Process Mapping Based on Memory Access Traces *

Eduardo H. M. Cruz, Marco A. Z. Alves, Philippe O. A. Navaux
Grupo de Processamento Paralelo e Distribuído
Programa de Pós-Graduação em Computação, Instituto de Informática
Universidade Federal do Rio Grande do Sul
{ehmcruz, mazalves, navaux}@inf.ufrgs.br

Abstract

Process mapping is a technique widely used in parallel machines to provide performance gains by improving the use of resources such as interconnections and cache memory hierarchy. The problem to find the best mapping is considered NP-Hard and, in shared memory environments, there is the additional difficulty to find the communication pattern, which is implicit and occurs through memory accesses. In this context, this work aims to improve the performance of parallel applications that use shared memory. For that, it was developed a method for analysis of the shared memory which identifies the mapping without requiring any previous knowledge of the application behavior. Applications from the NAS Parallel Benchmarks (NPB) were used in these experiments, showing performance gains of up to 42% compared to the native scheduler of the operating system.

1 Introdução

O aumento insustentável no consumo de potência e na complexidade do design e verificação tem guiado as indústrias de microprocessadores de propósito geral a apostarem na integração de múltiplos núcleos de processamento dentro do chip (*multi-core*), como uma solução para sustentar a lei de Moore [9]. Esses processadores focam na extração de paralelismo no nível de fluxo de execução (*threads*), gerando assim um ambiente propício para o aumento de desempenho de aplicações paralelas.

Dentro da arquitetura desses processadores *multi-core*, o subsistema de memória se apresenta como o provedor fornecendo grande vazão de dados para os diversos núcleos de processamento. Dessa maneira, a hierarquia de memória *cache* desses processadores *multi-core* tem sofrido constantes mudanças para melhor se adaptar às necessidades computacionais [1], escondendo assim, o *gap* de desempenho entre memória e processador [5].

Nesse ambiente multiprocessado com diversas camadas de memória *cache*, a troca de informações entre diferentes *threads* de um mesmo programa paralelo pode variar dependendo do núcleo de processamento que determinada *thread* está executando. Esta diferença introduzida pela memória *cache* varia quando grupo de núcleos de processamento compartilham uma mesma memória *cache*. Dessa forma, é possível imaginar que uma dada aplicação paralela que possua compartilhamento de dados entre suas *threads* poderá se beneficiar de um mapeamento de suas *threads* nos núcleos de forma a melhorar o acesso a esses dados compartilhados. Além disso, em máquinas multiprocessadas com mais de um *chip*, essa variação pode ser ainda mais brusca, onde os acessos inter-chips terão um maior custo no tempo de acesso a dados. Se forem consideradas ainda as futuras topologias introduzidas pelas NoC (*Network-on-Chip*)[3], os custos de troca de informação entre *threads* poderá sofrer maiores variações [4].

No contexto de máquinas NUMA (*Non-Uniform Memory Access*), a problemática fica bastante evidente, pois um determinado processador poderá sofrer grandes latências se os dados necessários estiverem muitos *hops* de distância de sua memória [11]. Dessa forma, os ganhos obtidos melhorando o mapeamento de processos nessas máquinas NUMA costumam ser bastante favoráveis. Entretanto, em máquinas UMA (*Uniform Memory Access*), os ganhos de desempenho devem ser mais difíceis de serem alcançados, uma vez que as latências de acesso são menores e tendem a camuflar os ganhos obtidos pela melhora no mapeamento das *threads*.

Nesse dado cenário, o objetivo desse artigo é melhorar o desempenho de aplicações paralelas executadas em máquinas UMA. Para isso, está sendo proposto um método para extração de informações sobre a aplicação e utilização dessas informações para melhorar o mapeamento dos processos dentro dos processadores *multi-core*. O mapeamento deverá levar em consideração a topologia da memória *cache* e os custos de troca de dados entre processadores e núcleos

*This project is supported by CNPq and CAPES.

978-0-7695-4274-4/10 $26.00 © 2010 IEEE
DOI 10.1109/WSCAD-SCC.2010.26

de processamento. Para a avaliação da proposta, foi utilizado um ambiente real multiprocessado com oito núcleos de processamento divididos entre dois chips. Como carga de trabalho, o conjunto de aplicações cientificas paralelas NAS-NPB [6] foi utilizado para comparar o desempenho do mapeamento automático do sistema operacional com o mapeamento otimizado fornecido por nossa técnica.

O artigo está organizado da seguinte maneira: A seção 2 apresenta os principais conceitos e trabalhos relacionados ao mapeamento de processos. A seção 3 trás a metodologia e a visão geral da proposta de mapeamento de processos. A carga de trabalho é apresentada seção 4. A seção 5 descreve a aquisição dos traços de acesso à memória e o tratamento dos traços para obter-se o mapeamento a ser avaliado. Os resultados experimentais são apresentados na seção 6 e os trabalhos relacionados na seção 7. Por fim, as conclusões e trabalhos futuros estão na seção 8.

