UNIVERSIDADE FEDERAL DO PAMPA

Gustavo Paim Berned

Um Estudo sobre a Otimização de Aplicações Paralelas através de Aprendizado de Máquina

Alegrete
2019

Gustavo Paim Berned

Um Estudo sobre a Otimização de Aplicações Paralelas através de Aprendizado de Máquina

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Arthur Francisco Lorenzon

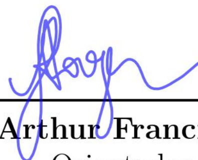
Alegrete
2019

Gustavo Paim Berned

Um Estudo sobre a Otimização de Aplicações Paralelas através de Aprendizado de Máquina

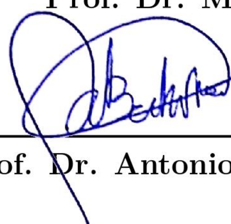
Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Trabalho de Conclusão de Curso defendido e aprovado em 25 de Junho de 2019
Banca examinadora:



Prof. Dr. Arthur Francisco Lorenzon
Orientador
UNIPAMPA

Prof. Dr. Marcelo Caggiani Luizelli
UNIPAMPA



Prof. Dr. Antonio Carlos Schneider Beck Filho
UFRGS

AGRADECIMENTOS

Quero agradecer em primeiro lugar os meus familiares que me deram todo apoio, carinho e força para nunca desistir dos meus objetivos, me orientando pela estrada da vida e guiando meus passos para seguir o melhor caminho. Obrigado Jefferson da Rosa Berned e Maria Inácia Paim Berned, pela formação do meu caráter, e por serem ótimos Pais. Meus irmãos Rafael e Bethânia, obrigado por fazerem parte da minha vida e serem quem são. A minha Esposa Janise Pereira Palma por cuidar de mim nesses dias dificeis de escrita e estudos, a minha sogra Inês pelos ótimos chazinhos da madrugada, aos meus enteados João victor e Eric Muriel por demonstrar seu afeto por mim na chegada tarde da noite da Faculdade, amo todos vocês.

Em segundo lugar os meus colegas de trabalho, principalmente do STIC, onde diversas vezes me auxiliaram nos meus trabalhos acadêmicos, sanando-me dúvidas. Obrigado, Angelo, Cristian, Daniel, Henrique, Júlio, Lidiane, Lucas, Sandro e Rafael por se serem ótimos colegas de profissão e além disto excelentes amigos.

Ao professor Dr. Arthur Francisco Lorenzon, por se demonstrar um execelente orientador, e consequentemente um amigo. Meu muito obrigado, por saber me conduzir nesta árdua jornada acadêmica nestes 6 meses de trabalho.

Ao Professor Antonio Carlos Schneider Beck Filho, pelas suas orientações em um breve encontro na UFRGS e por aceitar ser da minha banca de TCC, professor, muito obrigado.

E por último, o mais importante, agradeço a DEUS, por estar vivo, por te me concedido a honra e a beleza de viver, ter saúde e poder conviver com pessoas maravilhosas ao meu lado, DEUS, MUITO OBRIGADO.

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

RESUMO

Encontrar o número de *threads* que atende uma aplicação de maneira satisfatória não é uma tarefa trivial devido às diferentes variáveis envolvidas, tais como conjunto de entrada, organização hierarquica de memória e microarquitetura do processador. Entretanto, se uma aplicação tem comportamento similar quando diferentes conjuntos de entrada são utilizados, podemos usar um tamanho de entrada pequeno para predizer seu comporamento quando executar com um conjunto de entrada grande. Neste sentido, o objetivo deste trabalho consiste em utilizar os dados coletados pela execução de uma aplicação paralela com um conjunto de entrada pequeno para estimar o seu comportamento quando executada com um conjunto grande. A partir desta estimativa, pretende-se encontrar o grau de exploração de paralelismo no nível de *threads* que fornece o melhor desempenho, consumo de energia e/ou *Energy-Delay Product* (EDP) para a execução com a entrada grande.
Através da execução de dezessete *benchmarks* conhecidos e amplamente utilizados pela comunidade acadêmica em um processador AMD com 16 núcleos, este trabalho mostra que é possível usar o comportamento da execução com conjunto de entrada pequeno para otimizar a execução com um conjunto grande. Considerando a média de todas as aplicações, os resultados mostram uma redução de 9% do consumo de energia e 10% do EDP comparado a execução com o número de *threads* padrão do sistema. O trabalho mostram ainda que o método proposto possuí uma diferença média de apenas 2% no consumo de energia e 1% no EDP com relação a solução *oracle*, que executa a aplicação com o melhor número de *threads*, sem contar com o período de treinamento.

Palavras-chave: Computação Paralela. Aprendizado de Máquina. Otimização de desempenho e Energia.

ABSTRACT

Finding the number of threads that delivers a satisfactory outcome to an application is not a trivial task because of there are many variables involved, such as the application input set, hierarchical memory organization, and the processor microarchitecture. However, if an application has similar behavior when it is executed with different input sets, one can use a small input size to predict its behavior when a larger input set is considered. In this sense, the objective of this work is to use the data collected through the execution of a parallel application with a small input set to estimate its behavior when executing with a larger input set. From this estimation, we intend to find the degree of parallelism exploitation at the thread level that provides the best performance, energy consumption and/or Energy-Delay Product (EDP) to execute the application with the large input set. Through the execution of seventeen benchmarks well-known and widely used by the academic community on an AMD processor with 16-core, this work shows that it is possible to use the behavior of the execution with a small input set to optimize execution with a large input set. Considering the geometric mean of all applications, the results show a reduction of 9% in energy consumption and 10% in the EDP when compared to execution with the number of standard threads of the system. The work also shows that the proposed optimization method has an average difference of only 2% in energy consumption and 1% in EDP with respect to an oracle solution, which executes the application with the best number of threads.

Key-words: Parallel Computing. Machine Learning. Energy and Performance optimization.

LISTA DE FIGURAS

Figura 1 – Aplicação LAVAMD com diferentes tamanhos de entrada 23
Figura 2 – Exemplo de Arquiteturas *Multi-Core* 27
Figura 3 – Exemplo de Memória Distribuída 28
Figura 4 – Exemplo de Memória Compartilhada 29
Figura 5 – Modelo *Fork/Join* 30
Figura 6 – Exemplos de correlação de Spearman 34
Figura 7 – Exemplo de Regressão - KNN, $k = 4$ 37
Figura 8 – Organização das memórias cache do processdor Ryzen 7 1700 47
Figura 9 – Pequeno e KNN em relação ao *Baseline*: menor que 1.0 significa que estes cenários são melhores que o *Baseline*. 53
Figura 10 – Pequeno, KNN e *Baseline* em relação ao *Oracle* 55
Figura 11 – BP - Tempos 65
Figura 12 – LU - Tempos 65
Figura 13 – DC - Tempos 65
Figura 14 – KM - Tempos 65
Figura 15 – EP - Tempos 66
Figura 16 – HS - Tempos 66
Figura 17 – IS - Tempos 66
Figura 18 – MG - Tempos 66
Figura 19 – CG - Tempos 67
Figura 20 – NW - Tempos 67
Figura 21 – SRAD - Tempos 67
Figura 22 – STREAM - Tempos 67
Figura 23 – UA - Tempos 68
Figura 24 – LL - Tempos 68
Figura 25 – LM - Tempos 68
Figura 26 – JA - Tempos 69
Figura 27 – LT - Tempos 69
Figura 28 – BP - Energias 71
Figura 29 – LU - Energias 71
Figura 30 – DC - Energias 71
Figura 31 – KM - Energias 71
Figura 32 – EP - Energias 72
Figura 33 – HS - Energias 72
Figura 34 – IS - Energias 72
Figura 35 – MG - Energias 72
Figura 36 – CG - Energias 73
Figura 37 – NW - Energias 73

Figura 38 – SRAD - Energias 73
Figura 39 – STREAM - Energias 73
Figura 40 – UA - Energias 74
Figura 41 – LL - Energias 74
Figura 42 – LM - Energias 74
Figura 43 – JA - Energias 75
Figura 44 – LT - Energias 75
Figura 45 – BP - EDP 77
Figura 46 – LU - EDP 77
Figura 47 – DC - EDP 77
Figura 48 – KM - EDP 77
Figura 49 – EP - EDP 78
Figura 50 – HS - EDP 78
Figura 51 – IS - EDP 78
Figura 52 – MG - EDP 78
Figura 53 – CG - EDP 79
Figura 54 – NW - EDP 79
Figura 55 – SRAD - EDP 79
Figura 56 – STREAM - EDP 79
Figura 57 – UA - EDP 80
Figura 58 – LL - EDP 80
Figura 59 – LM - EDP 80
Figura 60 – JA - EDP 81
Figura 61 – LT - EDP 81

LISTA DE TABELAS

Tabela 1 – Tabela de Correlação 34
Tabela 2 – Tamanho das entradas das aplicações *Numerical Aerodynamic Simulation* (NAS) utilizadas nos experimentos. 44
Tabela 3 – Tamanho das entradas das aplicações Rodinia utilizados nos experimentos. 45
Tabela 4 – Tamanho das entradas Jacobi, *STREAM* e LULESH utilizados nos experimentos. 45
Tabela 5 – Coeficientes de correlação múltipla do tamanho P para G, MG e EG . 51

LISTA DE SIGLAS

CPU *Central Process Unit*

DSE *Design Space Exploration*

DVFS *Dynamic Voltage and Frequency Scalling*

EDP *Energy-Delay Product*

GCC *GNU Compiler Collection*

GPU *Graphics Processing Unit*

ICC *Intel C++ Compiler*

ILP *Instruction-Level Parallelism*

KNN *K-Nearest Neighbors*

MPI *Message Passing Interface*

NAS *Numerical Aerodynamic Simulation*

SMT *Simultaneous Multi-Threading*

TLP *Thread Level Paralelism*

SUMÁRIO

1	**INTRODUÇÃO**	**21**
1.1	**Objetivos**	**23**
1.2	**Estrutura do Texto**	**24**
2	**FUNDAMENTAÇÃO TEÓRICA**	**25**
2.1	**Programação Paralela em Sistemas *Multicore***	**25**
2.1.1	**Arquiteturas Multicore**	**26**
2.1.2	**Interfaces de Programação Paralela**	**27**
2.1.2.1	***Message Passing Interface*(MPI)**	**28**
2.1.2.2	**OpenMP**	**29**
2.1.2.3	**POSIX Threads**	**31**
2.2	***Design Space Exploration (DSE)***	**32**
2.3	**Correlação de Spearman**	**33**
2.4	**Aprendizado de Máquina**	**35**
2.4.1	**Regressão e o Método *K-Nearest Neighbors* (KNN)**	**35**
3	**TRABALHOS RELACIONADOS**	**39**
3.1	**Contexto deste Trabalho**	**42**
4	**METODOLOGIA**	**43**
4.1	***Benchmarks***	**43**
4.1.1	**NAS Parallel Benchmark**	**43**
4.1.2	**Rodinia**	**44**
4.1.3	**Outros Benchmarks**	**45**
4.1.3.1	**Método de Jacobi (JA)**	**46**
4.1.3.2	***STREAM***	**46**
4.1.3.3	**LULESH (LL)**	**46**
4.2	**Ambiente de Execução**	**46**
4.3	**Métricas Avaliadas**	**47**
4.4	**Correlações**	**48**
4.5	**KNN**	**49**
4.6	**Cenário de Validação**	**49**
5	**RESULTADOS**	**51**
5.1	**Análise da Correlação entre Entradas de Diferentes Tamanhos**	**51**
5.2	**Desempenho, Energia e EDP em relação ao *Baseline***	**52**
5.3	**Desempenho, Energia e EDP em relação ao *Oracle***	**53**
6	**CONCLUSÃO E TRABALHOS FUTUROS**	**57**

REFERÊNCIAS 59

APÊNDICES 63

APÊNDICE A – COMPORTAMENTO DAS APLICAÇÕES - TEMPOS 65

APÊNDICE B – COMPORTAMENTO DAS APLICAÇÕES - ENERGIAS 71

APÊNDICE C – COMPORTAMENTO DAS APLICAÇÕES - EDP 77

Índice 83

1 INTRODUÇÃO

Nas últimas décadas, computadores compostos de múltiplas unidades de processamento (e.g., núcleos) têm se tornado onipresentes independente do nicho de atuação, sejam eles em sistemas de propósito geral (*desktop* e *laptop*), sistemas embarcados (*smartphones* e *tablets*) ou em servidores de alto desempenho (PACHECO, 2011). Tais sistemas multiprocessados (ou simplesmente processadores *multicore*) surgiram em meados de 2004 devido a uma limitação térmica dos processadores com um único núcleo de processamento (*single core*), onde já não era mais possível aumentar a sua frequência de operação (PACHECO, 2011). 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. No entanto, com a introdução de mais núcleos em um só chip, surgiram novos desafios relacionados à utilização eficiente dos recursos computacionais disponíveis.

Diferentes abordagens têm sido adotadas para melhorar o uso dos recursos computacionais, tais como, exploração do paralelismo no nível de instrução (*Instruction-Level Parallelism* (ILP)) e paralelismo no nível de *threads* (*Thread Level Paralelism* (TLP)). Um exemplo típico de exploração de paralelismo em ILP são as arquiteturas do tipo superescalares, no qual vários *pipelines* de instruções independentes são utilizados, onde cada instrução (e.g., *load*, *store*) de um programa pode ser executado de forma simultânea (STALLINGS, 2010). Já o paralelismo em nível de *threads*, além de lidar com abordagens de maior granularidade (ALVES et al., 2007), explora o paralelismo não apenas no nível do hardware mas também no fluxo de instruções. Isso significa que é possível mais de uma *thread* ser executada por vez, diferentemente da superescalaridade, onde o paralelismo ocorre apenas nas instruções de uma única *thread* (MORAIS, 2014).

Apesar destas abordagens usufruirem de maneira satisfatória os recursos computacionais, ILP possui fluxos de instrução típicos com uma quantidade limitada de paralelismo devido à dependência de dados entre as instruções (WALL, 1991). Isto resulta em esforços consideráveis para projetar uma microarquitetura que trará apenas ganhos marginais de desempenho com sobrecarga de área e energia significativa. Mesmo se considerarmos um processador perfeito, a exploração do ILP alcançará um limite superior (OLUKOTUN et al., 200). Assim, torna-se mais comum a exploração do paralelismo em nível de TLP, onde nele a aplicação é dividida em várias tarefas (ou ainda, conjunto de instruções), que podem ser executadas concorrentemente, possibilitando assim, reduzir o seu tempo total de execução (VOLPATO, 2014).

Conforme (FOSTER, 1995), o principal objetivo da exploração do paralelismo (seja por ILP ou TLP) é reduzir o tempo envolvido na resolução de um problema. Neste sentido, diferentes áreas, como por exemplo, engenharias, ciências e médicas, têm explorado paralelismo no nível de threads para otimizar suas aplicações. Uma abordagem bastante tradicional é executar a aplicação com o maior número de *threads* possível, o que

naturalmente, é igual ao número de núcleos presentes na microarquitetura (LEE, 2010). No entanto, conforme discutido em (LORENZON, 2018), a execução com o número máximo de *threads* não necessariamente resulta no melhor desempenho (e.g., menor tempo de execução) devido a diferentes fatores de *hardware* e *software*. Neste sentido, encontrar o número ideal de *threads* para executar uma determinada aplicação naturalmente incorre no melhor desempenho possível para ela.

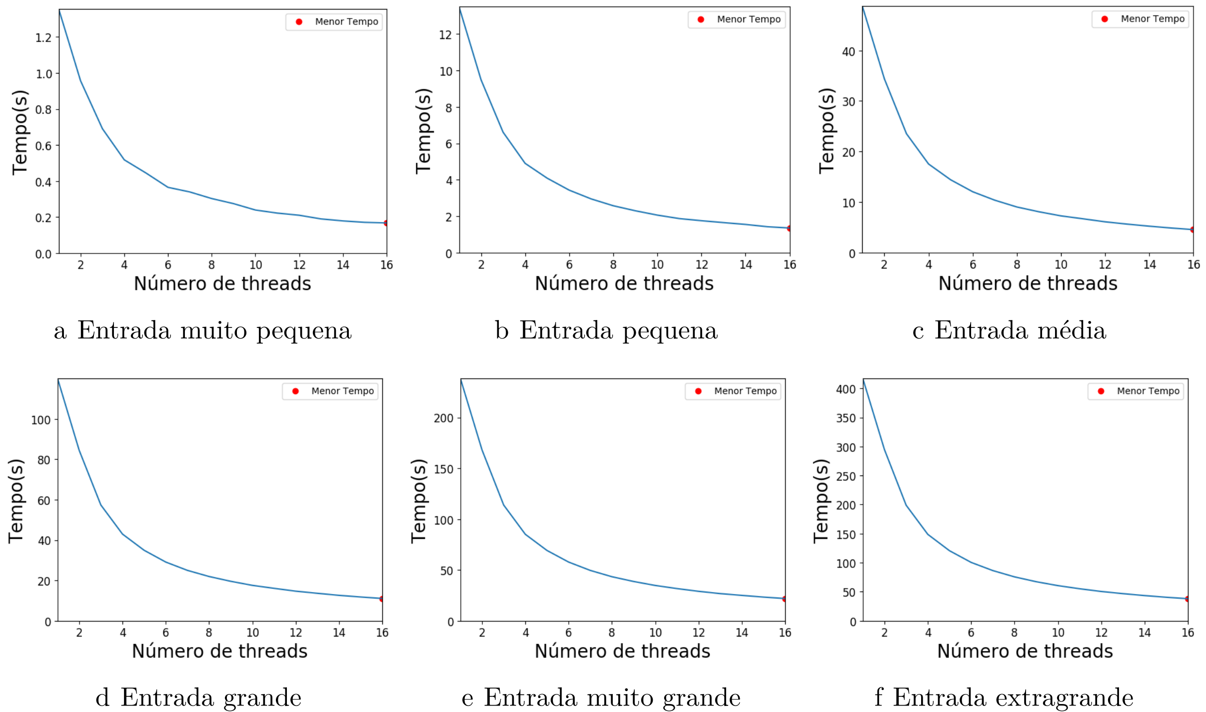
No entanto, tal tarefa é extremamente complexa e compreende um amplo espaço de exploração de projeto com diferentes variáveis que mudam de acordo com as características da aplicação (e.g., conjunto de entrada e número de regiões paralelas), do sistema operacional (e.g., versão do *kernel* e do compilador) e do *hardware* (e.g., número de núcleos e organização do sistema de memória). Portanto, abordagens que buscam pelo número ideal de *threads* enquanto a aplicação está executando são mais susceptiveis a lidar com este amplo espaço de exploração. Por outro lado, quando as aplicações são executadas com um conjunto de entrada suficientemente grande (e.g., que levam horas para executar), o tempo de treinamento (executar a aplicação de forma exaustiva ao longo de todas as *threads* disponível no ambiente de execução) para encontrar o melhor número de *threads* durante a execução da aplicação, aumentará substancialmente.

Considerando este cenário, uma possível abordagem para reduzir o tempo de busca pelo número ideal de *threads* é realizar o treinamento sob a execução da aplicação com um conjunto de entrada pequeno e predizer o seu comportamento para a execução com um conjundo de entrada suficientemente grande. No entanto, para que os resultados do treinamento sejam satisfatórios, a aplicação deve possuir comportamento similar com respeito a aceleração da aplicação conforme aumenta o número de *threads* na execução com diferentes conjuntos de entrada.

Um exemplo de aplicação que possui este comportamento é a LAVAMD da suite de *benchmarks* RODIDNIA (CHE et al., 2009), conforme ilustrado na Figura 1 para a execução em um processador AMD com 16 núcleos. Nesta Figura, pode-se observar que para os seis diferentes conjuntos de entrada (muito pequena para extragrande), a aplicação apresenta o mesmo comportamento de tempo de execução (eixo y) conforme o número de *threads* aumenta (eixo x). Neste sentido, é possível executar o algoritmo de treinamento para a execução com o menor conjunto de entrada, o que naturalmente reduz o tempo de treinamento, e usar a configuração obtida para executar a mesma aplicação com um conjunto de entradas suficientemente grande. Por exemplo, vamos considerar uma aplicação paralela que computa sobre um conjunto de dados muito pequeno e possui tempo de treinamento (busca pelo melhor número de *threads*) em torno de 10 segundos. Por outro lado, quando esta mesma aplicação for submetida a um volume de dados maior, seu tempo de treinamento aumentou consideravelmente para 1000 segundos. Neste sentido, ao aplicar o treinamento sobre a entrada pequena e prevendo corretamente o número de *threads* para a execução com a entrada grande resulta em uma redução de aproximadamente 99% no

tempo de treinamento.

