UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CIÊNCIAS DA COMPUTAÇÃO

David Grunheidt Vilela Ordine

Comparação de Tecnologias de Comunicação entre Clusters no Processador MPPA-256: Um Estudo com Aplicações do CAP Benchmarks

Florianópolis
8 de dezembro de 2020

David Grunheidt Vilela Ordine

Comparação de Tecnologias de Comunicação entre Clusters no Processador MPPA-256: Um Estudo com Aplicações do CAP Benchmarks

Trabalho de Conclusão de Curso submetido ao Curso de Bacharelado em Ciências da Computação da Universidade Federal de Santa Catarina para a obtenção do título de Bacharel em Ciência da Computação.
Orientador: Prof. Márcio Bastos Castro, Dr.

Florianópolis
8 de dezembro de 2020

Este trabalho é dedicado a minha família, que sempre me apoiou e esteve do meu lado, e também aos meus amigos, os quais me ajudaram a passar por todo o processo de escrita e implementação de uma maneira mais feliz.

AGRADECIMENTOS

Agradeço a todos os meus colegas de trabalho e de curso, os quais contribuíram significativamente para a conclusão deste trabalho, através da troca de experiência e conhecimentos técnicos. Em especial, agradeço ao meu orientador, Márcio Bastos Castro, e ao Pedro Henrique Penna, por despertarem em mim interesse na área da computação paralela e me ajudarem no processo de aprendizado e desenvolvimento deste trabalho.

For a number of years I have been familiar with the observation that
the quality of programmers is a decreasing function of the density of go
to statements in the programs they produce
(DIJKSTRA, 1968)

RESUMO

O principal método para o ganho em desempenho, no processo de evolução dos processadores *single-core*, foi o aumento da frequência de *clock* do processador, o qual, com a crescente desproporção entre o gasto energético e o aumento de performance, deixou de ser viável. Diz-se então que esta desproporção foi a barreira de evolução para esta classe de processadores. Soluções que utilizam processadores *multi-core*, por exemplo, supercomputadores, também enfrentam uma barreira similar, nos dias de hoje, ao agrupar diversos destes processadores em *clusters*, ou agrupar diversos núcleos em um mesmo *chip*. Processadores *manycore* de baixo consumo energético, como o MPPA-256 e o Adapteva Epiphany, surgiram como uma possível solução para este problema. Entretanto, devido a questões arquiteturais, como uma memória distribuída e limitada no *chip*, a implementação de uma aplicação que se beneficia totalmente do *hardware* de um processador desta classe mostra-se desafiadora. Porém, quando bem feita, sobressai alguns processadores *multi-core* do estado da arte, através do menor consumo energético. Neste trabalho foram propostas para o CAP Bench, um *benchmark* desenvolvido para avaliar o desempenho e o consumo de energia do MPPA-256, otimizações nas aplicações da versão atual e a criação de uma versão das aplicações que utiliza uma nova tecnologia de comunicação assíncrona entre *clusters*, com objetivo de analisar as duas tecnologias de cada versão. Os resultados mostraram que as aplicações que utilizam a nova biblioteca apresentam melhor desempenho sobre as aplicações da versão antiga. Isso se deve principalmente pela característica assíncrona desta biblioteca.

Palavras-chave: Benchmark. Manycore. Desempenho. Green-Computing.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE LISTAGENS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*. 20, 21, 29, 31, 33, 35, 37, 39, 41, 42, 44, 47, 49, 50, 58, 61
ASYNC	MPPA *Asynchronous Communication API*.21, 29, 32, 33, 34, 35, 39, 41, 43, 44, 45, 47, 49, 50, 56, 58, 61
CC	Cluster de Computação. . 28, 29, 32, 33, 35, 41, 42, 43, 44, 45, 46, 47, 49, 54
CMP	*Chip MultiProcessadores*. 25
CPU	*Central Processing Unit*. 23, 24, 25, 26, 27, 28, 37, 38
E/S	Entrada e Saída. 28, 29, 32, 33, 34, 35, 41, 42, 44, 45, 46, 49, 50, 54
FAST	*Features from Accelerated Segment Test*. 42, 50, 58
Flops	*Floating-point Operations per Second*. 19
FN	*Friendly Numbers*. 41
FPGA	 37
GF	*Gaussian Filter*. 43, 50, 58
GPU	Unidade de Processamento Gráfico. 26, 37, 38
HPC	*High-Performance Computing*. 20, 26, 37, 38
IPC	MPPA *Interprocess Communication API*. . 21, 29, 32, 33, 39, 41, 43, 44, 49, 50, 58, 61
IS	*Integer-Sort*. 46, 50
KM	*K-Means*. 45, 49, 50, 56
LU	*LU Factorization*. 44, 50, 56
MPI	*Message Passing Interface*. 29, 31, 32, 37, 38
NoC	*Network-on-Chip*. 28, 33, 49
NUMA	*Nonuniform Memory Access*. 23, 25
OpenMP	*Open Multi-Processing*. 29, 30, 31, 32, 37, 42
RAM	*Random-Access Memory*. 23

SIMD *Single Instruction Multiple Data*. 26
SO Sistema Operacional. 27
SPMD *Single Program, Multiple Data*. 31

UM Memória Unificada. 38
UMA *Uniform Memory Access*. 23, 24, 25

SUMÁRIO

1 INTRODUÇÃO

Os avanços na indústria de semicondutores, juntamente com a área de arquitetura de computadores, são notados desde a década de 1980, permitindo um crescimento anual em desempenho de processadores *single core* de 40% a 50% (LARUS; KOZYRAKIS, 2018). Porém, a necessidade de uma nova classe de processadores mostrou-se eminente ao se atingir um ponto onde a relação entre gasto energético e aumento em desempenho era desproporcional, havendo muita dissipação de calor para pouco crescimento em performance. Essa barreira de potência foi então a responsável pelo interesse da indústria de semicondutores na classe de processadores *multicore*.

Arquiteturas paralelas do tipo *multicore* atualmente seguem para uma barreira similar a encontrada pela *single core*, visto que, seu principal método de evolução, o aumento no número de núcleos em um mesmo *chip*, possui uma limitação, sendo esta o tamanho mínimo que um *transistor* pode alcançar, resultando no fim da possibilidade de alocação de mais núcleos em um mesmo espaço, tendo como única opção o aumento do tamanho do *chip*. Além disso, soluções que utilizam esse tipo de arquitetura, por exemplo, supercomputadores, estão encontrando o mesmo problema de escalabilidade entre dissipação de calor e ganho em desempenho que as *single core* encontraram no passado. A Figura 1 exemplifica esse problema, pois, utilizando a medida de performance *Floating-point Operations per Second (Flops)*, ou seja, a quantidade de operações de ponto flutuante que um computador realiza por segundo, compara seu crescimento com o aumento no número de núcleos dos supercomputadores com maior poder de computação do mundo ao passar dos anos, segundo a classificação do site TOP500[1], mostrando ao mesmo tempo a tendência em aumentar o número de núcleos e a difícil tarefa de encontrar escalabilidade entre esse aumento e o ganho em eficiência.

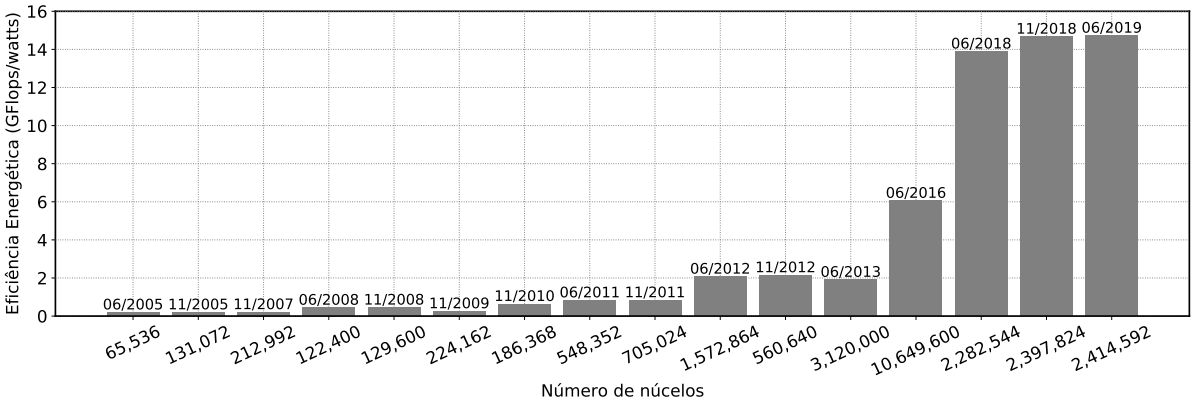
Com o interesse atual da comunidade científica em atingir o *Exascale* e, ao mesmo tempo, em computação voltada para a eficiência energética, pode-se então afirmar que as arquiteturas do tipo *multicore* não são mais uma solução viável para os supercomputadores. O alerta do Departamento de Defesa do Governo dos Estados Unidos (DARPA), uma das organizações mais importante do país, serve também como base para essa afirmação, o qual mostrou em um relatório (KOGGE et al., 2008) que, para ser viável, um supercomputador da era *Exascale* deve atingir uma eficiência energética de 50 GFlops/W, enquanto que, atualmente, o supercomputador com o maior poder de processamento do mundo atinge 14.719 GFlops/W e o de melhor eficiência energética atinge 16.876 GFlops/W. A Figura 2 mostra o crescimento na eficiência energética dos supercomputadores mais poderosos do mundo desde 2005[2], de acordo com o TOP500.

Buscando novos tipos de arquiteturas paralelas que apresentem as característi-

[1] Os dados do TOP500 estão disponíveis no seguinte site: https://www.top500.org/

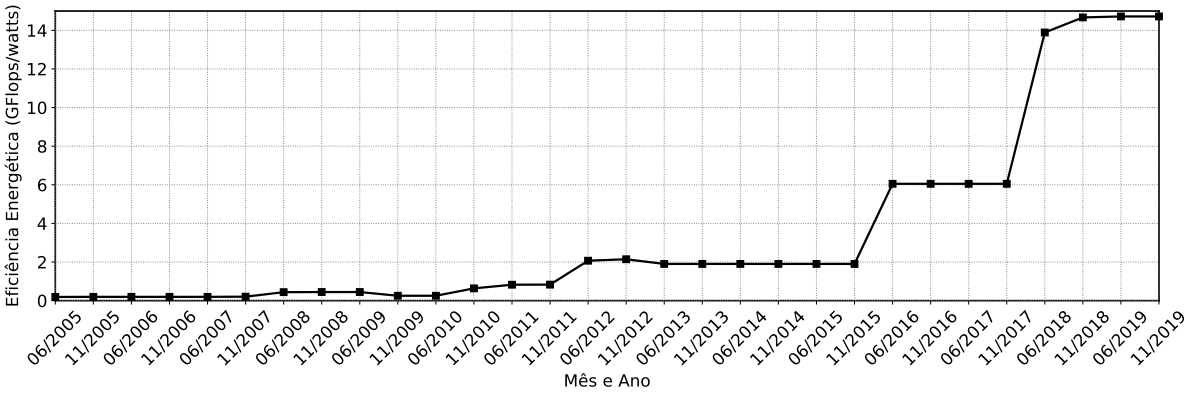
[2] Foi escolhido este ano como início pois nos anos anteriores a eficiência energética ainda era menor que 0.1 GFlops/W.

Figura 1 – Comparação da evolução da eficiência energética em relação ao número de núcleos do supercomputador número 1 do mundo ao passar dos anos segundo a classificação do TOP500.



Fonte: Gráfico desenvolvido pelo autor.

Figura 2 – Evolução da eficiência energética do supercomputador número 1 do mundo segundo a classificação do TOP500.



Fonte: Gráfico desenvolvido pelo autor.

cas faltantes no problema de balanceamento apresentado acima, pesquisadores da área de *High-Performance Computing (HPC)* realizaram diversos estudos voltados para essa questão, aplicando conceitos de *Green Computing* (KURP, 2008) no decorrer do desenvolvimento de suas soluções. Dentre estas soluções, temos o surgimento da classe de processadores *manycore* de baixa potência, como o MPPA-256 (DINECHIN et al., 2013), objeto de estudo deste trabalho, o Adapteva Epiphany (OLOFSSON; NORDSTRöM; ULABDIN, 2014), e o SW26010, utilizado no atual terceiro supercomputador mais poderoso do mundo, o *Sunway TaihuLight* (FU et al., 2016). Vale citar que o SW26010 desbancou em 2016 o supercomputador que assumia, desde 2013, a primeira posição do TOP500, obtendo duas vezes mais desempenho que esse e reduzindo em três vezes o consumo energético, explicando também o ganho elevado em eficiência em ambas as Figuras 1 e 2 no mês de junho de 2016.

Para avaliar o desempenho e consumo energético do MPPA-256, *Souza et al.* (SOUZA et al., 2016) propuseram o desenvolvimento do *benchmark* CAP Bench, o qual,

em sua primeira versão, utilizava uma *Application Programming Interface (API)* de comunicação síncrona entre processos denominada *MPPA Interprocess Communication API (IPC)* (DINECHIN et al., 2013). Essa API possui algumas deficiências, como o baixo nível de abstração, requerendo conhecimento prévio da arquitetura alvo para implementações paralelas eficientes, e a realização de sincronizações implícitas muitas vezes não necessárias nas operações de envio e recebimento de dados, o que leva a queda de desempenho da aplicação. Recentemente, uma nova API de comunicação denominada *MPPA Asynchronous Communication API (ASYNC)* foi desenvolvida pela empresa fabricante do MPPA-256. Esta API fornece um maior nível de abstração e a possibilidade de realização de comunicações assíncronas, aumentando o potencial de desempenho das aplicações. Portanto, mostra-se necessária a realização de um estudo para verificar os reais benefícios desta nova API.

1.1 OBJETIVOS

Com base no que foi exposto, são apresentados abaixo o objetivo geral e os objetivos específicos deste trabalho.

1.1.1 Objetivo Geral

Este trabalho tem por objetivo geral a adaptação da implementação das aplicações do CAP Bench para a API ASYNC, assim como um estudo do desempenho das aplicações em comparação com a implementação original com a API IPC no MPPA-256.

1.1.2 Objetivos Específicos

- Estudar as APIs de comunicação ASYNC e IPC no MPPA-256;
- Avaliar os custos e benefícios do MPPA-256 em relação ao desempenho e gasto energético, assim como sua utilidade para a Computação Sustentável (*Green Computing*);
- Comparar as APIs ASYNC e IPC a fim de prover métricas precisas para a escolha da melhor API de comunicação.

1.2 CONTRIBUIÇÕES DO TRABALHO

Este trabalho é continuação de um projeto de iniciação científica desenvolvido pelo autor, o qual resultou em um artigo publicado na Escola Regional de Alto Desempenho da Região Sul no ano de 2019:

- ORDINE, D. G. V.; PODESTA JUNIOR, E. ; PENNA, P. H. ; CASTRO, M. **Otimização de Aplicações do CAP Bench para o Processador MPPA-256.** In: Escola Regional de Alto Desempenho da Região Sul (ERAD/RS), 2019, Três de Maio. Anais da Escola Regional de Alto Desempenho da Região Sul (ERAD/RS). Porto Alegre: Sociedade Brasileira de Computação (SBC), 2019.

1.3 ORGANIZAÇÃO DO TRABALHO

Este trabalho está dividido da seguinte forma. O Capítulo 2 mostra os conceitos teóricos que foram utilizados para a produção deste trabalho. O Capítulo 3 apresenta alguns trabalhos relacionados a este. O Capítulo 4 apresenta a proposta do trabalho. O Capítulo 5 apresenta os resultados obtidos. Por fim, o Capítulo 6 conclui este trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados conceitos fundamentais sobre arquiteturas paralelas (Seção 2.1) e sobre o MPPA-256 (Seção 2.2). Então, são discutidas algumas tecnologias de programação paralela bastante difundidas na literatura, além das tecnologias existentes para o processador MPPA-256 (Seção 2.3).

