



Relatório Final - PIBIC 2018/2019

Projeto: Otimização do Benchmark CAP Bench para o Processador Manycore de Baixo Consumo Energético MPPA-256

Bolsista: David Grunheidt Vilela Ordine

Orientador: Prof. Dr. Márcio Castro

Laboratório de Pesquisa em Sistemas Distribuídos (LaPeSD), INE/UFSC

Florianópolis, 16 de Agosto de 2019

Resumo

Similar ao que aconteceu com os processadores *single-core*, ao longo de sua evolução, as tecnologias voltadas para computação de alto desempenho (HPC) depararam-se com uma barreira de potencia, a qual torna desvantajoso o *trade-off* entre gasto energético e ganho em desempenho. Processadores *manycore* de baixo consumo energético surgiram como uma possível solução para o problema, como o MPPA-256 e o Adapteva Epiphany. Devido a questões arquiteturais, como uma memória distribuída e limitada no *chip*, a implementação de aplicações que beneficiam-se totalmente do *hardware* mostra-se desafiadora. Neste projeto foram propostas otimizações para diversas aplicações do CAP Bench: um *benchmark* desenvolvido para avaliar o desempenho e o consumo de energia do MPPA-256. Os resultados mostram que as versões otimizadas das aplicações possuem um melhor desempenho que as versões anteriores. Isso se deve principalmente à otimização de partes da computação e pelo uso de uma nova biblioteca de comunicação assíncrona.

Palavras-chave: *manycores*, MPPA-256, comunicação assíncrona.

Conteúdo

1 Introdução

- 1.1 Justificativa
- 1.2 Objetivos

2 Revisão Bibliográfica

- 2.1 MPPA-256
- 2.2 CAP Bench
- 2.3 Comunicação Assíncrona no MPPA-256
- 2.4 Trabalhos Correlatos

3 Proposta e Implementação de Otimização no CAP Bench

- 3.1 Alterações no Friendly Numbers
- 3.2 Alterações no LU Factorization
- 3.3 Alterações no K-Means
- 3.4 Alterações no Gaussian Filter
- 3.5 Alterações no Features from Accelerated Segment Test
- 3.6 Alterações no Integer Sort

4 Resultados

5 Conclusão

6 Avaliação PIBIC: Benefícios e Formação Científica

1 Introdução

Para que os supercomputadores atuais consigam alcançar de forma definitiva a computação em *exascale*, é necessário que haja, de forma coesa, alto desempenho e consumo energético viável. Porém, assim como ocorreu com os avanços nas tecnologias de processadores *single-core*, os quais, nas últimas três décadas, permitiram aumento no desempenho de um processador a uma taxa anual de 40% a 50% [Larus and Kozyrakis 2008], a dissipação de calor nos supercomputadores que utilizam processadores do tipo *multicore* chegou a um ponto que não mais permitiu a escalabilidade proporcional das variáveis citadas acima.

Seguindo os conceitos de *Green Computing*, estudos foram realizados a fim de encontrar um *trade-off* positivo entre desempenho e gasto energético, centrado na redução do consumo de energia. O grande interesse da comunidade científica de HPC acerca deste tema foi um dos responsáveis por alavancar a produção dos *manycores* de baixa potência, tais quais, o MPPA-256 [de Dinechin et al. 2013], o SW26010, utilizado no supercomputador *Sunway TaihuLight* [Fu et al. 2016] e o Adapteva Epiphany [Olofsson et al. 2014].

Com propósito de validar as supostas qualidades do MPPA-256 e prover meios de comparação com outros processadores do estado da arte, Souza et al. implementaram o CAP Bench [Souza et al. 2016], *benchmark* que avalia ambos desempenho e gasto energético do processador, levando em conta diversos cenários. Em sua versão inicial, utilizava uma *Application Programming Interface* (API) de comunicação síncrona entre processos, denominada *Inter-Process Communication* (IPC) [de Dinechin et al. 2013]. Esta antiga API possui algumas deficiências, como baixo nível de abstração e realização de sincronizações implícitas, levando a queda de desempenho.

Neste trabalho, a fim de implementar a otimização proposta, realizou-se o porte do CAP Bench com a nova API de comunicação assíncrona entre processos da Kalray, a *MPPA Asynchronous Communication* (ASYNC) [Hascoët et al. 2017]. Esta API possui nível de abstração superior a IPC, além de diferir na implementação quanto ao modelo de lógica de memória. Assim, ela simplifica a elaboração de aplicações para o MPPA-256, além de ganhar em desempenho e reduzir o custo energético, devido a sua característica assíncrona.

