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

David Grunheidt Vilela Ordine

Comparação das tecnologias de comunicação entre clusters no processador MPPA-256 atráves do CAP Benchmarks

Florianópolis
24 de junho de 2020

David Grunheidt Vilela Ordine

Comparação das tecnologias de comunicação entre clusters no processador MPPA-256 atráves 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.
Coorientador: Pedro Henrique Penna, Dr.

Florianópolis
24 de junho de 2020

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Ravenhurst, Omar
Template LaTeX seguindo a RN 46/2019/CPG da UFSC / Omar Ravenhurst ; orientador, Ben Trovato, coorientador, Lars Thørväld, 2019.
666 p.

Tese (doutorado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, 2019.

Inclui referências.

1. Ciência da Computação. 2. Documentação. 3. LaTeX. I. Trovato, Ben. II. Thørväld, Lars. III. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. IV. Título.

David Grunheidt Vilela Ordine

Comparação das tecnologias de comunicação entre clusters no processador MPPA-256 atráves do CAP Benchmarks

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo curso de Graduação em Ciências da Computação.

Florianópolis, 24 de junho de 2020.

Prof. José Francisco Danilo de Guadalupe Correa Fletes, Me.
Coordenador do Curso

Banca Examinadora:

Prof. Márcio Bastos Castro, Dr.
Orientador
Universidade Federal de Santa Catarina

Pedro Henrique Penna, Dr.
Coorientador
Université Grenoble Alpes

Prof. Prof1, Dr.
Avaliador
Universidade Federal de Santa Catarina

Prof. Prof2, Dr.
Avaliador
Universidade Federal de Santa Catarina

Prof. Prof3, Dr.
Avaliador
Universidade Federal de Santa Catarina

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 meu co-orientador, 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 beneficia-se 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 projeto 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 até o momento mostram 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.

RESUMO ESTENDIDO

Introdução

Similar ao que aconteceu com os sistemas que utilizavam processadores *single core*, os supercomputadores da atualidade estão se deparando com uma barreira que impede o avanço em direção ao *Exascale*. A principal causa deste impedimento tem relação com as características intrínsecas da arquitetura de processadores *multicore* que compõem muitos destes supercomputadores. Assim como processadores *singlecore* não puderam mais aumentar a frequência de clock de um núcleo após um certo limite, só é possível diminuir o tamanho dos transistores que formam um núcleo de processamento até um certo ponto, e da mesma maneira, só é possível alocar uma certa quantidade de núcleos em um *chip*, antes que o gerenciamento desses núcleos fique algo inviável de ser implementado ou o tamanho do *chip* fique muito grande. Além disso, ao longo do tempo a relação entre dissipação de calor e ganho em performance mostrou-se não escalável para esta arquitetura. Assim, a comunidade científica de High-Performance Computing (HPC) começou a pesquisar e desenvolver novas arquiteturas que apresentassem melhor escalabilidade entre as variáveis citadas acima, surgindo então a classe de processadores *manycore* de baixo consumo energético, por exemplo, o MPPA-256, que será estudado neste trabalho.

Fundamentação Teórica

Atualmente, arquiteturas com múltiplos processadores são algo comum em diversos sistemas, porém, nem todos sabem como caracterizar estes sistemas de acordo com a disposição dos elementos que os compõem. Dentre essas arquiteturas, temos os sistemas multiprocessadores e os multicomputadores, sendo a principal diferença entre eles o compartilhamento ou não de memória por parte dos núcleos de processamento. Enquanto os sistemas de multiprocessadores conectam esses núcleos a uma memória através de um barramento, os multicomputadores conectam as unidades de processamento através de uma rede, onde cada unidade tem sua memória privada, e a comunicação entre as unidades é feita via troca de mensagens através de alguma API. Além disso, os multiprocessadores podem ser divididos em duas categorias, os UMA e NUMA, onde o que os difere é o tempo de acesso a uma palavra na memória ser o mesmo ou não, para qualquer palavra. Vale citar que o MPPA-256, processador usado neste trabalho, se enquadra na classe dos multiprocessadores. Para estes sistemas existem diversas bibliotecas e padrões voltados a programação paralela, sendo os mais comuns o MPI, padrão o qual bibliotecas de troca de mensagem entre processos se baseiam, e a OpenMP, API muito usada quando o assunto é *multithreading*. Ambos permitem abstrair qual plataforma o programa paralelo em questão irá executar, o que facilita a implementação deste programa, sendo esse um dos principais motivos pelo qual são amplamente adotados. Já para o MPPA-256, duas bibliotecas são estudadas neste trabalho. A biblioteca IPC, utilizada na primeira versão do *benchmark*, tem um baixo nível de abstração, sendo necessário conhecer vários aspectos da arquitetura do processador para que seja feita uma implementação otimizada e eficiente de alguma aplicação. Já a ASYNC tem um alto nível de abstração e permite a troca de mensagens de modo assíncrono, o que muitas vezes pode ser uma vantagem para determinadas aplicações.