Figura 1 – Aplicação LAVAMD com diferentes tamanhos de entrada



a Entrada muito pequena

b Entrada pequena

c Entrada média

d Entrada grande

e Entrada muito grande

f Entrada extragrande

Para esta tarefa, métodos de regressão não linear (e.g., exponencial, polinomial) ou ainda, métodos não paramétricos, podem ser utilizados para realizar uma predição sobre estes dados, tendo em vista que os comportamentos demonstrados não são lineares.

Portanto, prever o comportamento de aplicações paralelas com base em um conjunto de entrada pequeno pode levar a expressivos ganhos de desempenho e redução no consumo de energia da aplicação. Uma maneira de realizar esta predição é através do uso de métodos de aprendizado de máquina, os quais são abordados neste trabalho.

1.1 Objetivos

Com base no que foi descrito na Seção anterior, aplicações paralelas podem assumir comportamentos similares de tempo de execução, quando são submetidas a diferentes tamanhos de entrada. No entanto, esta também podem assumir comportamentos similares para consumo de energia e de EDP. Sendo assim, se estas possuirem tais semelhanças, o menor tamanho de entrada pode ser utilizado como treinamento por um método de aprendizado de máquina que poderá prever o comportamento da aplicação quando esta é submetida a tamanhos de entrada maiores. Desta forma, este trabalho tem como objetivo otimizar o desempenho, consumo de energia e EDP de aplicações paralelas que executam com conjunto de entrada suficientemente grande através da predição utilizando o menor conjunto de entrada possível. Para tal, os objetivos específicos precisam ser contemplados:

- Executar diferentes *benchmarks* de diferentes suites de aplicações para verificar o comportamento das métricas: tempo de execução; consumo de energia; e EDP, medindo seus coeficientes de correlação.
- Utilizar um método de aprendizado de máquina para prever o comportamento de uma aplicação paralela utilizando para treinamento os dados coletados após a execução da aplicação com o menor tamanho de entrada.
- Utilizar o melhor resultado obtido na execução com o conjunto de entrada pequeno para executar a mesma aplicação com o conjunto de entrada grande.
- Comparar os resultados obtidos com uma solução *Oracle*, que consiste do melhor resultado para a execução da aplicação. Esta comparação tem o objetivo de avaliar o quão longe as técnicas propostas estão da melhor possível.

1.2 Estrutura do Texto

Este trabalho esta organizado da seguinte maneira:

No Capítulo 2, serão discutidas as principais características dos processadores *multicore*. Em seguida, serão abordadas os principios inerentes as interfaces de programação paralela. Após, serão apresentados os fundamentos relacionados a *Design Space Exploration* (DSE). Será abordado o método de correlação de Spearman na Seção 2.3. E por fim, uma introdução sobre o tema de aprendizado de máquina.

O Capítulo 3 irá conter os trabalhos que possuem relação com esta monografia e que auxiliaram no desenvolvimento desta.

No Capítulo 4 será descrito como foi realizado o desenvolvimento deste trabalho, onde inicialmente serão abordados os *benchmarks* utilizados para experimentos na Seção 4.1. Logo após, na Seção 4.2, O ambiente de execução no qual foram submetidos a testes os *benchmarks* será apresentado. Em seguida, serão descritas as métricas de interesse extraídas de cada *benchmark* na Seção 4.3. Por fim, será descrito o método de regressão utilizado neste trabalho na Seção 4.5.

O Capítulo 5 apresentará os resultados inerentes a correlação múltipla entre os dados coletados sobre a execução do menor tamanho de entrada em relação aos maiores tamanhos de entrada, para cada uma das métricas tempo de execução, energia e EDP na Seção 5.1. A Seção 5.2 contém os resultados comparativos entre a os métodos de treinamento comparando-os com uma solução *Baseline*. Por fim, Seção 5.3 apresenta os resultados comparativos entre os métodos de treinamento e a solução *Oracle*. Por fim, a Conclusão deste trabalho e possibilidades de trabalhos futuros estarão presentes no Capítulo 6.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta de maneira introdutória os conceitos necessários para o entendimento deste trabalho. Inicialmente será abordado na Seção 2.1 o tema referente programação paralela em sistemas *multicore*. Após, na Seção 2.2 será discutido os conceitos pertinentes a DSE. Em seguida, será apresentado o métodos de correlação de Spearman e correlação multipla na Seção 2.3. E por fim, uma introdução abordando aprendizado de máquina será apresentado na Seção 2.4.

2.1 Programação Paralela em Sistemas *Multicore*

De 1986 até 2002 o desempenho dos microprocessadores aumentou, em média 50% por ano. Este aumento, que segue a Lei de Moore, fazia os usuários e os desenvolvedores de software frequentemente esperar a próxima geração de processadores para obter um aumento de desempenho em suas aplicações (PACHECO, 2011). Porém desde 2002, este aumento não teve a mesma expressão, ficando em torno de 20%. Esta diferença significante de desempenho induziu a indústria á uma mudança drástica no paradigma de *design* e manufatura de seus processadores (PACHECO, 2011). Neste sentido, ao invés dos fabricantes de processadores melhorar o desempenho de apenas um processador através do aumento de sua frequência de operação, eles começaram o desenvolvimento de processadores com mais de um núcleo.

No entanto, aumentar o número de núcleos em um único chip, causou um grande impacto para os desenvolvedores de *software*, pois a adição de mais núcleos em um circuito, não aumentaria o desempenho das suas aplicações. Desta forma, viu-se a necessidade de desenvolverem programas que pudessem usufruir de mais de um núcleo, ou seja, desenvolver programas paralelos.

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, de genoma e entre outras aplicações.

Tradicionalmente, os programas são desenvolvidos para serem computados sequencialmente, onde uma instrução é executada após a outra. Entretanto, a execução paralela de instruções sempre pode ser explorada. Arquiteturas superescalares podem executar simultaneamente instruções independentes dentro de um programa, desde que haja unidades funcionais suficientes para tal. Arquiterutas superescalares, então conseguem extrair o paralelismo em uma granularidade mais fina no nível de instruções (LORENZON, 2014), tal abordagem é conhecida como *ILP*.

Outra forma de se explorar o paralelismo é através da abordagem TLP. TLP provê paralelismo através da execução simultânea de diferentes *threads*, provendo uma granularidade mais grossa de paralelismo do que ILP(PACHECO, 2011), desta forma,

TLP consegue usufruir tanto das diversas unidades de processamento (núcleos) quanto de instruções.

2.1.1 Arquiteturas Multicore

Buscando aumentar o desempenho e tirar vantagem do número crescente de transistores disponíveis devido à melhora no processo de manufatura de circuitos integrados, a indústria passou a investir na tecnologia multicore como uma alternativa às limitações na exploração em paralelismo em nível de processos, que processadores do tipo superescalar demostravam; além de também se apresentar como uma solução para a alta potência consumida por estes (LORENZON, 2014). Processadores *multicore*, têm como principal característica, possuírem dois ou mais núcleos de processamento em um único circuito integrado, e, compartilham um mesmo espaço de endereçamento. Este espaço de endereçamento serve para que os núcleos possam trocar dados entre si através de instruções do tipo *load* (carga) e *store* (armazenamento).

Uma desvantagem das arquiteturas de processadores com múltiplos núcleos, é no acesso aos dados. Processadores anteriores que continham somente um núcleo, já possuiam uma pequena memória alocada no próprio chip, com a tarefa de reduzir a latência média de acesso à memória. Com as arquiteturas *multicore*, esta situação se torna mais grave, pois é necessário realizar a comunicação entre os núcleos do processador. Toda vez que um núcleo requistar dados, e este precisar ir até a memória principal para obtê-los, haverá um grande atraso nesta comunicação, uma vez que a memória principal possui uma alta latência de acesso as informações.

Um modo para atacar este problema é providenciar caches. Uma cache guarda as palavras (dados) de memória usadas mais recentemente em uma pequena memória rápida, o que acelera o acesso a elas. Se uma porcentagem suficientemente grande das palavras de memória estiver na cache, a latência efetiva de memória pode ter uma enorme redução (TANENBAUM, 2010).

Uma representação da arquitetura de um processador *multicore*, é mostrado na Figura 2. Onde C0, C1, C2 e C3 representam os núcleos do processador. Pode-se notar que existem duas memórias cache que são de acesso exlusiva de cada núcleo, L1 e L2, e uma outra memória cache que é compartilhada por todos, denominada L3 e a memória principal.

Como podemos ver na Figura 2, as memórias (L1, L2, L3) são internas ao *chip* do processador e por serem pequenas e próximas ao núcleo, possuem uma capacidade limitada de armazenamento, mas, o acesso aos dados possui uma latência pequena. Já a memória principal, contém mais capacidade de armazenamento, mas uma latência de acesso ao dados maior do que as memórias internais ao chip do processador.

Um dos principais motivos de atraso na comunicação entre os diferentes núcleos de um processador acontece quando é preciso acessar a memória compartilhada, uma vez que

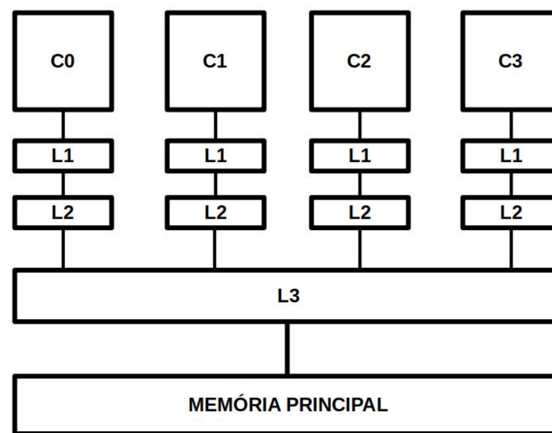


Figura 2 – Exemplo de Arquiteturas *Multi-Core*

um núcleo precisa requisitar dados de outro que está processando um determinado dado, o núcleo requisitante fica em estado de espera até que o núcleo que detém a informação termine sua tarefa, representando assim um aumento no tempo e no consumo de energia da aplicação. Isto acontece para que haja consistência nos dados que estão sendo processados, fazendo com que as *threads* não acessem locais de memória que estão sendo utilizadas por outras. Este tipo de comportamento é denominado exclusão mútua, no qual possui dois principais estados, bloqueado, indicando que uma determinada área de memória já está em uso e desbloqueado, onde a *thread* requisitante dos dados pode acessar determinda região de memória.

Hoje os processadores *multi-core* estão presentes em *desktops* e *laptops*, no qual são utilizados no dia-a-dia. Os primeiros processadores que possuiam a arquitetura *multicore*, surgiram em meados de 2004. Os processadores de uso doméstico hoje, possuem em sua grande maiora de 4 a 6 núcleos, enquanto que os processadores utilizados em sistemas de alta performance possuem entre 12 e 16 núcleos.

2.1.2 Interfaces de Programação Paralela

Muitos são os paradigmas computacionais utilizados em computação paralela, tais como paralelismo de dados, memória compartilhada, troca de mensagens, operações em memória remota, processos, e também, combinação dos anteriores. Tais modelos se diferenciam em vários aspectos, como por exemplo, se a memória disponível é localmente compartilhada ou geograficamente distribuída e, volume de comunicação tanto em *hardware* como em *software* (GROOP et al., 1998). Desta forma, para que um programa utilize de maneira eficiente uma determinada arquitetura, é necessário que as suas atividades paralelas sejam mapeadas sobre os recursos de processamento disponíveis (KIST, 2013). Para isto, existem as interfaces de programação paralela que auxiliam o programador na abstração do mapeamento de tais arquiteturas, tais interfaces são *Message*

Passing Interface (MPI) (GROOP et al., 1998),*Open Multi-Processing Application Program Interface*(OpenPM), (CHAPMAN et al., 2008),Pthreads (BUTENHOF, 1997) entre outras.

2.1.2.1 *Message Passing Interface*(MPI)

MPI é uma interface de programação paralela para as linguagens de programação C/C++ e Fortran que facilitou a comunicação em arquiteturas de memória distribuída, podendo esta também ser utilizada em arquiteturas de memória compartilhada. Um típico programa MPI é definido com uma coleção de processos onde estes se comunicam através de operações de comunicações.

Um sistema de memória distribuída consiste em múltiplos nodos de processamento independentes com módulos de memória privativos, conectados por uma rede de comunicação, tal como ilustra a Figura 3. A escalabilidade natural destes sistemas permitem o desenvolvimento de aplicações com um poder de computação muito alto. O projeto de softwares para este sistemas são mais complexos, porém o projeto de hardware é mais fácil que num sistema de memória compartilhada (CARNEIRO,).

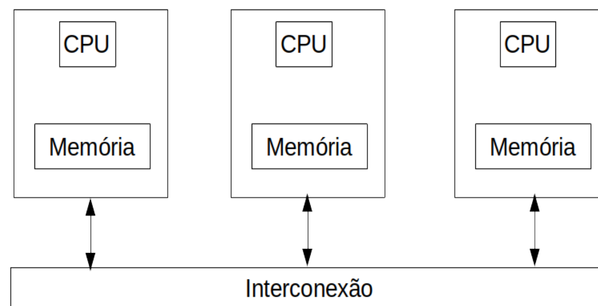


Figura 3 – Exemplo de Memória Distribuída

Nos sistemas multicore, as operações de comunicação através de troca de mensagens são abstraídas para filas de mensagens, que são objetos similares a *pipes* e filas do tipo FIFO (PELETTI et al., 2007).

Ao iniciar o programa, cada processo executa a função de inicialização do ambiente de execução MPI, o MPI_Init(). Já a finalização de um processo MPI ocorre através da chamada à função MPI_Finalize() (LORENZON, 2014).

Quando um ambiente MPI é inicializado, cada um dos processos criados, recebem seu identificador (*rank*) único dentro do comunicador através da função MPI_Comm_rank(). O comunicador é responsável por identificar um grupo de processos e representar os canais por onde estes deverão se comunicar através do MPI. Por padrão um comunicador chamado MPI_COMM_WORLD já existe. Para obter o total de processos de um determinado comunicador, este é obtido através da função MPI_Comm_size().

A comunicação entre os processos acontece através de envio/recebimento. Onde para enviar uma informação para outro processo é utilizado o método MPI_Send() e,

para receber informações de outros, utiliza-se o MPI_Recv().

As operações coletivas em MPI são divididas em operações de sincronização, movimento de dados e computação coletiva. As operações de sincronização são aquelas onde os processos aguardam até que todos os demais processos do grupo atinjam o ponto de sincronização, para então continuar o fluxo de execução. Por outro lado, as operações de movimento de dados são aquelas que permitem a troca de informações entre vários processos. As principais são *broadcast* (MPI_Bcast()), *scatter/gatter* (MPI_Scatter()/MPI_Gatter()) e todos para todos (MPI_Alltoall()). Por fim, as operações de computação coletiva, também chamadas de operações de reduções, são aquelas que realizam operações sobre os dados provenientes de diversos processos. As operações mais comuns são: soma, multiplicação, máximo e mínimo valor, entre outras (LORENZON, 2014).

2.1.2.2 OpenMP

OpenMP é uma interface de programação paralela baseado em memória compartilhada para as linguagens de programação C/C++ e Fortran. Ele consiste em um conjunto de diretivas para o compilador, variáveis de ambientes, e funções de biblioteca (LORENZON, 2014). OpenMP foi desenvolvido por um grupo de programadores e cientistas que acreditaram que escrevendo interfaces de programação paralela de larga escala e de alta performance era muito difícil, então eles definiram as especificações do OpenMP, para que programas baseados na abordagem de memória compartilhada pudessem ser desenvolvidos em alto nível (PACHECO, 2011).

Quando se programa utilizando OpenMP, é possível visualizar o sistema como uma coleção de núcleos, onde todos eles acessam a memória principal. Portando, qualquer mudança realizada por um núcleo (*Central Process Unit (CPU)*) na memória principal, esta será visível por todos os outros. A Figura 4 nos mostra quatro CPUs, onde através de uma interconexão, eles podem acessar a memória principal.

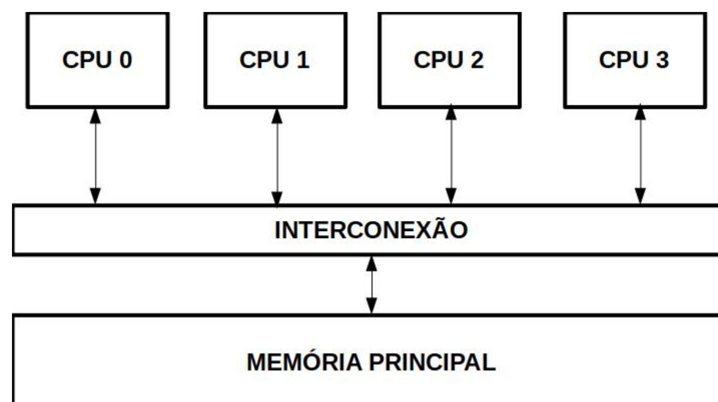


Figura 4 – Exemplo de Memória Compartilhada

O paralelismo em aplicações baseadas em OpenMP é explorada através de diretivas

de compilação, que permite ao programador simplificar qual região de código deverá ser executada em paralelo, ficando a cargo do compilador decidir de que maneira esta região será paralelizada. As principais diretivas existentes são, construtor paralelo, construtor de compartilhamento de trabalho e diretivas de sincronização, no qual serão explicados a seguir.

O **Construtor Paralelo** como é demonstrado na Figura 5, especifíca que um bloco estruturado de código deverá ser executados por múltiplas *threads*. Este construtor possui um modelo do tipo *Fork/Join*. Neste tipo de modelo inicialmente, existe um fluxo de execução principal no qual é chamado de *thread master*. Quando a *thread master* se deparar com um construtor paralelo, este se dividirá em 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.

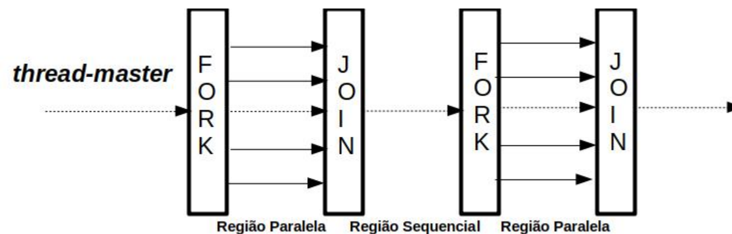


Figura 5 – Modelo *Fork/Join*

Os **construtores de compartilhamento de trabalho** são os responsáveis de informar ao compilador como a carga de trabalho deverá ser distribuídas entre as *threads*. Laços paralelos são a principal forma de distribuição da carga de trabalho. Onde uma estrutura de repetição é distribuida entre as *threads*. Contudo, a maioria dos sistemas utilizam de forma grosseira, blocos de particionamento da carga de trabalho. Por exemplo, se existe m iterações em um laço paralelo, então cada *thread* será responsável por executar m/T operações, onde T é o número total de *threads* (PACHECO, 2011). OpenMP possui diferentes escalonadores que auxiliam na distribuição das tarefas 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 um tipo 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.

Outro dois construtores que merecem ser citados são os construtores de seções paralelas e o construtor único. O **constutor 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 sinconização**, estas são utilizadas para que se evite as condições de corrida, assim dados não podem ser acessados por mais de uma *thread*. Existem três principais diretivas que restrigem acesso as informações, são **critical**, somente uma *thread* pode acessar uma tarefa por vez; **atomic**, esta diretiva é responsável por incubir uma *thread* a atualizar uma determinada região da memória atomicamente; e ***barrier***, que deve ser utilizada quando há a necessidade de realizar um sincronismo entre 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 ser finalizadas. Sempre que houver ponto de sincronização (explícito e implícito) entre as *threads*, no OpenMP elas entram em estado de *busy-waiting*, ou seja, elas executam instruções de acesso a memória para checar a variável de controle repetidamente até o final da sincronização (LORENZON, 2014).