1.1 Justificativa

É verdade que os processadores *manycore* de baixo consumo oferecem melhor eficiência energética do que os processadores *multi-core* de propósitos gerais. Todavia, estes processadores possuem algumas peculiaridades, as quais dificultam a implementação de aplicações científicas eficientes [Castro et al. 2013]. Pode-se citar, como parte destas limitações, as memórias *cache* de pequeno tamanho, ou a exigência de conformidade com a topologia *NoC*. De forma a tomar máximo proveito destes processadores, é preciso avaliar três diferentes níveis:

1. **Nível aplicativo.** Sabe-se que a grande maioria de aplicações científicas existentes são paralelizadas utilizando-se modelos de programação com memória compartilhada, por exemplo, *POSIX threads* e *OpenMP*, e modelos com memória distribuída, por exemplo, *MPI*. Do outro lado, processadores como o MPPA-256 necessitam que utilize-se ambos modelos de memória. O problema então passa em portar as aplicações científicas com este modelo híbrido.
2. **Nível intermediário.** A grande concentração de núcleos no mesmo chip exige que os programas paralelos explorem de maneira eficaz os recursos computacionais. Para que isso seja possível, mostra-se necessária a utilização de algoritmos de escalonamento e balanceamento de carga inteligentes capazes de se adaptarem a carga de trabalho das aplicações paralelas, permitindo assim uma melhor distribuição das tarefas a serem computadas nos núcleos de processamento [Penna et al. 2015]. Desta maneira, é preciso implementar estes algoritmos neste nível (ou seja, no ambiente de execução (*runtime*)), tornando o uso destes transparente para as aplicações.
3. **Nível de hardware.** A comunicação entre *clusters* é feita sobre as *NoCs*, as quais se baseiam nos conceitos utilizados nas redes de interconexão para computadores paralelos [Freitas et al. 2008]. Apesar de que as *NoCs* permitem grande escalabilidade em arquiteturas do tipo *manycore*, também adicionam um certo custo ao sistema, devido ao tráfego de dados pela rede. Logo, ambas estratégias de paralelização e ambiente de execução devem, sempre que possível, evitar que os processos em execução em *cores* ou *clusters* muito distantes se comuniquem frequentemente, removendo assim a possibilidade de gargalos de comunicação devido ao aumento do tempo de comunicação ou perdas de pacotes [de Freitas et al. 2010, Avelar et al. 2011].

1.2 Objetivos

O objetivo desta pesquisa de iniciação científica é propor e implementar a otimização do *benchmark* CAP Bench para o processador *manycore* de baixo consumo energético MPPA-256. Os objetivos específicos deste projeto de pesquisa estão elencados abaixo:

1. Investigar a viabilidade do uso do MPPA-256 para a computação científica de alto desempenho;
2. Estudar as APIs de comunicação existentes para o MPPA-256.
3. Implementar um conjunto de aplicações paralelas para o MPPA-256 (*benchmark*) utilizando-se da API ASYNC;
4. Avaliar os custos e benefícios do MPPA-256 em relação ao desempenho e ao consumo de energia, assim como sua utilidade para a Computação Sustentável (*Green Computing*);
5. Difundir a pesquisa e os seus resultados através de produção científica de qualidade, em periódicos e eventos relevantes na área de Processamento Paralelo e Distribuído.

Nas seções seguintes são apresentados o desenvolvimento e os resultados produzidos, de acordo com o cronograma e as atividades propostas deste projeto de pesquisa.

2 Revisão Bibliográfica

Esta seção apresenta a revisão bibliográfica sobre o processador *manycore* MPPA-256, o *benchmark* CAP Bench e a API utilizada para realizar a otimização proposta. Por fim, são apresentados alguns trabalhos relacionados.

2.1 MPPA-256

O MPPA-256 é um processador voltado ao baixo consumo energético, o qual, desenvolvido pela empresa francesa Kalray, reflete o estado da arte dos processadores *manycore*. A Figura 1 mostra uma visão geral da arquitetura do processador, possuindo este 16 *clusters* de computação (CCs) e 4 *clusters* de Entrada e Saída (E/S). Os *clusters* de E/S realizam comunicações com dispositivos externos, tais quais, no caso da máquina utilizada nesta pesquisa, memórias *Low Power Double Data Rate 3* (LPDDR3) de 2GB. Já os CCs possuem as seguintes características:

- 16 núcleos de processamento, chamados de *Processing Elements* (PEs), que executam, com frequência de 400 MHz, threads de usuário em modo ininterrupto e não preemptivo. Estes núcleos também possuem duas memórias *cache*, uma para dados e outra para instruções. Ambas são associativas 2-way privadas e possuem 32kB [Podestá et al. 2018].
- Gerenciador de recursos para gerenciar as comunicações de um determinado *cluster* e executar o sistema operacional.
- Memória compartilhada de 2MB, possibilitando, entre núcleos de um mesmo *cluster*, alta largura de banda e alta taxa de transferência.
- Dois controladores, para dados e controle, da Rede-em-Chip (*Network-on-Chip* (NoC)) que conecta os *clusters*.