2 Mapeamento de Processos

O mapeamento de processos é uma técnica que possibilita o aumento de desempenho em aplicações paralelas através de uma alocação mais eficiente dos recursos. O primeiro passo, para escolher o melhor mapeamento, é coletar informações sobre o padrão de acesso aos recursos, que tem uma dificuldade variável dependendo do paradigma de programação paralela adotado. Quando utilizado paradigmas de programação orientados a passagem de mensagens [2], a descoberta do padrão de comunicação se torna trivial, uma vez que um simples traço das comunicações efetuadas permitirá a descoberta do padrão de acessos aos recursos. Entretanto, com o uso de programação paralela para memória compartilhada em ambientes multiprocessados, a tarefa de descoberta do padrão de comunicação se torna mais difícil, uma vez que o principal recurso a ser considerado é a memória, já que é o meio mais utilizado para realizar a comunicação entre as diferentes *threads*.

Uma característica importante a ser explorada nos sistemas multiprocessados é que o tempo de comunicação entre os núcleos, o que chamamos de tempo de comunicação, pode ser entendido pelo tempo que demora para um processador escrever em uma variável compartilhada e então um segundo processador ler a mesma variável atualizada. Esse tempo de comunicação costuma ser heterogêneo, ou seja, depende da localização dos participantes da comunicação. Isto se deve a hierarquia de memória *cache*, já que podemos ter grupos de núcleos compartilhando diferentes níveis de memória *cache* ou ainda os núcleos de processamento podem estar localizados em processadores separados. Além disso, outras topologias podem ser introduzidas pelas NoCs, que têm seu uso previsto nos futuros processadores *many-core* com dezenas de núcleos. As diferenças nos tempos de comunicação tendem a ser proporcionais ao número de núcleos presentes e níveis na memória *cache*, o que tende a aumentar no futuro.

Escalonando as *threads* que compartilham o mesmo espaço de memória em núcleos que compartilham uma mesma memória *cache* propicia um melhor desempenho de que quando nenhuma memória *cache* é compartilhada [1]. Além disso, o tempo para dois núcleos dentro de um mesmo *chip* se comunicarem é menor do que quando as *threads* encontram-se em processadores separados. Essas diferença no desempenho se devem ao fato de que além de um melhor aproveitamento de espaço nas memórias *cache*, um menor número de invalidações deverá ocorrer a cada modificação dos dados, ou seja, apenas os níveis superiores (mais próximos do processador) devem receber os valores atualizados, reduzindo assim a sobrecarga imposta por protocolos de coerência. Dessa forma, as *threads* que mais compartilham memória devem ser escalonadas em núcleos mais próximos em relação à hierarquia de memória adotada.

O mapeamento de *threads* pode ser dinâmico ou estático. No mapeamento dinâmico de processos [13], a cada etapa de processamento as *threads* devem ser re-mapeadas para melhor se adequarem ao novo padrão de compartilhamento de dados. Já no mapeamento estático, o mapeamento é predefinido, assim, uma vez iniciada a computação, o mapeamento não será modificado. Tanto no caso dinâmico quanto no estático, a decisão sobre o mapeamento deve ser tomada com base nas informações passadas pelo programador ou através do compilador ou outra técnica que permita o acesso às informações de acesso à memória.

Para nossa proposta de mapeamento de processo, será adotada a política de mapeamento estático, por ser a de mais fácil implementação nesse trabalho inicial, sendo que nos futuros trabalhos será usada a implementação dinâmica. Além disso, a aquisição de dados sobre a aplicação será feita através de uma execução prévia em um simulador. Essa técnica será usada apenas para avaliação das potencialidades do mapeamento de processos, onde a criação de uma ferramenta em hardware para prover tais informações já está com seu projeto avançado.

3 Visão Geral da Proposta

Uma vez que pretendemos efetuar o mapeamento de *threads* em aplicações paralelas de memória compartilhada, uma boa maneira de determinar o padrão de compartilhamento e comunicação é monitorar o acesso à memória, já que a colaboração existente entre as diferentes *threads* é implícita e ocorre através de leituras e escritas na memória. Por isso, foi utilizado um método para coletar as informações necessárias que consiste em registrar todos os acessos à memória e aplicar ferramentas automatizadas para analisar tais informações.

A coleta de dados sobre o compartilhamento foi feita

utilizado o simulador Simics [8], que é uma plataforma de simulação completa do sistema em nível ISA que permite executar o sistema operacional e as aplicações sem modificações. Esse ambiente de simulação foi instrumentado de forma a fornecer informações sobre os acessos a dados que cada *thread* efetuou, isolado do sistema operacional. Mais detalhes sobre a obtenção dos traços de memória e instrumentação do simulador são apresentados na seção 5.

Como carga de trabalho, foram utilizadas o conjunto de aplicações paralelizadas com OpenMP presentes no *benchmark* NAS-NPB, que foram instrumentadas com o uso das *magic instructions*, que geram interrupções especiais no simulador, e tiveram seus traços de memória gerados no Simics, registrando para cada acesso à memória o ciclo de simulação em que ocorreu, o número identificador da *thread*, o endereço de memória virtual, o tamanho em *bytes* da operação e seu tipo. Cada aplicação foi executada dentro do simulador com 8 *threads* paralelas. Os detalhes sobre a carga de trabalho e o comportamento das aplicações serão apresentados na seção 4.

