., 2020). As razões para isto se dão por conta de limitações de *hardware* e *software*: saturação do barramento *off-chip*, sincronização entre às *threads*, acesso concorrente à memória compartilhada (LORENZON et al., 2019).

Para explorar melhor o uso do TLP, diversas abordagens têm sido utilizadas para encontrar um número de *threads* que otimize uma aplicação paralela não apenas em desempenho mas também em consumo de energia. Tais estratégias são compostas principalmente por duas fases: fase de aprendizagem e fase estável (LORENZON; BECK, 2019) · Durante o estágio de aprendizado, um algoritmo de busca é aplicado para encontrar o número ideal de *threads* para uma determinada aplicação considerando um determinado requisito. Este processo pode ser *online*, ou seja, enquanto a aplicação é executada, ou *offline*, antes da execução da aplicação. Após o término dessa fase, ou seja, a abordagem escolhida (*offline* ou *online*) convergiu para um determinado número de *threads*, a fase estável é ativada e a aplicação é executada com essa configuração até o seu final. Esta configuração pode ser dada por um número fixo de *threads* que serão utilizadas durante toda a execução; ou por um conjunto que compreende um número diferente de *threads* para cada região paralela particular da aplicação. Mesmo que as estratégias *online* possam se adaptar melhor às mudanças no ambiente de execução e parâmetros da aplicação que só são conhecidos em tempo de execução, eles podem adicionar uma sobrecarga significativa à operação da aplicação. Em alguns casos, essa sobrecarga se sobrepõe aos ganhos obtidos pela adaptabilidade extra oferecido (SRIDHARAN; GUPTA; SOHI, 2014).

Por outro lado, as estratégias *offline* podem usar algoritmos mais complexos e sofisticados para tentar tantas configurações quanto possível para a mesma aplicação sem incorrer em qualquer sobrecarga para a execução da aplicação, mas não podem lidar com as alterações no ambiente de execução. Portanto, quanto mais configurações forem testadas, mais longa será a fase de treinamento (BERNED et al., 2020). No entanto, muitas aplicações podem apresentar o mesmo comportamento em exploração de TLP quando diferentes conjuntos de entrada são considerados. Nesse cenário, o custo da fase de aprendizado, ou seja, o tempo gasto na convergência para uma solução, que é a principal limitação das estratégias *offline*, pode ser minimizado. Para melhor entendê-lo, vamos considerar o comportamento EDP do *benchmark* CG-NAS com dois conjuntos de entradas (de nossos experimentos, descritos na Seção 5.1) em um sistema AMD de 16 núcleos.

A Figura 1a mostra o EDP para a execução com um número diferente de *threads* (de 1 a 16) sobre uma entrada pequena (Menor que 2% da grande), enquanto a Figura 1b considera a execução sobre uma entrada grande (Entrada padrão da aplicação). Como se pode observar, a curva de EDP que varia em função do número de *threads* ativas é muito semelhante para ambas as entradas: o melhor EDP (menor é melhor) é obtido com quatro *threads*. Portanto, em vez de aplicar uma estratégia *offline* utilizando o conjunto de entrada grande, pode-se usar o pequeno conjunto de entrada para acelerar o processo



(a) CG-NAS: Entrada Pequena

(b) CG-NAS: Entrada Grande

(c) EDP de cada estratégia de busca normalizado em relação ao melhor resultado

Figura 1 – Comportamento de EDP do *benchmark* CG-NAS em um AMD 16-*cores*

de aprendizagem, cuja configuração resultante (fornecida em número de *threads*) pode ser usada posteriormente para executar a mesma aplicação utilizando o conjunto de entrada maior.

Entretanto, a principal vantagem em utilizar uma pequena entrada durante a fase de treinamento é a redução significativa do custo de aprendizado como mostrado na Figura 1c. A Figura apresenta o EDP da execução do CG-NAS com três estratégias distintas: melhor resultado - o melhor EDP possível, encontrado através da execução da aplicação com a entrada grande ao longo de todas as *threads*. *Offline* (L) - o EDP (aprendizagem + fase estável) para um algoritmo de pesquisa (por exemplo, um método *hill-climbing*) aplicado sobre a execução com o grande conjunto de entrada. E *Offline* (S) - o EDP quando o mesmo algoritmo de pesquisa é aplicado sobre a execução com a entrada pequena e o número de *threads* encontrado durante a busca é definido para executar a aplicação com o conjunto de entrada grande. Como pode ser observado, utilizando a estratégia *Offline*(S), o custo de aprendizado é reduzido significantemente para apenas 4%.

1.1 Objetivos

Diante disso, este trabalho propõe Hórus, um *framework* que utiliza uma metodologia genérica para diminuir significativamente os custos da fase de aprendizado para encontrar um número ideal de *threads*, inferindo o comportamento de execução de aplicações paralelas utilizando conjuntos de entrada menores que o original. Para isso, foram propostas duas estratégias de busca diferentes aplicadas sobre o pequeno conjunto de entrada, que podem ser definidas pelo desenvolvedor de software para serem aplicadas durante a fase de aprendizagem:

- **SEA**: O algoritmo de pesquisa é aplicado em uma granularidade mais grossa sobre a aplicação. Ou seja, visa encontrar um número ideal de *threads* para uma dada aplicação paralela, considerando que todas as regiões paralelas executam com o mesmo grau de TLP. Quando esta estratégia é empregada, o número de *threads*

é definido no início da execução e não muda. Por isso, cada etapa do algoritmo de busca (ou seja, a execução com uma configuração diferente enquanto converge para uma solução ideal) considera a execução inteira da aplicação com o pequeno conjunto de entrada. Visto que a busca converge para uma solução ideal, a aplicação é executada com a entrada grande definida com o número de *threads* encontrado durante a busca.

- **SPRA**: O algoritmo de busca é aplicado em uma granularidade mais fina. Ou seja, como muitas aplicações paralelas possuem mais de uma região paralela e cada uma pode ser melhor executada com um número diferente de *threads*, esta estratégia visa encontrar um grau ideal de TLP para cada região de uma determinada aplicação. Portanto, o algoritmo de aprendizagem é aplicado a cada região paralela conforme a aplicação é executada com a entrada pequena. Dado que a busca converge para uma solução ideal, a saída da fase de aprendizado é um conjunto com o melhor grau de TLP para cada região paralela da aplicação. Em seguida, a aplicação é executado com o conjunto de entrada grande, aplicando o número de *threads* encontrados durante a busca para cada região.

Ambos os algoritmos de busca foram implementados em uma estrutura que recebe do usuário um binário da aplicação a ser otimizada, dois conjuntos de entrada (pequeno e grande) e a estratégia de busca que deve ser empregada (SEA ou SPRA). Então, o algoritmo de aprendizagem encontra automaticamente uma configuração ideal para cada estratégia de busca e, em seguida, usa essa configuração para executar a aplicação com a grande entrada definida. O *framework* também contém um banco de dados que armazena as configurações encontradas pela fase de aprendizado para cada conjunto binário e de entrada da aplicação. Portanto, quando a aplicação é reexecutada com os mesmos parâmetros, não é necessário passar pela fase de aprendizado novamente.

1.2 Organização do Texto

O restante desta dissertação está organizada da seguinte maneira: no Capítulo 2 serão descritos os elementos que fazem parte da fundamentação teórica deste trabalho de pesquisa. O Capítulo 3 contém os trabalhos relacionados que auxiliaram na elaboração da proposta deste trabalho. O *framework* Hórus para melhoria de EDP para aplicações paralelas, foco deste trabalho, será apresentado no Capítulo 4. O Capítulo 5 contém a metodologia empregada para a realização dos experimentos, tais como, os *benchmarks* utilizados, ambiente de execução e quais cenários comparativos serão realizados para demonstrar a eficácia do Hórus. No Capítulo 6, serão discutidos os resultados comparativos apresentados no Capítulo 5, de modo que serão analisados os resultados de EDP obtidos por cada uma das estratégias, logo após, a convergência dos algoritmos empregados em SEA e SPRA e como uma última análise, serão apresentados os comparativos de EDP

encontrado por cada uma das estratégias empregada por Hórus em relação a melhor solução possível. Por fim, no Capítulo 7, constam as conclusões deste trabalho e os trabalhos futuros que devem estender Hórus.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os conceitos fundamentais que estão envolvidos neste trabalho. Na Seção 2.1, são apresentados os conceitos básicos sobre arquiteturas *multicore*. Na Seção 2.2, discute-se sobre a Interface de Programação Paralela (IPP) OpenMP para programação paralela em ambientes de memória compartilhada. Por fim, na Seção 2.3, são descritos os métodos de otimização.

2.1 Arquiteturas *Multicore*

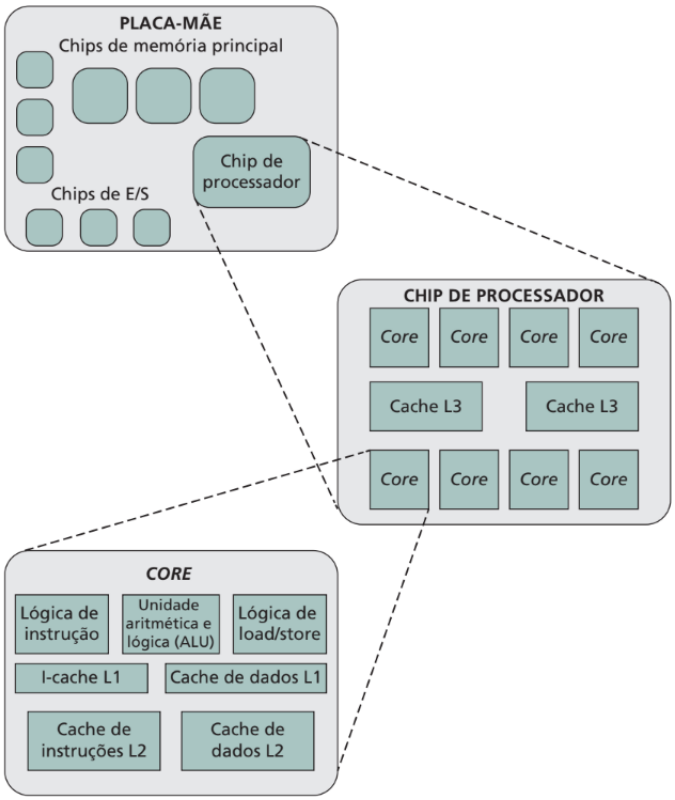
Até meados de 2004, a principal maneira de se obter ganhos significativos de desempenho em processadores se dava de duas maneiras: aumentar a frequência de operação dos processadores e/ou diminuir a densidade lógica (transistores cada vez menores)(STALLINGS, 2018). Entretanto, conforme estas duas estratégias eram utilizadas, diversos fatores tornavam este processo mais difícil de ser aproveitado, tendo como um dos principais fatores limitantes, a potência consumida. À medida que a densidade da lógica e a velocidade do *clock* em um chip aumentam, também aumenta a densidade de potência (Watts/cm^2). Portanto, a dificuldade em dissipar o calor gerado em chips de alta densidade e alta velocidade estava se tornando um sério problema de projeto (GIBBS, 2004).

Com base nesta dificuldade em aumentar o desempenho dos processadores, os desenvolvedores recorreram a uma técnica fundamentalmente nova: colocar diversos processadores no chip, com uma grande memória cache compartilhada. O uso de diversos processadores em um único chip, também chamado de múltiplos núcleos (*core*) ou *multicore*, proporciona o potencial para aumentar o desempenho sem aumentar a frequência do *clock* (BORKAR, 2003). Além disso se o *software* for projetado para usufruir de maneira efetiva dos múltiplos núcleos (Programação Paralela), duplicar o número de núcleos pode quase duplicar o desempenho do *software*. Desta forma, a estratégia foi usar dois ou mais processadores simplificados no chip em vez de um processador mais complexo. (STALLINGS, 2018).

Um processador *multicore*, também conhecido como chip multiprocessador (Figura 2), combina duas ou mais unidades de processamento (chamadas de *cores*) em uma peça única de silício. Em geral, cada *core* consiste em todos os componentes de um processador independente, como registradores, Unidade Lógica Aritmética (ULA), hardware de *pipeline* e unidade de controle, mais caches L1 de dados e de instruções. Além de vários cores, os chips *multicore* atuais incluem também cache L2 e, em alguns casos, cache L3 (STALLINGS, 2018). De forma resumida, os registradores são um conjunto de pequenas memórias situadas dentro de cada *core*, exercendo duas funções principais: permitir ao programador de linguagem de máquina minimizar acessos diretos à memória, sendo estes registradores visíveis ao usuário e os registradores utilizados pela unidade de controle, que possuem a finalidade de controlar a operação do processador, sendo estes últimos,

registradores de controle de estado.

Figura 2 – Exemplo de Arquitetura *Multicore*.

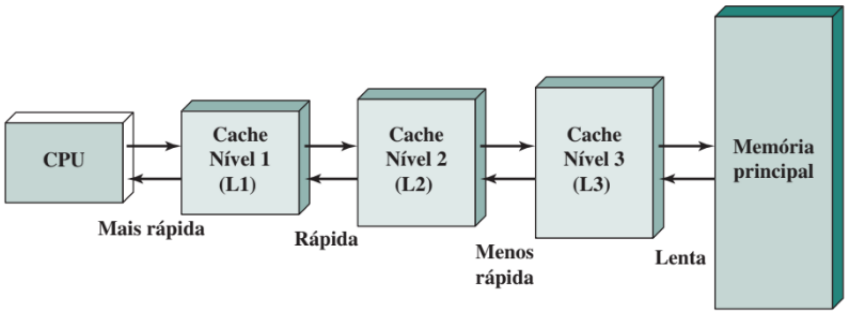


Adaptado de: (STALLINGS, 2018)

A ULA, é parte fundamental de um processador, responsável por realizar toda a lógica e aritmética sobre os dados. Todos os componentes que englobam um sistema de computação, enviam dados para serem computados por ela e depois retornam os resultados de volta para quem os solicitou. O pipeline de instruções consiste em uma série de operações a serem realizadas pelo processador sendo estas: buscar instrução, decodificar a instrução, buscar os operandos, realizar os cálculos e salvar os resultados. Além disso, em arquiteturas superescalares, é possível executar mais de uma instrução por vez, fazendo-se assim uso de paralelismo ao nível de instrução (ILP).

A cache é uma memória intermediária entre o processador e a memória principal (Figura 3) e geralmente possui até três níveis. As memórias caches L1 e L2 podem armazenar dados e instruções, já a cache L3, armazena somente dados. Quando o processador tenta ler uma palavra (dados) da memória, é feita uma verificação para determinar se a palavra está na cache. Se estiver, ela é entregue ao processador. Se não, um bloco da memória principal, consistindo em algum número fixo de palavras, é transferido para a cache, e depois a palavra é fornecida ao processador. Em virtude do fenômeno da localidade de referência, quando um bloco de dados (palavras) é carregado para a cache para satisfazer uma única referência de memória, é provável que haja referências futuras a esse mesmo local da memória ou a outras palavras no mesmo bloco (STALLINGS, 2018).

Figura 3 – Organização de cache de três níveis.

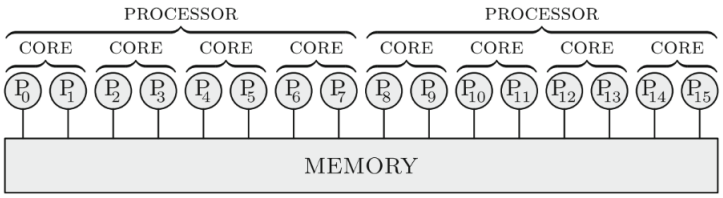


Adaptado de: (STALLINGS, 2018).

A Figura 3 mostra que, quanto mais próxima à memória cache está da CPU, mais rápido é o acesso às informações. Sendo assim, a cache L1 possui acesso mais rápido que acessos à cache L2, que por sua vez, possui acessos mais rápidos que à cache L3. Com esta memória mais próxima aos *cores*, evita-se acessos contínuos à memória principal, melhorando o desempenho do sistema computacional.

Além destes componentes descritos acima, existe um outro componente presente em processadores *multicore* atuais, denominado *Simultaneous Multithreading* (SMT). O SMT permite cada *core* executar múltiplos fluxos independentes chamados de *threads*. Para um programador, cada *core* dentro de cada processador, atua com diversas *threads* aptas a executarem suas próprias tarefas (TROBEC et al., 2018). A Figura 4 ilustra um sistema contendo dois processadores *multicore* e cada processador contém quatro *cores* distintos. Como este processador possui suporte a SMT, cada *core* pode executar 2 *threads* de maneira simultânea. Independentemente de como estão organizados fisicamente os *cores*, programadores podem assumir que seu sistema possui ao total 16 *cores*.

Figura 4 – Dois Processadores *quad-core* com suporte a SMT.



Fonte: (TROBEC et al., 2018)

Sendo assim, é possível explorar o paralelismo através de *threads*. O TLP provê paralelismo através da execução simultânea de diferentes *threads*, provendo uma granularidade mais grossa de paralelismo do que ILP (TROBEC et al., 2018). Desta forma, TLP consegue usufruir tanto das diversas unidades de processamento (*cores*) e cada *core* ainda consegue usufruir de ILP.

2.2 Programação Paralela

Programação paralela de uma forma geral, consiste na divisão das tarefas de uma aplicação entre os vários núcleos do processador, com o objetivo principal de reduzir o tempo de execução. Existem muitas aplicações que demandam muita carga de processamento e utilizam a computação paralela, como, por exemplo: cálculos de sequência de DNA, previsão do tempo, análise financeira, dinâmica de fluidos computacional entre outras aplicações.

Entretanto, construir aplicações paralelas depende fortemente do problema a ser resolvido e a arquitetura de computação paralela alvo. Esta situação fez com que vários fabricantes de sistemas de computação paralela desenvolvessem suas próprias IPPs (Interfaces de Programação Paralela), focando em características, muitas vezes, inexistentes em outras arquiteturas. Desta forma, o código desenvolvido com base nestas bibliotecas proprietárias era de difícil portabilidade. Com base nesta dificuldade, impulsionada pelos avanços na tecnologia de sistemas paralelos de computação, foi reconhecida a necessidade de interfaces de programação que fossem eficientes, funcionais e portáteis nas mais variadas arquiteturas disponíveis no mercado.

Desta forma, foram criadas diversas IPPs, que auxiliam os programadores nas abstrações destas arquiteturas, facilitando a implementação de programas paralelos. Deste modo, neste trabalho será descrita a IPP OpenMP, que permite ao programador explorar o paralelismo em ambientes com arquitetura de memória compartilhada.