É importante salientar que ambos CCs e *clusters* de E/S não podem acessar diretamente os dados armazenados na memória interna de um outro *cluster* que não ele mesmo. Logo, o processador possui um modelo de memória distribuído [Souza et al. 2016, Podestá et al. 2018]. Esta característica, comum a alguns processadores *manycore*, é fator desafiador para implementação de aplicações paralelas otimizadas no MPPA-256 [Franceschini et al. 2014]. Neste trabalho, *clusters* de E/S serão também chamados de *master* e CCs serão também chamados de *slaves*.

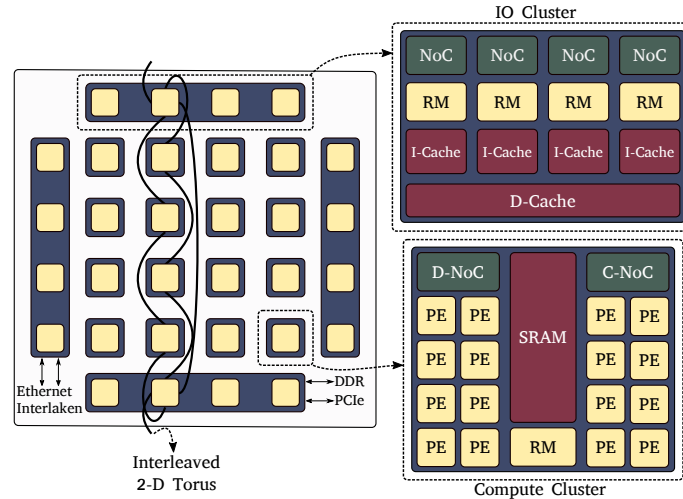


Figura 1: Visão arquitetural simplificada do MPPA-256 [Penna et al. 2018].

2.2 CAP Bench

O CAP Bench é um *benchmark* formado por 7 aplicações implementadas em C, diferindo na tecnologia de paralelismo utilizada dependendo da arquitetura alvo a ser testada. Atualmente, o *benchmark* opera sobre arquiteturas x86, utilizando OpenMP e futuramente POSIX Threads. Também opera sobre o gem5 e o MPPA-256. O módulo voltado ao MPPA-256 foi construído para testar todos os cenários de computação que este possa se deparar. Logo, as aplicações abrangem diversos problemas em diversos domínios, como grafos, ordenação e computação gráfica. Constituem o CAP Bench os seguintes kernels: (i) Features from Accelerated Segment Test; (ii) Friendly Numbers; (iii) Gaussian Filter; (iv) Integer Sort; (v) K-Means; (vi) LU Factorization; e (vii) Traveling-Salesman Problem.

Originalmente, as aplicações voltadas ao MPPA-256 foram desenvolvidas explorando a API IPC, da Kalray. Esta API, baseada no padrão POSIX IPC, lida com comunicações entre CCs, e entre CCs e *clusters* de E/S. Ao usar a IPC, é preciso lidar com paralelismo explícito, onde o programador implementa o comportamento do paralelismo e cada unidade de trabalho é independente em termos de dados e computação [Souza et al. 2016]

2.3 Comunicação Assíncrona no MPPA-256

Recentemente, a Kalray disponibilizou uma nova API para comunicação assíncrona e unilateral entre os *clusters* do MPPA-256, a ASYNC. Esta nova API abstrai o modelo de memória distribuído do MPPA-256 e simula um contexto de memória compartilhada. Desta maneira, é possível, em um determinado *cluster*, realizar operações assíncronas diretamente sobre os espaços de memória de outros *clusters*. Para isso, a API conta com diversas funcionalidades.

Em primeiro lugar, é preciso definir um segmento sobre uma porção de memória contígua de um *cluster*. Outros *clusters* podem então clonar este segmento, utilizando o *id* próprio e único deste, o qual foi definido na sua criação. Após a clonagem, operações do tipo put e get podem ser realizadas sobre aquele segmento. A operação get copia um intervalo de dados, dentro do segmento, para dentro da memória local de um *cluster*. Já a operação put envia um dado da memória local do *cluster* para um intervalo de memória contido no segmento. Também existem inúmeras operações variantes das put/get, por exemplo, funções que trabalham com intervalos espaçados dentro de um segmento, intervalos de blocos 2D e até 3D, entre outras.