2.1 ARQUITETURAS PARALELAS

Segundo *Tanenbaum et al.*, existem dois tipos de sistemas com múltiplos processadores: os multiprocessadores e os multicomputadores (TANENBAUM; BOS, 2014). Esta seção apresenta as principais características de cada uma dessas abordagens.

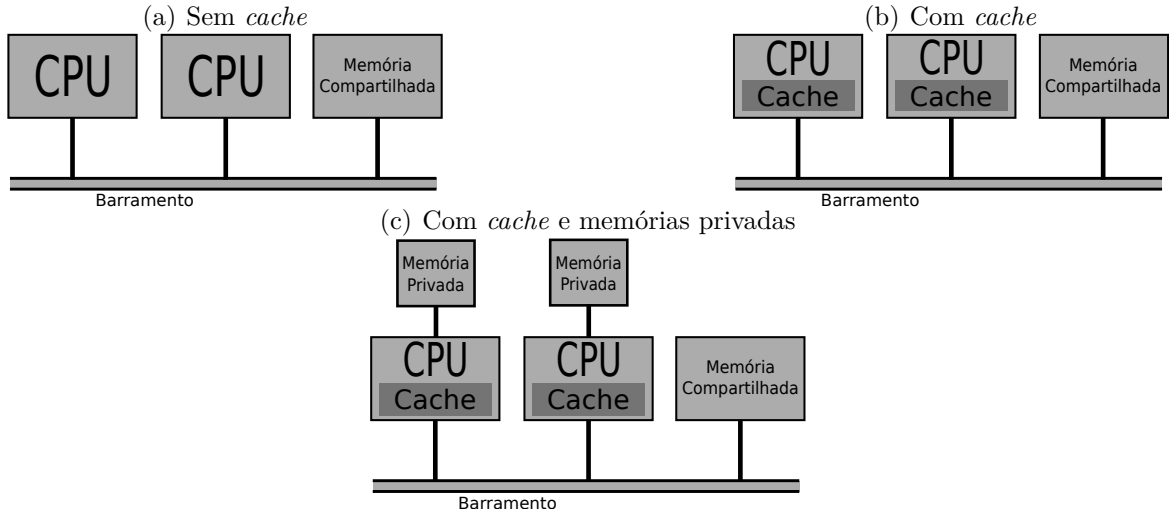
2.1.1 Multiprocessadores

A principal característica de uma arquitetura multiprocessador é o acesso compartilhado ao barramento de memória do sistema, a *Random-Access Memory (RAM)*, por diversas *Central Processing Units (CPUs)* (também chamados de núcleos em português ou *cores* em inglês). Cada programa em execução nesta arquitetura possui seu próprio espaço de endereçamento na RAM, o qual é compartilhado por todas as *threads* que o compõe. Toda a comunicação entre *threads* é feita de maneira implícita, através de operações de escrita e leitura. Devido ao compartilhamento de memória, há a possibilidade de ocorrer problemas de concorrência quando duas ou mais *threads* de um mesmo programa escrevem um valor em uma mesma posição de memória simultaneamente, ocasionando assim problemas de condição de corrida.

Multiprocessadores são também classificados em dois tipos, de acordo com a latência de acesso à memória. Nos multiprocessadores com acesso uniforme a memória, *Uniform Memory Access (UMA)*, a latência de acesso à memória é constante, independentemente do processador e do endereço de memória que está sendo acessado. Por outro lado, nos multiprocessadores com acesso não uniforme a memória, *Nonuniform Memory Access (NUMA)*, a latência é variável e depende da distância entre o processador e o bloco de memória que está sendo acessado.

A arquitetura mais simples de um multiprocessador UMA, exemplificada na Figura 3(a), envolve um único barramento conectando duas ou mais CPUs a um módulo de memória, permitindo que todas as CPUs realizem operações de leitura e escrita neste módulo. Quando uma CPU necessita ler alguma palavra da memória, primeiramente ela verifica se o barramento está ocupado. Caso não, informa à memória, através do barramento, qual endereço deseja obter o valor, aguardando o recebimento deste pelo mesmo barramento. Caso sim, a CPU aguarda a liberação do barramento. Para uma pequena quantidade de CPUs, o tempo de espera médio para o acesso ao barramento tende a ser

Figura 3 – Diferentes esquemas possiveis de um multiprocessador UMA baseado em barramento.



Fonte: Imagens desenvolvidas pelo autor, adaptadas de *Tanenbaum et al.* (TANENBAUM; BOS, 2014).

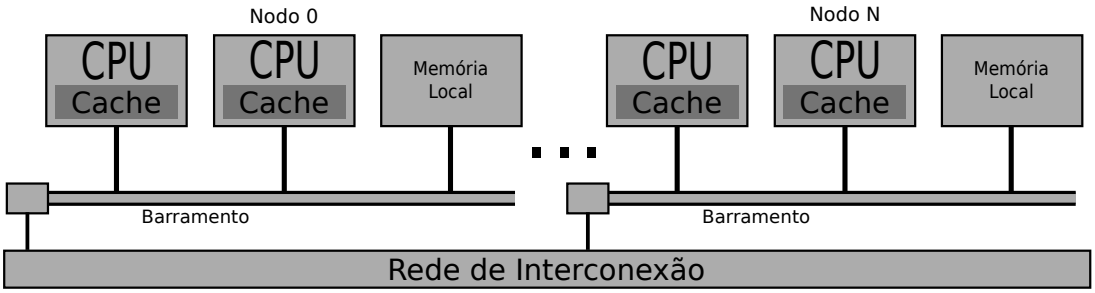
pequeno e tolerável. Porém, quando elevam-se em algumas dezenas o número de CPUs observa-se o principal problema deste exemplo de arquitetura UMA: a ociosidade, por muito tempo, de grande parte das CPUs, enquanto aguardam pelo acesso ao barramento.

Adicionar *caches* às CPUs, como na Figura 3(b), é uma solução para reduzir o gargalo imposto sobre o barramento, já que agora valores podem ser lidos diretamente da *cache* local, a qual está muito mais próxima da CPU e possui tempo de acesso muito menor. Outra possibilidade é adicionar, além das *caches*, memórias privadas locais, como na Figura 3(c). Compiladores podem colocar nessas memórias todos os dados que são somente de leitura, por exemplo, constantes e o código do programa, utilizando assim esta segunda configuração de forma otimizada. Ambas configurações removem grande parte do tráfego no barramento.

A adição de *caches* impõe o uso de protocolos de coerência para que não haja inconsistência entre os valores de um mesmo endereço de memória nas diferentes *caches*. Primeiramente, para otimizar as operações de leitura, quando uma palavra é referenciada, todo o bloco que contém essa palavra, geralmente de 32 ou 64 *bytes*, é colocado na *cache*. Já para garantir a coerência, cada bloco é marcado como sendo somente de leitura, podendo assim estar presente em outras *caches*, ou de leitura e escrita, não devendo estar presente em nenhuma outra *cache* neste caso. Quando uma CPU tenta alterar um valor que está presente em outras *caches* além de sua própria, o *hardware* do barramento informa essa operação às outras *caches*, as quais tratam esse contexto de duas formas. Caso o valor da *cache* seja o mesmo em memória, podem simplesmente descartá-lo, buscando o novo valor na memória se necessário. Caso outra *cache* tenha um valor diferente daquele em memória, é necessário ou salvá-lo na memória ou transferi-lo diretamente para a *cache* que solicitou a operação de escrita.

Quando necessita-se de um número de processadores na ordem das centenas, a

Figura 4 – Esquema genérico de um multiprocessador NUMA.



Fonte: Imagem desenvolvida pelo autor, adaptada de *Tanenbaum et al.* (TANENBAUM; BOS, 2014).

arquitetura UMA acaba sendo inviável. Assim, introduz-se a arquitetura NUMA, trazendo com ela a ideia de diferentes tempos de acesso para diferentes posições de memória. Multiprocessadores NUMA provêm essa escalabilidade implementando um espaço de endereçamento único para todas as CPUs através de uma rede de interconexão, como na Figura 4. Isso significa que o tempo de acesso a uma posição de memória será totalmente dependente do local que ela se encontra relativo ao local da CPU que requisitou este acesso. Logo, outra propriedade desta arquitetura é o acesso mais rápido à memória local de um ou um conjunto de CPUs, em comparação com o acesso à memória remota. Vale salientar que programas desenvolvidos para multiprocessadores UMA conseguem ser executados em arquiteturas NUMA, devido a ambas possuírem um espaço de endereçamento único. Porém, estes programas irão obter performance inferior, já que não foram otimizados para considerar as diferenças de tempo entre acesso à memória local e remota.

A medida que o tamanho de um *transistor* diminui, o número de *transistors* em um *chip* tende a aumentar. Diversas soluções exploram o que fazer com este número crescente de *transistors*, por exemplo, adicionar *caches* poderosas de muitos *megabytes* ou colocar duas ou mais CPUs neste mesmo *chip*. Em certo ponto, o aumento no tamanho da *cache* traz pouquíssimo ganho em porcentagem de *hit* (quantidade de vezes que é possível buscar um dado diretamente na *cache*), mostrando assim que o investimento no paralelismo trazido pelos múltiplos núcleos como recurso para ganho em desempenho é uma opção a se considerar. Assim, *chips multicore* são uma mescla de múltiplas CPUs com múltiplos níveis de *cache* inseridos em um espaço muito menor que um multiprocessador, sendo por isso também chamados de *Chip MultiProcessadoress (CMPs)*.

Apesar de serem parecidos, existem algumas diferenças entre os CMPs e os multiprocessadores. Primeiramente, em muitos CMPs ocorre o compartilhamento da *cache* nível 2 ou 3 entre suas CPUs, o que não acontece nos multiprocessadores, que possuem *caches* totalmente privadas em todos os níveis. Além disso, a probabilidade de que falhas em componentes compartilhados levem a impossibilidades em múltiplas CPUs ao mesmo tempo é muito maior nos CMPs, devido a proximidade de conexão das CPUs. Por fim, existem *chips multicore* em que todos os núcleos são feitos para atender a uma ampla gama de contextos, enquanto que em outros, além das CPUs principais, existem tam-

bém núcleos específicos para alguns problemas, como decodificação de áudio e vídeo ou interfaces de rede.

Apesar de não haver uma barreira de distinção entre um *chip manycore* ou *multicore*, pode-se chama-lo de *manycore* quando a perda de um núcleo tem um pequeno impacto na performance total do *chip*. Um problema com arquiteturas *manycore* é a escalabilidade entre manter as *caches* de todas as CPUs coerentes e ainda assim elevar o desempenho ao elevar o número de núcleos. Cientistas da área de HPC temem que essas duas variaveis não escalem proporcionalmente, tornando o custo de gerenciar essas *caches* tão alto que a adição de um novo núcleo de pouco ajudara no aumento em performance. Este problema é também conhecido como a barreira de coerência (*coherency wall*) (TANENBAUM; BOS, 2014).

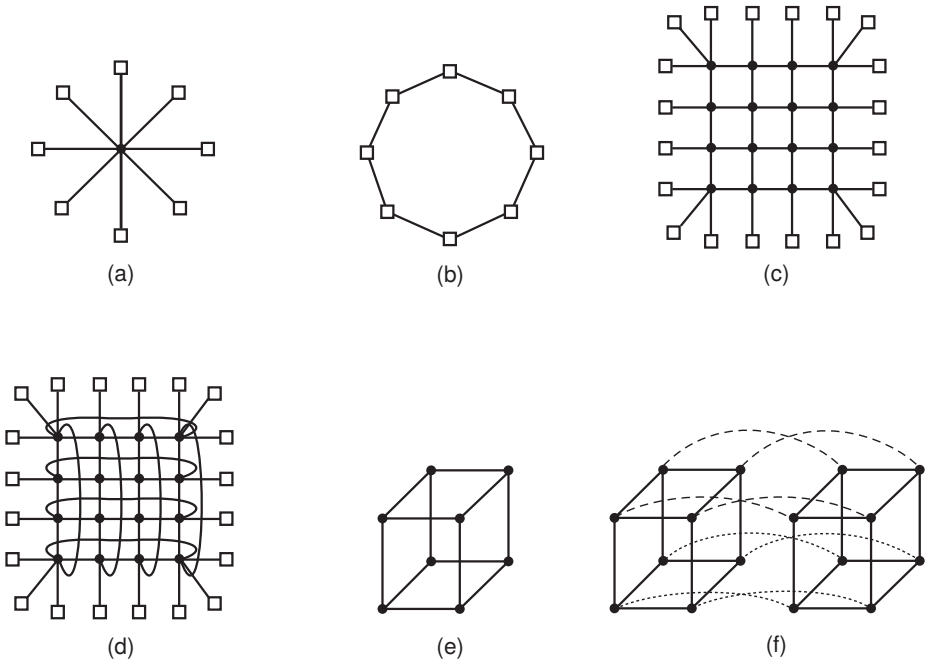
Para o futuro dos *manycores*, espera-se processadores que invistam mais na comunicação entre CPUs através da troca de mensagens extremamente rápidas via *hardware* e através de uma memória compartilhada, deixando de lado parte da coerência de *cache*. Uma Unidade de Processamento Gráfico (GPU) é um dos exemplos mais comuns de um processador *manycore*, possuindo milhares de pequenos núcleos especializados na rápida execução de cálculos e sem uma lógica complexa de *cache*, ou seja, priorizam o processamento. Desta maneira, GPUs são excepcionais para a execução paralela de pequenas tarefas, como a renderização de *frames* para jogos. Programar para uma GPU é uma tarefa difícil e muitas vezes algumas linguagens de programação especiais são utilizadas, como a OpenGL ou a CUDA, da NVIDIA. Essa dificuldade se dá, principalmente, pelo fato dos núcleos de uma GPU executarem exatamente a mesma instrução em diferentes fatias de um dado, ou seja, pelo fato da GPU ser uma máquina *Single Instruction Multiple Data (SIMD)*.

2.1.2 Multicomputadores

Multicomputadores surgiram na dificuldade de aumentar o poder de processamento de um multiprocessador quando se atinge grandes escalas em relação ao número de núcleos. Ao contrário dos multiprocessadores, multicomputadores não compartilham memória, sendo relativamente fáceis de se construir, tendo como componente principal um computador com uma placa de rede de alta performance, sem mouse, teclado e monitor. Neste sistema, também chamado de *Cluster Computers* ou *Cluster Of Workstations* (COWs), é necessário um *design* inteligente da rede que irá conectar os computadores para que se possa obter um alto desempenho.

Um nó de um multicomputador consiste em um computador, com uma CPU, memória, placa de rede e um HD. Diversas são as topologias possíveis para a rede que conecta os nós, como mostrado na Figura 5. Sistemas pequenos utilizam-se de apenas um *switch* para conectar os nós entre si, os quais são organizados em forma de estrela, como na Figura 5(a). Também é possível organizar os nós em forma de anel, onde cada nó se

Figura 5 – Tipos de topologias de rede de multicomputadores.



