PampaAffinity: Otimização de Aplicações Paralelas via Ajuste Dinâmico e Transparente do Grau de Paralelismo e Mapeamento de *Threads*[*]

Valmir T. Junior[1], Thiarles S. Medeiros[1], Janaína Schwarzrock[4], Samuel Xavier-de-Souza[2], Fábio D. Rossi[3], Marcelo C. Luizelli[1], Antonio Carlos S. Beck[4] e Arthur F. Lorenzon[1]

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

[2]Universidade Federal do Rio Grande do Norte – Natal – RN – Brasil

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

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

valmirthume.aluno@unipampa.edu.br

Abstract. *The design of applications that can efficiently use computational resources has become a challenge for end-users due to software and hardware characteristics that affect the scalability of many parallel applications. Hence, thread-throttling and thread-to-core mapping strategies have been employed to optimize the use of these computational resources. However, the design space exploration significantly grows with the number of cores in the architecture, making the task of finding an ideal configuration of the degree of parallelism and thread-to-core mapping challenging. Therefore, we propose PampaAffinity, a dynamic, automatic, and transparent approach to the end-user, which adjusts the number of threads and thread-to-core mapping policies for each parallel region of OpenMP applications. By executing thirteen applications on three multicore architectures, we show that PampaAffinity converges to an ideal solution with an average accuracy of 85% and optimizes the tradeoff between performance and energy consumption by 96.1% when compared to the standard way that parallel applications are executed.*

Resumo. *O desenvolvimento de aplicações que possam utilizar de maneira eficiente os recursos computacionais tem se tornado um desafio para os usuários devido às características do software e hardware que afetam a escalabilidade de muitas aplicações paralelas. Neste sentido, estratégias de ajuste dinâmico do número de threads e mapeamento de threads para núcleos de processamento têm sido empregadas para otimizar o uso destes recursos computacionais. No entanto, o espaço de exploração cresce significativamente com o número de núcleos da arquitetura, tornando a tarefa de encontrar uma configuração ideal de grau de paralelismo e mapeamento de threads desafiadora. Assim, nós propomos PampaAffinity, uma abordagem dinâmica, automática e transparente para o usuário, que realiza o ajuste do número de threads e políticas de mapeamento de threads para cada região paralela de aplicações OpenMP. Com a execução de treze aplicações em três arquiteturas multicore, mostramos que PampaAffinity converge para uma solução ideal com precisão média de 85% e otimiza o tradeoff entre desempenho e consumo de energia em 96.1% quando comparado à maneira padrão que aplicações paralelas são executadas.*

[*]Este trabalho foi parcialmente financiado pela FAPERGS nos projetos 19/2551-0001224-1, 19/2551-0001689-1 e 17/2551-0001193-7 e PROBIC-FAPERGS

1. Introdução

Devido ao aumento do número de núcleos em sistemas computacionais de alto desempenho (HPC - *high-performance computing*) ao longo dos anos, a exploração de paralelismo no nível de *threads* (TLP - *thread-level parallelism*) continua a ser amplamente empregada para melhorar o desempenho de aplicações executadas em *data-warehouses*. No entanto, a exploração do TLP também pode contribuir para o aumento do consumo de energia dos sistemas de HPC quando os recursos computacionais são utilizados de maneira ineficiente. Este cenário acontece, principalmente, quando aplicações com escalabilidade limitada são executadas com um número maior de *threads* do que o necessário e quando os recursos de *hardware* (núcleos e memória *cache*) são subutilizados devido ao emprego incorreto de estratégias de mapeamento de *threads*.

Para lidar com a escalabilidade limitada de aplicações paralelas, estratégias que realizam o ajuste dinâmico do número de *threads* podem ser empregadas. Neste cenário, o grau de paralelismo de uma aplicação é ajustado com base nos limites da escalabilidade, que podem estar relacionadas ao *software* (sincronização de dados e comunicação de dados) e/ou ao *hardware* (saturação das unidades funcionais e barramento *off-chip*) [Suleman et al. 2008, Joao et al. 2012, Schwarzrock et al. 2021, da Silva et al. 2021]. No entanto, embora estas estratégias oferecem melhor relação entre desempenho e consumo de energia do que a abordagem padrão (execução com o número de *threads* igual ao número de núcleos disponíveis no sistema), depender do Sistema Operacional para distribuir as *threads* entre os núcleos de processamento pode limitar os ganhos de desempenho e redução no consumo de energia.

Portanto, estratégias de mapeamento e alocação de *threads* são empregadas em conjunto com o ajuste dinâmico do número de *threads* para balancear o uso dos recursos de *hardware*. Estas estratégias têm o objetivo de otimizar o acesso à memória *cache* e reduzir a quantidade de acessos à memória principal decorrente da comunicação entre as *threads*. Como um exemplo, para aplicações com escalabilidade limitada pela quantidade de comunicação entre as *threads*, manter as *threads* vizinhas compartilhando os mesmos níveis de memória poderia acelerar a troca de dados. Por outro lado, aplicações que utilizam intensivamente a CPU podem ser otimizadas se as *threads* não forem alocadas na mesma unidade de processamento, evitando contenção das unidades funcionais. Para otimizar esta tarefa e deixa-lá transparente para o programador, interfaces de programação paralela, como por exemplo, o OpenMP, fornecem diferentes políticas de alocação.

Neste sentido, o comportamento das aplicações paralelas não está apenas relacionado ao número de *threads* ativas, mas também a como elas são distribuídas entre as unidades de processamento [Lorenzon and Beck Filho 2019]. Adicionalmente, arquiteturas com acesso não uniforme à memória (NUMA – *non-uniform memory access*) tornam o desafio de encontrar uma solução ideal (*threads* e estratégias de mapeamento) ainda mais complexo, uma vez que *threads* podem acessar dados armazenados na memória remota (de outro nodo NUMA), que possui maior tempo de acesso [Cruz et al. 2016]. Portanto, o espaço de exploração e projeto envolvido em encontrar uma configuração ideal é desafiador devido À natureza das aplicações paralelas e das arquiteturas multicore modernas.