2.2.1 OpenMP

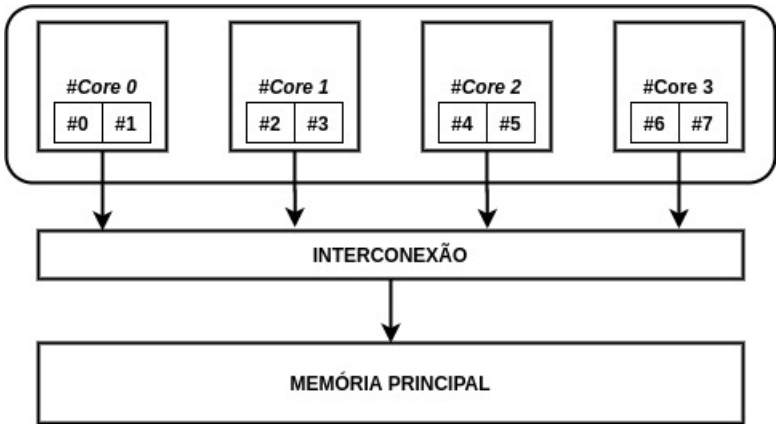
OpenMP é uma Interface de Programação Paralela para C/C++ e Fortran para ambientes com arquitetura de memória compartilhada. Consiste em diretivas de compilador, variáveis de ambiente e funções de biblioteca para especificar e controlar o trecho de código que deverá ser paralelizado. No entanto, é de responsabilidade do usuário identificar o paralelismo e inserir as estruturas de controle (condições de corrida, exclusão mútua) apropriadas para determinado trecho de código a ser paralelizado (PAS et al., 2017). OpenMP foi desenvolvido para que programas baseados na abordagem de memória compartilhada pudessem ser desenvolvidos em alto nível (PACHECO, 2011).

Ao se desenvolver programas paralelos para ambientes de memória compartilhada, o programador visualiza o sistema inteiro como um conjunto de *cores* de modo que estes através de uma interconexão (barramento do sistema) conseguem acessar a memória principal. A Figura 5 mostra um exemplo de um processador *multicore* com 4 *cores* e cada *core* possui duas *threads*. Desta maneira, o programador consegue usufruir das 8 *threads* disponíveis no sistema para realizar uma computação paralela e cada uma destas 8 *threads* acessa a memória principal do ambiente através de uma interconexão para endereçar seu espaço de memória. Portanto, qualquer mudança que uma *thread* realiza na memória

principal, esta pode ser acessada por todas as outras.

Devido ao fato do suporte para OpenMP estar crescendo (OPENMP, 2020), todos os principais compiladores fornecem suporte completo para OpenMP e continuam a implementar as especificações mais recentes. O comitê de linguagem OpenMP é muito ativo na adaptação das especificações às tendências em sistemas e aplicativos de computador. Enquanto isso, a lista oficial de membros do OpenMP empresas, universidades e laboratórios de pesquisa continuam a crescer (OPENMP, 2020). Isto se da pelo fato do OpenMP simplificar qual região de código deverá ser executada em paralelo, ficando a cargo do compilador decidir de que maneira esta região será paralelizada. De modo que, as principais diretivas existentes são: construtor paralelo, construtor de compartilhamento de trabalho e diretivas de sincronização, no qual serão explicados a seguir.

Figura 5 – Exemplo de Memória Compartilhada.



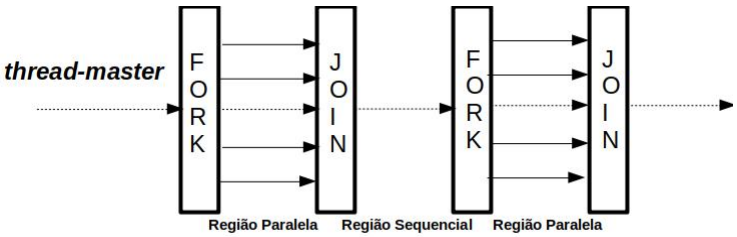
Fonte: O Autor.

O **Construtor Paralelo** como é ilustrado na Figura 6, especifica que um bloco estruturado de código deverá ser executado por múltiplas *threads*. Este construtor possui um modelo do tipo *Fork/Join*. Neste modelo, inicialmente existe um fluxo de execução principal chamado de *thread master*. Quando a *thread master* atinge um construtor paralelo, criará outras *threads*, tal processo é chamado de *Fork*. Ao fim da região paralela, uma barreira implícita força as *threads* aguardarem as demais *threads* terminarem suas execuções (*Join*). Quando ocorre o *Join*, apenas a *thread master* continua a execução do programa.

Os **construtores de compartilhamento de trabalho** são os responsáveis por informar ao compilador como a carga de trabalho deverá ser distribuída entre as *threads*. Laços paralelos são a principal forma de distribuição da carga de trabalho, nele uma estrutura de repetição é distribuída entre as *threads*.

OpenMP possui diferentes escalonadores que auxiliam na distribuição das iterações entre as *threads*. O escalonador ***static***, atribui a cada uma das *threads* pedaços estáticos utilizando uma política *round-robin*, ordenados pelo número de cada *thread*. No escalonador ***dynamic***, as iterações são atribuídas conforme solicitação. ***Guided*** é uma

Figura 6 – Modelo *Fork/Join*



Adaptado de: (PACHECO, 2011)

categoria de escalonador semelhante ao *dynamic*, porém o tamanho da carga de trabalho é decrementado a cada iteração. Por fim o escalonador ***runtime***, decide como a carga de trabalho será distribuída em tempo de execução.

Outros dois construtores são os construtores de seções paralelas e o construtor único. O **construtor de seção paralela** é encarregado de especificar diferentes regiões de código para serem executados em paralelo por diferentes *threads*, assim cada *thread* é responsável pela execução de cada bloco (LORENZON, 2014). Já o **construtor único** é atribuído para executar somente em uma *thread*, assim enquanto esta *thread* é executada, as outras aguardam em uma barreira, no final da região.

Por último, as **diretivas de sincronização** são utilizadas para evitar as condições de corrida, assim regiões de memória não podem ser acessados por mais de uma *thread* ao mesmo tempo. Existem três principais diretivas que restringem o acesso às informações **critical**, somente uma *thread* pode acessar uma região por vez; **atomic**, responsável por incumbir uma *thread* a atualizar uma determinada região da memória atomicamente; e ***barrier***, utilizada quando há a necessidade de sincronizar as *threads*.

A sincronização entre as *threads* também pode ser implícita, como ocorre no início e fim de uma região paralela. Por exemplo, sempre que ocorre a criação de novas *threads (fork)*, elas são sincronizadas entre si antes de iniciar a computação. O mesmo ocorre na finalização destas threads *(join)*, que são todas sincronizadas antes de serem finalizadas (PAS et al., 2017).

2.3 Métodos de Otimização

Métodos de otimização de maneira geral visam ajustar parâmetros que são auto-configuráveis onde, são utilizados em ambientes onde existe um problema de otimização multi-objetivo (LORENZON et al., 2019). Basicamente, os métodos utilizados para a otimização consistem em explorar um grande espaço de parâmetros que devem ser ajustados para encontrar o melhor *tradeoff* entre as métricas, como energia, tempo de execução. Desta forma, estes métodos podem ser utilizados para otimizar aplicações paralelas, de modo que a otimização pode, por exemplo, buscar um número ideal de *threads* que possua o menor EDP para a aplicação.

Inicialmente os dados são extraídos da aplicação que se deseja otimizar. No caso de aplicações paralelas, ferramentas de *profilling* podem ser utilizada para obter estes dados. Após isto, os dados obtidos previamente são fornecidos a um modelo que irá realizar uma busca pela melhor solução (número de *threads*) com base nestes valores. Tais modelos podem, utilizar redes neurais artificiais, métodos de regressão, meta-heurísitcas. Por fim, os resultados do método de otimização contém o melhor *tradeoff* entre os valores (número de *threads*) e as métricas (tempo de execução, energia ou EDP) (LORENZON et al., 2019)

Os métodos de otimização podem ser executados em diferentes momentos (fases) quando se trata de otimizar aplicações paralelas, podendo estes serem executados antes ou durante a execução da aplicação. Além destes, um terceiro método pode ser descrito, o método híbrido, de modo que a otimização ocorre em ambas as fases da aplicação. Com base nisto, será apresentado nas seções seguintes cada um dos três tipos de métodos de otimização de tal modo que, na Seção 2.3.1 será discutido sobre o método de otimização *offline*. Logo após, na Seção 2.3.2 a abordagem *online* e por fim, na Seção 2.3.3 o método híbrido de otimização.

2.3.1 Otimização *Offline*

Nesta abordagem, uma busca pela configuração ideal é realizada antes da execução da aplicação. Nesta categoria de abordagem, modelos de predição são muito utilizados, no qual utilizam uma variedade de modelos estatísticos para analisar os valores atuais e passados tais como podem ser vistos em (IPEK et al., 2005) (BARNES et al., 2008) (SHARKAWI et al., 2009). Os dados obtidos pelas predições são utilizados apenas para decidir qual a melhor configuração para a execução de uma aplicação. Um modelo preditivo é composto por alguns passos. Primeiramente, são coletados os dados de uma aplicação e com base nestes, é gerado um modelo. Um modelo então é formulado para que se possa aplicar algum método (regressão linear ou redes neurais) sobre os dados coletados. Após, são realizadas predições para a nova entrada de dados, e finalmente, são adicionados os novos dados para que o modelo seja validado. Assim sendo, não existe uma tomada de decisão e adaptação da aplicação em tempo de execução (LORENZON et al., 2019).

2.3.2 Métodos de Otimização *Online*

Neste tipo de otimização, os modelos consideram as informações obtidas em tempo de execução para realizar decisões e ajustar a execução da aplicação. Neste caso, diferentes características que só são conhecidas em tempo de execução, como o tamanho da entrada, são consideradas. Além disso, a adaptação usando informações dinâmicas é essencial para aplicações com comportamento variável, nas quais a carga de trabalho muda constantemente (RUSSEL, 2010). Entretanto, esta abordagem acaba adicionando uma sobrecarga (tempo de execução, consumo de energia) na aplicação paralela como um todo

durante o processo de otimização. Isto acontece porque a otimização ocorre de forma conjunta com a execução da aplicação.

2.3.3 Otimização Híbrida

A abordagem de otimização híbrida é a união das duas abordagens citadas anteriormente (*offline* e *online*). Desta forma, ao otimizar uma aplicação paralela com esta abordagem é necessário que as métricas avaliadas sejam coletadas antes da execução (para a abordagem *offline*) e durante a execução (para a abordagem *online*) e repassados para os seus respectivos algoritmos, meta-heurísticas ou modelos preditivos que irão buscar o número ideal de *threads* para a aplicação. Este tipo de método pode ser utilizado para auxiliar na redução do *overhead* gerado pela abordagem *online*, pois, a aplicação é otimizada antes da sua execução durante a utilização da abordagem *offline*.

2.4 Conclusão do Capítulo

Neste capítulo foram discutidos os principais assuntos que servem como referencial teórico para esta dissertação de mestrado. Primeiramente foi apresentado na Seção 2.1 uma visão geral sobre arquiteturas *multicore*. Após, na Seção 2.2, observou-se o conceito de programação paralela e como a IPP OpenMP pode ser utilizada para criar programas paralelos em ambientes de memória compartilhada. Por fim, na Seção 2.3 contém uma breve descrição dos métodos de otimização *online*, *offline* e híbrido.

3 TRABALHOS RELACIONADOS

Este capítulo apresenta os trabalhos que possuem temas correlatos com esta pesquisa. Tais trabalhos propõem diferentes abordagens para otimizar a execução de aplicações paralelas em ambientes *multicore*. Os trabalhos foram agrupados em três categorias: (*i*) técnicas que propõem modelos preditivos considerando a execução completa da aplicação; (*ii*) propostas de otimização que consideram tanto a execução parcial ou completa da aplicação, utilizando uma pequena entrada como treinamento; e (*iii*) técnicas de otimização *online* . Cada uma destas categorias é discutida em ordem cronológica nas seguintes subseções.

3.1 Métodos *Offline* que Consideram a Execução da Aplicação Inteira.

Esta seção aborda sobre os métodos *offline* de otimização para aplicações paralelas que utilizam a busca na entrada grande. Ou seja, métodos que realizam uma busca pelo número ideal de *threads* antes da aplicação ser executada. Os dados utilizados durante a fase de treinamento são obtidos através de ferramentas de *profilling* e, logo após, passados para os modelos preditivos ou meta-heurísticas que irá otimizar a aplicação.

O trabalho proposto por Ipek et al. (IPEK et al., 2005) emprega a utilização de redes neurais artificiais multicamada para prever o comportamento do desempenho de aplicações paralelas, em que estas foram treinadas com base nos dados de execução da plataforma que foi alvo do estudo. O modelo foi desenvolvido para ser automático utilizando apenas os dados passados como entrada. Como *benchmark* utilizou-se a aplicação SMG2000. O modelo proposto foi executado em duas plataformas paralelas de grande escala, obtendo assim erros de 5% e 7% em um espaço de parâmetros amplo e multidimensional.

Barnes et al. (BARNES et al., 2008) propõem o uso de regressões multivariadas para prever o desempenho de um grande número de processadores através de dados de treinamento obtidos de um pequeno número de processadores. Para isto, foram utilizadas três técnicas: uma aplicada a regressão multivalorada sobre o tempo de execução do passo de treinamento para prever o desempenho de um número grande de processadores e outras duas técnicas que refinam esta abordagem utilizando uma informação pré-processada para manipular a computação e a comunicação separadamente. O modelo proposto foi validado executando em sete *kernels* do *NAS Parallel Benchmark*, e Sweep3d no cluster Atlas com 1152 nós com quatro vias utilizando processadores AMD Opteron. Os resultados mostram um erro de predição entre 6.2% e 17.3%.

Sharkawi et al. (SHARKAWI et al., 2009) apresentam uma metodologia para projetar o desempenho de aplicações de alto desempenho em diferentes arquiteturas. Para isto, projetou-se o desempenho de oito aplicações paralelas em quatro ambientes com processadores diferentes, utilizando como base para a projeção, dados coletados de uma máquina IBM p575. Como resultado, constatou-se que o desempenho projetado para as

oito aplicações ficou dentro de 7.2% em média para o ambiente POWER6 e um desvio padrão de 5.3%.

Redes neurais artificiais (ANNs) foram utilizadas em Tiwari et al. (TIWARI et al., 2012) para prever o consumo de energia da memória e CPU, quando estas executam determinados *kernels*. Estas redes são treinadas utilizando dados empíricos obtidos da arquitetura alvo. Esta abordagem foi validada executando três *kernels* computacionais distintos (multiplicação de matrizes, *stencil*, e fatorização LU) em um Intel Xeon E5530 (no qual possui 2 processadores *quad-core*). Os resultados mostram que, uma vez que a rede é treinada, ela pode predizer o desempenho, e o consumo de energia da CPU e memória com um erro máximo de 5.5%.

Cabrera et al. (CABRERA et al., 2013) desenvolveram um modelo analítico para predizer o consumo de energia para o *benchmark* LINPACK, em sistemas de alto desempenho. A abordagem se baseia no modelo de desempenho apresentado em (CHOU et al., 2007), em que estes apresentam um modelo de predição de desempenho, utilizando-se de uma função de ponderação para levar em conta a sobrecarga de hardware durante a computação da aplicação, com a adição de novos parâmetros em relação à energia necessária para realizar comunicação e computação. Considerando também aplicações de alto desempenho, um modelo para estimar o consumo de energia em ambientes de HPC é proposto por Witkowski et al. (WITKOWSKI et al., 2013). Nele é utilizado um modelo de regressão linear multivariado para encontrar os dados do *hardware* que possuem alta correlação com consumo de energia e construir um modelo que estima o consumo de aplicações de alto desempenho.

Benedict et al. (BENEDICT et al., 2015) propõem um mecanismo para predizer o consumo de energia para aplicações em OpenMP utilizando *Random Forest Modeling* (RFM). Esta abordagem possui cinco passos principais: (*i*) uma entidade de análise realiza uma análise inicial das aplicações OpenMP independente da região paralela. (*ii*) Uma entidade otimizadora encontra uma solução ótima para o consumo de energia para uma determinada região do código considerendo aspectos de desempenho, (*iii*) A entidade otimizadora prepara uma lista de melhores configurações e as submete ao mecanismo de predição. (*IV*) O consumo de energia e desempenho para cada configuração são preditas utilizando a RFM. (*V*) Finalmente, os resultados da predição são enviados para a entidade otimizadora, onde esta deverá prover a melhor solução.

Por fim, Vidya et al. (VIDYA; SRIRAMAN; RUKMANI, 2019) utilizam uma metodologia no qual inicialmente um estudo sobre a eficiência energética do paralelismo em nível de *threads* de algumas aplicações com diferentes cargas de trabalho é realizado. Após isto, os resultados obtidos deste estudo são salvos em um disco secundário. De modo que, quando uma nova aplicação for executada, cerca de 12 caracterísitcas desta são comparadas com os resultados obtidos previamente, utilizando para isto o método *least hamming distance*, que irá escolher um número ótimo de *threads* para a nova aplicação.

Com base nesta proposta, os autores puderam mostrar ganhos de até 30% em energia quando um número de *threads* ótima é utilizada.

3.2 Métodos *Offline* que Consideram a Execução Parcial de Aplicações Paralelas ou Pequenas Entradas.

Os trabalhos descritos nesta seção também fazem uso da metodologia *offline* de otimização. Porém, estes utilizam durante a fase de treinamento apenas a execução parcial de uma aplicação. Sendo assim, são utilizados durante o treinamento uma pequena entrada para predizer o comportamento da aplicação com a entrada grande (regular).

Considerando a execução parcial de uma aplicação paralela, Yang et al. (YANG; MA; MUELLER, 2005) apresentam um modelo para predizer o tempo de execução de uma aplicação de larga escala para um sistema alvo. Utilizando pra isto, a combinação de desempenho conhecido de um sistema de referência e o desempenho relativo entre os dois sistemas resultante de uma pequena execução da aplicação. Como resultado, conseguiram predizer o tempo de execução com uma acurácia de 97% executando apenas 1% ou menos a aplicação como um todo.

Sodhi et al. (SODHI; SUBHLOK; XU, 2008) apresentam um *framework* para a construção automática de esqueletos de desempenho para prever o tempo de execução em ambientes distribuídos. O *framework* captura o ambiente de execução de uma aplicação e gera automaticamente um esqueleto de desempenho. Mais formalmente, um esqueleto de desempenho é um programa de curta execução gerado sinteticamente, cujo tempo de execução sempre reflete o desempenho do aplicativo que representa. A metodologia proposta mostrou que a execução de poucos segundos de uma aplicação paralela pode predizer o tempo de execução da aplicação como um todo com bastante precisão.

