Otimizando a Execução de Aplicações Paralelas em Ambiente de Nuvem Heterogênea*

Everton C. de Lima[1], Marcelo C. Luizelli[1], Fábio Rossi[2], Antonio Carlos S. Beck[3] e Arthur F. Lorenzon[1]

[1]Universidade Federal do Pampa – Alegrete – RS – Brasil

[2]Instituto Federal Farroupilha – Campus Alegrete – RS – Brasil

[3]Universidade Federal do Rio Grande do Sul – Porto Alegre – RS – Brasil

evertonlima.aluno@unipampa.edu.br

Resumo. *A computação na nuvem emerge como uma plataforma alternativa para a execução de aplicações de alto desempenho. Simultaneamente, a atualização de nodos computacionais nestes sistemas pode levar a uma heterogeneidade de recursos. Neste sentido, o desafio de executar aplicações paralelas na nuvem não está apenas relacionado a definição do melhor número de threads para a aplicação, mas também, a escolha ideal da arquitetura que irá executar tal aplicação. No entanto, as características de grau de paralelismo e capacidade computacional têm sido pouco exploradas para fazer a alocação de aplicações numa nuvem heterogênea. Portanto, neste artigo, mostramos que ao considerar o grau de paralelismo de uma aplicação e as características do nodo computacional, ganhos significativos de desempenho e consumo de energia podem ser obtidos quando comparado a maneira padrão com que aplicações são escalonadas num ambiente de nuvem heterogênea.*

1. Introdução

A computação em nuvem tem emergido como uma plataforma alternativa já consolidada para a execução de aplicações de alto desempenho de diferentes domínios, por exemplo, aprendizado de máquina e algoritmos de álgebra linear básica. Embora estes sistemas forneçam elasticidade (provisionamento sob demanda), customização e controle de recursos, eles são essencialmente *data centers* com uma alta exigência de energia para manter a operação [Masanet et al. 2020]. Portanto, ao executar tais aplicações paralelas na nuvem, é importante não apenas otimizar o tempo de execução, mas fazer isto com o menor impacto possível no consumo de energia para manter baixo os custos de operação.

No ímpeto pela melhoria de desempenho, a exploração do paralelismo no nível de *threads* (TLP - *thread-level parallelism*) tem sido amplamente utilizada pelos desenvolvedores de *software*. Nela, a carga de trabalho é dividida e executada de maneira concorrente por múltiplas unidades funcionais [da Silva et al. 2021]. A maneira comum adotada por usuários e desenvolvedores consiste no uso de todos os recursos computacionais disponíveis na arquitetura (*e.g.*, núcleos e memória). No entanto, uma vez que muitas aplicações paralelas têm sua escalabilidade limitada por questões relacionadas à

*Este trabalho foi parcialmente financiado pela FAPERGS nos projetos 19/2551-0001224-1 e 19/2551-0001689-1

software e/ou *hardware*, atribuir todos os recursos para tais aplicações não irá necessariamente resultar no melhor desempenho e consumo de energia [Suleman et al. 2008]. Neste sentido, definir um número ideal de *threads* para executar aplicações paralelas pode levar a uma redução significativa no tempo de execução das aplicações, diminuindo os custos relacionados a energia e alocação de recursos computacionais [Marques et al. 2021].

Adicionalmente, durante o ciclo de vida, os *clusters* de alto desempenho encontrados na nuvem podem ser atualizados com nodos computacionais que usam novas tecnologias e tipicamente possuem velocidade computacional mais rápida que os nodos originais. Esta atualização de recursos computacionais resulta em um *cluster* heterogêneo, onde cada nodo será capaz de entregar desempenho diferente de acordo com a aplicação alvo. Esta heterogeneidade no nível de recursos resulta em novos desafios relacionados a execução de aplicações paralelas na nuvem: além de definir o melhor número de *threads* no nível da aplicação, também é importante alocar esta aplicação para executar no nodo computacional que forneça os melhores resultados de desempenho e consumo de energia.

Neste contexto, diferentes estratégias têm sido propostas para otimizar a execução de aplicações em um ambiente de nuvem computacional. RLSK (Reinforcement Learning Scheduler Kubernetes) é uma estratégia baseada em aprendizado por reforço para escalonar tarefas independentes entre múltiplos *clusters* de maneira adaptável [Huang et al. 2020]. De maneira similar, [Orhean et al. 2018] propõem um algoritmo de aprendizado por reforço para determinar a melhor política de escalonamento em sistemas distribuídos considerando a heterogeneidade dos nós. [Charr et al. 2015] propõem um algoritmo para otimizar o custo-benefício entre desempenho e consumo de energia através da escolha da melhor frequência de operação da CPU em ambiente heterogêneo. No entanto, no melhor do nosso conhecimento e conforme também destacado na Seção 2.3, este é o primeiro trabalho que propõe otimizar aplicações paralelas em ambiente de nuvem computacional heterogênea via a melhor combinação de grau de paralelismo de uma aplicação e poder computacional do nodo.