A Figura 2 mostra um exemplo de ...

2.4 Trabalhos Correlatos

3 Proposta e Implementação de Otimização no CAP Bench

Para a realização da otimização proposta, foi implementado, em todas as aplicações, uma nova lógica de comunicação entre *clusters* de E/S e CCs, e entre CCs, onde o uso da ASYNC substituiu por completo o uso da antiga

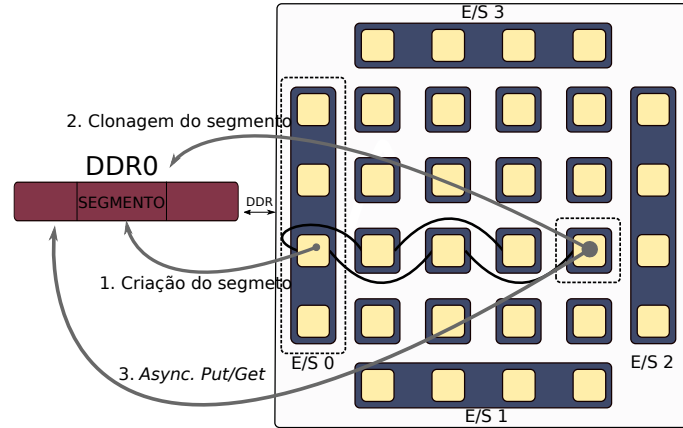


Figura 2: Etapas da utilização da API ASYNC.

IPC. Como a nova API simula um modelo de memória compartilhada em suas operações, trabalhar com ela torna a tarefa de otimização consideravelmente mais fácil. Também são abrangidos, nesta nova lógica, diversas modificações sobre em qual tipo de *cluster* é computado determinada operação de um *kernel*, com objetivo de usufruir do poder de processamento dos CCs, quando necessário, ou da memória local dos *clusters* de E/S, onde inicializa-se o dado a ser trabalhado. Todas as alterações serão mostradas detalhadamente, para cada *kernel*, nesta seção. Uma descrição detalhada de cada um dos *kernels* do CAP Bench pode ser encontrada em [Souza et al. 2016].

3.1 Alterações no Friendly Numbers

Primeiramente, foi alterado a implementação da função que calculava a soma de todos os divisores de um certo inteiro. Em sua versão antiga, esta função realizava a busca dos divisores no intervalo entre 2 e aquele inteiro. Já na otimizada, a busca é feita entre 2 e a metade deste inteiro, pois, acima da metade, só existe ele como divisor de si mesmo. Quanto a utilização da ASYNC, foi criado um segmento sobre o *array* de itens a serem passados dos *clusters* de E/S para os CCs, onde cada *cluster* de computação (CC) possui um intervalo de *offsets* no qual pode realizar operações sobre este segmento. Este intervalo é definido, em tempo de execução, no processo mestre no *cluster* de E/S, o qual repassa estas informações aos *slaves* no momento que os inicializa.

3.2 Alterações no LU Factorization

Funções que setavam um novo pivô sobre a linha da matriz sendo iterada em algum CC, realizando buscas por toda matriz e subsequentes trocas de linhas e colunas, foram removidas, pois, para calcular a fatoração LU não são permitidas tais operações, já que, ao realizá-las, é feito a decomposição PLU da matriz original, onde P é uma matriz de permutação. Além disso, a matriz L resultante não mais seria triangular inferior. Assim, sem as computações citadas acima, já é de se esperar ganho em desempenho enorme. Erros relacionados a passagem de tarefas dos *clusters* de E/S para os CCs também foram corrigidos, pois, na versão antiga, os blocos dentro da matriz eram passados aos *slaves* desalinhados, gerando descontinuidade na informação e resultado final errôneo. Para tirar proveito da ASYNC neste *kernel*, foi criado um segmento sobre a matriz original, representada por um *array* unidimensional. Porém, diferentemente do FN, ao iniciar os *slaves*, só é informado a eles o tamanho da matriz a ser decomposta. Auxiliar ao segmento da matriz, foi criado um segmento para conter os *offsets* que serão utilizados pelos CCs, no segmento da matriz, em uma certa iteração. É sobre este segmento que, em diversos momentos, cada CC retira a informação sobre qual intervalo de *offsets*, no segmento da matriz, ele deve pegar o dado. Não foi preciso informar aos *slaves*, em sua inicialização, qual é seu *offset* no segmento de *offsets* visto que seu próprio *id* pode ser utilizado para isto.