Dadas as considerações acima, este artigo propõe *PampaAffinity*: uma abordagem que emprega um algoritmo de aprendizagem *online* para encontrar a combinação do número de *threads*, política de mapeamento e estratégia de afinidade de *threads* de aplicações OpenMP para otimizar a relação entre desempenho e consumo de energia, representado pela métrica *energy-delay product* (EDP). *PampaAffinity* é completamente dinâmico: através do aprendizado em tempo de execução, ele é capaz de se adaptar automaticamente a diferentes conjuntos de entradas das aplicações, assim como a micro-

arquitetura na qual está sendo empregado. Adicionalmente, *PampaAffinity* é transparente ao usuário: sua implementação interna a biblioteca de linkagem dinâmica do OpenMP faz com que possa ser utilizado para otimizar aplicações OpenMP sem a necessidade de recompilação ou alterações no código fonte.

Nós validamos *PampaAffinity* através da execução de treze aplicações paralelas em três arquiteturas multicore: um processador Intel e dois AMD. Nós comparamos a precisão (isto é, a capacidade de convergir para uma solução ideal) com o melhor resultado encontrado por uma busca exaustiva que considera o espaço de exploração do número de *threads*, políticas de mapeamento e estratégias de afinidade de *threads*. Assim, mostramos que *PampaAffinity* é capaz de convergir para uma solução ideal ou próxima da ideal na maioria dos casos, com uma precisão média de 85%. Adicionalmente, ao comparar com a maneira padrão que aplicações OpenMP são executadas, *PampaAffinity* é capaz de apresentar EDP 96.1% melhor no processador Intel com 88 núcleos, 70.9% e 71.5% melhor nos processadores AMD com 24 e 128 núcleos, respectivamente.

O restante deste artigo está organizado como segue. Na Seção 2 é descrito a fundamentação teórica, além dos trabalhos relacionados. Então, PampaAffinity é apresentado na Seção 3. A metodologia utilizada na avaliação de PampaAffinity é descrita na Seção 4. Por fim, os resultados experimentais são discutidos na Seção 5 enquanto que a conclusão é destacada na Seção 6.

2. Fundamentação Teórica

2.1. Políticas de Posicionamento de *Threads*

O posicionamento (*placement* no OpenMP) de *threads* envolve a definição da granularidade a ser usada. *Socket*, *core*, ou *threads* de *hardware* são os *places* possíveis a serem definidos. Cada *place* é, naturalmente, composto por um conjunto de Unidades de Processamentos (PUs - *Processing Units*), que representam as *threads* de hardware onde a aplicação será executada. No OpenMP [OpenMP Architecture Review Board 2018], há três políticas pré-definidas de *placement* de *threads*: ***Sockets***: O número de *places* é igual ao número de *sockets* na arquitetura; ***Cores***: O número de *places* é igual ao de *cores* do sistema; ***Threads***: A granularidade mais fina possível, onde cada *core* pode executar mais que uma *thread* quando ele implementa *Simultaneous Multithreading - SMT*. Uma vez que a granularidade está definida, a política de *placement* irá informar ao Sistema Operacional a qual *place* as *threads* serão alocadas. As *threads* serão alocadas em uma *PU* de um dado *place* conforme uma estratégia de afinidade (discutido na Seção 2.2).

2.2. Estratégias de Afinidade de *Threads*

A afinidade trata do mapeamento de *threads* para *PUs*. O OpenMP possui cinco estratégias de afinidade pré-definidas que são utilizadas com as políticas de *placement*:

Master: As *threads* são alocadas em *PUs* que estão no mesmo *place* da *thread master* (i.e., a *thread* que executa a parte sequencial, cria outras *threads* e distribui a carga de trabalho entre elas). Quando a política de *placement* é definida como *Sockets*, as *threads* são distribuídas pelas *PUs* que estão no mesmo *socket* da *thread master*. Quando a política de *placement* é definida como *Core*, elas serão alocadas nas *PUs* com o mesmo *core id*. E, para a política de *placement Threads*, as *threads* irão compartilhar a mesma *PU* da *thread master*.

Close: As *threads* são mantidas próximas à *thread master* em partições contíguas de *places*, ou seja, após alocar uma *thread* em um *place*, a próxima *thread* será alocada no *place* subsequente. Quando o número de *threads* (NT) for maior que o de *places* (P), grupos de *threads* serão criados com *threads* de numeração consecutiva. Cada grupo terá

tamanho S_p definido por $\lfloor NT/P \rfloor \leq S_p \leq \lceil NT/P \rceil$. Então, cada grupo será alocado em um *place* consecutivo. Quando usadas juntamente com a política *Sockets* e com o número de *threads* menor ou igual aos número de *places* ($NT \leq P$), cada *thread* em execução será alocada em uma *PU* de cada *socket*. Caso contrário, quando $NT > P$, mais de uma *thread* será alocada nas *PU*s de um mesmo *socket*. De forma análoga, quando a granularidade é definida para *Cores* e $NT \leq P$ cada *thread* em execução será alocada em uma *PU* de cada *core*. Quando $NT > P$, mais que uma *thread* será alocada para as *PU*s de um mesmo *core*. Por fim, quando a granularidade *Threads* é definida, as *threads* em execução serão alocadas seguindo os identificadores de cada *PU*, ou seja, *thread* 0 na *PU* 0, *thread* 1 na *PU* 1, e assim sucessivamente.