Considerando a discussão acima e o espaço de exploração que ainda não foi coberto pela área, este trabalho realiza o uso das características do grau de paralelismo disponível na aplicação para fazer a sua distribuição para os nodos computacionais em um ambiente de nuvem heterogêneo. Assim, as contribuições deste artigo são: (*i*) Análise da escalabilidade de aplicações paralelas em nodos com diferentes capacidades de poder computacional; (*ii*) Análise do impacto no desempenho, consumo de energia e no custo-benefício entre estes dois quando aplicações paralelas com diferentes graus de paralelismo são executadas em um ambiente com arquiteturas heterogêneas. (*iii*) Comparação de uma estratégia de alocação de aplicação para arquitetura alvo baseada no grau de paralelismo da aplicação e características da arquitetura com o estado da arte. Pela execução de vinte aplicações paralelas de diferentes domínios em um ambiente de nuvem com três arquiteturas *multicore* distintas (AMD-16, AMD-24 e AMD-64), cada uma capaz de fornecer diferente capacidade computacional, nós mostramos que:

- A mesma aplicação paralela pode ter comportamento diferente de acordo com o nodo computacional na qual ela está sendo executada. Por exemplo, a aplicação CG do NAS *Parallel Benchmark* escala de maneira ideal em uma arquitetura de 16 unidades de processamento (AMD-16). No entanto, devido às características de organização de memória, esta aplicação não escala bem em uma arquitetura de

64 unidades de processamento (AMD-64).
- Por meio da combinação ideal de aplicação paralela e arquitetura alvo, o custo-benefício entre desempenho e energia de uma aplicação pode ser melhorado em até 97.4 $\times$.
- Considerar o grau de TLP de uma aplicação e a capacidade computacional da arquitetura alvo na hora de executar tal aplicação em um ambiente heterogêneo apresenta ganhos significativos quando comparado ao escalonador padrão encontrado no *Kubernetes*: 43% de melhoria no desempenho e 36% de redução no consumo de energia.

O restante do artigo está organizado como segue. Na Seção 2, a fundamentação teórica e os trabalhos relacionados são discutidos. Então, a metodologia é apresentada na Seção 3. Os resultados experimentais são discutidos na Seção 4. Por fim, a conclusão e trabalhos futuros são destacados na Seção 5.

2. Fundamentação Teórica

2.1. Escalabilidade de Aplicações Paralelas

A prática comum adotada por desenvolvedores de *software* em HPC e computação em nuvem é executar aplicações paralelas usando todos os recursos disponíveis. No entanto, diferentes trabalhos têm mostrado que essa abordagem não necessariamente fornecerá o melhor desempenho, economia de energia e resultado de EDP(*Energy-Delay Product*) [Suleman et al. 2008, Lorenzon and Beck Filho 2019]. As causas estão relacionadas tanto a questões de *software* e *hardware*, e as principais são discutidas a seguir.

I. *Saturação de barramento de comunicação entre processador e memória principal*. Para aplicações que operam sob enorme quantidade de dados privados à cada *thread* que devem ser constantemente buscados da memória principal, o barramento *off-chip* desempenha um papel decisivo na escalabilidade da aplicação paralela [Schwarzrock et al. 2020]. Uma vez que cada *thread* opera sobre blocos de dados diferentes, a demanda pelo barramento de comunicação aumenta linearmente com o número de *threads*. No entanto, a largura de banda do barramento não aumenta em relação ao número de *threads*, pois é limitada pelo número de pinos de I/O [Ham et al. 2013]. Portanto, quando o barramento de comunicação satura, nenhuma melhoria de desempenho é obtida se aumentar o número de *threads* ativas.

II. *Sobrecarga de sincronização de dados entre threads*. Em aplicações paralelas, seções críticas são implementadas para garantir a integridade dos dados durante a execução. Assim, apenas uma única *thread* executará esta região crítica de cada vez, serializando uma parte da execução. Neste sentido, quando o número de *threads* aumenta, mais *threads* devem ser serializadas dentro das seções críticas, influenciando negativamente no tempo total gasto para sincronização.

III. *Número de acessos concorrentes à memória compartilhada*. Em arquiteturas de memória compartilhada, a comunicação entre as *threads* ocorre através de acesso a dados localizados em regiões compartilhadas. Uma vez que estas regiões estão mais distantes do processador (*e.g.*, último nível da memória *cache* e memória principal), esta comunicação pode se tornar um gargalo do desempenho. Além disso, este cenário também pode ser afetado por outros fatores relacionados à exploração do TLP, por exemplo, afinidade de *threads* e dados.

2.2. Computação de Alto Desempenho na Nuvem

A Computação em Nuvem foi estabelecida ao longo dos anos como um *padrão de fato* para a execução de aplicações devido à sua capacidade de provisionar recursos sob demanda pela Internet [Liu et al. 2012]. Embora o provisionamento de recursos em ambientes de nuvem seja transparente para os clientes, nos bastidores, distintas tecnologias trabalham em conjunto para oferecer suporte a funcionalidades importantes, como elasticidade e alta disponibilidade [Márquez et al. 2018]. De uma perspectiva de infraestrutura, os processadores multi-core aprimoraram a capacidade de processamento e o ressurgimento da virtualização proporcionou maior flexibilidade de provisionamento para atender às demandas dos clientes. Do ponto de vista do *software*, arquiteturas distribuídas, como microsserviços, permitiram um melhor uso dos recursos, dando origem a aplicações projetadas para obter o melhor da nuvem [Márquez et al. 2018].