Após a geração dos traços de acesso à memória de cada aplicação da carga de trabalho, uma ferramenta foi desenvolvida na linguagem C++ para ler tais traços. O programa recebe como entrada o arquivo gerado pelo simulador Simics, o número de *threads* usados na simulação e o tamanho da linha de memória *cache* a ser considerado na análise. Após o processamento, são exibidas informações sobre a quantidade de memória acessada por cada *thread* e quanta memória foi compartilhada por cada par de *threads*. Essas informações preenchem o que chamamos de matriz de compartilhamento, que é utilizada para escolher o melhor e o pior mapeamento das *threads*. A discussão sobre a forma de melhor utilizar a tabela de mapeamentos está descrita também na seção 5.

A máquina utilizada nos experimentos contém 2 processadores *Intel Xeon E5405*, cada um com 4 núcleos, totalizando 8 núcleos de execução, sendo que a memória *cache* L2 é compartilhada por cada par de núcleos. A arquitetura pode ser observada na Figura 1. O tempo de comunicação entre os *cores* que compartilham a cache L2 (0 e 2, por exemplo) é inferior a quando apenas o processador é compartilhado (0 e 4, por exemplo), sendo este último inferior a quando os *cores* encontram-se em processadores diferentes.

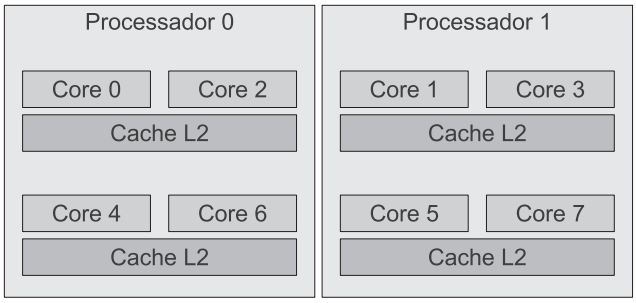


Figura 1. Arquitetura utilizada nos experimentos.

4 Carga de Trabalho

As aplicações utilizadas nos experimentos fazem parte da carga de trabalho *Numerical Aerodynamic Simulation Parallel Benchmark* (NAS-NPB) versão 3.3.1, paralelizada com OpenMP. Essa carga de trabalho é formada por diversas aplicações relacionadas a métodos numéricos de simulações aerodinâmicas para computação científica. Esses aplicativos foram projetados para comparar o desempenho de computadores paralelos, sendo formados por *kernels* e problemas de simulação de dinâmica de fluídos computacionais (CFD - *Computational Fluid Dynamics*) derivados de importantes aplicações das classes aerofísicas, de maneira que as simulações de CFD reproduzem grande parte dos movimentos de dados e computação encontrada em códigos de dinâmica de fluídos completos [6].

As seguintes aplicações fazem parte deste estudo: BT (Block Tridiagonal), CG (Conjugate Gradient), MG (Multigrid), EP (Embarassingly Parallel), SP (Scalar Pentadiagonal), LU (Lower and Upper triangular system), IS (Integer Sort), FT (fast Fourier Transform), UA (Unstructured Adaptive). Todas essas aplicações executam operações de ponto-flutuante de precisão dupla (64 bits) e são programadas utilizando Fortran-90 (BT, CG, MG, EP, SP, LU, FT, UA), com exceção da aplicação IS, que é programada em C e efetua operações em inteiros.

As aplicações BT, SP e LU têm seu paralelismo formado tipicamente por divisão de domínio espacial, com compartilhamento de dados nas bordas de cada subdomínio e apresentam acessos de forma linear aos dados. As aplicações CG e UA apresentam característica de acesso randômico a dados. A aplicação MG trabalha sobre dados contíguos, entretanto, em cada etapa de computação, o padrão de acesso varia, sendo que inicialmente o acesso é desalinhado, e conforme a grade é refinada em cada etapa, os acessos se tornam mais lineares. Assim, essa aplicação pode ser considerada, do ponto de vista de memória, um misto entre as aplicações BT, SP e LU de acesso linear, e as aplicações CG e UA de acesso irregular.

A aplicação EP é nativamente paralela e possui muito pouco compartilhamento de dados entre as *threads*. O desempenho dessa aplicação pode ser utilizado como referência de desempenho computacional de pico de uma determinada máquina. A aplicação IS, apresenta comportamento contrário a aplicação EP, já que executa muitos acessos a dados compartilhados além dos acessos lineares aos dados armazenados. Por fim, a aplicação FT apresenta eta-

pas de acesso linear com etapas de acesso compartilhado entre as diversas *threads*.

O NAS-NPB conta com diversos tamanhos de entrada para os problemas, sendo que para este trabalho foram utilizados o W e A. O tamanho W, embora seja pequeno, pode ser classificado como problema de tamanho real e foi através dele que foram obtidos os traços de memória. O tamanho A é o mais utilizado em testes de máquinas reais, apresenta um tamanho mais robusto e tempo de execução mais elevado.

5 Proposta de Mapeamento em Memória Compartilhada

O mapeamento estático é realizado em duas etapas: geração da matriz de compartilhamento e análise da matriz para atribuição das afinidades de cada *thread*.