Spread: As *threads* são distribuídas de forma espalhada pelas partições de *places*. Primeiramente divide-se em subpartições, sendo o número de subpartições igual ao menor valor entre o número de threads (NT) e de *places*(P). Quando $NT \leq P$ então cada *thread* será atribuída a uma subpartição que contém $\lfloor P/NT \rfloor$ ou $\lceil P/NT \rceil$ *places* consecutivos. Ou seja, haverá NT subpartições de *places*. Assim busca-se manter o maior afastamento entre as *threads* que estão sendo alocadas. Caso contrário, quando $NT > P$ serão criados grupos de *threads* com números consecutivos de *threads*. Cada grupo terá tamanho S_p definido por $\lfloor NT/P \rfloor \leq S_p \leq \lceil NT/P \rceil$. Então, cada grupo será alocado em um *place* consecutivo. Observa-se que quando $NT > P$, essa afinidade possui o mesmo comportamento da afinidade *Close*.

False: Afinidade de *threads* e a política de *placement* são desabilitadas e o Sistema Operacional define em qual *PU* uma dada *thread* irá ser alocada, além disso as *threads* poderão se mover pelas *PU*s disponíveis. Com essa estratégia de afinidade definida mesmo havendo a definição da forma de criação dos *places* o sistema desconsidera e aloca as *threads* da forma que julgar mais adequado.

True: O Sistema Operacional é responsável por definir em qual *place* cada *thread* ficará alocada até seu término de execução, logo uma vez que as *threads* são alocadas não serão movidas entre os *places* existentes. Cabe destacar que a *thread* pode ser movida entre as *PU*s que compõe o *place* em que ela está alocada.

2.3. Trabalhos Relacionados

Os seguintes trabalhos otimizam aplicações paralelas por meio do ajuste do número de *threads* (quantidade de TLP). [Suleman et al. 2008] propõem FDT, um *framework* que encontra uma configuração ideal do número de *threads* usando modelos de limitações de escalabilidade devido a sincronização de dados e saturação de largura de banda de memória. [Sridharan et al. 2014] propõem Varuna, uma biblioteca que usa modelos de predição para encontrar a melhor configuração de número de threads para otimizar aplicações paralelizadas com Pthreads ou TBB. Para fazer uso de Varuna, a aplicação precisa ser recompilada. [Wang et al. 2016] propõem *NuCore*, um algoritmo para ajuste de TLP direcionado para sistemas NUMA com objetivo de maximizar uso de largura de banda e melhorar desempenho. [Lorenzon et al. 2018] propõem Aurora, uma biblioteca para otimizar EDP de aplicações paralelas OpenMP por meio do ajuste do número de *threads*. A biblioteca usa um algoritmo baseado em *hill-climbing* para aprendizado. No entanto, nenhum destes trabalhos se preocupa em melhorar o mapeamento de *threads*.

Os seguintes trabalhos objetivam melhorar o mapeamento de *threads* para otimizar aplicações paralelas. [Broquedis et al. 2010] propuseram uma extensão para a biblioteca do OpenMP no qual *threads* são alocadas nas unidades de processamento de forma a maximizar o reuso de dados nas caches. A biblioteca depende de informações que devem ser anotadas na aplicação pelo programador. As abordagens propostas por [Cruz et al. 2012] e [Diener et al. 2013] buscam alocar threads que mais se comunicam

em PUs que compartilham caches. Para fazer isso de forma automática, as abordagens estimam a comunicação entre threads analisando dados nas TLBs [Cruz et al. 2012] ou tabela de páginas [Diener et al. 2013]. Essas técnicas necessitam modificação de hardware [Cruz et al. 2012] ou do sistema operacional [Diener et al. 2013], o que dificulta a utilização das abordagens propostas. Além disso, nenhuma dessas abordagens se preocupa com o ajuste de TLP.

Ao invés de analisar comunicação entre *threads*, uma forma simples e efetiva para melhorar o mapeamento das mesmas é fazer uso de estratégias pré-definidas de afinidade. [Eichenberger et al. 2012] propuseram diferentes estratégias de afinidade de *threads* e políticas de granularidade de alocação para otimizar mapeamento de *threads* em aplicações OpenMP. Contudo, a escolha pelas melhores configurações era responsabilidade do programador. Nosso trabalho automatiza a escolha dos melhores parâmetros para essas configurações, além de ajustar o TLP. [Papadimitriou et al. 2019] aplicaram otimização de estratégia de afinidade de *threads* (escolhendo entre duas: *spread* ou *close*) combinado com ajuste de voltagem e frequência do processador com objetivo de economizar energia em servidores ARM. [De Sensi et al. 2016] propuseram *Nornir*, um algoritmo para encontrar uma configuração ideal de número de *threads*, frequência da CPU, e estratégia de afinidade de *threads* para satisfazer requisitos de performance ou consumo de potência. *Nornir* considera apenas duas estratégia de afinidade diferentes. Neste trabalho, além de consideramos cinco estratégias no espaço de busca, também consideramos a variação da granularidade de alocação (políticas de posicionamento).

Contribuição: Este trabalho é o primeiro a otimizar EDP de aplicações paralelas OpenMP por meio do ajuste automático das configurações de (i) número de *threads*, (ii) estratégia de afinidade de *threads*, e (iii) política de posicionamento. Nossa abordagem não requer trabalho do programador, pois escolhe e ajusta as melhores configurações de maneira automática e transparente. Ou seja, nossa abordagem não requer modificação de hardware ou SO, nem alteração ou recompilação da aplicação.