Trabalhos Correlatos
Desenvolvimento
Resultados preliminares
Conclusão

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

ABSTRACT

Throughout the evolving process of *single-core* processors, the main method to gain performance was to increase the processor *clock* frequency, which led to the growing disproportion between energy consumption and increase in performance, making this method not viable anymore. This disproportion was then the barrier to the evolution of this class of processors. *Multi-core* processors solutions, for example, supercomputers, also face a similar barrier nowadays when grouping this processors into *clusters*, or grouping several *cores* into a single *chip*. Low consumption *manycore* processors, for instance, the MPPA-256 and the Adapteva Epiphany, are arising to solve this problem. However, due to architectural characteristics, such as a limited and distributed memory, implementing applications that fully benefits from the hardware of a processor of this class is not an easy task. Yet, when a good implementation is done, it can outstand state-of-the-art processors, through lower energy consumption. This project proposes to the CAP Bench, a *benchmark* developed to evaluate both MPPA-256 performance and energy consumption, optimizations to its applications and the implementation of a new version, using a new communication technology, based on asynchronous primitives, aiming to analyze the technologies used in each version. The results until now show that the applications that use the new technology have a better performance than the old ones. This is due, mainly, by the asynchronous characteristic of this library.

Keywords: Benchmark. Manycore. Performance. Green-Computing.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE LISTAGENS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface. 33, 34, 41, 43, 46, 47, 49, 55
ASYNC	MPPA Asynchronous Communication API. 33, 34, 41, 45, 46, 47
CC	Cluster de Computação. 40, 41, 45, 46, 47
CMP	Chip MultiProcessadores. 37
CPU	Central Processing Unit. 35, 36, 37, 38, 39, 40, 49, 50
DHT	Distributed Hash Table. 55
E/S	Entrada e Saída. 40, 41, 45, 46, 47
Flops	Floating-point Operations per Second. 31
FPGA	 49
GPU	Unidade de Processamento Gráfico. 38, 49, 50
HPC	High-Performance Computing. 13, 32, 34, 38, 49, 50
IPC	MPPA Interprocess Communication API. 33, 34, 41, 45
MPI	Message Passing Interface. 41, 43, 44, 49, 50
NoC	Network-on-Chip. 40, 45
NUMA	Nonuniform Memory Access. 35, 37
OpenMP	Open Multi-Processing. 41, 42, 43, 49
RAM	Random-Access Memory. 35
SIMD	Single Instruction Multiple Data. 38
SO	Sistema Operacional. 39
SPMD	Single Program, Multiple Data. 43
SQ	Square Matrix. 55
UM	Memória Unificada. 50
UMA	Uniform Memory Access. 35, 36, 37

W3C	World Wide Web Consortium	55

LISTA DE SÍMBOLOS

$\leftarrow$	Atribuição
$\exists$	Quantificação existencial
$\rightarrow$	Implicação
$\wedge$	E lógico
$\vee$	Ou lógico
$\neg$	Negação lógica
$\mapsto$	Mapeia para
$\sqsubseteq$	Subclasse (em ontologias)
$\subseteq$	Subconjunto: $\forall x \,.\, x \in A \rightarrow x \in B$
$\langle \ldots \rangle$	Tupla
$\forall$	Quantificação universal
mmmmm	Nenhum sentido, apenas estou aqui para demonstrar a largura máxima dessas colunas. Ao abrir o ambiente `listadesimbolos`, pode-se fornecer um argumento opcional indicando a largura da coluna da esquerda (o default é de 5em): `\begin{listadesimbolos}[2cm]` .... `\end{listadesimbolos}`

SUMÁRIO

1 INTRODUÇÃO

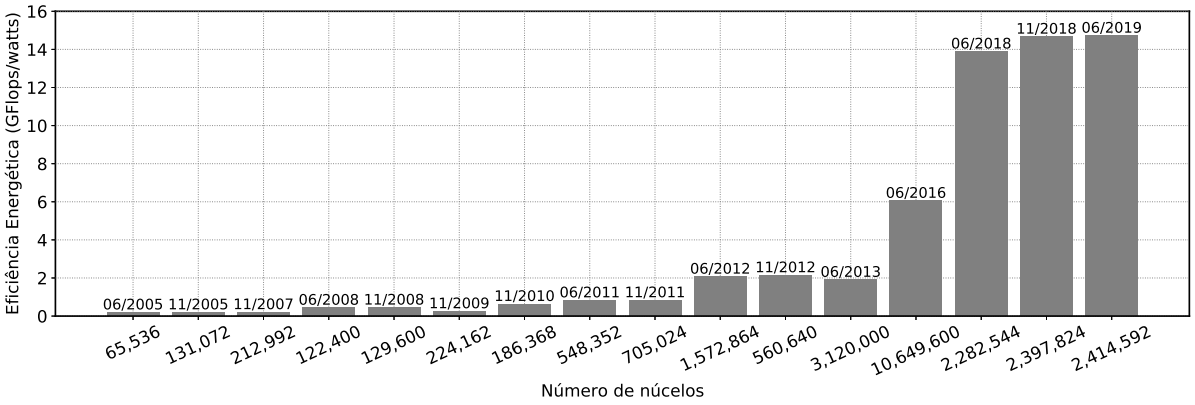
Na última década, a indústria de semicondutores vem investindo largamente na pesquisa e produção de *chips* com múltiplos núcleos de processamento em seu interior, chamados de *multicore*. Os avanços nessa indústria, juntamente com a área de arquitetura de computadores, são notados desde a década de 1980 em diante, permitindo um crescimento anual em desempenho de 40% a 50% (LARUS; KOZYRAKIS, 2018) para uma outra classe de processadores nesse período, os de um único núcleo ou *single core*. Porém, a necessidade de uma nova classe de processadores mostrou-se eminente ao se atingir um ponto onde o *trade-off* 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 pelas *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 os *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 os anos, segundo o *ranking* TOP500, 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.[1]