2.1.2.3 POSIX Threads

POSIX Threads ou simplesmente Pthreads, é uma API para programação em memória compartilhada para linguagens C/C++. Diferentemente do OpenMP, onde o paralelismo é expresso em alto nível de abstração com a inserção de diretivas no código sequencial, o paralelismo em Pthreads é explícito através de funções da biblioteca. Ou seja, o programador é responsável por realizar o gerenciamento das threads (criação/finalização), a distribuição da carga de trabalho e o controle de execução (BUTENHOF, 1997).

Similarmente ao OpenMP, inicialmente um programa executa um fluxo sequencial e quando é necessário realizar uma computação em paralelo novas *threads* são criadas através da função *pthread_create()*. Para finalizar uma *thread* basta invocar a função *pthread_exit()*. Já a sincronização entre estas se da pela função *pthread_join()* bloquendo-as até a finalização das demais.

Operações de ***mutex*** são utilizadas para implementar seções criticas e operações atômicas (GRAMA et al., 2003). Estas são definidas invocando a função pthread_mutex(). Para definir o início de uma seção crítica utiliza-se a o método pthread_mutex_lock() e, para realizar a sincronização entre as *threads* utiliza-se o método pthread_join(), que faz com que estas fiquem bloqueadas até a finalização das demais.

Ao se utilizar as funções do tipo *mutex*, se esta habilitando acesso exclusivo à uma *thread* para uma determinada região de memória que é compartilhada. Desta forma, são necessárias variáveis de condição para permitir que uma determinada *thread* aguarde

até que uma determinada condição seja satisfeita para que esta possa ter acesso a região exclusiva, onde, tais variáveis são implementadas através da função *pthread_cond()*.

No entanto, assim que uma thread termina a execução de uma região crítica, ela segue seu fluxo de execução. Dependendo da aplicação, a etapa x+1 só pode começar após a finalização da etapa x. Portanto, fazem-se necessárias uso barreiras entre estas etapas, onde as *threads* só continuarão a ser executadas quando todas as demais estiverem no mesmo ponto de execução. Este comportamento é fornecido através de funções de sincronização do tipo pthread_barrier() (LORENZON, 2014).

Diferente do OpenMP, em Pthreads a sincronização ocorre pelo bloqueio das threads através de mutex (TANENBAUM, 2010). Desta forma, um *mutex* pode ter dois estados: livre ou ocupado. Quando uma *thread* precisa acessar uma determinada região crítica, esta chama uma *mutex_lock*. Se este estiver livre, a seção crítica esta livre à outras *threads* e estas podem entrar nesta região. Porém, se esta estiver ocupada, as demais *threads* não podem fazer uso desta região até que a *thread* que está fazendo uso da região a libere.

2.2 *Design Space Exploration (DSE)*

Este é um método que visa ajustar parâmetros que são auto-configuráveis. Muito utilizado em ambientes onde existe um problema de otimização multi-objetivo. Basicamente, o problema do *DSE* é explorar um grande espaço de parâmetros que devem ser ajustados para encontrar o melhor *trade-off* entre as métricas, como energia, desempenho, EDP, etc. Inicialmente os dados de entrada que deverão ser avalidados, são fornecidos ao DSE. Tais valores são calculados com base em um modelo de classificação, tal como redes neurais, regressão linear, modelos matemáticos, entre outros modelos. Por fim os resultados do DSE, contém o melhor *tradeoff* entre os valores e as métricas (LORENZON, 2018). O DSE pode ser executado em diferentes momentos durante a execução de uma aplicação, ela pode ser performada de modo *online* ou *offline*, com ou sem decisão de adptação em tempo de execução.

Informação *offline* sem decisão de adaptação em tempo de execução: Nesta abordagem, DSE é realizado por completo antes da execução da aplicação. Similiar a modelos de predição, no qual utilizam uma variedade de modelos estatísicos para analizar os valores atuais e passados. Os dados obtidos pelas predições são utilizados apenas para decidir qual a melhor configuração para a execução de uma aplicação. Assim sendo, não existe uma tomada de decisão e adaptação de uma aplicação em tempo de execução. Um modelo preditivo é composto por alguns passos. Primeiramente, são coletados os dados de uma aplicação e com base neles, é gerado um modelo. Um modelo estatístico 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 novos dados para que o modelo seja validado.

Informação *offline* com decisão de adaptação em tempo de execução: Esta abordagem é um pouco diferente da vista anteriromente, principalmente no que diz respeito a obtenção dos dados em tempo de execução. Neste modelo, os dados são capturados durante a execução da aplicação, e pasados para que sejam feitas novas computações sobre estes novos dados. Uma das limitações desta abordagem é a necessidade de executar o modelo a cada obtenção de novos dados.

Informação *online* sem tempo de adaptação em tempo de execução: As abordagens desta classe são aquelas que consideram o ambiente atual de uma arquitetura de microprocessadores quando a aplicação está sendo compilada. No entanto, o modelo verifica as características da microarquitetura e a aplicação para ajudar o compilador a gerar um código otimizado para o ambiente. Contudo, não há adaptação da aplicação em tempo de execução.

Informação *online* com tempo de adaptação em tempo de execução: Diferentemente da anterior, nesta classe, o modelo considera as informações obtidas em tempo de execução para realizar decisões e ajustar a execução da aplicação. Neste caso, diferentes características só são conhecidas em tempo de execução, como o comprimento 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.

2.3 Correlação de Spearman

A correlação fornece um número, indicando como duas variáveis variam conjuntamente. Mede a intensidade e a direção da relação linear ou não-linear entre duas variáveis. É um indicador que atende à necessidade de se estabelecer a existência ou não de uma relação entre essas variáveis sem que, para isso, seja preciso o ajuste de uma função matemática.

Dois métodos são comumente utilizadas para medir a intensidade de entre duas variáveis, o método de correlação de Pearson e de Spearman. A correlação obtida através do coeficiente de Pearson, que é a medida de correlação mais conhecida, é linear. Assim, nos casos em que a relação entre as variáveis seja não linear (quadrática, cúbica,exponencial, etc.), ela não será medida adequadamente. Nesses casos os dados devem ser transformados para a obtenção da medida adequada. O outro coeficiente de correlação utilizado, o de Spearman, por realizar uma transformação de postos, pode ser utilizado nas situações em que a relação entre os pares de dados não é linear (PONTES,).

O método de Spearman visa associar duas variáveis de forma não paramétrica, não necessitando que a relação entre as variáveis seja linear, nem que elas sejam quantitativas.

Valor de ρ (+ ou -)	**Interpretação**
0.00 a 0.19	Correlação muito fraca
0.20 a 0.39	Correlação fraca
0.40 a 0.69	Correlação Moderada
0.70 a 0.89	Correlação Forte
0.90 a 1.00	Correlação Perfeita

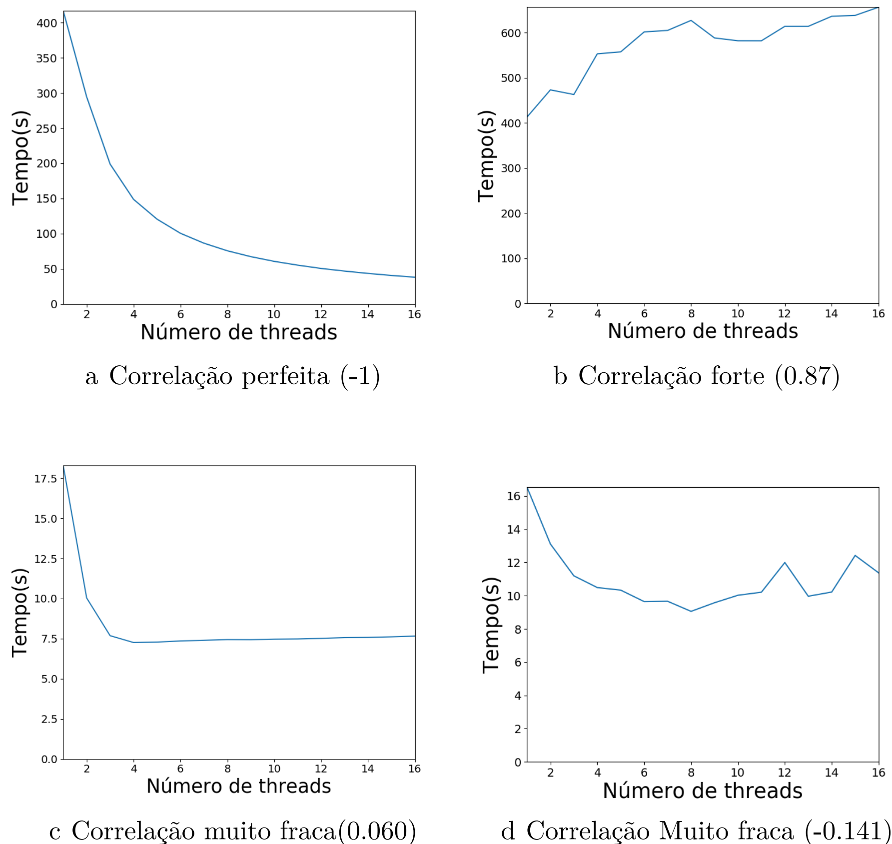
Tabela 1 – Tabela de Correlação

O método é definido da seguinte forma:

$$\rho = 1 - \frac{6\sum_{i=1}^{n} d^2}{n(n^2-1)} \tag{2.1}$$

onde $d^2 = R(x_i) - R(y_i)$ é a diferença de postos da i-ésima posição de x e y.

O valor obtido por este método varia entre 1 e -1, a Tabela 1 exibe como podem ser interpretados os valores obtidos pelo método de Spearman e a Figura 6 exibe como o tempo de execução (eixo y) de uma aplicação paralela se comporta ao longo de 16 *threads* (eixo x) juntamente com o seu coeficiente de correlação.



a Correlação perfeita (-1)

b Correlação forte (0.87)

c Correlação muito fraca(0.060)

d Correlação Muito fraca (-0.141)

Figura 6 – Exemplos de correlação de Spearman

Quando aumenta-se o número de *threads* e há diminuição no tempo de execução, a correlação passa a ser perfeita negativa, tal como ilustra a Figura 6a, por outro lado, quando aumenta-se o número de *threads* e há um aumento no tempo de execução tal como

pode ser visto na Figura 6b, esta é uma correlação forte positiva. Porém, quando a partir de um determinado número de *threads* não há ganho de desempenho como pode ser visto na Figura 6c ou quando há oscilações de tempo entre as diferentes *threads* (Figura 6d), a correlação então passa a ser fraca, positiva em 6c e negativa 6d.

2.4 Aprendizado de Máquina

Aprendizado de máquina utiliza a teoria da estatística para construir um modelo matemático, uma vez que a sua tarefa principal é fazer inferências a partir de uma amostra. O papel do aprendizado de máquina na ciência da computação é duplo: primeiro, no treinamento, através de algoritmos eficientes para resolver o problema de otimização bem como para armazenar e processar a enorme quantidade de dados. Segundo, uma vez que um modelo é aprendido, sua representação e solução algorítmica para inferência também precisam ser eficientes. Em certas aplicações, a eficiência do algoritmo de aprendizagem ou inferência, nomeadamente, a sua complexidade espacial e temporal, pode ser tão importante quanto sua precisão preditiva (ALPAYDIN, 2014).

Em aprendizadmo de máquina, existem três tipos principais de aprendizado: supervisionado, não supervisionado, e por reforço. Eles são brevemente discutidos a seguir:

- O aprendizado de máquina do tipo supervisionado é uma técnica de aprendizado que recebe um conjunto de dados, onde estes possuem seus próprios atributos e, entre os atributos do conjunto a informação que queremos encontrar está presente. Esta informação é também conhecida como rótulo (MARSLAND, 2009). Um dos métodos que fazem parte deste tipo de aprendizado são os métodos de regressão, no qual será descrito na subseção a seguir.

- Os métodos que compõe o aprendizado não supervisionado, não são providos dos resultados de saída, isso os torna adequados para uso em tarefas de processamento que são complexas em comparação com os algoritmos de aprendizado supervisionados (BURNS, 2019).

- O aprendizado por reforço ocorre quando um algoritmo é apresentado com exemplos que não possuem rótulos, similar ao aprendizado não supervisionado. Contudo, o exemplo pode ser acompanhado por uma recompensa positiva ou negativa, dependendo da solução no qual o algoritmo foi proposto (BURNS, 2019).

2.4.1 Regressão e o Método KNN

Uma análise de regressão do tipo paramétrica gera uma equação (e.g., $Y = \alpha + \beta X$) para descrever a relação estatística entre uma ou mais preditoras e a variável de resposta para predizer novas observações. A regressão linear normalmente usa o método de estimativa de mínimos quadrados ordinários que deriva a equação minimizando a soma

dos resíduos quadrados. Entretanto, uma análise de regressão não paramétrica é composta por encontrar um passado semelhante do conjunto de treinamento utilizando uma medida de distância adequada e interpolando eles para encontrar uma saída correta.

Para realizar uma regressão não-paramétrica, métodos baseados em vizinhança podem ser utilizados, tal como o método KNN. Este é um algoritmo utilizado de maneira geral para classificar objetos com base nos exemplos de treinamentos que estão mais próximos de um determinado espaço de características. Porém, este também pode ser utilizado para realizar regressões sobre um determinado conjunto de dados. Para realizar a implementação do algoritmo KNN são necessários além do conjunto de dados:

- Um método para realizar o cálculo de distância entre os exemplos de treinamento;

- Definir um valor K (número de vizinhanças) no qual serão utilizados para interpolar o ponto a ser predito;

O KNN utilizado para regressão calcula a distância de uma nova amostra para todos os elementos do conjunto. Com as distâncias calculadas, a nova amostra irá receber os valores com base em uma média aritmética entre os seus K vizinhos mais próximos. Não há uma regra específica para a escolha de um valor de K, porém para determinar tal valor pode-se utilizar a segunite equação: $k = \sqrt{n}$, onde n é o número total de amostras. Para realizar o cálculo da distância, existem alguns métodos conhecidos (e.g., Manhatan, Minkoviski, Mahalanobis) e existe também a distância Euclidiana, conforme é definido em 2.2.

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2} \tag{2.2}$$

A Figura 7 exibe o tempo de execução (Linha azul) da aplicação LAVAMD (CHE et al., 2009) sendo executada ao longo de 16 *threads*, onde estes dados de tempo foram utilizadas para o treinamento do método KNN, onde deste resultou-se na regressão (Linha tracejada). Neste exemplo utilizou-se o método euclidiano para medir as distâncias e o valor de $k = 4$.

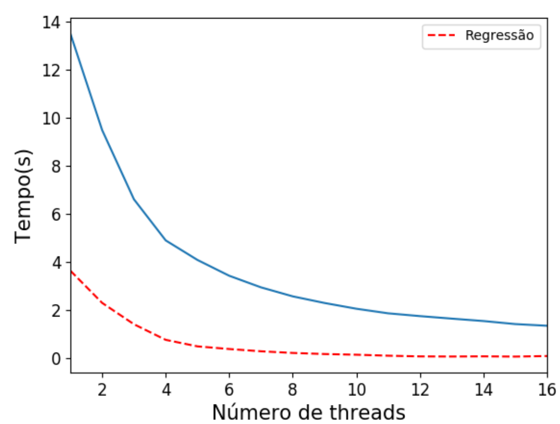


Figura 7 – Exemplo de Regressão - KNN, $k = 4$

3 TRABALHOS RELACIONADOS

Este capítulo irá discutir de forma sucinta os trabalhos desenvolvidos sobre análise de desempenho e consumo energético de aplicações que utilizam diferentes interfaces de programação paralela em ambientes *multi-core* e, trabalhos que apresentem métodos preditivos para tais aplicações que possuem similaridade com esta monografia e que serviram também de auxílio para o desenvolvimento desta. Os trabalhos estão organizados de acordo com o ano de publicação.

Em 2002, Taylor et al. (TAYLOR et al., 2002) propuseram um modelo analítico pra prever o desempenho de três aplicações paralelas do NAS *Parallel Benchmark*: BT, LU e SP. O modelo consiste em um acoplamento de desempenho, no qual quantifica a iteração entre *kernels* adjacentes em aplicações paralelas, dando mais acurácia ao modelo. Os resultados foram validados em uma máquina com 80 processadores, mostrando que quanto maior o acoplamento de desempenho, melhor é a acurácia do modelo.

Para reduzir a sobrecarga de desempenho de um modelo preditivo, Yang et al. (YANG et al., 2005) propuseram uma abordagem que utiliza execuções parciais de uma aplicação para prever seu comportamento. A idéia é prever o tempo de execução total de uma aplicação em larga escala através da execução de um pequeno *test drive* da aplicação. Dois *benchmarks* da ASCI Purple *suite* foram utilizados para validar o modelo em dez diferentes plataformas *multicore*. Os resultados mostram que a abordagem proposta pode prever o desempenho com uma acurácia de até 97% ou mais. Além disto, no melhor caso, ele adiciona uma sobrecarga de apenas 1% no tempo de execução total. No entanto, o trabalho foca apenas na otimização de desempenho, deixando de lado o consumo de energia e EDP.

O trabalho proposto por Ipek et al (IPEK et al., 2005) foi de empregar o uso de redes neurais artificiais multicamada para prever o comportamento de desempenho de aplicações paralelas, onde estas foram treinadas com base nos dados dados de execução da plataforma no qual foi alvo do estudo. O modelo foi desenvolvido para ser automático utilizando apenas os dados passados como entrada. Como *benchmark* foi utilizado a aplicação SMG2000. O modelo proposto foi executado em duas plataformas paralelas de grande escala onde obtiveram erros de 5% e 7% em um espaço de parâmetros amplo e multidimensional.

Uma comparação entre um processador *single-core* e *dual-core* utilizando um processador AMD Opteron foi apresentado por Pase et al. (PASE; ECKL, 2005). Os autores mediram o desempenho em ambos os processadores com o *Hight Performance LINPACK* (HPL) *benchmark* e mostraram que o processador *dual-core* é até 60% mais rápido que o *single-core*. Eles validaram também a latência da memória e a sua vazão, mostrando que o *dual-core* possui uma vazão 10% maior que o processador *single-core*.

Em Singh et al (SINGH et al., 2007) foi proposto um modelo automático baseado em redes neurais multicamadas artificiais, onde esta foi treinada com base nos dados de

execução das aplicações. Foram utilizados duas aplicações, SMG2000 e o LINPACK, onde puderam mostrar de forma esparsa dados de desempenho em duas plataformas radicalmente diferentes em espaços de parâmetros grandes e multidimensionais e mostraram que os modelos baseados nesses dados podem prever o desempenho em 2% a 7% dos tempos reais de execução das aplicações.

Barnes et al. (BARNES et al., 2008) explorou 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. Eles propuseram 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 pŕe-processada para manipular a computação e a comunicação separadamente. O modelo proposto foi validado executando 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%.

Mallón et al. (MALLóN et al., 2009) comapararam o desempenho de MPI, *Unified Parallel C*(UPC), e OpenMP em arquiteturas *multicore* através do subconjunto de aplicações do NAS *Parallel Benchmark*. As execuções foram realizadas em um ambiente com memória híbrida (distribuída e compartilhada) e apenas memória compartilhada. Os resultados mostram que para o ambiente de memória híbrida, MPI tem no geral melhor desempenho comparado com o UPC devido ao melhor uso da memória cache. Quando considerando o ambiente de memória compartilhada, apesar da comunicação entre as *threads* do OpenMP ocorrerem através de memórias compartilhadas, MPI e UPC performaram melhor em ambos os casos, devido ao melhor uso da memória *cache*.

