XXIV Escola Regional de Alto Desempenho da Região Sul

XXIV ERAD/RS

Florianópolis, SC

24, 25 e 26 de abril de 2024

MINICURSOS

REALIZAÇÃO

SBC Sociedade Brasileira de Computação

CE-ACPAD

ORGANIZAÇÃO

UniSENAI

UDESC

unioeste

APOIO

UNIJUÍ Universidade Regional

UFPR Universidade Federal do Paraná

UFRGS Universidade Federal do Rio Grande do Sul

UFSC

Universidade Federal de Santa Maria 1960

LNCC Laboratório Nacional de Computação Científica

INSTITUTO FRIOL

PATROCINADORES

DIAMANTE

SDC Smart Data Computing

DELL Technologies

VIRCOS

ASRock Rack

NVIDIA

Eletrobras

OURO

fineasy TECH

versatus HPC

LexisNexis Risk Solutions

Hewlett Packard Enterprise

PRATA

ENGIE

CIPHER

BRONZE

NEWROUTE

ARTHUR FRANCISCO LORENZON (ED) – UFRGS
MARCO ANTONIO ZANATA ALVES (ED) - UFPR

MINICURSOS DA XXIV ESCOLA REGIONAL DE ALTO DESEMPENHO DA REGIÃO SUL

Porto Alegre
Sociedade Brasileira de Computação – SBC
2024

Dados Internacionais de Catalogação na Publicação (CIP)

E74 Escola Regional de Alto Desempenho da Região Sul (24. : 24 – 26 abril 2024 : Florianópolis)

Minicursos da ERAD-RS 2024 [recurso eletrônico] / organização: Arthur Lorenzon ; Marco Antonio Zanata Alves. Dados eletrônicos. – Porto Alegre: Sociedade Brasileira de Computação, 2024.

68 p. : il. : PDF ; 4.9MB

Modo de acesso: World Wide Web.
Inclui bibliografia
ISBN 978-85-7669-579-0 (e-book)

1. Computação – Brasil – Evento. 2. Processamento de Alto Desempenho. 3. Programação paralela. I. Lorenzon, Arthur. II. Alves, Marco Antonio Zanata. III. Sociedade Brasileira de Computação. VI. Título.

CDU 004(063)

Ficha catalográfica elaborada por Annie Casali – CRB-10/2339

Biblioteca Digital da SBC – SBC OpenLib

Índices para catálogo sistemático:

1. Ciência e tecnologia dos computadores : Informática – Publicação de conferências, congressos e simpósios etc. ... 004(063)

ERAD/RS 2024

XXIII Escola Regional de Alto Desempenho da Região Sul
24 a 26 de abril de 2024
https://cradrs.github.io/eradrs2024/

A XXIV Escola Regional de Alto Desempenho da Região Sul (ERAD/RS 2024) será realizada entre os dias 24 e 26 de abril de 2024, no Centro Universitário SENAI Santa Catarina (UniSENAI), em Florianópolis/SC. A ERAD/RS é realizada anualmente pela Sociedade Brasileira de Computação (SBC) e pela Comissão Regional de Alto Desempenho da Região Sul (CRAD/RS).

O público alvo da ERAD/RS 2024 são alunos, profissionais e professores/pesquisadores que atuam direta ou indiretamente na computação de alto desempenho e em áreas correlatas. O evento engloba os três Estados da região sul do Brasil (RS, SC e PR).

Os principais objetivos são:

- Qualificar os profissionais do sul do Brasil nas áreas que compõem o processamento de alto desempenho;
- Prover um fórum regular onde possam ser apresentados os avanços recentes nessas áreas;
- Discutir formas de ensino de processamento de alto desempenho nas universidades.

A programação da ERAD/RS 2024 foi composta por sessões técnicas, com a apresentação de 44 trabalhos nos Fóruns de Iniciação Científica e de Pós-graduação. Além disso, o evento proporcionou aos participantes 6 minicursos, 3 palestras científicas, 8 palestras industriais, os workshops de Inteligência Artificial (WIA) e Women in High Performance Computing (WHPC), uma Maratona de Programação Paralela, 2 tutoriais e a reunião anual da CRAD/RS.

A edição da 2024 da ERAD/RS foi coordenada pelos professores Eduardo Camilo Inacio (UFSC/UniSENAI), Guilherme Galante (Unioeste) e Guilherme Piêgas Koslovski (UDESC). O Fórum de Iniciação Científica teve coordenação dos professores Charles Miers (UDESC) e Luis Bona (UFPR). Na coordenação do Fórum de Pós-graduação, estiveram os professores Edson Padoin (Unijuí) e Márcio Castro (UFSC). Os Minicursos foram coordenados pelos professores Arthur Lorenzon (UFRGS) e Marco Alves (UFPR). A Maratona de Programação Paralela teve coordenação dos professores Maurício Pillon (UDESC), João Lima (UFSM) e Odorico Mendizabal (UFSC), além da colaboração especial do professor Calebe Bianchini (Mackenzie). A frente da coordenação do WHPC, esteve a professora Luciana Frigo (UFSC). Na coordenação do WIA, estiveram os professores Matheus Serpa (UFRGS) e Manuel Binelo (Unijuí). A equipe de apoio local foi composta pelos professores Willian Daniel de Mattos, Henrique Benedetto Neto e Maristela Schleicher Silveira, e pelos colaboradores Lucas Rack e Fernando Tironi, todos do UniSENAI, além dos discentes Bruno Miranda (UFSC), Gabriel Moura, Gregori Monteiro, João Rizzo, Laura Ferrari, Madson Carvalho e Yuri Plotze, estes do UniSENAI.

Índice

Mensagem da Coordenação Geral

Com imensa satisfação, saudamos calorosamente à vigésima quarta edição da Escola Regional de Alto Desempenho da Região Sul, a ERAD/RS 2024. Nesta edição, o evento tem lugar na cidade de Florianópolis, capital do Estado de Santa Catarina, entre os dias 24 e 26 de abril de 2024. A ERAD/RS é um evento promovido anualmente pela Sociedade Brasileira de Computação (SBC), pela Comissão Especial de Arquitetura de Computadores e Processamento de Alto Desempenho (CE-ACPAD) e pela Comissão Regional de Alto Desempenho da Região Sul (CRAD/RS) desde 2001.

Esta escola, com foco primordialmente regional, visa capacitar profissionais da Região Sul do Brasil nas áreas relacionadas ao Processamento de Alto Desempenho (PAD) e fornecer um espaço regular para a apresentação dos mais recentes avanços nesses domínios, além de compartilhar e discutir metodologias de ensino do tema nas Instituições de Ensino Superior (IES) do sul do Brasil.

Adicionalmente, a ERAD/RS, por meio das atividades realizadas, promove a integração entre a academia, a sociedade e a indústria. Ao longo dos três dias do evento, as palestras científicas e industriais, conduzidas por especialistas renomados, as apresentações de trabalhos acadêmicos, os minicursos, os workshops, a maratona de programação paralela e os tutoriais, incentivam a troca de experiências e o compartilhamento do conhecimento. A interação entre estudantes, professores, pesquisadores e profissionais fortalece ainda mais essa sinergia, proporcionando diferentes perspectivas e oportunizando uma análise mais abrangente de questões atuais e futuras para a área.

Com o intuito de ampliar seu alcance e maximizar seu impacto, a ERAD/RS é itinerante, sendo sediada por diferentes instituições dos três estados da Região Sul a cada ano. Em 2024, a responsabilidade pela organização deste encontro recai sobre o Centro Universitário SENAI Santa Catarina (UniSENAI), com o apoio da Universidade do Estado de Santa Catarina (UDESC), da Universidade Estadual do Oeste do Paraná (Unioeste) e da Universidade Federal de Santa Catarina (UFSC). A Região Sul do Brasil tem sido reconhecida por muitos anos como referência na área de PAD e a ERAD/RS reflete esse reconhecimento e reafirma essa posição de destaque ao investir na formação de novos pesquisadores e na atualização dos pesquisadores que estabeleceram suas bases nos estados da região.

Estar à frente da organização deste importante e tradicional evento foi uma grande honra e satisfação. Reconhecemos também a significativa contribuição das instituições que historicamente acompanham e participam de forma efetiva da realização e evolução da ERAD/RS todos os anos. Nesta edição, a ERAD/RS teve o apoio, por meio de patrocínio, das empresas Vircos Tecnologia, SDC, ASRock Rack, Dell Technologies, NVIDIA, Fineasy Tech, Versatus HPC, LexisNexis Risk Solutions, Hewlett Packard Enterprise, Cipher Phone e New Route.

Expressamos nosso agradecimento especial a todos os autores e autoras que submeteram seus trabalhos para as sessões técnicas, ao corpo de revisores que compuseram o comitê de programa, aos patrocinadores e aos convidados, que gentilmente aceitaram nossos convite e, sem dúvida, enriqueceram a ERAD/RS com suas contribuições.

Por último, mas não menos importante, agradecemos a todos os coordenadores, que estiveram à frente da organização das diferentes atividades oferecidas na ERAD/RS, por terem assumido este desafio, com todas as suas responsabilidades, e pelo sucesso na sua condução.

Agradecemos a todos pela presença e desejamos que a ERAD/RS 2024 seja extremamente proveitosa.

Eduardo Camilo Inacio (UFSC/UniSENAI), Guilherme Galante (Unioeste) e Guilherme Koslovski (UDESC)
Coordenadores Gerais da ERAD/RS 2024

Mensagem da Coordenação dos Minicursos

A Escola Regional de Alto Desempenho da Região Sul (ERAD/RS) é um evento anual promovido pela Sociedade Brasileira de Computação (SBC) e pela Comissão Regional de Alto Desempenho da Região Sul (CRAD/RS). A escola, que neste ano completa os seus vinte e quatro anos, foi realizada entre os dias 24 e 26 de abril de 2024, na cidade de Folrianópolis/RS, na UniSENAI.

Um dos objetivos da ERAD/RS é qualificar profissionais da região sul nas diversas áreas que envolvem e Processamento de Alto Desempenho (PAD). Com este intuito, todo o ano são selecionados minicursos introdutórios e avançados em tópicos estratégicos e de interesse à comunidade. Não diferente, neste ano de 2024 foram selecionados seis minicursos, dos quais três estão formatados como capítulo deste livro. Os minicursos aqui representados, apresentam tópicos da área de PAD, os quais irão certamente agradar os participantes do evento. Nesta edição estamos investigando temas como introdução à exploração do paralelismo em arquiteturas *multicore*, tópicos avançados de programação paralela com o modelo de tarefas em OpenMP, e uma discussão sobre o impacto do uso de contêineres na programação paralela. A área de PAD continua a evoluir, revisitando conhecimentos consolidados de OpenMP, computação na nuvem e estratégias de exploração de paralelismo em resposta a demandas emergentes em setores diversos como saúde, cidades inteligentes, agricultura e automação de sistemas, entre outros.

Os coordenadores dos minicursos expressam gratidão aos autores por compartilharem seus conhecimentos através da submissão de minicursos de alto nível para esta edição da escola. Agradecemos também aos coordenadores e organizadores da ERAD/RS 2024 pelo apoio na seleção dos minicursos e na realização do evento. Acreditamos firmemente que o conhecimento é o motor de uma nova sociedade focada em tecnologia e ética, e os conteúdos apresentados nesta edição refletem esse compromisso. Desejamos a todos uma excelente ERAD/RS 2024, repleta de aprendizado e troca de experiências.

Arthur Francisco Lorenzon (UFRGS) e Marco Antonio Zanata Alves (UFPR)
Coordenadores dos Minicursos da ERAD/RS 2024

Comitês Organizadores

Coordenação Geral

- Eduardo Camilo Inacio (UniSENAC)
- Guilherme Koslovski (UDESC)
- Guilherme Galante (UNIOESTE)

Fórum de Pós-Graduação

- Edson Padoin (Unijui)
- Márcio Castro (UFSC)

Fórum de Iniciação Científica

- ACharles Miers (UDESC)
- Luis Bona (UFPR)

Minicursos

- Arthur Lorenzon (UFRGS)
- Marco Alves (UFPR)

Maratona Paralela

- Mauricio Pilon (UDESC)
- João Lima (UFSM)
- Odorico Mendizabal (UFSC)
- Calebe Bianchini (Mackenzie)

Women in HPC

- Luciana Frigo (UFSC)

WIA

- Matheus Serpa (UFRGS)
- Manuel Binelo (Unijui)

Minicursos

Capítulo 1

Exploração do Paralelismo nas Arquiteturas de Computadores Atuais

Guilherme Galante

Resumo

A exploração do paralelismo em arquiteturas de computadores atuais tem se tornado uma área de extrema relevância e interesse, impulsionada pela necessidade de lidar com cargas de trabalho cada vez mais complexas e exigentes. Com o avanço das tecnologias e a demanda por desempenho computacional escalável, tem-se buscado diferentes formas de aproveitar o paralelismo em diversos níveis, desde instruções vetoriais do processador até a exploração de sistemas de memória distribuída e aceleradores. Essa abordagem permite a execução de múltiplas tarefas simultaneamente, resultando em ganhos significativos de desempenho. Sob este escopo, este minicurso tem o objetivo de oferecer uma visão geral dos diferentes aspectos do paralelismo nas arquiteturas de computadores atuais.