5.1 Geração da Matriz de Compartilhamento

Para definir o mapeamento estático das *threads*, é necessário uma análise prévia da aplicação para se obter as informações constituintes da matriz de compartilhamento e achar a disposição ideal das *threads* do processo. Tal mapeamento é então utilizado para as futuras execuções da aplicação, desde que seu padrão de compartilhamento permaneça inalterado.

Para gerar os traços de memória, o simulador Simics foi instrumentado de maneira a registrar os acessos à memória física das *threads* das aplicações alvo. Isto foi feito através da utilização de gatilhos, que são eventos que disparam alguma função parametrizada quando sua condição é satisfeita. Os gatilhos são implementados na linguagem *Python*, assim como todos os *scripts* para o Simics. Os gatilhos empregados foram o *Core_Breakpoint_Memop*, *Core_Magic_Instruction* e *Core_Trackee_Active*.

O gatilho *Core_Breakpoint_Memop* é disparado em todos os acessos à memória, sendo possível determinar o endereço físico e virtual, tamanho da operação e tipo, se é leitura, escrita ou busca de instrução. O *Core_Magic_Instruction* permite que os programas sendo simulados realizem chamadas à procedimentos do Simics, onde a função de *callback* foi utilizada para capturar o número identificador de cada *thread*. O *Core_Trackee_Active* monitora trocas de contexto de processos específicos, possibilitando determinar quando as *threads* da aplicação analisada estão sendo executadas pelo simulador.

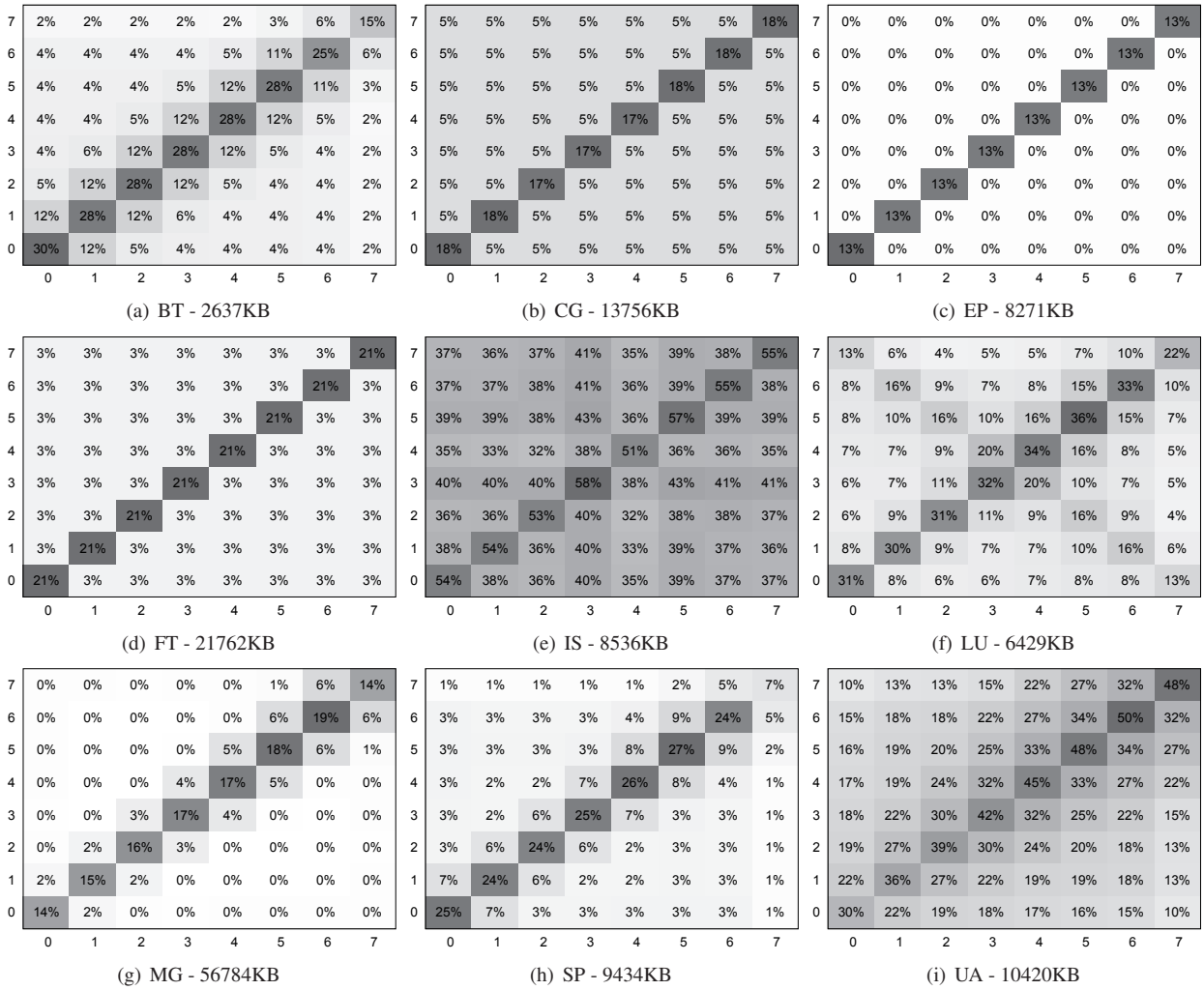
A Figura 2 contém o compartilhamento de memória do NPB com o tamanho W. É possível verificar que os gráficos obtidos são um reflexo ao comportamento das aplicações. O BT, SP e LU possuem um grau maior de compartilhamento a cada 2 *threads*, no caso do BT e SP as adjacentes, e para o LU as mais distantes, sendo que tal compartilhamento corresponde às bordas do domínio que foi dividido. O EP praticamente não apresenta compartilhamento. O CG e a FT possuem uma distribuição homogênea de compartilhamento. O UA possui características da CG, porém com maior intensidade de compartilhamento a cada par. O MG também possui um maior compartilhamento a cada par, porém com menores índices que o BT e SP. O IS tem uma distribuição irregular, não sendo possível identificar um padrão.