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 que realiza o *Exascale* deve atingir uma performance 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

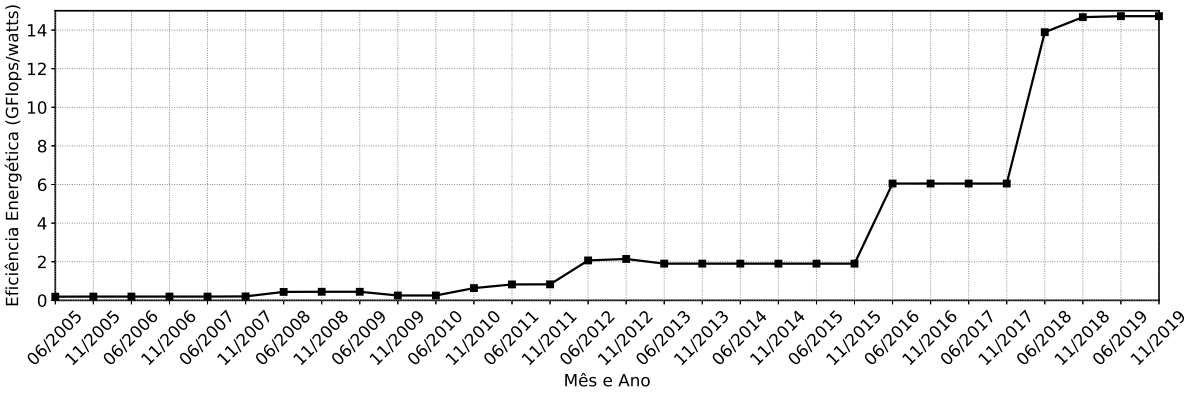
[1] Os dados do *ranking* TOP500 estão disponíveis no site TOP500: https://www.top500.org/

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 o *ranking* TOP500.



Fonte: Gráfico desenvolvido pelo autor.

Figura 2 – Evolução da eficiência energética do supercomputador número 1 do mundo segundo o *ranking* TOP500.



Fonte: Gráfico desenvolvido pelo autor.

poderosos do mundo desde 2005[2], segundo o *ranking* TOP500.

Buscando novos tipos de arquiteturas paralelas que apresentem as características faltantes no problema de balanceamento apresentado acima, pesquisadores da área de 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; UL-ABDIN, 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 *ranking* TOP500, obtendo duas vezes mais desempenho que esse e reduzindo em três vezes o consumo energético, explicando também o ganho

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

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.* 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. Ao realizar a otimização do *benchmark*, *David et al.* o portaram com a MPPA Asynchronous Communication API (ASYNC), uma API com maior nível de abstração e com conceitos de assincronismo, aumentando o potencial de desempenho da aplicação. Além disso, alterações na implementação de todas as aplicações foram realizadas de modo a otimizá-las ainda mais.

Portanto, para realizar uma comparação justa entre as APIs citadas acima, faz-se necessário atualizar a lógica de implementação das aplicações da versão antiga do *benchmark*, para que essas se equivalham às novas implementações, criando assim um ambiente propício para comparar aspectos puramente das tecnologias de comunicação citadas, utilizando as duas versões do *benchmark* para isso. A comparação entre ambas as implementações será responsável por determinar qual API se comporta de maneira mais robusta no MPPA-256 em certos contextos, onde os dados acerca do tempo de execução, quantidade de dados enviados e recebidos e gasto energético de cada aplicação serão as métricas para essa determinação. Assim, teremos dados de execução das duas APIs numa mesma versão de placa do MPPA-256, resultando numa base de dados concreta para a tomada de decisão sobre qual das duas APIs escolher na hora de implementar uma nova aplicação.

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

O objetivo deste trabalho é obter dados concretos acerca da execução de aplicações de diversos domínios de problemas no MPPA-256, utilizando as duas APIs já citadas e o CAP Bench, podendo assim comparar as execuções de cada aplicação em cada cenário específico possível dentro do processador, obtendo identificadores precisos que, em momentos futuros, possam apontar qual das duas APIs utilizar, dependendo do domínio de problema de uma certa aplicação.