1.1. Introdução

Recentemente, a demanda por capacidades computacionais tem crescido exponencialmente, impulsionada por uma variedade de aplicações e setores. A sociedade atual depende cada vez mais da computação para impulsionar a inovação, resolver problemas complexos e melhorar a eficiência em diversas áreas de conhecimento [Pacheco and Malensek 2021]. Algumas das áreas que demonstram uma crescente demanda por capacidades de computação incluem:

- Ciência de Dados e Inteligência Artificial (IA): Com o aumento da coleta de dados, a análise e o processamento desses dados para extrair informações significativas requerem poder computacional considerável. Algoritmos de IA, como aprendizado de máquina e redes neurais, exigem enormes recursos de computação para treinamento e inferência.
- Pesquisa Científica e Simulações: Disciplinas como física, química, biologia e climatologia dependem fortemente de simulações computacionais para modelar fenô-

menos complexos, prever resultados e entender o mundo natural em um nível fundamental.

- Medicina e Biologia Computacional: A análise de dados genômicos, simulações de processos biológicos e o desenvolvimento de novos medicamentos são algumas das áreas que se beneficiam significativamente do poder computacional para acelerar a descoberta e a inovação.
- Finanças e Mercados: Em mercados financeiros altamente dinâmicos, algoritmos de negociação de alta frequência e modelagem de riscos exigem capacidades computacionais significativas para análise de dados em tempo real e tomada de decisões automatizadas.
- Entretenimento e Mídia: Com o avanço da tecnologia de realidade virtual (VR) e realidade aumentada (AR), assim como a produção de conteúdo multimídia de alta qualidade, a demanda por renderização de gráficos em tempo real e processamento de vídeo aumentou drasticamente.

Essas são apenas algumas das muitas áreas que têm grandes demandas por computação. Em essência, praticamente todos os campos da ciência, da indústria e da sociedade contemporânea dependem, em maior ou menor grau, do poder computacional para avançar em suas respectivas áreas de atuação.

Nesse contexto, uma das alternativas para satisfazer essas demandas é o emprego de computação paralela. Computação paralela é uma abordagem na qual múltiplas tarefas computacionais são executadas simultaneamente, seja dividindo uma única tarefa em partes menores que podem ser executadas em paralelo ou realizando tarefas independentes simultaneamente. Essa técnica é fundamental para lidar com cargas de trabalho intensivas e complexas, proporcionando um aumento significativo no desempenho e na eficiência dos sistemas computacionais.

A computação paralela envolve uma interação complexa entre hardware e software para aproveitar ao máximo os recursos disponíveis. No lado do hardware, empregam-se arquiteturas específicas, como processadores multicore, clusters de computadores e aceleradores especializados, para suportar a execução simultânea de múltiplas tarefas. Por outro lado, no aspecto do software, é fundamental ter algoritmos paralelos eficientes e estratégias de programação que distribuam efetivamente o trabalho entre os diversas unidades de processamento. Isso requer uma compreensão profunda da arquitetura subjacente do hardware e a habilidade de desenvolver código que possa ser executado de forma concorrente e coordenada.

1.2. Explorando o paralelismo nas arquiteturas atuais

O uso de paralelismo em arquiteturas modernas desempenha um papel fundamental no aumento do desempenho e na eficiência dos sistemas computacionais. Em vez de depender exclusivamente do aumento da frequência do clock dos processadores para melhorar o desempenho, como feito no passado, as arquiteturas modernas exploram o paralelismo em vários níveis. Isso inclui a utilização de instruções vetoriais e SIMD (*Single Instruction, Multiple Data*), o uso de GPUs e de arquiteturas paralelas de memória comparti-

lhada e distribuída [Wilkinson and Allen 2005]. Nas próximas seções são apresentados mais detalhes sobre os diferentes níveis de paralelismo nas arquiteturas e os modelos de programação que podem se usados em cada um deles.

1.2.1. Extensões vetoriais

As extensões vetoriais permitem que os processadores modernos realizem operações em paralelo em conjuntos de dados de forma muito mais eficiente do que seria possível com instruções de processamento escalares tradicionais. Existem várias extensões vetoriais disponíveis em processadores modernos, cada uma com suas próprias características e conjuntos de instruções.

Um exemplo são as instruções vetoriais *Advanced Vector Extensions* de 512 bits (AVX-512), introduzidas pela Intel em seus processadores x86. As instruções AVX permitem que um único processador execute operações em paralelo em múltiplos elementos de dados em um vetor, conhecido como vetor SIMD (*Single Instruction, Multiple Data*). Cada vetor SIMD pode conter vários elementos de dados, e uma única instrução AVX é capaz de realizar uma operação em todos esses elementos simultaneamente, resultando em um aumento significativo na taxa de execução de instruções e no desempenho geral do sistema.

As instruções AVX-512 são projetadas para operar em registradores de 512 bits. Isso significa que até 16 números de ponto flutuante de precisão simples (float de 32 bits) ou 8 números de ponto flutuante de dupla precisão (double de 64 bits) podem ser processados em paralelo em uma única instrução AVX. Além disso, as instruções AVX incluem uma ampla variedade de operações aritméticas, lógicas e de manipulação de dados, como adição, subtração, multiplicação, divisão, operações bitwise, mistura de dados e carregamento/armazenamento de dados de memória, o que permite uma ampla gama de operações paralelas em conjuntos de dados de diferentes tipos.

A Figura 1.1 mostra uma operação de soma de vetores usando as instruções vetoriais. Para o Intel AVX-512, pode-se armazenar 8 elementos de ponto flutuante de dupla precisão (*double*) em cada vetor, já que cada elemento desse tipo possui 64 bits e o vetor tem 512 bits, então 512/64 = 8 elementos *double*. A operação de soma é realizada para os 8 elementos usando uma única instrução SIMD. Como resultado, o Intel AVX pode potencialmente ser até 8 vezes mais rápido que a implementação escalar.

Note que ao escrever código que utiliza essas extensões vetoriais, é necessário verificar se o hardware de destino oferece suporte a essas instruções. Por exemplo, em sistemas operacionais Linux é possível verificar o suporte usando o comando:

```
grep avx /proc/cpuinfo
```

Como resposta ao comando exibe-se as flags do processador, destacando as extensões AVX (destacadas em vermelho), como ilustrado na Figura 1.2.

As extensões vetoriais podem ser exploradas de diferentes formas:

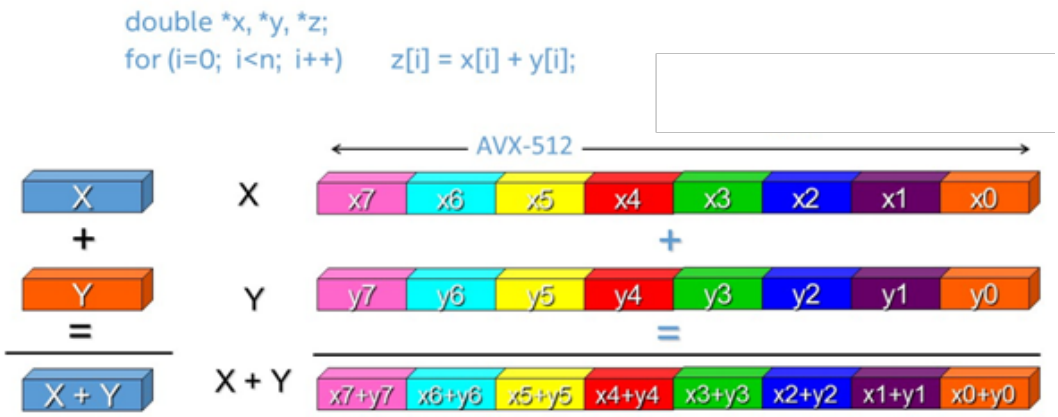

Figura 1.1. Soma de vetores usando AVX-512.

```
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts
 acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs
bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor
ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_
timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l2 invpcid_single cdp_l
2 ssbd ibrs ibpb stibp ibrs_enhanced tpr_shadow flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1
 avx2 smep bmi2 erms invpcid rdt_a avx512f avx512dq rdseed adx smap avx512ifma clflushopt clwb intel_
pt avx512cd sha_ni avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves split_lock_detect dtherm ida arat
 pln pts hwp hwp_notify hwp_act_window hwp_epp hwp_pkg_req vnmi avx512vbmi umip pku ospke avx512_vbmi
2 gfni vaes vpclmulqdq avx512_vnni avx512_bitalg avx512_vpopcntdq rdpid movdiri movdir64b fsrm avx512
```

Figura 1.2. Suporte a extensões vetoriais AVX.

- **Compilação com suporte AVX**: Os compiladores geralmente têm opções para habilitar o suporte a AVX. Ao compilar programas usando esses compiladores, você pode aproveitar automaticamente as instruções AVX.
- **Bibliotecas otimizadas**: Muitas bibliotecas populares de computação numérica, como *Intel Math Kernel Library* (MKL) e *Intel Integrated Performance Primitives* (IPP), são otimizadas para aproveitar as instruções AVX-512 quando disponíveis. Isso significa que, ao usar essas bibliotecas em seu código, você pode obter automaticamente benefícios de desempenho ao executar operações numéricas intensivas.
- **Programação manual**: Para aplicações que exigem otimização extrema ou para desenvolvedores que desejam tirar o máximo proveito das instruções AVX-512, é possível escrever código assembly ou Intrinsics AVX-512 diretamente. Os Intrinsics AVX-512 são funções fornecidas pelo compilador C (como o GCC ou o Clang) que correspondem diretamente às instruções de assembly AVX-512.

A Figura 1.3 mostra um exemplod e soma de vetores em Assembly e usando os C Intrinsics. As instruções AVX estão destacadas em cores. Em ambos os casos, há o carregamento dos vetores para os registradores vetorias (vermelho), a soma é realizada (azul), e então o resultado é atribuído a um vetor (verde). Note que a operação de soma dos vetores é realizada utilizando uma única instrução.

```
section .data
vec1 dd 1.0, 2.0, 3.0, 4.0    ; vetor 1
vec2 dd 4.0, 3.0, 2.0, 1.0    ; vetor 2
result dd 0.0, 0.0, 0.0, 0.0  ; vetor resultado

section .text
global _start

_start:
    ; Carregando os vetores
    vmovups zmm0, [vec1]
    vmovups zmm1, [vec2]

    ; Realizando a operação de soma
    vaddps zmm2, zmm0, zmm1

    ; Armazenando o resultado
    vmovups [result], zmm2
```

```
int main() {
    // Vetores de entrada
    float vec1[] = {1.0f, 2.0f, 3.0f, 4.0f};
    float vec2[] = {4.0f, 3.0f, 2.0f, 1.0f};
    // Vetor de resultado
    float result[16];

    // Carregando os vetores em registradores ZMM
    __m512 vec1_avx = _mm512_loadu_ps(vec1);
    __m512 vec2_avx = _mm512_loadu_ps(vec2);

    // Realizando a operação de soma
    __m512 result_avx = _mm512_add_ps(vec1_avx, vec2_avx);

    // Armazenando o resultado em memória
    _mm512_storeu_ps(result, result_avx);
}
```

Figura 1.3. Programação AVX: Assembly e Intrinsics.

1.2.2. Memória Compartilhada: Multicores e Multiprocessadores

Computadores paralelos de memória compartilhada são sistemas de computação nos quais múltiplos processadores ou núcleos de processamento têm acesso a um único espaço de memória compartilhada. Isso significa que todos os processadores/núcleos podem ler e escrever na mesma área de memória, facilitando a comunicação e o compartilhamento de dados. Os sistemas de memória compartilhada podem variar em escala, desde sistemas com alguns núcleos de processamento (multicore) até sistemas com centenas ou milhares de processadores.

Um processador multicore combina dois ou mais unidades de processador (chamadas núcleos) em uma única peça de silício. Normalmente, cada núcleo consiste em

todos os componentes de um processador independente, como registradores, ULA, hardware de pipeline e unidade de controle, além de cache L1 de instruções e dados. Além dos múltiplos núcleos, os chips multicore contemporâneos também incluem cache L2 e, cada vez mais, cache L3 [Stallings 2015], como ilustrado na Figura 1.4. Todos os núcleos podem acessar diretamente todos os dados armazenados na memória principal. Comparado com a arquitetura tradicional de núcleo único, arquitetura multicore fornece computação muito maior capacidade e é muito mais eficiente em termos energéticos.