Araujo et al. (ARAUJO et al., 2010) realizou uma análise sobre tempo e de consumo de energia de duas interfaces de aplicações paralelas MPI e OpenMP em um arquitetura de memória compartilhada. Onde puderam concluir que os *benchmarks* desenvolvidos em MPI apresentaram melhores resultados em relação a OpenMP. Na mesma linha, Balladini et al. (BALLADINI et al., 2011) analizaram a influência do OpenMP e MPI no consumo de energia e o ambiente dos sistemas em diferentes frequências de *clock* das CPUs. Quatro aplicações do NAS, escritos em OpenMP e MPI foram executadas em um processador *dual socket* Intel Xeon *dual core*. Os resultados mostram que o tempo de execução, eficiência energética, e o gasto maximo de energia depende não só apenas do tipo de aplicação mas também do modelo de implementação do programa.

Em Huang et al. (HUANG et al., 2010) foi proposto um modelo de regressão baseado em polinônio, onde foram extraídas características durante a execução do programa e utilizadas como amostras para o modelo. A forma compacta e explicitamente polinomial do modelo estimado pode revelar informações importantes sobre o programa de computador (por exemplo, recursos e suas combinações não lineares que dominam o tempo de

execução), permitindo uma melhor compreensão do comportamento do programa. Foram utilizados três programas muito comum na acadêmia para avaliar o método predito, onde obtiveram erros menores que 7% utilizando um pequeno conjunto de amostras.

Redes neurais artificiais (ANNs) foram utilizadas em (TIWARI, 2012) para prever o consumo energético 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%.

(PORTERFIELD et al., 2013) analizou os fatores que podem influenciar o consumo de energia de aplicações OpenMP compiladas com *Intel C++ Compiler* (ICC) e *GNU Compiler Collection* (GCC). Por meio de contadores de desempenho presentes na microarquitetura Intel SandyBridge, os autores mediram o consumo de energia de uma variedades de programas em OpenMP. A avaliação revelou variações no consumo de energia dependendo do algoritmo, o level de otimização deste compilador, o número de *threads* e a temperatura do chip. Na maiora das aplicações, o aumento no número de *threads* permite melhor desempenho, mas com um aumento substâncial no consumo de energia.

Em (CAO et al., 2013) os autores analizaram o desempenho de escalabilidade das interfaces de programação paralela OpenMP, Intel TBB e Cilk++. Quinze aplicações paralelas de diferentes domínios foram executadas em um sistema de 32 *cores*. Os resultados mostram que a maioria das aplicações OpenMP não são linearmente escaláveis com 32 *threads*. Entre as razões para este desempenho "pobre", os autores afirmam que a estratégia de escalonamento fornecida pelos sistema de tempo de execução pode impactar negativamente o desempenho.

O trabalho desenvolvido por Fasiku et al. (FASIKU et al., 2014) apresentou uma comparação de desempenho entre os processadores AMD e Intel *dual-core*. Foi utilizado o *benchmark Standard Performance Evaluation Corporation (SPEC)* para medir o desempenho de ambos os processadores. Os resultados experimentais mostram que o Intel *dual-core* é em torno 6.62% mais rápido que o processador AMD. Além do mais, os autores compararam a vazão de ambos, no qual este foi 1.06 vezes maior que o processador Intel. De acordo com os autores, Intel foi melhor porque tem uma comunicação entre núcleos mais rápida, compartilhamento dinâmico de *cache* entre os núcleos e tamanho menor de cache L2 quando comparado com o processador AMD.

Lorenzon et al. (LORENZON, 2018) propôs Aurora, um *framework* para aplicações OpenMP que visa encontrar o melhor número de *threads* em tempo de execução para determinada região paralela, tendo um baixo *overhead*, sendo esta totalmente transparente para o usuário, sem realizar modificações de código ou recompilação. Foram

executados 15 *benchmarks* em quatro processadores *multi-core*. Aurora obteve ganhos em EDP de até 98%, 86% e 91% em comparação com a execução padrão do OpenMP.

3.1 Contexto deste Trabalho

Conforme apresentado na Seção anterior, os trabalhos que objetivam medir e comparar o consumo de energia e desempenho de aplicações utilizando diferentes arquiteturas de processadores foram desenvolvidos por (FASIKU et al., 2014) e (PASE; ECKL, 2005). Também analisaram tais métricas os trabalhos propostos por (ARAUJO et al., 2010), (CAO et al., 2013) e (BALLADINI et al., 2011), porém nestes as avaliações foram feitas sobre diferentes interfaces de programação paralela, e em, (MALLóN et al., 2009) foi verificado também como estas interfaces se comportam em diferentes organizações de memória. Por fim, os trabalhos de (BARNES et al., 2008), (TIWARI, 2012), (TAYLOR et al., 2002) e (YANG et al., 2005), utilizaram diferentes técnicas de aprendizado de máquina (e.g., regressões paramétricas e redes neurais) para realizar previsões sobre consumo de energia ou sobre desempenho, apenas.

Desta forma este trabalho final de conclusão de curso visa extender os trabalhos discutidos acima através da verificação de como uma aplicação paralela se comporta em termos de desempenho e de consumo de energia quando submetida a diferentes tamanhos de conjuntos de entrada. E, utilizar diferentes estratégias (e.g., método de regressão não paramétrico e treinamento *off-line*) sobre os dados obtidos através da execução da aplicação com um conjunto de entrada pequena para prever seu comportamento de desempenho, consumo de energia quando executada com um conjunto de entrada grande.

4 METODOLOGIA

Este capítulo irá apresentar os *benchmarks* utilizados como experimentos neste trabalho, descrevendo de forma breve cada um destes, bem como o tamanho de entrada utilizados nas execuções na Seção 4.1. A Seção 4.2 irá apresentar o ambiente de execução onde os *benchmarks* foram submetidos a testes. As métricas que foram avaliadas no qual são de interesse desta trabalho são descritas na Seção 4.3. Para identificar como uma aplicação paralela se comporta quando executada no seu menor tamanho de entrada em relação a outros tamanhos, será descrito na Seção 4.4. Na Seção 4.5 será abordado a implementação do método KNN. Por fim, na Seção 4.6 será apresentado os cenários de validação que foram utilizados para comparar os métodos de predição.

4.1 *Benchmarks*

Dezessete aplicações de diferentes conjuntos de benchmarks foram utilizadas: 7 *kernels* do NAS Parallel Benchmark (BAILEY et al., 1994); 7 aplicações da suíte Rodinia (CHE et al., 2009); 3 aplicações amplamente conhecidas pela comunidade acadêmica: método de Jacobi e *STREAM* e o *benchmark* LULESH (KARLIN; KEASLER; NEELY, 2013). As aplicações são descritas nas subseções seguintes.

4.1.1 NAS Parallel Benchmark

O NAS é um conjunto de pequenos programas desenvolvidos para avaliar o desempenho de super computadores. Os seguintes kernels foram considerados:

- ***Integer Sort* (IS)**: Ordenação de inteiros com acesso aleatório a memória.
- **Embarrassingly Parallel (EP)**: Fornece uma estimativa de limites superiores alcançáveis para pontos flutuantes sem comunicação significativa entre os processadores.
- **Conjugate Gradient (CG)**: Método do gradiente conjugado que computa a aproximação para um menor auto-valor de uma matriz, esparsa, grande, positiva.
- **Multi-Grid (MG)**: Obter uma solução aproximada de um problema discreto de Poisson utilizando o método *V-cycle Multigrid*.
- ***Lower-Upper Gauss-Seidel solver (LU)*** : Resolução de uma matriz cujo os elementos estão acima da diagonal superior ou abaixo dela fazendo uso do método *Symmetric Successive Over-Relaxation* (SSOR).
- ***Data Cube* (DC)**: Realiza movimentação de dados entre os diferentes níveis de memória, utilizando grandes quantidades de dados.

- ***Unstructured Adaptive mesh* (UA)**: Mede o desempenho de acessos irregulares à memória, utilizando malhas não estruturadas, fazendo uso de equações de transferência de calor.

A tabela 2 exibe os tamanhos de entrada utilizados nos experimentos para as aplicações do NAS. Para melhor entedimento, estas entradas foram classificadas como Pequena, sendo esta o menor tamanho de entrada, Média, Grande e Muito Grande, sendo as duas últimas os maiores tamanhos disponível para cada aplicação.

Aplicação	**Tamanho das Entradas**			
	Pequena	**Média**	**Grande**	**Muito Grande**
EP	2^{24}	2^{28}	2^{30}	2^{32}
MG	128^3	256^3	256^3	512^3
CG	1400	14000	75000	150000
IS	65536	8388608	33554432	134217728
LU	33^3	64^3	102^3	162^3
DC	100000	-	1000000	-
UA	4	6	7	8

Tabela 2 – Tamanho das entradas das aplicações NAS utilizadas nos experimentos.

4.1.2 Rodinia

Rodinia é um outro conjunto de *benchmark* que mede a performance de CPUs *multicore* e *Graphics Processing Unit* (GPU). Este conjunto é caracterizado por possuir uma ampla variedade de padrões de comunicação paralela, técnicas de sincronização e consumo de energia.

***K-means*(KM)** é um algoritmo de agrupamento muito utilizada em mineração de dados, buscando classificar uma nova amostra com base nos seus *k* vizinhos mais próximos.

***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.

***HotSpot*(HS)** é uma ferramenta de simulação técnica usada para estimar a temperatura de um processador baseada na arquitetura *floor plan* e simula medidas de consumo energético. Este *benchmark* inclui uma simulação térmica transiente 2D, onde iterativamente computa uma série de equações diferenciais por blocos de temperaturas.

***Back Propagation*(BP)** é um algoritmo de aprendizado de máquina que treina os pessos de nós conectados em uma rede neural. A aplicação é baseada em duas fases: a fase adiante, no qual as ativações são propagadas para a entrada da camada de saída, e a fase de retorno, onde o erro entre os valores observados e os requisitados são propagados para as camadas anteriores para ajustar os pesos e o *bias*

SRAD é um algoritmo de difusão baseado em equações diferenciais parciais e é utilizado para remover irregularidades em uma imagem sem sacrificar as características importantes. É utilizado mundialmente em imagens de ultrassom e aplicações de imagens de radar.

***Leukocyte Tracking*(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.

***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.

A Tabela 3 exibe os tamanhos de entrada utilizado nas aplicações do conjunto RODINIA. De forma similar aos do conjunto NAS, para melhor entendimento, as entradas foram classificadas como Pequena (P), no qual esta representa o menor tamanho de entrada para a aplicação , Média (M), Grande(G), Muito Grande (MG) e Extragrande (EG), sendo as duas últimas os maiores tamanhos disponível para entrada nas aplicações.

Aplicação	**Tamanho das Entradas**				
	Pequena	**Média**	**Grande**	**Muito Grande**	**Extragrande**
BP	20×10^2	20×10^4	10×10^5	20×10^6	-
HS	256^2	512^2	1024^2	-	-
KM	204800	-	819200	-	-
LM	5^3	15^3	20^3	25^3	30^3
LT	16	32	64	128	256
NW	4096^2	10×10^3	20×10^3	30×10^3	40×10^3
SRAD	512^2	1024^2	2048^2	4096^2	-

Tabela 3 – Tamanho das entradas das aplicações Rodinia utilizados nos experimentos.

4.1.3 Outros Benchmarks

A Tabela 4 mostra os tamanhos de entrada utilizados nas aplicações, sendo elas Pequena (P) o menor tamanho, Média (M), Grande (G), Muito Grande (MG) e Extragrande (EG), onde as duas últimas classificações representam o maior tamanho utilizado nas execuções.

Aplicação	**Tamanho das Entradas**				
	Pequena	**Média**	**Grande**	**Muito Grande**	**Extragrande**
JA	128^2	512^2	1024^2	2048^2	4096^2
STREAM	10×10^6	20×10^6	40×10^6	80×10^6	-
LL	20^3	30^3	40^3	50^3	60^3

Tabela 4 – Tamanho das entradas Jacobi, *STREAM* e LULESH utilizados nos experimentos.

4.1.3.1 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. Tal método pode ser definido da seguinte forma:

$$x_i^{k+1} = \frac{1}{a_{ii}} \left[b_i - \left(\sum_{\substack{j=1 \\ j \neq i}}^{j=n} a_{ij} x_j^{(k)} \right) \right] \quad i = 1, 2, ..., n \tag{4.1}$$

As iterações continuam até que a diferenteça entre os valores obtidos nas sucessivas iterações sejam muito pequenas. As iterações podem ser interrompidas quando o valor absoluto do erro relativo estimado de todas as incógnitas for menor que algum valor predeterminado (GILAT et al., 2008):

4.1.3.2 *STREAM*

STREAM 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).

4.1.3.3 LULESH (LL)

O LULESH é um aplicativo altamente simplificado, codificado apenas para resolver um problema simples de explosão de Sedov com respostas analíticas, mas representa os algoritmos numéricos, o movimento de dados e o estilo de programação típico em aplicativos científicos baseados em C ou C ++ (KARLIN; KEASLER; NEELY, 2013).

4.2 Ambiente de Execução

Os experimentos deste trabalho foram realizados num processador Ryzen 7 1700, equipado com oito núcleos de processamento que operam em uma frequência base de 3.0 GHZ e uma frequência máxima de 3.7 GHZ (i.e., quando o Turbo CORE está ativo), com suporte a tecnologia *Simultaneous Multi-Threading (SMT).* O sistema de memória do processador consiste em três níveis de *cache* onde a L1 possui 768 Kb de memória para dados e instruções, a L2, 4 MB para dados, e a cache L3 16 MB. As *caches* L1 e L2 são privadas para cada um dos núcleos e a memória L3 é compartilhada entre pares de quatro *cores*. Em todas as execuções, o *governor* do *Dynamic Voltage and Frequency Scalling* (DVFS) foi configurado como *ondemand* (padrão do sistema operacional Linux), no qual o processador opera conforme a carga de demanda da aplicação. Também foi configurado a afinidade da CPU, para uma afinidade rígida, pois esta medida permite

organizar melhor onde cada *thread* deverá ser executada na CPU. A Figura 8, mostra como estão organizadas as memórias cache do processador Ryzen 7 1700.

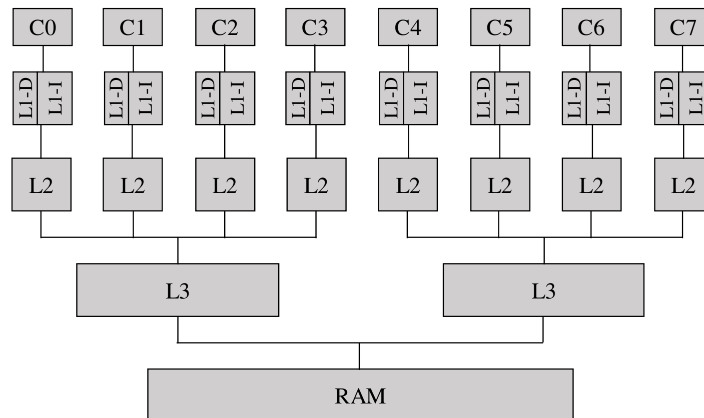


Figura 8 – Organização das memórias cache do processdor Ryzen 7 1700

Com base no trabalho de Marques et al. (MARQUES et al., 2019), a tecnologia *Turbo Core* presente no processador Ryzen 7 1700, não apresenta ganhos significativos de desempenho e redução no consumo de energia quando o número de *threads* é superior a quatro em aplicações paralelas. Neste sentido, optou-se por desativar esta tecnologia no desenvolvimento deste trabalho.

Os *benchmarks* foram executados com diferentes números de *threads*: de 1 que representa a execução sequencial até 16, que é o máximo permitido pelo ambiente de execução. O compilador utilizado foi o GCC 8.1.0, onde as aplicações foram compiladas com base nas configurações disponibilizadas por cada uma das suites. O sistema operacional utilizado para os testes foi o Ubuntu server 16.04 com versão do *kernel* 4.15.

4.3 Métricas Avaliadas

Para realizar a coleta dos dados utilizou-se o *framework* Aurora desenvolvido por Lorenzon, et al. (LORENZON, 2018). Com os dados coletados com o auxílio do *framework*, foi possível analisar as métricas que são de interesse deste trabalho: tempo de execução, consumo de energia e EDP.

Tempo de Execução: O tempo total de execução de uma aplicação são classificados de acordo com o uso de dos seguintes recursos: CPU, usuário e tempo do sistema. No primeiro, é calculado o tempo total que o CPU utilizou para processar instruções do programa ou do sistema operacional. No segundo, é o tempo total executado somente pela aplicação. E o último, é o tempo que o processador gasta no sistema operacional executando as tarefas da aplicação (LORENZON, 2014). Esta foi obtida através da função *omp_get_wtime()* própria do OpenMP.

Em computação paralela utiliza-se também a métrica chamada *speedup*. Esta taxa mostra o quão mais rápida uma aplicação paralela é em comparação com a execução

sequencial. Considerando apenas o desempenho, quanto maior o *speedup*, melhor é o desempenho da implementação paralela. A equação que define o *speedup* é descrita abaixo:

$$Speedup = \frac{TempoSequencial}{TempoParalelo} \tag{4.2}$$

Consumo de Energia: O consumo de energia pode ser descrito de duas principais formas: dinâmica e estática. No primeiro, a energia é consumida enquanto entradas são ativadas, com uma capacitância carregando e descarregando, sendo ela proporcional à atividade de troca do circuito, conforme dado a equação 4.3.

$$Pdinâmica = CV^2Af \tag{4.3}$$

onde C é a capacitância, V é a tensão de alimentação, A é a taxa de troca em um período de relógio sobre o total de portas lógicas e, f é a frequência com qual a porta esta sendo chaveada.

O consumo estático consiste na energia que é consumida quando o *hardware* é ligado conforme a equação 4.4, onde V é a tensão de alimentação e I é o total de corrente que flui sobre o dispositivo.

$$Pestático = I \times V \tag{4.4}$$

Energia em *joules* é o total de energia consumida (P) sobre um tempo (T) é determinada pela integral dada na equação 4.5. Esta métrica foi obtida através da biblioteca APM desenvolvida por (HACKENBERG, 2013) no qual foi atribuída a *framework* Aurora, para obter estes valores de processadores AMD.

$$Energia = \int_0^T P_i \tag{4.5}$$

Energy-Delay Product: Esta métrica é usada para estudar o custo-benefício entre o total de energia consumida e o tempo de execução de uma aplicação. Sua fórmula, consiste na multiplicação da energia consumida pelo tempo de execução (LORENZON, 2014), é mostrada abaixo.

$$EDP = Energia \times Tempo \tag{4.6}$$

4.4 Correlações

Para verificar se uma aplicação paralela se comporta de maneira similar para às métricas tempo de execução, consumo de energia e EDP sobre diferentes tamanhos de entrada, foram medidos os coeficientes de correlação da aplicação sobre as métricas em relação ao número de *threads*. Para tal, foi utilizado o método de correlação de Spearman definido na Seção 2.3. Inicialmente, os coeficientes foram obtidos de forma individual

para cada tamanho de entrada para uma determinada métrica (e.g., tempo de execução, consumo de energia e EDP). Posteriormente, utilizou-se a equação 4.7 para verificar correlação do menor tamanho (P) em relação aos tamanhos de entrada Grande, Muito Grande e Extragrande.

$$R = \sqrt{\frac{r^2_{yx_1} + r^2_{yx_2} - 2 \times r_{yx_1} \times r_{yx_2} \times r_{x_1x_2}}{1 - r^2_{x_1x_2}}} \tag{4.7}$$

onde r_{yx_1} é a correlação entre x_1 e y, r_{yx_2} a correlação entre x_2 e y e, x_1x_2 e a correlação entre x_1 e x_2, onde o valor a ser obtido é positivo indicando a força entre a corelação.

4.5 KNN

Para realizar uma regressão sobre os dados de menor entrada, foi implementado o método KNN na linguagem de programação Python 3.6.7, devido a sua fexibilidade de se ajustar a diferentes comportamentos e pela sua facilidade de implementação. Inicialmente são passados como parâmetro para o método, uma matriz coluna contendo os dados de uma determinada métrica obtida previamente, um vetor de amostras no qual se deseja realizar o treinamento, onde neste trabalho utilizou-se os mesmos dados originais para treinamento, um valor de k para buscar-se os k valores mais próximos de uma determinada amostra e realizar a média simples entre eles, no qual optou-se pelo valor de $k = 4$. O método implementado pode ser visto no algoritmo 1.

```
Data: X, sample, k
Result: value
array_distances = [];
dimension = X.shape;
for j to dimension[1] do
    for i to dimension[0] do
        array_distances.append(sqrt(pow(sample[j] - X[i][j]),2));
    end
    ordered = sorted(array_distances)[:k];
end
value = sum(ordered)/k
```

Algorithm 1: Método KNN implementado na linguagem de programação Python 3.6.7.