Apesar de seus benefícios, as primeiras implantações de nuvem não conseguiam atender à demanda de aplicações de computação intensiva que exigem respostas rápidas (por exemplo, *Big Data* e *Analytics*) devido à sobrecarga intrínseca gerada pelos *hypervisores* [Barham et al. 2003]. Como resultado, tecnologias de contêineres leves, como *Docker*, começaram a ganhar terreno em ambientes de nuvem, oferecendo níveis de desempenho muito próximos aos ambientes não virtualizados. *Docker* é considerado o *framework* mais popular para construir, empacotar aplicações em imagens e executar aplicações dentro de contêineres. Esses contêineres contêm todos os dados necessários (por exemplo, bibliotecas e binários). No entanto, ele não oferece suporte a recursos de gerenciamento mais avançados, como balanceamento de carga, autorrecuperação e elasticidade automática.

Assim, plataformas de orquestração como *Kubernetes* [Thurgood and Lennon 2019] tornaram-se padrão em ambientes de nuvem ao fornecer recursos que abrangem todo o ciclo de vida da aplicação. Ela oferece diferentes ferramentas de escalonamento para se adaptar a ambiente de execução heterogêneos, exemplo, o *Kube-scheduler* que é o escalonador padrão do *Kubernetes*. Ele consiste de um processo de plano de controle responsável por verificar quando novos *pods* contendo *containers* são criados e selecionar o nó computacional que irá processar a carga de trabalho baseado em uma estratégia de *round-robin*. Embora diferentes escalonadores podem ser usados para realizar tal tarefa, neste trabalho nós consideramos o *kube-scheduler* por ser a implementação referência.

2.3. Trabalhos Relacionados

Os seguintes trabalhos otimizam o desempenho e o consumo de energia de aplicações paralelas para ambiente de arquitetura heterogênea. [Chen et al. 2012] utilizam uma abordagem de programação dinâmica para minimizar o consumo de energia em servidores heterogêneos. [Takouna et al. 2012] propõem uma solução para otimizar o custo-benefício entre energia e desempenho através da técnica de *dynamic voltage frequency scaling* (DVFS). Similarmente, [Charr et al. 2015] propõem um algoritmo *online* para selecionar a frequência da CPU em plataformas heterogêneas que otimiza o custo-benefício entre economia de energia e desempenho.

[Park and Abraham 2011] propõem uma estratégia para melhorar o *energy-delay product* (EDP) através de DVS (*dynamic voltage scaling*) explorando as margens de

temporização. [Li and Martinez 2005] realizam um estudo analítico que considera diferentes *knobs*, por exemplo, granularidade de paralelismo e nível de tensão/frequência da CPU, para otimizar o desempenho, energia e EDP de aplicações paralelas executadas em um processador *multicore*. [Makrani et al. 2018] propõem um provisionamento proativo de recursos *online* para atribuir uma configuração de hardware adequada para otimizar o desempenho e eficiência energética de cargas de trabalho na nuvem. [Khaleghzadeh et al. 2021] propõem um algoritmo para otimizar o tempo de execução e energia de aplicações paralelas que executam em plataformas heterogêneas de alto desempenho. [Maghsoud et al. 2021] apresentam uma técnica para otimizar o EDP de aplicações paralelas. Para tanto, um modelo de predição é construído para determinar o número de núcleos e frequência de CPU ideal do processador. Adicionalmente, o trabalho apresenta uma técnica de escalonamento baseada no *work-stealing*.

3. Metodologia

Nesta seção, apresentamos as aplicações utilizadas nos experimentos bem como as características do ambiente de execução e como a avaliação foi realizada.

3.1. Conjunto de Aplicações

Nós consideramos a execução de vinte aplicações já paralelizadas e escritas em Linguagem C/C++ de diferentes suítes de *benchmarks*: **Oito kernels e pseudo-aplicações** do *NAS Parallel Benchmarks* [Bailey et al. 1991]: BT, CG, EP, FT, LU, MG, SP, and UA. **Três aplicações** do *Rodinia Benchmark Suite* [Che et al. 2009]: *Hotspot* (HS), *LU Decomposition* (LUD), and *Streamcluster* (SC). **Três aplicações** da suíte do *Parboil Benchmarks* [Stratton et al. 2012]: *Breadth-First Search* (BFS), *Lattice-Boltzmann Method Fluid Dynamics* (LBM), and *Magnetic Resonance Imaging - Gridding* (MRI). **Seis aplicações** de diferentes domínios: *fast Fourier transform* (FFT), the *high performance conjugate gradient* benchmark (HPCG) [Dongarra et al. 2015], Método iterativo de *Jacobi* (JA), *n-body* (NB), *Poisson* (PO), and *STREAM* (ST). As aplicações foram executadas com a entrada padrão definida em cada suíte de *benchmarks*.