1.1.2 Objetivos Específicos

- Investigar a viabilidade do uso do MPPA-256 para a área de HPC.
- Estudar aspectos das APIs de comunicação existentes no MPPA-256, mais especificamente, a ASYNC e a IPC.
- 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 de uma das duas numa futura implementação.

1.2 CONTRIBUIÇÕES DO TRABALHO

Este trabalho é continuação de um projeto de iniciação científica desenvolvido por *David et al.*, o qual resultou em um resumo expandido 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 dessa dissertação. O Capítulo 3 apresenta alguns trabalhos relacionados a este. O Capítulo 4 contém toda a proposta deste projeto, detalhando tudo que foi feito, assim como explicando as métricas que serão expostas nos resultados. O Capítulo 5 apresenta os resultados preliminares já obtidos. Para finalizar, o Capítulo 6 conclui este trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados conceitos relacionados a Computação Paralela, por exemplo, padrões arquiteturais, na Seção 2.1, e tecnologias de programação, na Seção 2.3. Também são mostrados algumas características do MPPA-256 na Seção 2.2.

2.1 ARQUITETURAS PARALELAS

Existem três tipos de sistemas com múltiplos processadores, segundo *Tanenbaum et al.*: os multiprocessadores, os multicomputadores e os sistemas distribuídos (TANENBAUM; BOS, 2014). São detalhados nesta seção conceitos acerca das arquiteturas multiprocessadores e multicomputadores.

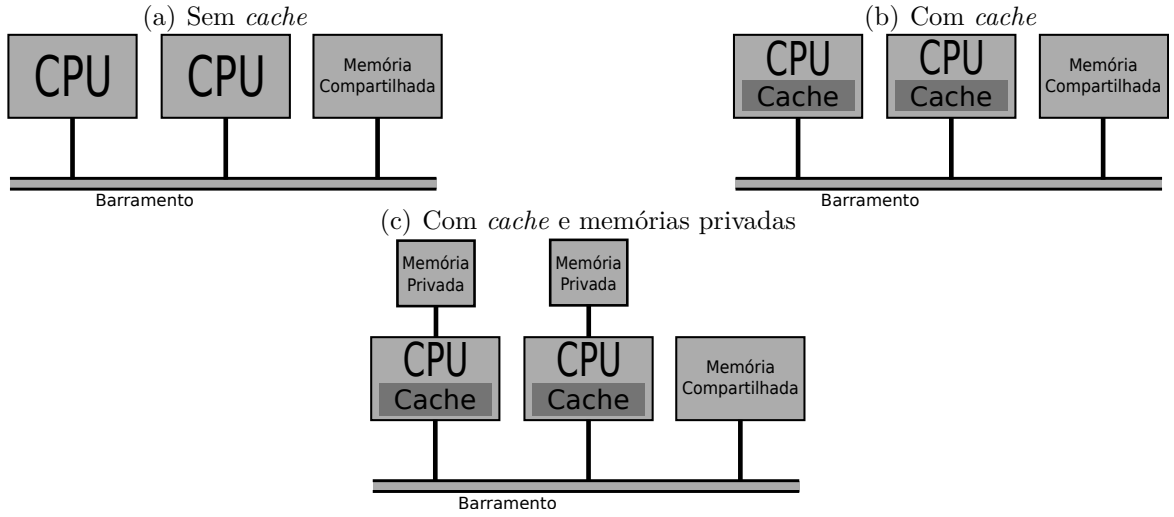
2.1.1 Multiprocessadores

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). Programas executando em qualquer um dessas CPUs possuem espaços de endereçamento físicos únicos na RAM e as *threads* de um desses programas fazem uso de um mesmo espaço de memória, através de operações de escrita e leitura, para se comunicarem. Um fato peculiar dessa arquitetura é a possibilidade de ocorrer problemas de concorrência quando duas ou mais *threads* de um mesmo programa executam em diferentes CPUs, onde, na visão de uma CPU, ela escreve um valor em uma posição de memória e lê outro valor daquela mesma posição, pois uma *thread* executando em outra CPU alterou o valor daquela posição.

Multiprocessadores são também classificados em dois tipos, de acordo com a velocidade de acesso a uma posição de memória. Quando uma certa palavra na memória pode ser lida na mesma velocidade que qualquer outra, são chamados de multiprocessadores com acesso uniforme a memória - Uniform Memory Access (UMA). Já quando a velocidade de leitura em diferentes posições de memória muda, são chamados de multiprocessadores com acesso não uniforme a memória - Nonuniform Memory Access (NUMA).

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 esteja, 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 pequeno e tolerável. Porém, quando elevam-se em algumas dezenas o número de CPUs

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).