Em relação aos trabalhos que utilizam entradas pequenas, Zhang et al. (ZHANG; CHENG; SUBHLOK, 2016) propõem *DwarfCode*, um preditor de desempenho para aplicações híbridas (OpenMP, MPI) que coleta *traces* da execução com entradas pequenas que imita o ambiente da aplicação como um todo. Sendo assim, estes *traces* são utilizados para prever o desempenho da aplicação quando for executada sobre um tamanho de maior entrada. Jayakumar et al. (JAYAKUMAR; MURALI; VADHIYAR, 2015) apresentam um *framework* que extrai assinaturas (dados de perfil) de aplicações paralelas e executa a identificação automática das diferentes fases da aplicação. Sendo assim, assinaturas de diferentes fases são comparadas com o perfil de aplicações de referência armazenados em uma base de dados. O desempenho destas referências são utilizadas para predizer o desempenho das diferentes fases para outras aplicações.

Uma técnica pra estimar o número de *threads* e o *governor Dynamic Voltage and Frequency Scaling* (DVFS) que fornece o melhor desempenho e consumo de energia para aplicações paralelas é proposto por Sensi et al. (SENSI, 2016). A ideia principal é executar o programa com algumas poucas configurações (número de *cores* e frequência) e então,

prever o ambiente de outras configurações através de uma regressão linear múltipla.

3.3 Métodos *Online*

Os trabalhos descritos nesta seção irá abordar técnicas de otimização do tipo *online*. Sendo assim, as abordagens contidas nestes trabalhos, fazem uso de algoritmos que buscam ajustar uma aplicação paralela enquanto esta é executada.

Suleman et al. (SULEMAN; QURESHI; PATT, 2008) propõem um *framework* denominado *Feedback-Driven Threading*(FDT), que adapta o número de *threads* para uma dada aplicação paralela de maneira dinâmica utilizando dados desta em tempo de execução. Sendo assim, FDT consiste em um compilador específico que realiza uma análise em uma porção de regiões paralelas de uma aplicação OpenMP, inserindo instruções na entrada e na saída de seções criticas e, executa-a sequencialmente para analisar pontos de sincronização e comunicação entre as *threads*. O *framework* então, utiliza esta análise para estimar o número de *threads* para uma dada região paralela.

LIMO (CHADHA; MAHLKE; NARAYANASAMY, 2012) é um sistema que gerencia dinamicamente o número de threads em tempo execução de uma aplicação paralela. Tendo como objetivo maximizar o desempenho e a eficiência energética. De tal forma que LIMO monitora o progresso das *threads* junto com os recursos compartilhados com o *hardware* para determinar o melhor número de *threads* a serem executadas, além de verificar o nível de tensão e frequência dos núcleos. Como resultado, mostram que com a adaptação dinâmica, LIMO provê em média 21% de melhora no desempenho e redução no consumo de energia de 2x menos em um sistema de 32 núcleos sobre a execução padrão de aplicações da suite de *benchmark* PARSEC com 32 *threads*.

Thread Tailor (LEE et al., 2010) ajusta o número de *threads* para otimizar algumas partes específicas do sistema, tais como cache e memória. A abordagem funciona da seguinte maneira: (i) programadores criam uma aplicação paralela que utiliza um alto número de *threads*; (ii) o arquivo binário criado é examinado de maneira *offline* para coletar estatísticas independente do número de *threads*, comunicação e sincronização para formar um grafo de comunicação. (iii) em tempo de execução, o compilador dinâmico obtém um *footprint* do estado do sistema para determinar a quantidade de recursos livres disponíveis e decidir o número ótimo de *threads*; (IV) baseado nestas informações, o compilador dinâmico gera código para novas *threads*, intercepta futuras chamadas para criação de *threads* e as redireciona para novas *threads*.

Porterfield et al. (PORTERFIELD et al., 2013) propõem uma abordagem adaptativa em tempo de execução que ajusta automaticamente o número de *threads* baseando-se na utilização dos recursos do sistema. Esta abordagem extende Qthreads (WHEELER; MURPHY; THAIN, 2008) (uma biblioteca paralela) para ser utilizada com MAESTRO, uma biblioteca dinâmica em tempo de execução para adaptação de consumo de energia e concorrência de aplicações paralelas. Li et al. (LI et al., 2010) abordam uma metodologia

para melhorar o consumo de energia de aplicações paralelas que utilizam as IPP's MPI e OpenMP de forma conjunta. Para tal, foi implementado um modelo de previsão que ajusta em tempo de execução o DVFS e o *Dynamic Concurrency Throttling*(DCT) para estas aplicações.

ParallelismDial (PD) (SRIDHARAN; GUPTA; SOHI, 2013) é um modelo que automaticamente otimiza o desempenho de um programa para um outro ambiente de execução. PD monitora a eficiência do sistema, regula o grau de paralelismo, e continuamente adapta a execução através de uma heurística de ponto ótimo de operação. Em Sridharan et al. (SRIDHARAN; GUPTA; SOHI, 2014), PD foi extendido para o sistema Varuna. Este abrange dois componentes: (i) uma *engine* analítica que monitora mudanças continuamente no sistema utilizando contadores de desempenho do *hardware*, ambientes de modelo de execução, e determina o grau ótimo de paralelismo; e (ii) um administrador que regula automaticamente a execução para corresponder ao grau de paralelismo determinado pela *engine* analítica. PD e Varuna compreendem um monitor do sistema que intercepta a criação de *threads* e *tasks* das IPPs *Pthreads*, *Thread Building Blokcs*(TBB) e Prometheus criando uma *pool* de *tasks* para otimizar o seu grau de paralelismo.

OpenMPE (ALESSI et al., 2015) é uma extensão para gerenciar o consumo de energia de de aplicações OpenMP, de maneira que, os programadores devem expor melhorias para o consumo de energia através de diretivas nos códigos OpenMP. Shafik et al. (SHAFIK et al., 2015) apresenta um modelo adaptativo e escalável para minimizar o consumo de energia para programas OpenMP, no qual abrange dois passos: (i) inserção de anotação no código pelo programador nas partes sequenciais e paralelas do código para habilitar a minimização de energia com requerimentos de desempenho especificado; (ii) o sistema de *runtime* lê estes requerimentos de desempenho e utiliza estas informações para guiar a redução do conumo de energia.

3.4 Contribuições Deste Trabalho

Conforme foi descrito nos trabalhos da Seção anterior, é possível verificar que diferentes estratégias e metodologias têm sido propostas para otimizar aplicações paralelas. Entretanto, estes trabalhos apresentam algumas limitações, as quais são destacadas abaixo e podem ser vistas na Tabela 1.

Os trabalhos mencionados na Seção 3.1 consideram a execução inteira da aplicação durante o período de treinamento. Sendo assim, sempre que há mudanças no ambiente de execução (e.g., processador e sistema operacional) a aplicação deverá ser treinada novamente, aumentando significantemente a sobrecarga das técnicas de aprendizado. Já as abordagens discutidas na Seção 3.2, limitam-se em realizar modificação ou recompilação do código fonte, ou então ter a necessidade de se executar muitas aplicações para que se possa construir uma base de dados para alimentar um modelo preditivo. Além do mais, mesmo que metodologias *offline* tal como Dutot el al. (DUTOT et al., 2017) apresenta

Tabela 1 – Resumo comparativo de Hórus em relação aos trabalhos citados.

Trabalho Relacionado	**Execução Parcial**	**Sem Modificar o Código**	**Múltiplas IPPs**
(IPEK et al., 2005)		X	
(BARNES et al., 2008)		X	
(SHARKAWI et al., 2009)		X	
(TIWARI et al., 2012)		X	
(CABRERA et al., 2013)		X	
(WITKOWSKI et al., 2013)		X	
(BENEDICT et al., 2015)		X	
(VIDYA; SRIRAMAN; RUKMANI, 2019)		X	
(YANG; MA; MUELLER, 2005)	X	X	
(SODHI; SUBHLOK; XU, 2008)	X		
(ZHANG; CHENG; SUBHLOK, 2016)	X		
(JAYAKUMAR; MURALI; VADHIYAR, 2015)	X		
(SENSI, 2016)	X		
(SULEMAN; QURESHI; PATT, 2008)		X	
(CHADHA; MAHLKE; NARAYANASAMY, 2012)		X	
(PORTERFIELD et al., 2013)		X	X
(SRIDHARAN; GUPTA; SOHI, 2013)		X	
(SRIDHARAN; GUPTA; SOHI, 2014)		X	X
Framework Hórus	X	X	X

não necessite acesso a uma base de dados antes da execução da aplicação, a técnica de aprendizado proposta nesta pesquisa é muita mais eficiente, pois utiliza durante a fase de aprendizado uma entrada pequena, tornando o processo de busca pelo número de *threads* ideal para a aplicação mais rápida.

Quando comparamos a proposta deste trabalho com as dicutidas na Seção 3.3, as principais diferenças se dão nos seguintes pontos: (i) a maioria dos trabalhos apresentam metodologias apenas para uma IPP. Por outro lado, a metodologia proposta por este trabalho pode ser utilizada para otimizar aplicações paralelas de outras IPP's (*Message Passing Interface* (MPI), Pthreads) ao mesmo tempo e (ii) Esta estratégia é totalmente transparente para o usuário, sem realizar modificações no código fonte ou no binário da aplicação.

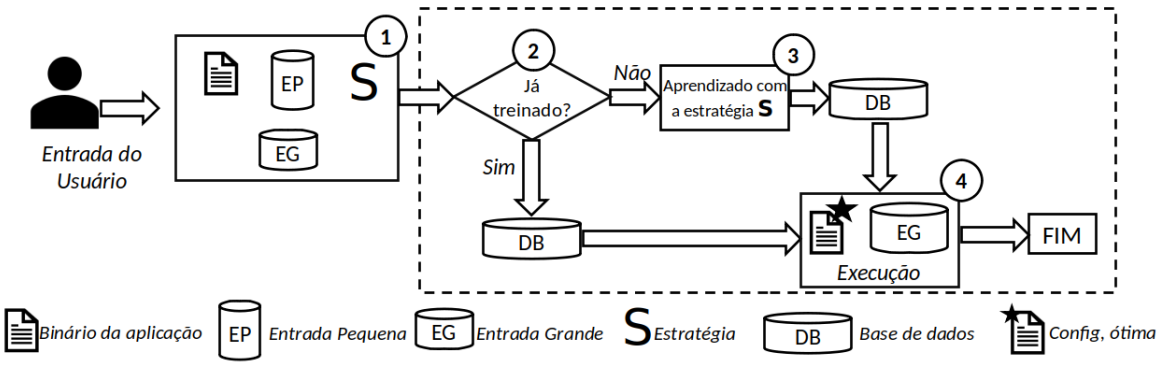
4 *FRAMEWORK* HÓRUS

Este capítulo descreve o *framework* Hórus que otimiza o EDP de aplicações paralelas com baixo custo de aprendizado para estratégias *offline* de otimização. A Seção 4.1 contém o fluxo de execução do *framework*. Na Seção 4.2 consta a estratégia de busca que visa encontrar um número de *threads* para uma aplicação paralela como um todo e, a Seção 4.3 apresenta a estratégia de busca que é utilizada pelo Hórus para encontrar um número de *threads* ideal para cada região paralela de uma dada aplicação.

4.1 Otimizando a Fase de Treinamento de Estratégias *Offline*

De maneira geral, o fluxo de execução do Hórus proposto neste trabalho é ilustrado na Figura 7. De modo que, os principais elementos que constituem Hórus são: o binário da aplicação; uma entrada pequena para ser utilizada durante a fase de treinamento pela busca do número de *threads* ideal; uma entrada grande, que representa a execução da aplicação real a ser otimizada após a fase de busca; uma estratégia a ser empregada SEA ou SPRA, discutidas mais adiante nesta Seção; uma única base de dados (arquivo de texto) que irá armazenar as configurações ideais (número de *threads*) encontradas para cada aplicação após a fase de busca.

Figura 7 – Fluxo de execução Hórus



Fonte: O Autor.

1. O primeiro passo consiste na definição das entradas por parte do usuário: o binário da aplicação, com uma entrada pequena que será utilizada na fase de aprendizado; entrada original, para a execução da aplicação; e uma estratégia a ser empregada durante a fase de busca (no qual será discutida mais adiante nesta Seção).

2. Hórus verifica em uma base de dados se ambas as entradas já foram treinadas. Esta base de dados armazena as configurações encontradas durante a fase de treinamento (Passo 3) para cada aplicação. Se a aplicação já foi treinada, ou seja, já foi encontrado um número de *threads* ideal para a aplicação, Hórus não aplica o algoritmo de aprendizado novamente previsto na fase 3. Este passa diretamente à fase 4, que consiste em executar o binário da aplicação com a entrada original com base nas configurações previamente armazenadas na base de dados. Com a utilização desta base de dados, é possível reduzir o custo de aprendizado de aplicações que são executadas frequentemente e não possuem mudanças no código.

3. Este passo, chamado de fase de aprendizado, aplica o algoritmo durante a execução da aplicação com a entrada pequena. O algoritmo de busca que deve ser aplicado é definido no passo 1 (no qual será descrito com mais detalhes posteriormente). Uma vez que a busca converge para uma solução, ela é armazenada na base de dados para ser utilizada em execuções futuras.

4. O último passo é a execução da aplicação com o tamanho original com *(i)* o número de *threads* encontrado durante a fase de treinamento (passo 3) ou *(ii)* o número de *threads* que foi armazenado na base de dados.

A seguir, será descrito cada uma das estratégias que o usuário poderá empregar durante a fase de busca. A primeira consiste em buscar um número de *threads* para a aplicação inteira, ou seja, o mesmo número de *threads* para todas as regiões paralelas da aplicação. E outra estratégia busca encontrar um número de *threads* ideal para cada região paralela da aplicação.

4.2 SEA: Buscando um Número Ideal de *Threads* para a Aplicação Inteira

A estratégia de busca SEA compreende em encontrar um grau de TLP para a aplicação inteira considerando que todas as regiões paralelas executam com o mesmo número de *threads*. A aplicação paralela então pode ser executada por no máximo n *cores* distintos $C = \{C_1, C_2, ..., C_n\}$. O problema de otimização no qual se está interessado busca uma atribuição da aplicação paralela A para um subconjunto de *threads/core* em C. Denota-se por C^* o conjunto de todos os subconjuntos de *threads/core* em C, tal que, $C^* = \cup_n \binom{C}{n}$. Uma atribuição viável pode ser definida como uma função $\phi : R \rightarrow C^*$ que garante um subconjunto de *threads/core* para uma aplicação paralela.

Portanto, denota-se por $\mathbb{M} : (A \times n) \rightarrow \mathbb{A}^+$ o EDP obtido após a execução da aplicação paralela nos n *cores* distintos. Desta maneira, uma atribuição viável ϕ consiste em encontrar uma atribuição válida ϕ que conduz a um valor mínimo para $\mathbb{M}$. Em outras palavras, a estratégia tenta encontrar um número de *threads* que obtiver o menor valor

Algorithm 1 Estratégia de Busca SEA

Entrada: A: Aplicação Paralela
$S \leftarrow \{Input\}$: entrada pequena de A
$C \leftarrow \{C_1, C_2, \ldots, C_n\}$: set of threads/cores
α: Número inicial de *threads*
β: Aumento no número de *threads*
n: Número de *cores*

```
1: φ' ← ∞: mínimo EDP encontrado até o momento
2: M' ← M(A, α)
3: while M' ≤ φ' and 1 ≥ α ≤ n do
4:     φ' ← M'
5:     α ← α · β
6:     M' ← M(A, α)
7: end while
8: if M' ≥ φ' and α ≤ k then
9:     upper ← α and lower ← α/β
10:    while lower ≤ upper do
11:        α' ← (upper + lower)/2
12:        M' ← M(A, α')
13:        if M' ≥ φ' then
14:            upper ← α' and lower ← α/β
15:        end if
16:        if M' ≤ φ' then
17:            upper ← α and lower ← α'
18:        end if
19:        if |upper − lower| = 2 then
20:            lateral movement
21:        end if
22:    end while
23: end if
24: Return(φ(α))
```

para $\mathbb{M}$ de n *cores* escolhidos em C. Diz-se então que ϕ está otimizada se este é o menor valor da função $\mathbb{M}$. Formalmente,

$$arg\ min_{\forall \phi}(\mathbb{M}(\phi))$$

O algoritmo 1 recebe como entrada do Hórus um conjunto de *cores* C, uma entrada pequena da aplicação A, e três parâmetros: (i) α, indicando um número de *threads* inicial para a aplicação A. Hórus calcula este valor de acordo com a arquitetura disponível, de modo que: $\alpha = 2$ se o resto $\frac{n}{3}$ for 0 caso contrário, $\alpha = 3$; (ii) β indicando um fator de aumento no número de *threads* para A, neste caso o fator de aumento será sempre o dobro da quantidade de *threads* em α; e (iii) o número máximo de *cores* disponíveis n. O procedimento inicia aumentando exponencialmente (de acordo com o valor β) o número de *threads* $\alpha^{'}$ dado a A (linhas 3 até 7), enquanto minimiza o valor da função $\mathbb{M}(\phi) = M^{'}$. Ao encontrar o máximo local para a função M, o algoritmo executa uma busca utilizando o método *hill-climbing* modificado (linhas 10 até 22) no intervalo onde o algoritmo tenha encontrado o menor valor para $\mathbb{M}$. Para evitar mínimos locais e platôs durante o processo, e erroneamente convergir para um número de *threads* incorreto, o algoritmo faz uso de movimentos laterais, ou seja, quando $|upper - lower| = 2$ um ponto vizinho do espaço

de busca atual será testado. Isto é executado antes de selecionar o melhor número de *threads* testando uma configuração vizinha em outro ponto do espaço de busca que ainda não foi testado. Utilizando este algoritmo é possível convergir para um valor mínimo em no máximo $O(\log_\beta n)$ iterações. Quando o algoritmo de aprendizado encontra um número de *thredas* ideal para a aplicação A com a entrada pequena, o valor é retornado para o Hórus.

Encontrar um número de *threads* ideal para a execução de uma aplicação paralela pode ser considerado um problema de otimização convexo, no qual para este tipo de problema, existe apenas um único número específico de *threads* que entrega o melhor resultado para EDP. Algoritmos do tipo *Hill-Climbing* são adequados para este tipo de problema por dois motivos: (*i*) estes algoritmos fazem progressos rápidos em direção a uma solução porque geralmente é muito fácil melhorar um estado ruim (RUSSEL, 2010); *(ii)* por ter uma baixa complexidade. Além do mais, outros autores já tem mostrado que quando o método *hill-climbing* é utilizado com outras abordagens para guiar a busca, na maioria dos casos estes algoritmos chegam muito próximo da solução ótima, escapando de mínimos locais e platôs (TABORDA; ZDRAVKOVIC, 2012).