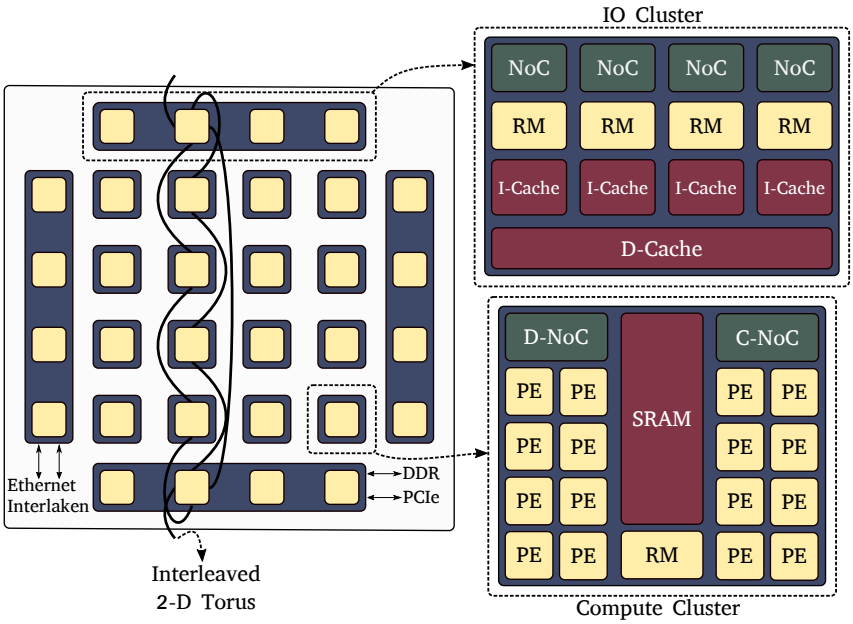
Fonte: (TANENBAUM; BOS, 2014).

conecta aos nós da sua esquerda e direita, como na Figura 5(b), eliminando a necessidade de um *switch*. Porém, o problema dessas arquiteturas é a escalabilidade, a qual dificulta o ganho em desempenho a medida que se aumenta o número de nós devido ao tempo de viagem dos dados entre nós.

Topologias mais complexas, como a malha (*grid* ou *mesh*), mostrada na Figura 5(c), ou a *double torus*, mostrada na Figura 5(d), são mais escaláveis do que as apresentadas anteriormente. Nessas topologias, os nós são conectados em *switches* e estes são conectados entre si, formando um *layout* de malha no sistema. Dentre as duas citadas, a *double torus* é mais escalável devido a conexão entre nós nas arestas da rede, trazendo assim conexões extras ao sistema, o que aumenta a tolerância a faltas e o desempenho, já que o caminho entre estes nós se torna menor. Este tipo de rede possui uma propriedade chamada de diâmetro, que é o caminho mais longo entre dois nós. Para topologias bidimensionais como a malha, o diâmetro aumenta proporcionalmente a raiz quadrada do número de nós. *Layouts n* dimensionais, como mostrado na Figura 5(e) (tridimensional) e na Figura 5(f) (quadrimensional), são ainda mais escaláveis, já que o diâmetro diminui à medida que se aumenta o número de dimensões da rede, tendo como única desvantagem o custo elevado, devido ao grande número de ligações presentes entre nós e *switches*.

A comunicação entre processos rodando em diferentes CPUs, num multicomputador, ocorre através da troca de mensagens entre estes. Basicamente, o Sistema Operacional (SO) presente no multicomputador é o responsável por realizar essa troca, através de funções acessíveis somente a ele. Porém, bibliotecas podem fornecer abstrações a essas

Figura 6 – Visão arquitetural simplificada do MPPA-256.



Fonte: (PENNA et al., 2018).

funções, tornando a troca de mensagens também disponível para os processos usuário e as simplificando, visto que abstraem toda uma lista de invocações de funções em uma única função. Essa troca de mensagens pode ser reduzida a duas funções, chamadas de *send* e *receive*. A função *send* é responsável por enviar uma mensagem de uma CPU para outra, passando parâmetros como o destino da mensagem e o endereço onde aquela mensagem se encontra. Já a função *receive* é responsável por receber a mensagem, tendo como parâmetros, por exemplo, o endereço de onde a mensagem será lida e o endereço onde será armazenada. Por fim, essas funções podem ser síncronas, bloqueando o processo que envia ou recebe a mensagem até que a operação seja concluída, ou assíncronas, não bloqueando o processo que realizou tal operação.

2.2 MPPA-256

Desenvolvido pela empresa francesa Kalray, o MPPA-256 é um processador de baixa potência que reflete o estado da arte dos *manycores* (DINECHIN et al., 2013). Uma visão geral do processador é mostrada na Figura 6. 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 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 *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 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), se assemelhando, neste sentido, a um multicomputador.

2.3 DESENVOLVIMENTO DE APLICAÇÕES PARALELAS

Durante o domínio de processadores *single core* no mercado, o aumento de desempenho de aplicações era fortemente influenciado pelo aumento da frequência de *clock* dos processadores. Isso removia parte da responsabilidade do desenvolvedor em implementar melhorias na aplicação, já que bastava o *upgrade* no *hardware* para obter essas melhorias. Nestes processadores, instruções são executadas de forma sequencial em um único núcleo. Já em processadores *multicore* e *manycore*, existem múltiplos núcleos executando diferentes instruções, possivelmente, de diferentes programas. Essa divisão de tarefas pode levar a diversos novos problemas, como *deadlock* ou condições de corrida, como dito anteriormente.

Diversas APIs voltadas a programação paralela foram criadas para simplificar tanto a solução desses problemas como o desenvolvimento de aplicações que exploram o paralelismo de maneira eficaz. A seguir serão apresentadas duas das mais conhecidas APIs para multiplataformas, a *Open Multi-Processing (OpenMP)* e o *Message Passing Interface (MPI)*, assim como duas APIs específicas para o MPPA-256, a ASYNC e a IPC.

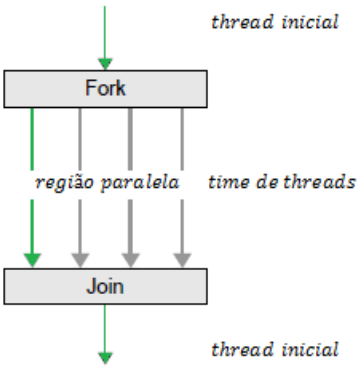
2.3.1 Bibliotecas multiplataforma

2.3.1.1 Open Multi-Processing

A OpenMP é uma das bibliotecas mais utilizadas para implementação de aplicações paralelas nas linguagens C, C++ e Fortran. Sua fama vem, principalmente, da abstração que a API fornece, sendo possível ser utilizada em inúmeras plataformas. Por ser uma API voltada ao *multithreading*, utiliza-se do compartilhamento de memória entre as *threads* de um mesmo programa para criar beneficios ao desenvolvedor, como variáveis de ambiente.

Esta biblioteca é centrada no modelo *fork-join* (Figura 7), onde, em determinados momentos do fluxo de execução de uma aplicação, temos a *thread* principal instanciando outras *threads* através de diretivas de compilação fornecidas pela própria API, as quais serão executadas de forma independente, paralelamente ou concorrentemente entre si.

Figura 7 – Esquema do modelo *fork-join*.



Fonte: (ALMEIDA et al., 2019).

Listagem 1 – Execução de um *loop* de forma paralela.

Fonte: o autor.

Listagem 2 – Leitura e armazenamento seguro em variável compartilhada entre *threads*.

Fonte: o autor.

A *thread* principal também realiza uma sincronização com as *threads* criadas através de uma barreira implícita, aguardando o término de todas as *threads* para continuar com a execução. A criação de novas *threads* é feita ao atingir uma região paralela, definida por algumas diretivas de compilação, como `#pragma omp parallel for`, que paraleliza a execução de um *loop* entre múltiplas *threads*. O algoritmo 1 produz, de forma paralela, um *array* com 10 posições, onde cada posição armazena um inteiro igual ao índice daquela posição.

As diretivas `private`, `default` e `reduction` permitem, na sequência, definir quais variáveis terão escopo privado, qual será o escopo padrão das variáveis, e qual variável será feito uma redução sobre e qual será o tipo de redução (reduções são operações primitivas seguras sobre uma variável compartilhada entre múltiplas *threads*). O algoritmo 2 é parte de uma das aplicações do CAP Bench, a *Friendly Numbers*, e utiliza essas três diretivas para realizar uma contagem paralela, a qual será armazenada na variável `partial_friendly_sum`. Como é possível observar, com a adição de uma única linha paraleliza-se a execução de uma computação, sendo esta a maior vantagem da OpenMP.

Também pode-se definir qual será o nível de trabalho de cada *thread* com a diretiva `schedule`. Através desta diretiva, definem-se três tipos de escalonamentos para as *threads*: `static`, `dynamic` e `guided`. Com o `static`, todas as *threads* irão receber a mesma quantidade de trabalho, sendo este o escalonamento padrão. Já a diretiva `dynamic` é utilizada quando as iterações podem ter uma grande diferença no seu tempo de execução, realizando uma atribuição dinâmica de tarefas, onde cada *thread* recebe uma nova tarefa

Listagem 3 – Exemplo de uma aplicação usando a MPI.

Fonte: o autor.

ao terminar a iteração atual. Por fim, `guided` é similar ao `dynamic`, porém, a *thread* começa recebendo um grande número de iterações, mas a medida que a execução avança, o número de iterações atribuídas a cada *thread* é reduzido para melhorar o balanceamento de carga.

2.3.1.2 Message Passing Interface

Diferentemente da OpenMP, o MPI é baseada no modelo *Single Program, Multiple Data (SPMD)*, onde um mesmo programa é executado por diferentes processos, cada qual com o seu próprio espaço de endereçamento. Assim, o MPI é usada em supercomputadores para abstrair a difícil tarefa de implementar nestes o paralelismo através da troca de mensagens entre processos em baixo nível. Com esta API de alto nível, implementar o envio e recebimento de mensagens torna-se algo tão simples como chamar uma função, já que o MPI abstrai diversas etapas em uma única função. A API também adiciona identificadores únicos e um grupo de comunicação para cada processo, os quais são usados como parâmetros em diversas de suas funções. Além disso, determinadas funções podem ser executadas de forma síncrona ou assíncrona, aumentando o desempenho da aplicação.

Um fluxo simples de implementação utilizando o MPI começa com a função `MPI_Init()`, que inicia o ambiente de execução MPI. Na sequência, obtém-se o *id* de um processo através da função `MPI_Comm_rank()`, a qual recebe um comunicador, geralmente o comunicador padrão `MPI_COMM_WORLD` que inclui todos os processos MPI, como primeiro parâmetro e o endereço da variável que será armazenado o *id* do processo como segundo. Também é possível obter o número máximo de processos em um grupo de comunicação através da função `MPI_Comm_size()`, a qual recebe no primeiro parâmetro um comunicador e no segundo o endereço da variável em que será armazenado este número. Além disso, existem as funções de envio e recebimento de mensagens, `MPI_Send()` e `MPI_Recv()`, as quais recebem parâmetros como o *buffer* de dados sobre o qual será realizado a leitura ou armazenamento da mensagem, a quantidade de dados, o tipo do dado e o *id* do processo que realizará o envio ou recebimento da mensagem, além de alguns parâmetros adicionais. Por fim, a função `MPI_Finalize()` termina a execução do ambiente MPI.

Ao contrário das funções `MPI_Recv()` e `MPI_Send()`, que são voltadas para comunicação entre dois processos, funções como `MPI_Bcast()` e `MPI_Barrier()` são feitas para que haja comunicação entre um grupo de processos. Com a `MPI_Bcast()` define-se o envio de uma mensagem de um processo para todos os outros processos associados a um grupo de comunicação. Já com a `MPI_Barrier()` cria-se uma barreira, onde um certo

processo, ao chegar nesta barreira, aguarda todos os outros processos de um grupo de comunicação chegarem nela antes de continuar sua execução.

O algoritmo 3 mostra um exemplo de implementação simples usando o MPI. Neste exemplo, na linha 10 o processo com *id* igual a 0 envia um *broadcast* para todos os outros processos, os quais respondem na linha 14 ao processo de *id* 0, que recebe estas respostas nas linhas 16-18. É importante notar que as linhas 16-18 são executadas somente pelo processo com *id* igual a 0, enquanto que as linhas 13-14 são executadas por processos com *id* maior que 0.

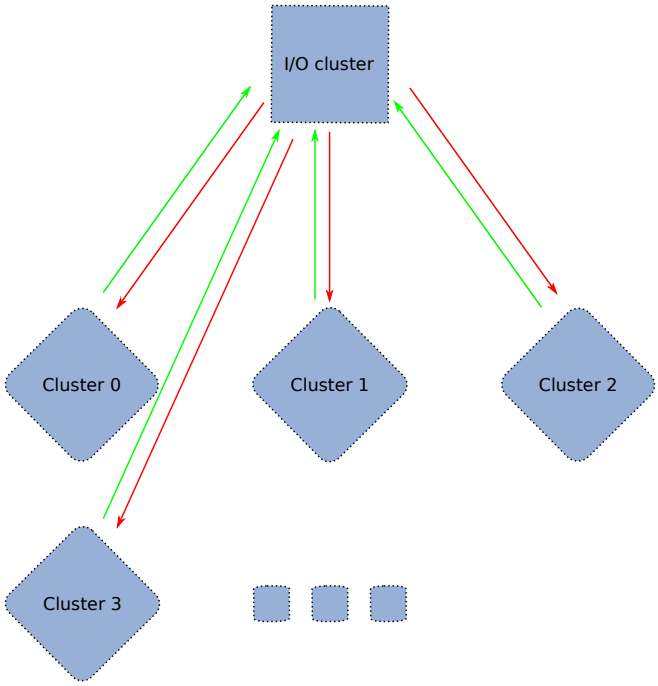
2.3.2 Bibliotecas específicas para o MPPA-256

Na arquitetura do MPPA-256, os *clusters* de E/S gerenciam os CCs, em um modelo conhecido como mestre/escravo, como exemplificado na Figura 8. os CCs e *clusters* de E/S executam binários diferentes e fornecem algumas abstrações distintas de programação. Ao ativar um *cluster*, associa-se um processo a ele, logo, a inicialização de CCs acontece através de um processo mestre citando explicitamente em sua implementação o caminho do código binário que será executado no CC. Cada processo escravo pode criar até 16 *threads* do tipo POSIX, cada uma sendo executada em um núcleo diferente dentro daquele *cluster*. A criação de *threads* nos CCs pode ser feita explicitamente com uso de POSIX *threads* ou de maneira transparente com uso do OpenMP. Infelizmente, o *runtime* do MPPA-256 não fornece uma implementação do MPI para comunicação entre CCs. Portanto, é necessário utilizar, neste caso, interfaces de programação proprietárias, como a IPC (DINECHIN et al., 2013) e, mais recentemente, ASYNC.

Com a IPC, um processo mestre em execução em um dos quatro *clusters* de E/S pode lançar outros processos nos CCs passando argumentos tradicionais, como `argc` e `argv`. Porém, toda a lógica de comunicação e sincronização sobre a rede NoC é abstraída através da realização de operações sobre descritores de arquivos e através do uso das diretivas do padrão POSIX IPC. O *design* da API é baseado no modelo *pipe-and-filters* (LAU; WANG, 2007), onde os processos POSIX são os componentes atômicos e os objetos de comunicação são as conexões. Além disso, os objetos de comunicação possuem portas de transmissão e recepção de dados, chamadas de portais, as quais são abertas em dois possíveis modos, `O_WRONLY` (somente escrita) ou `O_RDONLY` (somente leitura), através da função `mppa_open()`, que recebe no primeiro parâmetro o *path* para o descritor de arquivo de um determinado portal e no segundo o modo no qual será aberto.

Abrindo portais de comunicação específicos para cada *cluster*, um para leitura e outro para escrita, e definindo quais *clusters* ficarão em cada ponta desses portais, pode-se implementar uma aplicação que segue um fluxo semelhante à Figura 9. Nela, o processo mestre lança dois processos escravos, os quais abrem portais que definem um caminho para o *cluster* que os lançou. Um destes processos não faz uso da função `mppa_read()` pois recebeu todos os dados necessários através dos parâmetros passados

Figura 8 – Fluxo de uma aplicação seguindo o modelo *mestre/escravo* no MPPA-256.