3.2. Ambiente de Execução

Os experimentos foram realizados em ambiente de nuvem heterogêneo composto de quatro arquiteturas multicores, conforme mostrado na Figura 1: Um nodo *master*, responsável por encaminhar as aplicações para execução em um dos três nodos trabalhadores (AMD-16, AMD-24 e AMD-64). A frequência de operação de cada CPU foi configurada para ajustar de acordo com a carga de trabalho aplicação, através do *governor* DVFS *ondemand*, que é o *governor* padrão usado na maioria das versões Linux. Cada nodo estava usando o Sistema Operacional Linux Ubuntu com kernel v. 5.13.0, *kubernetes* v. 1.23.6 com o *Docker* 20.10.17, *build* 100c701. Cada aplicação foi compilada com GCC/G++ 10.2, usando a flag de otimização $-O3$. A alocação e afinidade de cada *thread* foi configurada através da variável de ambiente do OpenMP: `OMP_PROC_BIND=CLOSE` e `OMP_PLACES=CORES`. Isto faz com que as *threads* de uma mesma aplicação sejam alocadas próximas uma das outras.

As aplicações foram executadas em lote na arquitetura em questão. Para executar cada aplicação em um nodo trabalhador, a seguinte metodologia foi utilizada. As aplicações foram encapsuladas dentro de um *container* através da ferramenta *Docker*,

contendo também as bibliotecas e binários necessários para a execução da mesma. Então, para cada aplicação, foi criado um arquivo de configuração (*Kubernetes Manifest*) do *pod* para descrever como o *container* deveria executar em cada nodo computacional (*e.g.*, grau de TLP). Com este arquivo, o seguinte comando foi utilizado para despachar o pod para execução: "*kubectl apply -f arquivo.yaml*", onde o último parâmetro representa o arquivo de configuração do *pod*.

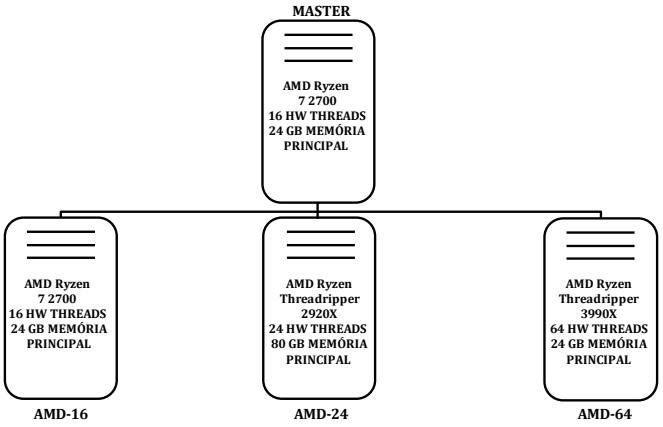


Figura 1. Ambiente utilizado nos experimentos

Na análise apresentada na próxima seção, nós consideramos o tempo de execução, consumo de energia e o EDP. Esta métrica é utilizada para estudar o custo-benefício entre o total de energia consumida e o tempo de execução de uma aplicação. Sua fórmula consiste da multiplicação da energia pelo tempo de execução. Ela vem sendo bastante utilizada pois possibilita analisar, em um único valor, a relação entre o consumo de energia e o desempenho. Por exemplo, considerando dois cenários: (*i*) uma aplicação é executada em 100 segundos e consumiu 10 joules de energia; (*ii*) uma aplicação é executada em 50 segundos e consumiu 40 joules de energia; o cenário *i* teria EDP de 1000, enquanto que o cenário *ii* teria EDP de 2000. Isso mostra que, embora o cenário *i* tenha sido duas vezes mais lento, ele possui a melhor relação de consumo de energia e desempenho.

O EDP da execução de cada aplicação foi obtido pela multiplicação do tempo de execução (em segundos) pelo consumo de energia (em Joules). O tempo foi obtido com a função do OpenMP `omp_get_wtime`. Por outro lado, o consumo de energia foi obtido diretamente dos contadores de hardware presentes nos processadores modernos via *Application Power Management* [Hackenberg et al. 2013]. Os resultados apresentados na Sessão 4 são uma média de dez execuções de cada configuração com um desvio padrão menor que 0.5%.

4. Resultados Experimentais

Nesta seção, os resultados obtidos através dos experimentos realizados são apresentados e discutidos. Para tanto, uma análise do comportamento de escalabilidade das aplicações em cada arquitetura paralela é apresentada na Seção 4.1. Já na Seção 4.2, nós discutimos os resultados obtidos através dos experimentos da execução das aplicações paralelas em cada cenário considerado.

4.1. Análise de Escalabilidade

Para analisar a escalabilidade de cada aplicação paralela, isto é, encontrar o número de *threads* que entrega o melhor resultado em cada arquitetura, nós executamos cada aplicação em cada sistema *multicore* com diferentes números de *threads*: de 1 até o número de núcleos disponíveis na arquitetura. Esta análise é importante para que possamos analisar o melhor resultado obtido no ambiente heterogêneo de acordo com a arquitetura utilizada. A Tabela 1 destaca o número de *threads* que apresentou o melhor resultado de EDP para cada aplicação e arquitetura multicore. Adicionalmente, a Figura 2 mostra o EDP da execução de cada aplicação com o melhor número de *threads* normalizado pela execução com o número de *threads* igual ao número de núcleos, que corresponde a maneira padrão que aplicações paralelas são executadas. Nesta figura, quanto menor for a barra, maior é a melhoria no EDP. Conforme pode-se observar, uma parte significativa das aplicações tem melhor resultado de EDP quando são executadas com um número de *threads* menor que o número de núcleos. Diferentes são as razões para a falta de escalabilidade, conforme já discutido na Seção 2: sincronização de dados, acessos concorrentes à memória compartilhada e saturação do barramento *off-chip* [Suleman et al. 2008].