5.2 Afinidade de Threads com a Hierarquia de Memória

A memória compartilhada pode ser analisada em diferentes grupos de *threads*. Entretanto, após coletar os dados para guiarem o mapeamento, a escolha da melhor alocação é considerado um problema NP-Difícil. Além disso, se forem verificados todas as combinações possíveis, tem-se uma complexidade espacial exponencial para se armazenar as informações sobre o compartilhamento, inviabilizando o uso do mapeamento. Entretanto, como nos processadores atuais poucos núcleos estão ligados à uma mesma *cache* (geralmente 2 núcleos para cada *cache* L2), pode-se utilizar com uma eficácia razoável uma matriz que armazena informações de compartilhamento a cada par de *threads*, reduzindo-se a complexidade espacial para $\Theta(N^2)$, sendo N o número de *threads*.

A partir dos dados da matriz de compartilhamento, deve-se escolher quais *threads* devem ficar mais próximas na hierarquia de memória. A tarefa de se achar os pares de *threads* que possuem mais memória compartilhada, pode ser modelada como um problema de emparelhamento máximo de custo mínimo em grafos, que consiste de: dado um grafo $G = (V, E)$, deseja-se achar um subconjunto M de E no qual todos os vértices $v \in V$ incidem em no máximo um elemento de M e que as somas dos pesos das arestas seja mínimo. Segundo Kolmogorov [7], este problema pode ser resolvido com uma complexidade temporal de $O(N^3)$ através do uso do algoritmo de Edmonds.

O grafo pode ser montado diretamente a partir da matriz de compartilhamento, basta considerar cada *thread* um vértice e a quantidade de memória compartilhada o peso da aresta. Desta forma, é gerado um grafo completo que pode posteriormente ser processado pelo algoritmo de Edmonds, sendo necessário um tratamento para que se ache o custo máximo. Em Osiakwan e Akl [10], é descrito um algoritmo paralelo para se encontrar emparelhamentos perfeitos de custo máximo em grafos valorados completos que possui uma complexidade temporal de $O(\frac{N^3}{P} + N^2 \lg N)$, onde N é o número de vértices (*threads*) e P é o número de processadores. Obtêm-se como resultado os pares de *threads* que possuem mais memória compartilhada entre si. Esta



(a) BT - 2637KB (b) CG - 13756KB (c) EP - 8271KB

(d) FT - 21762KB (e) IS - 8536KB (f) LU - 6429KB

(g) MG - 56784KB (h) SP - 9434KB (i) UA - 10420KB

Figura 2. Uso de memória do NPB com 8 threads.

informação é extremamente relevante, já que em muitas arquiteturas as memórias *cache* L2 estão ligadas a somente 2 núcleos.

Mesmo sabendo-se quais as *threads* que devem compartilhar a memória *cache* L2, como a arquitetura utilizada neste trabalho ainda impõe mais um nível de hierarquia a ser explorado, deve-se descobrir quais pares de pares de *threads* devem ser alocados a cada processador. Para tanto é necessário criar uma outra matriz de compartilhamento, porém com cada linha e coluna representando os pares de *threads* escolhidos na primeira etapa e os elementos da matriz representando a memória compartilhada pelas 4 *threads*. Entretanto, a matriz de compartilhamento original não fornece informações sobre memória compartilhada a cada grupo de 4 *threads*. Um método heurístico deve ser então aplicado para determinar os valores da nova matriz. A função heurística aplicada neste trabalho foi:

$$H_{(x,y),(z,k)} = M_{(x,z)} + M_{(x,k)} + M_{(y,z)} + M_{(y,k)}$$

onde (x, y) e (z, k) correspondem a pares encontrados na primeira etapa, e $M_{(t_1,t_2)}$ é a quantidade de memória compartilhada pelas *threads* t_1 e t_2. Após obtida a matriz gerada pela função heurística, basta novamente gerar o grafo e aplicar o algoritmo de Edmonds para descobrir quais *threads* devem compartilhar o processador.

