Otimização de Aplicações do CAP Bench para o Processador MPPA-256

David Ordine[1], Emmanuel Podestá Jr.[1], Pedro Henrique Penna[2], Márcio Castro[1]

[1] Laboratório de Pesquisa em Sistemas Distribuídos (LaPeSD)
Universidade Federal de Santa Catarina (UFSC) – SC, Brasil

[2]Universidade de Grenoble Alpes (UGA) – Grenoble, França

david.ordine@grad.ufsc.br, emmanuel.podesta@posgrad.ufsc.br,
pedro.penna@univ-grenoble-alpes.fr, marcio.castro@ufsc.br

Resumo. *Processadores manycore de alta eficiência energética, como o MPPA-256, surgiram para tratar o consumo energético excessivo na Computação de Alto Desempenho (CAD). Neste trabalho são propostas otimizações para algumas aplicações do benchmark CAP Bench, utilizando uma nova interface de comunicação para o MPPA-256. Os resultados obtidos mostram uma redução no consumo de energia e no tempo de execução das aplicações utilizadas.*

1. Introdução

Para supercomputadores alcançarem o *Exaescale* é necessário um alto desempenho e um consumo energético viável. Devido à escalabilidade não proporcional das variáveis mencionadas, a comunidade científica se depara com uma barreira de potência. Estudos foram realizados para encontrar um balanceamento entre o desempenho e o consumo de energia, mantendo como foco principal a viabilidade energética do sistema. Desta forma, há um crescente interesse na comunidade por processadores *manycore* de baixa potência, como o MPPA-256 [de Dinechin et al. 2013], Adapteva Epiphany [Olofsson et al. 2014] e o SW26010, utilizado no supercomputador *Sunway TaihuLight* [Fu et al. 2016].

Com intuito de avaliar o desempenho e consumo energético do MPPA-256, *Souza et al.* propuseram um *benchmark* denominado CAP Bench, o qual utiliza a *Application Programming Interface* (API) de comunicação síncrona entre processos no MPPA-256, denominada *Inter-Process Communication* (IPC) [de Dinechin et al. 2013]. Esta API, além de proporcionar perda de energia no uso de *hardware* para sincronização, requer conhecimento prévio da arquitetura alvo, tornando difícil a implementação de aplicações que utilizam todos os núcleos do processador, devido ao seu baixo nível de abstração.

Para realizar a otimização proposta neste trabalho, a nova API *MPPA Asynchronous Communication* (ASYNC) [Hascoët et al. 2017] foi explorada, tendo esta um maior nível de abstração e diferenças na implementação quanto ao modelo de lógica de memória, simplificando a elaboração de programas para o processador. Além disso, a API potencializa a redução no consumo de energia devido a sua característica assíncrona.

2. Fundamentação Teórica

2.1. MPPA-256

Desenvolvido pela empresa francesa Kalray, o MPPA-256 é um processador de baixa potência que reflete o estado da arte dos *manycores*. Uma visão geral do processador é mostrada na Figura 1. O MPPA-256 possui 16 *clusters* de computação (CCs) e 4 *clusters* de Entrada e Saída (E/S). Cada CC possui: **(i)** 16 núcleos para executar *threads* de usuário em modo ininterrupto e não preemptivo, os quais atuam com frequência de 400 MHz; **(ii)** um gerenciador de recursos responsável por executar o sistema operacional e gerenciar

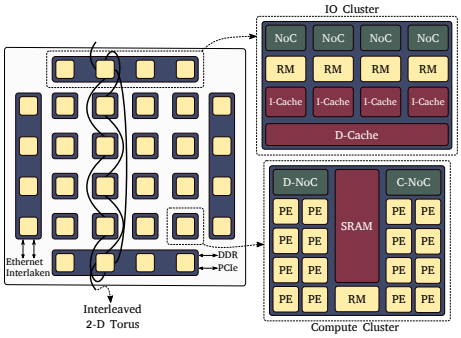

Figura 1. Visão arquitetural simplificada do MPPA-256 [Penna et al. 2018].

comunicações; **(iii)** uma memória compartilhada de 2MB, possibilitando alta largura de banda e taxa de transferência entre núcleos de um mesmo *cluster*; e **(iv)** dois controladores de Rede-em-Chip (*Network-on-Chip* – NoC), um para dados e outro para controle. Cada núcleo possui duas memórias *cache*, uma para dados e outra para instruções. As *caches* são associativas *2-way* privadas e possuem 32kB [Podestá et al. 2018].

Por outro lado, *clusters* de E/S realizam comunicações com dispositivos externos, onde dois destes apresentam acesso às memórias externas *Low-Power Double Data Rate 3 (LPDDR3)* de 2GB. É importante salientar que um *cluster* de computação (CC) não pode acessar diretamente os dados da memória de outros *clusters*. Logo, o processador apresenta um modelo de memória distribuído [Souza et al. 2016, Podestá et al. 2018].