3. PampaAffinity: Otimizando o Número de Threads, Posicionamento e Afinidade de Threads

PampaAffinity é uma abordagem dinâmica e online, sendo dividido em duas fases. A primeira fase é a de busca, onde *PampaAffinity* considera uma etapa de *warm-up*, que é a primeira vez que uma região paralela da aplicação será executada, provocando misses na memória *cache* e TLB (*Translation Lookaside Buffer*), o que degrada o desempenho. Portanto, considerar esta etapa durante a convergência do algoritmo de busca poderia levar a um resultado longe do ideal. Neste sentido, durante a etapa de *warm-up*, *PampaAffinity* executa cada região com a configuração padrão (número máximo de *threads* disponível no sistema, *place* e afinidade configurados para a configuração padrão do Sistema Operacional. Após esta etapa, *PampaAffinity* aplica o algoritmo de busca, descrito a seguir, sobre cada região paralela da aplicação. A cada vez que uma região paralela é executada, uma nova configuração (número de *threads*, *place* e afinidade) é avaliada.

Uma vez que a busca converge para uma solução em cada região, a fase estável é iniciada. Deste momento, até o final da execução, a região paralela executa com a configuração a ela atribuída. Por fim, *PampaAffinity* é responsável por reconfigurar o número de *threads*, *place* e afinidade sempre que a aplicação entra em uma região paralela. Nós implementamos *PampaAffinity* internamente à biblioteca GNU OpenMP [Chapman et al. 2007]. Neste sentido, como as aplicações OpenMP são linkadas dinamicamente com a biblioteca OpenMP, qualquer aplicação OpenMP que explora o paralelismo no nível de laços pode usufruir das vantagens do *PampaAffinity*.

Sejam os recursos de *hardware* e as estratégias de alocação descritos como: o

conjunto de c distintas unidades de processamento definido por $C = \{C_1, C_2, ..., C_c\}$; o conjunto de p distintas políticas de *placement* definido por $P = \{P_1, P_2, ..., P_p\}$; e, t diferentes estratégias de afinidade de *threads* definidas por $T = \{T_1, T_2, ..., T_t\}$; e, o conjunto de r regiões paralelas da aplicação definido por $R = \{R_1, R_2, ..., R_r\}$. O problema objeto de estudo é encontrar um subconjunto de *threads*/*cores* em C, com uma política de *placement* em P, e com a afinidade de *threads* em T que otimiza o *trade-off* entre desempenho e consumo de energia (EDP - *energy-delay product*) na execução de uma região paralela de R pertencente à aplicação A. O algoritmo de aprendizagem, descrito em Algoritmo 1, realiza a busca da configuração ideal conforme os valores obtidos da execução. Denota-se $\mathbb{M} : (C \times P \times T) \rightarrow \mathbb{R}^+$ função $\mathbb{M}$ como a medida obtida da execução de uma região paralela cujo domínio é o espaço de busca compreendido pela combinação dos distintos *cores* de C aplicando as políticas de *placement* de P e estratégias de afinidade de *threads* de T. Definindo como ϕ o conjunto de configuração de c, p e t e ϕ' como o valor da métrica avaliada para ϕ na qual busca-se aproximar do máximo valor de $\mathbb{M}$.

Primeiramente são caracterizadas as entradas necessárias, sendo elas: o conjunto de *cores* C; o conjunto de políticas de *placement* P; o conjunto de estratégias de afinidade de *threads* T para uma aplicação A; e, dois parâmetros: o número de *threads* inicial (α) para executar a aplicação e o fator de incremento do número de *threads* (β). O parâmetro α é calculado e definido pelo algoritmo de busca de acordo com a arquitetura alvo. O algoritmo então seleciona a política de *placement* padrão e a estratégia de afinidade de *threads* para a fase inicial. *PampaAffinity* possui três fases distintas de otimização. Na primeira é realizado o ajuste do número de *threads*, na segunda o ajuste da politica de *placement* e por fim, o ajuste da estratégia de afinidade de *threads*.

Para o ajuste do número de *threads*, é empregado o algoritmo *hill-climbing* modificado. Inicialmente, a aplicação é executada com α *threads* e é medido o M'. Para as próximas execuções o número de *threads* é definido utilizando β como fator multiplicativo de α, sendo o novo valor de $\alpha = \alpha * \beta$. Porém, o valor de α somente é atualizado com base no fator β enquanto se maximiza a função de avaliação $M' = \mathbb{M}(\phi)$ (linhas 7 a 10). Neste momento o *hill-climbing* utiliza um passo numa escala geométrica com razão β. Uma vez que não é encontrado um valor maior que o anterior entende-se que há um máximo local. Então o algoritmo continua a buscar uma solução ϕ' dentro do intervalo entre o máximo valor encontrado e o último ponto analisado. Esse intervalo é delimitado na linha 12 e o ponto médio é definido e avaliada a função M', assim o algoritmo *hill-climbing* tem como passo a metade dos pontos a avaliar entre o mínimo e o máximo podendo aumentar ou diminuir o número de *threads* a avaliar. Avaliado o ponto médio define-se o próximo subintervalo a ser analisado, à direita caso M' apresente um valor maior ou à esquerda caso contrário. Uma vez que a busca pelo número ideal de *threads* foi concluída, é feita a variação das políticas de *placement* (linhas 23 a 28). Da mesma forma que na etapa anterior, busca-se maximizar a função M' definindo a política p a ser utilizada. As estratégias de afinidade de *threads* são avaliadas por último, após a definição da política de *placement* ideal.