3.3 Alterações no K-Means

Neste *kernel*, diversas simplificações foram feitas. Primeiramente, para criar somente um segmento sobre a porção de dados a ser apurada e utilizar a ASYNC de modo otimizado, foi necessário transformar o modelo de dados da aplicação. Antigamente, os pontos eram definidos como *structs*, chamadas de vetores, contendo o

tamanho (dimensão do vetor) e um ponteiro para os elementos (coordenadas do vetor). Porém, como todos os vetores possuem o mesmo tamanho, já que a dimensão é tratada globalmente, foi possível otimizar este modelo de dados, setando todos os pontos em um único *array* unidimensional, o qual criou-se um segmento sobre. Vale salientar que a criação deste *array* toma muito menos instruções do que a criação do *array* de *arrays* anterior. Desta maneira, em um cenário com P pontos, todos com dimensão X , cada ponto ocupa X posições do *array* e o tamanho total deste é de $X \times P$ posições. Para recalcular os *centroids*, na antiga versão, toda a computação de cálculo da distância euclidiana média era feita nos CCs. Isto requeria que os CCs enviassem a informação sobre suas populações parciais para os *clusters* de E/S, os quais passariam de volta a soma destes dados a todos os CCs. Já na nova, nos CCs é realizado somente a primeira etapa deste cálculo, sendo esta a soma dos seus vetores parciais. Para calcular a média, a população total é necessária, a qual está facilmente disponível nos *clusters* de E/S após determinada iteração. Assim, menos comunicações são feitas entre *clusters*, mais especificamente, 3 operações *GET* e 3 *PUT*, ao contrario das 5 operações *GET* e 5 *PUT* da antiga versão.

na ver
são an
tiga nã
tinha
put/ge

3.4 Alterações no Gaussian Filter

Nesta nova versão, dois segmentos foram criados: um sobre o *array* que guarda a máscara a ser aplicada na imagem e outro sobre um *array* que guarda o *chunk*, ou pedaço da imagem, a ser trabalhado em determinada iteração. Sobre o segmento da máscara, todos os CCs tem acesso completo a ele e pegam todo o dado que ele contem, ou seja, toda a máscara. Da mesma maneira, todo o segmento do *chunk* pode ser acessado por todos os CCs, porém, só há um acesso por CC por vez. O método de comunicação dos dados no segmento de *chunk* é feito em duas etapas. Na primeira, o *cluster* de E/S realiza somente escrita, definindo as tarefas, e os CCs realizam somente leitura, sinalizando ao *master*, após armazenar o dado daquele segmento em uma variável local, que estes podem prosseguir com a inserção do próximo *chunk* para o próximo CC. Na segunda, após já terem feito toda computação sobre aquele *chunk*, os *slaves* aguardam o *master* sinalizar que está na hora de escrever o dado calculado sobre o segmento, e, quando sinalizados, escrevem-no. Já o *master*, aguarda cada CC sinalizar que tal dado já foi escrito no segmento. Este método é ligeiramente melhor que o anterior, pois só é preciso realizar operações *GET* e *PUT* no lado dos *slaves*, já que, no lado do *master*, só é necessário fazer um copia de memória para o endereço onde o segmento foi criado, o qual encontra-se na memória local.

3.5 Alterações no Features from Accelerated Segment Test

Para a otimização deste *kernel*, foram criados 3 segmentos sobre os dados originais. Primeiro, um segmento para a máscara, o qual todos os CCs tem acesso completo. Em seguida, um sobre a imagem original e um sobre a imagem a ser filtrada, os quais cada CC tem acesso somente a um intervalo de *offsets* em uma certa iteração. As comunicações entre *master* e *slaves* também foram simplificadas, tornando os *slaves* mais autônomos. Na versão antiga, os *slaves* iteravam dentro de um `while(true)`, onde, para terminar a execução, era necessário a passagem de mensagens entre *clusters* de E/S e CCs sinalizando este término ou não (as mensagens eram passadas em toda iteração). Na nova versão, os CCs já sabem, ao serem inicializados, qual será o número de operações que irão realizar, podendo executar estas operações dentro de um `for`, iterando de 0 até o número máximo de operações. Assim, a antiga lógica de mensagens do *kernel* foi excluída. Simplificando, o *cluster* de E/S informa todas os dados necessários na inicialização dos *slaves*, onde, após isso, não realiza mais nenhuma sincronização, aguardando somente o resultado final dos CCs contendo as somas parciais dos *corners*. Quando inicializados, os CCs também conhecem quais intervalos de *offset*, dentro do *array* da imagem original e do *array* da imagem a ser filtrada, irão trabalhar em todas as iterações, pois, ao conhecer os intervalos iniciais, conseguem calcular os das próximas iterações.