4.6 Cenário de Validação

Para validar se uma aplicação paralela pode ser otimizada utilizando apenas um cojunto muito pequeno de entrada através de aprendizado de máquina, foram considerados quatro diferentes cenários comparativos:

- ***Baseline***: a aplicação é executada com o número máximo de *threads* do *hardware* presente no sistema, isto é, 16, no maior tamanho de entrada.

- **Pequeno (P)**: Este cenário consiste dos seguintes passos:
  - Executar a aplicação sobre o menor tamanho de entrada ao longo de 16 *threads*.
  - Obter a *thread* que possui menor valor entre as métricas.
  - Em posse da *thread* com menor valor, esta foi utilizada para realizar a computação da aplicação no maior tamanho de entrada.
- **KNN**: Este cenário visa utilizar os dados obtidos após a execução da aplicação sobre o menor tamanho de entrada para serem utilizados como treinamento para o método KNN, onde os seguintes passos foram adotados:
  - Executar a aplicação com o seu menor tamanho de entrada ao longo das 16 *threads*.
  - Utilizar os resultados obtidos ao longo das 16 *threads* como conjunto de treinamento para o método de regressão utilizando o KNN.
  - Sobre o resultado da regressão, foi obtida a *thread* que possui menor valor entre as métricas e, esta foi utilizada para executar a aplicação no seu maior tamanho de entrada.
- ***Oracle***: A aplicação é executada com o melhor número de *threads* para cada métrica no maior tamanho de entrada. Tal número foi obtido através da execução exaustiva da aplicação com os diferentes números de *threads*: 1 até 16.

5 RESULTADOS

Este capítulo apresenta os resultados de correlação múltipla para as métricas tempo de execução, consumo de energia e EDP para cada um dos *benchmarks* executados sobre diferentes tamanhos de entrada ao longo de 16 *threads* (Seção 5.1). Também, serão apresentados os resultados comparativos entre os cenários de validação, onde primeiramente será comparado os cenários P e KNN em relação ao *Baseline* na Seção 5.2. E, por fim a Seção 5.3 discutirá os resultados comparativos entre os cenários P, KNN e *Baseline* em relação ao *Oracle*. Os resultados de comportamento de cada métrica tempo de execução, consumo de energia e EDP de todas as aplicações estão presentes apêndices A, B e C respectivamente.

5.1 Análise da Correlação entre Entradas de Diferentes Tamanhos

A Tabela 5.1 exibe os coeficientes de correlação múltipla utilizando a equação 4.7 para cada uma das métricas avaliadas: tempo de execução, consumo de energia e EDP. A correlação foi realizada entre os conjuntos de entrada pequeno (P) para os tamanhos Grande (G), Muito Grande (MG) e Extragrande (EG). Os resultados são expressos entre 0 e 1 (quanto mais próximo de 1, mais forte é a associação entre as variáveis) e demonstram a intensidade de corelação entre o menor e maior tamanho de entrada. Importante destacar que para as aplicações onde não foi possível executar com um conjunto de tamanho grande por razões de *hardware* insuficiente (e.g., pouca memória RAM para alocar as estruturas de dados), o resultado é representado por um hífen (–);

BENCHMARK	**TEMPO**			**ENERGIA**			**EDP**		
	P/G	**P/MG**	**P/EG**	**P/G**	**P/MG**	**P/EG**	**P/G**	**P/MG**	**P/EG**
BP	0,61	0,32	-	0,34	0,35	-	0,47	0,32	-
CG	0,91	0,90	-	0,97	0,94	-	0,95	0,91	-
DC	0,71	-	-	0,72	-	-	0,83		-
EP	1	1	-	1	1	-	1	1	-
HS	0,97	-	-	0,91	-	-	0,92	-	-
IS	1	1	-	1	1	-	1	1	-
JA	0,99	0,99	0,99	0,99	0,99	1	1	1	1
KM	0,86	-	-	0,88	-	-	0,80		-
LL	0,96	0,95	0,96	0,92	0,92	0,90	0,95	0,95	0,96
LM	1	1	1	1	1	1	1	1	1
LT	0,97	1	0,98	0,96	0,99	0,46	0,98	0,98	0,97
LU	0,91	0,95	-	0,84	0,54	-	0,87	0,94	-
MG	0,66	0,79	-	0,78	0,96	-	0,97	0,91	-
NW	1	1	1	1	1	1	1	1	-
SRAD	0,86	0,86	-	0,89	0,89	-	0,79	0,78	-
STREAM	1	0,70	-	1	0,77	-	1	0.65	-
UA	1	1	-	0,55	0,80	-	0,71	0,70	-

Tabela 5 – Coeficientes de correlação múltipla do tamanho P para G, MG e EG

A Tabela destaca correlações múltiplas perfeitas (1) para as aplicações EP, IS, LM e NW independente da métrica. Isto demonstra que estas aplicaçoes se comportam de

maneira similar quando são executadas com os conjuntos de entrada pequeno e grandes. Considerando apenas o tempo de execução, as aplicações CG, JA, LT, LU também possuem correlações perfeitas. Por outro lado, as aplicações que possuem maior diferença nos valores de correlação para tempo são as aplicações BP, MG e *STREAM*, demonstrando que há uma diferença entre o comportamento da aplicação com menor tamanho em relação aos maiores.

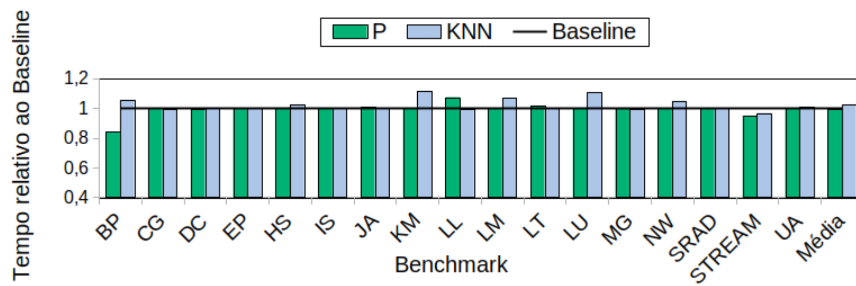
Quando considerada apenas o consumo de energia de maneira geral, as aplicações possuem correlação forte entre os conjuntos de entrada. No entanto, algumas aplicações apresentam diferenças significativas nos seus valores de coeficientes, como por exemplo, LU,UA, *STREAM* e MG. Já para o EDP, a mudança mais significativa fica por conta da aplicação BP, tendo valores de coeficiente de 0,47 e 0,32 para P/G e P/MG respectivamente. Duas aplicações possuem somente um coeficiente de correlação múltipla sendo elas DC e HS, tendo estas correlações para a métrica tempo de 0,71, 0,97 respectivamente, para a métrica energia 0,72, 0,91 e, por último para EDP 0,83, 0,92.

5.2 Desempenho, Energia e EDP em relação ao *Baseline*

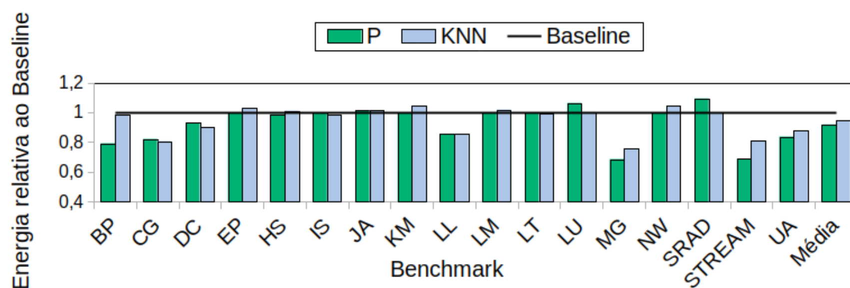
A Figura 9 apresenta os resultados de desempenho (Figura 9a), consumo de energia (Figura 9b) e EDP (Figura 9c) de todos os *benchmarks* comparando os cenários P e KNN em relação ao *Baseline* juntamente com a sua média geométrica.

P em relação ao *Baseline*: como pode ser observado na Figura 9, na maioria dos casos, não há ganhos significativos de desempenho tendo este uma média de redução de 0.009% para todos os *benchmarks*. As aplicações BP e *STREAM* são as aplicações que possuem reduções significativas de 16% e 6% no seu tempo de execução respectivamente, mas para a aplicação LL há um aumento de 7%. Por outro lado, há reduções significativas no consumo de energia e EDP, tendo uma média de redução de 9% e 10%, no qual, as maiores reduções ficam por conta das aplicações MG e *STREAM*, tendo estas ganhos de 32% no consumo de energia em ambos e, 33% e 35% para EDP respectivamente.

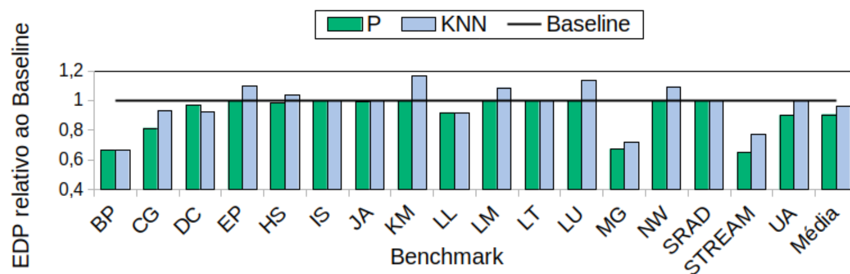
KNN em relação ao *Baseline*: observando à Figura 9 podemos notar que o método KNN não melhora de forma geral o desempenho das aplicações, tendo apenas uma redução na aplicação *STREAM* de 4% e, em média possui um acréscimo no tempo de execução de apenas 2%. Porém, o método possui reduções no consumo de energia e EDP, onde as maiores reduções de consumo de energia são nas aplicações CG, MG e *STREAM* sendo elas de 20%, 15% e 19% respectivamente, tendo como média uma redução de 6% no consumo. Para EDP, as maiores reduções foram obtidas nas aplicações BP, MG e *STREAM* sendo elas de 34%, 28% e 23% respectivamente, tendo uma média geral de redução para EDP de 4%.



a Tempo de execução



b Consumo de energia



c EDP

Figura 9 – Pequeno e KNN em relação ao *Baseline*: menor que 1.0 significa que estes cenários são melhores que o *Baseline*.

5.3 Desempenho, Energia e EDP em relação ao *Oracle*

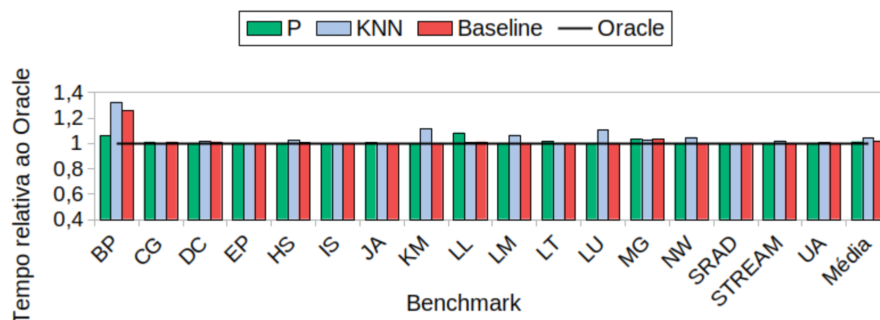
A Figura 10 apresenta os resultados de desempenho na Figura 10a, consumo de energia na Figura 10b e EDP na Figura 10c onde nesta consta um comparativo entre os cenários P, KNN, *Baseline* em relação ao *Oracle*.

P em relação ao *Oracle*: como pode ser visto na Figura 10, o cenário P possui um acréscimo médio de apenas 1% para tempo de execução, tendo como maiores aumentos as aplicações LL e BP, onde estas possuem uma adição de 8% e 5%, porém, em 11

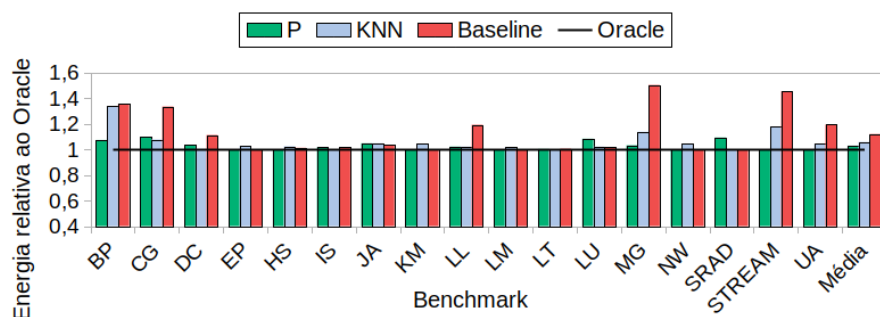
aplicações o tempo destas é igual ao tempo do melhor caso. Para a métrica energia, este cenário apresenta um aumento médio de apenas 2% em relação ao melhor caso, sendo as aplicações que possuem maior acréscimo de energia SRAD e CG com 9% ambas, no entanto, em 7 aplicações o consumo de energia é igual ao melhor caso. Para EDP, esta cenário teve um acréscimo de 2% em média, sendo a aplicação que mais contribuiu para este acréscimo foi a BP com 13%. De forma semelhante ao tempo de execução, em 11 aplicações o EDP foi igual ao melhor caso.

KNN em relação ao *Oracle*: em relação ao melhor caso, o método KNN possui para tempo de execução, uma média de 4% de acréscimo para tempo de execução, no qual a aplicação que mais contribui para tal média é a aplicação BP com 13% de aumento no tempo da aplicação, já as aplicações KM e LU possuem aumentos de 11% e 12% no tempo respectivamente. É possível notar que em 5 aplicações o método KNN conseguiu obter tempos iguais ao melhor caso, sendo elas, CG, EP, IS, JA, SRAD. Para a métrica consumo de energia, há um aumento médio no consumo de nergia de apenas 4% em relação ao melhor caso, com maior contribuição para esta média sendo a aplicação BP, com 33%. Para EDP, o KNN possui um aumento médio de apenas 8%.

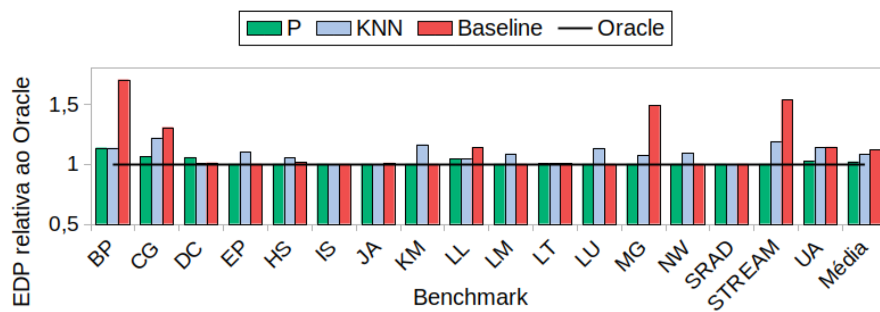
Baseline* em relação ao *Oracle: comparando a execução máxima de *threads* disponível para o ambiente (16) em relação ao melhor caso, este cenário apresentou aumento médio de apenas 2% para a métrica tempo de execução sendo a aplicação BP a que teve maior contribuição para esta média, tendo um aumento de 25%. Porém, para a métrica consumo de energia, este cenário apresentou aumento médio significativo de 11%, sendo as aplicações BP, CG, LL, MG, STREAM e UA, as que mais contribuiram para esta média tendo aumentos de 35%, 33%, 19%, 50%, 45% respectivamente. Para a métrica EDP, a média de aumento é de 12%, tendo aplicações como BP,CG *STREAM*, MG, as aplicações com maior aumento para esta métrica, tendo onde estas valores de 70%, 30%, 48% e 54% respectivamente.



a Tempo de execução



b Consumo de energia



c EDP

Figura 10 – Pequeno, KNN e *Baseline* em relação ao *Oracle*

6 CONCLUSÃO E TRABALHOS FUTUROS

Em programação paralela o principal objetivo é reduzir o tempo de execução de uma aplicação sequencial distribuindo as tarefas da aplicação em diferentes núcleos. Para isto, utiliza-se de forma corriqueira o maior número de *threads* disponível na microarquitetura que se dispõe para executar tal aplicação, ou ainda, executa-se uma aplicação sobre um conjunto de dados muito grande e infere-se números aleatórios de *threads* afim de encontrar o melhor número de *threads* para tal aplicação, porém, isto pode levar a gastos execessivos de tempo e consumo de energia. Desta forma, faz-se necessário a utilização de métodos que possam estimar com o menor erro possível o número de *threads* para uma aplicação que demanda muito tempo para ser executada.

Neste sentido, este trabalho teve como objetivo verificar se uma aplicação paralela que utiliza um volume muito grande de dados para ser processado pode ser predita com base na execução desta mesma aplicação utilizando como treinamento um volume de dados muito pequeno. Para tal, inicialmente foi necessário verificar como esta aplicação se comporta em diferentes tamanhos de entrada, correlacionando o menor tamanho de entrada e o maior através de correlação múltipla, onde foi possível verificar que há fortes correlações entre o menor volume de dados e o maior. Após verificar como as aplicações se comportam em diferentes tamanhos de entrada através de correlações, dois métodos *offline* foram utilizados para prever o número de *threads* ideal, o primeiro utilizou apenas a *thread* que obteve o menor valor entre as métricas e utilizando esta para executar a aplicação com maior conjunto de entrada, o segundo, utilizou os dados obtidos após a execução da aplicação sobre uma entrada pequena e realizado uma regressão sobre estes dados através do método KNN, utilizando a *thread* obtida pelo método para executar a aplicação com maior volume de dados. Estes métodos foram comparados com a execução da aplicação sobre o número máximo de *threads* disponível no ambiente e o número de *threads* ideal para cada métrica, onde esta foi obtida através de uma execução exaustiva da aplicação com diferentes números de *threads*.

Os resultados mostram que executando a aplicação sobre um volume de dados muito pequeno e utilizando a *thread* que possui o menor valor entre as métricas, tempo de execução, consumo de energia e EDP, possui uma redução média de 0,009% para tempo, 9% para energia e 10% para EDP quando comparados com a utilização do número máximo de *threads*, e, aumento médio de 1%, 2% e 1% para tempo, energia e EDP quando comparado ao *Oracle*. Já os resultados obtidos pela regressão mostram uma aumento médio de 2% para tempo e reduções médias de 6% e 4% para energia e EDP quando comparados com a utilização do número máximo de *threads* e, aumentos de de 4%, 5% e 8% em relação ao *Oracle*.

Desta forma, pode-se concluir que, ao se utilizar a execução de uma aplicação paralela sobre pequeno volume de dados para estimar um número de *threads* para a mesma aplicação que irá computar um volume de dados muito maior se demonstrou uma boa

prática na maioria dos *benchmarks* testados, tendo em vista que o tempo de treinamento para encontrar um número de *threads* que tem boas reduções para consumo de energia e EDP é muito pequeno quando compara-se com a busca exaustiva sobre o maior volume de dados, como por exemplo, na aplicação EP, onde a execução da aplicação sobre as 16 *threads* levou-se 7.3 segundos para obter a melhor *thread*, enquanto que, a solução *Oracle* precisou de 1875, neste caso há uma redução no tempo de treinamento de 99.6%. Já o método de regressão, este necessita de mais dados para treinamento para obter resultados melhores, mesmo assim, devido ao tempo de treinamento sobre o menor tamanho, possui erros aceitáveis. Entretanto, podem existir aplicações onde este método não obtenha resultados satisfatórios, principalmente quando o comportamento da aplicação sobre o menor conjunto de dados for muito diferente do tamanho de entrada maior, neste caso, outras abordagens devem ser utilizadas.