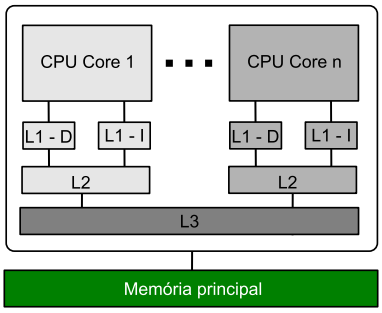

Figura 1.4. Organização de um processador multicore.

Embora os fabricantes de hardware continuem aumentando o número de núcleos em chips de CPU, o número de núcleos não pode ser aumentado ilimitadamente devido a limitações físicas. Para atender a necessidade de computadores com capacidades superiores de processamento, vários chips de CPU são integrados em uma única máquina. Esse tipo de arquitetura é chamada de multiprocessador.

Em sistemas de memória compartilhada com vários processadores, a interconexão pode conectar todos os processadores diretamente à memória principal ou cada processador pode ter uma conexão direta com um bloco de memória principal, e os processadores podem acessar os blocos de memória principal uns dos outros por meio de hardware especial incorporado aos processadores. No primeiro tipo de sistema, o tempo para acessar todos os locais de memória será o mesmo para todos os núcleos, enquanto no segundo tipo, um local de memória ao qual um núcleo está diretamente conectado pode ser acessado mais rapidamente do que um local de memória que precisa ser acessado por meio de outro chip. Assim, o primeiro tipo de sistema de sistema é chamado de sistema de acesso uniforme à memória (*Uniform memory access* - UMA), enquanto o segundo tipo é chamado de sistema de acesso não uniforme à memória (*Non-Uniform memory access* - NUMA).

Em teoria, máquinas UMA podem teoricamente alcançar um grande número de processadores, mas na prática, há limitações físicas e de design que podem restringir o número máximo de processadores em uma única máquina. Geralmente, sistemas UMA podem acomodar dezenas a centenas de processadores em uma única máquina, dependendo do projeto específico e das restrições de hardware. Por sua vez, sistemas NUMA podem escalar para um número significativo de processadores, mas a capacidade de expansão é geralmente mais flexível do que em sistemas SMP, podendo suportar centenas ou até mesmo milhares de processadores. Figura 1.5 ilustra a diferença entre os dois tipos de arquitetura.

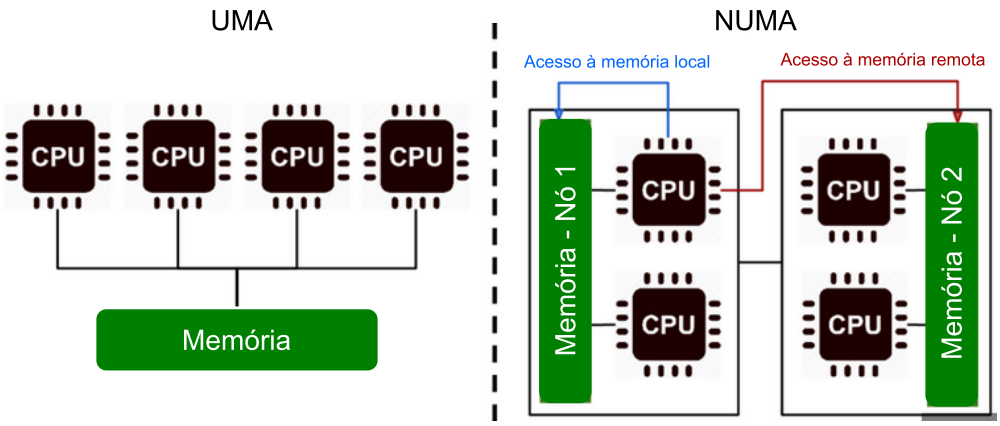

Figura 1.5. UMA vs. NUMA.

Um dos modelos de programação mais comuns para computadores paralelos de memória compartilhada é o modelo de programação paralela com threads. Nesse modelo, cada thread de execução é atribuída a um núcleo de processamento e pode acessar diretamente o espaço de memória compartilhada. Isso permite que as threads cooperem entre si, trocando informações e coordenando suas atividades por meio da memória compartilhada.

OpenMP é uma API (*Application Programming Interface*) de programação paralela que é amplamente utilizada para aproveitar o paralelismo de threads em arquiteturas de memória compartilhada [Chandra et al. 2001]. A interface é definida pela coleção de diretivas de compilador (`#pragmas`), funções de biblioteca, e e variáveis de ambiente. Esses elementos permitem que os usuários indicar a um compilador as partes de um programa sequencial que podem ser executado em paralelo.

Na Figura 1.6 apresenta-se um código-fonte C usando OpenMP para a soma de vetores utilizando múltiplas threads. Este código realiza a soma de dois vetores a e b e armazena o resultado no vetor c. Ele utiliza a diretiva `#pragma omp parallel for` para distribuir o laço for em paralelo entre as threads disponíveis. Para isso, o programa começa a execução com uma única thread, chamada de thread mestre. Quando a primeira região paralela, definida pela diretiva `parallel`, é encontrada, a thread mestre cria um time de threads que executam uma parte das somas dos vetores em diferentes núcleos ou processadores.

Além do OpenMP, existem várias outras APIs e bibliotecas que podem ser utilizadas para programação multithread em diferentes plataformas e linguagens de programação. Algumas das mais comuns incluem *Pthreads*, *Threading Building Blocks* (TBB), *Cilk*, entre outras.

```c
#include <stdio.h>
#include <omp.h>

#define N 1000000

int main() {
    int i;
    double a[N], b[N], c[N];

    // Inicializando os vetores
    for (i = 0; i < N; i++) {
        a[i] = i; b[i] = i * 2;
    }

    // Paralelizando a soma dos vetores
    #pragma omp parallel for shared(a, b, c) private(i)
    for (i = 0; i < N; i++)
    {
        c[i] = a[i] + b[i];
    }

    return 0;
}
```

Figura 1.6. OpenMP: Soma de vetores.

1.2.3. Memória distribuída: Clusters

Um cluster de computadores é uma arquitetura de sistema distribuído composta por vários computadores interconectados e trabalhando juntos como se fossem uma única entidade de processamento. Geralmente, esses computadores individuais, chamados de nós, são conectados por meio de uma rede de alta velocidade, permitindo comunicação eficiente entre eles, como ilustrado na Figura 1.7. Note que os nós desses sistemas geralmente são sistemas de memória compartilhada com um ou mais processadores de vários núcleos.

Nesses sistemas, cada nó de processamento opera de forma autônoma e pode executar tarefas independentes, ou colaborar com outros nós para realizar tarefas mais complexas. A comunicação entre os nós é realizada por meio de troca de mensagens pela rede de comunicação, e os dados precisam ser explicitamente transferidos entre os nós quando necessário, considerando que cada nó acessa apenas a sua própria memória. Esses sistemas são frequentemente utilizados em ambientes de computação de alto desempenho e em aplicações distribuídas, onde a escalabilidade e a capacidade de processamento em paralelo são essenciais, podendo apresentar dezenas ou até milhares de nós.

Em relação às APIs de programação para clusters, geralmente utilizam-se implementações do padrão de *Message Passing Interface* (MPI). O MPI permite a execução de aplicações paralela tanto em sistemas de memória distribuída quanto em sistemas de memória compartilhada. Tipicamente, uma aplicação MPI consiste em vários processos que podem trocar dados enviando mensagens, seja utilizando primitivas de comunicação ponto a ponto ou primitivas de comunicação coletiva.

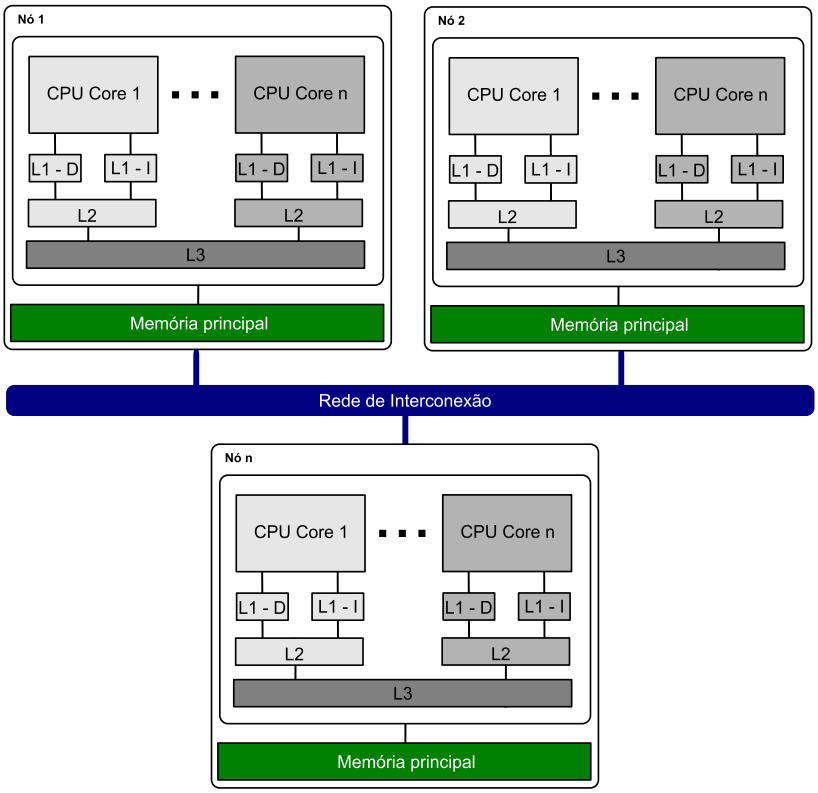

Figura 1.7. Cluster de computadores.

Na Figura 1.8 apresenta-se um trecho de código MPI (em linguagem C) usando as primitivas de comunicação ponto a ponto `Send` e `Recv` para o envio de dados. A operação Send é usada para enviar dados de um processo para outro. Ela inclui o endereço do buffer de dados a ser enviado, o tamanho dos dados a serem enviados, o tipo de dados a serem enviados e o identificador do processo de destino. O processo de destino deve estar pronto para receber a mensagem antes que a operação send seja concluída. Por outro lado, a operação Recv é usada para receber dados de um processo remoto. Ela especifica o endereço do buffer onde os dados recebidos serão armazenados, o tamanho máximo dos dados a serem recebidos, o tipo de dados a serem recebidos e o identificador do processo de origem [Silva et al. 2022].

Essas operações de comunicação são usadas em uma variedade de cenários em MPI. Por exemplo, em programas paralelos simples, um processo pode enviar dados para outro processo para coordenação ou troca de informações. No exemplo, temos dois processos (P0 e P1), sendo que o processo P0 envia um array de 100 elementos do tipo double para o processo P1.

```
//Processo enviando mensagem
double buffer[100];
int to=1, tag=123, error;
error=MPI_Send(buffer, 100, MPI_DOUBLE, to, tag, MPI_COMM_WORLD);


//Processo recebendo a mensagem
double buffer[100];
MPI_Status status;
int from=0, tag=123, error;
error=MPI_Recv(buffer, 100,MPI_DOUBLE, from, tag, MPI_COMM_WORLD, &status);
```

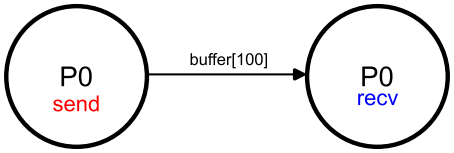

Figura 1.8. MPI: Troca de mensagens ponto a ponto.

1.2.4. GPGPUs

Inicialmente projetadas para renderizar gráficos complexos em jogos e aplicações de modelagem 3D, as GPUs modernas evoluíram para serem utilizadas também em tarefas de computação intensiva, como simulações científicas, aprendizado de máquina, análise de dados e muito mais.

A exploração do paralelismo em placas gráficas, conhecida como GPGPU (*General-Purpose computing on Graphics Processing Units*), é uma técnica que utiliza as unidades de processamento gráfico para realizar tarefas de propósito geral além do processamento gráfico tradicional. As GPUs são altamente paralelas e possuem muitos núcleos de processamento, o que as torna ideais para acelerar uma variedade de aplicações que podem se beneficiar do processamento em paralelo.

Por exemplo, as GPUs da NVIDIA são divididas em vários SMs (*Streaming Multiprocessors*), que por sua vez, executam grupos de threads, chamados de warps. Cada SM possui vários núcleos, chamados de CUDA cores. Cada CUDA core possui pipelines completos de operações aritméticas e de pontos flutuantes. A Figura 1.9 apresenta a arquitetura da GPU NVIDIA A100, equipada com 6.912 (64 por SM) núcleos CUDA memória dedicada de 40GB de memória. A GPU e sua memória associada geralmente são fisicamente separadas da CPU e de sua memória. Na documentação da NVIDIA, a CPU junto com sua memória associada é frequentemente chamada de *host*, e a GPU junto com sua memória é chamada de *device*.