Esta estratégia pode ser utilizada com aplicações paralelizadas com as IPPs mais utilizadas, como OpenMP, PThreads, e MPI. Para configurar o número de *threads*/processos que devem ser criadas durante o algoritmo de treinamento (e para a execução da aplicação após a fase de aprendizado), foi considerado a maneira padrão que isto é feito para cada IPP. Aplicações OpenMP têm o seu número de *threads* definido através da variável de ambiente OMP_NUM_THREADS em sistemas operacionais Linux. Para aplicações implementadas em PThreads, o grau de TLP é definido através da definição do número máximo de *threads* que podem ser criadas dentro de um único processo (i.e., o número máximo de *threads* que a função pthread_create pode criar). Além disso, para aplicações paralelizadas através da IPP MPI o número de processos é definido utilizando a opção -np do comando mpirun/mpiexec.

O EDP de cada execução da aplicação durante o algoritmo de aprendizado é obtido multiplicando o tempo de execução (em segundos) pela energia consumida (em joules). O tempo de execução foi obtido através da função time() presente na biblioteca time do Python versão 3. Por outro lado, a energia consumida é obtida diretamente dos contadores de *hardware* presentes nos processadores modernos. No caso para processadores Intel, o *Running Average Power Limit* (RAPL)(HäHNEL et al., 2012) foi utilizado, enquanto que para obter a energia em processadores AMD foi utilizado a biblioteca *Application Power Management*(HACKENBERG et al., 2013).

4.3 SPRA: Buscando um número Ideal de *threads* para cada Região Paralela da Aplicação

Para esta estratégia, o objetivo é encontrar um número ideal de *threads* para cada região paralela para uma dada aplicação. Portanto, dado uma aplicação A com m regiões paralelas independentes $P = P_1, P_2, ..., P_m$. Cada região paralela em P pode ser executado por até n *cores* distintos $C = C_1, C_2, ..., C_n$. Desta forma, denota-se $A(p)$ o conjunto de regiões paralelas para uma dada aplicação A. O problema de otimização para esta estratégia busca um conjunto de *cores/threads* para uma dada região paralela P presentes em C. Sendo assim, C^* representa o conjunto de todos os subconjuntos de *threads/cores* em C, isto é, $C^* = \cup_n \binom{C}{n}$ Uma garantia confiável pode ser definida coma uma função $\phi : P \rightarrow C^*$ que garante o subconjuto de *threads/cores* para as regiões paralelas. Entretanto, denota-se por $\mathbb{M} : (P \times n) \rightarrow \mathbb{P}^+$ o EDP da execução da região paralela em n *cores* distintos. Uma atribuição otimizada ϕ consiste em encontrar uma atribuição válida ϕ que conduz a um valor mínimo para $\mathbb{M}$. Sendo assim, uma garantia ϕ é otimizada se este for o mínimo valor da função $\mathbb{M}$.

O algoritmo de aprendizado (2) recebe como entrada do Hórus o conjunto de *cores* C, um conjunto de regiões paralelas P e três parâmetros: (i) α descrevendo o número inicial de *threads* para uma dada região P_i, $(ii)\beta$ representando um fator do aumento do número de *threads* para uma determinada região P_i, e (iii) o número máximo de *cores* disponíveis n. O processo de busca por si só é muito similar a executada pela estratégia SEA, com algumas diferenças: (i) esta é aplicada para cada região paralela p da aplicação A. Consequentemente, a aplicação é apenas executada uma vez durante a fase de treinamento, e (ii) o resultado da busca é uma lista contendo o número ideal de *threads* para cada região paralela (denotado por $P.\phi(\alpha)$). Então, quando o algoritmo de aprendizado termina, esta lista é retornado para o Hórus, e a aplicação A é executada com a entrada grande contendo o número ideal de *threads* para cada região.

Para avaliar e validar esta estratégia de busca, ela foi implementada internamente a biblioteca GNU OpenMP (*libgomp*). Por isso, esta estratégia funciona para qualquer aplicação OpenMP que explora *loops* paralelos mas não influencia a execução de aplicações OpenMP implementadas com seções. *Libgomp* é dinâmicamente acoplado para aplicações OpenMP, então qualquer modificação neste código é inteiramente transparente para as aplicações. Para calcular o EDP, o tempo de execução de cada região paralela é obtido através da função omp_get_wtime, implementada pelo OpenMP. Para o consumo de energia, esta utiliza os mesmos contadores de *hardware* e bibliotecas na estratégia SEA.

Algorithm 2 Estratégia de Busca SPRA

Entrada: A: Aplicação Paralela
$P \leftarrow \{P_1, P_2, \ldots, P_p\}$: Conjunto de regiões paralelas de A
$S \leftarrow \{Input\}$: Entrada pequena de A
$C \leftarrow \{C_1, C_2, \ldots, C_n\}$: conjunto de *threads/cores*
α: Número inicial de *threads*
β: Aumento do número de *threads*
n: Número de *cores*

$\phi' \leftarrow \infty$: mínimo encontrado até o momento
for cada região paralela $P_i \in P$ **do**
  $M' \leftarrow \mathbb{M}(P_i, \alpha)$
  while $\mathbb{M}' \leq \phi'$ and $1 \geq \alpha \leq n$ **do**
    $\phi' \leftarrow \|\mathbb{M}'$
    $M' \leftarrow \mathbb{M}(P_i, \alpha)$
  end while
  if $\mathbb{M}' \geq \phi'$ and $\alpha \leq k$ **then**
    while $lower \leq upper$ **do**
      $\alpha' \leftarrow (upper + lower)/2$
      $M' \leftarrow \mathbb{M}(A, \alpha')$
      if $\mathbb{M}' \geq \phi'$ **then**
        $upper \leftarrow \alpha'$ and $lower \leftarrow \alpha/\beta$
      end if
      if $\mathbb{M}' \leq \phi'$ **then**
        $upper \leftarrow \alpha$ and $lower \leftarrow \alpha'$
      end if
      if $|upper - lower| = 2$ **then**
        lateral movement
      end if
    end while
  end if
end for
return$(P.\phi(\alpha))$

5 METODOLOGIA

Este capítulo tem como objetivo apresentar a metodologia desenvolvida durante este trabalho. De modo que na Seção 5.1 serão apresentados os *benchmarks* utilizados e os tamanhos dos dados utilizados durante a realização dos experimentos. Por fim na Seção 5.2, será apresentado os ambientes no qual os *benchmarks* foram submetidos a execução e elencar quais os cenários comparativos nos quais os resultados serão analisados.

5.1 Conjunto de *Benchmarks*

Neste trabalho foram utilizadas dezoito aplicações paralelas implementadas nas linguaguens de programação C/C++ com o auxilio da biblioteca OpenMP para explorar o paralelismo no nível de *threads*. De modo que, todas estas aplicações foram compiladas com o GCC/G++ 9.2, utilizando a *flag* de otimização -O3. Tais aplicações são descritas abaixo,:

- **Nove *kernels*** do conjunto *NAS Parallel Benchmark*(BAILEY et al., 1991).
  - *Block Tri-diagonal* (**BT**): algoritmo implementado para resolver equações de sistemas de equações tridimensionais através de um sistema de blocos tridiagonais de 5x5 no qual são resolvidos sequencialmente em cada dimensão.
  - *Conjugate Gradiente* (**CG**): utiliza o método de gradiente conjugado pra calcular a aproximação de sistemas particulares de equações lineares.
  - *Embarrassingly Parallel* (**EP**): estima os limites de desempenho para operações de ponto flutuante.
  - *Discrete 3D fast Fourier Transform* (**FT**): realiza o calculo da transformada rápida de Fourier (*Fast Fourier Transform* (FFT)) um sistema tridimensional de equações diferenciais parciais aplicando a FFT em cada dimensão separadamente.
  - *Lower-Upper Gauss-Seidel*(**LU**): resolve equações de sistemas lineares em três dimensões pela divisão em blocos de sistemas triangulares superior e inferior.
  - *Multi-Grid*(**MG**): calcula a solução de sistemas de equações de três dimensões com malhas, variando entre malhas mais finas e mais grossas.
  - *Scalar Penta-diagonal* (**SP**): este *kernel* tem como finalidade solucionar um sistema de equações lineares através da aproximação de Beam-Warning, separando as dimensões e resolvendo sequencialmente em cada uma com um sistema penta-diagonal.
  - *Unstructured Adaptive mesh*(**UA**): mede o desempenho de acessos irregulares à memória, utilizando malhas não estruturadas, fazendo uso de equações de transferência de calor.

  - *Integer Sort* (**IS**): Ordenação de inteiros com acesso aleatório a memória.

- **Quatro aplicações** do conjunto Rodinia (CHE et al., 2009), sendo elas:

  - *Hotspot*(**HS**): é uma ferramenta de simulação técnica usada para estimar a temperatura de um processador baseado na arquitetura *floor plan* e simular medidas de consumo energético. Este *benchmark* inclui uma simulação térmica transiente 2D, onde iterativamente computa uma série de equações diferenciais por blocos de temperaturas.
  - *LavaMD*(**LM**): este método calcula o potencial de partículas e a realocação devido a forças mútuas entre partículas dentro de um grande espaço 3-D. Esse espaço é dividido em cubos ou caixas grandes, que são alocados para nós de clusters individuais.
  - *Leukocyte*(**LT**): detecta e rastreia o movimento de leucócitos em um vídeo de microscópio de vasos sanguíneos. Nesta aplicação, células são detectadas no primeiro *frame* do vídeo e então rastreia-os através dos *frames* subsequentes.
  - *Needleman-Wunchs* (**NW**): é um método de otimização global para alinhamentos de sequência de DNA. Potenciais pares de sequencias são organizadas em uma matriz 2-D. O algoritmo preenche a matriz com pontos que representam o valor do caminho máximo ponderado que termina nesta célula. O caminho de volta é usado para procurar um alinhamento ótimo entre as células.

- **Cinco aplicações** de diferentes domínios:

  - Método de Jacobi(**JA**): O método de Jacobi é um método iterativo utilizado para solucionar sistemas de equações lineares do tipo $Ax = b$, onde A é uma matriz quadrada, x são as incógnitas e b é um vetor conhecido (GILAT et al., 2008).
  - *Stream*(**ST**): é uma aplicação que mede a largura de banda de memória em Mb/s e a taxa de computação correspondente para *kernels* de vetor simples. Este *benchmark* é projetado especificamente para trabalhar com conjunto de dados muito maiores do que a cache disponível em qualquer sistema, de modo que os resultados sejam (presumidamente) mais indicativos do desempenho de aplicativos muito grandes no estilo vetorial (MCCALPIN, 1991-2007).
  - *Fast-Fourier-Transform*(**FFT**): calcula a transformada de Fourier discreta para uma dada sequência (PETERSEN; ARBENZ, 2004).
  - *N-body*(**NB**): computa a simualção de um sistema dinâmico de particulas (BHATT et al., 1992)
  - *Poisson*(**PO**): calcula uma solução aproximada para a equação de Poisson (QUINN, 2004).

Tabela 2 – Diferença de tamanho entre a entrada pequena e a entrada regular para todos os *benchmarks*.

Classe *Benchmark*	**Bench.**	**Conjunto de Entrada Pequeno**		**Conjunto de Entrada Regular**	
		#Iterações	***Tamanho da Estrutura de Dados***	***#Iterações***	***Tamanho da Estrutura de Dados***
Número de Iterações	***NB***	100	700	10000	700
	JA	30	4096 x 4096	7000	4096 x 4096
	PO	0.001 (limiar.)	768 x 768	0.000001 (limiar.)	768 x 768
	ST	20	10M	5000	10M
	LT	8	-	512	-
	LM	10	-	40	-
	HS	100	1024 x 1024	1M	1024 x 1024
Híbrido	***MG***	4	128 x 128 x 128	20	256 x 256 x 256
	UA	50	250 (nº elements)	200	33500 (nº elementos)
	EP	2^{24}	2^{24}	2^{32}	2^{32}
	FT	6	64 x 64 x 64	20	512 x 512 x 512
	LU	50	12 x 12 x 12	250	162 x 162 x 162
	IS	2^{16}	2^{16}	2^{27}	2^{27}
	CG	15	1400 linhas	75	150000 linhas
	FFT	20	20^2	28	28^2
Tamanho da Estrutura de Dados	***BT***	200	64 x 64 x 64	200	162 x 162 x 162
	SP	400	64 x 64 x 64	400	162 x 162 x 162
	NW	-	10k x 10k x 10k	-	40k x 40k x 40k

A estratégia proposta neste trabalho visa executar uma aplicação com uma entrada pequena para otimizar a sua execução sobre o conjunto de dados maiores (entrada regular). Para isto, foi fixado uma entrada pequena que represente menos que 2% do ambiente de execução da entrada regular. De acordo com as diferenças de cada entrada, os *benchmarks* foram classificados em três classes, como pode ser visto na Tabela 2:

- *Número de Iterações*: a diferença entre a entrada pequena e a regular se da pelo número de iterações que a aplicação executa[1]. Entretanto, as outras variáveis pertencentes a aplicação não sofrem alterações (Tabela 2). Por exemplo, a aplicação NB, tem dois parâmetros de entrada, número de corpos (*body*) e quantidades de iterações (*steps*). Deste modo, a diferença entre as entradas se dá pelo número de iterações: 100 para a pequena e 10000 para a regular.

- *Estrutura de Dados*: apenas o tamanho da estrutura de dados (e.g, tamanho do vetor, matriz ou malha) muda na execução da entrada pequena para a entrada grande (Tabela 2). Como exemplo a aplicação SP possui três parâmetros: tamanho da malha (matriz tri-dimensional), número de iterações e passo do tempo. Desta maneira, a difereça entre as entradas se da apenas no tamanho da malha: 64 X 64 X 64 na entrada pequena e 162 X 162 X 162 para entrada grande.

- *Híbrida*: o número de iterações e o tamanho da estrutura de dados mudam conforme o conjunto de entrada varia de pequeno a grande (Tabela 2). Um exemplo é o *Kernel*

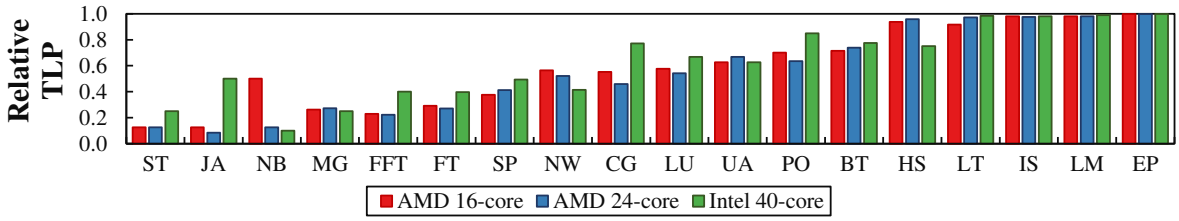
[1] Para a aplicação PO, o número de vezes que a aplicação irá iterar se da através de um limiar, onde a computação do método de Poisson irá terminar quando o erro produzido for menor que o limiar. Sendo assim, quando menor o limiar, mais iterações serão executadas.

CG, onde a diferença entre os dois conjuntos de entrada é o número de repetições e o tamanho da estrutura de dados.

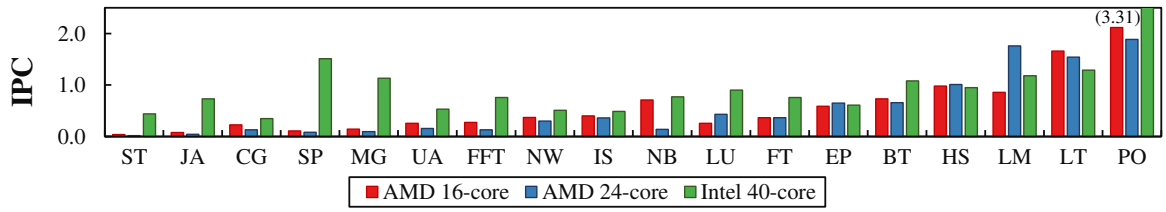
As aplicações foram executadas com duas entradas diferentes conforme foi ilustrado na Tabela 2: a entrada pequena, utilizada durante a fase de treinamento para ambas estratégias (SEA e SPRA, como descritos nas Seções 4.2 e 4.3 respectivamente); e a entrada regular, utilizada para validar e comparar SEA e SPRA com outras técnicas.

Além disso, como pode ser observado na Figura 8, os *benchmarks* selecionados também cobrem diferentes graus de TLP, Instruções por Ciclo (IPC), e comportamentos em relação acessos à memória *cache*. TLP informa o grau de paralelismo em nível de *threads* de uma aplicação de acordo com as definições dos autores em Blake et al. (BLAKE et al., 2010): de forma que a média do total de concorrência durante a execução do programa apresentou pelo menos um *core* ativo. Quanto mais próximos estes valores estiverem de 1.0 (normalizado pelo total de *cores* disponíveis), maior é grau de TLP da aplicação. ST é um exemplo de *benchmark* que possui o menor TLP disponível: menos de 20% da sua execução é realizada em paralelo, independente do sistema *multicore*. Por outro lado, LM, IS e EP são *benchmarks* que apresentam os maiores grau de TLP: mais de 98% da aplicação é executada em paralelo.

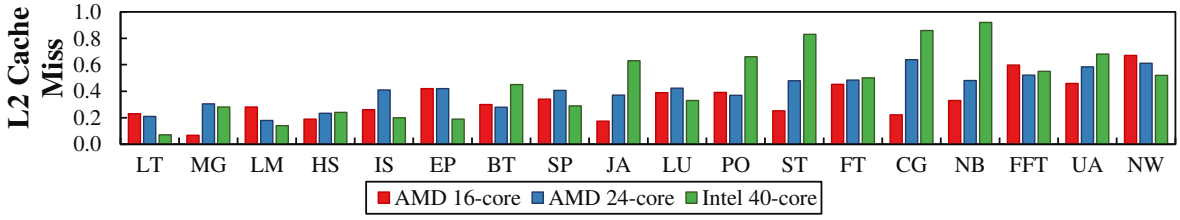
A métrica IPC reflete o grau de ILP disponível da aplicação e quanto esta é computacionalmente intensiva (*CPU-Bound*) ou ainda, se esta realiza muito acessos a memória (*Memory-Bound*). IPC é importante por duas razões principais: com um IPC alto, mais sobrecarga é colocada na microarquitetura (principalmente as que possuem SMT), no qual deve lidar com o problema de saturação no processador (um dos gargalos mais conhecidos da literatura). IPC mais alto também tende a diminuir o tempo de espera de *threads* desbalanceadas (e portanto, diminuir algumas oportunidades de otimização). Por outro lado, aplicações com baixo IPC tem muitos problemas de dependência de dados que devem ser endereçados em tempo de execução para garantir a exatidão dos resultados. Consequentemente, eles demandam uma quantidade grande de comunicação entre as *threads* no qual devem lidar com a saturação do barramento e da memória compartilhada, bem como a execução desbalanceada devido a necessidade de sincronização extra dos dados. Como pode ser observado na Figura 8b os *benchmarks* contemplam diferentes graus de ILP. Por fim, a quantidade de *miss* nas caches L2 e L3 representam o numero de acessos executados pelas *threads* durante a execução da aplicação. Portanto, a alta taxa de *miss* na L2 e L3, indicam que há mais tempo gasto pela aplicação acessando a memória compartilhada, no qual reflete no grau de ILP e TLP. As Figuras 8c e 8d mostram a taxa de *miss* nas caches L2 e L3 para cada *benchmark* em cada um dos sistemas *multicore* quando a aplicação está executando o número de *threads* igual ao número de *cores* disponíveis.