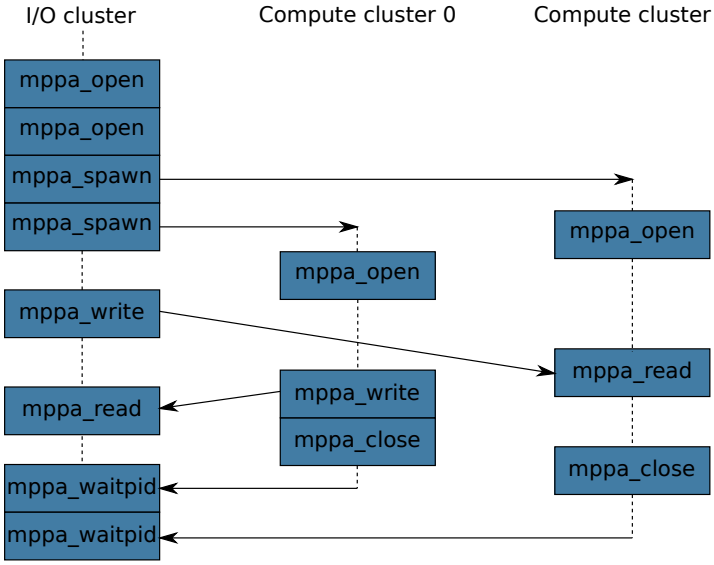


Fonte: Documentação do MPPA sobre IPC.

pelo processo de E/S na chamada a função `mppa_spawn()`. Assim, este processo só precisa realizar a computação sobre aqueles dados e devolver o resultado para o *cluster* de E/S através do portal que os liga, usando a função `mppa_write()`. Já o outro *cluster* faz uso da função `mppa_read()` para ler dados enviados pelo processo mestre, mas não envia de volta nenhuma informação. Após realizar suas tarefas, os CCs precisam fechar os descritores de arquivo dos portais que foram abertos e usados, o que é feito através da função `mppa_close()`. Por fim, o processo mestre precisa esperar o término da execução dos processos escravos antes de continuar a sua execução, o que é feito com a função `mppa_waitpid()`.

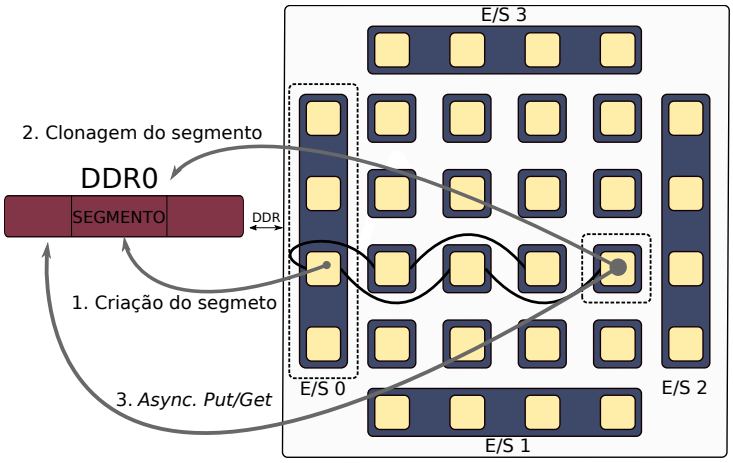
Tendo em vista a complexidade envolvida na utilização da API IPC, a fabricante do MPPA-256 desenvolveu recentemente uma nova API que fornece uma maior abstração e facilidade de programação, denominada ASYNC. Baseada em princípios de comunicação unilateral, os quais são aplicados em diversas APIs usadas em supercomputadores, como a API PNNL ARMCI (NIEPLOCHA et al., 2006), a ASYNC é organizada em diversos conceitos que permitem abstrair ainda mais a implementação de paralelismo no MPPA-256. Dentre esses conceitos, pode-se citar três mais importantes: **(i)** segmentos de memória, ou seja, memória que não é diretamente acessível através de um dos núcleos de um *cluster*; **(ii)** operações `PUT/GET`, as quais possuem diversos modos de realização, como linear, espaçada, por etapas e em blocos 2D ou 3D; e **(iii)** o modo assíncrono como as operações `PUT/GET` são realizadas, significando que essas funções retornam assim que completam a escrita ou leitura sobre uma porção de memória, não realizando qualquer

Figura 9 – Fluxo de uma aplicação usando funções do tipo POSIX da IPC no MPPA-256.



Fonte: Documentação do MPPA sobre IPC.

Figura 10 – Fluxo de uma aplicação usando a API ASYNC no MPPA-256.



Fonte: o autor.

tipo de sincronização com algum possível *cluster* que irá receber aquela informação.

Na Figura 10, onde os *clusters* são representados por quadrados amarelos, é demonstrado o fluxo de implementação do paralelismo entre um processo mestre e um CC ao usar a ASYNC. Primeiramente um *cluster* de E/S cria um segmento sobre uma porção de dados localizados na sua memória local através da função `mppa_async_segment_create()`, associando a ele um identificador único do tipo `unsigned long long`. Após a criação do segmento, um *cluster* de computação deve clonar esse segmento através da função `mppa_async_segment_clone()`, usando, entre outros parâmetros, o identificador do segmento desejado. Chamadas essas duas funções, operações do tipo `PUT/GET` podem ser realizadas sobre o segmento através das funções `mppa_async_put()` e `mppa_async_get()`. Além das operações mostradas na figura, é preciso inicializar o contexto antes de realizar

Listagem 4 – Definição das macros de sincronização em um *cluster* de E/S.

Fonte: o autor.

Listagem 5 – Definição das macros de sincronização em um *cluster* de computação.

Fonte: o autor.

qualquer uma dessas operações, o que é feito com a função `mppa_async_init()`, e finalizar este contexto após realizar todas as operações necessárias, o que é feito com a função `mppa_async_final()`.

Operações `PUT` devem ser feitas com cuidado, caso contrário podem sobrescrever operações anteriores que ainda não foram lidas por algum *cluster*. Na versão ASYNC do CAP Bench, sincronizações são feitas através de operações `poke` sobre um segmento padrão construído na inicialização do contexto da API. Essas sincronizações são definidas através de macros, que são expandidas para chamadas as funções `mppa_async_poke()` e `mppa_async_evalcond()`, as quais, respectivamente, alteram um valor remoto do tipo `long long` e aguardam uma expressão booleana se tornar verdadeira antes de continuar a execução. Os algoritmos 4 e 5 mostram como são definidas essas macros em um *cluster* de E/S e em um CC, respectivamente.

3 TRABALHOS CORRELATOS

O tema central deste trabalho é de grande interesse por parte da comunidade científica de HPC, existindo inúmeras pesquisas voltadas a medir o desempenho de processadores *multicore*, *manycore* e *chips* FPGA, as quais são diretamente relacionadas a esta. Neste capítulo serão apresentados algumas pesquisas científicas voltadas a medição de parâmetros de desempenho de processadores que utilizam uma dessas duas arquiteturas.

Nabi et al. (NABI; VANDERBAUWHEDE, 2018) implementaram uma versão do *benchmark* STREAM para arquiteturas FPGA, GPUs e CPUs, chamando-o de STREAM-Multiplataforma, para dar foco a portabilidade de sua versão. O *benchmark* STREAM original foi desenvolvido em 1995 por *McCalpin et al.* (MCCALPIN, 1995) e é um dos *benchmarks* mais utilizados para medir a largura de banda da memória em computadores. A principal contribuição deste trabalho é a introdução de um *benchmark* que faça esse tipo de medição em dispositivos FPGA, já que faltam *benchmarks* desse tipo para estes dispositivos. Além disso, parâmetros genéricos e específicos para certas arquiteturas foram introduzidos na nova versão, os quais afetam diretamente a largura de banda da memória. Os autores afirmam que, sendo esta largura de banda um dos principais gargalos das aplicações HPC (ASANOVIć et al., 2006), este *benchmark* possibilita mais um passo na direção de tornar convencional o uso de placas FPGA.

Já *Bennett et al.* (BENNETT et al., 2016) desenvolveram um *benchmark* para avaliar a aptidão de um supercomputador ao executar simulações de modelos da física quântica que vão além do modelo padrão. Grande parte dos *benchmarks* dessa área são derivados do modelo *Quantum ChromoDynamics (QCD)*, que descreve a forte interação entre *quarks* e *gluons*. Porém, este modelo não tem a flexibilidade necessária para examinar outros que o estendem, como é o caso dos modelos da Teoria de Gauge, que inclui o QCD. Assim, a inovação deste *benchmark* é ser derivado de implementações que simulam modelos da Teoria de Gauge, que é mais extensa.

Dentre os *benchmarks* clássicos da comunidade de HPC, o *NAS Parallel Benchmark* (BAILEY et al., 1991) se destaca por ter sido concebido puramente através de algoritmos, descrito por seus autores como um *benchmark* com especificação de papel e caneta. Segundo os autores, optar por este caminho é uma maneira de evitar grande parte das dificuldades que uma implementação convencional de um *benchmark* para sistemas HPC enfrenta. Originalmente 7 *kernels* foram descritos para o NAS, os quais simulam diferentes cargas de trabalho encontradas em aplicações HPC. Todos estes *kernels* usam ou uma API de comunicação entre processos, como a MPI, ou uma API para sistemas com memória compartilhada, como a OpenMP.

Outro *benchmark* clássico para aplicações CUDA + MPI é o OMB-GPU (BUREDDY et al., 2012), que estende o *OSU Micro-Benchmarks (OMB)* adicionando *kernels* que avaliam a performance de comunicações MPI em sistemas de *clusters* de GPUs. O

OMB-GPU surgiu durante o crescimento no uso de GPUs em supercomputadores, devido a necessidade de um *benchmark* padronizado para avaliar as novas bibliotecas MPI que foram desenvolvidas para suportar comunicações MPI nesses novos sistemas. Com o desenvolvimento da Memória Unificada (UM) para aplicações CUDA (tecnologia que mescla *software* e *hardware* para criar um espaço de endereço único e acessível por qualquer GPU ou CPU de um sistema), houve a necessidade de um novo *benchmark* específico para os sistemas que passaram a utilizar essa tecnologia, já que os *benchmarks* clássicos da época, como o OMB ou o OMB-GPU, não eram aptos para isso, pois não capturavam corretamente o comportamento do driver CUDA neste novo contexto. Assim, *Manian et al.* desenvolveram o OMB-UM (MANIAN et al., 2019), uma extensão do OMB para suportar aplicações CUDA + MPI + UM.

Diferentemente dos *benchmarks* anteriores, *Kelly et al.* (KELLY et al., 2011) desenvolveram o *benchmark* NWSC para testar um supercomputador especifico, o *NCAR-Wyoming Supercomputing Center (NWSC)*, que estava sendo construído para suprir as necessidades computacionais de aplicações HPC de 5 domínios científicos: **(i)** Física Atmosférica; **(ii)** Clima Espacial; **(iii)** Oceanografia; **(iv)** Modelagem de Subsuperfícies; e **(v)** Ciência Computacional, Estatística e Matemática. Este supercomputador seria o próximo sistema HPC do Centro Nacional de Pesquisas Atmosféricas dos Estados Unidos, suportando grande parte da necessidade de poder computacional da comunidade científica de HPC, logo, viu-se a necessidade de construção de um *benchmark* específico para avaliá-lo.

Dentre as mais recentes pesquisas voltadas ao desenvolvimento de *benchmarks* para a área de HPC temos o Mirovia, um *benchmark* desenvolvido por *Hu et al.* (HU; ROSSBACH, 2019) para tirar proveito de arquiteturas de GPUs modernas, medindo de forma precisa os atuais sistemas heterogêneos, com processadores *multicore*, *manycore* e GPUs. O Mirovia foi desenvolvido na necessidade de um *benchmark* que levasse em conta as novas tecnologias que esses sistemas possuem, como a memória unificada, já que os *benchmarks* clássicos, como o Rodinia (LEE et al., 2009) ou o SHOC (DANALIS et al., 2010), foram desenvolvidos antes do surgimento destas. Além disso, devido ao crescente interesse em redes neurais e aprendizagem profunda, alguns *kernels* com foco nesta área foram adicionados. Assim, o Mirovia tem como base *kernels* de *benchmarks* clássicos e adiciona *kernels* de aplicações de maior interesse atual para melhor caracterizar os sistemas heterogêneos modernos.

Por fim, *Tian et al.* (TIAN et al., 2017) desenvolveram o BigDataBench-S, um *benchmark* que avalia a performance de sistemas de gerenciamento de *Big Data*. Os autores ressaltam que o número de dados gerados por aplicações científicas é cada vez maior e que os atuais *benchmarks* desse contexto foram feitos para sistemas que focam ou na Internet ou em alguma área científica específica. Assim, o BigDataBench-S surge como um *benchmark* representativo para avaliar os sistemas de gerenciamento e análise de dados gerados por aplicações, equipamentos e dispositivos de propósito científico.

Nenhum dos *benchmarks* descritos anteriormente foi desenvolvido com foco em *manycores* de baixa potência, como o MPPA-256. Nesse sentido, o CAP Bench (SOUZA et al., 2016) é o único *benchmark* conhecido para esse processador, permitindo exercitar diferentes funcionalidades dele. Até então, o CAP Bench era implementado com a API IPC do MPPA-256 (DINECHIN et al., 2013). No próximo capítulo será discutido como o CAP Bench foi modificado para funcionar com a nova API ASYNC do MPPA-256. Esta nova versão do CAP Bench é a principal contribuição do presente trabalho.

4 DESENVOLVIMENTO

Para a realização deste trabalho, foram feitos dois tipos de alteração em todas as aplicações do CAP Bench. Primeiro, criou-se uma nova versão de cada aplicação, na qual foram realizadas modificações em sua lógica de processamento para que pudesse utilizar a API ASYNC. Após isso, alterou-se as versões antigas, que usam a API IPC, para que ficassem equivalente as novas aplicações em relação ao seu fluxo de processamento. Nesta seção serão mostradas todas as alterações feitas em cada aplicação e em cada etapa. Uma descrição mais detalhada de cada um dos *kernels* pode ser encontrada no trabalho de *Souza et al.* (SOUZA et al., 2016).

4.1 ALTERAÇÕES NA *FRIENDLY NUMBERS*

Segundo a teoria dos números, dois números naturais e inteiros são amigáveis se compartilham a mesma abundância. A abundância A de um número n é definida como $A(n) = \frac{\sigma(b)}{n}$, onde $\sigma(n)$ representa a soma de todos os divisores de n. A aplicação *Friendly Numbers (FN)* calcula e compara as abundâncias de todos os números em um certo intervalo, determinando quais pares de números são amigáveis. O padrão paralelo usado nessa aplicação é o *MapReduce*.

A primeira aplicação portada com a ASYNC foi a *FN*, tendo pequenas alterações devido a sua baixa complexidade. Em ambas as versões, existe um *array* de tamanho fixo que define as tarefas de cada *cluster* segundo intervalos de *offsets*, onde em cada posição existem três variáveis do tipo `int` agrupadas em uma `struct`. Basicamente, o *cluster* de E/S define um intervalo de números e seta cada número na variável `number` de cada `struct`, enquanto que os CCs devem calcular as variáveis `num` e `den`. O algoritmo 6 detalha como é feita a implementação de ambos *array* e `struct`.

Listagem 6 – Definição das taréfas por parte do *cluster* de E/S.

Fonte: o autor.

