Assessing Parallel Thread Mapping Approaches on Shared Memory SMT Architectures

A. Amorim, and H. Freitas, *Member, IEEE*

***Abstract*—For better performance in multi-core architectures, when running parallel applications, thread mapping has become a relevant task. One way to explore this approach is to consider the memory hierarchy and the existing communication between threads. However, thread mapping is an NP-Hard problem and, therefore, heuristics have been studied to find solutions that are closer to optimal. In this context, the goal of this work is to evaluate the use of the BRD, Greedy and K-means heuristics for mapping threads in multi-core architectures with shared memory and simultaneous multithreading (SMT) support. All heuristics present performance improvements. The results show gains of up to 40.77% (with SMT) in execution time and 46.81% (with SMT) in power consumption when compared to the default Linux mapping strategy. Concerning all scenarios, SMT represents 72.22% of results with better execution time and 44.44% of results with better energy consumption.**

***Index Terms*—Thread mapping, Static mapping, Shared memory, Memory hierarchy, Simultaneous multithreading.**

I. Introdução

EM referência aos processadores single-core, o paralelismo baseado em fluxos de instruções (*threads*) surge como alternativa para aumentar desempenho com foco em vazão de *threads* [1], [2]. A tecnologia SMT (*Simultaneous Multithreading*) consiste em um único núcleo executar simultaneamente, em seu pipeline superescalar, mais de uma *thread* (fluxo de instruções), fazendo melhor uso dos recursos. Para tanto, é necessário que haja para cada *thread* em execução simultânea um conjunto de registradores, que consiste em registros de dados, status e controle [3]. Portanto, a ilusão de mais de um núcleo advindo do suporte SMT pode impactar na melhoria de desempenho dos processadores multi-core, que surgiram como alternativa decorrente da melhor exploração da integração de transistores, reduzindo as restrições de evolução presentes na geração single-core anterior.

O ganho de desempenho em processadores multi-core é devido principalmente a execuções paralelas das aplicações [4], fazendo com que mais de uma *thread* possa ser executada ao mesmo tempo por diversos núcleos. Em execuções de aplicações paralelas as *threads* muitas vezes precisam se comunicar. A comunicação pode ocorrer de forma implícita por memória compartilhada ou explícita por troca de mensagens.

Em arquiteturas multi-core, é muito comum o compartilhamento de memória cache, e.g. L2 (Fig. 1), por mais de um núcleo. Quando o espaço de endereçamento é compartilhado entre as *threads*, o compartilhamento tende a ser vantajoso, aumentando o *cache hit*. No entanto, a exploração da localidade espacial e temporal pode ser prejudicada caso *threads* trabalhem em espaços de endereços diferentes, aumentando o *cache miss*. Fato é que, os núcleos que compartilham dados (i.e. *cache hit*) em cache de mesmo nível se beneficiam da localidade espacial e temporal dos dados e, portanto, comunicam mais rápido. A falta de um determinado dado na cache compartilhada e mais próxima do núcleo prejudica o desempenho do programa em execução, pois há um sobrecusto da latência de acesso aos níveis mais distantes na hierarquia de memória [5]. Para evitar tais problemas, é importante mapear *threads* que se comunicam mais em núcleos mais próximos, que compartilham memória cache de mesmo nível e que, preferencialmente, possuem o menor tempo de acesso possível. O mapeamento de *threads* [6] pode ser realizado de forma estática ainda na fase de projeto da arquitetura ou aplicação, ou de forma dinâmica durante a execução da aplicação.

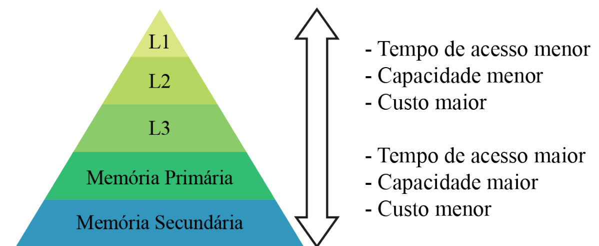


Fig. 1. Hierarquia de memória.

O problema de mapeamento de *threads* é considerado NP-Difícil, pois a solução ótima só pode ser garantida após testar todos os mapeamentos possíveis, ou seja, todas as permutações das *threads* nos núcleos. O custo de realizar permutações é O(n!), sendo n a quantidade de *threads*. Por isso, heurísticas que fornecem soluções aproximadas são pesquisadas para tratar tais problemas [7].

Desse modo, é importante saber qual a relação existente entre as características das heurísticas de mapeamento e as características das aplicações a serem mapeadas, que refletem em uma melhoria de desempenho. Sendo assim, o objetivo deste trabalho é avaliar o uso de diferentes heurísticas de mapeamento de *threads* levando em consideração a quantidade de dados compartilhados entre cada par de *threads*, a arquitetura dos processadores, a hierarquia de memória e a utilização ou não do suporte SMT. Logo, a principal contribuição deste trabalho é a análise do uso de duas heurísticas que mapeiam as *threads* ao avaliar o problema com granularidade fina (algoritmos Guloso e Biparticionamento Recursivo Dual -

Amanda Amorim é Mestre em Informática pelo Programa de Pós-graduação em Informática, PUC Minas, Belo Horizonte, MG, 30535-901 BR e-mail: amanda.amorim@sga.pucminas.br.

Henrique Freitas é professor do Programa de Pós-graduação em Informática, PUC Minas, Belo Horizonte, MG, 30535-901.

BRD), e outra com granularidade grossa (algoritmo K-means), ao mapear *threads* de aplicações com compartilhamento de dados homogêneo (compartilhamento balanceado entre todos os pares de *threads*) e heterogêneo (desbalanceado). Além disso, na literatura não foi encontrado o uso do *K-means* para mapear *threads* no contexto de memória compartilhada.

II. TRABALHOS CORRELATOS

No trabalho realizado por He et al. [8], os autores propõem o NestedMP, um conjunto estendido de diretivas para aninhamento de *threads* em OpenMP com o objetivo de reduzir o impacto de mapeamento sub-ótimo de *threads* em núcleos. Os resultados apresentaram melhoria significativa no OpenMP para o compilador GCC. Em Liu et al. [9], os autores propõem realizar o mapeamento de *threads* com foco na quantificação da distância de núcleos em uma arquitetura com rede de interconexão em anel. Os resultados apresentaram em média 25% de ganho em relação ao mapeamento padrão do Linux. Dey et al. [10] propõem o ReSense, um sistema *runtime* que usa características da aplicação para que dinamicamente seja realizado o mapeamento. Os resultados apresentaram até 46,56% melhor *throughput* em relação ao sistema operacional nativo. Já Castro et al. [11], apresentam um abordagem de mapeamento de *threads* para aplicações de memória transacional com uso de aprendizado de máquina. Os resultados alcançaram uma melhoria de até 6,37% em comparação com o padrão de mapeamento do Linux.

Cruz, Diener e Navaux [12] elaboraram uma revisão da literatura além de uma discussão sobre resultados obtidos pelos autores que apontam melhorias nas execuções de aplicações científicas. Em trabalho do mesmo grupo [6], é proposto um mapeamento baseado na afinidade entre *threads*. O problema de encontrar as *threads* que compartilham mais memória é resolvido com o algoritmo Edmonds. Os resultados foram obtidos por meio de execuções em uma máquina *Uniform Memory Access* (UMA) com cache L2 compartilhada. O *NAS Parallel Benchmark* (NPB) [13] foi utilizado com os tamanhos W e A, sendo os traços de acessos à memória gerados com o tamanho W. Houve um ganho máximo de 42% ao comparar o mapeamento proposto com mapeamento padrão do Linux. Em Cruz et al. [14], essa mesma metodologia foi aplicada em duas máquinas *Non-Uniform Memory Access* (NUMA) com cache L3 compartilhada. O NPB foi utilizado com tamanhos W, A e B. Os resultados chegaram a 75% de ganho de desempenho em relação ao mapeamento padrão do Linux.