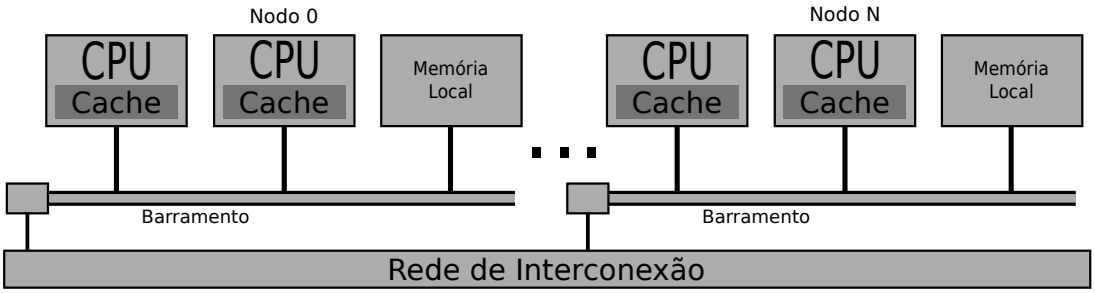
observa-se o principal problema deste exemplo de arquitetura UMA: a ociosidade, por muito tempo, de grande parte das CPUs, enquanto aguardam pelo acesso ao barramento.

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, o código do programa, strings e pilhas, utilizando assim esta segunda configuração de forma otimizada. Ambas configurações removem grande parte do tráfego no barramento, tornando seu uso exclusivamente para as variáveis compartilhadas entre *threads*.

A adição de *caches* impõe 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 provém essa escalabilidade implementando um espaço de endereçamento único para todas as CPUs através de uma rede de interconexão, como na Figura 4, o que causa a diferença nos tempos de acesso, os quais serão totalmente dependentes do local da memória que se deseja acessar um valor 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, também chamadas de núcleos (em inglês *cores*), 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 também 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 coerencia (*coherency wall*) (TANENBAUM; BOS, 2014).

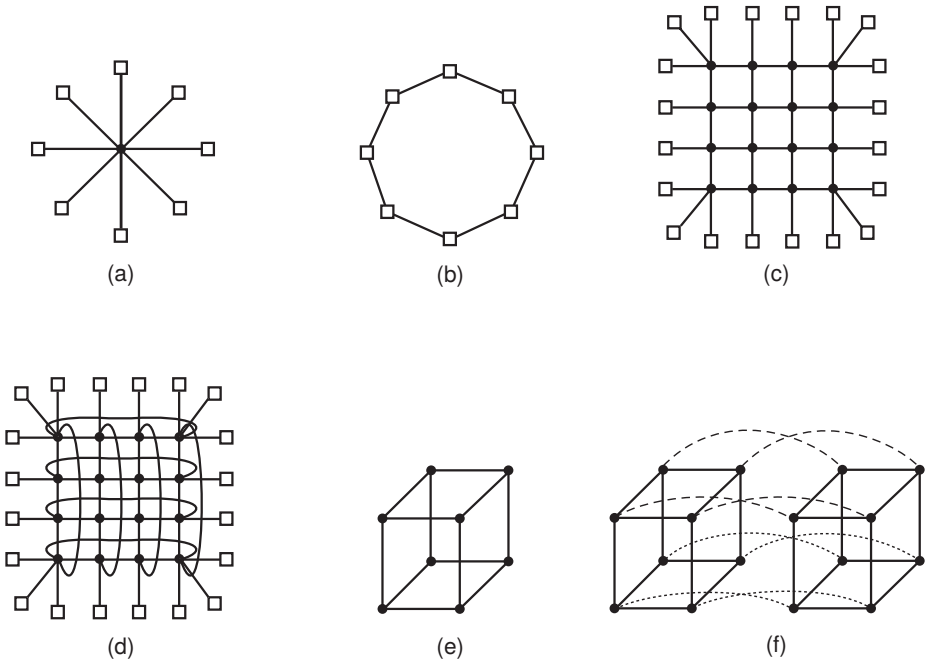
Para o futuro dos *manycore*, espera-se processadores que invistam mais na comunicação entre CPUs através da troca de mensagens extremamente rápidas via *hardware* e através de uma memória compartilhada, deixando de lado parte da coerência de *cache*. Uma Unidade de Processamento Gráfico (GPU) é um dos 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 então 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

Figura 5 – Tipos de topologias de rede de multicomputadores.



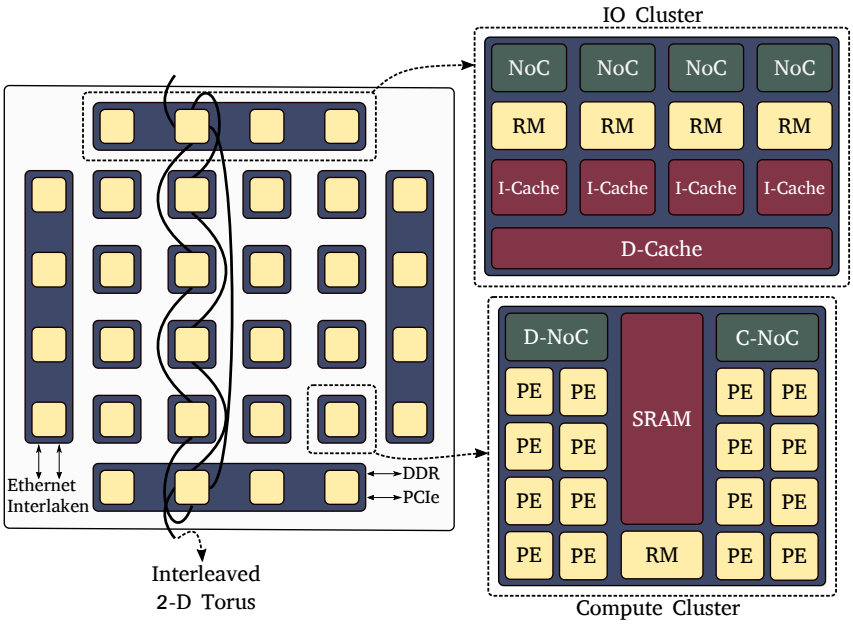
Fonte: (TANENBAUM; BOS, 2014).