6 Resultados

A carga de trabalho NPB foi alterada de maneira a suportar atribuição de afinidades para cada *thread*, isto é, determinar em qual núcleo cada uma será executada. Isto foi feito com o uso da chamada de sistema *sched_setaffinity* do Linux. Para cada aplicação do *benchmark*, foram realizados experimentos com dois tamanhos de dado de entrada (W e

	Processador 0				Processador 1			
	Cache 0		Cache 1		Cache 0		Cache 1	
Núcleo	0	2	4	6	1	3	5	7
BT	0	1	2	3	4	5	6	7
CG	0	2	5	6	1	7	3	4
EP	0	1	6	7	2	3	4	5
FT	0	7	1	2	3	4	5	6
IS	0	1	3	5	2	6	4	7
LU	0	7	1	6	2	5	3	4
MG	0	1	2	3	4	5	6	7
SP	0	1	2	3	4	5	6	7
UA	0	1	2	3	4	5	6	7

Tabela 1. Afinidades para o NPB com 8 threads para o melhor mapeamento.

	Processador 0				Processador 1			
	Cache 0		Cache 1		Cache 0		Cache 1	
Núcleo	0	2	4	6	1	3	5	7
BT	0	4	2	5	1	6	3	7
CG	0	1	5	7	2	4	3	6
EP	0	2	6	7	1	3	4	5
FT	0	3	2	6	1	5	4	7
IS	0	7	5	6	1	3	2	4
LU	0	5	3	6	1	4	2	7
MG	0	7	2	5	1	4	3	6
SP	0	7	2	5	1	4	3	6
UA	0	5	2	6	1	4	3	7

Tabela 2. Afinidades para o NPB com 8 threads para o pior mapeamento.

Aplicação	Ganho	
	Melhor vs SO	Melhor vs Pior
BT	18,95%	5,97%
CG	26,5%	-0,31%
EP	11,6%	0,12%
FT	42,65%	0%
IS	40,59%	0%
LU	4,17%	5,74%
MG	12,57%	0%
SP	12,27%	14,58%
UA	6,47%	3,96%
Média	10,1%	7,2%

Tabela 3. Ganhos em relação ao SO nativo e pior mapeamento para o tamanho W.

Aplicação	Ganho	
	Melhor vs SO	Melhor vs Pior
BT	1,98%	2,27%
CG	4,23%	1,63%
EP	1,93%	-0,35%
FT	0,06%	-1,44%
IS	6,63%	0%
LU	1,3%	1,71%
MG	1,3%	0,87%
SP	2,92%	4,78%
UA	3,48%	4,24%
Média	2,4%	3%

Tabela 4. Ganhos em relação ao SO nativo e pior mapeamento para o tamanho A.

A), executando com 8 *threads* paralelas. Os resultados apresentados foram obtidos com base em 100 repetições de cada experimento.

A Figura 3 contém os tempos de execução do NAS-NPB com o tamanho de entrada W e a Tabela 3 exibe os ganhos percentuais. Os desvios padrões máximos foram de 25,08%, 1,25% e 1,88% respectivamente para o SO, melhor e pior mapeamento. Três configurações foram experimentadas em cada teste: escalonamento nativo pelo sistema operacional, melhor mapeamento e pior mapeamento. O melhor mapeamento consiste em colocar as *threads* que compartilham mais memória mais próximas na hierarquia de memória, já o pior mapeamento faz o contrário, fixando-se as mesmas em níveis mais distantes. As Tabelas 1 e 2 contém as afinidades utilizadas nos experimentos melhor e pior mapeamento, respectivamente.

A aplicação BT, com o melhor mapeamento, obteve um ganho de 18,9% em relação ao escalonamento pelo SO e 6% em relação ao pior mapeamento, já que o compartilhamento de memória é heterogêneo, pois as *threads* adjacentes possuem mais memória compartilhada. O motivo de o sistema operacional obter um pior desempenho que o pior mapeamento é porque o mesmo altera esporadicamente o core no qual as *threads* estão executando, podendo assim ocorrer perda de desempenho devido a uma maior taxa de *cache miss*. O compartilhamento de memória do LU, assim como no BT, é heterogêneo, porém, os maiores índices ocorrem entre as *threads* mais distantes. Entretanto, para o LU, o ganho comparado ao SO foi inferior ao com o pior mapeamento, ficando claro que a sobrecarga gerada pela migração de núcleos foi inferior que a causada pela distância dos núcleos alocados às *threads* com maior compartilhamento.

As aplicações CG, EP e FT possuem um compartilhamento de memória homogêneo entre as *threads*, fazendo com que não houvesse diferença significativa de desempenho entre o melhor e pior mapeamento. Novamente, devido ao fato do escalonador nativo do sistema operacional realizar trocas nos *cores* que executam cada fluxo, houve uma grande melhoria de desempenho, chegando até 42% para o FT. O compartilhamento do IS é o mais complexo dentre os *benchmarks* analisados, sendo que, apesar de possuir quantidades distintas de memória compartilhada entre as *threads*, tal diferença é muito pequena, fazendo com que os resultados sejam semelhantes aos obtidos com o CG e EP.