A versão ASYNC possibilitou a redução de grande parte do fluxo de operações desta aplicação. Anteriormente, era necessário explicitamente realizar operações de envio e recebimento de mensagens no *cluster* de E/S, através das macros `data_send()` e `data_receive()`, sendo essas operações totalmente bloqueantes. Com a introdução dos segmentos pela API ASYNC, duas linhas de código substituem ambas operações. Nesse caso, um segmento é criado através da função `mppa_async_segment_create()`, sobre o *array* de tarefas. Assim, qualquer CC que cloná-lo poderá acessar seu conteúdo e modificá-lo, sendo essas modificações totalmente acessíveis ao cluster de E/S através do mesmo *array* de tarefas.

Dessa maneira, os CCs só são lançados depois que todas as tarefas estão definidas no *array* e o segmento foi criado. Então, um *cluster* de computação só precisa fazer uma operação `GET` em um certo intervalo no segmento, processar os dados, e enviá-los de volta através de uma operação `PUT` no mesmo intervalo no segmento. Como cada CC possui um intervalo de *offsets* diferente, não há sobreposição de tarefas e não existe nenhuma condição de corrida sobre uma certa posição no segmento. Por fim, como as operações `PUT` dos CCs alteram o *array* no qual o segmento foi definido no *cluster* de E/S, não há necessidade do processo *master* realizar um `GET` sobre aquele segmento, bastando acessar o *array* de tarefas diretamente. Logicamente, se o processo *master* ler o *array* antes da hora, irá obter valores `null` para as variáveis `num` e `den`, logo, é feita uma única sincronização, onde os *clusters* de E/S aguardam a sinalização dos CCs de que determinada tarefa já foi computada e colocada em um certo intervalo do segmento, antes de ler aquele intervalo.

Mudanças sem relação com as APIs também foram implementadas em ambas as versões. No código dos *slaves*, existe uma função que computa a soma dos divisores de um número. Nesta função, era feito um *loop* iniciando em 2 e terminando no número, onde em cada iteração verifica-se se o número daquela iteração era divisor do número passado como argumento para a função. Com a otimização, alterou-se o *loop* para iterar até metade do número passado como argumento, já que, acima da metade, o único divisor de um número é ele mesmo. Outra otimização feita foi em relação ao paralelismo para verificar quais números tinham as variáveis `num` e `den` iguais. Na versão antiga utilizou-se *POSIX Threads* para fazer uma soma parcial e depois uma redução na *thread* principal, gastando mais de 50 linhas nessa lógica. Na nova versão, com a utilização de diretivas do OpenMP, reduziu-se essas 50 linhas em 5, com um fluxo de execução muito mais fácil de ser entendido.

4.2 ALTERAÇÕES NA *FEATURES FROM ACCELERATED SEGMENT TEST*

Features from Accelerated Segment Test (FAST) é um algoritmo de detecção de cantos que segue o padrão paralelo *Stencil*. Esse algoritmo é muito usado para extrair pontos importantes em uma imagem, assim como mapear objetos em aplicações de visão computacional. O algoritmo usa um círculo de 16 *pixels* para testar se um certo ponto p é um canto. Cada *pixel* no círculo é classificado através de um inteiro que varia de 1 a 16. Se todos os N *pixels* do círculo possuírem um brilho maior do que o brilho do candidato p somado a um limiar t ou possuírem um brilho menor do que o brilho do candidato p subtraído do mesmo limiar t, então p é classificado como um canto.

A versão antiga da *FAST* sofreu otimizações em partes que ocupavam a maior porcentagem do seu tempo de execução. Nesta versão, o *cluster* de E/S realizava o envio de tarefas aos CCs através de iterações em um `for`, enquanto que os CCs aguardavam as tarefas em um `while(true)`, recebendo, primeiramente, uma mensagem sinalizando

se naquela iteração era necessário computar algo ou se poderia encerrar sua execução. Porém, como todas as variáveis utilizadas no `for` do processo *master* poderiam ser passadas como argumento no momento de lançar um CC, foi implementado isso em ambas as novas versões, o que resultou na eliminação de toda a lógica de enviar uma mensagem ao *slave* para sinalizar se este deve ou não continuar sua execução. Assim, implementou-se um `for` com início e fim bem definidos no código do *slave*, eliminando o `while(true)` deste e reduzindo de quatro para uma o número de mensagens que os *slaves* recebem por iteração. Além disso, em ambas versões, reduziu-se de dois para um o número de mensagens enviadas dos *slaves* para o *master* por iteração, pois na nova versão, em vez de retornar a cada iteração ambos o número de cantos detectados e parte da nova imagem, só é retornado o segundo, somando o primeiro em cada iteração e o retornando ao fim de todas elas.

Na versão ASYNC foram criados 3 segmentos sobre os dados necessários para os *slaves*. Primeiro, um segmento sobre a máscara a ser aplicada na imagem que se quer detectar os cantos, o qual todos os CCs tem acesso completo. Em seguida, um sobre a imagem original e um sobre o *array* que irá conter o resultado do filtro aplicado a esta imagem, os quais os CCs só tem acesso a uma lista de intervalos de *offsets*. Assim, nesta versão a otimização é ainda melhor, pois com toda a abstração que os segmentos trazem, a troca de mensagens cai pela metade, já que somente os *slaves* precisam realizar operações `PUT` e `GET`, bastando ao *master* criar os segmentos sobre os dados. Além disso, com as otimizações do parágrafo anterior, os *slaves* agora sabem exatamente o número de operações que devem executar e, como conhecem os intervalos de *offsets* que podem acessar dentro dos segmentos da imagem original e imagem filtrada, são totalmente autônomos para pegar e responder as tarefas nestes segmentos de forma totalmente assíncrona e paralela. Essa autonomia também se dá pois estes intervalos são diferentes para todos os *slaves*, não havendo a repetição de nenhum intervalo em dois CCs distintos, o que elimina a possibilidade de condições de corrida e permite um paralelismo com eficiência máxima.

4.3 ALTERAÇÕES NA *GAUSSIAN FILTER*

O algoritmo *Gaussian blur filter* aplica um filtro de suavização em determinada imagem, buscando reduzir seu ruído e alcançar uma imagem mais suave. Para isso, uma máscara bidimensional previamente computada é aplicada sobre uma imagem através de uma operação com uma matriz de convolução. O padrão paralelo desta aplicação é o *Stencil*.

Poucas foram as alterações da *Gaussian Filter (GF)*, sendo esta a aplicação com menos modificações, dentre todas as outras. Quanto a versão IPC, nada foi modificado. Já a versão ASYNC seguiu a mesma lógica da versão IPC, substituindo as chamadas das funções síncronas de envio e recebimento de mensagens para chamadas as funções assíncronas de `PUT` e `GET`. Para realizar esta substituição, foram criados dois segmentos

de dados, sendo um sobre o *array* que guarda a máscara a ser aplicada na imagem e outro sobre o *array* que guarda o pedaço da imagem a ser trabalhado em determinada iteração, chamado de *chunk*. Ambos podem ser completamente acessados pelos *slaves*, porém, no segundo só é feito um acesso por vez. Assim, o método de comunicação de dados sobre este segmento é separado em duas etapas, as quais, dependendo do tamanho da imagem a ser aplicada o filtro, podem ou não acontecer diversas vezes durante a execução da aplicação.

Na primeira etapa, o *cluster* de E/S realiza somente escritas sobre o segmento, onde cada escrita é seguida de uma leitura por algum CC, o qual armazena o que foi lido em sua memória local e sinaliza ao processo *master* para prosseguir com a inserção da próxima tarefa para o próximo CC. Já na segunda etapa, após terem feito toda computação sobre o *chunk* que receberam, os CCs aguardam o processo *master* sinalizar que está na hora de escrever o dado calculado sobre o mesmo segmento em que foi recebido o *chunk*, e, quando sinalizados, escrevem-no. Enquanto isso, o *cluster* de E/S aguarda cada CC sinalizar que tal dado já foi escrito no segmento. Apesar de complexo, este método é ligeiramente melhor que o da versão IPC, visto que as operações `PUT` e `GET` de envio e recebimento de mensagens só são realizadas no lado dos *clusters* de computação, já que no processo *master* só é necessário copiar o dado do segmento para algum lugar dentro do *array* que irá conter a nova imagem. Assim, similar às demais aplicações, a troca de mensagens é reduzida pela metade.

4.4 ALTERAÇÕES NA *LU FACTORIZATION*

A aplicação com maior número de modificações em sua lógica foi a *LU Factorization (LU)*, principalmente por causa da implementação errônea do algoritmo na primeira versão. Basicamente, o algoritmo de decomposição LU transforma uma matriz A em duas matrizes, uma triangular inferior (L) e uma triangular superior (U), de modo que $A = L * U$. Porém, o erro da primeira versão foi implementar permutações de trocas de linha e coluna, as quais caracterizam outra decomposição, chamada de PLU e definida pela equação $P * A = L * U$, onde P é o agregado final de todas as matrizes de permutação aplicadas sobre A. Assim, funções de percorriam extensivamente a matriz em busca de um novo pivô e funções que trocavam linhas e colunas foram removidas. Além disso, blocos dentro da matriz eram enviados de forma errada aos *slaves*, gerando um resultado final totalmente errado. A Figura 11 mostra como um bloco de 9 elementos (3x3) deveria ser passado, segundo a matriz B, e como era passado primeira versão, segundo a matriz A (os blocos estão pintados em vermelho). Vale salientar que esse erro só foi possível pois a variável que guarda a matriz na verdade é um *array* unidimensional.

Todos os erros citados no parágrafo anterior foram solucionados em ambas as versões. Quanto a versão ASYNC, para tirar proveito desta API foi criado um segmento sobre a matriz original e um segmento sobre um *array* de tarefas. Deste modo, em uma

Figura 11 – Exemplo de um bloco passado a um *slave* em cada versão da LU.

$$A = \begin{bmatrix} 1 & 5 & 3 & 8 \\ 12 & 7 & 15 & 12 \\ 3 & 12 & 92 & 8 \\ 6 & 9 & 3 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 5 & 3 & 8 \\ 12 & 7 & 15 & 12 \\ 3 & 12 & 92 & 8 \\ 6 & 9 & 3 & 1 \end{bmatrix}$$

certa iteração, um processo *slave* consulta o segmento de tarefas usando seu próprio *id* como *offset*, recuperando um tarefa que contém, entre outras variáveis, os *offsets* que serão aplicados para pegar um bloco no segmento da matriz. Quando um CC não possui mais tarefas, o *cluster* de E/S coloca uma mensagem de finalização no *offset* deste *slave* dentro do segmento de tarefas, o qual a recebe e finaliza sua execução. Devido aos dois segmentos citados acima, as operações `PUT` e `GET` são feitas somente pelos CCs, enquanto que o processo *master* só realiza sinalizações, quando coloca uma tarefa no segmento, e sincronizações, quando todos os *slaves* finalizam uma iteração de computação.

4.5 ALTERAÇÕES NA *K-MEANS*

K-Means (KM) é um algoritmo de clusterização de dados muito usado na área de análise de dados. Dado um conjunto de n pontos em um espaço de dimensão d, o problema passa por particionar estes n pontos em k conjuntos. No CAP Bench, os pontos são inicializados com coordenadas aleatorias e divididos aleatoriamente entre as k partições. Assim, os centróides iniciais são computados (um centróide representa a média de todos os pontos de uma partição k), os pontos são novamente divididos entre as partições levando em conta a distância mínima euclidiana entre cada ponto e o centróide da sua partição e, por fim, os centróides são recalculados. Todo esse procedimento é repetido até que nenhum centróide mude de posição. Durante a execução, o número de pontos em cada partição pode variar, implicando em diferentes tempos de computação para cada centróide de cada partição. O padrão paralelo da KM é o *Map*.

Diversas simplificações na lógica do *kernel* KM foram feitas. Primeiramente, devido a grande quantidade de dados enviados aos CCs e a diferença nos tipos destes dados, resolveu-se adotar uma nova estratégia, desta vez utilizando o segmento padrão `MPPA_ASYNC_DDR_0`, que é sempre criado na inicialização da ASYNC, para o envio destes. Desta forma, através da função `mppa_async_malloc()`, cada *slave* aloca uma porção de memória neste segmento para cada uma das variáveis que vão utilizar em suas iterações e comunicações. Esta função então retorna um *offset* apontando para onde aquele espaço foi alocado dentro do segmento, e todos os *offsets* são enviados para o processo *master*, através de outro segmento especialmente criado para isso. Ao término desta primeira etapa, o processo *master* saberá onde ler e escrever todos os dados no

segmento padrão e, em etapas subsequentes, todas as comunicações dos dados computados serão feitas neste segmento, utilizando estes *offsets*.

Pensando na otimização do envio e recebimento dos dados em cada iteração, o modelo de dados da aplicação também foi simplificado. Na versão original, 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, inicializando todos os pontos em unico *array* unidimensional. Desse modo, em um cenário com P pontos, todos com dimensão X, cada ponto ocupa X posições do *array* e o tamanho total deste é de $X * P$ posições. Logo, fica muito mais fácil separar os pontos em intervalos e enviar cada intervalo como tarefa para um certo *cluster* de computação.

Por fim, a lógica de cálculo dos centróides (média dos elementos de um certo grupo) também foi alterada. Na versão original, toda a computação do cálculo da distância euclidiana média era feita nos CCs. Isto requeria que os CCs enviassem a informação sobre suas populações parciais (número de elementos de um grupo) para o processo *master*, que na sequência os enviava de volta a soma destes dados. Já na nova versão, nos CCs é realizado somente a primeira etapa deste cálculo (a soma dos vetores parciais). Para calcular a média destes vetores, a população total é necessária, a qual está facilmente disponível no processo *master* após determinada iteração. Assim, o cálculo da média é feito no *cluster* de E/S, o que resulta em menos comunicações feitas entre os processos, gerando uma redução de 2 operações de escrita e 2 operações de leitura por iteração.

4.6 ALTERAÇÕES NA *INTEGER-SORT*

Integer-Sort (IS) é um algoritmo que soluciona o problema de ordenar um grande número de inteiros. A variação do algoritmo implementado no CAP Bench foi o *bucket-sort*, o qual divide os elementos a serem ordenador em baldes. Um balde é uma estrutura que armazena uma certa quantidade de inteiros. A aplicação é inicializada com entradas aleatórias para o conjunto dos inteiros, os quais são gerados em um intervalo entre o até $2^{20} - 1$. Por fim, o IS usa o padrão paralelo Dividir para Conquistar.

A aplicação *IS* sofreu poucas alterações quanto a sua lógica, otimizando somente a etapa de recebimento dos *buckets* nos *slaves*. Na versão antiga, para qualquer variação da classe de tamanho do problema, a ordenação parcial dos *buckets* era feita sempre com o número máximo de elementos que um *bucket* armazena na maior classe de problema. Como existem 16 núcleos em um determinado *cluster*, se o número de elementos de um *bucket* é divisível por 16 a paralelização do trabalho é simplificada, e, como o número máximo de elementos da maior classe abrange todas as classes inferiores e é também divisível por 16, resolveu-se utilizar este. Porém, isso tomava muito tempo da ordenação, já que era preciso iterar sobre uma grande porção de elementos *dummy* no *array* de

elementos daquele *bucket*. Na nova versão a ordenação é feita com o número de elementos de um *bucket* somados a, no máximo, 16 novos elementos de valor igual ao máximo inteiro possível. Assim, o número total de elementos sempre será divisível por 16 e as iterações sobre elementos *dummy* não afetam o tempo total de execução da aplicação.