um *switch* para conectar os nós entre si, os quais são então organizados em forma de estrela, como na Figura 5(a). Também é possível organizar os nós em forma de anel, onde cada nó se 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, se dá através da troca de mensagens entre estes. Basicamente, o Sistema Opera-

Figura 6 – Visão arquitetural simplificada do MPPA-256.



Fonte: (PENNA et al., 2018).

cional (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 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 *manycore*. 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 Rede-em-Chip (Network-on-Chip (NoC)), um para dados e outro para controle. Cada núcleo possui duas memórias *cache*, uma para dados e outra para instruções. As *caches* são associativas *2-way* privadas e possuem 32kB (PODESTá et al., 2018).

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

2.3 DESENVOLVIMENTO DE APLICAÇÕES PARALELAS

Durante o domínio de processadores *single core* no mercado, para que uma aplicação ganhasse aumento em performance, bastava esta ser executada em um processador com maior frequência de *clock*. 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.

Diversas APIs voltadas a programação paralela foram criadas para simplificar tanto a solução desses problemas como o desenvolvimento de aplicações que implementam paralelismo. Abaixo 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.

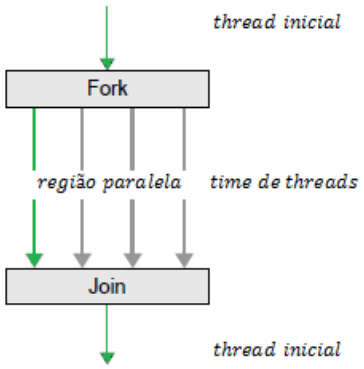
2.3.1 Bibliotecas multiplataforma

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.

```
1    static void createArray() {
2      int array[max], index;
3      int index_max = 10;
4      #pragma omp parallel for
5      for (index = 0; i < index_max; index++)
6          array[index] = index;
7    }
```

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* se dá 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 código 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 código 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`. Ambas implementações mostram que, com a adição de uma única linha, muda-se completamente o fluxo de execução do programa, 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

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

```
1   static int partial_friendly_sum = 0;
2   ...
3   static void countFriends() {
4     int i; /* Loop indexes. */
5
6     #pragma omp parallel for private(i) default(shared) reduction(+: partial_friendly_sum)
7     for (i = offset; i < offset + tasksize; i++) {
8       for (int j = 0; j < i; j++) {
9         if ((allTasks[i].num == allTasks[j].num) && (allTasks[i].den == allTasks[j].den))
10          partial_friendly_sum++;
11      }
12    }
13  }
```

Fonte: o autor.

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 ao terminar a iteração atual. Por fim, `guided` é similar ao `dynamic`, porém, a *thread* começa recebendo um grande número de iterações, e se adaptando conforme executa, recebendo mais ou menos iterações, dependendo do tempo que leva para executá-las.

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 tendo acesso a uma determinada região de memória. 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`, como primeiro parâmetro e o endereço da variável que será armazenado o *id* do processo como segundo. Também

Listagem 3 – Exemplo de uma aplicação usando a MPI.

```
1  int main(int argc, char **argv) {
2    int rank, size;
3
4    char mensagem_bcast[25] = "Transmitindo um broadcast";
5    char mensagem_bcast_recebido[18];
6
7    MPI_Init(argc, argv);
8    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9    MPI_Comm_size(MPI_COMM_WORLD, &size);
10   MPI_Bcast(&mensagem_bcast, 25, MPI_CHAR, 0, MPI_COMM_WORLD);
11
12   if (rank > 0) {
13     mensagem_bcast_recebido[18] = "Broadcast recebido";
14     MPI_Send (&mensagem_bcast_recebido, 18, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
15   } else if (rank == 0) {
16     for (int i = 1; i < size; i++)
17       MPI_Recv(&mensagem_bcast_recebido, 18, MPI_CHAR, i, MPI_ANY_TAG, MPI_COMM_WORLD,
18   }
19
20   MPI_Finalize();
21   return 0;
22 }
```

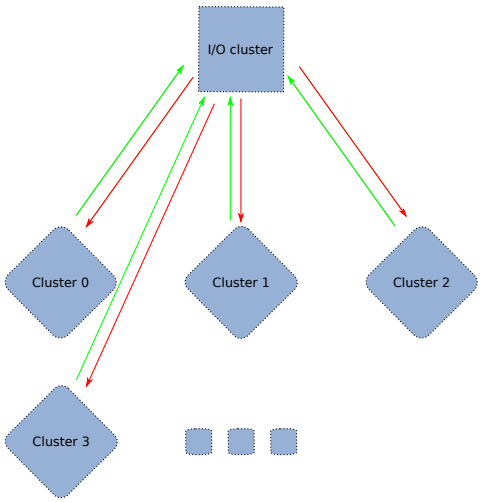
Fonte: o autor.

