Impacto do Emprego da Afinidade de Processador em uma Arquitetura com Tecnologia *Clustered MultiThreading*

Carlos Alexandre de Almeida Pires[1], **Marcelo Lobosco**[1]

[1]Grupo de Educação Tutorial do Curso de Engenharia Computacional
Universidade Federal de Juiz de Fora (UFJF) – Juiz de Fora, MG – Brasil

carlos.alexandre@engenharia.ufjf.br, marcelo.lobosco@ice.ufjf.br

Abstract. *This paper evaluates the impact of affinity scheduling on the performance of parallel applications executing on multicore processors with the Clustered MultiThreading technology. The results have shown gains up to 1.8 times when affinity scheduling was used.*

Resumo. *Este trabalho investiga o impacto da afinidade de processador no desempenho de aplicações paralelas executadas em ambientes com a tecnologia Clustered MultiThreading. Os resultados mostram que o uso de afinidade pode levar a ganhos de desempenho de até 1,8 vezes.*

1. Introdução

A microarquitetura Bulldozer, desenvolvida pela AMD para o mercado de computadores pessoais e servidores, foi lançada em 2011 e representou uma grande mudança em relação à microarquitetura anterior, K10, por ter sido desenvolvida do zero. De fato, se considerado que K10 é baseada em outra microarquitetura, K8, pode-se considerar que Bulldozer representou a primeira grande alteração de projeto nas microarquiteturas dos processadores AMD desde 2003. Uma das principais características da microarquitetura Bulldozer é a sua construção modular. Cada módulo é composto, dentre outros, por duas unidades de inteiros, uma unidade ponto-flutuante (*Floating Point Unit* ou FPU), uma *cache* L1 de instruções, duas *caches* L1 de dados (uma para cada unidade de inteiro, mas ambas compartilhadas com a FPU), uma *cache* L2 (unificada para dados e instruções), uma unidade de busca, uma unidade de decodificação e uma unidade de despacho de instruções. A partir de um agrupamento de módulos são criados os processadores, adicionando-se a eles a *cache* L3, que é compartilhada por todos os módulos. A Figura 1 esboça os principais componentes do *pipeline* da microarquitetura Bulldozer, as unidades de inteiros e ponto-flutuante. Um processador com 16 núcleos, como o ilustrado na figura, será composto por oito destes módulos, já que o sistema operacional reconhece cada módulo como dois núcleos lógicos. Do ponto de vista do desempenho para aplicações *multithreading*, cada módulo pode comportar-se de modo distinto, dependendo das características da aplicação. Se a aplicação paralela executa majoritariamente instruções com tipos de dados inteiros, o módulo parecerá para a aplicação, de fato, como um processador com dois núcleos, visto que ambas as unidade de inteiros podem ser empregadas para executar as instruções. Contudo, caso a aplicação paralela execute majoritariamente instruções com tipos de dados ponto-flutuante, cada módulo parecerá com um único núcleo, visto que existe apenas uma FPU disponível para uso. Este esquema também é conhecido como *Clustered MultiThreading* (CMT), onde algumas partes do processador são compartilhadas por duas

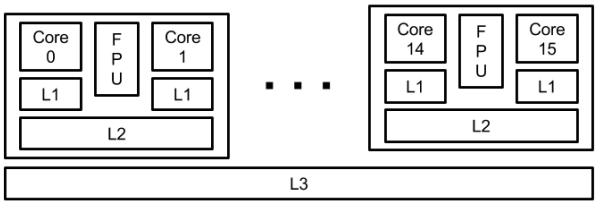

Figura 1. Módulos da microarquitetura Bulldozer, com destaque para as unidades de inteiros, ponto-flutuante e caches.

threads, como é o caso da FPU, enquanto outras são de uso privativo para cada *thread*, como a unidade de inteiros.