Em Diener et al. [15], são estudadas duas estratégias para o mapeamento estático: i) a heurística Guloso que mapeia pares de *threads* em pares de núcleos que compartilham cache, e um ii) algoritmo que faz busca exaustiva pela melhor solução. Segundo os autores a solução exaustiva só é viável em tempo de execução para mapear até 16 *threads*. As avaliações dos resultados são feitas em relação ao tempo de execução em duas máquinas. Uma composta por dois processadores com compartilhamento da cache L3, cada núcleo 2-SMT, neste caso há compartilhamento da cache L2. A outra máquina possui um processador com quatro núcleos 8-IMT. Segundo os autores, eles atingiram até 45% de redução no tempo de execução em relação a solução padrão do Linux em alguns casos.

Uma estratégia de mapeamento de *threads* no contexto de processamento de dados de *stream* é proposto em Zheng et al. [16]. Para gerar os mapeamentos foram utilizadas informações como: compartilhamento de dados entre as *threads*, obtidos por meio do monitoramento de acessos as caches; e contenção de cache, inferida pelas taxas de faltas geradas por *thread* ao último nível de memória cache. O algoritmo de mapeamento possui uma combinação de estratégias como Guloso e particionamento de grafos da ferramenta SCOTCH [17]. Para avaliar os resultados foram usadas três máquinas: a primeira com 4 núcleos do tipo UMA, a segunda com 8 núcleos UMA e por último uma com 8 núcleos NUMA. Os resultados mostram uma melhora de desempenho de até 1.8x em relação ao mapeamento padrão do sistema operacional.

Em Diener et al. [18], é realizado mapeamento de dados e *threads* estático e dinâmico com base em traces de memória gerados com a ferramenta *Pin Dynamic Binary Instrumentation*. O mapeamento estático foi feito com o algoritmo BRD [17]. Nos experimentos, foi utilizada uma máquina com quatro processadores (8 núcleos, 2-SMT). O ganho máximo de desempenho obtido foi de 54% (20% em média) e redução do consumo de energia de até 37% (11% em média).

Sendo assim, a contribuição deste trabalho em relação aos correlatos está em reunir em uma única pesquisa a avaliação de três abordagens para mapeamento de *threads* no contexto de memória compartilhada com ou sem SMT, segundo o tempo de execução e o consumo energético das execuções das aplicações do NPB. Foram utilizadas as abordagens: Dividir para Conquistar da heurística BRD de Pellegrini [17], o Guloso implementado na heurística de mesmo nome por Oliveira et al. [19] e Avelar et al. [20], e a estratégia de clusterização *K-means* adaptada em uma heurística de mapeamento por Avelar [20]. Essas duas últimas foram desenvolvidas em trabalhos anteriores do grupo de pesquisa, mas no contexto de comunicação explícita por troca de mensagens. Na literatura, não foi encontrado o uso do algoritmo *K-means* para mapeamento de *threads* que se comunicam por memória compartilhada, tal como apresentado neste trabalho.

III. HEURÍSTICAS DE MAPEAMENTO DE THREADS

Nesta seção são apresentados os três algoritmos de mapeamentos usados neste trabalho. As entradas dos algoritmos são: um grafo que representa a arquitetura alvo; e um grafo que representa o compartilhamento de dados entre as *threads* das aplicações, no qual os vértices representam as *threads* e as arestas valoradas a quantidade de dados compartilhados entre cada par de vértices.

Os algoritmos de mapeamento foram classificados em algoritmos que analisam o problema com granularidade fina ou grossa. Com granularidade grossa as *threads* e os núcleos de processamento são avaliados e mapeados em grupos. Já com granularidade fina as suas características são avaliadas de forma individual, mapeando uma *thread* por vez.

A. Algoritmo Biparticionamento Recursivo Dual - BRD

O algoritmo de mapeamento BRD é baseado na abordagem de divisão e conquista, mapeando de forma recursiva as

threads nos núcleos de processamento. Ele está implementado na ferramenta Scotch [17]. O algoritmo recursivamente divide os núcleos de processamento em dois conjuntos e com base nos conjuntos formados divide as *threads* também em dois grupos. Após só restar um núcleo no conjunto, as *threads* do grupo associado a ele são mapeadas nesse núcleo de processamento. Sendo assim, essa heurística pode ser classificada como uma abordagem que analisa o problema com granularidade fina, dado que o problema é dividido em subproblemas menores até haver somente um núcleo para receber o subconjunto de *threads* associado a ele.

B. Algoritmo Guloso

A estratégia utilizada em algoritmos gulosos consiste em sempre escolher a melhor solução local, não garantindo que a solução global será ótima. O algoritmo usado neste trabalho foi implementado por Avelar [20] com base no proposto por Oliveira et al. [19]. Essa implementação foi feita para mapear *threads* em arquiteturas onde os núcleos se comunicam por meio de uma rede formada por links e roteadores. Para mapear as *threads* em arquiteturas com compartilhamento de dados em hierarquia de memória foi necessária uma adaptação no algoritmo.

Na adaptação feita para este trabalho, é escolhida a *thread* que mais compartilha dados com as outras, e é mapeada no núcleo que está conectado a unidade de memória, que está conectada com o maior número de núcleos e/ou unidades de memórias (vértice do tipo unidade de memória com maior grau). Nos passos seguintes, escolhe-se a *thread* ainda não mapeada que mais compartilha dados com a última *thread* mapeada e a associa ao núcleo, ainda não usado, mais próximo do último núcleo mapeado. Assim como a heurística BRD, essa heurística também pode ser classificada como uma abordagem que analisa o problema com granularidade fina, dado que é avaliada e mapeada *thread* por *thread* a núcleo por núcleo.

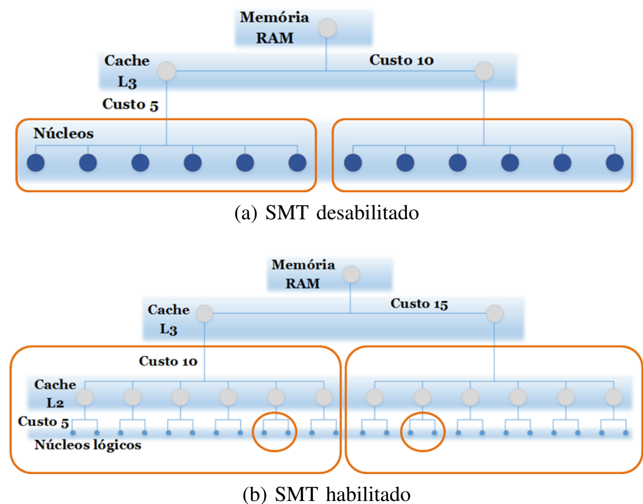
C. Algoritmo K-means

O *k-means* é um algoritmo de clusterização que foi adaptado por Avelar [20], para mapear processos em núcleos em redes-em-chip com passagem de mensagem. A estratégia baseia-se em agrupar os processos que se comunicam mais (neste trabalho, *threads* que compartilham mais dados) em *clusters* de núcleos de processamento. Neste trabalho, esses *clusters* de núcleos são os grupos de núcleos que compartilham alguma unidade de memória. O agrupamento das *threads* é feito baseado na distância euclidiana mínima. Assim como os outros algoritmos usados, essa abordagem é uma heurística, não garantindo a solução ótima.