Como pretensões futuras de trabalho, visa se utilizar ambientes que possuam microarquiteturas com uma quantidade maior de núcleos, com o objetivo de verificar se os resultados em relação ao maior número de *threads* possam ser melhorados. Utilizar outros métodos de aprendizado de máquina ou ainda melhorar o que foi utilizado, com o intuito de realizar um comparativo entre os métodos e verificar qual destes oferece melhores resultados de tempo de execução, consumo de energia e EDP.

REFERÊNCIAS

ALPAYDIN, E. **Introduction to Machine Learning**. [S.l.: s.n.], 2014. Citado na página 35.

ALVES, M. A. Z. et al. **Influência do compartilhamento de cache l2 em um chip multiprocessado sob cargas de trabalho com conjuntos de dados contíguos e não contíguos.** 2007. Citado na página 21.

ARAUJO, L. R. de et al. **Analise de Desempenho e do Consumo Energético em MPI e OpenMP para Aplicacaçõess do NAS Parallel Benchmarks em uma Arquitetura com Memoria Compartilhada**. 2010. Citado 2 vezes nas páginas 40 e 42.

BAILEY, D. et al. **THE NAS PARALLEL BENCHMARKS**. 1994. Citado na página 43.

BALLADINI, J. et al. **Impact of parallel programming models and cpus clock frequency on energy consumption of hpc systems.** 2011. Citado 2 vezes nas páginas 40 e 42.

BARNES, B. et al. **A regression-based approach to scalability prediction.** 2008. Citado 2 vezes nas páginas 40 e 42.

BURNS, S. **Python Machine Learning**. [S.l.: s.n.], 2019. Citado na página 35.

BUTENHOF, D. R. **Programming with POSIX threads.** 1997. Citado 2 vezes nas páginas 28 e 31.

CAO, Y. et al. **al. Performance analysis of current parallel programming models for many-core systems.** 2013. Citado 2 vezes nas páginas 41 e 42.

CARNEIRO, T. G. de S. **Memória Compartilhada e Distribuída** $_{N}otasdeAula.Citadonapágina28.$

CHAPMAN, B. et al. **Using OpenMP: Portable Shared Memory Parallel Programming**. 2008. Citado na página 28.

CHE, S. et al. **Rodinia: A Benchmark Suite for Heterogeneous Computing**. 2009. Citado 3 vezes nas páginas 22, 36 e 43.

FASIKU, A. I. et al. **Performance Evaluation of Multicore Processors**. 2014. Citado 2 vezes nas páginas 41 e 42.

FOSTER, I. **Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.** 1995. Citado na página 21.

GILAT, A. et al. **Métodos Numéricos para Engenheiros e Cientistas. Uma introdução com aplicações usando MATLAB**. 2008. Citado na página 46.

GRAMA, A. et al. **Introduction to Parallel Computing. 2nd edn.** 2003. Citado na página 31.

GROOP, W. et al. **Using MPI: Portable Parallel Programming with the Message-passing Interface.** 1998. Citado 2 vezes nas páginas 27 e 28.

HACKENBERG, D. **Power measurement techniques on standard compute nodes: A quantitative comparison.** 2013. Citado na página 48.

HUANG, L. et al. Predicting execution time of computer programs using sparse polynomial regression. In: LAFFERTY, J. D. et al. (Ed.). **Advances in Neural Information Processing Systems 23**. Curran Associates, Inc., 2010. p. 883–891. Disponível em: <http://papers.nips.cc/paper/4145-predicting-execution-time-of-computer-programs-using-sparse-polynomial-regression.pdf>. Citado na página 40.

IPEK, E. et al. **An Approach to Performance Prediction for Parallel Applications**. 2005. Citado na página 39.

KARLIN, I.; KEASLER, J.; NEELY, R. **LULESH 2.0 Updates and Changes**. [S.l.], 2013. 1-9 p. Citado 2 vezes nas páginas 43 e 46.

KIST, D. M. **Interfaces para Programação Paralela, uma Alternativa para Anahy1**. 2013. Citado na página 27.

LEE, J. **Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications.** 2010. Citado na página 22.

LORENZON, A. F. **Avaliação do Desempenho e Consumo Energético de Diferentes Interfaces de Programação Paralela em Sistemas Embarcados e de Propósito Geral**. 2014. Citado 9 vezes nas páginas 25, 26, 28, 29, 30, 31, 32, 47 e 48.

LORENZON, A. F. **Aurora, Seamless Optimization of OpenMP Applications**. 2018. Citado 4 vezes nas páginas 22, 32, 41 e 47.

MALLóN, D. A. et al. **Performance evaluation of mpi, upc and openmp on multicore architectures.** 2009. Citado 2 vezes nas páginas 40 e 42.

MARQUES, S. M. V. et al. **Avaliando o Impacto do AMD Turbo Core no Consumo de Energia e Desempenho de Aplicações Paralelas**. 2019. Citado na página 47.

MARSLAND, S. **Machine Learning, An Algorithmic Perspective**. [S.l.: s.n.], 2009. Citado na página 35.

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.

MORAIS, E. F. de. **Paralelismo no Nível de *threads***. 2014. Citado na página 21.

OLUKOTUN, K. et al. **The Future of Microprocessors**. 200. Citado na página 21.

PACHECO, P. **An Introduction to Parallel Programming**. [S.l.: s.n.], 2011. Citado 4 vezes nas páginas 21, 25, 29 e 30.

PASE, D. M.; ECKL, M. A. **A Comparison of Single-Core and Dual-Core Opteron Processor Performance for HPC**. 2005. Citado 2 vezes nas páginas 39 e 42.

PELETTI, F. et al. **Energy-Efficient Multiprocessor System-on-Chip for Embedded Computing: Exploring Programming Models and Their Architectural Support.** 2007. Citado na página 28.

PONTES, A. C. F. **ENSINO DA CORRELAÇÃO DE POSTOS NO ENSINO MÉDIO**. Citado na página 33.

PORTERFIELD, A. K. et al. **Power Measurement and Concurrency Throttling for Energy Reduction in OpenMP Programs**. 2013. Citado na página 41.

SINGH, K. et al. **Predicting parallel application performance via machine learning approaches**. 2007. Citado na página 39.

STALLINGS, W. **COMPUTER ORGANIZATION AND ARCHITECTURE: Designing for Performance**. 2010. Citado na página 21.

TANENBAUM, A. S. **Organização Estruturade de Computadores**. [S.l.: s.n.], 2010. Citado 2 vezes nas páginas 26 e 32.

TAYLOR, V. et al. **Using kernel couplings to predict parallel application performance.** 2002. Citado 2 vezes nas páginas 39 e 42.

TIWARI, A. **Modeling power and energy usage of hpc kernels.** 2012. Citado 2 vezes nas páginas 41 e 42.

VOLPATO, D. G. **Exploração de Diferentes Níveis de Paralelismo Visando a Redução da Área de Processadores Embarcados**. 2014. Citado na página 21.

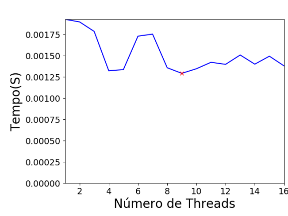
WALL, D. W. **Limits of Instruction-Level Parallelism**. 1991. Citado na página 21.

YANG, L. et al. **Cross-platform performance prediction of parallel applications using partial execution.** 2005. Citado 2 vezes nas páginas 39 e 42.

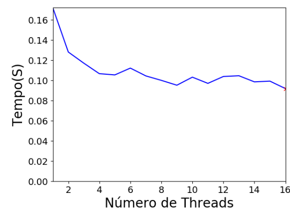
Apêndices

APÊNDICE A – COMPORTAMENTO DAS APLICAÇÕES - TEMPOS

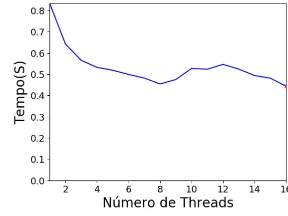
Figura 11 – BP - Tempos



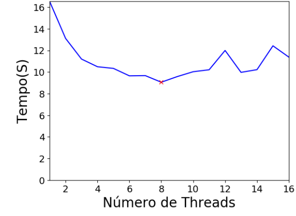
a Entrada Pequena



b Entrada média

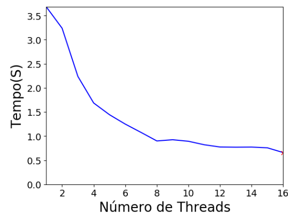


c Entrada Grande

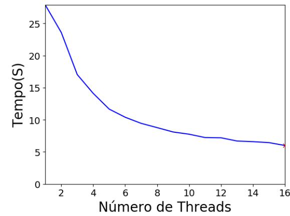


d Entrada Muito grande

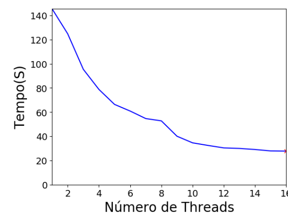
Figura 12 – LU - Tempos



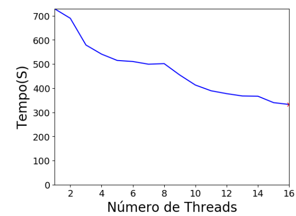
a Entrada Pequena



b Entrada média

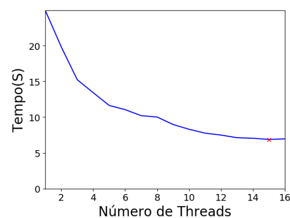


c Entrada Grande

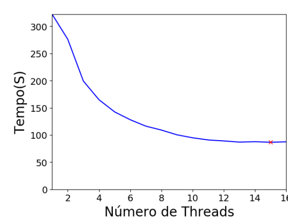


d Entrada Muito grande

Figura 13 – DC - Tempos

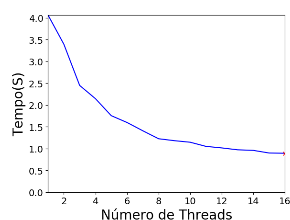


a Entrada Pequena

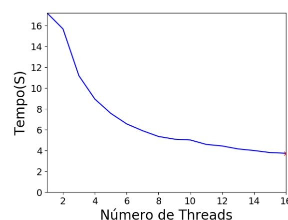


b Entrada Grande

Figura 14 – KM - Tempos

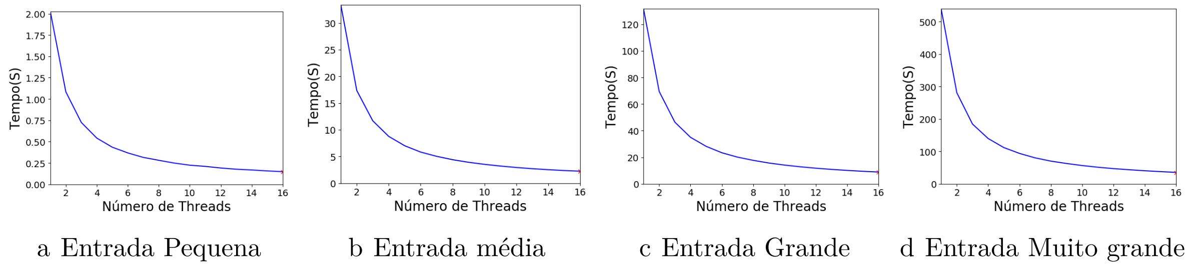


a Entrada Pequena



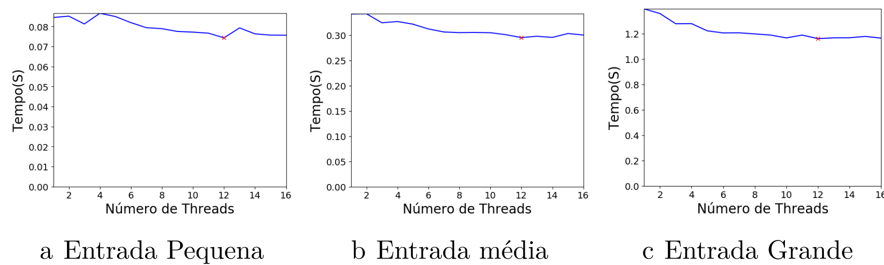
b Entrada Grande

Figura 15 – EP - Tempos



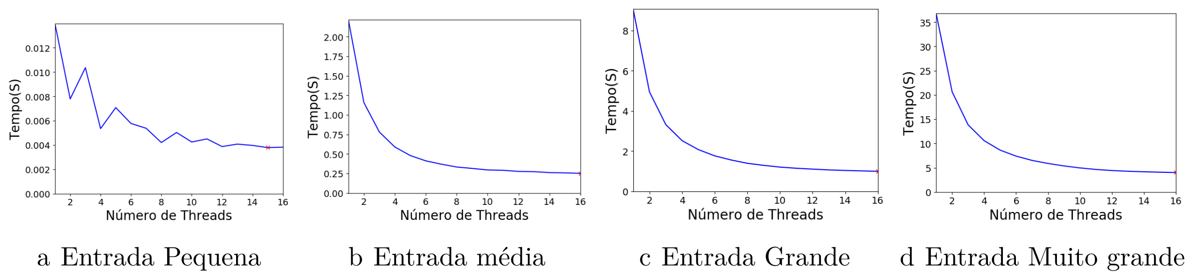
a Entrada Pequena b Entrada média c Entrada Grande d Entrada Muito grande

Figura 16 – HS - Tempos



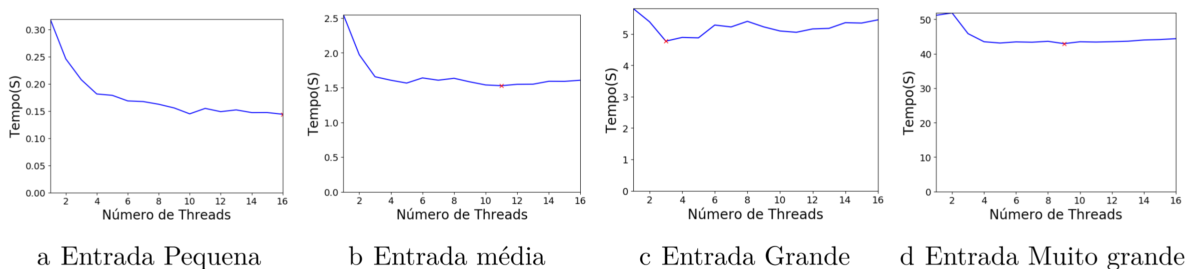
a Entrada Pequena b Entrada média c Entrada Grande

Figura 17 – IS - Tempos



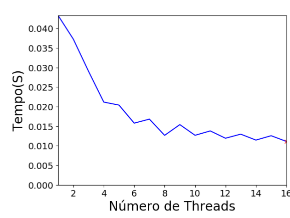
a Entrada Pequena b Entrada média c Entrada Grande d Entrada Muito grande