3.6 Alterações no Integer Sort

Nesta última otimização, foi criado um segmento sobre todos os *minibuckets* a serem passados para cada CC em determinada iteração. Assim, como temos, no MPPA-256, 16 CCs ao total, este segmento foi criado sobre um *array* de 16 posições, onde cada posição possui um *minibucket* (em execuções com menos de 16 CCs, as posições extras não são utilizadas). Como o *array* é do tipo inteiro, há uma simplificação na hora de passar a informação aos CCs. Para realizar a comunicação, os elementos de um *minibucket* são colocados no espaço do segmento reservado ao CC sendo iterado naquele instante, enviando um sinal a este CC de que o dado está pronto para ser

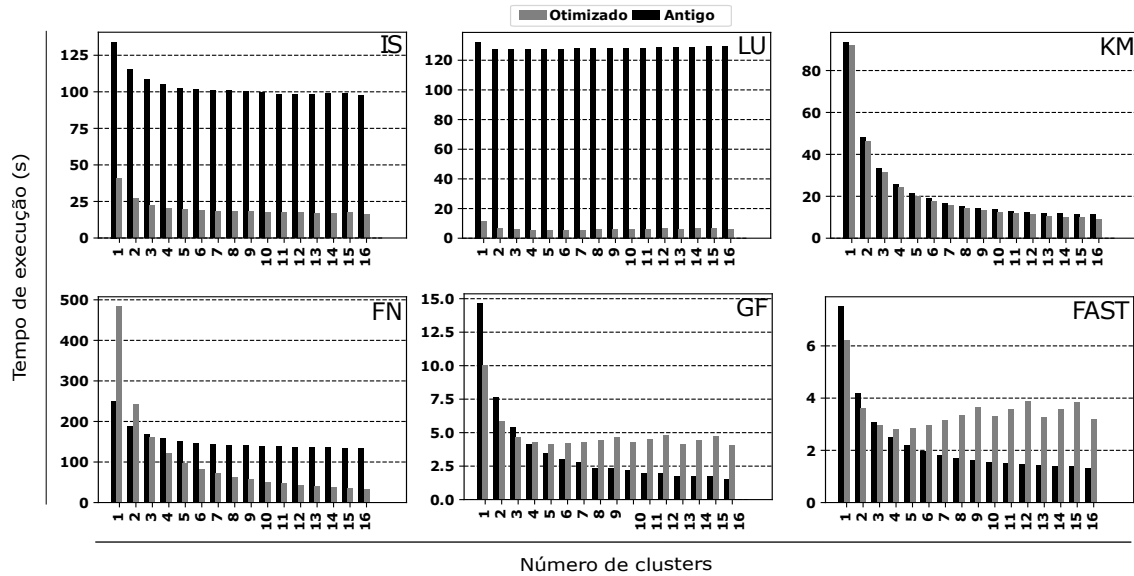


Figura 3: Tempos de execução de cada aplicação, variando-se o número de *clusters* para a classe *default*.

computado. Também foram realizadas otimizações nas lógicas de ordenamento. Na antiga versão, para qualquer variação de classe de problema, a ordenação, nos CCs, era feita sempre com o número máximo de elementos permitidos, o que tomava um tempo muito elevado na aplicação. Na nova versão, é realizado a ordenação dos elementos passados somados a, no máximo, 16 novos elementos, onde todos estes novos possuem o valor inteiro máximo possível. Isto é possível, pois como há 16 PEs em cada CC, precisamos que o número de elementos passados para ordenação seja sempre divisível por 16, a fim de paralelizar o trabalho. A antiga versão utilizava o número máximo, pois este era também divisível por 16. Após realizar o ordenamento, os CCs colocam o *array* de volta no mesmo intervalo dentro do segmento de *minibuckets*, enviando um sinal ao *master* de que o trabalho foi feito e o dado está pronto.

4 Resultados

Para coletar os resultados de desempenho das novas versões das aplicações, cada aplicação foi executada com a classe de problema *default*, variando-se o número de *clusters* no intervalo de 1 a 16. No CAP Bench temos diversas classes de problemas, cada qual variando seu tamanho de entrada, podendo assim testar diferentes propriedades do MPPA-256. Cada variação foi repetida 10 vezes, obtendo-se o resultado final pela média das repetições, o qual observou-se desvio padrão menor que 1%. Ademais, os resultados foram comparados com os obtidos por Souza et al. [Souza et al. 2016]. No geral, observa-se ganho em desempenho e, na média, redução de gasto energético também.