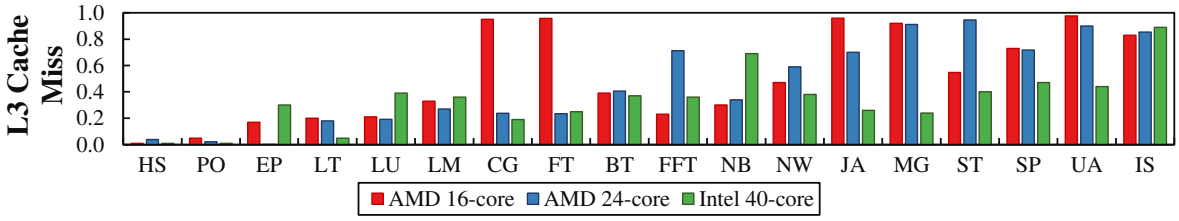
(a) TLP correspondente para cada um dos *benchmarks* em cada ambiente *multicore* - normalizados de acordo com o número máximo de *threads* (e.g, 16 *threads* no AMD 16-*cores*, 24 *threads* para o AMD 24-*cores*, e 40 *threads* para o Intel 40-*cores*)



(b) IPC médio para cada um dos *benchmars* para cada um dos ambientes *multicore*



(c) Taxa de *miss* na cache L2 para cada *benchmark* em cada um dos ambientes *multicore*



(d) Taxa de *miss* na cache L3 para cada *benchmark* em cada um dos ambientes *multicore*

Figura 8 – Principais características de cada *benchmark*

5.2 Ambiente de Execução

Os experimentos foram realizados em três processadores *multicore*, como ilustrado na Tabela 3. A frequência das CPUs foram configuradas para se ajustar de acordo com a carga de trabalho da aplicação, utilizando *ondemand* como *governor* do DVFS, sendo este o padrão aplicado pela maiora dos sistemas operacionais Linux. Os resultados apresentados na capítulo 6 representam a média de dez execuções das aplicações paralelas contendo um desvio padrão menor que 0.05%.

Inicialmente os resultados da metodologia proposta foram comparados com os seguintes cenários:

- **Execução padrão de aplicações paralelas (STD)**: a aplicação foi executada com o número máximo de *threads* disponível no sistema. Isto é o comportamento padrão de qualquer aplicação sem a intervenção do usuário.

Tabela 3 – Arquiteturas *Multicore*

	AMD Ryzen 7 1700	AMD Ryzen 9 3900x	Intel Xeon E5-2650 v3
Microarquitetura	Zen	Zen	Haswell
#*Cores* Físicos	8	12	20
#*Threads*	16	24	40
L1 Cache de Dados	32kB	32kB	32kB
L1 Cache de Instrução	64kB	32kB	32kB
L2 Cache (total)	4MB	6MB	5.1MB
L3 Cache (total)	16MB	64MB	50MB
Memória Principal	16GB	32GB	128GB
Sistema Operacional	Linux v.4.15	Linux v.4.15	Linux v.4.15

- **OMP_Dynamic (OMP_Dyn)**: uma funcionalidade presente na IPP OpenMP que ajusta de maneira dinâmica o número de *threads* para cada região paralela de uma aplicação, objetivando fazer o melhor uso dos recursos do sistema (e.g, memória e processador) (CHAPMAN; JOST; PAS, 2007).

- **Offline(L)**: esta abordagem utiliza o mesmo algoritmo de aprendizado implementado pela estratéiga SEA (4.2) porém utiliza a entrada grande (regular) durante o treinamento. Com isto, o objetivo é medir o quão distante nossa proposta de inferência está quando comparado com os resultados obtidos pelo treinamento utilizando os dados originais. Em outras palavras, analisamos a compensação entre ser menos preciso e mais rápido devido ao nosso método de inferência proposto e ser mais lento durante o aprendizado (usando um conjunto de dados maior), mas potencialmente mais preciso.

- **Oracle**: neste cenário a aplicação é executada com o número de *threads* ideal para cada região paralela sobre os dados de entrada regular (entrada grande). O número de *threads* ideal foi obtido através de uma busca exaustiva em cada uma das regiões paralelas de cada aplicação sendo 1 até n *threads*, onde n é o número máximo de *threads* suportado pelo *hardware*. Então, foi pré-configurado a aplicação para utilizar a quantidade de *threads* encontrada anteriormente. Sendo assim, esta foi a que obteve o melhor resultado de EDP para cada aplicação sem o custo de aprendizado. Com esta configuração, foi possível medir quão distante as soluções encontrada pelas estratégias SEA e SPRA propostas neste trabalho estão distantes dos resultados ótimos.

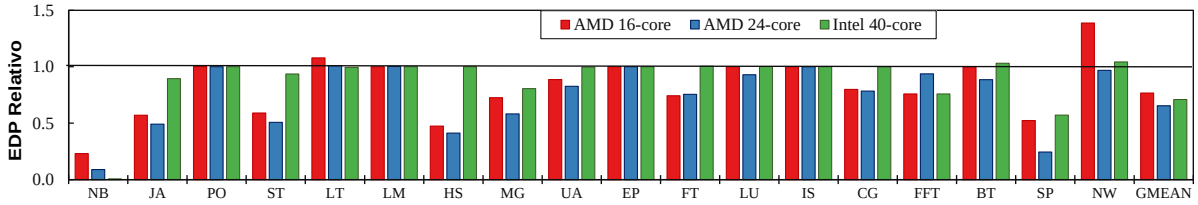
6 RESULTADOS EXPERIMENTAIS

Este capítulo tem como finalidade apresentar os resultados obtidos após serem realizadas diferentes análises envolvendo o *framework* Hórus. De modo que na Seção 6.1 constam os resultados comparativos de EDP entre as estratégias de busca SEA e SPRA em relação aos cenários (STD e OMP_Dynamic) no qual foram definidos na Seção 5.2. A Seção 6.2 exibe os resultados referente a convergência de cada estratégia de busca, ou seja, se estas são capazes ou não de encontrar uma configuração ótima que deverá entregar o menor EDP para uma aplicação paralela. Além desss resultados, é mostrado neste capítulo, a diferença entre os valores de EDP encontrados entre o melhor resultado possível (*Oracle*) e as estratégias de busca utilizadas pelo Hórus na Seção 6.3. Por fim, a Seção 6.4 apresenta uma breve análise sobre os resultados obtidos e, prover diretrizes para os desenvolvedores de *software* que pretendem utilizar Hórus como ferramenta para otimizar suas aplicações.

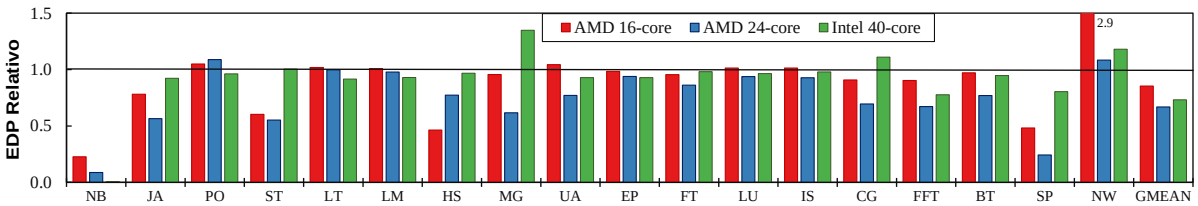
6.1 Resultados para EDP

Nesta seção, discute-se os resultados para EDP encontrados pelas estratégias de pesquisa SEA e SPRA do *framework* Hórus ao executar a aplicação com a entrada grande ajustada com o número de *threads* encontrados durante a fase de aprendizagem utilizando a entrada pequena.

A Figura 9 apresenta os resultados para todo o conjunto de *benchmarks* ao executar o conjunto de entrada grande, junto com a média geométrica (GMEAN) considerando os três ambientes *multicores*. As Figuras 9a e 9b comparam as estratégias de pesquisa SEA e SPRA em relação a execução STD da aplicação (representada pela linha preta). Os resultados estão normalizados de acordo com STD, ou seja, valores menores que 1.0 significam que as estratéfias SEA e SPRA são melhores que STD.

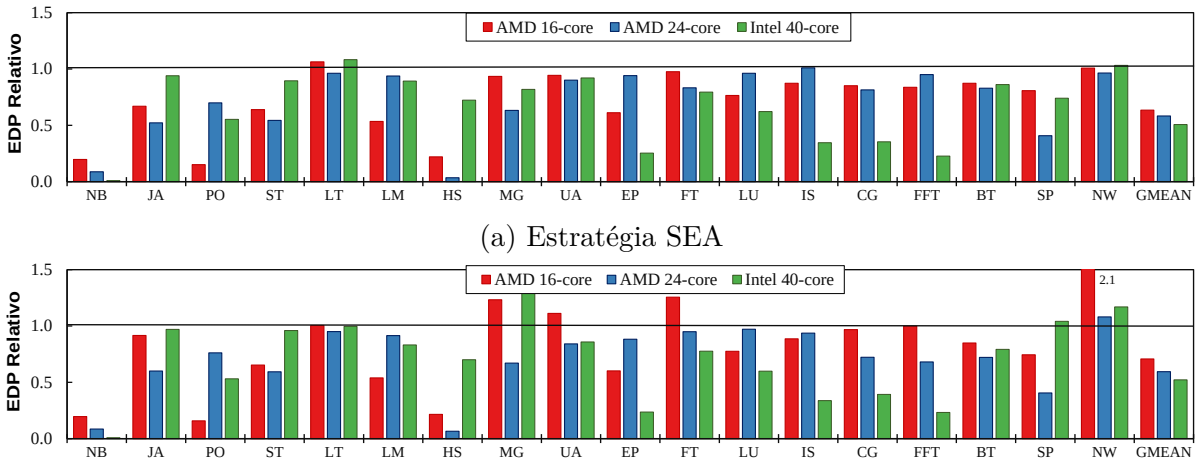


(a) SEA estratégia de busca



(b) SPRA estratégia de busca

Figura 9 – Resultados para EDP para cada estratégia de busca comparado com o STD – valores menores que 1.0 significa que a estratégia proposta é melhor



(a) Estratégia SEA

(b) Estratégia SPRA

Figura 10 – Resultados de para cada estratégia de busca comparado ao OpenMP Dynamic – valores menores que 1.0 significa que a estratégia proposta é melhor

SEA e SPRA vs STD. Ambas as estratégias SEA e SPRA mostram ganhos de EDP na maioria dos casos em relação ao STD como pode ser visto na Figura 9. Em alguns casos específicos, executar a aplicação com o melhor número de *threads* possui o menor EDP, as estratégias SEA e SPRA possuem resultados muito similares ao STD. Isto se da pelas características do algoritmo de aprendizado discutidos no Capítulo 4. O melhor resultado foi obtido para a aplicação NB no qual é uma aplicação que apresenta baixo grau de TLP e assim, o melhor EDP é alcançado com um número de *threads* menor do que com a execução com o número de *threads* disponíveis no sistema. Neste caso a melhoria de EDP é de 99% em ambas as estratégias para o ambiente Intel 40-*core*. Quando é considerado a média geométrica (o conjunto inteiro de *benchmark* e os ambientes *multicore*), SEA e SPRA provem melhorias de 30% e 26% para EDP respectivamente.

SEA e SPRA vs OpenMP Dynamic. Como pode ser observado na Figura 10, SEA e SPRA provêm melhor EDP em relação ao recurso *OpenMP Dynamic* presente no OpenMP. Isto acontece pelas características de implementação do *OpenMP_Dynamic*. Ou seja, o recurso não utiliza nenhum algoritmo de busca e também não considera o ambiente da aplicação paralela para definir um número de *threads*. Ao invés disso, *OpenMP Dynamic* se baseia nos útlimos quinze minutos de execução da aplicação para definir o grau de exploração de TLP para cada região paralela (OpenMP Architecture Review Board, 2021). Portanto, na maioria dos casos, o número de *threads* definido para executar uma dada aplicação acaba acarretando em resultados de EDP muito distante do ideal. O melhor cenário para as estratégias *SEA* e *SPRA* é para o *benchmark* NB: a diferença de EDP é de 99% para o sistema com Intel 40-*core*. Se considerarmos a média geométrica para o conjunto inteiro de *benchmark* e processadores, *SEA* e *SPRA* atingem melhorias para EDP de 43% e 40% respectivamente.

SEA vs SPRA. A Figura 11 apresenta os resultados de EDP para cada *ben-*

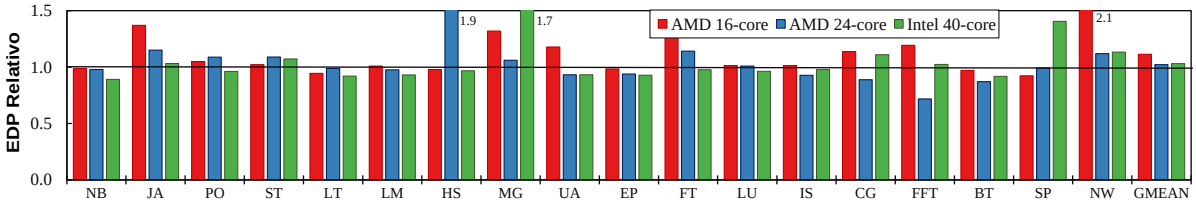


Figura 11 – SEA vs. SPRA: EDP normalizado para os resultados de SEA, sendo assim, os resultados menores que 1.0 significam que SPRA é melhor que a estratégia de busca SEA

chmark em cada sistema *multicore* junto com a média geométrica. Os resultados estão normalizados para a estratégia SEA, deste modo, os valores menores que 1.0 significam que SPRA é melhor. Devido ao fato da estratégia SPRA buscar por um número ideal de *threads* para cada região paralela, isto garante um menor EDP em aplicações onde cada região é melhor executada com um grau de TLP diferente (e.g., UA, CG, FFT, BT, e SP) como mostrado na tabela 5. O melhor resultado para a estratégia SPRA é para o *benchmark* FFT executado no sistema AMD 24-*cores*: EDP 29% menor que SEA. Por outro lado, esta adaptabilidade em uma granularidade mais fina provida pela estratégia SPRA não garante o melhor resultado para aplicações onde: (*i*) o ambiente de cada região paralela quando o tamanho de entrada muda (HS, MG e NW); e (*ii*) a sobrecarga para mudar o número de *threads* entre duas regiões paralelas diferentes durante a execução supera os ganhos obtidos ao encontrar a melhor combinação durante a fase de aprendizado.

Estes dois pontos serão discutidos com mais detalhe na seção 6.2.2. Consequentemente, o pior caso de SPRA é para a execução de NW para o ambiente AMD 16-*cores*, de modo que o resultado para EDP é 1.1 vezes pior que a estratégia SEA. Quando consideramos a média geométrica o conjunto inteiro de *benchmarks*, SEA foi capaz de atigir melhores resultados de EDP: 13% menor no AMD 16-*cores*; 2.1% no AMD 24-*cores*; e 3.1% no Intel 40-*core*.

6.2 Convergência dos algoritmos de aprendizado

Esta seção discute sobre como os algoritmos de aprendizado implementados pelas estratégias SEA e SPRA conseguem convergir para um número ideal ou muito próximo ao ideal de *threads*, com o intuito de se obter o menor EDP utilizando durante a fase de treinamento apenas uma pequena entrada da aplicação original. Portanto, a estratégia SEA que busca um número de *threads* ideal para aplicação inteira é analizada na Seção 6.2.1. Por fim, a estratégia SPRA que visa otimizar a aplicação com base em uma granularidade mais fina (número de *threads* por região paralela) é apresentado na Seção 6.2.2.

6.2.1 Análise da Estratégia SEA

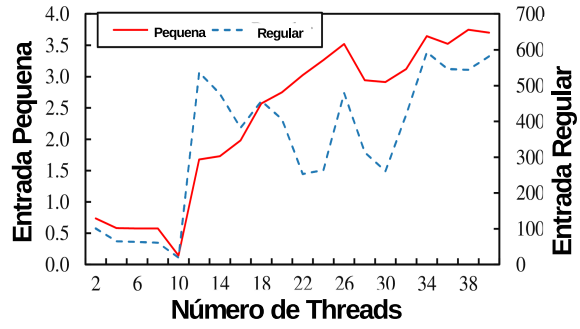
A Tabela 4 exibe o número de *threads* encontrado durante a fase de aprendizado da estratégia SEA para executar cada aplicação com o tamanho da entrada grande em cada um dos processadores. Comparou-se estes valores ao número de *threads* encontrado pela estratégia *Offline(L)* (como descrito na Seção 5.2). Como pode ser observado, a estratégia SEA foi capaz de obter tanto o número ótimo ou muito próximo do melhor número de *threads* para a maioria dos *benchmarks*.

SEA foi capaz de encontrar o número ideal ou muito próximo do ideal devido as seguintes situações: (*i*) aplicações onde apenas o número de iterações mudou da entrada pequena para a grande (de NB até HS); e (*ii*) aplicações das quais o número de iterações e o tamanho da estrutura de dados mudaram (e.g., EP, FT, LU, IS, CG, e FFT). Sendo assim, SEA conseguiu convergir para uma configuração ideal de modo que a restrição de escalabilidade (e.g., saturação da largura de banda da memória, concorrência ao acesso à memória, e sincronização de dados) (SULEMAN; QURESHI; PATT, 2008), (LORENZON et al., 2019), (SUBRAMANIAN et al., 2013),(JOAO et al., 2012),(RAASCH; REINHARDT, 2003), (LEVY et al., 1996) de aplicações paralelas é o mesmo para ambas as entradas. Portanto, a curva de EDP tem um ambiente similar para a entrada pequena e grande quando o número de *threads* varia. Como um exemplo, vamos considerar o ambiente da aplicação ST no sistema Intel 40-*core*, mostrado na Figura 12. Esta é uma aplicação que opera sobre uma grande quantidade de dados compartilhados entre todas as *threads*. Porém, existe um ponto ótimo onde a sobrecarga imposta pelos acessos a memória compartilhada não sobrepõe os ganhos pela exploração do paralelismo. A Figura 12b mostra que este ponto é a execução com mais do que 10 *threads*, de tal maneira que o número de acessos concorrentes à memória compartilhada aumenta significativamente (Figura 12a), e não há melhorias futuras de EDP.