4. Metodologia

4.1. Benchmarks

Para a realização dos experimentos, treze aplicações foram utilizadas: **Seis kernels, pseudo-aplicações e benchmarks para computação desestruturada do NAS Parallel Benchmark** [Bailey et al. 1991]: *bt.C.x*, *cg.C.x*, *ft.C.x*, *mg.C.x*, *sp.C.x* e *ua.C.x*. **Uma aplicação da suíte de benchmarks do Rodinia** [Che et al. 2009]: *LUD*. **Seis aplicações de diferentes domínios**: *FFT*, *HPCCG*, *Lulesh2.0*, *Nbody*, *Método de Jacobi* e *Stream*. As aplicações do Rodinia foram executadas com o conjunto de entrada padrão disponibi-

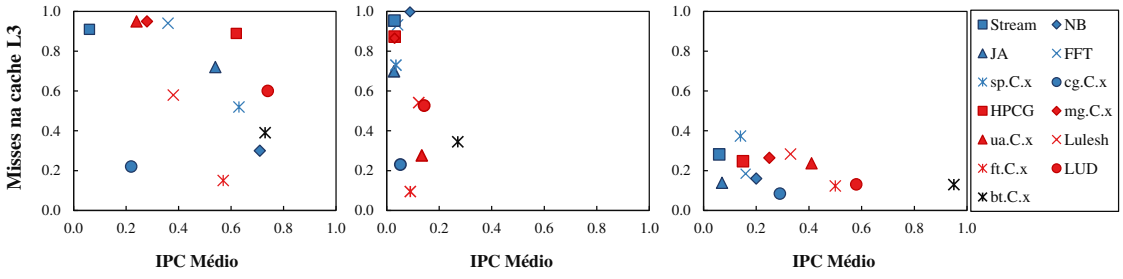
Algorithm 1 Algoritmo de Aprendizagem

Input: $C \leftarrow \{C_1, C_2, \ldots, C_k\}$: conjunto de threads/cores
$P \leftarrow \{HWThreads, Cores, Sockets\}$: conjunto de políticas de placement
$T \leftarrow \{false, close, master, spread, true\}$: conjunto de afinidades de *threads*
α: número inicial de *threads*
β: fator de incremento do número de *threads*

1: $p \leftarrow P\{1\}$: define a primeira política de *placement* a ser avaliada
2: $t \leftarrow T\{1\}$: define a primeira estratégia de afinidade a ser avaliada
3: $\phi(p, t, \alpha) \leftarrow -\infty$: melhor combinação de *placement*, afinidade e grau de TLP até o momento
4: $\phi' \leftarrow \infty$: valor mínimo encontrado até o momento
5: **for** each parallel region **do**
6: $M' \leftarrow \mathbb{M}(p, t, \alpha)$
7: **while** $M' \geq \phi'$ and $\alpha \leq totalCores$ **do**
8: $\phi' \leftarrow M'$ and $\phi \leftarrow (p, t, \alpha)$
9: $\alpha \leftarrow \alpha \times \beta$ and $M' \leftarrow \mathbb{M}(p, t, \alpha)$
10: **end while**
11: **if** $M' \leq \phi'$ and $\alpha \leq totalCores$ **then**
12: $upper \leftarrow \alpha$ and $lower \leftarrow \alpha/\beta$
13: **while** $lower \leq upper$ **do**
14: $\alpha' \leftarrow (upper + lower)/2$ and $M' \leftarrow \mathbb{M}(p, t, \alpha')$
15: **if** $M' \geq \phi'$ **then**
16: $\phi' \leftarrow M'$ and $\phi \leftarrow (p, t, \alpha')$ and $lower \leftarrow \alpha'$
17: **end if**
18: **if** $M' < \phi'$ **then**
19: $upper \leftarrow \alpha'$
20: **end if**
21: **end while**
22: **end if**
23: **for** each p in P **do**
24: $M' \leftarrow \mathbb{M}(p, t, \alpha)$
25: **if** $M' \geq \phi'$ **then**
26: $\phi' \leftarrow M'$ and $\phi \leftarrow (p, t, \alpha)$
27: **end if**
28: **end for**
29: **for** each t in T **do**
30: $M' \leftarrow \mathbb{M}(p, t, \alpha)$
31: **if** $M' \geq \phi'$ **then**
32: $\phi' \leftarrow M'$ and $\phi \leftarrow (p, t, \alpha)$
33: **end if**
34: **end for**
35: **end for**

lizada junto com a suíte; E as demais aplicações com tamanho médio, de acordo com as características de cada aplicação.

As aplicações escolhidas possuem diferentes comportamentos com relação ao acesso a memória e utilização do processador. Neste sentido, nós caracterizamos as aplicações de acordo com a taxa de *misses* na cache L3 e o IPC médio, conforme mostrado



a) AMD24 b) AMD128 c) Intel88

Figura 1. Comportamento de cada aplicação nas arquiteturas alvo

Tabela 1. Principais características de cada arquitetura

	AMD Ryzen 9 3900	**Intel Xeon E5-2699 v4**	**AMD Threadripper 3990x**
Microarquitetura	Zen 2	Broadwell	Zen 2
Número de núcleos	12	44	64
Número de threads	24	88	128
Cache L3 (total)	64MB	55MB	256MB
Memória RAM	32GB	256GB	64GB
Nome	AMD24	Intel88	AMD128

na Figura 1. Estas aplicações são representativas para o nosso experimento pois possuem diferentes características com relação ao acesso à memória principal e uso do processador (IPC médio): aplicações com baixo/médio/alto IPC e aplicações com baixo/médio/alto número de *misses* na cache L3. Os dados para esta classificação foram retirados diretamente dos contadores de *hardware* através do AMDuProf e do Intel PCM.

4.2. Ambiente de Execução

Os experimentos foram realizados em três arquiteturas multicores, conforme mostrado na Tabela 1: AMD Ryzen 9 3900X, Intel Xeon E5 2640-v2[1] e AMD Ryzen Threadripper 3990x. Nós usamos o Sistema Operacional Linux Ubuntu com *kernel* v. 5.6.0. Cada aplicação foi compilada com GCC/G++ 10.2.0, usando a *flag* de otimização -O3. Cada aplicação foi executada com o *governor* do DVFS (*dynamic voltage and frequency scaling*) configurado para *ondemand*, onde a frequência de operação do processador é ajustada de acordo com a carga de trabalho.