As arquiteturas de computação paralela heterogênea CPU + GPU evoluíram porque a CPU e a GPU possuem atributos complementares que permitem que as aplicações tenham melhor desempenho usando os dois tipos de processadores. Portanto, para um desempenho ideal, é necessário usar CPU e GPU para a aplicação, executando as partes na CPU (sequenciais ou paralelas) e as partes paralelas de dados intensivos na GPU, conforme mostrado na Figura 1.10. Escrever código dessa forma garante que as carac-

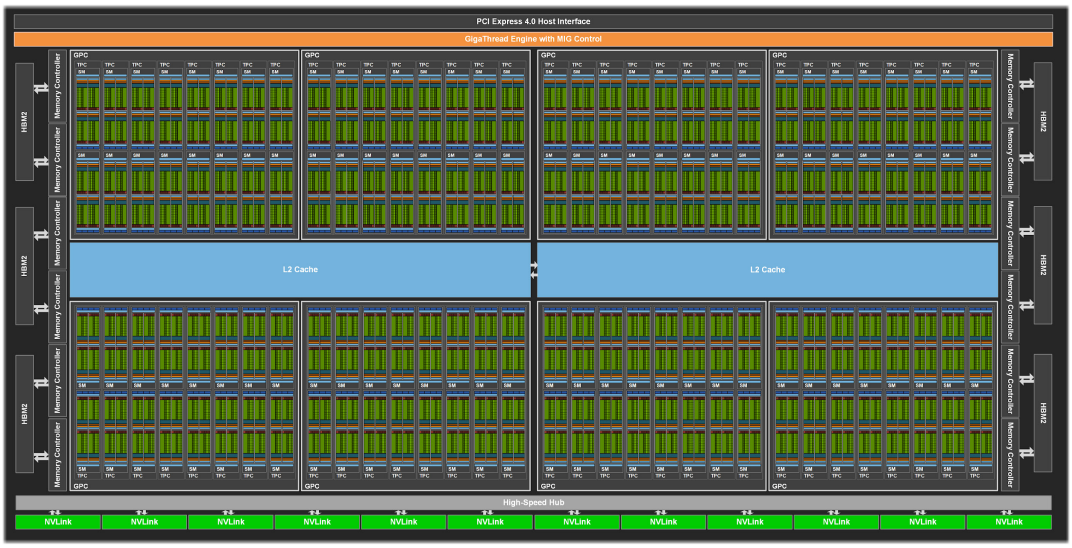

Figura 1.9. Arquitetura NVIDIA A100.

terísticas da GPU e da CPU se complementem, levando à utilização total do poder computacional do sistema combinado. Para suportar a execução conjunta de CPU + GPU de uma aplicação, a NVIDIA projetou um modelo de programação chamado CUDA [Cheng et al. 2014]. CUDA oferece um modelo de programação que permite aos desenvolvedores escreverem código para ser executado nas GPUs NVIDIA usando uma extensão da linguagem de programação C/C++.

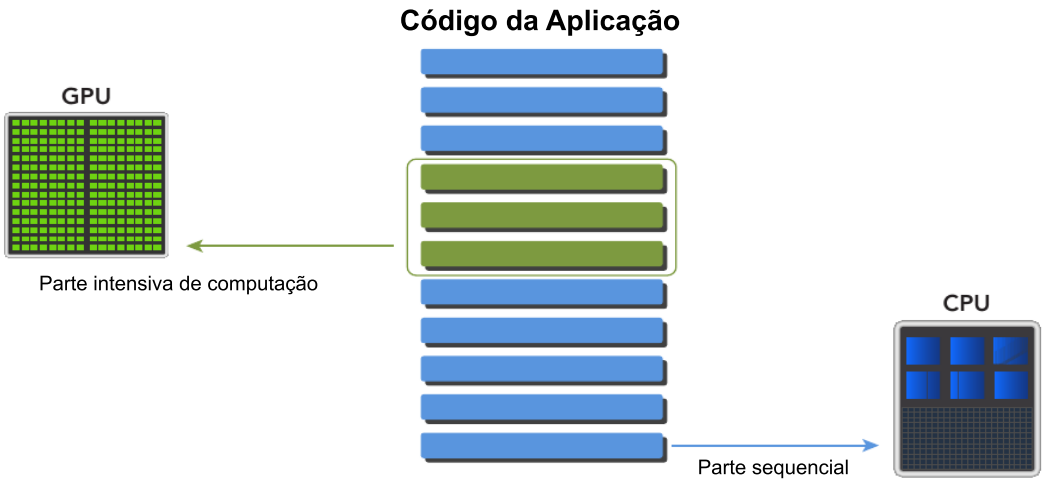

Figura 1.10. Computação heterogênea usando CPU + GPU. Adaptado de [Cheng et al. 2014]

Na Figura 1.11 compara-se um programa hello world em C e CUDA lado a lado. A principal diferença entre a implementação C e CUDA é o especificador __global__ e a sintaxe <<< ... >>>. O especificador __global__ indica uma função executada nna

GPU (device). Essa função pode ser chamada através do código host (a função *main()* no exemplo) e também é conhecida como *kernel*. Quando um kernel é chamado, sua configuração de execução é fornecida através da sintaxe $<<<$ ... $>>>$, por exemplo, *cuda_hello* $<<<1,1>>>()$.

C

```c
void c_hello(){
    printf("Hello World!\n");
}

int main() {
    c_hello();
    return 0;
}
```

CUDA

```c
__global__ void cuda_hello(){
    printf("Hello World from GPU!\n");
}

int main() {
    cuda_hello<<<1,1>>>();
    return 0;
}
```

Figura 1.11. Hello World em C e CUDA.

A curva de aprendizado do CUDA pode ser desafiadora devido à complexidade da programação de GPU e à necessidade de entender detalhes da arquitetura e hierarquia de memória. No entanto, há outras alternativas de mais alto nível, como o OpenACC [Silva et al. 2022]. OpenACC é uma API de programação paralela projetada para simplificar o desenvolvimento de aplicações para GPUs e outros aceleradores. O OpenACC permite adicionar diretivas (#pragmas) ao código existente para indicar onde e como paralelizar, muito semelhante ao OpenMP, como pode-se observar na Figura 1.12.

```c
#include <stdio.h>
#define N 1000000000

int main(void) {
   double pi = 0.0f; long i;
   #pragma acc parallel loop reduction(+: pi)
   for (i=0; i<N; i++)
   {
     double t=(double) ((i+0.5)/N);
     pi += 4.0/(1.0+t*t);
   }
   printf("pi=%f\n",pi/N);
   return 0;
}
```

Figura 1.12. cálculo de PI usando OpenACC.

Neste exemplo, calcula-se Este uma estimativa do valor de π (pi) usando o método da soma de Riemann. Destaca-se a linha "`#pragma acc parallel loop reduction(+: pi)`" que indica a paralelização do loop e a aplicação uma redução à variável pi. Isso significa que cada thread irá calcular sua própria soma parcial de pi e, no final, todas as somas parciais serão somadas para obter o valor final de pi. Note que com uma única linha adicional de código é possível paralelizar código para GPUs.

1.2.5. Cloud Computing

Ambientes de nuvem oferecem uma ampla variedade de recursos e serviços que podem ser utilizados para explorar o paralelismo e executar tarefas computacionais de forma escalável. Essas plataformas permitem que os desenvolvedores implantem e gerenciem sistemas distribuídos e paralelos na nuvem, aproveitando recursos de computação sob demanda para lidar com cargas de trabalho de tamanho e complexidade diversas.

Atualmente, os provedores de nuvem oferecem uma ampla gama de opções de máquinas virtuais e instâncias projetadas especificamente para atender às demandas de processamento de alto desempenho. Essas instâncias não apenas abrangem CPUs com múltiplos núcleos mas também podem incluir GPUs dedicadas. Além disso, a flexibilidade da computação em nuvem permite a criação de clusters sob demanda, onde diversas instâncias podem ser facilmente combinadas para formar um ambiente distribuído de processamento paralelo.

Para exemplificar, as instâncias P3[1] do Amazon EC2 são otimizadas para aplicações HPC e machine learning distribuído. Essas instâncias fornecem até 100 Gbps de taxa de transferência de redes, 96 vCPUs Intel Xeon, 8 GPUs NVIDIA V100 Tensor Core com 32 GiB de memória cada. Outro exemplo, é a ferramenta AWS ParallelCluster para a implantação e o gerenciamento de clusters de computação de alta performance na Amazon. O ParallelCluster[2] usa uma interface gráfica de usuário (GUI) simples ou um arquivo de texto para modelar e provisionar os recursos necessários de forma automatizada.

1.3. Considerações Finais

A exploração de paralelismo em arquiteturas modernas representa uma abordagem fundamental para lidar com os desafios crescentes de processamento de dados em um mundo cada vez mais digital e orientado por dados. A utilização de múltiplos níveis de paralelismo possibilita a realização de tarefas complexas de forma rápida e eficiente. No entanto, é importante reconhecer que a exploração do paralelismo também apresenta desafios, como a necessidade de desenvolver e otimizar software para aproveitar totalmente os recursos paralelos disponíveis. O contínuo avanço e aprimoramento das arquiteturas modernas garantem que o paralelismo continuará desempenhando um papel central na evolução da computação e no enfrentamento dos desafios computacionais do futuro.

Para saber mais sobre arquiteturas paralelas e programação paralela, recomenda-se [Pacheco and Malensek 2021, Wilkinson and Allen 2005, Silva et al. 2022] e demais referências citadas.

[1] `https://aws.amazon.com/pt/ec2/instance-types/p3/`
[2] `https://aws.amazon.com/pt/hpc/parallelcluster/`

Referências

[Chandra et al. 2001] Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Menon, R. (2001). *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Cheng et al. 2014] Cheng, J., Grossman, M., and McKercher, T. (2014). *Professional CUDA C Programming*. Wrox Press.

[Pacheco and Malensek 2021] Pacheco, P. and Malensek, M. (2021). *An introduction to parallel programming*. Morgan Kaufmann, Oxford, England, 2 edition.

[Silva et al. 2022] Silva, G. P., Bianchini, C. P., and Costa, E. B. (2022). *Programação Paralela e Distribuída: com MPI, OpenMP e OpenACC para computação de alto desempenho*. Casa do Código.

[Stallings 2015] Stallings, W. (2015). *Computer Organization and Architecture*. Pearson, Upper Saddle River, NJ, 10 edition.

[Wilkinson and Allen 2005] Wilkinson, B. and Allen, M. (2005). *Parallel programming - techniques and applications using networked workstations and parallel computers (2. ed.)*. Pearson Education.

Capítulo 2

Programação Paralela com OpenMP: Modelo de Tarefas

Calebe P. Bianchini, Gabriel P. Silva

Abstract

This mini-course aims to present parallel programming techniques using the OpenMP task parallelization model. The various directives and clauses available to implement this model will be covered, as well as the details that must be observed to ensure the correctness and best performance of the final code.

Resumo

Este minicurso tem como objetivo apresentar técnicas de programação paralela utilizando o modelo de paralelização de tarefas do OpenMP. Serão abordadas as diversas diretivas e cláusulas disponíveis para implementar esse modelo, além dos detalhes que devem ser observados para garantir a corretude e melhor desempenho do código final.

2.1. Introdução

O OpenMP [OpenMP ARB, 2015] é um padrão que define uma API (Interface de Programação de Aplicações) para a programação de múltiplas *threads* em computação paralela. Ele é usado principalmente em sistemas com múltiplos processadores ou núcleos de processamento com memória compartilhada para acelerar a execução de programas.

Os princípios fundamentais do OpenMP incluem: facilidade de uso, com emprego de diretivas de compilador; portabilidade, podendo ser compilado e executado em diferentes sistemas sem modificações significativas; uso de um modelo de programação baseado em *threads*; utilização de *pragmas* (linhas iniciadas por **#pragma**) inseridas no código-fonte como diretivas para informar ao compilador como o paralelismo deve ser aplicado; a criação, nas regiões paralelas, de múltiplas *threads* para executar o código paralelamente; a criação, normalmente, de uma equipe de *threads* trabalhadoras logo na primeira região paralela encontrada, como forma de otimizar o desempenho; uso de mecanismos para

controlar e sincronizar as *threads*, como diretivas para especificar regiões críticas (que devem ser executadas por apenas uma *thread* por vez) e para lidar com a exclusão mútua.

O paralelismo de laços [Silva et al., 2022] foi amplamente explorado nas primeiras versões do padrão OpenMP, sendo utilizado para acelerar significativamente a execução de laços que são computacionalmente intensivos. Essa técnica consiste em dividir as iterações de um laço em blocos de iterações com tamanho igual, onde cada bloco é executado por uma *thread* diferente. Para que o paralelismo de laços funcione corretamente, as iterações do laço não podem ter dependências entre si, ou seja, o resultado de uma iteração não pode depender do resultado de outra iteração.