Nas aplicações MG, SP e UA, assim como na BT, há um maior compartilhamento entre as *threads* adjacentes. Para

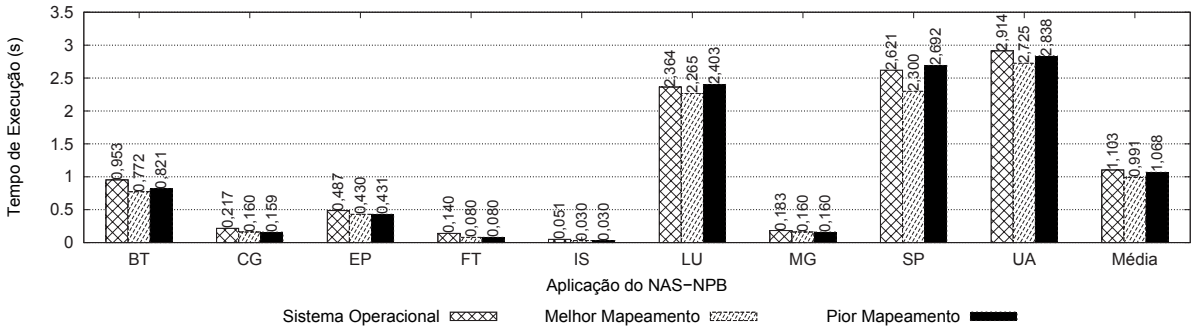


Figura 3. Tempo de execução da carga de trabalho NPB com dados de entrada tamanho W.

o MG, houve um ganho de 16% e 0% em relação ao SO e pior mapeamento, respectivamente. O SP obteve uma melhora de desempenho de 12,2% perante ao SO e 14,58% comparado ao pior mapeamento. Para o UA, houve uma melhora de 6,4% em relação ao SO e 3,9% perante o pior mapeamento. Tais resultados indicam que outras métricas além da quantidade total de memória compartilhada devem ser consideradas para o mapeamento, pois apesar do MG, SP e UA seguirem o mesmo padrão do BT, desempenhos distintos foram obtidos.

A Figura 4 contém os tempos de execução do NAS-NPB para o tamanho de entrada A e a Tabela 4 os ganhos percentuais. Os desvios padrões máximos foram de 8,11%, 7,55% e 5,79% respectivamente para o SO, melhor e pior mapeamento. Os ganhos para o tamanho A foram ligeiramente inferiores ao com o tamanho W, isto devido ao fato de a memória compartilhada ser muito inferior ao total de memória utilizado em relação ao tamanho W. Para aplicações com um padrão mais heterogêneo de compartilhamento, tais como o BT, SP, LU e UA, o ganho de desempenho em relação ao SO foi inferior do que o em relação ao pior mapeamento. Para as outras aplicações, que possuem um potencial de ganho inferior, o ganho em relação ao SO foi superior ao comparado com o pior mapeamento. Um dos motivos para tal comportamento divergente do tamanho W é que, como as aplicações com o tamanho A consomem mais memória, ocasionando uma maior taxa de *cache miss*, pode ocorrer um mascaramento da sobrecarga imposta pela migração de núcleo pelo escalonador nativo do SO.

7. Trabalhos Relacionados

No trabalho de Rodrigues [12], aplicações MPI (troca de mensagens) foram mapeadas através da instrumentação dos *wrappers* para as chamadas MPI, assim, foram criados traços contendo informações sobre as mensagens enviadas. No artigo, ganhos de desempenho de até 9,16% foram obtidos atribuindo aos processadores, os processos de acordo com a quantidade de dados enviados aos vizinhos. A principal diferença deste trabalho para o aqui apresentado é o paradigma alvo de programação paralela.

Em Thekkath e Eggers [14], foi analisado o impacto do mapeamento de processos em arquiteturas *Multithreaded*. Apesar de, em teoria, ocorrer uma redução dos *cache miss* compulsórios e de invalidações, não houve ganho de desempenho. Os autores atribuíram tal resultado ao fato de que as aplicações testadas não apresentavam características propícias ao mapeamento, principalmente por requererem pouca comunicação entre as diversas *threads* paralelas.

O mapeamento em arquiteturas NUMA foi avaliado em [11], mostrando como tais arquiteturas são sensíveis à disposição das *threads* nos diferentes núcleos. Foi desenvolvida uma infra-estrutura para controlar a afinidade de memória em plataformas NUMA com coerência de cache. Os resultados foram comparados à política de gerência tradicional do Linux, obtendo-se ganhos de até 31%.

O trabalho de Tam [13] apresenta uma técnica para mapeamento dinâmico de processos usando informações de *hardware counters* presentes no processador IBM Power-5. O mecanismo de mapeamento foi implementado diretamente no kernel do linux, e permitiu uma redução de até 70% nos acessos *off-chip*. Essa redução na quantidade de comunicação resultou em um ganho de desempenho de até 7%. Nesse trabalho, é importante ressaltar que essa técnica levou a resultados interessantes, embora os contadores de *hardware* não sejam os mais adequados para indicar o mapeamento, por fornecerem estatísticas considerando tudo o que foi executado (sistema operacional mais aplicações) em uma fatia de tempo.