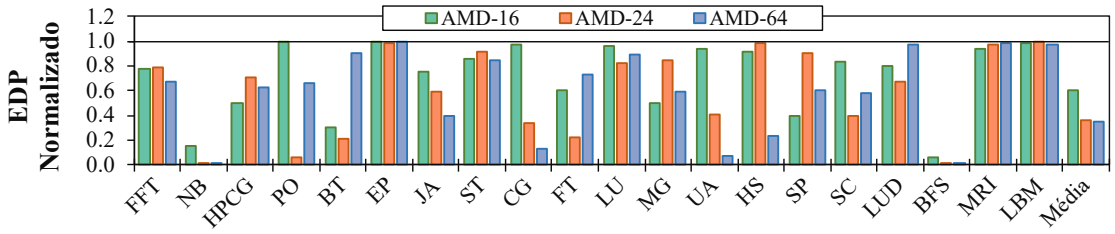


Figura 2. Melhor resultado encontrado pela busca exaustiva normalizado pela execução padrão. Quanto menor o valor, melhor.

Tabela 1. Número de *threads* encontrado pela busca exaustiva em cada arquitetura, que entrega o melhor resultado de EDP

	FFT	NB	HPCG	PO	BT	EP	JA	ST	CG	FT	LU	MG	UA	HS	SP	SC	LUD	BFS	MRI	LBM
AMD-16	12	6	2	16	16	16	2	16	16	12	16	12	16	16	2	16	16	2	2	16
AMD-24	4	2	2	12	8	24	2	24	4	6	12	12	6	24	2	4	8	2	2	24
AMD-64	4	4	2	24	32	64	2	64	4	8	20	4	64	32	2	12	12	2	2	64

Para melhor entender este cenário, vamos considerar o comportamento da aplicação *BFS*, que apresenta nível baixo de escalabilidade independente da arquitetura alvo devido a quantidade de operações de sincronização de dados. Assim, quanto maior o número de *threads* que precisam executar a região crítica, maior será o tempo de sincronização, o que eventualmente aumentará o tempo de execução e consumo de energia, piorando o EDP resultante. Este comportamento é ilustrado na Figura 3 para a execução da aplicação *BFS* no AMD-64. Ela destaca para cada número de *threads*, o tempo de execução (eixo *y* primário) dividido em duas partes: o tempo para executar as regiões paralelas e para sincronizar; e o consumo de energia (eixo *y* secundário). Conforme pode ser observado, a partir da execução com 4 threads, inclusive, a sincronização leva mais tempo para executar que a própria execução da região paralela. Portanto, o EDP piora e não é possível atingir melhores resultados com o aumento no número de *threads*.

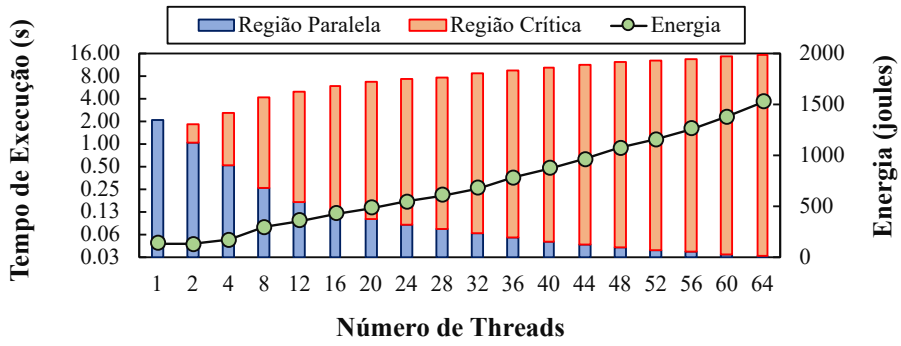


Figura 3. Escalabilidade da aplicação BFS na arquitetura AMD-64 (tempo em log_2)

Adicionalmente, conforme destacado na Tabela 1, o número ideal de *threads* para executar uma dada aplicação muda de acordo com a arquitetura. Por exemplo, a aplicação *FT* é melhor executada com 12 *threads* no AMD-16, enquanto o melhor resultado em EDP é obtido com 6 e 8 *threads* no AMD-24 e AMD-64, respectivamente. Esta heterogeneidade no melhor número de *threads* está diretamente relacionada às características da aplicação e da arquitetura alvo, o que influencia diretamente no resultado bruto de tempo de execução, consumo de energia e EDP.