Entretanto, ao lidar com problemas em que o fluxo de execução não é regular, a exploração do paralelismo torna-se mais complexa, mesmo quando há várias partes que podem ser executadas de forma independente. Nessas situações, é viável empregar o paralelismo de tarefas (*tasks*), que consiste na execução simultânea de diferentes tarefas independentes dentro do programa [Bianchini et al., 2019]. Essa abordagem permite uma melhor exploração do paralelismo e, consequentemente, melhora no desempenho [van der Pas et al., 2017].

O objetivo deste minicurso é apresentar o modelo de paralelismo de tarefas por meio de uma fundamentação teórica, mas também mostrando o seu impacto em exemplos práticos com o uso do OpenMP 4.5.

2.2. Tarefas

Este seção apresenta os principais conceitos relacionado ao modelo de programação de tarefas do OpenMP [OpenMP ARB, 2015, van der Pas et al., 2017].

2.2.1. Terminologia

As tarefas são unidades de trabalho cuja execução pode ser adiada ou realizada imediatamente. Elas são compostas pelo código a ser executado, um ambiente de dados (que é iniciado na sua criação) e variáveis de controle internas (IVCs em inglês – veja a Seção 2.12). O programador deve garantir que tarefas diferentes possam ser executadas simultaneamente, sem conflito no acesso aos dados compartilhados.

As tarefas são geradas quando uma *thread* encontra uma construção **task, taskloop, paralell, target** ou **teams** (ou qualquer construção combinada que especifique alguma dessas construções). Uma **região de tarefa** é uma região composta por todo o código encontrado durante a execução de uma tarefa, sendo que uma região paralela é composta por uma ou mais regiões de tarefas implícitas.

As tarefas podem receber diversas classificações, que serão utilizadas ao longo deste texto, tais como: **tarefa explícita**, que é gerada quando uma construção **task** é encontrada; uma **tarefa implícita** é gerada por uma região paralela implícita ou quando uma construção **parallel** é encontrada; para uma determinada *thread*, a **tarefa atual** correspondente à região de tarefa que ela está executando; uma tarefa é uma **tarefa filha** da sua região de tarefa geradora, sendo que uma região de tarefa filha não faz parte da região de tarefa geradora; as **tarefas irmãs** são tarefas filhas de uma mesma região de tarefa; uma **tarefa descendente** é a tarefa filha de uma região de tarefa ou de uma de suas

regiões de tarefas descendentes.

Uma **tarefa não vinculada** é uma tarefa que, quando sua região de tarefa é suspensa, pode ser retomada por qualquer *thread* na equipe. Ou seja, a tarefa não está vinculada a nenhuma *thread*. Uma **tarefa não postergada** é uma tarefa para a qual a execução não é adiada em relação à sua região de tarefa geradora. Ou seja, sua região de tarefa geradora é suspensa até que a execução da *tarefa não postergada* seja concluída. Uma **tarefa incluída** é uma tarefa para a qual a execução é sequencialmente incluída na região de tarefa geradora. Ou seja, uma tarefa incluída não é adiada e é executada imediatamente pela *thread* que a encontra. Uma **tarefa mesclada** é aquela na qual o ambiente de dados, incluindo as ICVs, é o mesmo que o de sua região de tarefa geradora. Tanto uma *tarefa não postergada* quanto uma *tarefa incluída* podem se tornar uma **tarefa mesclada**. Uma **tarefa final** força todas as suas tarefas filhas a se tornarem *tarefas finais e incluídas*.

As tarefas também podem ter uma relação de ordenação entre duas tarefas irmãs: a tarefa dependente e uma tarefa predecessora previamente gerada. A **dependência de tarefa** é cumprida quando a tarefa predecessora foi concluída. Em outras palavras, a **tarefa dependente** não pode ser executada até que suas **tarefas predecessoras** tenham sido concluídas.

Finalmente, devemos notar que no OpenMP, embora os termos *construct* e *directive* sejam conceitos relacionados e utilizados indistintamente ao longo deste texto, eles têm significados ligeiramente diferentes:

- **Directive (Diretiva)**: é uma instrução específica no código-fonte que é reconhecida pelo compilador OpenMP e que instrui as *threads* sobre como proceder em uma região paralela. As diretivas iniciam por **#pragma omp** no código fonte C/C++, seguidas por uma palavra-chave que define a ação a ser tomada (por exemplo, ‘parallel‘, ‘for‘, ‘sections‘, ‘task‘, entre outras). As diretivas podem ser seguidas por um bloco de código delimitado por chaves ‘{}‘ ou um comando específico.
- **Construct (Construção)**: Refere-se à combinação de uma diretiva OpenMP específica e o código delimitado por essa diretiva. Por exemplo, um **#pragma omp parallel** em C/C++ é uma diretiva que cria uma região paralela, e todo o bloco de código dentro desse ‘parallel‘ é chamado de ”construct” porque é a combinação da diretiva ‘parallel‘ com o código que está sendo executado em paralelo.

2.2.2. Modelo de Tarefas

Apesar de ser um conceito simples em sua essência, o uso das tarefas apresenta diversas nuances e particularidades que procuraremos mostrar a seguir.

No OpenMP, as construções como **sections**, **for** e **single** são utilizadas para o paralelismo de dados, enquanto a construção **task** é projetada para o paralelismo de tarefas, frequentemente lidando com padrões irregulares no acesso à memória. Para um melhor entendimento dessas construções utilizadas para paralelismo de dados, veja a referência [Silva et al., 2022].

A diretiva **task** cria tarefas independentes que podem ser executadas por qualquer *thread* disponível. Elas podem ser atribuídas dinamicamente às *threads,* permitindo uma

distribuição mais flexível do trabalho. Cada *thread* pode executar uma ou mais tarefas alocadas no *pool* de tarefas. O seu uso é adequado para dividir o trabalho em tarefas menores e independentes, sendo escalável para um grande número de tarefas, sem a necessidade de determinar antecipadamente a estrutura de trabalho.

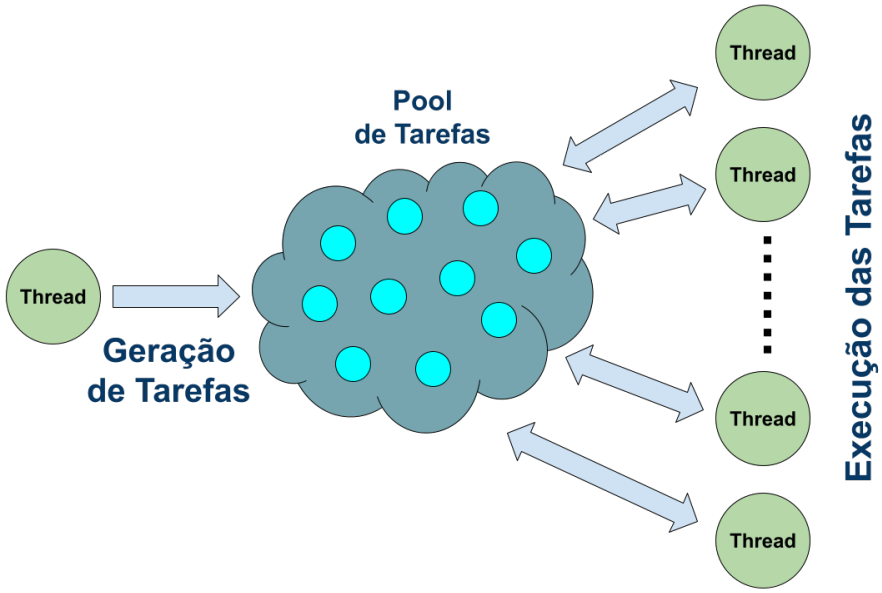

Figura 2.1. Modelo de Tarefas do OpenMP

O modelo de tarefas do OpenMP é representado na Figura 2.1, onde uma *thread* em execução gera as tarefas e as registra em um *pool* de tarefas. Estas, por sua vez, serão retiradas e executadas por uma equipe de *threads* até que não haja mais tarefas no *pool* de tarefas. O ambiente de execução OpenMP é quem decide sobre o escalonamento das tarefas, ou seja, sempre que uma *thread* alcança um ponto de escalonamento de tarefas (veja a Seção 2.2.3), o ambiente de execução pode realizar uma troca de tarefas, iniciando ou retomando a execução de uma tarefa diferente que esteja ligada à equipe de *threads* atual.

Uma vez que uma *thread* começa a executar uma tarefa, ela é designada para executar a tarefa até a sua conclusão, mesmo que possa suspender sua execução em um ponto de escalonamento para retomá-la mais tarde. Ou seja, a menos que uma cláusula **untied** (veja a Seção 2.3) seja utilizada na declaração da tarefa, a *thread* sempre estará vinculada à tarefa que iniciou.

As tarefas podem ser usadas para executar partes do trabalho em laços do tipo **while**; percorrer nós em uma lista encadeada ou em uma árvore de grafo; ou ainda em um laço normal (com uma construção **taskloop** – veja a Seção 2.6). Ao contrário do compartilhamento de trabalho, com escalonamento estático, das iterações de um laço (como no caso do **parallel for**), uma tarefa frequentemente é enfileirada e depois desenfileirada para execução por qualquer uma das *threads* de uma equipe dentro de uma região paralela. A geração de tarefas pode ser feita por um única *thread* geradora (criando tarefas irmãs) ou

por várias *threads* geradoras como no percurso recursivo de uma árvore de grafo.

2.2.3. Pontos de Escalonamento

Um ponto de escalonamento de tarefa é um ponto durante a execução da região de tarefa atual no qual ela pode ser suspensa para ser retomada mais tarde; ou o ponto de conclusão da tarefa, após o qual a *thread* em execução pode mudar para uma região de tarefa diferente. Quando uma *thread* encontra um ponto de escalonamento de tarefa, ela pode tomar uma das seguintes ações:

- iniciar a execução de uma *tarefa vinculada* ligada à equipe de *threads* atual;
- retomar qualquer região de tarefa suspensa, ligada à equipe atual e à qual a *thread* está vinculada;
- iniciar a execução de uma *tarefa não vinculada* ligada à equipe atual;
- retomar qualquer *região de tarefa não vinculada* suspensa, ligada à equipe atual.

Se mais de uma das opções acima estiverem disponíveis, o resultado é indefinido. Os pontos de escalonamento de tarefas dividem dinamicamente as regiões de tarefas em partes, onde cada parte é executada sem interrupções do início ao fim. As diferentes partes da mesma região de tarefas são executadas na ordem em que são encontradas. Na ausência de construções de sincronização de tarefas, a ordem na qual as *threads* executam as partes de diferentes tarefas escalonáveis é indeterminada. As tarefas são criadas explicitamente nos seguintes pontos de escalonamento:

i- Quando se encontra uma construção **task** → uma tarefa explícita é criada. Veja mais detalhes na Seção 2.3;

ii- Quando se encontra uma construção **taskloop** → tarefas explícitas para partes das iterações do laço são criadas. Essa diretiva oferece controles semelhantes aos encontrados na construção **task**. Veja mais detalhes na Seção 2.6;

iii- Quando se encontra uma construção **target** → uma tarefa *target*, para ser executada em outro dispositivo, é criada. Veja mais detalhes na Seção 2.11.

Além disso, os pontos de escalonamento podem ocorrer implicitamente nos seguintes locais:

- no ponto imediatamente seguinte à geração de uma tarefa explícita com a diretiva **task**;
- após o ponto de conclusão de uma região de tarefa, ao final da diretiva **task**;
- em uma região **taskyield**;
- em uma região **taskwait**;
- no final de uma região **taskgroup**;

- em uma região de barreira implícita ou explícita;
- o ponto imediatamente após a geração de uma região **target**;

2.2.4. Pontos de Sincronização

Em certos pontos do código, que são chamados de pontos de sincronização de tarefas, é garantido que todas as tarefas anteriores já terminaram. Os pontos de sincronização explícitos de tarefas, com uso de diretivas, podem ser:

1. **#pragma omp taskwait**: Esta diretiva é usada para aguardar explicitamente a conclusão de todas as tarefas associadas antes de continuar a execução do código. É uma forma de sincronização explícita para tarefas. Veja mais detalhes na Seção 2.8.
2. **#pragma omp barrier**: Embora seja mais comumente usado para sincronizar *threads* em uma região paralela, pode também ser utilizado para sincronizar tarefas. Este comando espera que todas as *threads* ou tarefas ativas naquele ponto terminem antes de prosseguir. Veja mais detalhes na Seção 2.7.
3. **#pragma omp taskgroup**: É usado para definir um grupo de tarefas, permitindo aguardar a conclusão de todas as tarefas em um determinado grupo. Todas as tarefas dentro do grupo devem ser concluídas antes que a execução do código prossiga após a diretiva **taskgroup**. Veja mais detalhes na Seção 2.9.

No OpenMP, embora haja pontos de sincronização explícitos para garantir a execução das tarefas, há também pontos implícitos nos quais é garantido que as tarefas anteriores já tenham sido executadas. Estes pontos de sincronização implícitos ocorrem, por exemplo, em todas as barreiras implícitas do OpenMP, como as que ocorrem ao final de uma região paralela ou de uma diretiva **single**.