As entradas do algoritmo são: o grafo de compartilhamento de dados entre as *threads*, a quantidade de *clusters* desejado (k), e a distância mínima de compartilhamento de dados (d) entre as *threads* de cada *cluster*. Inicialmente, o algoritmo distribui as *threads* de forma aleatória entre os k *clusters* e calcula os centroides dos *clusters* formados. Eles são calculados por meio da média entre o compartilhamento de dados das *threads* de cada *cluster*. Posteriormente, as *threads* são redistribuídas entre os *clusters* com base na distância euclidiana mínima entre as *threads* e os novos centroides são calculados. Esses passos se repetem enquanto as distâncias euclidianas forem maiores que d e pelo menos uma *thread* mudar de *cluster*.

Os *clusters* formados pelo *k-means* podem ter a quantidade de *threads* heterogênea, não sendo *clusters* de *threads* possíveis de serem mapeadas em grupos de núcleos de processamento de forma balanceada, pois alguns núcleos teriam que receber mais *threads* que outros. Este problema foi modelado em um grafo bipartido valorado, sendo: os vértices do grupo A as *threads*, os vértices do grupo B os *clusters*, e o peso das arestas as distâncias das *threads* aos *clusters*, e resolvido aplicando o algoritmo de *matching* mínimo em um grafo bipartido valorado proposto por Bertsekas [21].

Para mapear as *threads* na arquitetura alvo deste trabalho com SMT desabilitado (Fig.2a), o *k-means* foi executado para formar 2 *clusters*, onde cada *cluster* é composto por 6 *threads* que compartilham as caches L3. Já para mapear as *threads* na arquitetura com SMT habilitado (Fig.2b) foram utilizadas duas estratégias diferentes, gerando assim dois possíveis mapeamentos. A primeira estratégia consiste em dividir as *threads* em 12 *clusters* com 2 *threads* cada (compartilhamento das caches L2), calcular a matriz de compartilhamento de dados entre os 12 *clusters* formados, e agrupar os 12 *clusters* em 2 *clusters* de 6 *clusters* cada (compartilhamento das caches L3). A segunda estratégia consiste em agrupar as *threads* primeiro de acordo com o compartilhamento da L3, 2 *clusters* de 12 *threads* cada, e então, dividir cada *clusters* de 12 *threads* em 6 *clusters* de 2 *threads* que compartilham a L2.



(a) SMT desabilitado

(b) SMT habilitado

Fig. 2. Grafos das arquiteturas alvo.

Diferentemente das outras duas heurísticas, a heurística *K-means* pode ser classificada como uma abordagem que analisa o problema com granularidade mais grossa, dado que a análise para a redistribuição das *threads* nos *clusters* é feita com base na média da quantidade de dados compartilhados entre as *threads* pertencentes a cada *cluster*.

IV. Metodologia

Para atingir o objetivo deste trabalho seguiu-se a metodologia apresentada nesta seção. Primeiro, foram selecionadas

como cargas de trabalho os programas do *benchmark NPB*, sendo eles: *Block Tri-diagonal* (BT), *Conjugate Gradient* (CG), *Embarrassingly Parallel* (EP), *Fast Fourier Transform* (FT), *Integer Sort* (IS), *Lower-Upper Gauss-Seidel* (LU), *Multi-Grid* (MG), *Scalar Penta-diagonal* (SP) e *Unstructured Adaptive* (UA) [13]. A versão selecionada do NPB foi a 3.3, com uso da biblioteca *OpenMP*, com entradas da classe W e A. As aplicações do NPB versão 3.3, possuem sete tamanhos de cargas, esses tamanhos recebem o nome de classes de problemas [22]. A classe W foi selecionada porque, apesar de ser considerada pequena, ela representa problemas executados em computadores de estação de trabalho, e possui um tamanho viável para gerar e processar os traces de acesso à memória, necessários para a metodologia deste trabalho. Já a classe A representa problemas executados em computadores paralelos. Esses programas foram usados com escalonamento estático de dados a fim de garantir que cada *thread* trabalhe sempre com os mesmos blocos de iterações.

A escolha da máquina utilizada nos experimentos foi guiada por dois critérios. Sendo eles, conter hierarquia de memória cache em dois grupos de processadores distintos e suportar a tecnologia SMT. Dessa forma, a máquina utilizada é composta por dois processadores Intel(R) Xeon(R) CPU E5-2620 v2. Esses processadores possuem 6 núcleos de processamento com cache L1 (32kB de instruções e 32kB de dados) e L2 (256kB/núcleo - totalizando 1,5MB) privadas a cada núcleo e cache L3 (15MB) compartilhada. Com a tecnologia SMT habilitada a máquina passa a suportar 24 *threads*, com compartilhamento de cache L2 a cada par de *threads* e compartilhamento da cache L3 pelas 12 *threads* executadas em cada processador. Para acessar os dados na cache L2 o tempo é inferior ao gasto para acessá-los na cache L3 que é, por sua vez, inferior a acessá-los na memória RAM. Para os algoritmos de mapeamento usados neste trabalho, a arquitetura foi modelada em grafos (Fig. 2). Os vértices representam os núcleos de processamento e as unidades de memória, e as arestas representam a conexão existentes entre eles.

O Algoritmo *K-means* é um algoritmo de clusterização. Portanto, é importante observar na Fig. 2a que os núcleos da arquitetura formam dois grupos distintos que compartilham cache L3. Quando a tecnologia SMT está habilitada, Fig. 2b, além dos *clusters* que compartilham cache L3, há doze *clusters* que compartilham cache L2.

A ferramenta que gera os traces de acesso à memória principal criada e utilizada por Alves [23], com base no *framework* Pin da Intel [24], foi usada neste trabalho para gerar os traces de acesso à memória dos programas do NPB. Os traços coletados são armazenados em N arquivos, sendo N a quantidade de *threads* usadas na execução do programa. Cada arquivo possui as informações de acessos à memória realizados pela sua respectiva *thread*. Os dados utilizados foram: o tipo de acesso na memória, leitura (r) ou escrita (w); o tamanho da operação de memória e o endereço inicial dos dados acessados. Os traces coletados foram processados de modo a contabilizar a quantidade de dados compartilhados entre cada par de *threads*. Para tanto, foi implementada uma tabela *hash* com lista encadeada onde cada registro representa um endereço de memória e possui quantas vezes ele foi acessado por *thread*. Após todos os traces terem sido lidos e as informações armazenadas na tabela *hash*, os registros da tabela foram lidos para gerar a matriz de compartilhamento. A matriz foi preenchida segundo a Eq. 1, apresentada a seguir:

$$M_{(x,y)} = \sum_{i=0}^{Q} \min\{R_i[x], R_i[y]\} \quad (1)$$

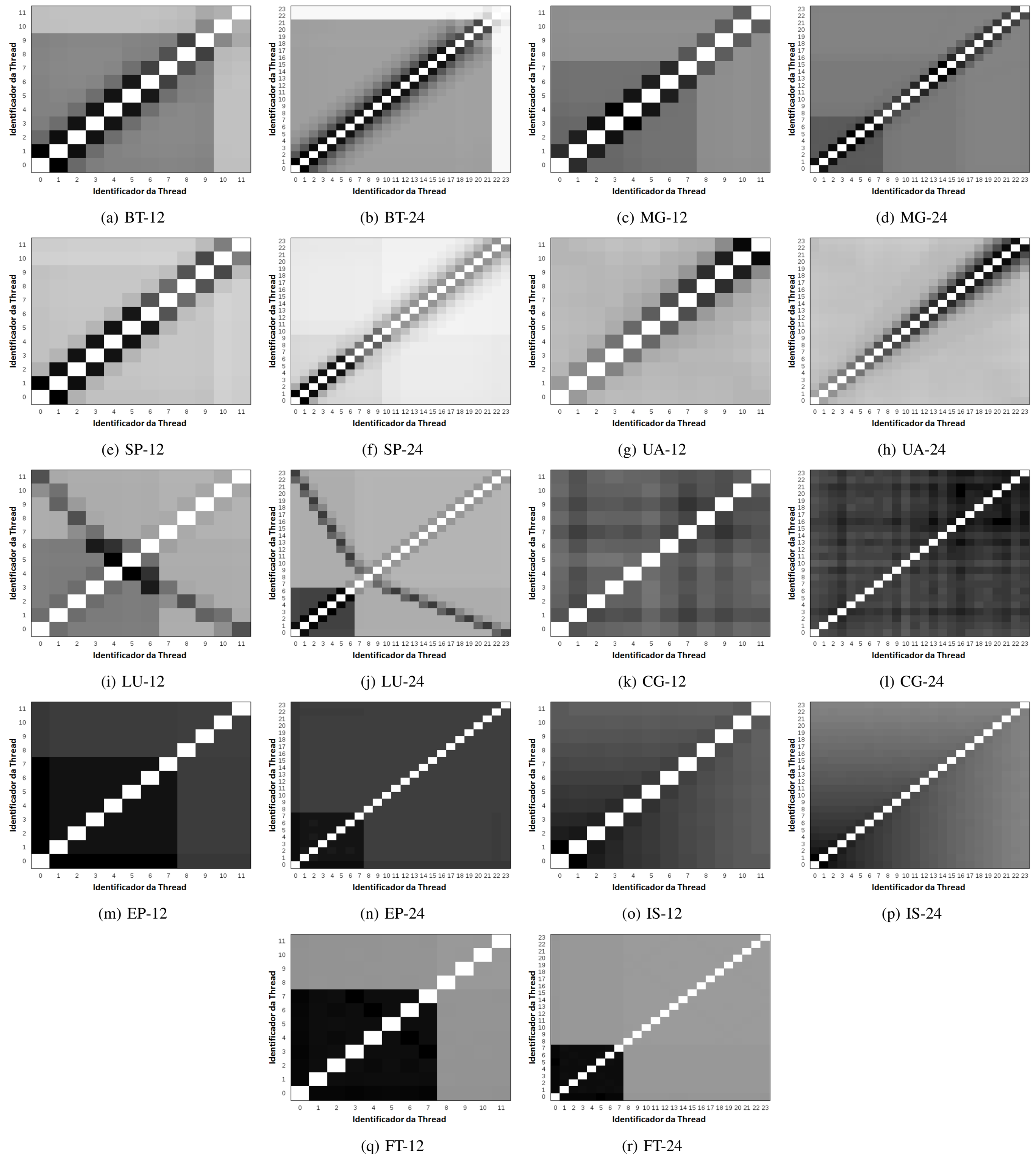
(i) M é a matriz de compartilhamento entre as *threads*.
(ii) x e y são os índices da matriz M variando de 0 a N, N sendo o número de *threads*.
(iii) R são vetores de acessos aos endereços de memória, um para cada endereço, onde cada índice desses vetores representa uma *thread*.
(iv) i percorre cada um desses registros variando de 0 a Q, onde Q é a quantidade de endereços acessados pelos programas.

Os resultados apresentados neste artigo são as médias de 50 execuções e intervalo de confiança, cujo nível de confiança é de 95%, das cargas de trabalho do NPB, com tamanhos das classes W e A, mapeados com as heurísticas BRD, Guloso e *K-means*. Os mapeamentos das aplicações com tamanhos da classe A foram gerados com base nas matrizes de compartilhamentos das mesmas com tamanho da classe W. Essa medida foi necessária devido ao tamanho dos traces de acesso à memória gerados para a classe W e a quantidade de memória RAM necessária para processá-los ser inviável para gerar as matrizes de classes maiores. Os mapeamentos para a classe A foram gerados sobre a hipótese das aplicações manterem os mesmos padrões de compartilhamento de dados, ao aumentarem a quantidade de dados processados. Os resultados dos mapeamentos gerados pelas heurísticas são comparados com a estratégia padrão do Linux com 12 *threads*. Nos gráficos, não são apresentados os resultados da estratégia padrão do Linux com 24 *threads*, SMT habilitado, porque eles são consideravelmente piores que os outros, atrapalhando assim, a visibilidade dos demais resultados nos gráficos. A única exceção são as execuções da carga EP.

V. Resultados e Discussão

Conforme explicados na Seção IV, nesta seção os resultados são apresentados em forma de gráfico das médias dos tempos de execuções e dos consumos de energia, além dos respectivos intervalos de confiança, sendo um gráfico para cada aplicação. Com o objetivo de ajudar na análise dos gráficos, também são apresentadas tabelas com as porcentagens de ganho de desempenho em relação a execução padrão do Linux (Tab. I, II, III e IV). Essas tabelas possuem as melhores porcentagens em cada cenário considerando os intervalos de confiança. Já os melhores resultados desconsiderando os intervalos de confiança estão em negrito.

Os compartilhamentos de dados entre as *threads* nas execuções do NPB, classe W, são ilustrados na Fig. 3, com 12 e 24 *threads*. Os tons de cinzas mais escuros representam mais comunicações entre os respectivos pares de *threads*. É importante ressaltar que a escala de cinza usada nas matrizes de cada aplicação é diferente, sendo assim, a intensidade de comunicação não pode ser comparada entre as matrizes.



(a) BT-12 (b) BT-24 (c) MG-12 (d) MG-24

(e) SP-12 (f) SP-24 (g) UA-12 (h) UA-24

(i) LU-12 (j) LU-24 (k) CG-12 (l) CG-24

(m) EP-12 (n) EP-24 (o) IS-12 (p) IS-24

(q) FT-12 (r) FT-24

Fig. 3. Matrizes de compartilhamento de dados entre duas *threads* dos programas do NPB, classe W, executados com 12 e 24 *threads*.

Com base nas matrizes de compartilhamento de dados entre as *threads* de cada aplicação estudada, as mesmas foram classificadas como aplicações com padrão de compartilhamento de dados homogêneo (compartilhamento balanceado entre todos os pares de *threads*) ou heterogêneo (desbalanceado). Para as cargas que possuem compartilhamento de dados heterogêneos, BT (Fig. 3a e 3b), MG (Fig. 3c e 3d), SP (Fig. 3e e 3f) e UA (Fig. 3g e 3h) concentrados entre pares de *threads* adjacentes e LU (Fig. 3i e 3j) entre pares de *threads* mais distantes, as heurísticas BRD e Guloso apresentaram, de modo geral, melhores resultados em tempo de execução (BT, MG, LU, SP e UA, Fig. 4, 5, 6, 7 e 8, respectivamente), sendo