8 Conclusões

Para os futuros processadores de arquitetura *multi-core* e *many-core* com dezenas de núcleos de processamento, técnicas para aperfeiçoar o uso dos recursos computacionais podem influenciar no desempenho final do sistema. Assim, a escolha correta do mapeamento de processos nessas arquiteturas multiprocessadas, visando o melhor uso da hie-

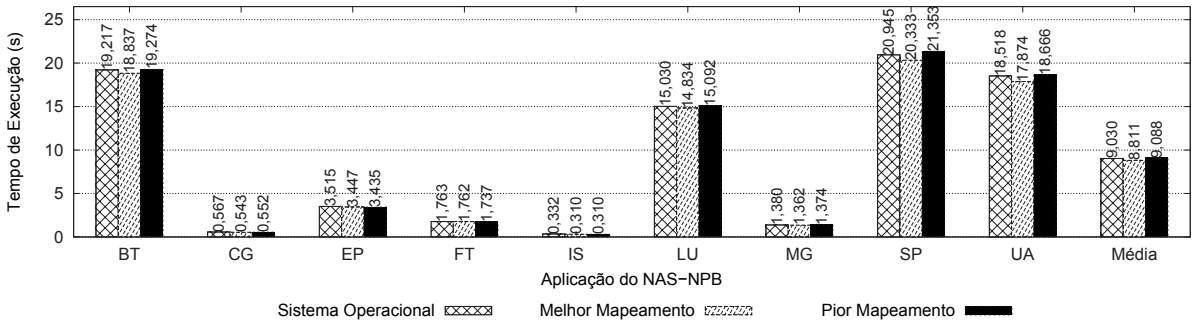


Figura 4. Tempo de execução da carga de trabalho NPB com dados de entrada tamanho A.

rarquia de memória para aperfeiçoar a comunicação das *threads* de aplicações paralelas, possui grande potencial para trazer aumento de desempenho.

Este artigo apresentou uma técnica para o mapeamento estático de aplicações em máquinas UMA, que utiliza a extração de traços de acesso à memória para guiar a escolha do melhor mapeamento de processos em uma dada arquitetura, considerando a topologia de memória *cache* com custo de troca de dados entre processadores e núcleos de processamento.

Para validar a proposta, foi avaliado o desempenho de uma carga de trabalho paralela formada de aplicações científicas, comparando o mapeamento do sistema operacional sem nossa técnica, com o mapeamento estático no início da execução sugerido pelas análises de comportamento de cada aplicação.

Dessa maneira, foram obtidos ganhos de desempenho máximo de 42% comparando o nosso mapeamento com o efetuado pelo sistema operacional, onde se obteve em média 10% de ganho para as aplicações executando com dados de entrada tamanho W e 2,4% processando os dados de entrada tamanho A.

Como trabalhos futuro, pretendemos criar mecanismos de descoberta de padrões de acesso dividido por etapas da aplicação a fim de fornecer informações para o mapeamento dinâmico das aplicações paralelas. Além disso, o desenvolvimento de uma técnica de descoberta dos traços de memória que não necessite do simulador já está sendo desenvolvida para os futuros trabalhos.

Referências

[1] M. A. Z. Alves, H. C. Freitas, and P. O. A. Navaux. Investigation of shared l2 cache on many-core processors. In *Proceedings Workshop on Many-Core*, pages 21–30, Berlin, 2009. VDE Verlag GMBH.

[2] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn. Mpipp: an automatic profile-guided parallel process placement toolset for smp clusters and multiclusters. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 353–360, New York, NY, USA, 2006. ACM.

[3] G. De Micheli and L. Benini. *Networks on Chips: Technology and Tools*. Morgan Kaufmann, 2006.

[4] H. C. Freitas, D. M. Colombo, F. L. Kastensmidt, and P. O. A. Navaux. Evaluating network-on-chip for homogeneous embedded multiprocessors in fpgas. In *Proceedings ISCAS: Int. Symp. on Circuits and Systems*, pages 3776–3779, 2007.

[5] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, USA, 4th edition, 2007.

[6] H. Jin, M. Frumkin, and J. Yan. The openmp implementation of nas parallel benchmarks and its performance. In *Technical Report: NAS-99-011*, 1999.

[7] V. Kolmogorov. Blossom V: A new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation*, 1(1):43–67, 2009.

[8] P. Magnusson et al. Simics: A full system simulation platform. *IEEE Computer Micro*, 35(2):50–58, Feb 2002.

[9] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Int. Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 2–11. IEEE, 1996.

[10] C. Osiakwan and S. Akl. The maximum weight perfect matching problem for complete weighted graphs is in pc. In *Parallel and Distributed Processing, 1990. Proceedings of the Second IEEE Symposium on*, pages 880 –887, 9-13 1990.

[11] C. Pousa, M. Castro, L. Fernandes, A. Carissimi, and J. Méhaut. Memory affinity for hierarchical shared memory multiprocessors. In *21st International Symposium on Computer Architecture and High Performance Computing-SBAC-PAD*, 2009.

[12] E. Rodrigues, F. Madruga, P. Navaux, and J. Panetta. Multi-core aware process mapping and its impact on communication overhead of parallel applications. In *Computers and Communications, 2009. ISCC 2009. IEEE Symposium on*, pages 811 –817, 5-8 2009.

[13] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. *SIGOPS Oper. Syst. Rev.*, 41(3):47–58, 2007.

[14] R. Thekkath and S. J. Eggers. Impact of sharing-based thread placement on multithreaded architectures. In *ISCA '94: Proceedings of the 21st annual international symposium on Computer architecture*, pages 176–186, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.