2.3. Diretiva task

2.3.1. Sintaxe

A diretiva **task** é utilizada na criação de tarefas e possui diversas cláusulas que modificam o seu comportamento. A sua sintaxe é a seguinte:

#pragma omp task [cláusula[[,] cláusula] ...]
{bloco-estruturado}

Quando uma *thread* encontra uma construção **task**, uma tarefa é gerada a partir do código associado ao bloco estruturado correspondente. Por bloco estruturado entenda-se um trecho de código delimitado por chaves { } que define uma unidade lógica e coesa de operações, com uma única entrada no topo e uma única saída na parte inferior, sendo que o acesso ao bloco estruturado não pode ser resultado de um desvio e o ponto de saída não pode ser um desvio para fora do bloco estruturado.

A *thread* que encontra a diretiva **task** pode executar imediatamente a tarefa ou adiar sua execução. Neste último caso, qualquer *thread* na equipe pode executar a tarefa. A conclusão da tarefa pode ser garantida com uso de construções de sincronização de tarefas descritas anteriormente na Seção 2.2.4.

2.3.2. Exemplos

No Exemplo 2.1 observamos que a tarefa deve ser criada dentro de uma região paralela, mas com uso da diretiva **single**. Ou seja, apenas uma *thread* deve encontrar a diretiva **task**, caso contrário, múltiplas tarefas seriam criadas, em número igual ao total de *threads* da região paralela. Neste exemplo não há uma ordem pré determinada para a impressão das mensagens.

```
1  int main() {
2      #pragma omp parallel
3      {
4          #pragma omp single
5          {
6              #pragma omp task
7              {
8              printf("Esta é uma tarefa executada por uma thread
   ↪ paralela.\n");
9              }
10             printf("Esta é a execução fora da tarefa.\n");
11         }
12     }
13     return 0;
14 }
```

Exemplo 2.1. Forma de Utilização

```
1  typedef struct node node;
2  struct node {
3      int data;
4      node * next;
5  };
6  void process(node * p) {
7       /* o trabalho é feito aqui */
8  }
9  void increment_list_items(node * head) {
10      #pragma omp parallel
11      {
12          #pragma omp single
13          {
14              node * p = head;
15              while (p) {
16                   #pragma omp task
17                   // p é firstprivate por padrão
18                   process(p);
19                   p = p->next;
20              }
21          }
22      }
23 }
```

Exemplo 2.2. Lista Encadeada

O Exemplo 2.2 [OpenMP ARB, 2016] demonstra como usar a diretiva **task** para processar elementos de uma lista encadeada em paralelo. A *thread* que executa a região única (‘single‘) gera todas as tarefas explícitas, que são então executadas pelas *threads* na equipe atual. O ponteiro ’p’ é **firstprivate** por padrão na diretiva **task**, portanto não é necessário especificá-lo em uma cláusula **firstprivate**.

```
1  int fib(int n) {
2  int i, j;
3      if (n<2)
4          return n;
5      else {
6          #pragma omp task shared(i)
7          i=fib(n-1);
8          #pragma omp task shared(j)
9          j=fib(n-2);
10         #pragma omp taskwait
11         return i+j;
12 }
13 int main() {
14 int n = 20;
15     #pragma omp parallel
16     {
17         #pragma omp single
18         fib(n);
19     }
20 }
```

Exemplo 2.3. Fibonacci

No Exemplo 2.3 a função *fib()* deve ser chamada de dentro de uma região paralela para que as diferentes tarefas especificadas sejam executadas em paralelo. Contudo, apenas uma *thread* da região paralela deve chamar *fib()* a menos que se deseje múltiplos cálculos de Fibonacci concorrentes. A diretiva **taskwait** é obrigatória para a correta execução da função, assim como a declaração das variáveis ’i’ e ’j’ como compartilhadas. Esse código apresenta uma certa ineficiência por gerar tarefas para valores muito pequenos de ’n’. Uma possível solução é o uso das cláusulas **if** ou **final**, que são apresentadas na seção a seguir.

2.3.3. Cláusulas

Tabela 2.1. Cláusulas da diretiva task

if ([task :] expressao-escalar)	**private** (lista)
final (expressao-escalar)	**firstprivate** (lista)
untied	**shared** (lista)
default (shared | none)	**depend** (tipo-de-dependencia : lista)
mergeable	**priority** (valor-da-prioridade)

Várias cláusulas podem ser usadas para gerenciar e otimizar a geração de tarefas, assim como reduzir a sobrecarga de execução e realizar o balanceamento de carga. A Ta-

bela 2.1 apresenta as cláusulas que podem ser utilizadas com a diretiva **task**. As cláusulas **if** e **final** são usadas para controlar a geração e a execução das tarefas.

A cláusula **if** recebe uma expressão booleana que determina se a tarefa deve ser gerada ou não. Se a expressão for **verdadeira**, a tarefa é gerada e colocada no *pool* de tarefas para ser executada por uma das *threads* da equipe. Se a expressão for **falsa**, uma *tarefa não postergável* é gerada e a *thread* que a encontra deve executá-la sequencialmente e apenas ao final retomar a execução da região de tarefa atual.

Se a expressão da cláusula **final** é avaliada como **verdadeira**, a tarefa gerada será uma *tarefa final e incluída*. Neste caso, todas as tarefas que ela gerar serão executadas sequencialmente pela *thread* que a encontrou, sem serem colocadas no *pool*. Todas as construções **task** encontradas durante a execução de uma *tarefa final* gerarão *tarefas finais e incluídas*. Se a tarefa não for final, as tarefas que ela gerar serão tratadas normalmente.

No máximo uma cláusula **if** ou **final** pode aparecer na diretiva **task**. A forma de uso das cláusulas **if** e **final** pode ser vista no Exemplo 2.4. Nesse trecho de código, a função *foo (n)* é criada como uma tarefa se 'n' for maior que 4. Se 'n' for menor que 2, a tarefa é final e todas as tarefas geradas serão executadas sequencialmente. Caso contrário, as tarefas geradas serão colocadas no *pool* para serem executadas por outras *threads*.

```
1 #pragma omp parallel
2 {
3   #pragma omp single
4   {
5     #pragma omp task if (n > 4) final (n < 2)
6     foo (n);
7   }
8 }
```

Exemplo 2.4. Cláusulas if e final

O Exemplo 2.5 ilustra a diferença entre as cláusulas **if** e **final**. A cláusula **if** tem um efeito local. No primeiro conjunto de tarefas, aquela que possui a cláusula **if** será não postergável, mas a tarefa aninhada dentro dessa tarefa não será afetada pela cláusula **if** e será criada normalmente. Por outro lado, a cláusula **final** afeta todas as construções **task** na *região de tarefa final* mas não a própria *tarefa final*. No segundo conjunto de tarefas, as tarefas aninhadas serão criadas como *tarefas incluídas*. Note também que as condições para as cláusulas **if** e **final** são normalmente opostas.

A cláusula **final** pode ser utilizada para otimizar a execução do exemplo do cálculo da série de Fibonacci, evitando a geração de tarefas, quando o valor de 'n' for muito pequeno, como mostrado no Exemplo 2.6. Meça o tempo de execução das duas versões e verifique a diferença.

A expressão da cláusula **if** e a expressão da cláusula **final** são avaliadas no contexto fora da construção **task**, e não é especificada nenhuma ordem para essas avaliações.

```
1  void bar(void);
2
3  void foo () {
4  int i;
5      #pragma omp task if(0) // Esta tarefa é não postergável
6      {
7          #pragma omp task // Esta tarefa é uma tarefa regular
8          for (i = 0; i < 3; i++) {
9               #pragma omp task // Esta tarefa é uma tarefa regular
10              bar();
11         }
12     }
13     #pragma omp task final(1) // Esta tarefa é uma tarefa regular
14     {
15         #pragma omp task // Esta tarefa é incluída
16         for (i = 0; i < 3; i++) {
17              #pragma omp task // Esta tarefa também é incluída
18              bar();
19         }
20     }
21 }
```

Exemplo 2.5. Cláusulas if e final

```
1  int fib(int n) {
2  int i, j;
3      if (n<2)
4          return n;
5      else {
6          #pragma omp task shared(i) final (n<10)
7          i=fib(n-1);
8          #pragma omp task shared(j) final (n<10)
9          j=fib(n-2);
10         #pragma omp taskwait
11         return i+j;
12 }
```

Exemplo 2.6. Fibonacci otimizado

Uma *thread* que encontra um ponto de escalonamento de tarefas dentro da região de tarefa pode suspender temporariamente a execução dessa região de tarefa. Por padrão, uma tarefa é vinculada e sua região de tarefa suspensa só pode ser retomada pela *thread* que iniciou sua execução. Contudo, se a cláusula **untied** estiver presente em uma construção **task**, qualquer *thread* na equipe pode retomar a execução da região de tarefa após uma suspensão. A cláusula **untied** é ignorada se uma cláusula **final** estiver presente na mesma construção **task** e a expressão da cláusula **final** for avaliada como verdadeira, ou se a tarefa for uma *tarefa incluída*.

Reforçando o que já dissemos, a construção **task** inclui um ponto de escalonamento de tarefas imediatamente após a geração da tarefa explícita e outro também no seu ponto de conclusão.

```
1  #define LARGE_NUMBER 10000000
2  double item[LARGE_NUMBER];
3  extern void process(double);
4
5  int main() {
6      #pragma omp parallel
7      {
8          #pragma omp single
9          {
10             int i;
11             #pragma omp task untied
12             // i é firstprivate, item é shared
13             {
14                  for (i=0; i<LARGE_NUMBER; i++)
15                       #pragma omp task
16                       process(item[i]);
17                  }
18             }
19     }
20     return 0;
21 }
```

Exemplo 2.7. Cláusula untied

No Exemplo 2.7, as tarefas são geradas em um laço executado por uma *tarefa não vinculada* ("untied task"). Durante essa geração, o limite de tarefas não atribuídas do ambiente de execução pode ser atingido. Nesse caso, a *thread* responsável pelo laço de geração de tarefas, ao alcançar o ponto de escalonamento de tarefas na diretiva **task**, pode suspender a tarefa atual e iniciar a execução de uma das tarefas ainda não atribuídas.

Se as outras *threads* finalizarem as demais tarefas antes que a *thread* responsável pelo laço termine a execução da sua tarefa, qualquer outra *thread* está qualificada para retomar a execução do laço de geração de tarefas, já ele está sendo executado por uma *tarefa não vinculada*.

Sem o uso desta cláusula, as outras *threads* seriam forçadas a ficar inativas até que a *thread* geradora concluísse sua tarefa mais demorada, já que o laço de geração de tarefas estaria sendo executado por uma *tarefa vinculada*.

Quando a cláusula **mergeable** está presente em uma construção **task**, a tarefa gerada é uma *tarefa mesclável*, ou seja, uma tarefa para a qual o ambiente de dados, incluindo as variáveis de controle internas (veja a Seção 2.12), é o mesmo que o de sua região de tarefa geradora. Isso significa que o compilador pode decidir unir duas ou mais tarefas em uma única tarefa se elas forem executadas na mesma *thread*, com o objetivo de melhorar a eficiência e reduzir a sobrecarga de execução de tarefas pequenas.

Ao adicionar a cláusula **mergeable** a uma diretiva **task**, você permite que o ambiente de execução faça essa otimização, caso julgue apropriado. Isso pode economizar os custos associados à criação, escalonamento e execução de tarefas individuais separadas, como em situações onde você tem muitas tarefas pequenas ou quando a fusão dessas tarefas pode resultar em uma redução da sobrecarga de gerenciamento de tarefas.

```
1  #include <stdio.h>
2  void foo ( ) {
3      int x = 2;
4      #pragma omp task shared(x) mergeable
5      {
6          x++;
7      }
8      #pragma omp taskwait
9      printf("%d\n",x); // imprime 3
10 }
```

Exemplo 2.8. Cláusula mergeable

No Exemplo 2.8 o uso da cláusula **mergeable** é seguro. Como 'x' é uma variável compartilhada, o resultado não depende se a tarefa é mesclada ou não (ou seja, a tarefa sempre incrementará a mesma variável e sempre calculará o mesmo valor para 'x').

```
1  #include <stdio.h>
2  void foo () {
3  int x = 2;
4      #pragma omp task mergeable
5      {
6      x++;
7      }
8      #pragma omp taskwait
9      printf("%d\n",x); // imprime 2 ou 3
10 }
```

Exemplo 2.9. Cláusula mergeable

O Exemplo 2.9 [OpenMP ARB, 2016] demonstra um uso incorreto da cláusula **mergeable**. Neste exemplo, a tarefa criada acessará diferentes instâncias da variável 'x' se a tarefa não for mesclada, já que 'x' é **firstprivate**. Porém, acessará a mesma variável 'x' se a tarefa for mesclada. Como resultado, o comportamento do programa é indefinido e ele pode imprimir dois valores diferentes para 'x' dependendo das decisões tomadas pelo ambiente de execução.