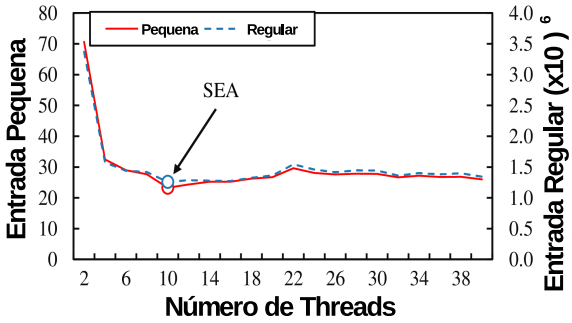
Tabela 4 – Número de *threads* encontrado pela estratégia SEA e a estratégia *Offline(L)* para todos os *benchmarks*

		NB	**JA**	**PO**	**ST**	**LT**	**LM**	**HS**	**MG**	**UA**	**EP**	**FT**	**LU**	**IS**	**CG**	**FFT**	**BT**	**SP**	**NW**
AMD	***SEA***	8	2	16	2	12	16	15	10	8	16	3	16	16	8	10	12	3	14
16-core	***Offline(L)***	8	2	16	2	12	16	15	5	8	16	3	16	16	6	10	16	2	16
AMD	***SEA***	3	2	12	2	24	24	23	6	10	24	3	11	24	8	7	12	7	6
24-core	***Offline(L)***	3	2	12	2	24	24	23	4	6	24	3	10	24	5	5	10	4	8
Intel	***SEA***	4	12	40	10	36	40	40	8	20	40	8	40	40	40	40	40	20	28
40-core	***Offline(L)***	4	12	40	10	36	40	40	12	20	40	10	40	40	40	40	40	16	40

Outro exemplo é a aplicação NB, onde o fator limitante para a falta de escalabilidade é a sincronização dos dados. Consequentemente, quanto maior o número de *threads*, maior é o tempo de sincronização, no qual eventualmente deve piorar o EDP quanto mais *threads* são executadas concorrentemente. Este ambiente é mostrado na 13 para a execução com a entrada pequena e grande no ambiente AMD 16-*core*. O total de EDP reduz quando o número de *threads* aumenta de 1 até 8 em ambas entradas. Entretanto, a partir

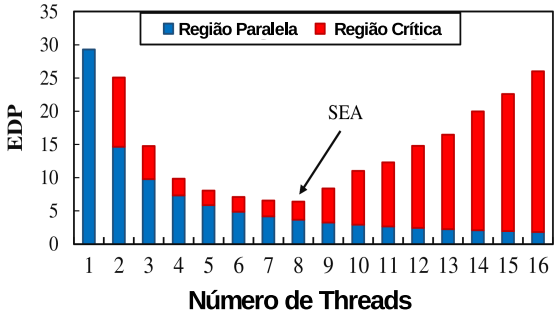


(a) Número de acessos a memória compartilhada

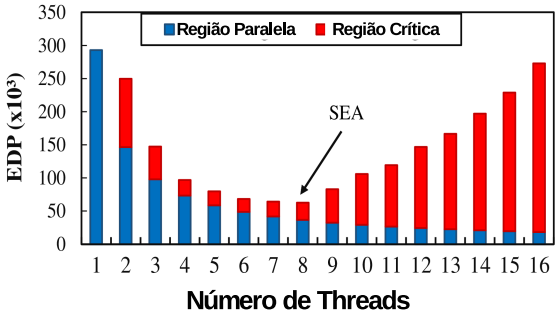


(b) Resultados Para EDP

Figura 12 – Ambiente de execução do *benchmark* ST no sistema Intel 40-*core*



(a) NB - Entrada Pequena



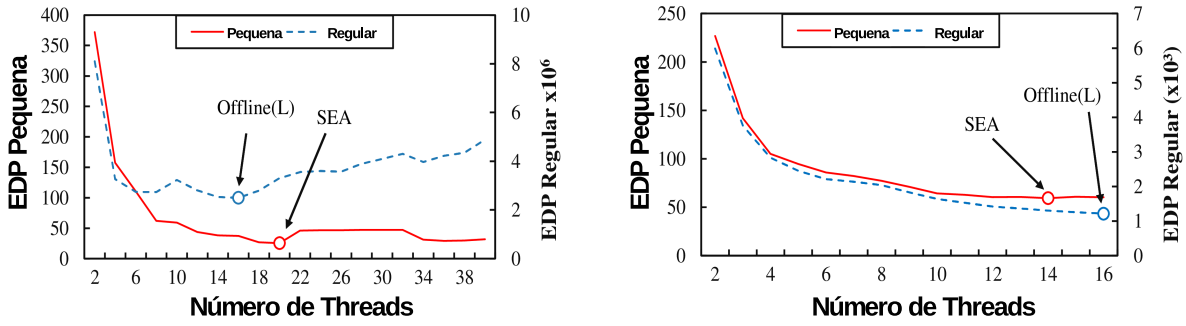
(b) NB - Entrada Regular

Figura 13 – Ambiente de execução do *benchmark* NB para o sistema AMD 16-*core*

deste ponto em diante, a sincronização consome mais EDP do que a execução de código da região paralela.

Apesar da estratégia SEA não conseguir encontrar um número ideal de *threads* em alguns casos, essa estratégia de busca foi capaz de prover reduções significativas de EDP ao encontrar um grau de TLP que provê resultados similares ao ideal. Isto pode ser observado na execução do *benchmark* SP no sistema Intel 40-*core* na Figura 14a. Durante a fase de aprendizado da estratégia, SEA encontrou 20 como o número de *threads* que entregaria o melhor EDP para a entrada grande, no qual não foi este o melhor resultado. No entanto, por encontrar um número de *threads* muito próximo do ideal (20 ao invés de 16 *threads*), o EDP foi reduzido em 43% quando comparado a execução padrão desta aplicação (**STD**) 9a. Ambiente similar foi observado também para os seguintes *benchmarks*: MG, CG, e SP para o ambiente com AMD 16-*core*; FT, LU, MG, CG, UA, BT, SP, e NW para o sistema com AMD 24-*core*; e MG, FFT, para o Intel 40-*core*.

Por outro lado, existem alguns casos que a estratégia SEA não é apta a encontrar um número de *threads* ótimo provendo um resultado de EDP pior que a execução padrão (STD). Como exemplo, tem-se a execução da aplicação NW no ambiente AMD 16-*core* 14b. Durante a fase de aprendizado foi encontrado 14 como sendo o número de *threads* que entregaria o melhor EDP, mas o melhor número de *threads* para ser executado com



(a) Execução SP no Intel 40-*core* (b) Execução NW no AMD 16-*core*

Figura 14 – Ambiente para EDP dos *benchmarks* SP e NW

a entrada grande é 16. Por não encontrar um número de *threads* próximo do ideal, o EDP foi aumentado em 37% quando comparado com a execução STD (no qual detém o melhor resultado). O mesmo ambiente também foi observado para o *benchmark* LT para o ambiente AMD 16-*core*; e BT e NW para o sistema com Intel 40-*core*

6.2.2 Análise da Estratégia SPRA

Nesta seção, será discutido como a estratégia SPRA é capaz de convergir para um grau de TLP ideal ou muito próximo ao ideal para cada região paralela. Para isto, a Tabela 5 exibe o número de *threads* encontrado durante a fase de aprendizado para cada região da aplicação. Estes valores encontrados foram comparados com o número ideal encontrado pela solução Oracle (descrita na seção 5.2). Como pode ser observado na Tabela 5, para aplicações com mais de uma região paralela, o número ideal de *threads* para cada região é usualmente diferente. Como um exemplo, vamos considerar a execução da aplicação PO no sistema AMD 16-*core*. Este *benchmark* possui cinco regiões paralelas, de tal modo que 16 *threads* é o melhor para a primeira e terceira região; 10 para a segunda; 2 para a quarta; e 12 *threads* para a quinta região. Consequentemente, pelo fato de encontrar o grau de TLP ideal ou muito próximo ao ideal para cada região paralela, SPRA foi capaz de garantir resultados melhores para EDP na maioria dos casos em comparação a execução da aplicação de maneira padrão (9b)

Devido ao fato de algumas aplicações apresentarem o mesmo ambiente para EDP quando o conjunto de entrada muda de pequeno para grande, a fase de aprendizado da estratégia SPRA foi capaz de de convergir para para um grau de TLP ideal para cada região. Como um exemplo disso, tem-se à execução da aplicação LT no ambiente INTEL 40-*core*, como pode ser observado na Tabela 5 e na Figura 15. Esta aplicação possui três regiões paralelas, e a estratégia SPRA encontrou o mesmo número de *threads* que a solução ORACLE para cada uma dessas regiões. Dado o fato desta estratégia de busca ser de uma granularidade mais fina, o EDP de SPRA é 8% menor que a estratégia SEA, no qual essa não está apta a otimizar cada região paralela individualmente. Ambiente

Tabela 5 – Número de *Threads* encontrado pela estratégia SPRA para cada Região Paralela para todos os *Benchmarks*

	AMD 16-core		**AMD 24-core**		**Intel 40-core**	
	SRPA	***Oracle***	***SRPA***	***Oracle***	***SRPA***	***Oracle***
NB	8	8	3	3	4	4
JA	2	2	2	2	12	12
PO	16, 10, 16 16, 16	16, 10, 16 2, 12	14, 24, 24 24, 24	16, 24, 12 12, 12	40,10,40 40, 40	40,10,40 40, 40
ST	16, 16, 4	16, 16, 2	24, 24, 4	24, 24, 3	40, 40, 10	40, 40, 10
LT	16, 16, 12	16, 16, 12	24, 24, 22	24, 24, 22	40, 40, 38	40,40,38
LM	16	16	24	24	40	40
HS	15	15	22	23	30	30
MG	2, 2, 2, 2 2, 4, 4, 2 2, 2, 16	2, 2, 2, 2 4, 4, 2, 8 2, 2, 16	6, 3, 3, 3 6, 6, 3, 3 3, 3, 24	6, 3, 3, 3 6, 6, 3, 12 3, 3, 24	5, 40, 40, 40, 5 5, 5, 10, 5 5, 40	5, 5, 5, 5, 10 10, 10, 10, 5 5, 40
UA	2, 2, 16, 16 16, 16, 2, 2 2, 4, 4, 2, 2 4, 4, 8, 4, 2 4, 2, 2, 2, 4 4, 2, 2, 2, 2 2, 2, 4, 8, 8 4, 2, 4, 4, 5 4, 4, 4, 2, 4 2, 4, 8, 4, 4 2, 2, 2, 11, 4 2, 16, 16	12, 2, 16, 16 16, 16, 2, 2 2, 4, 8, 12, 10 12, 5, 8, 4, 2 2, 2, 2, 2, 4 4, 10, 4, 8, 4 10, 8, 5, 16, 14 2, 2, 12, 12, 8 8,14, 11, 2, 4 12, 2, 2, 8, 12 5, 5, 4, 5, 5 14, 16, 16	3, 3, 24, 24 24, 3, 3, 3 3, 3, 3, 3, 3 3, 6, 3, 3, 3 3, 3, 6, 3, 3 3, 3, 3, 3, 3 3, 3, 6, 7, 3 3, 3, 3, 6, 6 3, 3, 7, 6, 3 3, 6, 3, 3, 7 3, 3, 3, 3, 3 12, 24, 24	6, 3, 24, 24 24, 24, 3, 3 3, 3, 12, 7, 6 12, 12, 12, 3, 6 6, 3, 3, 6, 3 3, 7, 6, 12, 3 6, 6, 7, 24, 24 7, 3, 12, 12, 12 12, 12, 6, 12, 6 12, 6, 3, 12, 18 3, 7, 7, 7, 6 12, 24, 24	5, 5, 40, 40 40, 40, 5, 5 5, 5, 12, 5, 5 5, 5, 10, 5, 5 5, 5, 5, 10, 5 5, 5, 5, 10, 5 12, 5, 5, 20, 20 13, 5, 10, 10, 10 10, 12, 10, 13, 10 20, 10, 10, 10, 10 20, 5, 5, 5, 10 20, 40, 40	20, 5, 40, 40 40, 40, 5, 5 5, 5, 37, 10, 10 20, 38, 37, 5, 5 5, 5, 5, 5, 5 5, 5, 10, 27, 5 13, 10, 20, 38, 40 27, 13, 10, 20, 35 35, 20, 20, 20, 5 10, 20, 10, 20, 35 20, 10, 10, 5, 20 37, 40, 40
EP	16, 16 16, 16	16, 16 16, 16	24, 24 24, 24	24, 24 24, 24	40, 40 40, 40	40, 40 40, 40
FT	16, 2, 2, 4 8, 8, 4, 4 16	16, 2, 2, 4 2, 8, 4, 4 16	24, 3, 3, 12 12, 12, 3, 12 24	24, 3, 3, 6 6, 6, 3, 6 24	40, 5, 5, 20 20, 20, 5, 5 40	40, 5, 5, 40 40, 20, 5, 10 40
LU	2, 2, 16, 2 8, 5, 2, 16 16, 16	2, 2, 16, 2 4, 4, 14, 16 16, 16	24, 3, 24, 3 21, 6, 12, 24 24, 24	3, 3, 24, 3 6, 7, 12, 24 24, 24	40, 5, 40, 5 28, 10, 35, 40 40, 40	5, 5, 40, 5 40, 12, 40, 40 40, 40
IS	16, 16, 8 16, 16	16, 16, 16 16, 16	24, 24, 21 24, 24	24, 24, 21 24, 24	40, 40, 28 40, 40	40, 40, 40 40, 40
CG	16, 8, 16, 16 16, 2, 2, 16	16, 4, 16, 16 16, 8, 2, 16	23, 4, 24, 24 24, 3, 3, 24	24, 3, 24, 24 24, 3, 3, 24	40, 35, 40, 40 40, 5, 5, 40	40, 40, 40, 40 40, 20, 10, 40
FFT	4, 5, 2	4, 5, 2	3, 7, 3	6, 7, 3	5,5,5	5,38,5
BT	2, 16, 10, 8 8, 12, 4, 16, 16, 16	2, 16, 2, 15 12, 16, 2, 16 16, 16	2, 24, 6, 12 12, 22, 2, 24 24, 24	12, 8, 2, 24 24, 24, 2, 24 24, 24	5, 40, 37, 35 40, 35, 10, 40 40, 40	5, 40, 20, 40 40, 40, 5, 40 40, 40
SP	16, 2, 2, 2 8, 2, 8, 2 4, 2, 2, 16, 16, 16	16, 2, 2, 2 2, 2, 2, 2 2, 2, 2, 16 16, 16	24, 3, 3, 6 12, 3, 6, 3 6, 6, 3, 24 24, 24	24, 3, 3, 6 6, 3, 6, 3 6, 3, 3, 24 24, 24	40, 5, 28, 20 37, 20, 20, 20 35, 10, 20, 40 40, 40	40, 5, 20, 20 5, 10, 13, 5 13, 13, 12, 40 40, 40
NW	2,8	2,16	3, 16	3, 22	5, 13	5, 28

muito similar, no qual SPRA convergiu para um número ideal de *threads* para cada região paralela, foi observado também para as seguintes aplicações: NB, JA, LT, LM, HS, e EP para o sistema com AMD 16-*core*; NB, JA, LT, LM, EP, e IS para o sistema com AMD-24-*core*; e NB, JA, PO, ST, LT, LM, HS, e EP para o ambiente com Intel 40-*core*.

Porém, existem alguns casos no qual a estratégia SPRA não convergiu para um número de *threads* ideal mas alcançou um valor muito próximo desse. Um exemplo desse cenário é a execução da aplicação CG no sistema com AMD 24-*core*. Figura 16a exibe o EDP para as regiões paralelas #1 e #2 quando executando cada uma das regiões com diferentes números de *threads* (de 2 a 24). Devido a falta de escalabilidade (TLP

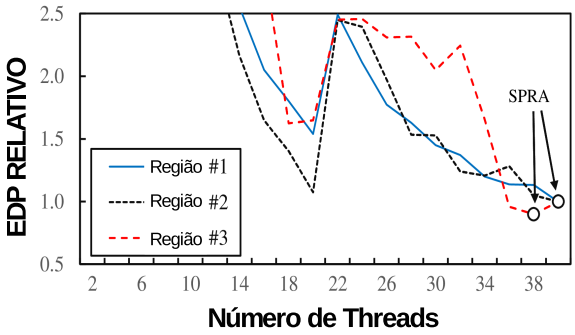
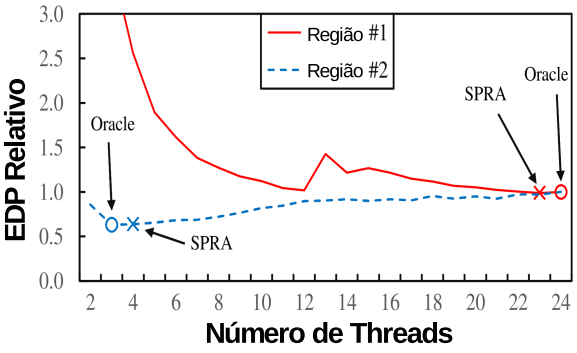
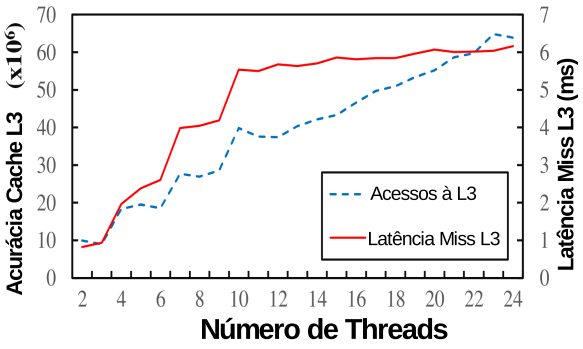


Figura 15 – Ambiente de EDP para cada uma das regiões paralelas da aplicação LT no sistema Intel 40-*core*. EDP está normalizado para a execução com 40 *threads*

limitado) da região 2, o melhor resultado alcançado é apenas com 3 *threads*. Esta falta de escalabilidade é um resultado do alto acesso da memória compartilhada, como pode ser visto na Figura 16b. Isto mostra que o número de acessos a cache L3 e o atraso para cada *miss* na L3 (no qual é a única compartilhada com todos os *cores*) cresce significantemente conforme aumenta-se do número de *threads*. Como resultado, o EDP aumenta conforme se executa esta região com mais de três *threads* (Figura 16a). Por outro lado, devido ao fato da região #1 possuir um grau de TLP maior (alta escalabilidade), o melhor resultado para esta é alcançado com o número máximo de *threads* disponível no sistema.



(a) Ambiente de EDP para as regiões #1 e #2 normalizado em relação a execução com 24 *threads*



(b) Ambiente para miss e latência na cache L3 para a região #2.