2.2. CAP Bench

Constituído por 7 aplicações, desenvolvidas em C e baseadas em algoritmos para resolução de problemas científicos, o CAP Bench abrange problemas em diversos domínios, como grafos, ordenação e computação numérica. Estas aplicações possuem uma série de características que possibilitam uma avaliação completa de diferentes cenários. Por exemplo, algumas aplicações gastam mais tempo utilizando a CPU do que realizando comunicações entre *clusters* e/ou E/S e vice-versa.

Em sua versão original, as aplicações do CAP Bench foram implementadas no MPPA-256 utilizando-se a API IPC. Baseada no padrão POSIX IPC, esta API lida com comunicações entre CCs, e entre CCs e *clusters* de E/S. Ao usar a IPC, é preciso lidar com paralelismo explícito, ou seja, cada unidade de trabalho é totalmente independente em termos de dados e computação, onde o programador determina como o paralelismo acontece, respeitando o padrão IPC [Souza et al. 2016].

3. Versão Otimizada do CAP Bench para o MPPA-256

Recentemente, a Kalray disponibilizou uma a nova API de comunicação assíncrona e unilateral denominada ASYNC. A nova API permite que otimizações possam ser realizadas nas aplicações já existentes e implementadas com o antigo padrão IPC. A API ASYNC oferece funções que simulam um modelo de memória compartilhado. Logo, espaços de memória locais podem ser manipulados por qualquer *cluster* através de operações assíncronas do tipo `put` e `get`.

Segmentos sobre determinadas porções de memória de um *cluster* são definidos, de modo que outros *clusters* possam realizar operações `put`/`get` sobre estas, mesmo não fazendo parte de suas memórias locais. A operação `get` copia os dados de um segmento remoto para o segmento local em um *cluster*. Por outro lado, a operação `put` copia os dados de um segmento local para um segmento remoto. Cada segmento possui um

identificador próprio e único, definido na sua criação. A API disponibiliza inúmeras variantes das operações `put/get`. Alguns exemplos são as funções que trabalham com segmentos contíguos, espaçados, em blocos (2D e 3D), entre outras.

3.1. Aplicações Portadas

Nesse trabalho são propostas versões otimizadas para duas aplicações do CAP Bench: *Friendly numbers* (FN) e *LU Factorization* (LU). A primeira aplicação, busca descobrir a quantidade de números com a mesma abundância (soma dos divisores de um número divido por este número). Já a segunda, decompõe uma matriz em uma multiplicação de duas matrizes triangulares, uma inferior e outra superior.

Mudanças na implementação de ambas as aplicações foram realizadas. Na aplicação LU foram removidas funções de mudanças de linhas e colunas, designadas aos processos escravos, visto que estas operações são proibidas no cálculo da fatoração LU. Consequentemente, funções que vasculham toda a matriz a procura de novos pivôs também foram removidas, tornando a lógica da aplicação correta e diminuindo o tempo de execução nos *clusters*. Na aplicação FN, toda a lógica de comparação entre abundâncias de números, realizada no *cluster* de E/S, foi movida para os CCs. Além disso existe uma função que calcula a soma de divisores de um número. Em sua versão antiga, a função buscava todos os divisores entre 2 e este número. Já a otimizada busca entre 2 e a metade do número, visto que, acima da metade de um número, só existe ele como seu divisor.

4. Resultados Experimentais

Para coletar os resultados de desempenho das novas versões das aplicações FN e LU, cada aplicação foi executada variando-se a classe de entrada entre *small*, *default*, *huge* e o número de *clusters* entre 2, 4 e 16, respectivamente (na classe *default* a variação foi de 1 a 16 *clusters*, onde os resultados são mostrados na Figura 2). Cada classe foi definida para testar diferentes propriedades do MPPA-256 e diferem na ordem de grandeza do dado de entrada [Souza et al. 2016]. Repetiu-se cada experimento 5 vezes, obtendo-se o resultado final pela média das repetições, onde foi observado um desvio padrão entre resultados menor que 1.00%. Também foram colhidos resultados quanto ao tempo de execução dos clusters de E/S (*master*) e CCs (*slaves*), assim como os tempos de comunicação e o tempo total de execução (Tabela 1). Ademais, comparações foram realizadas diretamente com os resultados obtidos por Souza *et al.* [Souza et al. 2016].

Tabela 1. Tempos de execução (em segundos) das aplicações segundo a variação de *clusters* e classes de entrada do antigo e novo *benchmark*.