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 através da função do OpenMP `omp_get_wtime`. Por outro lado, o consumo de energia foi obtido diretamente dos contadores de *hardware* presentes nos processadores modernos. No caso do processador Intel Xeon, a biblioteca *Running Average Power Limit* (RAPL) foi usada [Hähnel et al. 2012], enquanto que a biblioteca *Application Power Management* foi usada para o processador AMD Ryzen [Hackenberg et al. 2013]. Os resultados apresentados na Seção 5 são uma média de dez execuções com desvio padrão menor que 0.5%.

5. Resultados

Nesta seção, nós apresentamos e discutimos os resultados obtidos através dos experimentos realizados. Inicialmente, mostramos na Tabela 2 as melhores configurações encontradas para as principais regiões paralelas de cada aplicação (i.e., aquelas regiões que possuem tempo de execução acima de 0.01 segundos ou que executam mais de 1000 vezes) através de uma busca exaustiva. Esta busca considera a execução de cada região paralela com as possíveis configurações de número de *threads*, política de *placement* e afinidade. Cada configuração está organizada como segue: Número de *Threads* - *Placement* - Afinidade. Por exemplo, a melhor configuração encontrada para a execução do benchmark NB no sistema AMD24 é utilizando 12 *threads*, *placement* igual à *Cores* e afinidade igual à *Spread*. Com este experimento, queremos mostrar que não existe uma única configuração capaz de entregar o melhor resultado para todas as aplicações ao mesmo tempo.

Conforme podemos observar na Tabela 2, não existe uma única configuração de grau de paralelismo (número de *threads*) e estratégias de alocação de *threads* capaz de atingir o melhor EDP ao mesmo tempo para todas as aplicações. Por exemplo, a melhor configuração encontrada para a aplicação NB executando no AMD128 é diferente

[1]Alguns experimentos deste trabalho utilizaram os recursos da infraestrutura PCAD, http://gppd-hpc.inf.ufrgs.br, no INF/UFRGS

da encontrada no processador Intel88. Isto se deve, principalmente, às características intrínsecas de cada arquitetura (i.e., hierarquia de memória e poder de processamento) e aplicação. Para as aplicações com baixo grau de TLP e alta taxa de comunicação entre as *threads* (e.g., JA, FFT, sp.C.x e STREAM), os melhores resultados são obtidos com um número de *threads* e estratégias de alocação que evitam a saturação do barramento *off-chip*.

Por outro lado, para aplicações com grau médio de TLP (e.g., mg.C.x e HPCG), o melhor resultado é atingido com uma configuração que reduz a contenção de *cache* e misses na memória cache *L3*. Por exemplo, a segunda região paralela da aplicação HPCG é melhor executada com a afinidade definida em "*close*", onde as *threads* são alocadas próximas umas das outras (22-S-C). Portanto, ao encontrar uma configuração ideal ou próxima do ideal para cada região paralela de uma aplicação OpenMP, o EDP de uma aplicação pode ser otimizado quando comparado à execução padrão de aplicações paralelas. Na maneira padrão, o número de *threads* é igual ao número máximo de *hardware threads* disponíveis no sistema e as estratégias de *placement* e afinidade a cargo do Sistema Operacional. No restante do texto, referimos a esta configuração como *baseline*.

Tabela 2. Configurações encontradas pela busca exaustiva para cada aplicação e arquitetura: (*Número de threads-placement-afinidade*)

	AMD24	AMD128	Intel88
NB	12-C-S	3-HWT-F	3-HWT-F
JA	3-C-T	2-C-S	9-HWT-F
sp.C.x	12-C-S, 3-HWT-F, 12-C-T, 3-HWT-F, 12-HWT-F	3-C-S, 2-HWT-F, 25-C-S, 8-HWT-F, 64-HWT-F	3-C-T, 9-HWT-F, 22-C-C, 9-C-T, 22-HWT-F
STREAM	3-C-S	32-C-S	9-S-T
FFT	12-C-S, 12-HWT-F	64-C-S, 64-HWT-F	22-S-S, 22-S-C
HPCG	12-C-C, 24-C-S, 12-HWT-F	12-HWT-F, 128-C-C, 64-HWT-F	6-S-C, 22-S-C, 22-HWT-F
bt.C.x	12-C-S, 12-C-S, 3-HWT-F, 12-HWT-F, 12-HWT-F	7-C-S, 128-C-T, 128-C-T, 20-C-C, 64-HWT-F	8-S-T, 88-C-S, 88-C-C, 22-HWT-F, 22-HWT-F
cg.C.x	12-HWT-F, 12-HWT-F, 3-HWT-F, 12-C-T, 24-HWT-F	64-HWT-F, 24-C-S, 64-HWT-F, 20-C-T, 128-HWT-F	22-HWT-F, 22-HWT-F, 22-C-T, 88-S-S, 88-C-T
mg.C.x	12-HWT-F, 24-HWT-F, 12-C-T, 24-C-T, 12-C-S	64-HWT-F, 128-C-S, 128-HWT-F, 64-C-T, 128-C-S	22-HWT-F, 22-C-T, 22-HWT-F, 88-S-S, 88-C-T
ft.C.x	4-C-T, 12-C-T, 24-C-S, 24-C-T, 12-C-S	64-HWT-F, 24-C-T, 20-C-S, 20-C-S, 64-C-C	88-HWT-F, 22-HWT-F, 22-C-T, 88-S-S, 22-HWT-F
LUD	3-C-S, 12-C-T	7-HWT-F, 24-C-S	11-HWT-F, 22-S-S
ua.C.x	24-C-C, 12-T-F, 3-C-S, 24-T-F, 3-C-S, 24-C-S, 12-C-T, 24-T-F	64-HWT-F, 64-C-C, 14-C-T, 20-C-S, 64-C-C, 128-HWT-F, 128-C-S, 128-C-T	22-C-S, 4-S-T, 6-S-C, 88-T-F, 22-C-T, 3-S-S, 6-S-T, 88-C-T
LULESH	3-T-F, 3-C-C, 12-T-F, 3-C-T, 3-C-T, 3-C-C, 3-C-C, 3-C-T	16-HWT-F, 24-C-C, 64-C-T, 20-C-C, 64-HWT-F, 8-HWT-F, 16-C-S, 24-C-C	22-C-T, 9-T-F, 22-T-F, 22-S-T, 22-T-F, 22-C-C, 3-T-F, 3-S-T