Figura 16 – Ambiente das duas regiões paralelas do *benchmark* CG no sistema AMD 24-*core*

Quando a estratégia SPRA é executada para encontrar uma configuração de *threads* ideal para o *benchmark* CG sobre a entrada pequena, o algoritmo de aprendizado encontrou 23 e 4 *threads* como sendo as melhores configurações para as regiões #1 e #2, respectivamente, no qual não é o correto. Entretando, apesar disso, por esta configuração estar muito próxima do número de *threads* ideal para cada região, esta estratégia obteve redução de 12% para EDP em comparação com a estratégia SEA. Ambiente muito similar, mas em taxas diferentes, puderam ser observadas para os seguintes *benchmarks* na Figura

11: SP para o AMD 16-*core*; UA, FFT e BT para o AMD 24-*core*; UA e BT para o Intel 40-*core*.

Por outro lado, a estratégia SPRA não foi capaz de convergir para uma solução próxima ao ideal em aplicações das quais cada região paralela não possui o mesmo comportamento de EDP quando tamanhos de entradas diferentes são utilizadas. Um exemplo deste cenário é a aplicação NW, independente do sistema *multicore* na qual foi submetida a teste como mostrado na Tabela 5. Isto é causado pelo fato de existirem problemas tanto de *hardware* e *software* que afetam a escalabilidade quando se executa a aplicação NW com uma entrada pequena e uma entrada grande (SULEMAN; QURESHI; PATT, 2008). Isto é, quando submete-se a execução de NW com uma entrada pequena, os fatores mencionados anteriormente não afetam tanto a execução da aplicação, porém, quando o tamanho da entrada aumenta, estes fatores acabam influenciando significativamente no comportamento, gerando assim, comportamentos diferentes de EDP para ambos os tamanhos de entrada, impossibilitando a estratégia SPRA convergir para um número de *threads* ideal ou muito próximo ao ideal. Desse modo, o EDP resultante foi pior do que todas as estratégias (STD, OMP_Dynamic, e SEA). De modo análogo, isto também foi observado para os *benchmarks* MG e CG para o ambiente Intel 40-*core*.

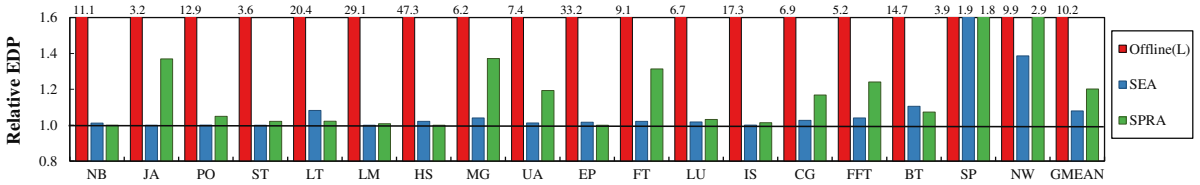
6.3 Diferença para a Solução Ótima

Esta seção contém os resultados comparativos encontrados pelas estratégias de busca SEA e SPRA (EDP utilizado durante a fase de aprendizado mais a execução com a entrada regular) em relação a solução *Oracle*, tal como foi definido na Seção 5.2.

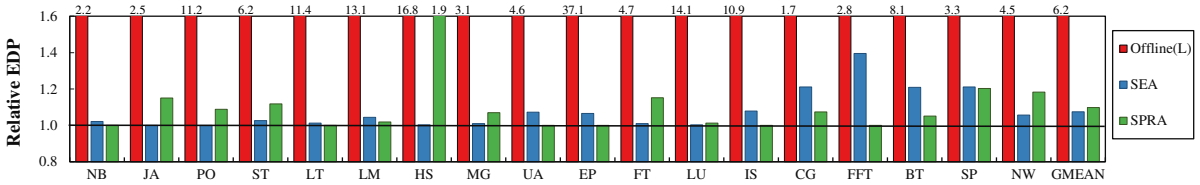
Oracle representa a execução de uma aplicação que contém o número de *threads* ideal para cada região, obtido através de uma busca exaustiva em todos os números de *threads* disponíveis (para cada região paralela) para cada aplicação. Como cada região paralela possui sua própria configuração que pode ser diferente da configuração de outra região paralela, o espaço de exploração e projeto cresce exponencialmente com o número de regiões paralelas. Portanto, uma vez que uma região paralela pode ter uma configuração diferente dos seus vizinhos, o espaço de exploração é muito grande, e uma busca exaustiva completa torna-se impraticável.

Por exemplo, considere uma aplicação com apenas cinco regiões paralelas executando em um ambiente com suporte a 24 *threads*. Para encontrar o melhor número de *threads* para cada região, seria necessário executar esta aplicação pelo menos 24^5 vezes (pois existem 24^5 conbinações possíveis de número de *threads* para serem executadas em cada região paralela). Se considerarmos que cada execução com a entrada grande (regular) fosse executada em torno de 30 segundos (sendo muito otimista), seriam necessários mais de 7 anos para serem testadas todas as possibilidades e encontrar uma solução ótima para apenas uma entrada em particular para um único processador em específico. Além disso, esta busca teria que ser realizada para cada nova entrada e para cada novo ambiente.

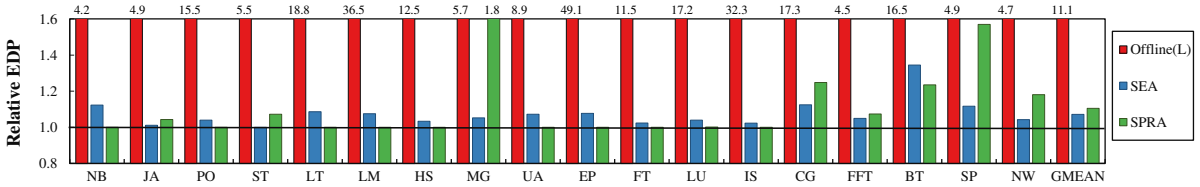
Pelo motivo apresentado anteriormente, foi verificado então a necessidade em experimentar uma busca exaustiva que de alguma forma fosse diminuído o espaço de exploração, para que este pudesse ser realizado em um tempo viável. Tal busca considera diferentes configurações para cada região paralela. Mesmo em tais casos onde o espaço de exploração é muito grande, foi realizado uma busca exaustiva pelo melhor resultado variando o número de *threads* de cada região paralela de forma individual, não considerando uma busca ao mesmo tempo (i.e., foi executado uma busca exaustiva local para cada região paralela, evitando o espaço de busca com crescimento exponencial). No entanto, é possível observar que utilizar esta abordagem não gera influências de uma região sobre a outra. Mesmo que essas buscas exaustivas limitadas provavelmente não convergirão para o melhor resultado, muito provavelmente irão convergir para um resultado muito próximo ao ideal.



(a) Sistema AMD 16-*core*



(b) Sistema AMD 24-*core*



(c) Sistema Intel 40-*core*

Figura 17 – EDP de cada estratégia normalizado em relação a solução *Oracle*

A Figura 17 exibe a diferença de cada estratégia de busca (*Offline(L), SEA e SPRA*) comparado com a solução *Oracle* que reresenta o melhor EDP para cada aplicação em cada ambiente *multicore*. Primeiramente, vamos discutir a sobrecarga gerada pela estratégia SEA em relação ao *Oracle*, que é causada por duas situações, (*i*) o custo durante a fase de aprendizado, representado pela execução do algoritmo de busca na entrada pequena; e (*ii*) a execução da aplicação paralela utilizando a entrada grande (regular) com a solução encontrada durante a busca, que é dado em número de *threads*, de modo que este nem sempre é o número ideal (encontrado pelo *Oracle*). A sobrecarga causada em cada uma das situações pode ser visualizada na tabela 6. As menores sobrecargas foram obtidas para aplicações onde a diferença entre a entrada pequena e a entrada grande se

Tabela 6 – Porcentagem de EDP gasto durante a fase de aprendizado (Custo de aprendizado) de cada estratégia e a sobrecarga em EDP devido a execução de cada aplicação com a solução não ideal.

		AMD 16-core		AMD 24-core		Intel 40-core	
		Custo Aprendizado	Solução Não ideal	Custo Aprendizado	Solução Não ideal	Custo Aprendizado	Solução Não Ideal
SEA	NB	1.2%	–	2.2%	–	12.3%	–
	JA	0.1%	–	0.1%	–	1.1%	–
	PO	0.1%	–	0.1%	–	4.0%	–
	ST	0.1%	–	2.6%	–	0.1%	–
	LT	8.2%	–	1.2%	–	8.6%	–
	LM	0.1%	–	4.4%	–	7.5%	–
	HS	2.2%	–	0.3%	–	3.4%	–
	MG	0.8%	3.3%	0.3%	0.6%	1.3%	4.0%
	UA	1.3%	–	1.3%	6.0%	7.3%	–
	EP	1.7%	–	6.6%	–	7.8%	–
	FT	2.2%	–	1.0%	–	1.1%	1.3%
	LU	1.9%	–	0.1%	0.2%	4.0%	–
	IS	0.1%	–	7.8%	–	2.3%	–
	CG	2.7%	–	1.8%	19.3%	12.4%	–
	FFT	4.0%	–	7.1%	32.5%	5.0%	–
	BT	2.2%	8.3%	4.1%	16.8%	34.5%	–
	SP	9.3%	40.8%	4.5%	16.6%	4.1%	7.6%
	NW	5.2%	33.5%	1.8%	3.9%	1.7%	2.6%
SPRA	NB	0.1%	–	0.3%	–	0.4%	–
	JA	37.0%	–	15.0%	–	4.3%	–
	PO	5.0%	–	5.1%	3.7%	0.9%	–
	ST	1.0%	1.3%	8.2%	3.6%	7.2%	–
	LT	2.2%	–	1.0%	–	0.3%	–
	LM	0.9%	–	1.9%	–	0.3%	–
	HS	0.1%	–	12.4%	42.0%	0.4%	–
	MG	7.7%	29.5%	1.9%	5.1%	15.0%	32.9%
	UA	8.4%	11.0%	0.3%	–	0.3%	0.7%
	EP	0.4%	–	0.7%	–	0.8%	–
	FT	17.9%	13.5%	6.8%	8.4%	0.7%	0.5%
	LU	1.0%	2.2%	0.2%	1.0%	0.2%	0.6%
	IS	0.4%	1.0%	0.1%	–	0.2%	0.4%
	CG	4.8%	12.0%	6.2%	1.2%	8.2%	16.6%
	FFT	24.1%	–	0.2%	–	1.1%	6.3%
	BT	2.5%	4.8%	1.5%	3.7%	2.2%	21.3%
	SP	10.3%	37.9%	7.9%	12.4%	9.0%	21.6%
	NW	18.3%	36.8%	9.9%	8.4%	4.5%	13.6%

dá apenas no número de iterações (de NB até HS). Esta baixa sobrecarga ocorre porque a carga de trabalho a ser realizada por cada *benchmark* é similar quando o conjunto de entrada varia, de modo que não há mudanças na estrutura de dados. Para aplicações onde a estrutura de dados também varia, há um aumeto considerável na sobrecarga, entre a entrada pequena e a entrada grande (Do MG até FFT). Por outro lado, em aplicações onde há variações somente na estrutura de dados, SEA apresenta os maiores valores para sobrecarga (Do BT ao NW). Isto ocorre pelo fato de haver mudanças na carga de trabalho quando o tamanho varia da entrada pequena para a grande, reduzindo o sucesso da estratégia SEA de encontrar o melhor número de *threads*. Entretanto, se considerarmos a média geométrica de sobrecarga para todos os conjuntos de *benchmarks* em cada sistema *multicore*, a diferença para a solução *Oracle* é de apenas 7%.

Quando consideramos a sobrecarga imposta pela estratégia SPRA, parte da sua sobrecarga também é gerada pelos mesmos motivos (*i*) e (*ii*) da estratégia SEA. Contudo,

devido ao fato da estratégia SPRA atuar com uma granularidade mais fina (número de *threads* para cada região paralela da aplicação), a situação (*ii*) também é afetada pelo custo de migração dos dados quando o número de *threads* que executa uma região R_i é diferente daquela que executa a região R_{i+1}. Para entender melhor este cenário, vamos considerar a execução da aplicação MG no sistema Intel 40-*core* (tabela 5). Quando uma região paralela muda de R_4 para R_5, o número de *threads* encontrado por SPRA também muda (40 para 5). Desse modo, os dados alocados para 40 *cores* durante a execução da região R_4 deve ser migrada para as memórias *cache* de 5 *cores* que irão executar a região R_5. Esta migração provoca custos extras relacionado ao aquecimento da cache, *miss* no último nível de *cache*, por exemplo, o que acaba gerando um impacto negativo de EDP para a aplicação inteira. Portanto, SPRA apresentou os piores resultados para aplicações onde há uma variação considerável no número de *threads* para cada região paralela (e.g., MG, CG, BT, e SP para o sistema Intel 40-*core*).

Por outro lado, a menor sobrecarga de EDP para a estratégia SPRA se da em aplicações onde cada região de uma dada aplicação apresenta o mesmo ambiente em ambos os tamanhos de entrada. Por que, a busca é realizada em cada região paralela quando a aplicação é executada com o tamanho de entrada pequena, o custo de aprendizado é menor que a estratégia SEA, pois SEA precisa executar a aplicação inteira muitas vezes durante a fase de aprendizado. Nos casos mais significantes, para a aplicação NB executada no sistema Intel 40-*core*, a estratégia SPRA apresentou uma sobrecarga 11% menor que a estratégia SEA. Resultados similares também foram observados para as seguintes aplicações: NB, LT, LM, UA, EP e IS. Quando comparado com a solução *Oracle*, a sobrecarga para otimizar o conjunto inteiro de *benchmark* e sistemas *multicore* é de apenas 13%.

Por fim a estratégia *Offline(L)* apresentou os piores resultados em relação a sobrecarga durante a fase de aprendizado independente do sistema *multicore*. Isto se dá pelo fato que o EDP gasto durante a fase de aprendizado, que é executado sobre a entrada grande da aplicação. Porém, na média o conjunto inteiro de aplicações e sistemas *multicore*, o custo de aprendizado apresentado pelas estratégias SEA e SPRA é 88% e 87% menor que a *Offline(L)* respectivamente.

6.4 Sumarização dos Resultados

Baseado nos experimentos mostrados na seção anterior, esta seção sumariza os principais resultados obtidos além de prover diretrizes para os desenvolvedores de *software* que desejam utilizar Hórus como uma ferramenta para otimizar suas aplicações paralelas.

- Ambas as estratégias SEA e SPRA apresentam resultados similares a melhor solução *Oracle* para aplicações onde (*i*) apenas o número de iterações muda da entrada pequena para a entrada grande; e (*ii*) o número de iterações e o tamanho da estru-

tura de dados também muda, para estes casos, a diferença para a solução *Oracle* foram menor que 1.0% na maioria dos casos.

- Os resultados obtidos reforçam a necessidade de se utilizar ambas as estratégias, devido ao fato de cada uma apresentar resultados diferentes para uma determinada aplicação.

- SPRA apresenta melhores resultados do que a estratégia SEA em aplicações com mais de uma região paralela e cada região executa melhor com um número de *threads* diferente. Sendo assim, para este tipo de aplicação (i.e., aplicações com cargas de trabalho desbalanceadas) nós recomendados utilizar a estratégia SPRA.

- Por causa da estratégia SEA apresentar uma menor sobrecarga (média geométrica), considerando a execução do conjunto inteiro de aplicações e processadores, quando o desenvolvedor de *software* não possui informações sobre a aplicação alvo (i.e., número de regiões paralelas e cargas de trabalho), recomenda-se a utilização desta estratégia.

- Independente da estratégia empregada (SEA ou SPRA), para manter o mesmo grau ótimo de TLP através de diferentes entradas, a única diferença entre as entradas pequena e grande deve ser o número vezes que o método irá iterar durante a execução (e.g., número de iterações). Por outro lado, quando o tamanho da estrutura de dados muda, é possível manter o mesmo grau ótimo de TLP entre as diferentes entradas.

Além disso, o intuito de definir faixas de entradas para cada classe de aplicação no qual não há novos ambientes emergentes (ambas SEA e SPRA serão capazes de encontrar resultados muito próximo ao ideal), é possível visualizar na Tabela 7 o tamanho relativo da entrada pequena em relação a entrada grande. A tabela também mostra a diferença de EDP encontrada por cada uma das estratégias SEA e SPRA em relação a melhor solução possível (*Oracle*). Como pode ser observado, para aplicações onde apenas o número de iterações é alterado (de NB até HS), é possível garantir resultados ótimos para EDP na maioria dos casos, utilizando uma entrada que representa apenas 2% do tamanho original das iterações. Para aplicações Híbridas (do MG até FFT), é necessário utilizar uma entrada pequena que representa menos de 30% de iterações e 12% para o tamanho das estrutura de dados, com o intuito de garantir resultados muito próximos ao ideal. Além do mais, para aplicações onde a diferença está apenas no tamanho da estutura de dados, foi necessário utilizar durante a fase de treinamento, tamanhos que representam menos que 6% o tamanho da entrada original, garantindo assim, resultados muito próximos ao ideal, na maioria dos casos.

Tabela 7 – Tamanho relativo da entrada pequena em relação a entrada grande utilizadas por SEA e SPRA e a diferença de EDP entre a solução encontrada por SEA/SPRA e a encontrada pela solução *Oracle*

Bench.	**% de iterações**	**% do tamanho**	**Diferença de EDP para o *Oracle***	
			SEA	**SPRA**
NB	1.00%	–	0.00%	0.00%
JA	0.04%	–	0.00%	0.00%
PO	0.10%	–	0.00%	0.00%
ST	0.40%	–	0.00%	0.01%
LT	1.50%	–	0.00%	0.00%
LM	1.50%	–	0.00%	0.00%
HS	0.01%	–	0.00%	0.01%
MG	20.00%	12.00%	1.99%	17.08%
UA	25.00%	0.07%	0.02%	0.19%
EP	0.03%	0.03%	0.00%	0.01%
FT	30.00%	0.01%	0.01%	3.76%
LU	20.00%	0.004%	0.01%	1.09%
IS	0.004%	0.004%	0.00%	0.08%
CG	20.00%	0.09%	0.03%	6.13%
FFT	71.00%	26.00%	0.03%	7.25%
BT	–	6.00%	0.11%	7.24%
SP	–	6.00%	17.26%	21.63%
NW	–	0.006%	6.94%	16.12%

7 CONCLUSÃO E TRABALHOS FUTUROS

Foi apresentado neste trabalho de pesquisa de mestrado o *Framework* Hórus, que visa encontrar um número de *threads* ideal que possua o menor EDP para uma aplicação paralela utilizando durante o processo de treinamento uma pequena entrada. Para isto, foram utilizados duas estratégias de busca: A primeira irá encontrar um número ideal de *threads* para a aplicação inteira (SEA) e a segunda busca encontrar um número de *threads* para cada região paralela da aplicação (SPRA). Além do mais, caso uma determinada aplicação já tenha sido treinada, esta não passará pelo processo de treinamento novamente, pois os resultados obtidos de cada uma das etratégias é armazenado em uma base de dados. Sendo assim, a aplicação é apenas executada com a sua entrada regular com o número de *threads* que consta na base dados diminuindo assim o tempo de busca.