Neste tipo de processador, mesmo em um cenário onde existam FPUs disponíveis em número suficiente para atender uma determinada aplicação paralela com demandas para processamento ponto-flutuante, podemos ter o desempenho comprometido por conta das opções de escalonamento efetuadas pelo sistema operacional (SO). Esse cenário pode ocorrer, por exemplo, quando o SO decide escalonar duas *threads* da aplicação paralela no mesmo módulo, ainda que existam outros módulos disponíveis para uso. A hipótese deste trabalho é que uma das formas de resolver esse problema, para aplicações que demandem FPUs em número menor ou igual ao número de módulos disponíveis no sistema, é empregar a técnica conhecida como afinidade de processador (AP) (*Processor Affinity*) [Squillante and Lazowska 1993] no escalonamento de processos. Com afinidade de processador habilitada, determina-se um núcleo preferencial para execução de uma *thread*. Neste caso, pode-se orientar o SO a escalonar as *threads*, na medida do possível, em módulos diferentes. O objetivo do trabalho é avaliar a validade da hipótese, utilizando para isso o seguinte método: executa-se um *benchmark* de computação paralela, com e sem o emprego de AP, e medem-se os tempos de computação, comparando-os em seguida. Esse trabalho está organizado como descrito a seguir. A Seção 2 descreve brevemente trabalhos correlatos. A Seção 3 apresenta com mais detalhes o método empregado para avaliar o impacto da AP, e a Seção 4 apresenta os resultados obtidos. Por fim, a seção 5 apresenta as conclusões e planos para trabalhos futuros.

2. Trabalhos Relacionados

Vários trabalhos anteriores avaliaram, sob diversos aspectos, o impacto da AP no desempenho de aplicações paralelas. Um dos trabalhos avaliou diretamente o impacto da AP no desempenho de aplicações paralelas utilizando a arquitetura Intel [Pires and Lobosco 2016]. Seus resultados mostraram que o uso da AP pode impactar positivamente o desempenho nesta arquitetura, desde que haja definição criteriosa da ordem em que os núcleos destes processadores sejam utilizados. Deve-se destacar que se propõe, nesse artigo, avaliar o desempenho em um processador com características distintas, visto que no processador Intel as FPUs não são compartilhadas entre os núcleos. Outro trabalho avaliou o impacto de diversas políticas de escalonamento no desempenho de aplicações paralelas [Gupta et al. 1991], sendo uma delas a política baseada na AP. Os resultados do artigo mostraram que a AP foi responsável pela melhoria das taxas de acerto da *cache* e pelo aumento da utilização do processador. Como o estudo foi reali-

zado no início da década de 1990, a arquitetura utilizada nos testes difere da utilizada neste estudo. Dois estudos mais recentes [Ribeiro et al. 2009, Pousa Ribeiro et al. 2011], realizados em uma arquitetura NUMA, apresentaram o impacto da AP em um conjunto de aplicações científicas. A microarquitetura Bulldozer, estudada nesse artigo, é uma arquitetura UMA, com a particularidade de empregar a tecnologia CMT. Outro estudo mostrou que o uso da AP pode impactar positivamente o desempenho das aplicações [Squillante and Lazowska 1993]. Esse artigo fez um estudo teórico, sem realizar uma análise voltada para a execução de aplicações paralelas, nem baseada em máquinas reais, como a feita neste trabalho.

3. Método

Para avaliar o impacto do uso da AP no desempenho de aplicações *multithreaded* em uma arquitetura que empregue a tecnologia CMT foi escolhido o *benchmark* NPB (*NAS Parallel Benchmarks*) em sua versão memória compartilhada implementada em OpenMP [Jin et al. 1999]. Foram então coletados os tempos de execução das aplicações paralelas com e sem o uso de AP no escalonamento das *threads* pelo SO. As aplicações do *benchmark* foram executadas com até 64 *threads* e diferentes tamanhos de entrada, conforme detalhado nesta seção.

3.1. NPB 3.3

Foi utilizada nos testes deste trabalho a versão 3.3 do NPB, um conjunto de *benchmarks* criado pela Divisão de Supercomputação Avançada da NASA (*National Aeronautic and Space Administration*). Os *benchmarks* são compostos por *kernels* e pseudo-aplicações. *Kernels* são trechos de códigos que podem ser encontrados em diversas aplicações. Neste artigo, escolhemos três *kernels* e duas pseudo-aplicações para os testes. Os *kernels* escolhidos foram EP, CG, e FT. As pseudo-aplicações utilizadas foram BT e SP. Os *kernels* e pseudo-aplicações foram implementados em Fortran e utilizam tipos de dados ponto-flutuante de precisão dupla. Todos os *benchmarks* possuem um parâmetro, titulado de classe, que define o tamanho dos problemas a serem resolvidos. São 7 classes de problema: S, W, A, B, C, D e E, sendo S a classe com menor demanda de memória e E a classe com maior demanda. Para os testes realizados nesse trabalho, foram utilizadas duas classes intermediárias de problemas, A e C. A classe C foi escolhida por ser a maior classe de problemas em que todos os *benchmarks* escolhidos executam com a memória disponível no ambiente experimental. A classe A foi utilizada para permitir inferir os impactos da AP na *cache*.