Com a adição da API ASYNC, criou-se um segmento sobre o *array* que armazena os *buckets* a serem enviados para os CCs em determinada iteração. Para enviar um *bucket* a um certo CC, os elementos deste são colocados no segmento descrito acima, num intervalo de *offsets* reservado aquele CC, enviando um sinal para este *slave* de que o dado está pronto para ser computado. Após realizada a ordenação parcial de um determinado *bucket*, os CCs colocam-o de volta no mesmo intervalo reservado dentro do segmento dos *buckets*, enviando um sinal ao processo *master* de que o trabalho foi feito e o dado está pronto.

5 RESULTADOS

Este capítulo apresenta os resultados obtidos nas execuções de cada *kernel* em cada versão do CAP Bench, comparando-os através de métricas definidas para as duas versões. Dentre as variáveis medidas, são comparadas: **(i)** o tempo de execução de cada CC em segundos; **(ii)** o tempo de execução do processo *master* em segundos, excluindo o tempo de envio e recebimento de dados e o tempo em que este se encontra bloqueado; **(iii)** o tempo de comunicação entre os processos *master* e *slave*, também em segundos, incluindo o tempo de sincronização de envio e recebimento de dados, no caso da IPC, ou da sincronização por meio de barreiras, no caso da ASYNC; **(iv)** a quantidade de dados que o processo *master* envia para os *slaves*, em *megabytes*; **(v)** a quantidade de dados que o processo *master* recebe dos *slaves*, também em *megabytes*; **(vi)** a potência média durante a execução da aplicação, em *watts*; e **(vii)** o consumo de energia durante a execução da aplicação, em *joules*.

As aplicações foram executadas com as classes de tamanho *tiny*, *small*, *standard*, *large* e *huge*. Para cada classe, variou-se o número de *clusters* de computação entre 1, 2, 4, 8 e 16. Além disso, 5 repetições foram realizadas para cada uma dessas variações. Assim, os resultados de cada métrica são sempre a média dessas execuções. Como o MPPA-256 possui características intrínsecas que garantem baixa variabilidade entre as execuções, somente 5 repetições foram suficientes para garantir um desvio padrão relativo menor que 1% nas métricas de tempo de execução e energia. Todos os experimentos consideram 16 núcleos de processamento ativos por *cluster* utilizado. As medições de energia foram coletadas através de sensores de energia e potência espalhados nos diferentes componentes de *hardware* do processador, considerando todos os *clusters*, os subsistemas de E/S, a NoC e a memória. Por fim, ambas versões foram executadas com a *flag* de otimização `-O3` do GCC.

Cada gráfico mostrado neste capítulo compara determinada métrica de uma aplicação que usa a API ASYNC *versus* esta mesma aplicação usando a API IPC. Essa comparação é então estendida para cada variação do número de *clusters* e para cada variação da classe de tamanho do problema. Para a aplicação KM, a classe *large* não foi executada com 1 *cluster* e a classe *huge* não foi executada com 1 e 2 *clusters*, pois segundo a lógica da aplicação, não é possível distribuir o trabalho para os *clusters* nesse contexto, já que a memória destes não suporta o tamanho da tarefa que seria enviada em uma certa iteração. Devido ao grande intervalo entre os valores mínimo e máximo de quase todos os gráficos, adotou-se nestes uma escala em potência de 2 para melhor visualização dos resultados. Os gráficos que não possuem este problema utilizam uma escala linear.

Por último, as tabelas deste capítulo exibem a redução ou aumento de determinada métrica através da fórmula $((ResultadoASYNC/ResultadoIPC) - 1) * 100$, mostrando a porcentagem de redução dessa métrica na versão ASYNC, caso positiva, ou de aumento, caso negativa. Desta maneira, as tabelas focam no melhor e pior resultado de

redução, informando em qual variação ocorreu esse resultado, ou seja, em qual classe de problema e com quantos *clusters*, disponibilizando também os resultados da medida na versão IPC e ASYNC que geraram essa porcentagem.

5.1 RESULTADO DAS MÉTRICAS DE TEMPO

As métricas de tempo mostram um resultado positivo quanto a otimização do CAP Bench utilizando a API ASYNC. Para os tempos de execução dos *slaves*, houve melhoria em todos os *kernels*, com a porcentagem de redução variando entre 57.14 % até 94.44 %, conforme mostrado na Tabela 1. Além disso, essa tabela mostra uma grande consistência na porcentagem de redução de todas as variações de execução, já que a diferença entre a redução mínima e máxima é pequena para todos os *kernels*, ou seja, independentemente da configuração da aplicação (número de *clusters* e tamanho do problema), a diferença entre o ganho mínimo e máximo é muito pequena. A Figura 13 mostra os resultados dos tempos de execução dos *slaves*.

Tabela 1 – Reduções ao comparar-se os tempos dos processos *slaves*.

App	Redução Mínima					Reduçã Máxima				
	Classe	NClusters	IPC(s)	ASYNC(s)	Redução(%)	Classe	NClusters	IPC(s)	ASYNC(s)	Redução(%)
FAST	Tiny	8	0.72	0.05	93.06	Tiny	16	0.36	0.02	94.44
FN	Huge	1	34412.07	1952.39	94.33	Small	16	267.9	15.14	94.35
GF	Tiny	4	1.38	0.09	93.48	Tiny	8	0.69	0.04	94.2
IS	Tiny	8	1.82	0.78	57.14	Huge	1	282.06	115.03	59.22
KM	Tiny	16	2.86	0.19	93.36	Standard	1	1427.49	91.55	93.59
LU	Tiny	2	0.28	0.09	67.86	Huge	2	40.15	9.02	77.53

Já para os tempos de execução do processo *master*, houve melhoria em todos os *kernels* com exceção do LU, conforme mostrado na Tabela 2. A porcentagem de redução variou entre -28.79 % até 70.2 %. A variação no número de *clusters* só afeta significativamente essa variável na aplicação KM, visto que este é o único *kernel* em que o peso de certas computações no processo *master* é consideravelmente influenciado pelo número de *clusters*. O motivo do aumento dessa métrica no LU é devido a utilização de menos *threads* na paralelização de uma rotina feita no *cluster* de E/S. Enquanto que na IPC é possível executar até 4 *threads* no processo *master*, com a ASYNC só podem ser executadas 3 *threads*, visto que já existe uma *thread* sendo usada para controle das comunicações da API. Os *kernels* FAST e GF não são mostrados nem na tabela nem no gráfico dessa variável, já que ambos não executam computações no processo *master*, tornando o resultado dessa variável igual a 0 para estes.

Por fim, os tempos de comunicação também melhoraram em todos os *kernels* com exceção do IS, o qual obteve aumento nessa variável em todas as classes de problemas ao executar com 8 e 16 *clusters*. A Tabela 3 mostra que a porcentagem de redução dessa métrica variou entre -213.16 % até 99.66 %, porém, a Figura 14 mostra que houve melhorias nesse *kernel* para todas as outras variações de *clusters*.

Tabela 2 – Reduções ao comparar-se os tempos do processo *master*.

App	Redução Mínima					Reduçã Máxima				
	Classe	NClusters	IPC(s)	ASYNC(s)	Redução(%)	Classe	NClusters	IPC(s)	ASYNC(s)	Redução(%)
FN	Tiny	1	0.14	0.08	42.86	Huge	1	118.54	35.32	70.2
IS	Standard	1	12.27	10.96	10.68	Huge	8	56.72	46.5	18.02
KM	Tiny	1	0.01	0.01	0.0	Tiny	4	0.02	0.01	50.0
LU	Huge	2	0.66	0.85	-28.79	Tiny	2	0.02	0.02	0.0

Tabela 3 – Reduções ao comparar-se os tempos de comunicação.

App	Redução Mínima					Reduçã Máxima				
	Classe	NClusters	IPC(s)	ASYNC(s)	Redução(%)	Classe	NClusters	IPC(s)	ASYNC(s)	Redução(%)
FAST	Tiny	16	1.44	1.16	19.44	Huge	8	109.55	1.25	98.86
FN	Huge	1	34412.4	976.35	97.16	Huge	16	2159.8	7.34	99.66
GF	Tiny	16	1.31	1.1	16.03	Huge	4	1162.81	20.57	98.23
IS	Tiny	16	1.14	3.57	-213.16	Huge	1	286.46	80.34	71.95
KM	Tiny	16	3.82	1.11	70.94	Huge	16	670.86	4.28	99.36
LU	Tiny	16	7.22	1.36	81.16	Large	8	193.78	6.2	96.8

Figura 12 – Tempos de execução do processo *master* para cada aplicação.

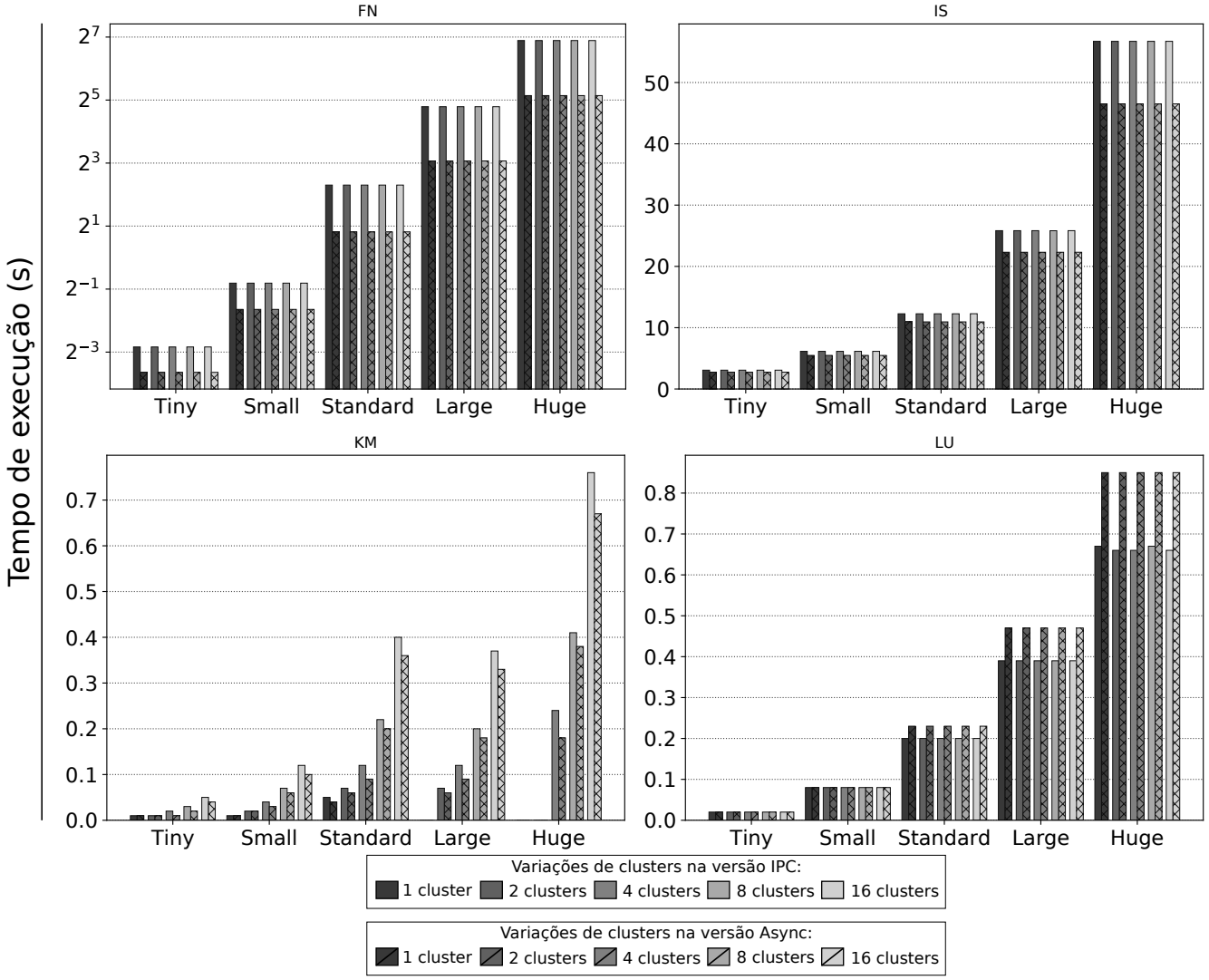


Figura 13 – Tempos de execução dos processos *slaves* para cada aplicação.