Neste sentido, a Figura 4 mostra uma comparação da execução das aplicações em cada arquitetura alvo para as métricas avaliadas neste trabalho: tempo de execução, energia e EDP. Cada barra representa o resultado da execução da aplicação em cada arquitetura normalizada pelo melhor resultado obtido entre as três arquiteturas alvo. Portanto, quanto mais próximo de 1.0, melhor o resultado. Conforme observado, não existe uma única arquitetura capaz de entregar o melhor resultado para todas as aplicações. Por exemplo, enquanto o melhor desempenho da *HPCG* é atingido na AMD-24, o melhor desempenho da *EP* é obtido na AMD-64.

Um outro ponto importante a se notar é que a arquitetura que entrega o melhor resultado de desempenho, energia e EDP para a mesma aplicação também muda, como por exemplo, para *FFT*, *NB*, *HPCG*, apenas para citar algumas. Para mostrar este comportamento, consideramos a aplicação *FFT*. O melhor desempenho é obtido no AMD-24, sendo 9% e 52% melhor quando comparada a execução nas arquiteturas AMD-16 e AMD-64, respectivamente. Por outro lado, o melhor consumo de energia é obtido no AMD-16 com o consumo de energia 49% e 60% menor que o obtido no AMD-24 e AMD-64, respectivamente. Por fim, o melhor EDP também é atingido na AMD-16, com significativa redução comparada às demais arquiteturas.

4.2. Considerando o Grau de Paralelismo e Características da Arquitetura para Otimizar Desempenho, Energia e EDP

Na seção anterior, mostramos que, utilizar uma combinação ideal de número de *threads* e arquitetura *multicore* para executar uma única aplicação em um ambiente heterogêneo proporciona ganhos significativos em tempo de execução, energia e EDP. No entanto, quando um lote de aplicações é submetida para execução em um *cluster* heterogêneo, diferentes estratégias podem ser utilizadas para alocar uma aplicação em uma determinada arquitetura. Neste sentido, para esta análise, consideramos as seguintes estratégias: (*i*) todas as aplicações são executadas em uma única arquitetura (AMD-16, AMD-24 e AMD-

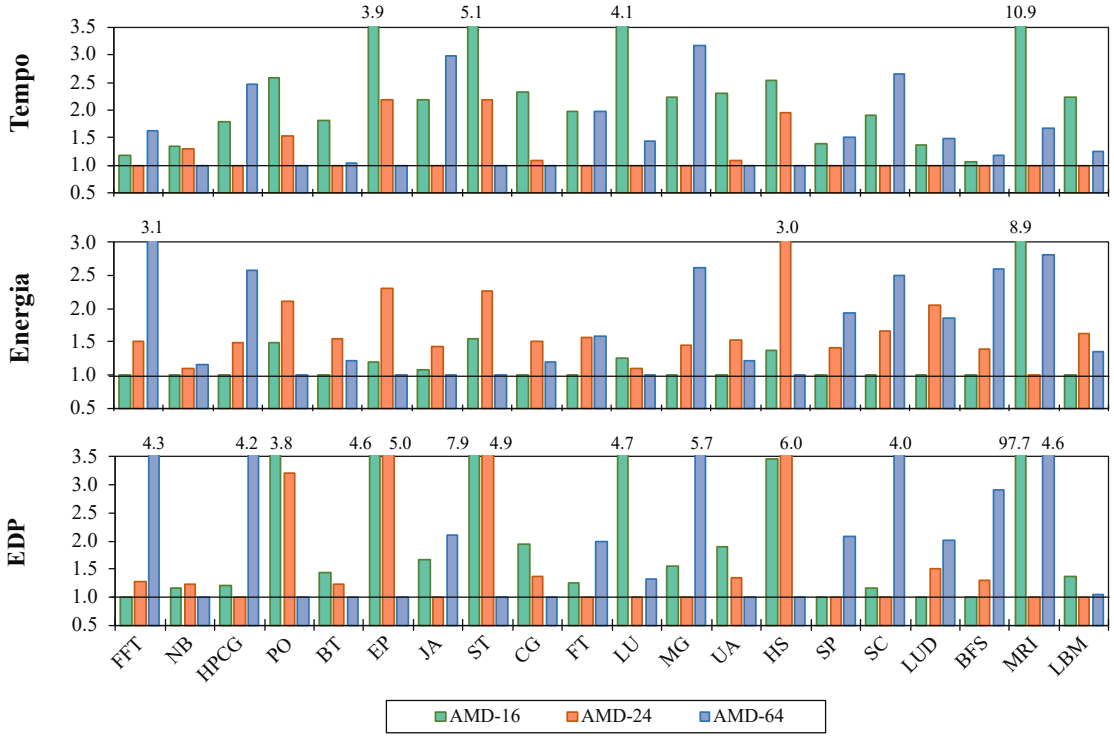
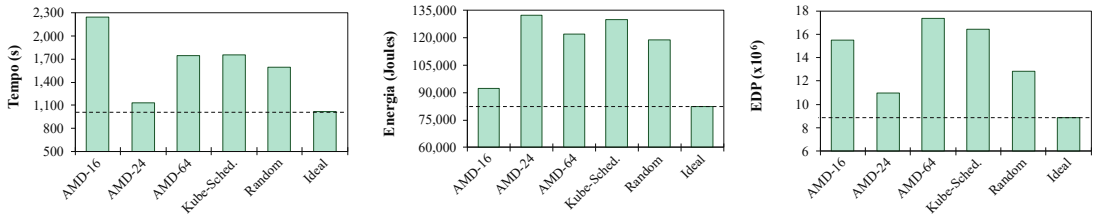