App.	Small – 2 Clusters				Default – 4 Clusters				Huge – 16 Clusters			
	Master	*Slave*	*Comm.*	*Total*	*Master*	*Slave*	*Comm.*	*Total*	*Master*	*Slave*	*Comm.*	*Total*
FN Antigo	7.47	213.94	213.9	221.43	29.9	214.04	214.0	243.95	478.3	214.60	214.6	692.97
FN Novo	0.00	120.56	80.87	121.28	0.00	120.64	49.00	122.09	0.00	121.13	16.18	127.01
LU Antigo	1.92	1.4200	36.17	38.630	4.66	2.4200	121.1	127.41	14.7	2.9300	566.7	589.15
LU Novo	0.08	0.5722	2.682	1.7373	0.22	0.9565	3.512	5.5153	0.85	1.1276	13.40	19.228

No geral, observam-se ganhos de desempenho e redução no consumo energético das aplicações. onde parte se deve ao uso da API ASYNC. Na FN, a redução no tempo de execução em comparação com a versão original das aplicações variou em função do número de *clusters* utilizados, sendo de 1.8% (3 *clusters*) até 77.6% (16 *clusters*). Por outro lado, a redução no tempo de execução da aplicação LU foi de aproximadamente 60.5%, independentemente do número de *clusters* utilizados. Com relação a energia, nota-se um ganho entre 23.2% (5 *clusters*) e 70.9% (16 *clusters*) na aplicação FN. Na aplicação LU, houve redução do consumo de energia entre 87.6% (1 *cluster*) e 92.7% (16 *clusters*).

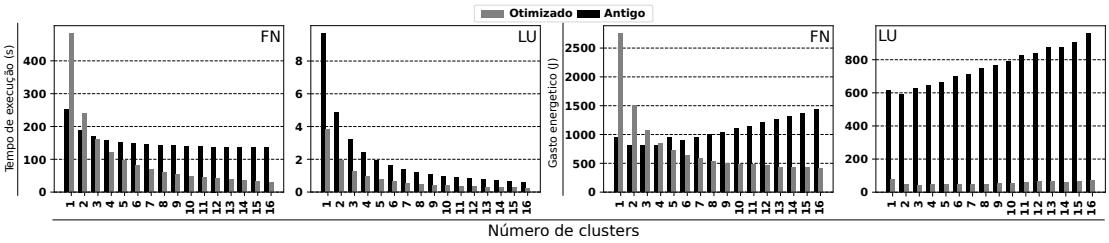

Figura 2. Tempo de execução e gasto energético de cada aplicação, variando o numero de *clusters* para a classe de entrada *default*.

O maior desempenho e consumo de energia da versão antiga com uma pequena quantidade de *clusters* na aplicação FN é esperado, já que toda a lógica de comparação foi movida para os *clusters* de computação, utilizando assim mais núcleos de processamento, aumentando o gasto energético. Além disso, foi necessário realizar sincronizações entre *clusters*, o que também contribui para o aumento do tempo de execução nesses casos.

5. Conclusão

Neste trabalho, foram propostas otimizações para duas aplicações do CAP Bench no MPPA-256, a fim de reduzir gasto energético e tempo de execução. Os resultados mostraram uma redução significativa nos tempos de execução (até 77.6%) e no consumo energético (até 92.7%) das aplicações através das otimizações implementadas e do uso da API ASYNC em comparação com a versão original das mesmas. Como trabalhos futuros, pretende-se otimizar as demais aplicações disponíveis no CAP Bench.

Referências

de Dinechin et al. (2013). A Distributed Run-Time Environment for the Kalray MPPA-256 Integrated Manycore Processor. In *International Conference on Computational Science (ICCS)*, volume 18, pages 1654–1663, Barcelona, Spain. Elsevier.

Fu, H. et al. (2016). The sunway taihulight supercomputer: System and applications. *SCIENCE CHINA Information Sciences*, 59(7):1–16.

Hascoët et al. (2017). Asynchronous one-sided communications and synchronizations for a clustered manycore processor. In *Proceedings of the 15th IEEE/ACM Symp. on Embedded Systems for Real-Time Multimedia - ESTIMedia '17*, pages 51–60, New York, New York, USA. ACM Press.

Olofsson et al. (2014). Kickstarting high-performance energy-efficient manycore architectures with epiphany. In *Asilomar Conf. on Signals, Systems and Computers*, pages 1719–1726. IEEE.

Penna et al. (2018). An operating system service for remote memory accesses in low-power noc-based manycores. In *12th IEEE/ACM International Symposium on Networks-on-Chip*, Torino, Italy.

Podestá et al. (2018). Energy efficient stencil computations on the low-power manycore mppa-256 processor. In *international European Conference on Parallel and Distributed Computing (Euro-Par)*, pages 642–655.

Souza et al. (2016). CAP Bench: A Benchmark Suite for Performance and Energy Evaluation of Low-power Many-core Processors. *Concurrency and Computation: Practice and Experience*, 29(4):e3892.