Block Tri-diagonal solver (BT) implementa um caso de dinâmica de fluidos computacional (CFD - *Computational Fluid Dynamics*) utilizando as equações de Navier-Stokes em três dimensões. É empregado na solução o método das diferenças finitas com aproximações pelo método ADI (*Alternating Direction Implicit*). *Conjugate Gradient* (CG) implementa o método do gradiente conjugado para calcular uma aproximação para o menor autovalor em uma matriz esparsa, não estruturada e com valores gerados aleatoriamente. Possui acesso irregular à memória. *Embarrassingly Parallel* (EP) gera pares de valores aleatórios seguindo uma distribuição gaussiana a partir do esquema polar de Marsaglia. Este *kernel* realiza poucas operações de comunicação ao longo de sua execução. *Fourier Transform* (FT) implementa o algoritmo da transformada rápida de Fourier em três dimensões. O *kernel* resolve três transformadas separadamente, uma para

cada dimensão. Os acessos à memória são bem distribuídos e regulares. Por fim, *Scalar Penta-diagonal solver* (SP) é uma aplicação semelhante a BT, porém usando um método de aproximação distinto, o método de Beam-Warming. A aplicação acessa a memória de modo regular.

3.2. Ambiente Experimental

Os testes foram feitos em um nó de um *cluster*, cujo acesso exclusivo é feito através da submissão de *jobs* a uma fila de tarefas gerenciada pelo OGE (*Oracle Grid Engine*). O nó escolhido para execução possui quatro processadores AMD Opteron 6272 (microarquitetura Bulldozer), cada um com dezesseis núcleos, totalizando assim 64 núcleos lógicos de processamento (32 módulos). Cada núcleo possui 16KB de *cache* L1 de dados. Dois núcleos compartilham 64 KB de *cache* L1 de instruções, 2MB de *cache* L2 de instruções e dados. Cada processador possui uma *cache* L3 de 16MB, usada para armazenar tanto dados como instruções, e compartilhada entre os dezesseis núcleos de cada processador. A máquina executa o SO Linux com *kernel* na versão 2.6.32. O compilador gFortran na sua versão 6.1.0 foi usado para compilar os programas. Para habilitar o uso da AP, foi utilizada a variável de ambiente do OpenMP **GOMP_CPU_AFFINITY**[1], de modo que as *threads* fossem distribuídas pelo mapa da topologia do sistema. As *threads* foram alternadas entre módulos diferentes de processadores diferentes. Apenas quando não é mais possível alocar uma única *thread* por bloco, são alocadas duas *threads* por módulo, o que ocorre apenas em aplicações com mais de 32 *threads*. O tempo de execução foi coletado pelo aplicativo **time**, disponível no SO e que possui precisão de 0,001 segundos. Em todas as configurações, todas as aplicações executaram no nó de modo exclusivo, sem concorrência com outros processos de outros usuários. Todas as aplicações foram executadas no mínimo 5 vezes, até que o desvio-padrão fosse menor do que 5%. Os valores reportados na seção de resultados representam a média aritmética simples para os tempos de execução coletados. Para comparar os impactos do uso da AP no desempenho das aplicações que executem em um ambiente com a tecnologia CMT, foi utilizada a fórmula clássica do desempenho [Patterson and Hennessy 2014], em que o desempenho é definido como o inverso do tempo de computação.