Figura 18 – MG - Tempos

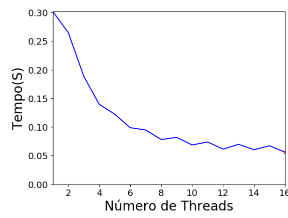


a Entrada Pequena b Entrada média c Entrada Grande d Entrada Muito grande

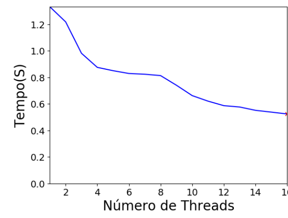
Figura 19 – CG - Tempos



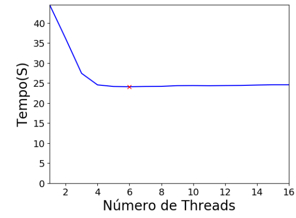
a Entrada Pequena



b Entrada média

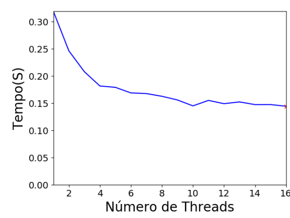


c Entrada Grande

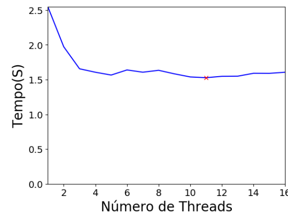


d Entrada Muito Grande

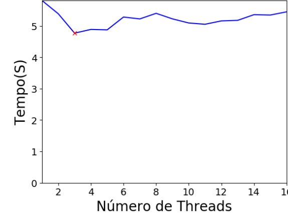
Figura 20 – NW - Tempos



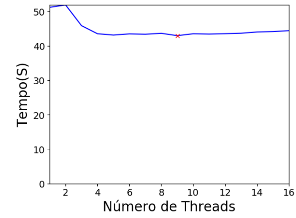
a Entrada Pequena



b Entrada média

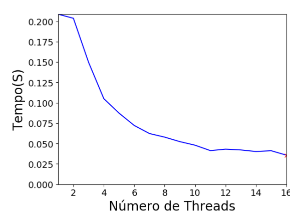


c Entrada Grande

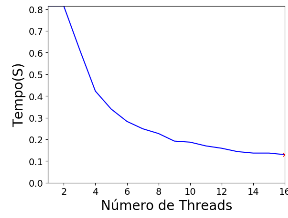


d Entrada Muito grande

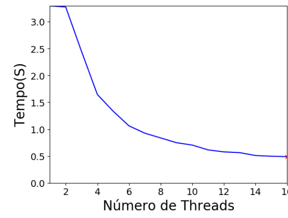
Figura 21 – SRAD - Tempos



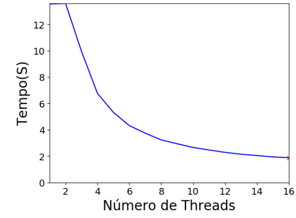
a Entrada Pequena



b Entrada média

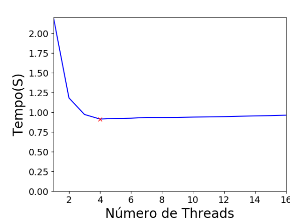


c Entrada Grande

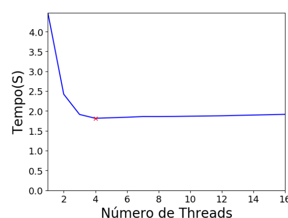


d Entrada Muito grande

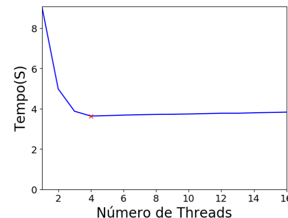
Figura 22 – STREAM - Tempos



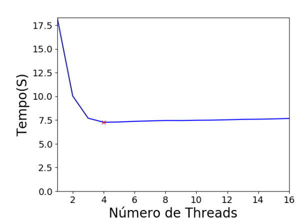
a Entrada Pequena



b Entrada média

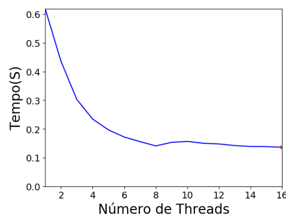


c Entrada Grande

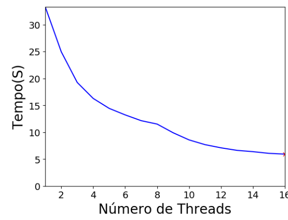


d Entrada Muito grande

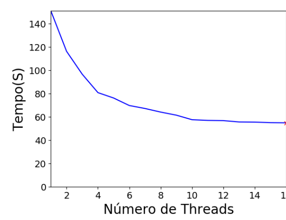
Figura 23 – UA - Tempos



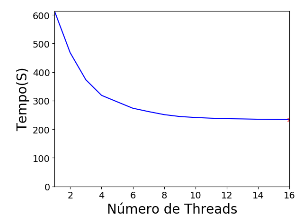
a Entrada Pequena



b Entrada média

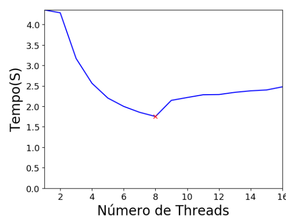


c Entrada Grande

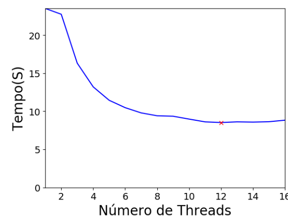


d Entrada Muito grande

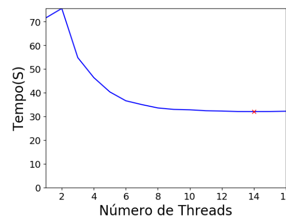
Figura 24 – LL - Tempos



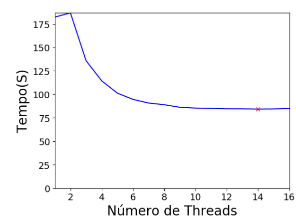
a Entrada Pequena



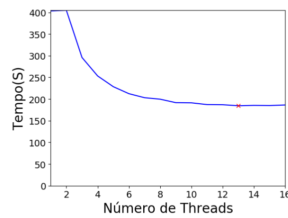
b Entrada média



c Entrada Grande

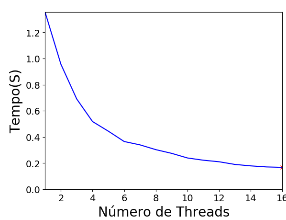


d Entrada Muito grande

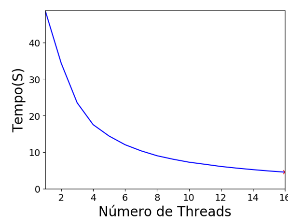


e Entrada Extragrande

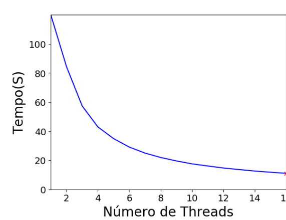
Figura 25 – LM - Tempos



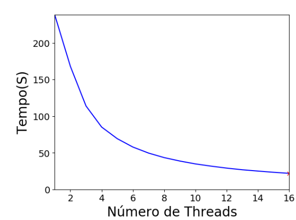
a Entrada Pequena



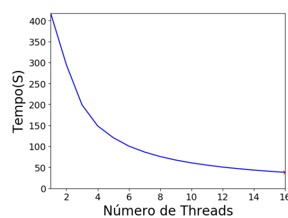
b Entrada média



c Entrada Grande

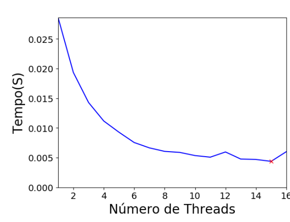


d Entrada Muito grande

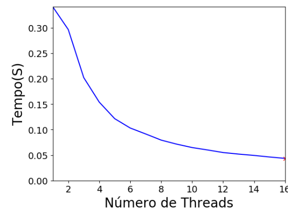


e Entrada Extragrande

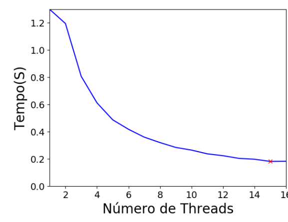
Figura 26 – JA - Tempos



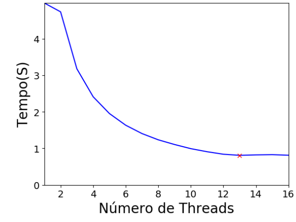
a Entrada Pequena



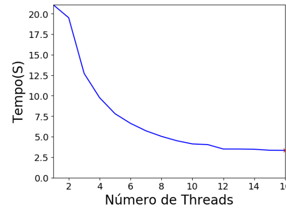
b Entrada média



c Entrada Grande

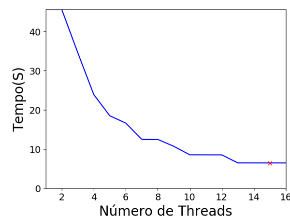


d Entrada Muito grande

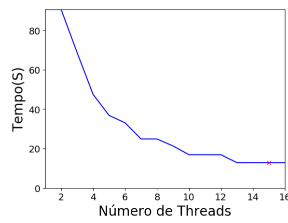


e Entrada Extragrande

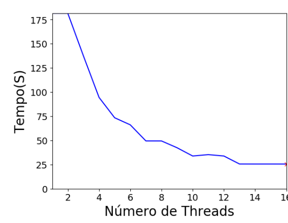
Figura 27 – LT - Tempos



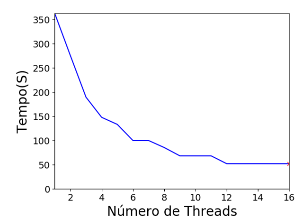
a Entrada Pequena



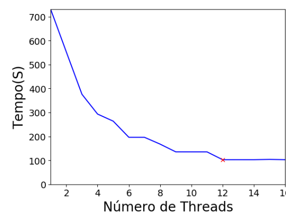
b Entrada média



c Entrada Grande



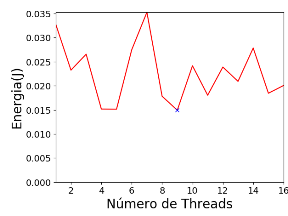
d Entrada Muito grande



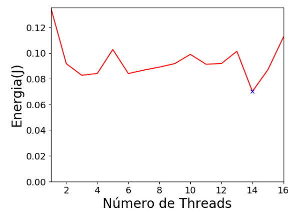
e Entrada Extragrande

APÊNDICE B – COMPORTAMENTO DAS APLICAÇÕES - ENERGIAS

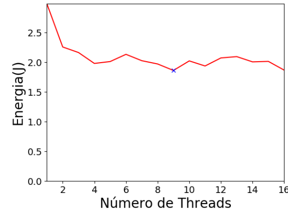
Figura 28 – BP - Energias



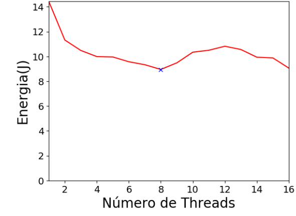
a Entrada Pequena



b Entrada média

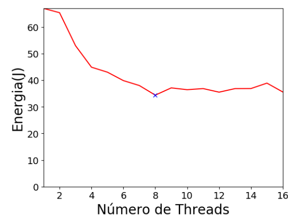


c Entrada Grande

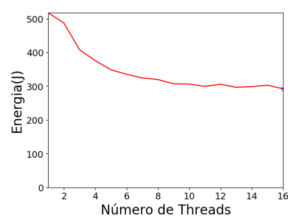


d Entrada Muito grande

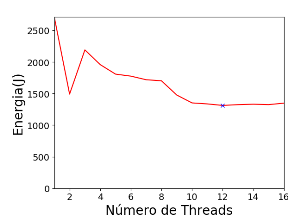
Figura 29 – LU - Energias



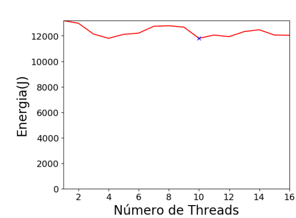
a Entrada Pequena



b Entrada média

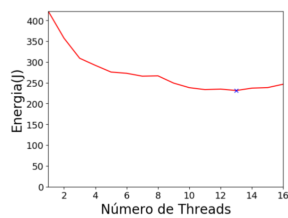


c Entrada Grande

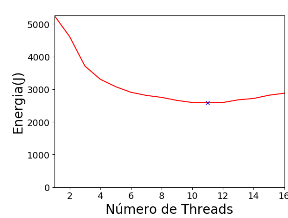


d Entrada Muito grande

Figura 30 – DC - Energias

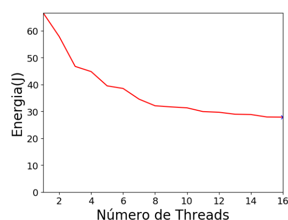


a Entrada Pequena

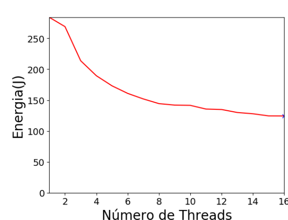


b Entrada Grande

Figura 31 – KM - Energias

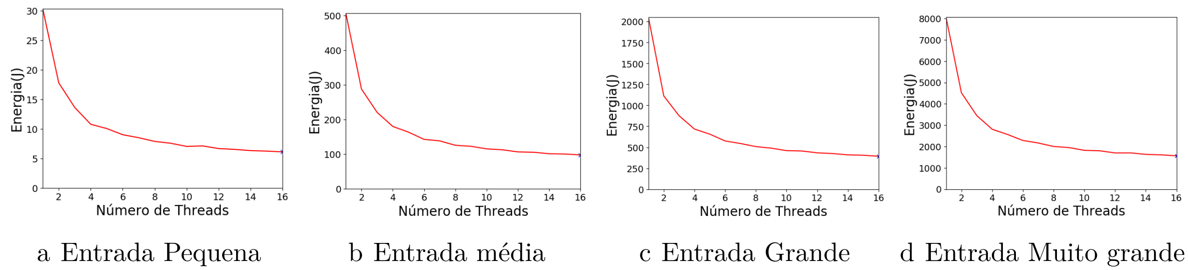


a Entrada Pequena



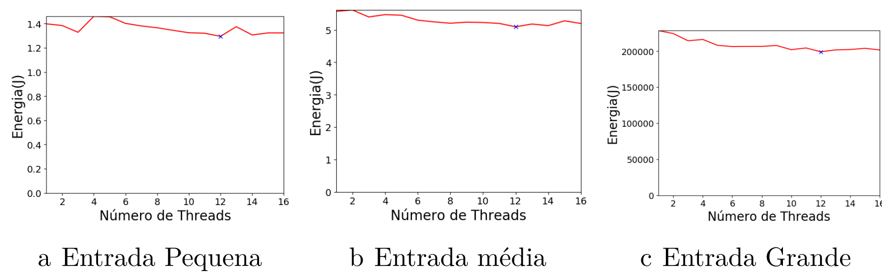
b Entrada Grande

Figura 32 – EP - Energias



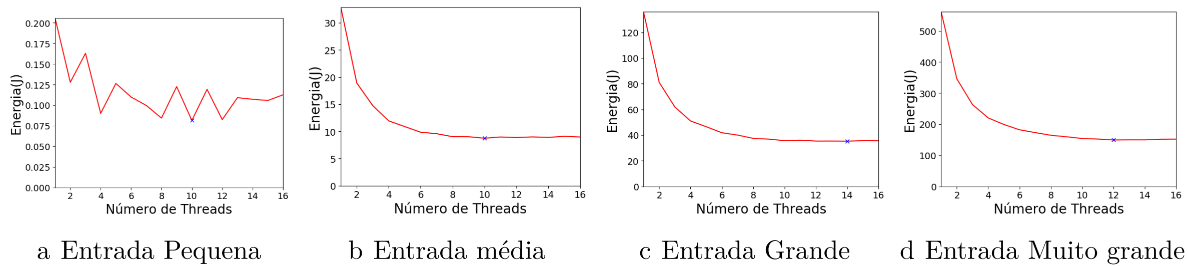
a Entrada Pequena b Entrada média c Entrada Grande d Entrada Muito grande

Figura 33 – HS - Energias



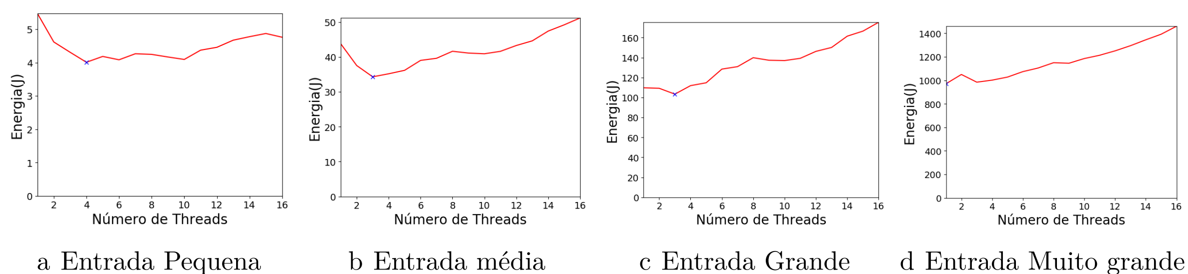
a Entrada Pequena b Entrada média c Entrada Grande

Figura 34 – IS - Energias



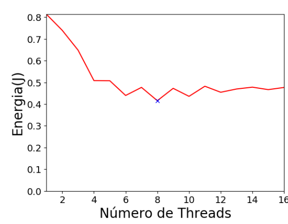
a Entrada Pequena b Entrada média c Entrada Grande d Entrada Muito grande