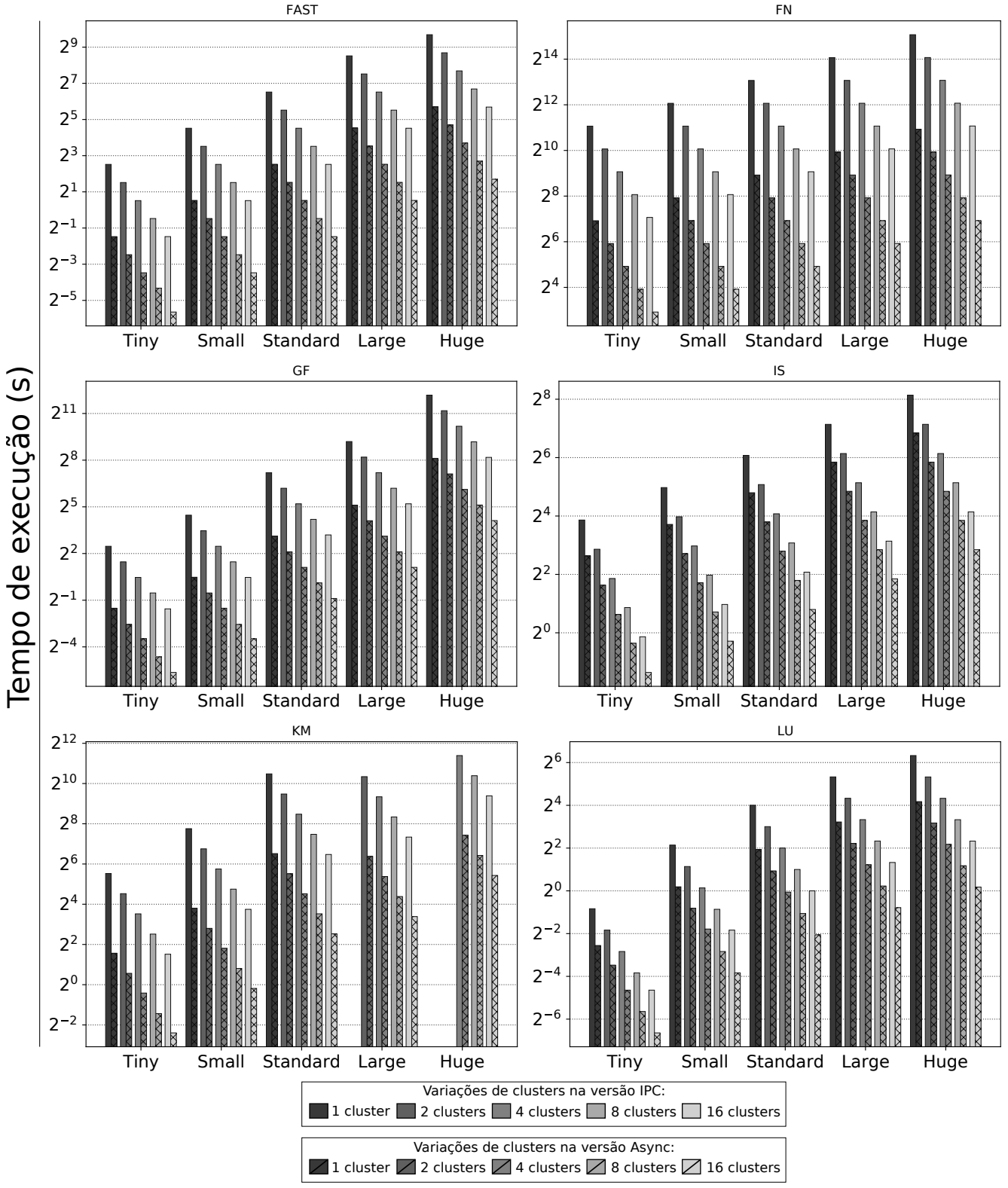
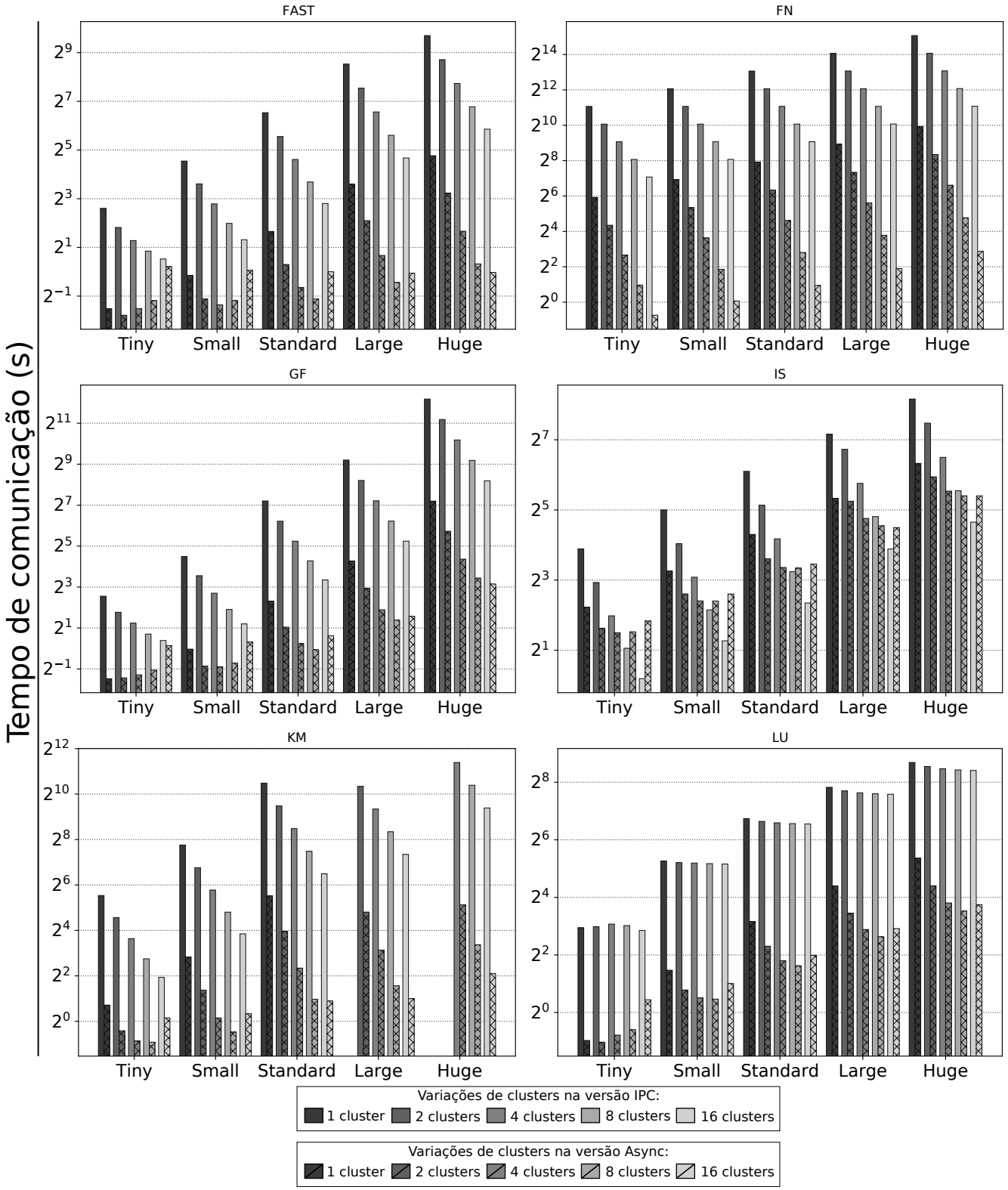


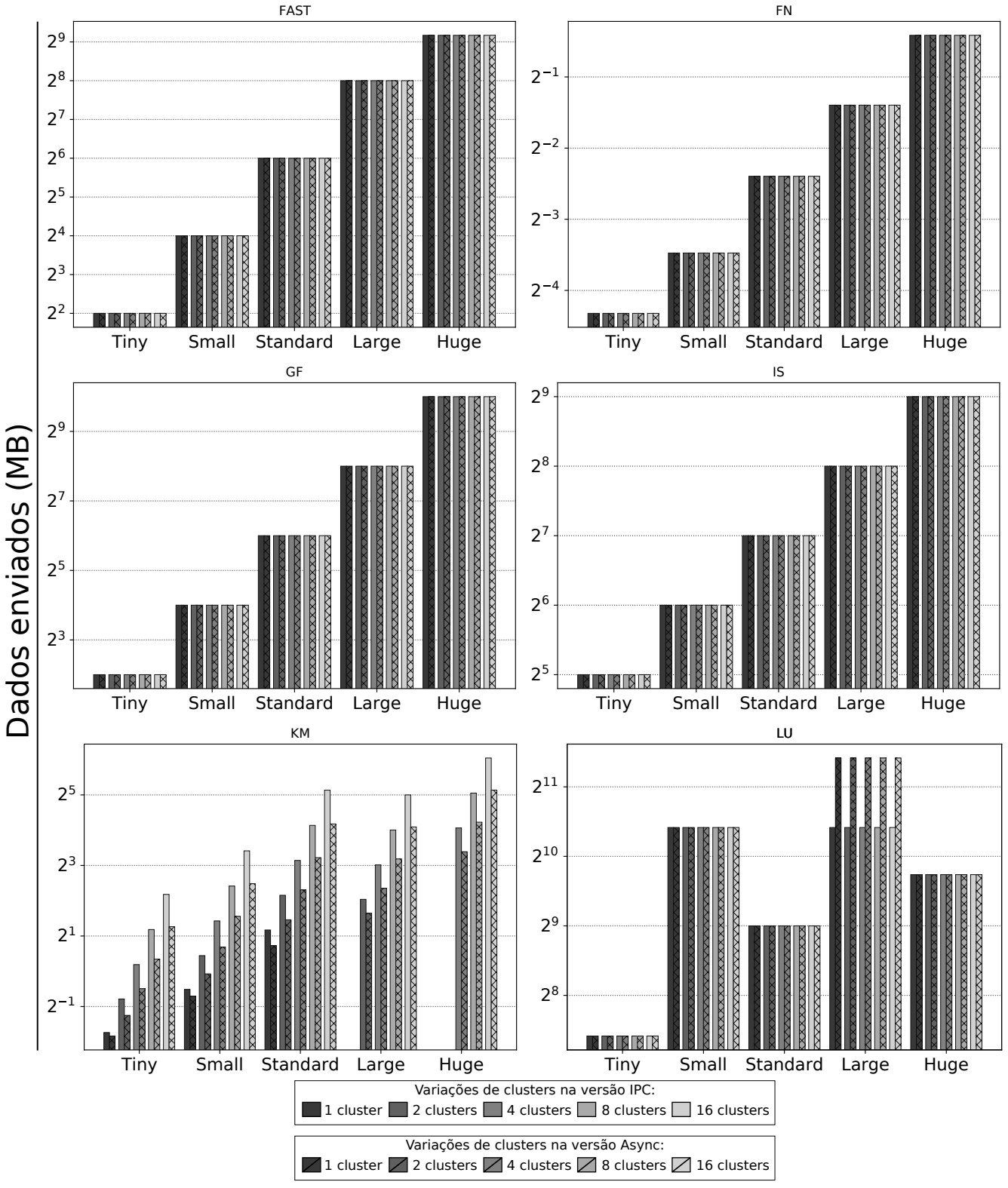
Figura 14 – Tempos de comunicação para cada aplicação.



5.2 RESULTADO DAS MÉTRICAS DE ENVIO E RECEBIMENTO DE DADOS

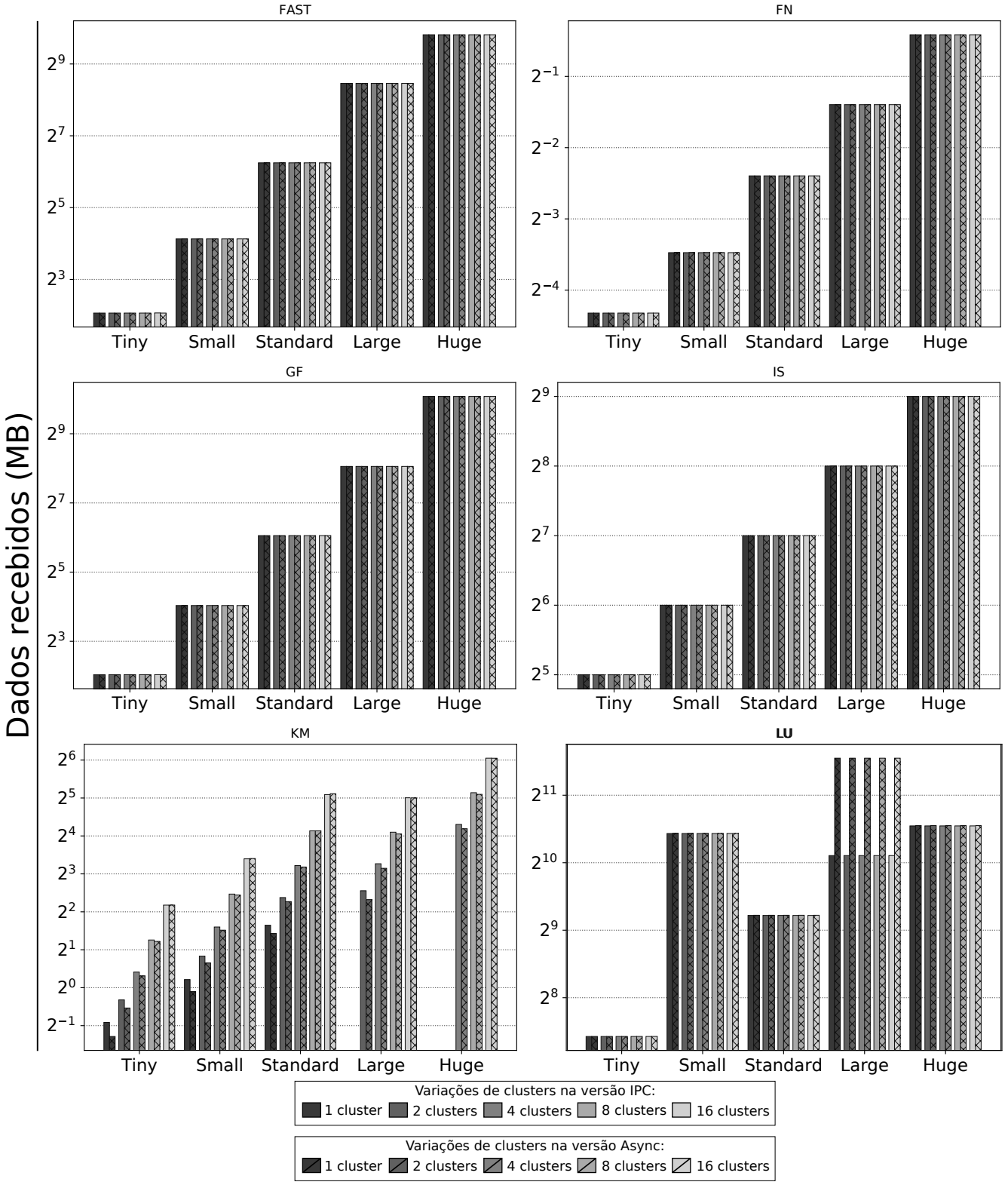
Um fator importante ao comparar-se a similaridade das aplicações em cada versão é a quantidade de dados trocados entre o *cluster* de E/S e os CCs. Nessa análise, os resultados das métricas de envio e recebimento de dados são mostrados somente via gráficos, na Figura 15 e 16, visto que estes já são suficientes para uma prova simplificada de que o peso de execução das aplicações é o mesmo para ambas as versões. Além disso, a tabela de redução mínima e máxima não faz sentido para a maioria das aplicações, já que nestas em todas as variações o resultado foi o mesmo, não havendo máximo ou mínimo.

Figura 15 – Quantidade de dados que o processo *master* envia aos *slaves*.



Duas aplicações são exceções para o que foi afirmado no parágrafo anterior. A KM obteve resultados com uma redução entre 6.67 % e 48.51 % para os dados enviados e entre -1.56 % e 19.83 % para os dados recebidos. Já a LU obteve resultados iguais para todas as classes de problema exceto a *large*, a qual teve uma redução de -100 % nos dados enviados e -171.88 % nos dados recebidos. Como as aplicações foram implementadas de modo semelhante em ambas versões, esses resultados são uma grande indicação de que, na versão ASYNC dessas duas aplicações, pode existir *bugs* em relação a medição dessa variável, mesmo não tendo sido encontrados após reanálise. Assim, tomam-se esses resultados como verdadeiros. Em conclusão, vale ressaltar a importância dessa métrica na busca por *bugs* em futuras aplicações no MPPA-256.

Figura 16 – Quantidade de dados que o processo *master* recebe dos *slaves*.



5.3 RESULTADO DAS MÉTRICAS DE ENERGIA

As métricas de energia confirmam o que foi observado na Seção 5.1 deste capítulo. Realmente a API ASYNC é mais custosa conforme o aumento no número de *clusters*, contudo, a IPC também apresenta esse comportamento. A Tabela 4 mostra que a potência média de execução das aplicações é maior na grande maioria das variações, o que é confirmado pela Figura 17. A redução dessa variável foi de -132.57 % até 3.75 %. De fato, a ASYNC é mais custosa do que a IPC nessa métrica pois aloca mais recursos do processador para gerenciar toda a abstração que proporciona.

Tabela 4 – Reduções ao comparar-se a potência média durante execução.

App	Redução Mínima					Reduçã Máxima				
	Classe	NClusters	IPC(W)	ASYNC(W)	Redução(%)	Classe	NClusters	IPC(W)	ASYNC(W)	Redução(%)
FAST	Small	16	4.52	5.14	-13.72	Small	1	4.27	4.11	3.75
FN	Large	16	4.79	10.25	-113.99	Huge	1	4.28	4.76	-11.21
GF	Huge	16	4.84	5.91	-22.11	Small	1	4.27	4.17	2.34
IS	Large	16	4.48	5.99	-33.71	Large	1	4.28	4.41	-3.04
KM	Huge	16	4.82	11.21	-132.57	Small	1	4.25	4.71	-10.82
LU	Huge	16	4.42	6.05	-36.88	Small	1	4.23	4.09	3.31

Mais importante que a potência média é o gasto energético total da aplicação. Apesar da biblioteca ASYNC ser mais custosa em questão de potência, como ela otimiza consideravelmente o tempo de execução das aplicações, o gasto energético acaba por ser menor na grande maioria delas. A Tabela 5 exibe as reduções dessa variável, as quais variaram entre -30.31 % e 93.64 % e a Figura 18 mostra os resultados de todas as variações.

Tabela 5 – Reduções ao comparar-se o gasto energético total.