O tempo de execução maior na versão antiga da FN, com uma pequena quantidade de *clusters*, é esperado, já que toda lógica de comparação foi movida para os CCs, utilizando assim mais núcleos de processamento, o que aumenta o número de comunicações e sincronizações entre CCs (o que é parte da lógica). Da mesma maneira, como mais núcleos estão em execução, há aumento no consumo de energia. Porém, ao aumentar-se o número de CCs, observa-se um ganho em desempenho.

Também justifica-se o tempo e gasto energético maior nas aplicações GF e FAST. Os tempos mostrados nos gráficos consideram o tempo total de execução da aplicação (na nova versão excluímos o tempo de *spawn* dos *slaves*). Porém, as versões do MPPA-256 em que cada versão do CAP Bench rodou, antiga e nova, são diferentes, possuindo peculiaridades próprias que afetam no tempo de execução, quando este aproxima-se de 0, onde tais peculiaridades não são medidas diretamente dentro do CAP Bench. Estas peculiaridades não tem relação com a ASYNC e podem criar um viés de que o resultado piorou, quando isto não é verdade.

Ambas as aplicações, FAST e GF, estão com a mesma lógica, pois utilizam o *master* somente para definição de tarefas e os *slaves* para os cálculos. Assim, ao analisar os tempos médios de computação dos *slaves* e de comunicação, vemos uma redução em todas as variações de execução, como por exemplo, na variação de execução com a classe *default* utilizando 16 *clusters*, onde o tempo médio dos CCs foi de 0,8 segundos para a versão

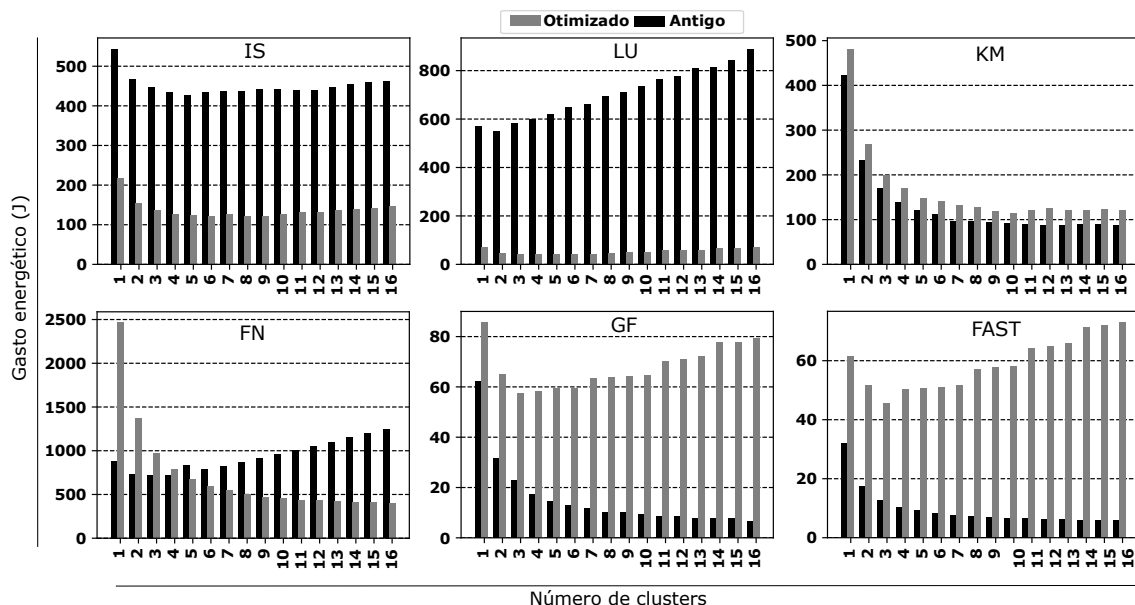


Figura 4: Gasto energético cada aplicação, variando-se o número de *clusters* para a classe *default*.

antiga, tomando 1,7 segundos de comunicação, enquanto que na nova, o tempo médio foi de 0,36 segundos, utilizando 1,08 segundos de comunicação. Da mesma maneira, como o gasto energético considera todo o tempo de execução, ou seja, o total citado acima, este também ficou maior para as aplicações GF, FAST e KM.