Placement: HWT (Threads), C (Cores) e S (Sockets);
Afinidade: F (False), T (True), M (Master), C (Close) e S (Spread);

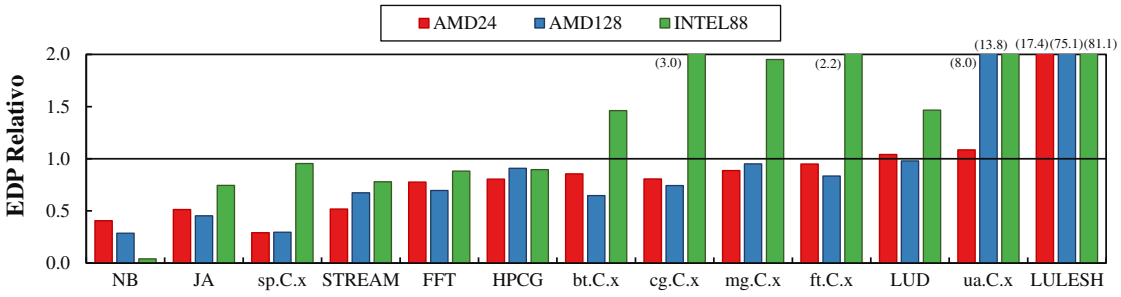


Figura 2. Resultados de EDP normalizado pelo *baseline*. Quanto menor o valor, melhor para *PampaAffinity*

Tabela 3. Precisão das configurações encontradas por *PampaAffinity* quando comparada à busca exaustiva,

	NB	JA	sp,C,x	STREAM	FFT	HPCG	bt,C,x	cg,C,x	mg,C,x	ft,C,x	LUD	ua,C,x	LULESH	Média Geométrica
AMD24	1,0	1,0	0,84	1,0	1,0	1,0	0,9	0,87	0,8	0,88	1,0	0,92	0,83	0,92
AMD128	1,0	1,0	0,69	1,0	1,0	0,8	0,6	0,75	0,7	0,66	1,0	0,78	0,66	0,80
Intel88	1,0	1,0	0,76	1,0	1,0	0,8	0,7	0,87	0,6	0,77	1,0	0,85	0,6	0,83

Considerando os resultados discutidos anteriormente, se justifica o uso de uma abordagem dinâmica e online que seja capaz de convergir para uma configuração ideal ou próxima do ideal para cada região paralela, com o objetivo de otimizar o EDP das aplicações paralelas. Assim, discutimos a seguir os resultados obtidos pelo emprego do *PampaAffinity*. A Figura 2 apresenta os resultados de EDP para todas as aplicações executadas em cada arquitetura alvo. O EDP é normalizado pelo *baseline* (representado pela linha preta), portanto valores abaixo de 1.0 significam que *PampaAffinity* é melhor que o *baseline*. Na maioria dos casos, *PampaAffinity* apresenta melhor EDP que o *baseline*. No melhor caso, EDP é otimizado em 70.9% no AMD24 (*sp.C.x*); 71.5% no AMD128 (*NB*); e 96.1% no Intel88 (*NB*). No entanto, para aplicações com características intrínsecas (e.g., *ua.C.x* e *LULESH*), *PampaAffinity* apresenta piores resultados que o *baseline*. Este comportamento é discutido em maiores detalhes ao final desta seção.

Com o objetivo de avaliar a precisão das configurações encontradas por *PampaAffinity*, nós comparamos os resultados obtidos pela ferramenta com o melhor resultado encontrado pela busca exaustiva (Tabela 2). Muito embora uma busca exaustiva irá sempre garantir o melhor resultado, ela exige a execução da mesma aplicação com todas as possíveis configurações, o que, naturalmente, irá aumentar o custo de aprendizado, se tornando ineficiente para ser usada durante o tempo de execução. A Tabela 3 apresenta a diferença nas configurações encontradas pelas duas abordagens para cada aplicação e arquitetura. Por exemplo, quando *PampaAffinity* foi capaz de encontrar a mesma solução para todas as regiões paralelas, a precisão é igual à 1.0. Portanto, quanto mais próximo de 1.0 é este valor, maior foi a precisão de *PampaAffinity*. Para a grande maioria das aplicações, *PampaAffinity* foi capaz de convergir para uma configuração similar a encontrada pela busca exaustiva, independente da arquitetura (e.g., NB, JA, STREAM, FFT, e LUD). Considerando a média geométrica das aplicações, pode-se observar que, quanto maior é o espaço de exploração e projeto, menor é a precisão do *PampaAffinity*, uma vez que o número de configurações a serem avaliadas aumenta significativamente.

No entanto, mesmo convergindo para configurações similares que a busca exaustiva, *PampaAffinity* apresentou piores resultados que o *baseline* nas aplicações *LUD*, *ua.C.x*, *LULESH*, independente da arquitetura alvo. Isto significa que o sobrecusto envolvido na troca das configurações (grau de paralelismo e estratégias de alocação e afinidade de *threads*) entre as regiões paralelas foi maior que o ganho obtido por utilizar a configuração ideal ou próxima da ideal para cada região paralela. Para identificar a origem deste sobrecusto, a Figura 3 apresenta dados coletados de contadores de *hardware* com relação a migração de *threads* entre CPUs durante a execução. Os valores são normalizados pela execução do *baseline*. Portanto, valores maior que 1.0 significa que *PampaAffinity* gerou maior número de migrações de *threads*. Neste sentido, os piores resultados de *PampaAffinity* ocorreram quando houve um número significativamente maior de migração de *threads* do que o *baseline*. Este comportamento acaba impactando também na quantidade de *misses* nos níveis de memória *cache* próxima do núcleo de execução, com influência negativa no desempenho e consumo de energia.