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 ambas 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 código 3 mostra um exemplo de implementação simples usando o MPI. Neste exemplo, na linha 11 o processo com *id* igual a 0 envia um *broadcast* para todos os outros processos, os quais respondem na linha 16 ao processo de *id* 0, que recebe esta resposta na linha 19. Assim, as linhas 18-20 são executadas somente pelo processo com *id* igual a 0, enquanto que as linhas 14-16 são executadas por processos com *id* maior que 0.

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



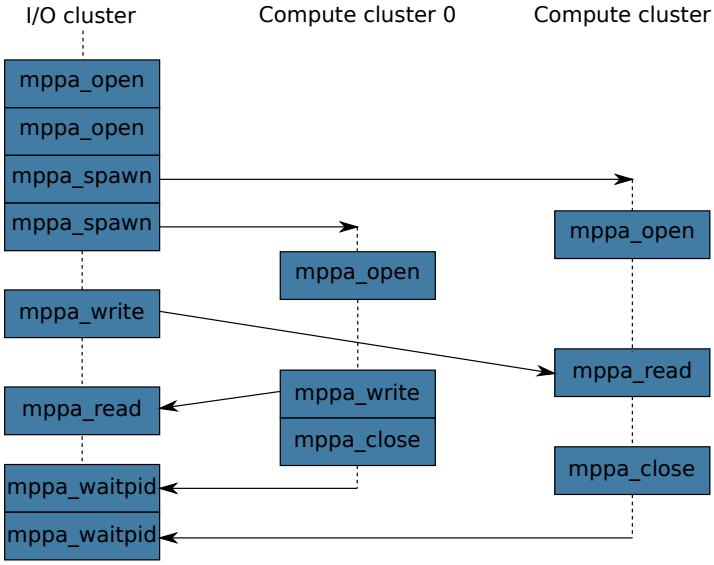
Fonte: Documentação do MPPA sobre IPC.

2.3.2 Bibliotecas específicas para o MPPA-256

Na arquitetura do MPPA-256 existem *clusters* que gerenciam outros *clusters*, chamados de mestres, e *clusters* específicos para a realização de computações, chamados de escravos. Por isso, as bibliotecas do processador fornecem funções voltadas a implementação de aplicações do tipo *mestre/escravo*, exemplificado na Figura 8, sendo uma das principais diferenças do processador para um x86 a divisão do programa em duas partes, cada uma compilada para um dos dois tipos de *clusters*. 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*. Ambas ASYNC e IPC seguem esses conceitos na implementação de suas abstrações, porém, por ser mais nova, a ASYNC permite uma abstração maior que a IPC.

Com a IPC, processos mestres localizados em um dos quatro *clusters* de E/S conseguem *spawnar* 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.

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

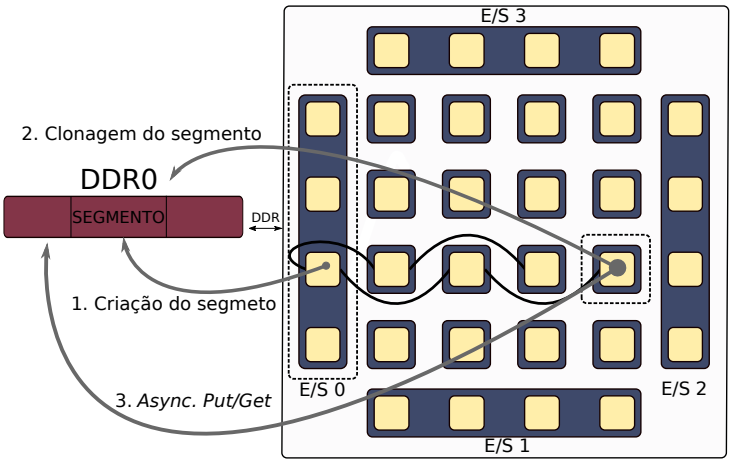


Fonte: Documentação do MPPA sobre IPC.

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 a Figura 9. Nela, o processo mestre *spawna* dois processos escravos, os quais abrem portais que definem um caminho para o *cluster* que os *spawnou*. Um destes processos não faz uso da função `mppa_read()` pois recebeu todos os dados necessários através dos parâmetros passados 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()`.