Figura 35 – MG - Energias

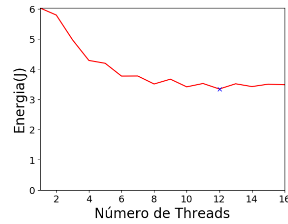


a Entrada Pequena b Entrada média c Entrada Grande d Entrada Muito grande

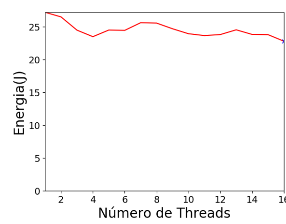
Figura 36 – CG - Energias



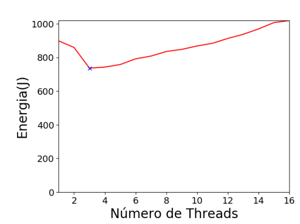
a Entrada Pequena



b Entrada média

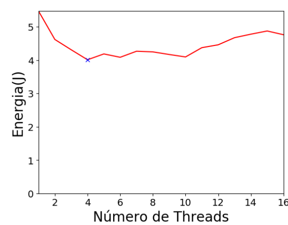


c Entrada Grande

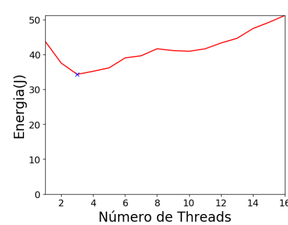


d Entrada Muito Grande

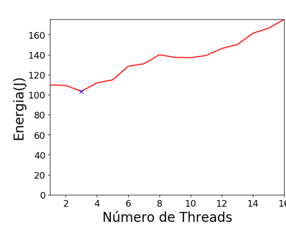
Figura 37 – NW - Energias



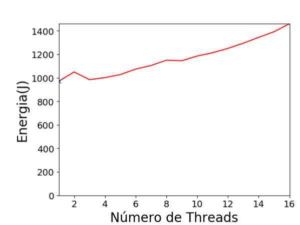
a Entrada Pequena



b Entrada média

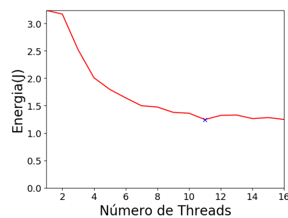


c Entrada Grande

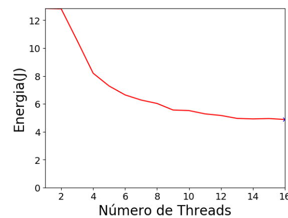


d Entrada Muito grande

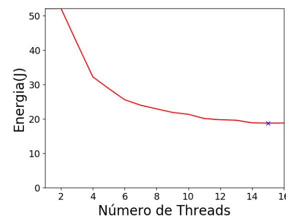
Figura 38 – SRAD - Energias



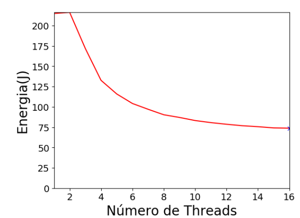
a Entrada Pequena



b Entrada média

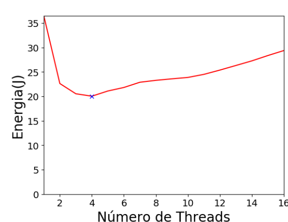


c Entrada Grande

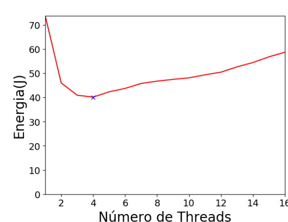


d Entrada Muito grande

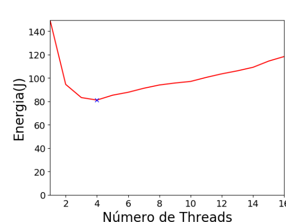
Figura 39 – STREAM - Energias



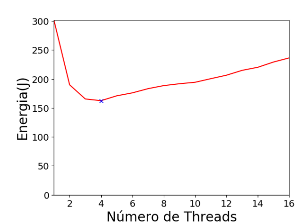
a Entrada Pequena



b Entrada média

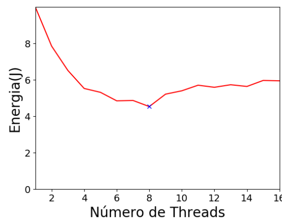


c Entrada Grande

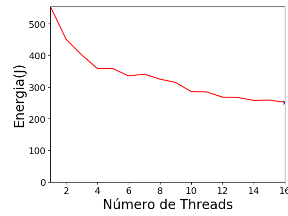


d Entrada Muito grande

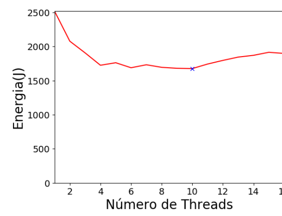
Figura 40 – UA - Energias



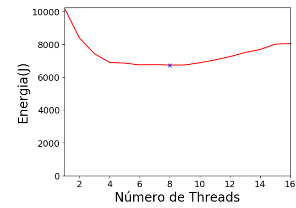
a Entrada Pequena



b Entrada média

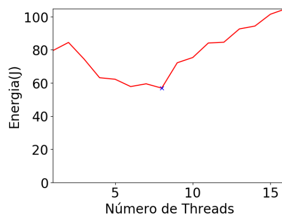


c Entrada Grande

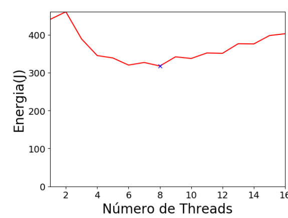


d Entrada Muito grande

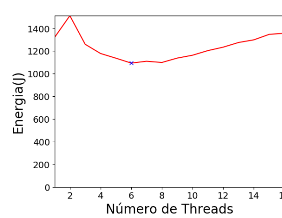
Figura 41 – LL - Energias



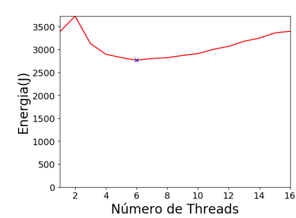
a Entrada Pequena



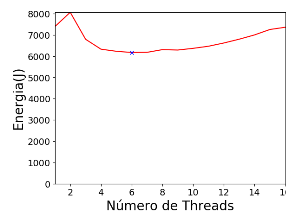
b Entrada média



c Entrada Grande

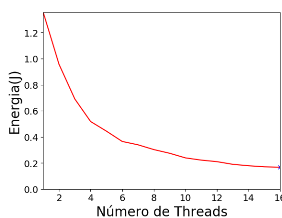


d Entrada Muito grande

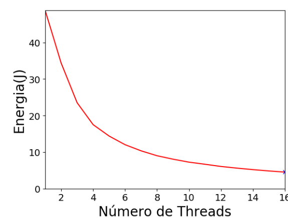


e Entrada Extragrande

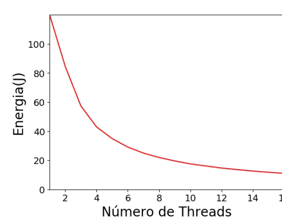
Figura 42 – LM - Energias



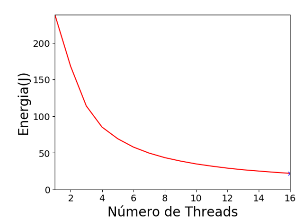
a Entrada Pequena



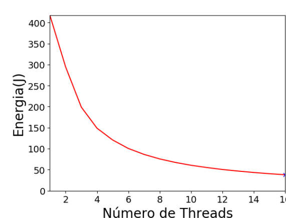
b Entrada média



c Entrada Grande

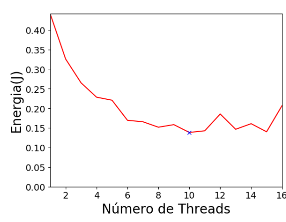


d Entrada Muito grande

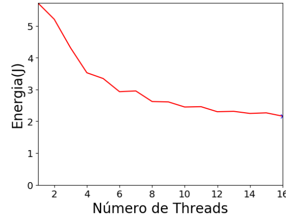


e Entrada Extragrande

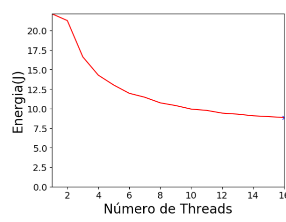
Figura 43 – JA - Energias



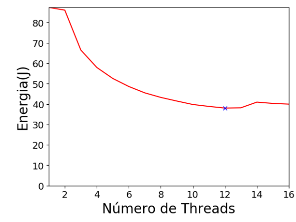
a Entrada Pequena



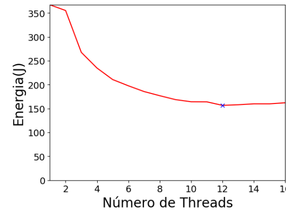
b Entrada média



c Entrada Grande

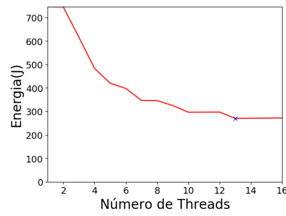


d Entrada Muito grande

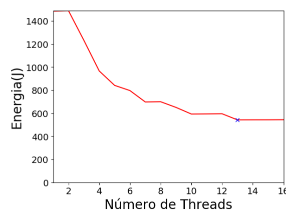


e Entrada Extragrande

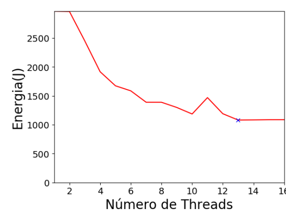
Figura 44 – LT - Energias



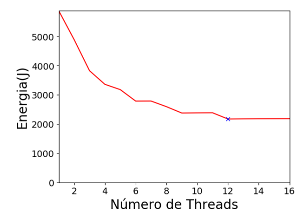
a Entrada Pequena



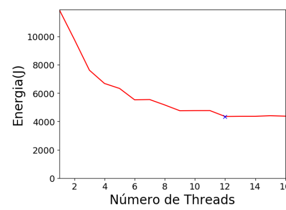
b Entrada média



c Entrada Grande



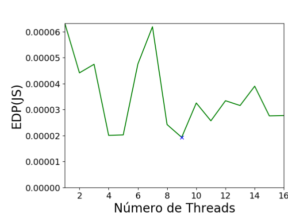
d Entrada Muito grande



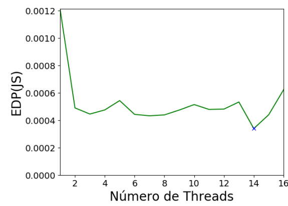
e Entrada Extragrande

APÊNDICE C – COMPORTAMENTO DAS APLICAÇÕES - EDP

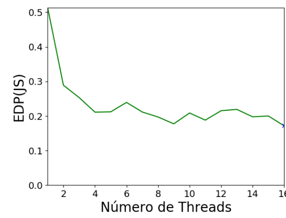
Figura 45 – BP - EDP



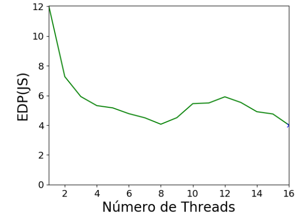
a Entrada Pequena



b Entrada média

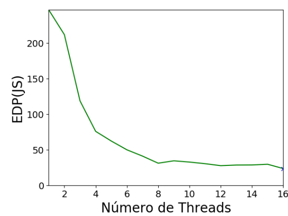


c Entrada Grande

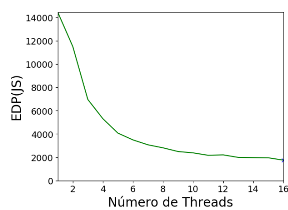


d Entrada Muito grande

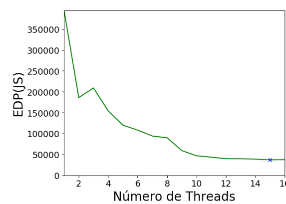
Figura 46 – LU - EDP



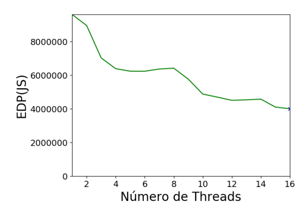
a Entrada Pequena



b Entrada média

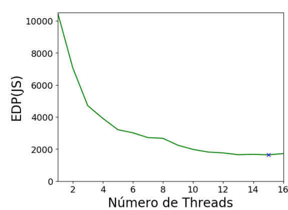


c Entrada Grande

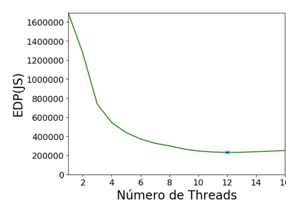


d Entrada Muito grande

Figura 47 – DC - EDP

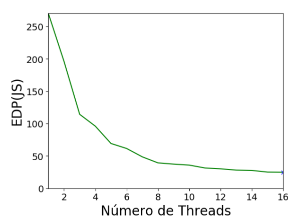


a Entrada Pequena

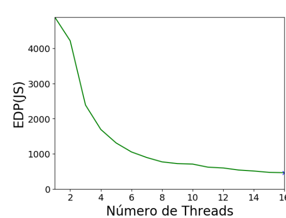


b Entrada Grande

Figura 48 – KM - EDP

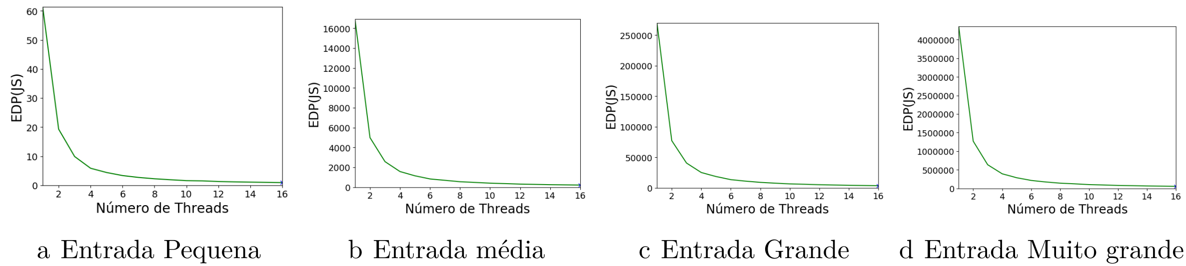


a Entrada Pequena



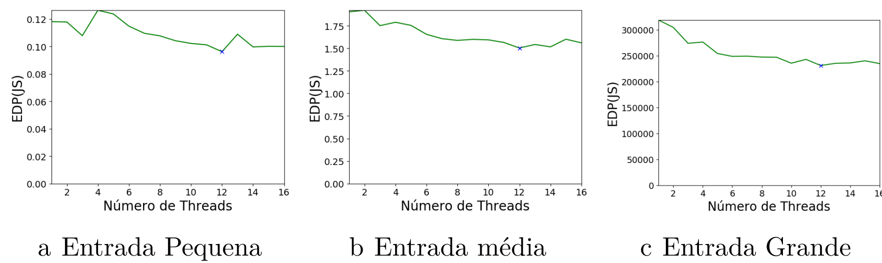
b Entrada Grande

Figura 49 – EP - EDP



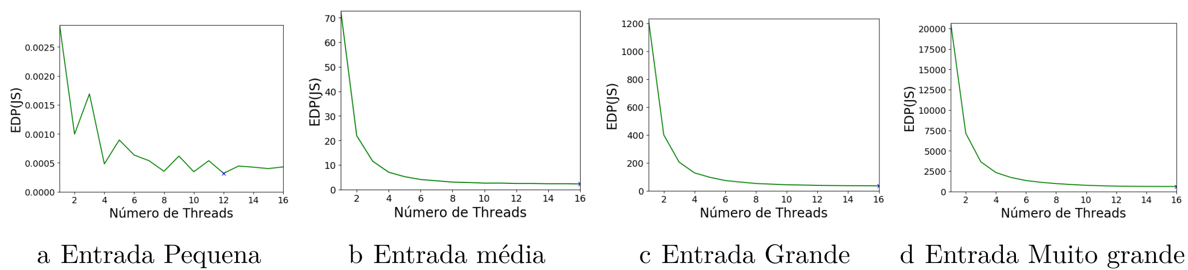
a Entrada Pequena b Entrada média c Entrada Grande d Entrada Muito grande

Figura 50 – HS - EDP



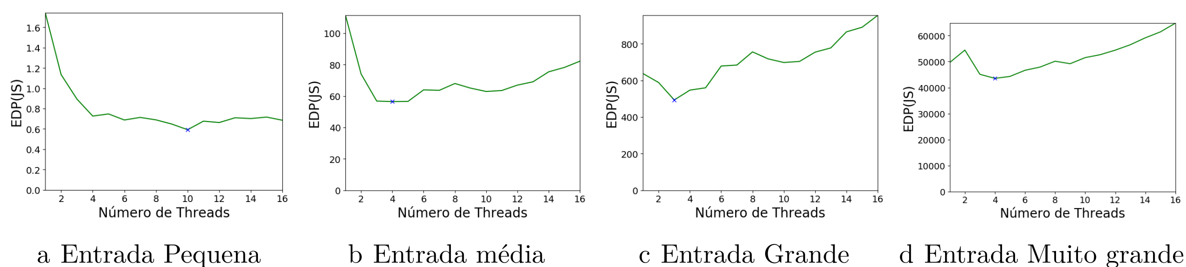
a Entrada Pequena b Entrada média c Entrada Grande

Figura 51 – IS - EDP



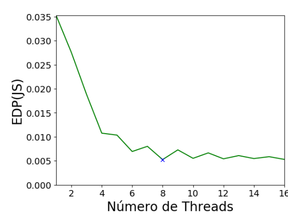
a Entrada Pequena b Entrada média c Entrada Grande d Entrada Muito grande

Figura 52 – MG - EDP

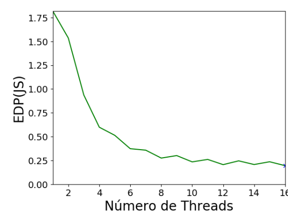


a Entrada Pequena b Entrada média c Entrada Grande d Entrada Muito grande

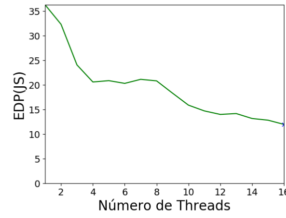
Figura 53 – CG - EDP



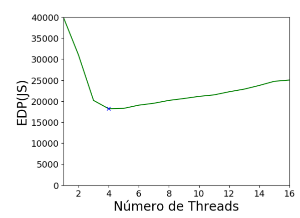
a Entrada Pequena



b Entrada média

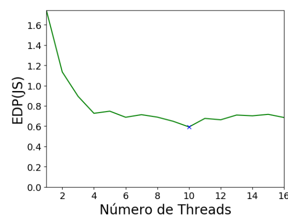


c Entrada Grande

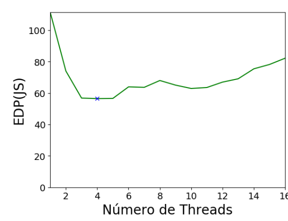


d Entrada Muito Grande

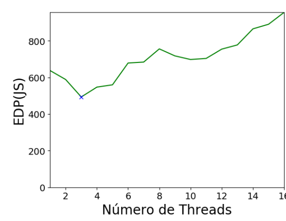
Figura 54 – NW - EDP



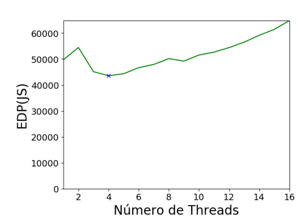
a Entrada Pequena



b Entrada média

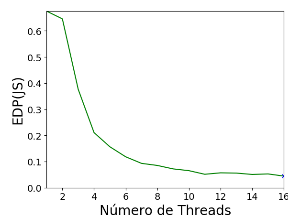


c Entrada Grande

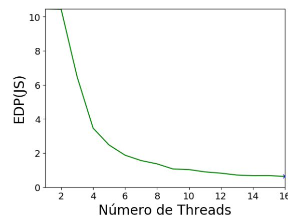


d Entrada Muito grande

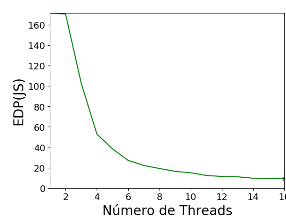
Figura 55 – SRAD - EDP



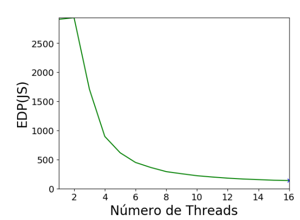
a Entrada Pequena



b Entrada média

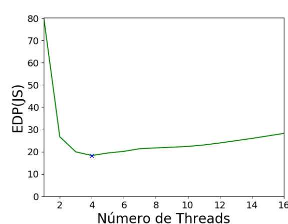


c Entrada Grande

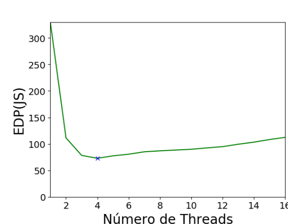


d Entrada Muito grande

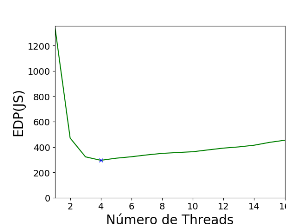
Figura 56 – STREAM - EDP



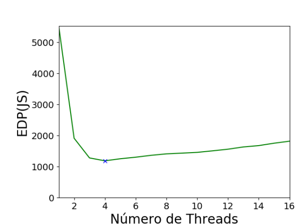
a Entrada Pequena



b Entrada média

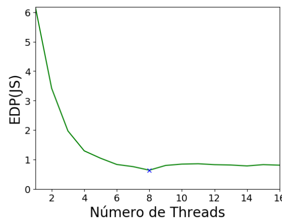


c Entrada Grande

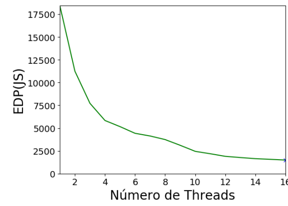


d Entrada Muito grande

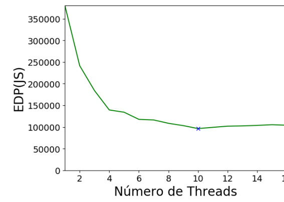
Figura 57 – UA - EDP



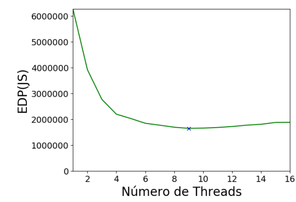
a Entrada Pequena



b Entrada média

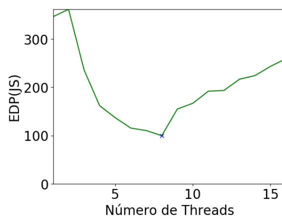


c Entrada Grande

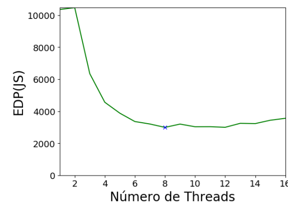


d Entrada Muito grande

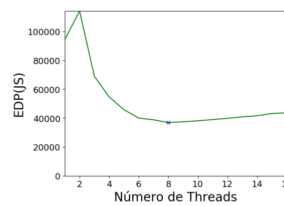
Figura 58 – LL - EDP



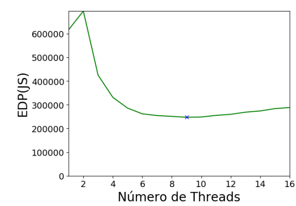
a Entrada Pequena



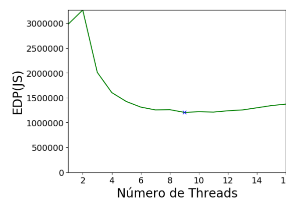
b Entrada média



c Entrada Grande

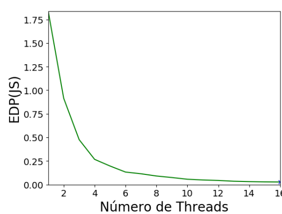


d Entrada Muito grande

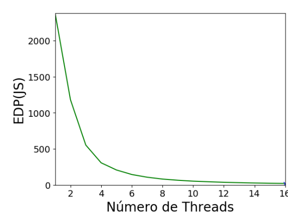


e Entrada Extragrande

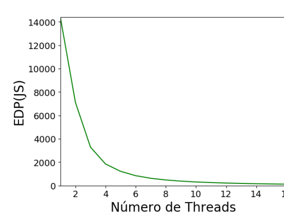
Figura 59 – LM - EDP



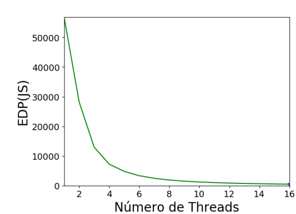
a Entrada Pequena



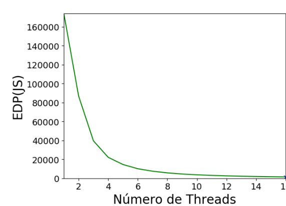
b Entrada média



c Entrada Grande

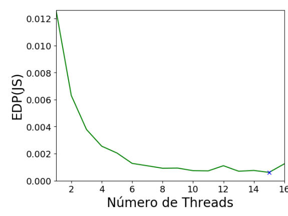


d Entrada Muito grande

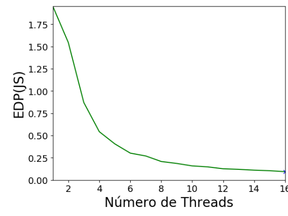


e Entrada Extragrande

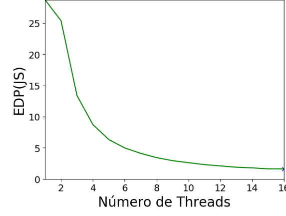
Figura 60 – JA - EDP



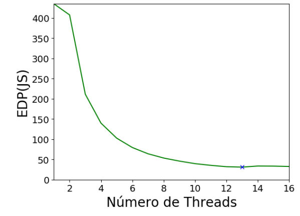
a Entrada Pequena



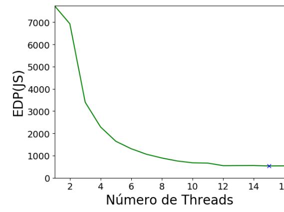
b Entrada média



c Entrada Grande

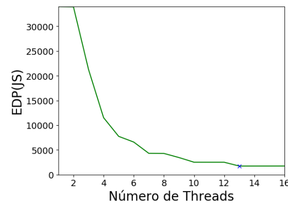


d Entrada Muito grande

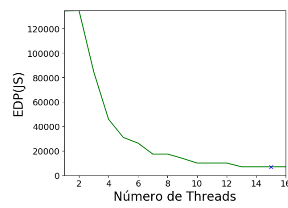


e Entrada Extragrande

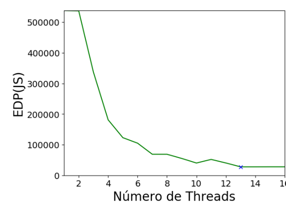
Figura 61 – LT - EDP



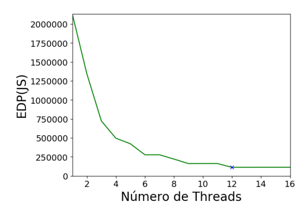
a Entrada Pequena



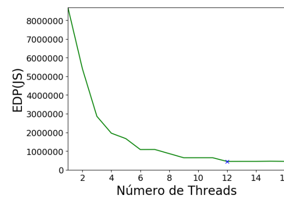
b Entrada média



c Entrada Grande



d Entrada Muito grande



e Entrada Extragrande

ÍNDICE

CPU, 29, 40, 41, 44, 46, 47

DSE, 24, 25, 32
DVFS, 46

EDP, 9, 19, 23, 24, 32, 42, 47–49, 51–54, 57, 58

GCC, 41
GPU, 44

ICC, 41
ILP, 21, 25

KNN, 19, 35, 36, 43, 49–54, 57

MPI, 28

NAS, 15, 19, 39, 40, 43–45

SMT, 46

TLP, 21, 25, 26