Como Hórus faz uso de uma pequena entrada para encontrar um número de *threads*, o tempo gasto durante o processo de busca é consideravelmente pequeno, o que torna Hórus uma ótima ferramenta para otimizar o EDP de aplicações paralelas em relação a outras aboradagens já discutidas na Seção 3. Isto se da por que, Hórus não gera nenhuma sobrecarga sobre a aplicação, pois ele faz uso de uma estratégia *offline* de otimização. Além do mais, Hórus é totalmente transparente para o usuário, e não efetua modificações no código, ou realiza recompilação do código fonte.

Desta forma, Hórus possui uma melhoria de EDP significativa na maioria dos casos em ambas as estratégias de busca (SEA e SPRA) para cada uma das aplicações em cada ambiente *multicore* quando comparado com a execução padrão (STD) (figura 9) tendo como redução em média (geométrica) de 30% para SEA e de 26% para SPRA. Além disso, ambas estratégias também apresentaram uma redução de EDP quando comparados os resultados apresentados pela técnica de otimização presente no OpenMP, OpenMP Dynamic (figura 10). Para este cenário, SEA e SPRA apresentam resultados de redução em média de 43% e 40%. Por fim, quando comparamos SEA e SPRA, SPRA possui melhores resultados em aplicações onde o número de *threads* para cada uma das regiões é diferente uma da outra. Porém, SEA possui resultados melhores onde o número de *threads* para cada região é a mesma. Ainda assim, SEA foi capaz de obter resultados melhores que SPRA para os três ambientes *multicore* com reduções de 13% para o AMD 16-*core*, 21% para o AMD 24-*core* e 3.1% para o Intel 40-*core*.

De acordo com o que foi discutido anteriromente, ambas as estratégias de busca implementada por Hórus, garantem uma redução significativa de EDP quando as comparamos com a execução padrão (STD) e o OpenMP_Dynamic. Isto se da pelo fato de que as estratégias possuem uma boa convergência para encontrar uma solução ótima (número de *threads*) que deverá garantir o melhor EDP na maioria dos casos para cada aplicação. O sucesso das estratégias de busca está atrelada ao comportamento ao longo das *threads* da entrada pequena, utilizada durante a fase de busca, ser semelhante a entrada grande, utilizada para executar a aplicação real. Para casos onde este cenário não acontece, Hó-

rus ainda assim, conseguiu resultados satisfatórios em comparação a execução padrão de aplicações paralelas.

Sendo assim, tendo em vista os resultados apresentados por Hórus para otimizar aplicações paralelas com baixo custo de aprendizado, este foi uma *framework* construido com base em um trabalho já publicado (BERNED et al., 2020). Tal trabalho buscou identificar a viabilidade em se otimizar aplicações paralelas utilizando um algoritmo de aprendizado tal como o utilizado pela estratégia SEA, que busca um número de *threads* para a aplicação inteira. Após a publicação deste primeiro trabalho, buscou-se extendê-lo de modo a adicionar um novo método de busca que visa encontrar um número de *threads* ideal para cada uma das regiões paralelas da aplicação, no qual originou esta dissertação de mestrado, que por sua vez, teve os seus resultados publicados em uma revista ((BERNED et al., 2021)). Além destes trabalhos, tendo em vista o bom desempenho dos algoritmos de busca empregados no Hórus, dois outros trabalhos utilizam as mesmas estratégias para otimizar o consumo de energia para aplicações paralelas, são eles:

- (MEDEIROS et al., 2020), no qual trabalho utilizou-se o mesmo algoritmo de busca implementado por SEA para diminuir a depreciação (*Aging*) que um processador pode sofrer caso tenha uma distribuição da carga de trabalho desbalanceada entre os vários núcleos do processador;
- (BERNED et al., 2021) neste trabalho, utilizou-se os algoritmos empregados durante a busca por SEA e SPRA para identificar qual o melhor número, mapeamento e afinidade das *threads* que entregam o menor EDP para aplicações paralelas.

Com base nos resultados obtidos e nas publicações já realizada utilizando as metodologias do *framework* Hórus, este se torna uma ferramenta muito útil para os desenvolvedores de *softwares* que pretendem encontrar uma configuração ideal (número de *threads*) para as suas aplicações., pois, foi possível demonstrar que utilizando uma entrada relativamente muito menor que a entrada grande (original), seja em quantidade de iterações, tamanho da estrutura de dados ou ambas, Hórus converge para resultados muito próximos ao ideal utilizando tamanhos que representam apenas 2% (em relação a entrada grande) para as aplicações onde somente as iterações mudam, 30% do tamanho original e 12% das iterações para aplicações híbridas e apenas 6% para aplicações onde só as estruturas de dados mudam. Desta forma, é possível encontrar uma configuração ideal ou muito próxima a esta para a aplicação paralela utilizando muito menos EDP. Desta forma, tendo em vista os benefícios em se utilizar o Hórus para otimzar aplicações paralelas, este será patenteado como *software* para que outros desenvolvedores possam otimizar suas aplicações paralelas e assim, entregar um produto que irá consumir o menor EDP possível, agregando valor ao produto.

Como trabalho futuro, tendo em vista a grande utilização de GPUs, FPGAs, co-processadores para realizar computações de alto desempenho, há também a necessidade

em reduzir o consumo de EDP destas arquiteturas, desta forma, Hórus será extendido para otimizar o EDP de aplicações que são submetidas a arquiteturas heterogêneas.

REFERÊNCIAS

ALESSI, F. et al. Application-level energy awareness for openmp. In: _____. **OpenMP: Heterogenous Execution and Data Movements: 11th IWOMP**. Cham: Springer, 2015. p. 219–232. ISBN 978-3-319-24595-9. Citado na página 37.

BAILEY, D. H. et al. The nas parallel benchmarks—summary and preliminary results. In: **ACM/IEEE SC**. USA: ACM, 1991. p. 158–165. ISBN 0-89791-459-7. Citado na página 45.

BARNES, B. J. et al. A regression-based approach to scalability prediction. In: **ICS**. NY, USA: ACM, 2008. p. 368–377. ISBN 978-1-60558-158-3. Citado 3 vezes nas páginas 31, 33 e 38.

BENEDICT, S. et al. Energy prediction of openmp applications using random forest modeling approach. In: **IEEE IPDPSW**. [S.l.: s.n.], 2015. p. 1251–1260. Citado 2 vezes nas páginas 34 e 38.

BERNED, G. et al. Combining thread throttling and mapping to optimize the edp of parallel applications. In: **2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)**. [S.l.: s.n.], 2021. Citado na página 66.

BERNED, G. P. et al. Decreasing the learning cost of offline parallel application optimization strategies. In: **2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)**. [S.l.: s.n.], 2020. p. 144–151. Citado 2 vezes nas páginas 20 e 66.

BERNED, G. P. et al. Low learning-cost offline strategies for edp optimization of parallel applications. **Journal of Systems Architecture**, v. 114, p. 101959, 2021. ISSN 1383-7621. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1383762120302101>. Citado na página 66.

BHATT, S. et al. Abstractions for parallel n-body simulations. In: **Scalable High Performance Computing Conf.** [S.l.: s.n.], 1992. p. 38–45. Citado na página 46.

BLAKE, G. et al. Evolution of thread-level parallelism in desktop applications. **SIGARCH Comput. Archit. News**, ACM, NY, USA, v. 38, n. 3, p. 302–313, 2010. ISSN 0163-5964. Citado na página 48.

BORKAR, S. Getting gigascale chips: Challenges and opportunities in continuing moore's law. **Queue**, Association for Computing Machinery, New York, NY, USA, v. 1, n. 7, p. 26–33, out. 2003. ISSN 1542-7730. Disponível em: <https://doi.org/10.1145/957717.957757>. Citado na página 25.

CABRERA, A. et al. Analytical modeling of the energy consumption for the high performance linpack. In: **2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing**. [S.l.: s.n.], 2013. p. 343–350. ISSN 1066-6192. Citado 2 vezes nas páginas 34 e 38.

CHADHA, G.; MAHLKE, S.; NARAYANASAMY, S. When less is more (limo): controlled parallelism forimproved efficiency. In: ACM. **Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems**. [S.l.], 2012. p. 141–150. Citado 2 vezes nas páginas 36 e 38.

CHAPMAN, B.; JOST, G.; PAS, R. v. d. **Using OpenMP: Portable Shared Memory Parallel Programming**. [S.l.]: The MIT Press, 2007. ISBN 0262533022, 9780262533027. Citado na página 50.

CHE, S. et al. Rodinia: A benchmark suite for heterogeneous computing. In: **IEEE Int. Symp. on Workload Characterization**. DC, USA: IEEE Computer Society, 2009. p. 44–54. ISBN 978-1-4244-5156-2. Citado na página 46.

CHOU, C.-Y. et al. An improved model for predicting hpl performance. In: CÉRIN, C.; LI, K.-C. (Ed.). **Advances in Grid and Pervasive Computing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 158–168. ISBN 978-3-540-72360-8. Citado na página 34.

DUTOT, P.-F. et al. Towards energy budget control in hpc. In: **IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing**. USA: [s.n.], 2017. p. 381–390. ISBN 978-1-5090-6610-0. Citado 2 vezes nas páginas 19 e 37.

GIBBS, W. A split at the core. **Scientific American**, v. 291, p. 96–101, 12 2004. Citado na página 25.

GILAT, A. et al. **Métodos Numéricos para Engenheiros e Cientistas. Uma introdução com aplicações usando MATLAB**. [S.l.: s.n.], 2008. Citado na página 46.

HACKENBERG, D. et al. Power measurement techniques on standard compute nodes: A quantitative comparison. In: **IEEE Int. Symp. on Performance Analysis of Systems and Software**. [S.l.: s.n.], 2013. p. 194–204. Citado na página 42.

HäHNEL, M. et al. Measuring energy consumption for short code paths using rapl. **SIGMETRICS Performance Evaluation Rev.**, ACM, NY, USA, v. 40, n. 3, p. 13–17, 2012. ISSN 0163-5999. Citado na página 42.

IPEK, E. et al. An approach to performance prediction for parallel applications. In: **Proceedings of the 11th International Euro-Par Conference on Parallel Processing**. Berlin, Heidelberg: Springer-Verlag, 2005. (Euro-Par'05), p. 196–205. ISBN 3-540-28700-0, 978-3-540-28700-1. Citado 3 vezes nas páginas 31, 33 e 38.

JAYAKUMAR, A.; MURALI, P.; VADHIYAR, S. Matching application signatures for performance predictions using a single execution. In: **IEEE IPDPS**. [S.l.: s.n.], 2015. p. 1161–1170. ISSN 1530-2075. Citado 2 vezes nas páginas 35 e 38.

JOAO, J. A. et al. Bottleneck identification and scheduling in multithreaded applications. In: **ASPLOS**. NY, USA: ACM, 2012. p. 223–234. ISBN 978-1-4503-0759-8. Citado na página 54.

LEE, J. et al. Thread tailor: Dynamically weaving threads together for efficient, adaptive parallel applications. **SIGARCH Comput. Archit. News**, ACM, NY, USA, v. 38, n. 3, p. 270–279, 2010. ISSN 0163-5964. Citado na página 36.

LEVY, H. M. et al. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In: **Int. Symp. on Computer Architecture**. [S.l.: s.n.], 1996. p. 191–191. ISSN 1063-6897. Citado na página 54.

LI, D. et al. Hybrid mpi/openmp power-aware computing. In: **IEEE IPDPS**. [S.l.: s.n.], 2010. p. 1–12. ISSN 1530-2075. Citado na página 36.

LORENZON, A. F. **Avaliação do Desempenho e Consumo Energético de Diferentes Interfaces de Programação Paralela em Sistemas Embarcados e de Propósito Geral**. 2014. Citado na página 30.

LORENZON, A. F.; BECK, A. C. S. **Parallel Computing Hits the Power Wall - Principles, Challenges, and a Survey of Solutions**. [S.l.]: Springer, 2019. (Springer Briefs in Computer Science). ISBN 978-3-030-28718-4. Citado na página 20.

LORENZON, A. F. et al. Aurora: Seamless optimization of openmp applications. **IEEE Transactions on Parallel and Distributed Systems**, v. 30, n. 5, p. 1007–1021, 2019. Citado 4 vezes nas páginas 20, 30, 31 e 54.

MCCALPIN, J. D. **STREAM: Sustainable Memory Bandwidth in High Performance Computers**. Charlottesville, Virginia, 1991–2007. A continually updated technical report. http://www.cs.virginia.edu/stream/. Disponível em: <http://www.cs.virginia.edu/stream/>. Citado na página 46.

MEDEIROS, T. S. et al. Aging-aware parallel execution. **IEEE Embedded Systems Letters**, p. 1–1, 2020. Citado na página 66.

OPENMP. [S.l.], 2020. Disponível em: <https://www.openmp.org/>. Citado na página 29.

OpenMP Architecture Review Board. **OpenMP API Specification: Version 5.1**. 2021. Disponível em: <https://www.openmp.org/spec-html/5.1/openmp.html>. Citado na página 52.

PACHECO, P. **An Introduction to Parallel Programming**. [S.l.: s.n.], 2011. Citado 2 vezes nas páginas 28 e 30.

PAS, R. van de et al. **Using OpenMP - The Next Step**. [S.l.: s.n.], 2017. Citado 3 vezes nas páginas 19, 28 e 30.

PETERSEN, W.; ARBENZ, P. **Introduction to Parallel Computing : A practical guide with examples in C**. [S.l.]: OUP Oxford, 2004. (Oxford Texts in Applied and Engineering Mathematics). ISBN 9780191513619. Citado na página 46.

PORTERFIELD, A. K. et al. Power measurement and concurrency throttling for energy reduction in OpenMP programs. In: **IEEE IPDPS**. [S.l.: s.n.], 2013. p. 884–891. Citado 2 vezes nas páginas 36 e 38.

QUINN, M. **Parallel Programming in C with MPI and OpenMP**. [S.l.]: McGraw-Hill Higher Education, 2004. ISBN 9780072822564. Citado na página 46.

RAASCH, S. E.; REINHARDT, S. K. The impact of resource partitioning on smt processors. In: **PACT**. [S.l.: s.n.], 2003. p. 15–25. ISSN 1089-795X. Citado na página 54.

RUSSEL, P. N. S. **Artificial Intelligence - A modern Aprroach**. [S.l.]: Pearson, 2010. Citado 2 vezes nas páginas 31 e 42.

SENSI, D. D. Predicting performance and power consumption of parallel applications. In: **2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)**. [S.l.: s.n.], 2016. p. 200–207. Citado 2 vezes nas páginas 35 e 38.

SHAFIK, R. A. et al. Adaptive energy minimization of openmp parallel applications on many-core systems. In: **Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures**. NY, USA: ACM, 2015. p. 19–24. ISBN 978-1-4503-3343-6. Citado na página 37.

SHARKAWI, S. et al. Performance projection of hpc applications using spec cfp2006 benchmarks. In: **2009 IEEE International Symposium on Parallel Distributed Processing**. [S.l.: s.n.], 2009. p. 1–12. ISSN 1530-2075. Citado 3 vezes nas páginas 31, 33 e 38.

SODHI, S.; SUBHLOK, J.; XU, Q. Performance prediction with skeletons. **Cluster Computing**, Kluwer Academic Publishers, Hingham, MA, USA, v. 11, n. 2, p. 151–165, jun. 2008. ISSN 1386-7857. Citado 2 vezes nas páginas 35 e 38.

SRIDHARAN, S.; GUPTA, G.; SOHI, G. S. Holistic run-time parallelism management for time and energy efficiency. In: ACM. **Proceedings of the 27th international ACM conference on International conference on supercomputing**. [S.l.], 2013. p. 337–348. Citado 2 vezes nas páginas 37 e 38.

SRIDHARAN, S.; GUPTA, G.; SOHI, G. S. Adaptive, efficient, parallel execution of parallel programs. In: **ACM SIGPLAN PLDI**. NY, USA: ACM, 2014. p. 169–180. Citado 3 vezes nas páginas 20, 37 e 38.

STALLINGS, W. **Arquitetura e Organização de Computadores**. [S.l.]: Pearson Education do Brasil, 2018. Citado 4 vezes nas páginas 19, 25, 26 e 27.

SUBRAMANIAN, L. et al. Mise: Providing performance predictability and improving fairness in shared main memory systems. In: **IEEE HPCA**. [S.l.: s.n.], 2013. p. 639–650. ISSN 1530-0897. Citado na página 54.

SULEMAN, M. A.; QURESHI, M. K.; PATT, Y. N. Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps. **SIGARCH Comput. Archit. News**, ACM, NY, USA, v. 36, n. 1, p. 277–286, 2008. ISSN 0163-5964. Citado 4 vezes nas páginas 36, 38, 54 e 59.

TABORDA, D.; ZDRAVKOVIC, L. Application of a hill-climbing technique to the formulation of a new cyclic nonlinear elastic constitutive model. **Computers and Geotechnics**, v. 43, p. 80 – 91, 2012. ISSN 0266-352X. Citado na página 42.

TIWARI, A. et al. Modeling power and energy usage of hpc kernels. In: **IEEE 26th IPDPS Workshops PhD Forum**. [S.l.: s.n.], 2012. p. 990–998. Citado 2 vezes nas páginas 34 e 38.

TROBEC, R. et al. **Introduction to Parallel Computing**. [S.l.]: Springer, 2018. Citado na página 27.

VIDYA, B.; SRIRAMAN, H.; RUKMANI, P. Optimized multi-threading to balance energy and performance efficiency. **International Journal of Recent Technology and Engineering**, v. 7, n. 6, p. 599–604, 2019. Cited By 0. Citado 2 vezes nas páginas 34 e 38.

WHEELER, K. B.; MURPHY, R. C.; THAIN, D. Qthreads: An API for programming with millions of lightweight threads. In: **IEEE IPDPS**. [S.l.: s.n.], 2008. p. 1–8. ISSN 1530-2075. Citado na página 36.

WITKOWSKI, M. et al. Practical power consumption estimation for real life hpc applications. **Future Gener. Comput. Syst.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 29, n. 1, p. 208–217, jan. 2013. ISSN 0167-739X. Citado 2 vezes nas páginas 34 e 38.

YANG, L. T.; MA, X.; MUELLER, F. Cross-platform performance prediction of parallel applications using partial execution. In: **SC**. USA: IEEE, 2005. p. 40–. ISBN 1-59593-061-2. Citado 2 vezes nas páginas 35 e 38.

ZHANG, W.; CHENG, A. M. K.; SUBHLOK, J. Dwarfcode: A performance prediction tool for parallel applications. **IEEE Transactions on Computers**, v. 65, n. 2, p. 495–507, Feb 2016. ISSN 0018-9340. Citado 2 vezes nas páginas 35 e 38.

ÍNDICE