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

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



Fonte: o autor.

porção de memória, não realizando qualquer 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 disso, é preciso inicializar o contexto antes de realizar 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 códigos 4 e 5 mostram como são definidas essas macros em um *cluster* de E/S e em um CC, respectivamente.

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

```
1  /* Synchronization between slaves and IO. */
2  #define send_signal(i)                                    \
3  poke(mppa_async_default_segment((i)), sig_offsets[(i)], 1) \
4
5  /* Synchronization between slaves and IO. */
6  #define wait_signal(i) {                                   \
7  waitCondition(&cluster_signals[(i)], 1, MPPA_ASYNC_COND_EQ, NULL); \
8  cluster_signals[(i)] = 0;                                  \
9  }
```

Fonte: o autor.

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

```
1  /* Synchronization between slaves and IO. */
2  #define send_signal()                        \
3  poke(MPPA_ASYNC_DDR_0, sigback_offset, 1) \
4
5  /* Synchronization between slaves and IO. */
6  #define wait_signal() {                                \
7  waitCondition(&io_signal, 1, MPPA_ASYNC_COND_EQ, NULL); \
8  io_signal = 0;                                         \
9  }
```

Fonte: o autor.

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. Segundos 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.

4 DESENVOLVIMENTO

5 RESULTADOS

6 CONCLUSÃO

Primeiro parágrafo da seção com uma frase sem sentido que só serve para ocasionar uma quebra e de demonstrar a configuração de indentação da primeira linha. Essa frase está aqui pois parágrafos de uma linha são feios.

Resultado do uso de siglas:

- Sigla que nunca expande: API;
- Sigla normal, expande no primeiro uso: Distributed Hash Table (DHT), mas não no segundo: DHT;
- Siglas com plurais automaticos: APIs e DHTs;
- Plural não-trivial: Square Matrices (SQs);
- Forçando uma expansão (e no plural) Distributed Hash Tables (DHTs);
- Usando uma sigla cujo comando é diferente da sigla: World Wide Web Consortium (W3C).

Resultado do glossário:

- Dois termos, polling e proxy;
- Plural: proxies.

Resultado de `index`: primeiro um link normal tomate, depois um capitalizado Tomate.

Definição 6.1. Exemplo de definição

Teorema 6.1. Exemplo de teorema

Prova 6.1. Exemplo de prova □

(Sub)enumerações e citações (verificar se OK com o idioma):

1. (**??**):
   a) **??**):
      i. Dijkstra (1968);
2. (**??**DIJKSTRA, 1968);
3. **??**Dijkstra (1968).

Resultado de `\autoref`s:

- Listagem 6;
- Algoritmo 1;
- Figura 11 tem subfiguras:
  - Figura 11(a)

Listagem 6 – Meta informações do presente documento.

```
\titulo{Template \LaTeX{} para testes e dissertações do LAPESD/UFSC}
\autor{Omar Ravenhurst}
\data{1 de agosto de 2019} % ou \today
\tese % ou \dissertacao
\titulode{Doutor em Ciência da Computação}
\orientador{Prof. Ben Trovato, Dr.}
\coorientador{Prof. Lars Thørväld, Dr.}

\membrobanca{Prof. Valerie Béranger, Dr.}{Universidade Federal de Santa Catarina}
\membrobanca{Prof. Mordecai Malignatus, Dr.}{Universidade Federal de Santa Catarina}
\membrobanca{Prof. Huifen Chan, Dr.}{Universidade Federal de Santa Catarina}
\coordenador{Prof. Charles Palmer, Dr.}
```

Fonte: o autor.

Algoritmo 1 – Exemplo do ambiente algorithimic.

```
1: procedure Closure(C, A)
2:     H ← ∅                                              ▷ Direct cache
3:     for i ∈ [1, n] do                  ▷ Parallel, (dynamic,32) scheduling
4:         H ← H ∪ DoImportantStuff(i)
```

Fonte: o autor.

Figura 11 – Exemplo de figura com duas subfiguras.

(a) O Makefile compila SVGs em PDFs usando o inkscape

(b) Brasão da UFSC.

Fonte: o autor.

– Figura 11(b)

- Tabela 2;
- ??;
- ??;
- ??;
- ??;
- ??.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio

Tabela 2 – Exemplo de tabela e símbolos

Esquerda	Coluna 1	90 graus	Parágrafo com `p{5cm}`
r_1	✓	✗	①
r_2	merged cell		②
r_3	③	④	⑤
r_4	⑥	⑦	⑧
r_5	⑨	x	y

Fonte: o autor.

Figura 12 – Segunda Figura.

Fonte: o autor.

placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

APÊNDICE A – EXEMPLO DE APÊNDICE

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

GLOSSÁRIO

polling A type of event delivery technique consisting of the consumer repeatedly asking the provider for the most recent events. 55

proxy A proxy shapiro1986 encapsulates remote servers and provides a single view to their services. The proxy can then intercept communication and provide additional functionality, such as message translation and performance enhancement. The client must take the initiative of selecting and using a proxy [p. 46,97]fielding2000.. 55

ÍNDICE

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.