A cláusula **priority** é uma sugestão para o escalonador de prioridade da tarefa gerada, através de um valor numérico não negativo. As tarefas com valor numérico mais alto tem recomendação para serem executadas antes das tarefas de prioridade mais baixa. A prioridade padrão, quando nenhuma cláusula **priority** é especificada, é zero (a menor prioridade).

Se um valor for especificado na cláusula **priority** for maior do que o valor máximo definido (ICV *max-task-priority-var*), então o ambiente de execução usará o valor dessa ICV. Um programa que dependa da ordem de execução das tarefas determinada por esse valor de prioridade pode ter um comportamento indeterminado, já que a prioridade é apenas uma sugestão e não é obrigatória.

```
1  void compute_array (float *node, int M);
2  void compute_matrix (float *array, int N, int M) {
3      int i;
4      #pragma omp parallel private(i)
5           #pragma omp single
6           {
7                for (i=0;i<N; i++) {
8                     #pragma omp task priority(i)
9                     compute_array(&array[i*M], M);
10               }
11          }
12 }
```

Exemplo 2.10. Cláusula priority

No Exemplo 2.10 calculamos *arrays* em uma matriz por meio de uma rotina 'compute_array'. Cada tarefa tem um valor de prioridade igual ao valor da variável de laço 'i' no momento de sua criação. Uma prioridade maior em uma tarefa significa que ela é candidata a ser executada mais cedo. A criação de tarefas ocorre em ordem crescente (de acordo com o espaço de iteração do laço), mas uma sugestão, por meio da cláusula **priority**, é fornecida para reverter a ordem de execução.

No máximo uma cláusula **priority** pode aparecer na diretiva **task**. Um programa que se desvia para dentro ou para fora de uma região de tarefa viola as especificações. Um programa não deve depender de qualquer ordenação das avaliações das cláusulas da diretiva **task**, nem de quaisquer efeitos colaterais das avaliações das cláusulas.

As demais cláusulas **default**, **private**, **firstprivate** e **shared**, são utilizadas para determinar o escopo das variáveis e têm o seu comportamento idêntico ao utilizado nas demais diretivas do OpenMP (veja mais detalhes em [Silva et al., 2022]).

A execução ordenada das tarefas pode ser realizada especificando as dependências com uma cláusula **depend**. Vamos estudar a cláusula **depend** com mais detalhes na Seção 2.5.

2.4. Escopo das variáveis

No OpenMP, a diretiva **task** é usada para criar tarefas independentes que podem ser aninhadas, criando uma hierarquia de tarefas. Mas uma complicação adicional que surge é determinar o escopo das variáveis utilizadas nesse conjunto de tarefas, que podem ter o seguinte comportamento:

1. Variáveis privadas: Normalmente, cada tarefa possui cópias privadas de suas variáveis locais. Ou seja, as variáveis declaradas dentro de uma tarefa não são acessíveis por outras tarefas.

2. Variáveis compartilhadas: Variáveis definidas fora do escopo das tarefas, como variáveis globais ou variáveis declaradas antes da região de tarefa, são compartilhadas por todas as tarefas criadas. Isso significa que todas as tarefas podem acessar e modificar essas variáveis.

3. Cláusulas de compartilhamento de dados: O OpenMP oferece cláusulas como **shared**, **private**, **firstprivate**, **lastprivate**, entre outras, que permitem controlar o escopo e o compartilhamento de variáveis entre as tarefas. Por exemplo, a cláusula **private** pode ser usada para garantir que cada tarefa tenha sua própria cópia privada de uma variável, enquanto a cláusula **shared** permite que todas as tarefas acessem e compartilhem uma variável comum.

4. Escopo das variáveis dentro de tarefas aninhadas: Se uma tarefa cria outras tarefas (tarefas aninhadas), as variáveis do escopo da tarefa pai são compartilhadas com as tarefas filhas, a menos que sejam explicitamente definidas como privadas.

Mas não é só isso. Os atributos de compartilhamento de dados de variáveis referenciadas em uma construção OpenMP podem ser *predeterminados*, *explicitamente determinados* ou *implicitamente determinados*, de acordo com as regras descritas a seguir.

Certas variáveis e objetos, relativos às tarefas, possuem atributos de compartilhamento de dados *predeterminados* da seguinte forma:

- Variáveis presentes em diretivas **threadprivate** são *threadprivate*.

```
1          int A[SIZE];
2          #pragma omp threadprivate(A)
3          // ...
4           #pragma omp task
5           {
6                 // A: threadprivate
7           }
8
```

- Variáveis com duração de armazenamento automático declaradas em um escopo dentro da construção são privadas.

```
1         #pragma omp task
2         {
3             int x = MN;
4             // Escopo de x: privado
5          }
6
```

- Objetos com duração de armazenamento dinâmico (malloc, new, etc.) são compartilhados.

```
1         int *p;
2         p = malloc(sizeof(float)*SIZE);
3         #pragma omp task
4         {
5              // *p: compartilhado
6         }
7
```

- Membros de dados estáticos são compartilhados.

```
1         class foo {
2               static int s = MN;
3           };
4         #pragma omp task
5         {
6               foo_A.s; // s@foo: compartilhado
7         }
8
```

- Variáveis com duração de armazenamento estático declaradas em um escopo dentro da construção são compartilhadas.

```
1         #pragma omp task
2         {
3               static int y;
4               // Escopo de y: compartilhado
5         }
6
```

- A(s) variável(eis) de iteração do laço nos laços for associados a uma construção **for**, **parallel for**, **taskloop** ou **distribute** é(são) privada(s).

Note que as variáveis com atributos de compartilhamento de dados *predeterminados* não podem ser listadas em cláusulas de atributos de compartilhamento de dados, exceto nos casos listados abaixo, o que substitui os atributos de compartilhamento de dados predeterminados da variável. Listamos aqui apenas o caso que se aplica às tarefas.

- A(s) variável(eis) de iteração do laço nos laços **for** associados a uma construção **for**, **parallel for**, **taskloop** ou **distribute** podem ser listadas em uma cláusula **private** ou **lastprivate**.

As variáveis com atributos de compartilhamento de dados *explicitamente determinados* são aquelas que são referenciadas em uma determinada construção e que estão listadas em uma cláusula de atributo de compartilhamento de dados (**private**, **shared**, **firstprivate**, etc.) naquela construção.

As variáveis com atributos de compartilhamento de dados *implicitamente determinados* são aquelas referenciadas dentro de uma construção específica, mas que não possuem atributos de compartilhamento de dados *predefinidos* e não estão *explicitamente* listadas em uma cláusula que defina tais atributos naquela construção. As suas regras são as seguintes:

- Em uma construção **parallel**, **teams** ou de geração de tarefas, os atributos de compartilhamento de dados dessas variáveis são determinados pela cláusula **default**. Se nenhuma cláusula **default** estiver presente, essas variáveis são compartilhadas.

- Para construções que não sejam construções de geração de tarefas ou construções **target**, se nenhuma cláusula **default** estiver presente, essas variáveis referenciam as variáveis com os mesmos nomes que existem no contexto envolvente.
- Em uma construção **target**, variáveis que não são mapeadas após a aplicação das regras de atributo de mapeamento de dados são **firstprivate**.
- Em uma construção de geração de tarefas órfãs, se nenhuma cláusula **default** estiver presente, os argumentos formais passados por referência são tratados como **firstprivate**.
- Em uma construção de geração de tarefas, se nenhuma cláusula **default** estiver presente, uma variável para a qual o atributo de compartilhamento de dados não é determinado pelas regras mencionadas anteriormente e que, no contexto envolvente, é determinada como compartilhada por todas as tarefas implícitas vinculadas à equipe atual, é compartilhada.
- Em uma construção de geração de tarefas, se nenhuma cláusula **default** estiver presente, uma variável para a qual o atributo de compartilhamento de dados não é determinado pelas regras mencionadas anteriormente é tratada como **firstprivate**.
- Um item da lista que aparece em uma cláusula **reduction** de uma construção de compartilhamento de trabalho não deve aparecer em uma cláusula **firstprivate** em uma construção de tarefa que seja encontrada durante a execução de qualquer uma das regiões de compartilhamento de trabalho resultantes.

2.5. Cláusula depend

A cláusula **depend** impõe restrições adicionais no escalonamento de tarefas ou iterações de laços. Essas restrições estabelecem dependências apenas entre tarefas irmãs ou entre as iterações de um laço. A sua sintaxe é a seguinte:

depend(tipo-de-dependencia : lista)

onde *tipo-de-dependencia* é um dos seguintes[1]: *in*, *out* ou *inout*.

Para o tipo de dependência *in*, se o local de armazenamento de pelo menos um dos itens da lista for o mesmo que o local de armazenamento de um item da lista de dependência *out* ou *inout* de uma construção de tarefa a partir da qual uma tarefa irmã foi gerada anteriormente, então a tarefa gerada será uma tarefa dependente dessa tarefa irmã.

Para os tipos de dependência *out* e *inout*, se o local de armazenamento de pelo menos um dos itens da lista for o mesmo que o local de armazenamento de um item da lista de dependência *in*, *out* ou *inout* de uma construção de tarefa a partir da qual uma tarefa irmã foi gerada anteriormente, então a tarefa gerada será uma tarefa dependente dessa tarefa irmã. Algumas das restrições da cláusula **depend** são:

- Itens da lista usados em cláusulas **depend** da mesma tarefa ou tarefas irmãs devem indicar locais de armazenamento idênticos ou locais de armazenamento disjuntos.

[1]Nota: Por simplificação não abordamos neste texto as dependências do tipo *source* ou *sink*.

- Itens da lista usados em cláusulas **depend** não podem ser seções de array de comprimento zero.
- Uma variável que faz parte de outra variável (como um elemento de uma estrutura), mas que não é um elemento de *array* ou uma seção de *array*, não pode aparecer em uma cláusula **depend**.

```
1  #include <stdio.h>
2  int main()  {
3      int x = 1;
4      #pragma omp parallel
5          #pragma omp single
6          {
7              #pragma omp task shared(x) depend(out: x)
8              x = 2;
9              #pragma omp task shared(x) depend(in: x)
10             printf("x = %d\n", x);
11         }
12     return 0;
13 }
```

Exemplo 2.11. Dependencia entre tarefas

2.6. Diretiva taskloop

A diretiva **taskloop** especifica que as iterações de um ou mais laços associados serão executadas em paralelo usando tarefas do OpenMP. As iterações são distribuídas entre as tarefas criadas pela construção e escalonadas para execução.

#pragma omp taskloop [cláusula[[,] cláusula] ...] new-line
for-loops

A diretiva **taskloop** impõe restrições na estrutura de todos os *for-loops* associados, que devem ter uma forma de laço canônica. A ordem de criação das tarefas do laço é indefinida, logo os programas que dependam de qualquer ordem de execução das iterações lógicas do laço violam as especificações.

Tabela 2.2. Cláusulas da diretiva taskloop

if ([taskloop :] expressao-escalar)	**private** (lista)
final (expressao-escalar)	**firstprivate** (lista)
untied	**shared** (lista)
default (shared | none)	**depend** (tipo-de-dependencia : lista)
mergeable	**priority** (valor-da-prioridade)
lastprivate(lista)	**collapse**(n)
grainsize(grain-size)	**num_tasks**(num-tasks)
nogroup	

O parâmetro da cláusula **grainsize** deve ser uma expressão de número inteiro positivo, sendo que o número de iterações lógicas do laço atribuídas a cada tarefa criada é

maior ou igual ao mínimo entre o valor da expressão de *grain-size* e o número de iterações lógicas do laço, porém menor que **duas vezes** o valor da expressão de *grain-size*.

Se a cláusula **num_tasks** for especificada, a construção **taskloop** criará tantas tarefas quanto o mínimo entre a expressão *num-tasks* e o número de iterações lógicas do laço. Cada tarefa deve ter pelo menos uma iteração lógica do laço. O parâmetro da cláusula **num_tasks** deve ter como resultado um número inteiro positivo.

Se nenhuma cláusula **grainsize** ou **num_tasks** estiver presente, o número de tarefas de laço criadas e o número de iterações lógicas do laço atribuídas a essas tarefas são definidos pela implementação.

A cláusula **collapse** pode ser usada para especificar quantos laços estão associados à construção **taskloop**. O parâmetro da cláusula **collapse** deve ser uma expressão que resulte em um número inteiro positivo constante. Se nenhuma cláusula **collapse** estiver presente, o único laço associado à construção **taskloop** é aquele que imediatamente segue a diretiva **taskloop**. Se mais de um laço estiver associado à construção **taskloop**, então as iterações de todos os laços associados são colapsadas em um espaço de iteração maior que é então dividido de acordo com as cláusulas **grainsize** e **num_tasks**. A execução sequencial das iterações em todos os laços associados determina a ordem das iterações no espaço de iteração colapsado.