App	Redução Mínima					Reduçã Máxima				
	Classe	NClusters	IPC(J)	ASYNC(J)	Redução(%)	Classe	NClusters	IPC(J)	ASYNC(J)	Redução(%)
FAST	Tiny	16	38.79	43.92	-13.23	Huge	1	3759.55	456.35	87.86
FN	Tiny	16	666.97	129.65	80.56	Standard	1	36549.92	2325.1	93.64
GF	Tiny	16	38.4	44.64	-16.25	Huge	1	20504.65	1783.83	91.3
IS	Tiny	16	52.2	68.02	-30.31	Large	1	751.9	395.02	47.46
KM	Tiny	16	47.55	46.21	2.82	Standard	1	6131.17	460.56	92.49
LU	Tiny	16	64.68	44.14	31.76	Huge	8	1535.77	103.8	93.24

É possível observar que o aumento dessa métrica na versão ASYNC acontece em contextos mais sensíveis. A Tabela 5 mostra que todas as reduções mínimas aconteceram com a classe *Tiny* e 16 *clusters* em execução, o que é esperado, pois é nessa variação em que, segundo a Figura 13, sempre ocorre o menor tempo de execução. Aliado a isso, é possível ver nos gráficos dessa variável que os *kernels* FAST e GF só obtiveram reduções negativas onde, novamente na Figura 13, a escala de tempo é baixíssima. Nesse contexto, como a escala do gasto energético dessas duas aplicações, se comparada às demais, também é baixa, o custo de inicialização do ambiente ASYNC pesa bastante no resultado final.

Figura 17 – Potência média durante a execução de cada aplicação.

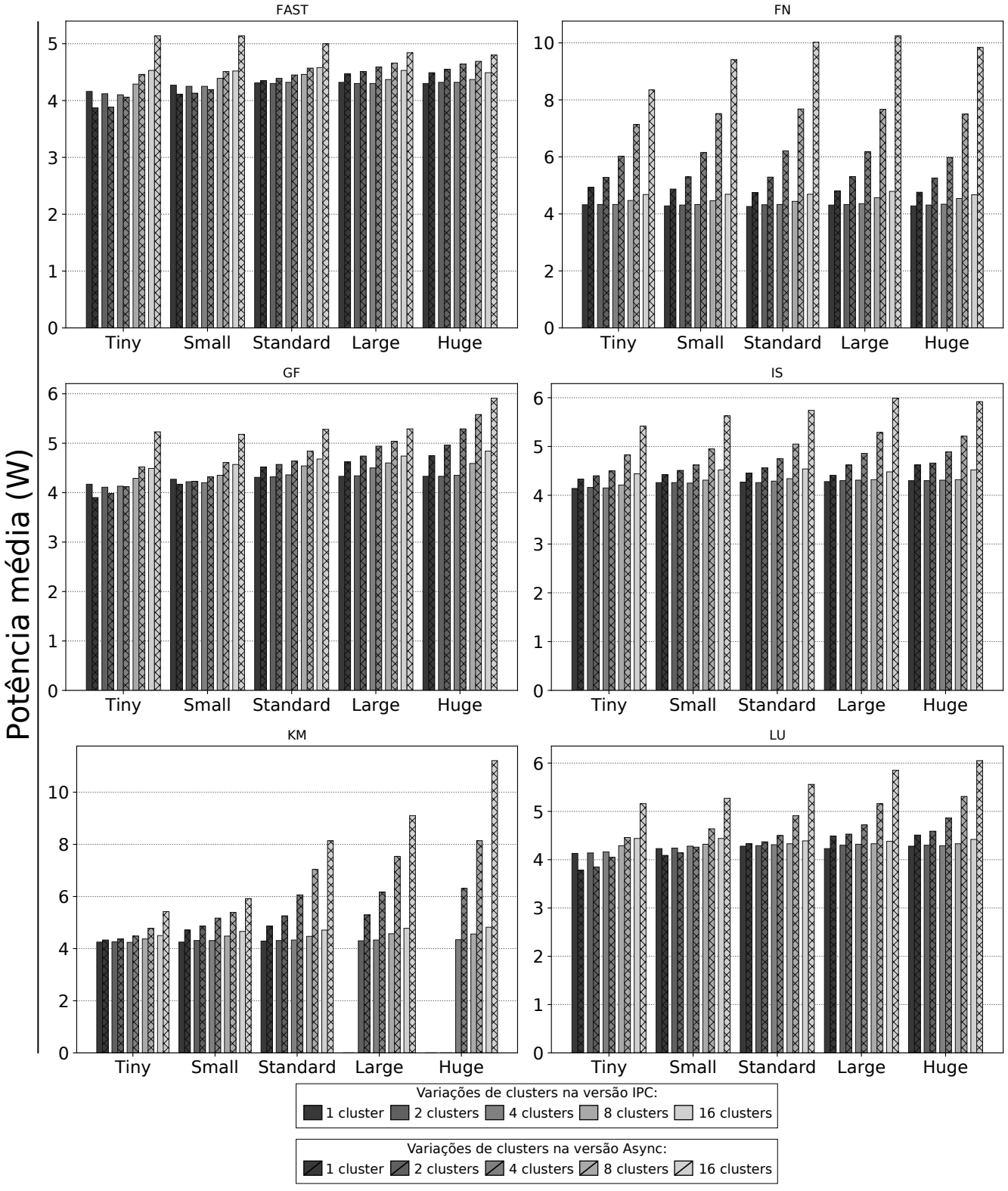
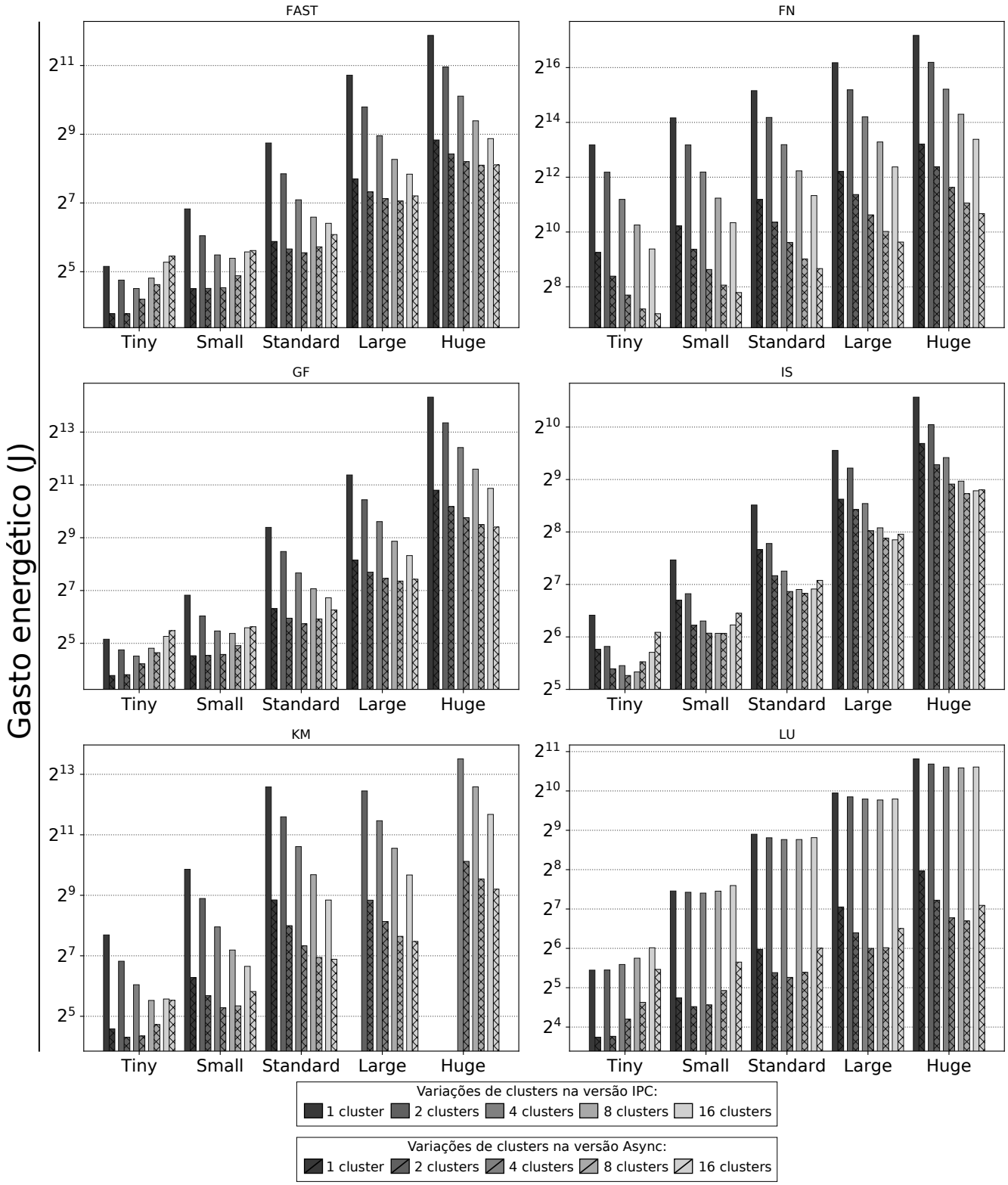


Figura 18 – Gasto energético de cada aplicação.



6 CONCLUSÃO

Neste trabalho, foi proposta uma nova versão do CAP Bench com o uso da API ASYNC, além de uma detalhada comparação dos novos resultados obtidos em relação à implementação original que utilizava a API IPC no processador MPPA-256. A solução proposta passa por medir diferentes variáveis durante a execução das aplicações do CAP Bench, a fim de compará-las, assim como comparar os resultados nos diferentes cenários que o CAP Bench abrange. Os resultados mostraram que a API ASYNC se sobressai em comparação com a IPC na maioria dos cenários, ao levar em conta todas as métricas consideradas. Também foi possível observar comportamentos característicos de cada biblioteca, por exemplo, o peso maior que a ASYNC exerce sobre a potência média.

De maneira geral, este trabalho mostra que diretivas assíncronas são, na maioria das vezes, a melhor escolha para projetos de execução paralela. Neste caminho e, associado a um processador de arquitetura voltada ao *Green Computing*, os resultados mostram também que, caso feita de maneira correta, APIs de comunicação assíncrona em processadores de arquitetura similar podem reduzir ainda o gasto energético de determinada aplicação, bastando somente uma implementação inteligente desta.

Como trabalhos futuros, pretende-se comparar os resultados das versão ASYNC com outros processadores *manycore* e *multicore* do estado da arte, a fim de avaliar se o MPPA-256 consegue competir com estes. Além disso, pretende-se analisar detalhadamente as aplicações CAP Bench que não obtiveram o resultado esperado nas variáveis de dados enviados e recebidos, a fim de reexaminar a possibilidade de *bugs* e corrigí-los, para que nestes trabalhos possam ser apresentados resultados que batam com o que foi esperado. Por fim, pretende-se utilizar os resultados desse trabalho para uma publicação em alguma revista associada a área de computação paralela e distribuída.

REFERÊNCIAS

ALMEIDA, R. et al. Implementação paralelizada do método do gradiente biconjugado estabilizado para a simulação de escoamentos bifásicos em reservatórios de petróleo. **ForScience**, v. 7, p. e00606, 06 2019.

ASANOVIć, K. et al. **The Landscape of Parallel Computing Research: A View from Berkeley**. [S.l.], 2006. Disponível em: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

BAILEY, D. et al. The nas parallel benchmarks. **The International Journal of Supercomputing Applications**, v. 5, n. 3, p. 63–73, 1991. Disponível em: https://doi.org/10.1177/109434209100500306.

BENNETT, E. et al. Bsmbench: A flexible and scalable hpc benchmark from beyond the standard model physics. In: . [S.l.: s.n.], 2016. p. 834–839.

BUREDDY, D. et al. Omb-gpu: A micro-benchmark suite for evaluating mpi libraries on gpu clusters. In: TRÄFF, J. L.; BENKNER, S.; DONGARRA, J. J. (Ed.). **Recent Advances in the Message Passing Interface**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 110–120. ISBN 978-3-642-33518-1.

DANALIS, A. et al. The scalable heterogeneous computing (shoc) benchmark suite. In: . [S.l.: s.n.], 2010. p. 63–74.

DIJKSTRA, E. W. Go to statement considered harmful. **Communications of the ACM**, v. 11, n. 3, p. 147–148, 1968.

DINECHIN, B. et al. A distributed run-time environment for the kalray mppa®-256 integrated manycore processor. **Procedia Computer Science**, v. 18, p. 1654–1663, 12 2013.

FU, H. et al. The sunway taihulight supercomputer: system and applications. **Sciece China. Information Sciences**, v. 59, p. 072001:1–16, 07 2016.

HU, B.; ROSSBACH, C. J. Mirovia: A benchmarking suite for modern heterogeneous computing. **CoRR**, abs/1906.10347, 2019. Disponível em: http://arxiv.org/abs/1906.10347.

KELLY, R. et al. The nwsc benchmark suite using scientific throughput to measure supercomputer performance. 11 2011.

KOGGE, P. et al. Exascale computing study: Technology challenges in achieving exascale systems. **Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Techinal Representative**, v. 15, 01 2008.

KURP, P. Green computing. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 51, n. 10, p. 11–13, out. 2008. ISSN 0001-0782. Disponível em: https://doi.org/10.1145/1400181.1400186.

LARUS, J.; KOZYRAKIS, C. Is tm the answer for improving parallel pro-gramming? **Communications of the ACM**, ACM, v. 51, n. 7, p. 80–88, 2018. ISSN 00010782. Disponível em: https://doi.org/10.1145/1364782.1364800.

LAU, K.; WANG, Z. Software component models. **IEEE Transactions on Software Engineering**, IEEE, v. 33, n. 10, p. 709–724, oct 2007. ISSN 0098-5589.

LEE, S. et al. Rodinia: A benchmark suite for heterogeneous computing. In: **2013 IEEE International Symposium on Workload Characterization (IISWC)**. Los Alamitos, CA, USA: IEEE Computer Society, 2009. p. 44–54. Disponível em: https://doi.ieeecomputersociety.org/10.1109/IISWC.2009.5306797.

MANIAN, K. et al. Omb-um: Design, implementation, and evaluation of cuda unified memory aware mpi benchmarks. In: . [S.l.: s.n.], 2019. p. 82–92.

MCCALPIN, J. Memory bandwidth and machine balance in high performance computers. **IEEE Technical Committee on Computer Architecture Newsletter**, p. 19–25, 12 1995.

NABI, S. W.; VANDERBAUWHEDE, W. Mp-stream: A memory performance benchmark for design space exploration on heterogeneous hpc devices. In: . [S.l.: s.n.], 2018. p. 194–197.

NIEPLOCHA, J. et al. High performance remote memory access communication: The armci approach. **The International Journal of High Performance Computing Applications**, v. 20, n. 2, p. 233–253, 2006. Disponível em: https://doi.org/10.1177/1094342006064504.

OLOFSSON, A.; NORDSTRöM, T.; UL-ABDIN, Z. Kickstarting high-performance energy-efficient manycore architectures with epiphany. **Conference Record - Asilomar Conference on Signals, Systems and Computers**, v. 2015, 12 2014.

PENNA et al. 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: [s.n.], 2018.

PODESTá et al. Energy efficient stencil computations on the low-power manycore mppa-256 processor. In: **international European Conference on Parallel and Distributed Computing (Euro-Par)**. [S.l.: s.n.], 2018. p. 642–655.

SOUZA et al. CAP Bench: A Benchmark Suite for Performance and Energy Evaluation of Low-power Many-core Processors. **Concurrency and Computation: Practice and Experience**, v. 29, n. 4, p. e3892, 2016. ISSN 1532-0634.

TANENBAUM, A. S.; BOS, H. **Modern Operating Systems**. 4th. ed. USA: Prentice Hall Press, 2014. ISBN 013359162X.

TIAN, X. et al. Bigdatabench-s: An open-source scientific big data benchmark suite. In: . [S.l.: s.n.], 2017. p. 1068–1077.

APÊNDICE A – CÓDIGO FONTE

[language=bash, label=l1,caption=run-mppa.sh]./appendix/run-mppa.sh