que, em algumas, os melhores resultados foram sem utilizar SMT (12 *threads*) e outras ao utilizá-lo, tendo escalabilidade para 24 *threads*. A carga SP, mesmo com a classe A, não apresentou escalabilidade para 24 *threads*. Isso ocorreu pois o compartilhamento de dados nesta carga é mais intenso entre as 11 primeiras *threads* tanto nas execuções com 12 *threads* (Fig 3e), quanto nas com 24 *threads* (3f). Fazendo com que as heurísticas de mapeamento para 24 *threads* concentrassem essas 11 *threads* em um único processador, com 6 núcleos físicos e 12 lógicos, sobrecarregando-o. Já para as cargas homogêneas EP (Fig. 9) e IS (Fig. 10) com as classes W e A, todas as heurísticas estudadas para as execuções com SMT tiveram bons resultados, o que indica que o melhor desempenho pode estar associado às execuções com SMT (24 *threads*) e estáticas, já que os mapeamentos influenciam menos, dado que as comunicações são mais uniformes entre todos os pares de *threads* (Fig. 3m, 3n, 3o e 3p). Para a carga CG (Fig. 11), que possui compartilhamento de dados mais homogêneo, com leve desbalanceamento em relação a alguns pares de *threads* (Fig. 3k e 3l), as heurísticas estudadas para mapear execuções com SMT (24 *threads*) foram melhores quando a aplicação é executada com a classe W. Já com a classe A, que processa mais dados, a heurística *K-means-L3-L2* com SMT foi melhor que as outras. Isto é um indício de que, para cargas com compartilhamentos de dados mais distribuídos e sem um padrão, essa heurística pode ser melhor que as outras. Para a carga FT (Fig. 12), que apresentou um maior desbalanceamento de dados, porém homogêneo entre as 8 primeiras *threads* (Fig. 3q, 3r), com a classe W, as heurísticas estudadas não conseguiram superar o desempenho do padrão do Linux sem SMT. Já com a classe A, a carga teve escalabilidade para as execuções com SMT (24 *threads*) apresentando melhores tempos de execução com as heurísticas Guloso e *K-means-L2-L3*. Com SMT é possível que as oito primeiras *threads* sejam executadas em um mesmo processador. Por esse motivo, as heurísticas estudadas obtiveram melhores resultados nas execuções com 24 *threads*. Com a classe W, os resultados foram equivalentes com o padrão Linux com 12 *threads*, porque são processados poucos dados. Mas com a classe A, os tempos de execução dos mapeamentos das heurísticas foram menores.

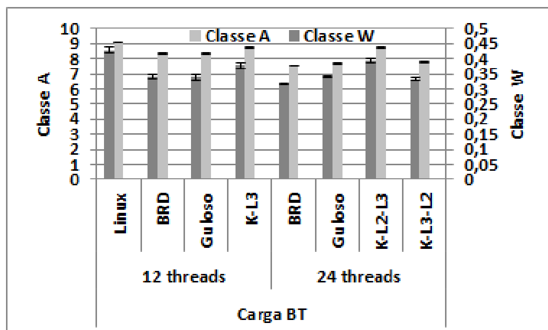


Fig. 4. Tempo de Execução (s) – Carga BT.

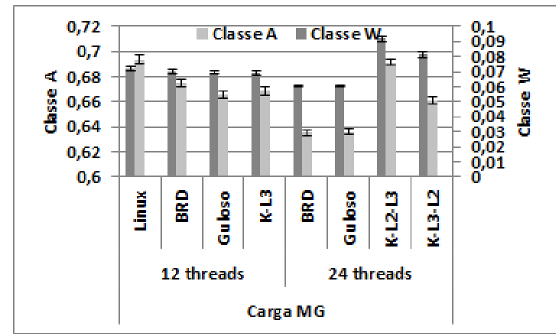


Fig. 5. Tempo de Execução (s) – Carga MG.

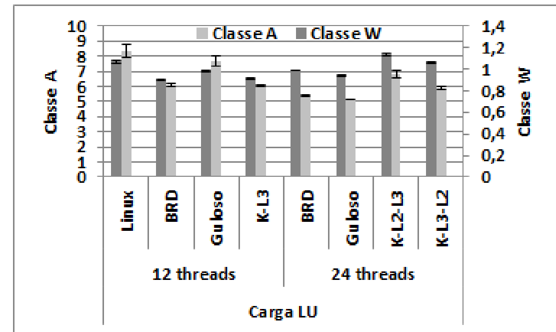


Fig. 6. Tempo de Execução (s) – Carga LU.

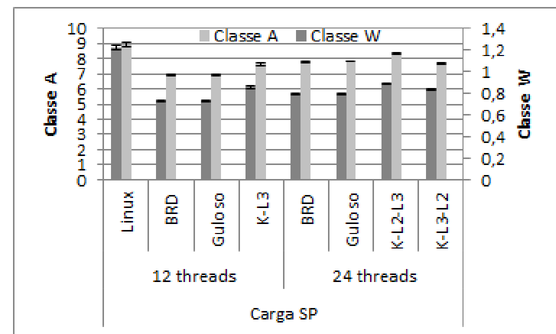


Fig. 7. Tempo de Execução (s) – Carga SP.

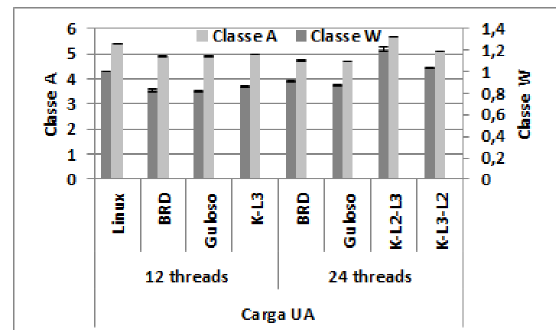


Fig. 8. Tempo de Execução (s) – Carga UA.

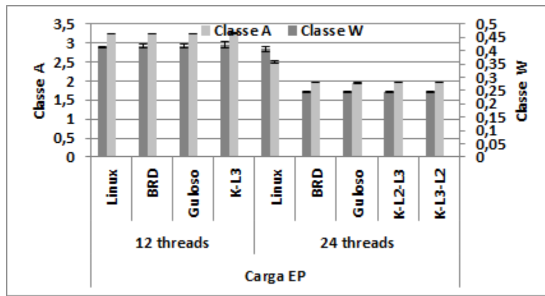


Fig. 9. Tempo de Execução (s) – Carga EP.

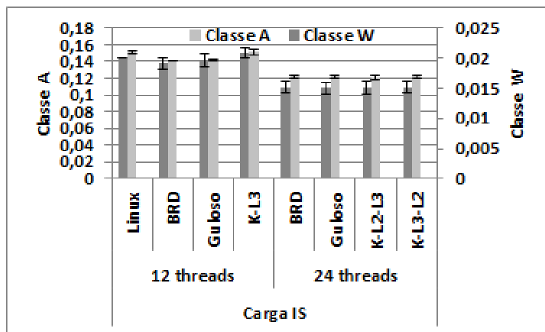


Fig. 10. Tempo de Execução (s) – Carga IS.

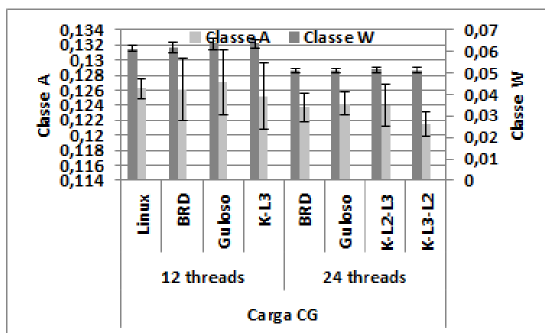


Fig. 11. Tempo de Execução (s) – Carga CG.

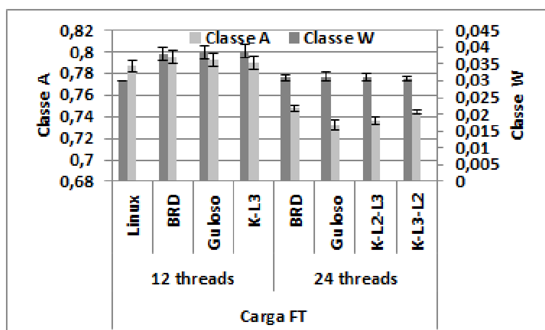


Fig. 12. Tempo de Execução (s) – Carga FT.