4. Resultados

As Tabelas 1 e 2 apresentam, respectivamente para as classes A e C, os tempos médios de execução para cada *benchmark*. Nesta tabela são apresentados os tempos com e sem o uso da AP para configurações com 1, 2, 4, 8, 16, 32 e 64 *threads*. Os resultados com uma única *thread* indicam, como seria de se esperar, tempos equivalentes para execuções com e sem o uso de AP. Deve-se ressaltar que as pequenas variações de tempo são menores que o desvio-padrão. Como regra geral para todas as aplicações e classes de problemas, pontuadas algumas exceções, observa-se que para configurações com número de *threads* entre 1 e 16 não existem grandes alterações no tempo de computação com o uso de AP. Já os melhores resultados com o uso da AP foram obtidos para a configurações com 32 e 64

[1] **GOMP_CPU_AFFINITY="0 16 32 48 8 24 40 56 2 18 34 50 10 26 42 58 4 20 36 52 12 28 44 60 6 22 38 54 14 30 46 62 1 17 33 49 9 25 41 57 3 19 35 51 11 27 43 59 5 21 37 53 13 29 45 61 7 23 39 55 15 31 47 63"**. Equivale ao uso da estratégia *compact* para uso com a variável de ambiente **KMP_AFFINITY**, com *flags granularity=fine*, *scatter*, disponível apenas no compilador Intel (este trabalho usou o compilador gFortran). Esta configuração foi empregada em todos os experimentos com afinidade habilitada.

threads. Resultados próximos ou iguais para os tempos com e sem o uso de AP para 4, 8 e 16 *threads* seriam esperados, visto que a probabilidade de duas *threads* serem alocadas em um mesmo módulo é baixa. Contudo, essa probabilidade aumenta significativamente para 32 *threads*, o que faz com que o uso da AP seja mais efetivo para melhorar o desempenho das aplicações, visto que a AP elimina a chance de duas *threads* compartilharem a FPU ao não alocá-las no mesmo módulo.

Para 64 *threads* e configurações entre 2 e 16 *threads* que tiveram ganhos de desempenho com o uso de AP, o melhor desempenho é explicado pelo impacto da *cache*: por fixar as *threads* sempre nos mesmos núcleos, temos a garantia que eliminaremos falhas a frio decorrentes de escalonamento da *thread* em um núcleo distinto ao que se encontrava na fatia de tempo anterior, o que pode ocorrer quando não se usa a AP. Essa hipótese é reforçada quando se observa o comportamento das aplicações com tamanhos distintos de classes. Com uma classe que demanda menor uso de memória (classe A), o uso da AP leva a melhores desempenhos do que para a classe com maior demanda por memória (classe C). Se uma parte significativa do conjunto de dados utilizados pela aplicação cabe na *cache* L1, o que provavelmente ocorre para aplicações regulares com boa localidade temporal/espacial e baixa demanda de memória, o uso da AP torna-se muito atrativo, ao reduzir falhas a frio. Por outro lado, uma situação em que não haveria vantagem no uso da AP, mesmo para configurações com 32 *threads*, é quando as taxas de falha de *cache* são altas. Neste cenário, pode não fazer tanta diferença usar sempre os mesmos núcleos para executar as *threads*, visto que a memória principal sempre deverá ser acessada para recuperar os dados necessários para a execução da aplicação. Apesar dos indícios fortes obtidos pelos resultados, tal hipótese ainda precisa ser confirmada pela instrumentação das aplicações para coletar o percentual de falhas de *cache* com e sem o uso da AP.

Tabela 1. Tempos médios de execução (s) para cada *benchmark* da classe A.

Benchmark	Uso de AP	Número de *Threads*						
		1	2	4	8	16	32	64
BT	Não	92,11	47,72	24,06	12,11	6,60	5,11	4,77
	Sim	90,56	46,79	23,72	12,10	6,55	4,79	4,52
Desempenho:		1,02	1,02	1,01	1,00	1,01	1,07	1,06
CG	Não	3,29	1,97	0,80	0,43	0,24	0,19	0,41
	Sim	3,29	1,83	0,80	0,43	0,24	0,18	0,22
Desempenho:		1,00	1,08	1,01	1,00	1,00	1,05	1,83
EP	Não	36,86	18,47	9,25	4,63	2,34	1,30	1,36
	Sim	36,84	18,48	9,25	4,64	2,33	1,21	1,32
Desempenho:		1,00	1,00	1,00	1,00	1,00	1,08	1,03
FT	Não	7,82	3,93	2,04	1,03	0,53	0,40	0,35
	Sim	7,89	3,94	1,97	1,02	0,53	0,33	0,31
Desempenho:		0,99	1,00	1,03	1,01	1,00	1,21	1,13
SP	Não	59,87	34,98	16,39	8,39	5,42	6,35	14,83
	Sim	60,00	32,37	15,91	8,34	5,31	5,52	11,88
Desempenho:		1,00	1,08	1,03	1,01	1,02	1,15	1,25