5 Conclusão

Neste trabalho, foram propostas otimizações para as aplicação do CAP Bench no MPPA-256, a fim de reduzir gasto energético e tempo de execução. Os resultados mostraram que houve esta redução para algumas das aplicações, através do uso da ASYNC, mas para outras, considerando o tempo total, não. Como trabalhos futuros, pretende-se estudar a fundo as peculiaridades que afetam o tempo total, citadas na seção de resultados, a fim de poder medi-las e excluí-las do tempo total, quando necessário, para realizar uma comparação mais justa, focando nas APIs ASYNC e IPC. Além disso, pretende-se atualizar a implementação do CAP Bench com a API IPC, de modo a torná-la equivalente a implementação utilizando a ASYNC, podendo assim realizar uma comparação totalmente voltada para as APIs de comunicação.

6 Avaliação PIBIC: Benefícios e Formação Científica

Este projeto de pesquisa contribuiu de inúmeras formas para minha formação acadêmica. Do começo ao fim foi algo engrandecedor e acredito que, ao longo da minha carreira profissional, irei utilizar diversos conhecimentos aqui adquiridos. Este foi o primeiro projeto no qual tive contato com uma documentação de API, e, assim como nosso primeiro contato com qualquer tecnologia nova, foi bastante desafiador. Através de muita dedicação e ajuda do meu orientador e colegas de trabalho, consegui quebrar as barreiras do conhecimento e entender a fundo como funciona a nova API da Kalray, a ASYNC.

Neste ciclo de pesquisa também produzi um artigo científico, o qual foi aprovado na décima nona Escola Regional de Alto Desempenho da Região Sul (ERAD/RS 2019), ocorrida na cidade de Três de Maio, no Rio Grande do Sul. Com esta aprovação, fui apresentá-lo nesta mesma cidade na data de realização do evento. Participar deste evento foi também uma experiência enriquecedora, pois pude conhecer projetos de diferentes níveis de complexidade, englobando tanto projetos de iniciação científica quanto os do estado da arte.

Com este projeto de pesquisa pude também descobrir o que pretendo seguir na minha carreira profissional, assim como ter plena certeza de que de quero continuar com a pesquisa e contribuir de forma significativa para o avanço tecnológico de toda comunidade científica. Utilizarei também os resultados deste para realização do meu trabalho de conclusão de curso, onde pretendo expandir o que já foi pesquisado, realizando comparações com outros processadores do estado da arte.

Referências

- [Avelar et al. 2011] Avelar, C. P., Oliveira, P. A. C., Freitas, H. C., and Navaux, P. O. A. (2011). Evaluating the problem of process mapping on network-on-chip for parallel applications. In *2011 Second Workshop on Architecture and Multi-Core Applications (wamca 2011)*, pages 18–23.
- [Castro et al. 2013] Castro, M., Francesquini, E., Nguélé, T. M., and Méhaut, J.-F. (2013). Analysis of computing and energy performance of multicore, numa, and manycore platforms for an irregular application. In *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '13, pages 5:1–5:8, New York, NY, USA. ACM.
- [de Dinechin et al. 2013] 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.
- [de Freitas et al. 2010] de Freitas, H. C., Schnorr, L. M., Alves, M. A. Z., and Navaux, P. O. A. (2010). Impact of parallel workloads on noc architecture design. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 551–555.
- [Francesquini et al. 2014] Francesquini, E., Castro, M., Penna, P. H., Dupros, F., de Freitas, H. C., Navaux, P. O. A., and Méhaut, J.-F. (2014). On the Energy Efficiency and Performance of Irregular Applications on Multicore, NUMA and Manycore Platforms. *Journal of Parallel and Distributed Computing (JPDC)*, 76:32–48.
- [Freitas et al. 2008] Freitas, H. C., Santos, T. G. S., and Navaux, P. O. A. (2008). Design of programmable noc router architecture on fpga for multi-cluster nocs. *Electronics Letters*, 44(16):969–971.
- [Fu et al. 2016] Fu, H. et al. (2016). The sunway taihulight supercomputer: System and applications. *SCIENCE CHINA Information Sciences*, 59(7):1–16.
- [Hascoët et al. 2017] 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.
- [Larus and Kozyrakis 2008] Larus, J. and Kozyrakis, C. (2008). Transactional memory. *Commun. ACM*, 51(7):80–88.
- [Olofsson et al. 2014] 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] 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.
- [Penna et al. 2015] Penna, P. H., Castro, M., Freitas, H. C., Broquedis, F., and Méhaut, J.-F. (2015). Uma Metodologia Baseada em Simulação e Algoritmo Genético para Exploração de Estratégias de Escalonamento de Laços. In *WSCAD 2015 - Simpósio em Sistemas Computacionais de Alto Desempenho*, Florianópolis, Brazil. SBC.
- [Podestá et al. 2018] 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] Souza, M. A. 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*.