TABELA I
MELHORES RESULTADOS EM TEMPO DE EXECUÇÃO - CLASSE W

App.	Heurísticas						
	Sem SMT			Com SMT			
	BRD	**Guloso**	**K-L3**	**BRD**	**Guloso**	**K-L2-L3**	**K-L3-L2**
	Cargas Heterogêneas						
BT				**25,71%**			
MG				15,57%	**16,13%**		
LU	**16,15%**		14,89%				
SP	40,36%	**40,67%**					
UA	17,39%	**17,76%**					
	Cargas Homogêneas						
CG				**17,10%**	16,77%	16,12%	16,44%
EP				40,60%	**40,77%**	40,69%	40,47%
IS				24 %	**25%**	24,5%	24%
FT							

Fonte: Dados da pesquisa.

TABELA II
MELHORES RESULTADOS EM TEMPO DE EXECUÇÃO - CLASSE A

App.	Heurísticas						
	Sem SMT			Com SMT			
	BRD	**Guloso**	**K-L3**	**BRD**	**Guloso**	**K-L2-L3**	**K-L3-L2**
	Cargas Heterogêneas						
BT				**17,25%**			
MG				**8,41%**	8,30%		
LU				36,12%	**38,52%**		
SP	22,06%	**22,34%**					
UA				12,30%	**12,91%**		
	Cargas Homogêneas						
CG							**3,72%**
EP				39,74%	**39,78%**	39,63%	**39,78%**
IS				19,31%	19,17%	**20,1%**	19,31%
FT					**6,98%**	6,45%	

Fonte: Dados da pesquisa.

Em relação ao consumo de energia, as cargas heterogêneas (BT, MG, LU, SP e UA, Fig. 13, 14, 15, 16 e 17, respectivamente) com a classe W consumiram menos ao serem mapeadas pelas heurísticas BRD e Guloso sem utilizar SMT. Já com a classe A, essas heurísticas também tiveram o menor consumo, mas as cargas BT, LU e UA tiveram escalabilidade para as execuções com SMT (24 *threads*). Para a carga homogênea CG (Fig. 18), as heurísticas Guloso, *K-means-L3-L2* e BRD ao utilizar SMT apresentaram os melhores resultados com a classe W. Com a classe A, além da *K-means-L3-L2* (com SMT), as heurísticas sem SMT Guloso e *K-means-L3* também consumiram menos energia. Com as cargas homogêneas EP (Fig. 19) e IS (Fig. 20), os menores consumos energéticos foram obtidos com todas as heurísticas estudadas com SMT. Nas execuções da carga IS com a classe A, a heurística *K-means-L2-L3* teve maior destaque no consumo de energia, todavia as outras heurísticas com SMT tiveram mais de 12,98% de ganho. Com a carga FT (Fig. 21), homogênea concentrada nas 8 primeiras *threads*, as heurísticas K-means-L3 e Guloso sem SMT consumiram menos energia nas execuções com a classe W. Com a classe A, a Guloso com SMT foi melhor.

Quanto ao uso da tecnologia SMT, em tempo de execução, a maior parte dos cenários foi melhor ao utilizar SMT. Já em consumo de energia, a maioria foi melhor sem utilizar SMT. As cargas homogêneas (EP IS e CG) foram melhores em tempo de execução ao usar SMT. Já em relação ao consumo energético, estas cargas também foram melhores com o uso de SMT, com exceção da carga CG com a classe A, que consumiu menos energia ao não utilizar SMT. As cargas heterogêneas (BT, MG, LU, SP e UA) variaram mais entre serem melhores com ou sem SMT, tanto em tempo de execução quanto em consumo energético. Com a classe A, somente a carga SP

executou em menos tempo ao não utilizar SMT. A carga FT com a classe W foi melhor sem usar SMT nas duas métricas. Já com a classe A ela foi melhor ao usar SMT. Esses resultados mostram que o uso ou não da tecnologia SMT é sensível a vários fatores, tais como tamanho da carga de trabalho (classe W e A) e padrão de comunicação.

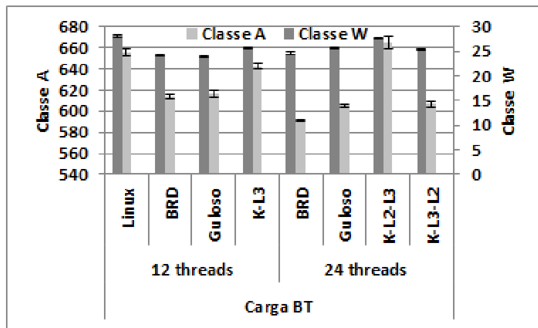


Fig. 13. Consumo de Energia (J) – Carga BT.

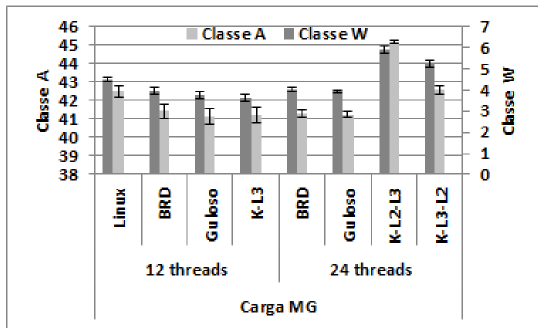


Fig. 14. Consumo de Energia (J) – Carga MG.

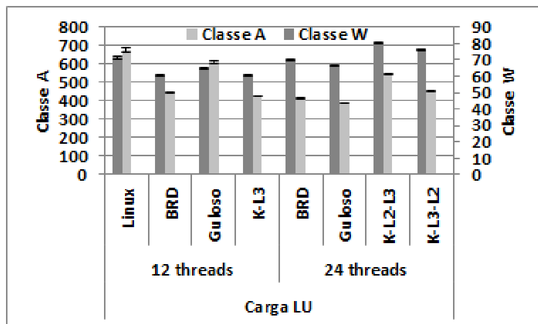


Fig. 15. Consumo de Energia (J) – Carga LU.

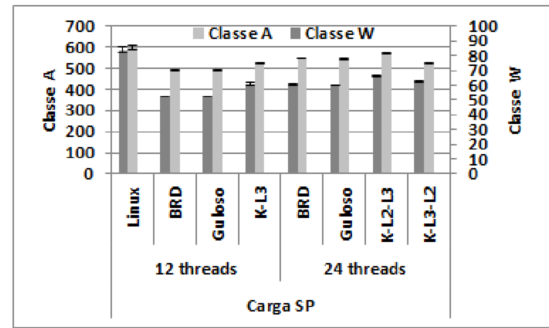


Fig. 16. Consumo de Energia (J) – Carga SP.

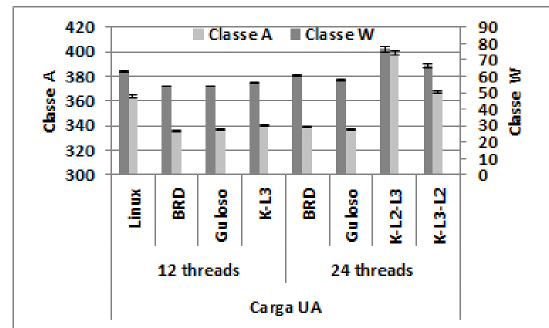


Fig. 17. Consumo de Energia (J) – Carga UA.