5. Conclusão

Este trabalho apresentou uma análise do impacto do escalonamento baseado em AP no desempenho de um conjunto de aplicações paralelas executadas em um ambiente de memória compartilhada distribuída que implementa a tecnologia CMT. A análise mostrou que todas as cinco aplicações e *kernels* avaliados tiveram ganhos de desempenho de até 1,4 vezes quando executadas com a AP habilitada em uma configuração com 32 *threads*. Houve também ganho na configuração com 64 *threads*, de até 1,8 vezes, mas

Tabela 2. Tempos médios de execução (s) para cada *benchmark* da classe C.

Benchmark	Uso de AP	Número de *Threads*						
		1	2	4	8	16	32	64
BT	Não	1550,21	782,95	395,18	199,32	106,83	81,37	68,74
	Sim	1547,48	783,80	393,26	199,54	107,35	65,67	65,69
Desempenho:		1,00	1,00	1,00	1,00	1,00	1,24	1,05
CG	Não	242,91	124,65	63,41	34,43	21,82	25,01	33,51
	Sim	242,78	123,43	62,91	33,60	21,39	23,36	32,15
Desempenho:		1,00	1,01	1,01	1,02	1,02	1,07	1,04
EP	Não	589,86	294,92	147,94	73,75	36,88	19,73	19,35
	Sim	588,92	294,89	147,39	73,79	36,88	18,58	19,50
Desempenho:		1,00	1,00	1,00	1,00	1,00	1,06	0,99
FT	Não	362,00	181,64	93,81	45,91	24,87	19,26	12,52
	Sim	353,39	180,24	90,70	46,12	24,21	13,94	11,51
Desempenho:		1,02	1,01	1,03	1,00	1,03	1,38	1,09
SP	Não	1152,96	595,20	311,86	192,74	291,82	366,87	455,56
	Sim	1155,62	593,35	303,75	169,31	205,41	291,19	421,29
Desempenho:		1,00	1,00	1,03	1,14	1,42	1,26	1,08

provavelmente decorrente da redução do número de falhas de cache a frio. Não foram observados ganhos expressivos de desempenho com as configurações com 1, 2, 4, 8 e 16 *threads*, com exceção de SP, que provavelmente também se beneficiou da redução das taxas de falhas a frio. Como trabalhos futuros, os códigos das implementações serão instrumentados para coletar o percentual de falhas de *cache*, de modo a confirmar ou refutar essa hipótese.

Referências

Gupta, A., Tucker, A., and Urushibara, S. (1991). The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. *SIGMETRICS Perform. Eval. Rev.*, 19(1):120–132.

Jin, H., Jin, H., Frumkin, M., Frumkin, M., Yan, J., and Yan, J. (1999). The openmp implementation of nas parallel benchmarks and its performance. Technical report, NASA.

Patterson, D. and Hennessy, J. L. (2014). *Arquitetura de Computadores: uma abordagem quantitativa*. Elsevier Brasil, 5 edition.

Pires, C. A. A. and Lobosco, M. (2016). Avaliação do impacto da afinidade de processador no desempenho de aplicações paralelas executadas em ambientes de memória compartilhada. In *Workshop de Iniciação Científica do XVII Simpósio de Sistemas Computacionais de Alto Desempenho*, pages 50–55.

Pousa Ribeiro, C., Castro, M., Méhaut, J.-F., and Carissimi, A. (2011). *Improving Memory Affinity of Geophysics Applications on NUMA Platforms Using Minas*, pages 279–292. Springer Berlin Heidelberg, Berlin, Heidelberg.

Ribeiro, C. P., Mehaut, J. F., Carissimi, A., Castro, M., and Fernandes, L. G. (2009). Memory affinity for hierarchical shared memory multiprocessors. In *2009 21st International Symposium on Computer Architecture and High Performance Computing*, pages 59–66.

Squillante, M. S. and Lazowska, E. D. (1993). Using processor-cache affinity information in shared-memory multiprocessor scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 4(2):131–143.