Figura 4. Resultados de cada aplicação executando com o número ideal de *threads* normalizado pelo melhor resultado obtido entre as arquiteturas

64); (*ii*) *Kube-Scheduler*, o escalonador padrão do *Kubernetes*, onde os *jobs* (aplicações) são distribuídas entre os nodos através da política *round-robin*; (*iii*) *Random*, um escalonador baseado em distribuição aleatória. (*iv*) *TLP*, onde considera a melhor atribuição entre aplicação - arquitetura considerando o TLP da aplicação (Tabela 2).

A Figura 5 ilustra o tempo, consumo de energia e EDP que cada estratégia obteve para executar todo o conjunto de aplicações descrito na Metodologia. Cada barra representa uma configuração, e a linha pontilhada destaca o resultado obtido pela configuração *TLP*. Para todas as métricas, quanto menor o valor, melhor. Conforme pode-se observar, o melhor desempenho, consumo de energia e EDP são obtidos quando a estratégia TLP é utilizada. Ela é 43% mais rápida e consome 36% menos energia que a estratégia adotada por padrão pelo *Kubernetes* (*Kube-Scheduler*). Adicionalmente, nenhuma outra estratégia é capaz de fornecer resultados similares considerando todas as métricas avaliadas: AMD-24 fornece o segundo melhor desempenho, mas também o pior consumo de energia. Por outro lado, AMD-16 fornece o segundo melhor consumo de energia, mas também o pior consumo de energia. A principal razão para este comportamento está na escolha otimizada em dois níveis: (*a*) no nível da aplicação, através do melhor número de *threads*; e (*b*) no nível de arquitetura, selecionando a arquitetura que fornece o melhor resultado para a respectiva aplicação. A Figura 6 destaca este comportamento.

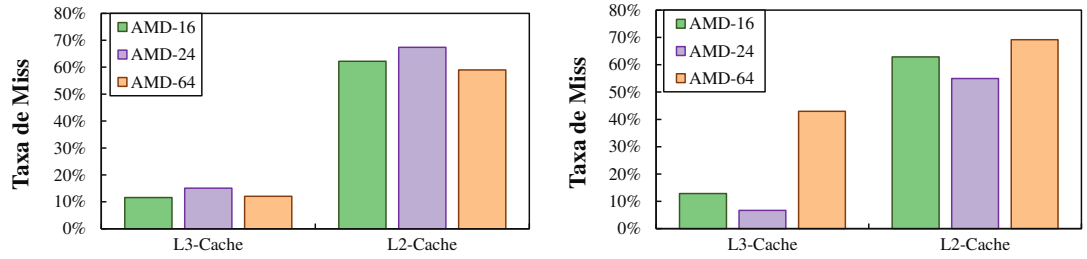
Para aplicações onde a escalabilidade é limitada pela quantidade de acessos à memória compartilhada (*e.g.*, memória *cache* L3 e principal), a arquitetura que proporcionou o melhor resultado foi aquela que atingiu a menor taxa de *misses* nos níveis de memória *cache* mais distantes do processador. Um exemplo é a aplicação *HPCG*, onde seu comportamento relacionado a taxa de *misses* nos níveis da *cache* é apresentado na



(a) Tempo de execução (b) Consumo de energia (c) EDP

Figura 5. Resultado da execução de todas as aplicações no ambiente heterogêneo considerando as diferentes estratégias

Figura 6.b. Nela, podemos observar que a arquitetura AMD-24 teve menor taxa de *misses* em ambos os níveis de memória cache, contribuindo assim para entregar um melhor resultado. Por outro lado, para aplicações com boa escalabilidade (*e.g.*, EP, HS e PO), o comportamento de memória tem pouca influencia, uma vez que o mais importante para o desempenho é a quantidade de recursos de *hardware* e o poder computacional disponível. Assim, a arquitetura AMD-64 sobressai com relação às demais arquiteturas. Por fim, a Figura 6.b ilustra que a taxa de misses entre as três arquiteturas permanece similar para a aplicação EP.



(a) EP (b) HPCG

Figura 6. Comportamento com relação ao número de *misses* nas *caches* L2 e L3

5. Conclusão e Trabalhos Futuros

Este artigo apresentou um estudo da execução de aplicações paralelas com diferentes características de grau de paralelismo em um ambiente de nuvem heterogêneo. Embora diferentes trabalhos tenham focado em otimizar o uso dos recursos computacionais na nuvem, a literatura carece de trabalhos que analisaram o potencial ganho de desempenho, redução no consumo de energia e EDP na execução de aplicações com diferentes características de escalabilidade em um ambiente heterogêneo. Como trabalhos futuros, pretende-se (*i*) definir um modelo algorítmico para encontrar a melhor combinação de aplicações paralelas, grau de paralelismo e arquitetura ideal.