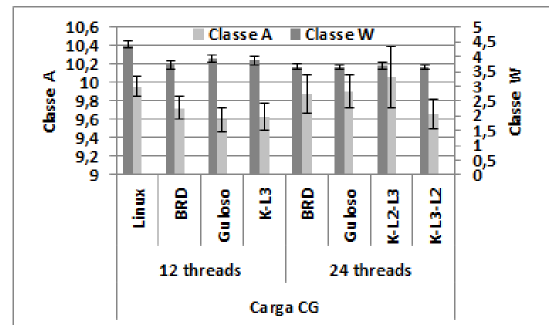


Fig. 18. Consumo de Energia (J) – Carga CG.

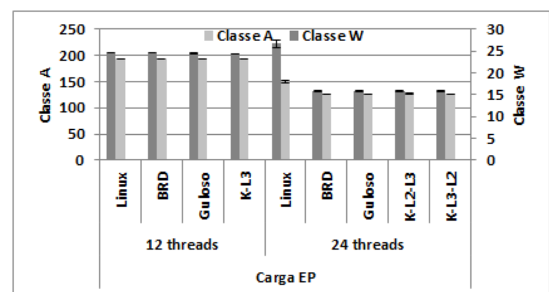


Fig. 19. Consumo de Energia (J) – Carga EP.

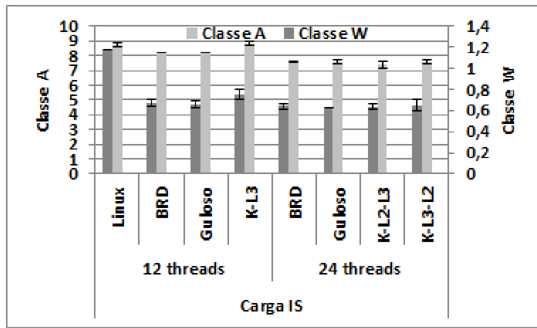


Fig. 20. Consumo de Energia (J) – Carga IS.

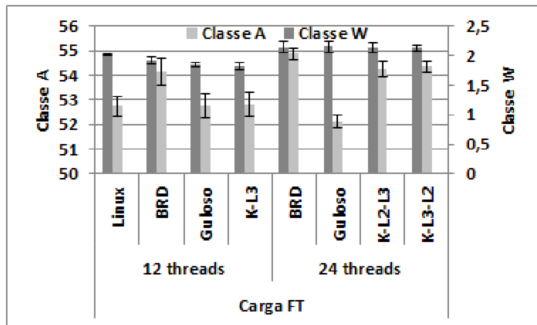


Fig. 21. Consumo de Energia (J) – Carga FT.

TABELA III
MELHORES RESULTADOS EM CONSUMO DE ENERGIA - CLASSE W

App.	Heurísticas						
	Sem SMT			Com SMT			
	BRD	Guloso	K-L3	BRD	Guloso	K-L2-L3	K-L3-L2
	Cargas Heterogêneas						
BT	14,08%	**14,80%**					
MG			**19,61%**				
LU	**15,91%**		15,27%				
SP	**37,82%**	37,7%					
UA	14,11%	**14,18%**					
	Cargas Homogêneas						
CG				17,08%	**17,64%**		17,42%
EP				35,6%	**35,94%**	35,53%	35,66%
IS				45,55%	**46,81%**	45,87%	44,50%
FT		8,34%	**9,6%**				

Fonte: Dados da pesquisa.

TABELA IV
MELHORES RESULTADOS EM CONSUMO DE ENERGIA - CLASSE A

App.	Heurísticas						
	Sem SMT			Com SMT			
	BRD	Guloso	K-L3	BRD	Guloso	K-L2-L3	K-L3-L2
	Cargas Heterogêneas						
BT				**9,78%**			
MG		**3,23%**	3,08%				
LU				38,30%	**42,06%**		
SP	17,86%	**18,30%**					
UA	**7,77%**	7,46%			7,39%		
	Cargas Homogêneas						
CG		**3,54%**	3,29%				3,01%
EP				**34,71%**	34,70%	34,35%	34,64%
IS						**15,05%**	
FT					**1,18%**		

Fonte: Dados da pesquisa.

A. Avaliação Geral dos Resultados

Os resultados mostraram que as heurísticas BRD e Guloso são melhores para as aplicações com padrão de compartilhamento de dados heterogêneo, enquanto que para as aplicações com padrão homogêneo, a heurística *K-means* também apresentou bons resultados, sendo melhor em tempo de execução com as cargas CG, EP e IS com a classe A. Isto mostra que diferentes heurísticas de mapeamento favorecem diferentes tipos de aplicações, quando executadas em processadores sem outras aplicações concorrentes. Em tempo de execução, dos 18 cenários (nove aplicações com a classe W e A), cinco tiveram melhor desempenho sem utilizar SMT, 13 foram melhores ao utilizar, e apenas uma, a carga FT com a classe W, não teve ganho com os mapeamentos estáticos. Já em consumo energético, 10 cenários consumiram menos energia sem utilizar SMT, e oito foram melhores ao utilizar. Os resultados dos cenários estudados indicam que o uso do SMT oferece, na maior parte dos casos, maior escalabilidade para as aplicações, e que o seu uso tende a ser melhor em tempo de execução, mas ao mesmo tempo gera um maior consumo energético. De todo modo, é importante ressaltar que diversos fatores influenciam no comportamento das aplicações ao utilizar ou não SMT, tais como: heterogeneidade/homogeneidade das aplicações; tamanho da carga de trabalho; contenção de recursos e relação entre quantidade de processamento e comunicação. Por tanto, cada cenário deve ser avaliado individualmente na decisão de usar ou não o SMT, e levar em consideração o objetivo de melhorar o tempo de execução ou o consumo energético.

As heurísticas de mapeamento estudadas conseguiram obter ganhos de desempenho mesmo ao executar as aplicações com cargas de trabalho maiores (classe A) que as usadas para gerar a matriz de compartilhamento (classe W). Nas aplicações LU e FT com a classe A o ganho no desempenho, em tempo de execução, foi ainda maior que com a classe W. Esta variação ocorre devido às características das aplicações. Pois, se de um lado o aumento dos dados processados pode aproveitar melhor os recursos de processamento, do outro este aumento pode sobrecarregar esses recursos por aumentar as taxas de faltas e as contenções dos mesmos. Estes resultados mostram que as cargas mantêm a proporção na distribuição de dados entre as *threads*, não alterando o custo de compartilhamento de dados entre elas e, portanto, podendo gerar o mapeamento baseado em uma quantidade de dados menor que a que será executada pela aplicação.

VI. CONCLUSÃO E TRABALHOS FUTUROS

Os resultados apresentados indicam que abordagens que geram o mapeamento com granularidade fina tendem a ser melhores para aplicações com padrões de comunicação heterogêneos, visto os resultados das heurísticas Guloso e Dividir para Conquistar (heurística BRD), que avaliam e mapeiam *thread* por *thread* a núcleo por núcleo. Já para aplicações com padrões de comunicação homogêneos, estratégias que trabalham com granularidade mais grossa tendem a ter melhores resultados, como a heurística *K-means* que gera a solução ao distribuir inicialmente as *threads* de forma aleatória em K *clusters*, e posteriormente as redistribui baseado na média da quantidade de dados compartilhados pelas *threads* de cada *cluster*. Ressaltam-se como contribuições a adaptação da heurística Gulosa e a heurística *K-means*, utilizada de

duas maneiras diferentes, para mapear em arquiteturas com hierarquia de memórias compartilhadas.