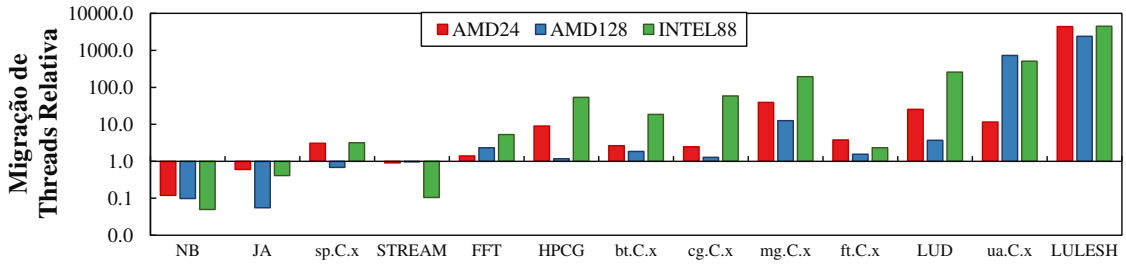


Figura 3. Número de migrações de *Threads* entre CPUs para cada aplicação. Resultado é normalizado pelo *baseline*. Quanto menor o valor, menor o número de migrações causadas por *PampaAffinity*

6. Conclusão

Nós apresentamos *PampaAffinity*, uma abordagem online capaz de encontrar uma configuração ideal ou próxima do ideal de grau de paralelismo e estratégias de mapeamento de *threads* que otimiza o EDP de aplicações paralelas. Através do algoritmo de busca *online*, *PampaAffinity* é: transparente para o usuário, no sentido que não exige modificações de código nem recompilação; e automático, no sentido que ajusta a configuração ideal para cada região paralela de maneira automática. Por conta disto, *PampaAffinity* atinge uma precisão média de 85% para a execução de treze aplicações em três arquiteturas multicore, sendo capaz de otimizar o EDP em até 96.1% quando comparado à maneira padrão que aplicações paralelas são executadas. Como trabalhos futuros, nós pretendemos ajustar o algoritmo de aprendizado para considerar o efeito negativo das migrações de *threads* entre CPUs quando diferentes políticas de mapeamento são selecionadas entre regiões paralelas vizinhas.

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 & summary and preliminary results. In *ACM/IEEE SC*, pages 158–165, USA. ACM.

Broquedis, F., Aumage, O., Goglin, B., Thibault, S., Wacrenier, P.-A., and Namyst, R. (2010). Structuring the execution of openmp applications for multicore architectures. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–10. IEEE.

Chapman, B., Jost, G., and Pas, R. v. d. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press.

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.

Cruz, E. H., Diener, M., and Navaux, P. O. (2012). Using the translation lookaside buffer to map threads in parallel applications based on shared memory. In *IEEE International Parallel and Distributed Processing Symposium*, pages 532–543. IEEE.

Cruz, E. H. M., Diener, M., Pilla, L. L., and Navaux, P. O. A. (2016). Hardware-assisted thread and data mapping in hierarchical multicore architectures. *ACM Trans. Archit. Code Optim.*, 13(3).

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.

De Sensi, D., Torquati, M., and Danelutto, M. (2016). A reconfiguration algorithm for power-aware parallel applications. *ACM Transactions on Architecture and Code Optimization*, 13(4):1–25.

Diener, M., Cruz, E. H., and Navaux, P. O. (2013). Communication-based mapping using shared pages. In *IEEE International Parallel and Distributed Processing Symposium*, pages 700–711. IEEE.

Eichenberger, A. E., Terboven, C., Wong, M., and an Mey, D. (2012). The design of openmp thread affinity. In *International Workshop on OpenMP*, pages 15–28. Springer.

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 *ISPASS*, pages 194–204.

Hähnel, M., Döbel, B., Völp, M., and Härtig, H. (2012). Measuring energy consumption for short code paths using rapl. *SIGMETRICS Perf. Evaluation Rev.*, 40(3):13–17.

Joao, J. A., Suleman, M. A., Mutlu, O., and Patt, Y. N. (2012). Bottleneck identification and scheduling in multithreaded applications. In *ASPLOS*, pages 223–234, NY, USA. ACM.

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.

Lorenzon, A. F., De Oliveira, C. C., Souza, J. D., and Beck, A. C. S. (2018). Aurora: Seamless optimization of openmp applications. *IEEE Transactions on Parallel and Distributed Systems*, 30(5):1007–1021.

OpenMP Architecture Review Board (2018). OpenMP api specification: Version 5.0.

Papadimitriou, G., Chatzidimitriou, A., and Gizopoulos, D. (2019). Adaptive voltage/frequency scaling and core allocation for balanced energy and performance on multicore cpus. In *IEEE International Symposium on High Performance Computer Architecture*, pages 133–146. IEEE.

Schwarzrock, J., de Oliveira, C. C., Ritt, M., Lorenzon, A. F., and Beck, A. C. S. (2021). 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.

Sridharan, S., Gupta, G., and Sohi, G. S. (2014). Adaptive, efficient, parallel execution of parallel programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 169–180.

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. *ACM Sigplan Notices*, 43(3):277–286.

Wang, W., Davidson, J. W., and Soffa, M. L. (2016). Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale numa machines. In *IEEE International Symposium on High Performance Computer Architecture*, pages 419–431. IEEE.