Referências

Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. (1991). The nas parallel benchmarks and summary and preliminary results. In *ACM/IEEE SC*, pages 158–165, USA. ACM.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177.

Charr, J.-C., Couturier, R., Fanfakh, A., and Giersch, A. (2015). Energy consumption reduction with dvfs for message passing iterative applications on heterogeneous architectures. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 922–931. IEEE.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *IEEE Int. Symp. on Workload Characterization*, pages 44–54, DC, USA. IEEE Computer Society.

Chen, J.-J., Huang, K., and Thiele, L. (2012). Dynamic frequency scaling schemes for heterogeneous clusters under quality of service requirements. *Journal of Information Science and Engineering*, 28(6):1073–1090.

da Silva, V. S., Nogueira, A. G., de Lima, E. C., de A. Rocha, H. M., Serpa, M. S., Luizelli, M. C., Rossi, F. D., Navaux, P. O., Beck, A. C. S., and Francisco Lorenzon, A. (2021). Smart resource allocation of concurrent execution of parallel applications. *Concurrency and Computation: Practice and Experience*, page e6600.

Dongarra, J., Heroux, M. A., and Luszczek, P. (2015). Hpcg benchmark: A new metric for ranking high performance computing systems. *Knoxville, Tennessee*.

Hackenberg, D., Ilsche, T., Schone, R., Molka, D., Schmidt, M., and Nagel, W. E. (2013). Power measurement techniques on standard compute nodes: A quantitative comparison. In *IEEE ISPASS*, pages 194–204.

Ham, T. J., Chelepalli, B. K., Xue, N., and Lee, B. C. (2013). Disintegrated control for energy-efficient and heterogeneous memory systems. In *IEEE HPCA*, pages 424–435.

Huang, J., Xiao, C., and Wu, W. (2020). Rlsk: A job scheduler for federated kubernetes clusters based on reinforcement learning. In *2020 IEEE International Conference on Cloud Engineering (IC2E)*, pages 116–123.

Khaleghzadeh, H., Fahad, M., Shahid, A., Manumachu, R. R., and Lastovetsky, A. (2021). Bi-objective optimization of data-parallel applications on heterogeneous hpc platforms for performance and energy through workload distribution. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):543–560.

Li, J. and Martinez, J. F. (2005). Power-performance considerations of parallel computing on chip multiprocessors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(4):397–422.

Liu, F., Tong, J., Mao, J., Bohn, R., Messina, J., Badger, L., and Leaf, D. (2012). *NIST Cloud Computing Reference Architecture: Recommendations of the National Institute of Standards and Technology*. CreateSpace Independent Publishing Platform, USA.

Lorenzon, A. F. and Beck Filho, A. C. S. (2019). *Parallel computing hits the power wall: principles, challenges, and a survey of solutions*. Springer Nature.

Maghsoud, Z., Noori, H., and Pour Mozaffari, S. (2021). Peps: Predictive energy-efficient parallel scheduler for multi-core processors. *The Journal of Supercomputing*, 77(7):6566–6585.

Makrani, H. M., Sayadi, H., Motwani, D., Wang, H., Rafatirad, S., and Homayoun, H. (2018). Energy-aware and machine learning-based resource provisioning of in-memory analytics on cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 517–517.

Marques, S. M., Medeiros, T. S., Rossi, F. D., Luizelli, M. C., Beck, A. C. S., and Lorenzon, A. F. (2021). Synergically rebalancing parallel execution via dct and turbo boosting. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 277–282. IEEE.

Márquez, G., Villegas, M. M., and Astudillo, H. (2018). A pattern language for scalable microservices-based systems. In *ECSA*, NY, USA. ACM.

Masanet, E., Shehabi, A., Lei, N., Smith, S., and Koomey, J. (2020). Recalibrating global data center energy-use estimates. *Science*, 367(6481):984–986.

Orhean, A. I., Pop, F., and Raicu, I. (2018). New scheduling approach using reinforcement learning for heterogeneous distributed systems. *Journal of Parallel and Distributed Computing*, 117:292–302.

Park, J. and Abraham, J. A. (2011). A fast, accurate and simple critical path monitor for improving energy-delay product in dvs systems. In *IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 391–396. IEEE.

Schwarzrock, J., de Oliveira, C. C., Ritt, M., Lorenzon, A. F., and Beck, A. C. S. (2020). A runtime and non-intrusive approach to optimize edp by tuning threads and cpu frequency for openmp applications. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1713–1724.

Stratton, J., Rodrigues, C., Sung, I., Obeid, N., Chang, L., Anssari, N., Liu, G., and Hwu, W. (2012). Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*.

Suleman, M. A., Qureshi, M. K., and Patt, Y. N. (2008). Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps. *SIGARCH Comput. Archit. News*, 36(1):277–286.

Takouna, I., Dawoud, W., and Meinel, C. (2012). Energy efficient scheduling of hpc-jobs on virtualize clusters using host and vm dynamic configuration. *ACM SIGOPS Operating Systems Review*, 46(2):19–27.

Thurgood, B. and Lennon, R. G. (2019). Cloud computing with kubernetes cluster elastic scaling. In *ICFNDS*, NY, USA. ACM.