Em trabalhos futuros, sugere-se: gerar traces de acesso à memória com os tempos dos acessos, visando gerar matrizes de compartilhamento de dados mais precisas, ao considerar a ordem de leituras e escritas nos endereços de memória; aplicar a metodologia apresentada em arquiteturas NUMA com mais núcleos, com objetivo de avaliar o desempenho das heurísticas estudadas ao mapear em arquiteturas mais complexas e maiores; e caracterizar as aplicações de acordo com a utilização de recursos, a fim de identificar os seus gargalos. Além disso, é possível desenvolver escalonadores inteligentes, baseados em algoritmos de *machine learning*, fazendo uso de políticas de prioridade ou conhecimento de histórico de execuções.

AGRADECIMENTOS

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001. Os autores agradecem ao CNPq, à FAPEMIG, e à Microsoft pelo suporte parcial na execução deste trabalho.

REFERÊNCIAS

[1] H. C. Freitas, "Arquitetura de noc programável baseada em múltiplos clusters de cores para suporte a padrões de comunicação coletiva," Doutorado, Universidade Federal do Rio Grande do Sul, Instituto de Informática, Programa de Pós-Graduação em Computação, Rio Grande do Sul, Brasil, 2009. [Online]. Available: http://hdl.handle.net/10183/16656

[2] D. Koufaty and D. T. Marr, "Hyperthreading technology in the netburst microarchitecture," *IEEE Micro*, vol. 23, no. 2, pp. 56–65, March 2003.

[3] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*. New York: CRC Press, 2010.

[4] E. Moreno, F. A. Lima, and W. R. A. Dias, "Performance analysis of a low cost cluster with parallel applications and arm processors," *IEEE Latin America Transactions*, vol. 14, no. 11, pp. 4591–4596, Nov 2016.

[5] D. A. Patterson and J. L. Hennessy, *Arquitetura de computadores : uma abordagem quantitativa*. Rio de Janeiro: Elsevier, 2014.

[6] E. H. M. Cruz, M. A. Z. Alves, and P. O. A. Navaux, "Process mapping based on memory access traces," in *Anais...* Petrópolis, RJ, Brasil: Computing Systems (WSCAD-SCC), 2010 11th Symposium on, Oct 2010, pp. 72–79.

[7] S. Bokhari, "On the mapping problem," *Computers, IEEE Transactions on*, vol. C-30, no. 3, pp. 207–214, March 1981.

[8] J. He, W. Chen, and Z. Tang, "Nestedmp: Enabling cache-aware thread mapping for nested parallel shared memory applications," *Parallel Computing*, vol. 51, pp. 56 – 66, 2016, special Issue on Parallel Programming Models and SystemsSoftware for High-End Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016781911500143X

[9] G. Liu, T. Schmidt, R. Dömer, A. Dingankar, and D. Kirkpatrick, "Optimizing thread-to-core mapping on manycore platforms with distributed tag directories," in *The 20th Asia and South Pacific Design Automation Conference*, Jan 2015, pp. 429–434.

[10] T. Dey, W. Wang, J. W. Davidson, and M. L. Soffa, "Resense: Mapping dynamic workloads of colocated multithreaded applications using resource sensitivity," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 41:1–41:25, Dec. 2013. [Online]. Available: http://doi.acm.org/10.1145/2555289.2555298

[11] M. Castro, L. F. W. Góes, C. P. Ribeiro, M. Cole, M. Cintra, and J. Méhaut, "A machine learning-based approach for thread mapping on transactional memory applications," in *2011 18th International Conference on High Performance Computing*, Dec 2011, pp. 1–10.

[12] E. H. M. Cruz, M. Diener, and P. O. A. Navaux, *Thread and Data Mapping for Multicore Systems - Improving Communication and Memory Accesses*. Springer, 2018.

[13] NPB, "Nas parallel benchmarks," 2018, disponível em http://www.nas.nasa.gov/publications/npb.html. Acesso em set.

[14] E. H. M. Cruz, M. A. Z. Alves, A. Carissimi, P. O. A. Navaux, C. P. Ribeiro, and J. F. Mehaut, "Using memory access traces to map threads and data on hierarchical multi-core platforms," in *Proceedings...* Anchorage, AK, USA: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, May 2011, pp. 551–558.

[15] M. Diener, F. Madruga, E. Rodrigues, M. Alves, J. Schneider, P. Navaux, and H. U. Heiss, "Evaluating thread placement based on memory access patterns for multi-core processors," in *Proceedings...* Melbourne, Australia: 2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC), Sept 2010, pp. 491–496.

[16] F. Zheng, C. Venkatramani, R. Wagle, and K. Schwan, "Cache topology aware mapping of stream processing applications onto cmps," in *Proceedings...* Philadelphia, PA, USA: 2013 IEEE 33rd International Conference on Distributed Computing Systems, July 2013, pp. 52–61.

[17] F. Pellegrini, "Scotch and libscotch 6.0 user's guide," *Technical report, Universite Bordeaux 1 and LaBRI*, pp. 1–163, 2014.

[18] M. Diener, E. H. Cruz, L. L. Pilla, F. Dupros, and P. O. Navaux, "Characterizing communication and page usage of parallel applications for thread and data mapping," *Performance Evaluation*, vol. 88–89, pp. 18–36, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016653161500019X

[19] P. A. C. Oliveira, C. P. Avelar, S. J. F. Guimarães, and H. C. Freitas, "A greedy heuristic for process mapping on networks-on-chip," in *Anais...*, XII Simposio em Sistemas Computacionais (WSCAD-SSC). Vitória: WSCAD-SSC, Out. 2011, pp. 1–8.

[20] C. P. Avelar, "Avaliação de abordagens de mapeamento de processos em redes-em-chip para aplicações paralelas," Mestrado, Pontifícia Universidade Católica de Minas Gerais, Belo Horizonte, 2014. [Online]. Available: http://www.biblioteca.pucminas.br/teses/Informatica$_{A}velarCP_{1}.pdf$

[21] D. P. Bertsekas, "The auction algorithm: A distributed relaxation method for the assignment problem," *Annals of operations research*, vol. 14, no. 1, pp. 105–123, 1988.

[22] NPB, "Problem sizes and parameters in nas parallel benchmarks," 2019, disponível em https://www.nas.nasa.gov/publications/npb_problem_sizes.html. Acesso em mai.

[23] M. A. Z. Alves, "Increasing energy efficiency of processor caches via line usage predictors," Doutorado, Universidade Federal do Rio Grande do Sul, Instituto de Informática, Programa de Pós-Graduação em Computação, Rio Grande do Sul, Brasil, 2014. [Online]. Available: http://hdl.handle.net/10183/96062

[24] Intel, "Pin - a dynamic binary instrumentation tool," 2018, disponível em https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool. Acesso em set.

Amanda Amorim Possui graduação em Ciência da Computação (2013) e mestrado em Informática (2017) pela Pontifícia Universidade Católica de Minas Gerais. Seus interesses de pesquisa são focados em arquitetura de computadores, programação paralela e mapeamento de *threads*.

Henrique Freitas Possui graduação em Ciência da Computação (2000) e mestrado em Engenharia Elétrica (2003) pela Pontifícia Universidade Católica de Minas Gerais e doutorado (2009) em Ciência da Computação pela Universidade Federal do Rio Grande do Sul. Atuou como pesquisador convidado (2015-2016) no grupo de pesquisa CORSE do INRIA Grenoble, França. Atualmente é professor da Pontifícia Universidade Católica de Minas Gerais. Tem interesse de pesquisa em computação de alto desempenho, computação heterogênea, arquiteturas paralelas e programação paralela.