O contador de iteração para cada laço associado é calculado antes da entrada para o laço mais externo. Se a execução de qualquer laço associado alterar quaisquer dos valores usados para calcular qualquer um dos contadores de iteração, então o comportamento é indefinido.

As cláusulas **if**, **final**, **priority**, **untied**, **depend**, **mergeable** quando presentes em uma construção **taskloop** tem comportamento similar ao descrito para a construção **task** (veja a Seção 2.3).

As demais cláusulas **default**, **private**, **firstprivate**, **lastprivate**, **shared**, são utilizadas para determinar o escopo das variáveis e têm o seu comportamento idêntico ao utilizado nas demais diretivas do OpenMP (veja mais detalhes em [Silva et al., 2022]).

Por padrão, a construção **taskloop** executa como se estivesse contida em uma construção **taskgroup** sem nenhuma declaração ou diretiva fora da construção **taskloop**. Assim, a construção **taskloop** cria uma região **taskgroup** implícita. Se a cláusula **nogroup** estiver presente, nenhuma região de **taskgroup** implícita será criada. As restrições da construção **taskloop** são as seguintes:

- Um programa que entra ou sai de uma região **taskloop** está em desacordo com as especificações.
- Todos os laços associados à construção **taskloop** devem ser perfeitamente aninhados; ou seja, não deve haver código intermediário nem nenhuma diretiva OpenMP entre quaisquer dois laços.
- No máximo uma cláusula **grainsize** ou uma cláusula **num_tasks** pode aparecer em uma diretiva **taskloop**, que são mutuamente exclusivas e não podem aparecer na mesma diretiva **taskloop**.

```
1  void long_running_task(void);
2  void loop_body(int i, int j);
3  void parallel_work(void) {
4  int i, j;
5    #pragma omp taskgroup
6    {
7      #pragma omp task
8      long_running_task(); // pode executar em paralelo
9      #pragma omp taskloop private(j) grainsize(500) nogroup
10     for (i = 0; i < 10000; i++) { // pode executar em paralelo
11         for (j = 0; j < i; j++) {
12                 loop_body(i, j);
13         }
14     }
15   }
16 }
```

Exemplo 2.12. Diretiva taskloop

O Exemplo 2.12 [OpenMP ARB, 2016] mostra como executar uma tarefa de longa duração simultaneamente com tarefas criadas com uma diretiva **taskloop** para um laço com quantidades desequilibradas de trabalho para suas iterações. A cláusula **grainsize** especifica que cada tarefa deve executar pelo menos 500 iterações do laço. A cláusula **nogroup** remove o grupo de tarefas implícito da construção **taskloop**; a construção explícita **taskgroup** no exemplo garante que a função não seja encerrada antes que a tarefa de longa duração e os laços tenham concluído a sua execução.

2.7. Diretiva barrier

A construção **barrier** é uma diretiva independente que especifica uma barreira explícita no ponto em que a construção aparece. A sua sintaxe é a seguinte:

#pragma omp barrier new-line

Todas as *threads* da equipe que executam a região paralela vinculada devem executar a região de barreira e completar a execução de todas as tarefas explícitas vinculadas a esta região paralela antes que qualquer uma delas continue a sua execução além da barreira. A região de barreira inclui um ponto implícito de escalonamento de tarefa na região de tarefa atual. As seguintes restrições se aplicam à construção **barrier**:

- Cada região de barreira deve ser encontrada por todas as *threads* em uma equipe ou por nenhuma delas, a menos que o cancelamento tenha sido solicitado para a região paralela mais interna que a envolve.
- A sequência de regiões de compartilhamento de trabalho e regiões de barreira encontradas deve ser a mesma para todas as *threads* em uma equipe.
- As diretivas **flush**, **barrier**, **taskwait** e **taskyield** são diretivas independentes e não podem ser a subdeclaração imediata de uma instrução **if**.

2.8. Diretiva taskwait

A construção **taskwait** é uma diretiva independente que especifica uma espera pela conclusão das tarefas filhas (mas que não inclui suas descendentes) da tarefa atual, cuja sintaxe da construção é a seguinte:

#pragma omp taskwait

A região **taskwait** inclui um ponto implícito de escalonamento de tarefas na região da tarefa atual, que é suspensa até que todas as tarefas filhas que foram geradas antes da região **taskwait** completem sua execução. Embora pareça simples, alguns detalhes importantes devem ser observados quando do uso de tarefas, que o uso da diretiva **taskwait** pode ajudar.

```
1  struct node {
2  struct node *left;
3  struct node *right;
4  };
5  extern void process(struct node *);
6  void postorder_traverse( struct node *p ) {
7      if (p->left)
8          #pragma omp task // p é firstprivate por padrão
9          postorder_traverse(p->left);
10     if (p->right)
11         #pragma omp task // p é firstprivate por padrão
12         postorder_traverse(p->right);
13     #pragma omp taskwait
14     process(p);
15 }
```

Exemplo 2.13. Diretiva taskwait

O Exemplo 2.13 [OpenMP ARB, 2016] mostra como percorrer uma estrutura semelhante a uma árvore usando tarefas explícitas. Observe que a função **postorder_traverse** deve ser chamada de dentro de uma região paralela para que as tarefas especificadas sejam executadas em paralelo.

Espera-se que a travessia seja feita em pós-ordem, como no código sequencial. Entenda-se por pós-ordem como um tipo de travessia ou ordenação dos elementos em que os nós filhos ou subordinados são visitados (processados) antes de se visitar o próprio nó raiz. É uma abordagem para percorrer a estrutura de forma recursiva em que os nós inferiores são visitados sempre antes dos nós superiores.

Também observe que, sem uso da diretiva **taskwait**, as tarefas serão executadas sem uma ordem especificada. Neste exemplo, forçamos uma travessia em pós-ordem da árvore adicionando uma diretiva **taskwait**, que garante que os filhos esquerdo e direito foram executados antes de processarmos o nó atual.

2.9. Diretiva taskgroup

Um **conjunto taskgroup** é um conjunto de tarefas que são logicamente agrupadas por uma região **taskgroup**. A construção **taskgroup** especifica uma espera pela conclusão

das tarefas filhas da tarefa atual e *todas* as suas tarefas descendentes. A sua sintaxe é:

#pragma omp taskgroup new-line
bloco-estruturado

Quando uma *thread* encontra uma construção **taskgroup**, ela começa a executar a região. Todas as tarefas filhas geradas na região **taskgroup** e todas as suas descendentes que se vinculam à mesma região paralela que a região **taskgroup** fazem parte do conjunto de tarefas associado à região **taskgroup**.

Há um ponto implícito de escalonamento de tarefa no final da região **taskgroup**. A tarefa atual é suspensa no ponto de escalonamento de tarefa até que todas as tarefas no **conjunto taskgroup** concluam a sua execução.

```
1  int main() {
2     int i;
3     tree_type tree;
4     init_tree(tree);
5     #pragma omp parallel
6     #pragma omp single
7     {
8         #pragma omp task
9         start_background_work();
10        for (i = 0; i < max_steps; i++) {
11             #pragma omp taskgroup
12             {
13                 #pragma omp task
14                 compute_tree(tree);
15              } // espera a travessia da árvore neste passo
16        check_step();
17        }
18   } // apenas aqui espera-se que a tarefa em
19     // background esteja completada
20  print_results();
21  return 0;
22  }
```

Exemplo 2.14. Diretiva taskgroup

No Exemplo 2.14 [OpenMP ARB, 2016], as tarefas são agrupadas e sincronizadas usando a construção **taskgroup**. Inicialmente, uma tarefa (a tarefa que executa a chamada *start_background_work()* é criada na região paralela e, mais tarde, uma travessia de árvore paralela é iniciada (a tarefa que executa a raiz das chamadas recursivas *compute_tree()*). Ao sincronizar as tarefas no final de cada travessia da árvore, o uso da construção **taskgroup** garante que a tarefa em segundo plano anteriormente iniciada não participe da sincronização e permaneça livre para ser executada em paralelo. Isso é oposto ao comportamento da construção **taskwait**, que incluiria as tarefas em segundo plano na sincronização.

2.10. Diretiva taskyield

A diretiva **taskyield** especifica que a tarefa atual pode ser suspensa em favor da execução de alguma outra tarefa. A sua sintaxe é a seguinte:

#pragma omp taskyield new-line

A região **taskyield** inclui um ponto explícito de escalonamento de tarefa na região de tarefa atual.

O Exemplo 2.15 [OpenMP ARB, 2016] ilustra o uso da diretiva **taskyield**. As tarefas no exemplo calculam algo útil e, em seguida, realizam algumas computações que precisam ser feitas em uma região crítica. Ao utilizar **taskyield** quando uma tarefa não consegue obter acesso à região crítica, a implementação pode suspender a tarefa atual e escalonar alguma outra tarefa que possa realizar alguma operação útil.

```
1  void algumacoisa_util (void);
2  void algumacoisa_critica (void);
3  void foo (omp_lock_t * lock, int n)
4  {
5      int i;
6      for ( i = 0; i < n; i++ )
7      #pragma omp task
8      {
9          algumacoisa_util();
10         while ( !omp_test_lock(lock) ) {
11         #pragma omp taskyield
12         }
13     algumacoisa_critica();
14     omp_unset_lock(lock);
15     }
16 }
```

Exemplo 2.15. Diretiva taskyield

2.11. Diretiva target

O OpenMP pode direcionar a execução de certos trechos de código para aceleradores (como GPUs) ou outros dispositivos que estejam conectados ao computador hospedeiro. Isso pode ser realizado com o uso da diretiva **target**.

Uma tarefa **target** é uma *tarefa mesclável* que é gerada por uma construção **target**, **target enter data**, **target exit data** ou **target update**. A construção **target** consiste em uma diretiva de **target** e uma região de execução. A região *target* é executada no dispositivo padrão ou no dispositivo especificado na cláusula **device**, normalmente um acelerador.

Infelizmente não foi possível apresentar maiores detalhes dessas diretivas neste minicurso e apresentamos apenas o Exemplo 2.16 onde a diretiva **target** cria um ambiente de execução para a execução paralela do laço em um dispositivo de destino (como uma GPU), distribuindo as iterações do laço entre as *threads* da equipe.

```
1  int main() {
2      const int array_size = 100;
3      int array[array_size];
4      #pragma omp parallel for
5      for (int i = 0; i < array_size; ++i) {
6          array[i] = i;
7      }
8      #pragma omp target
9      #pragma omp teams distribute parallel for
10     for (int i = 0; i < array_size; ++i) {
11         printf("Thread %d: Valor %d do array.\n",\
12         omp_get_thread_num(), array[i]);
13     }
14     return 0;
15 }
```

Exemplo 2.16. Diretiva target

2.12. Variáveis Internas de Controle

As ICVs armazenam informações como o número de *threads* a serem utilizadas para futuras regiões paralelas, o escalonamento a ser utilizado para laços de compartilhamento de trabalho e se o paralelismo aninhado está habilitado ou não.

Elas recebem os valores iniciais do próprio ambiente de execução, seja a partir de variáveis de ambiente do OpenMP ou através de chamadas para rotinas da API do OpenMP. O programa só pode recuperar os valores dessas ICVs através das rotinas da API do OpenMP. Existe uma correspondência entre as ICVs e as variáveis de ambiente que pode ser vista na Tabela 2.3.

Tabela 2.3. ICV e Variáveis de Ambiente

ICV	Variável de Ambiente
dyn-var	OMP_DYNAMIC
nest-var	OMP_NESTED
nthreads-var	OMP_NUM_THREADS
run-sched-var	OMP_SCHEDULE
bind-var	OMP_PROC_BIND
stacksize-var	OMP_STACKSIZE
wait-policy-var	OMP_WAIT_POLICY
thread-limit-var	OMP_THREAD_LIMIT
max-active-levels-var	OMP_MAX_ACTIVE_LEVELS
place-partition-var	OMP_PLACES
cancel-var	OMP_CANCELLATION
default-device-var	OMP_DEFAULT_DEVICE
max-task-priority-var	OMP_MAX_TASK_PRIORITY

2.13. Estudos de caso - Quicksort

Uma exemplo prático da aplicação das técnicas de paralelismo de tarefas é no algoritmo de ordenação Quicksort. Ele pode ser descrito facilmente como uma operação recursiva da seguinte maneira [Kumar, 2002][van der Pas et al., 2017]:

1. Escolha um elemento do arranjo de dados, chamando-o de *pivô*.
2. Coloque o *pivô* em sua posição final e ordenado em relação aos demais elemento do arranjo, mantendo os menores elementos à esquerda dele, bem como os maiores elementos à direita. Essa fase é chamada de *particionamento*.
3. *Recursivamente*, faça os passos 1 e 2 para as partes esquerda e direita do arranjo.
4. A *recursão* termina quando não houver nenhum outro elemento a ser ordenado.

Uma forma simplificada de visualizar a recursão existente na execução do algoritmo Quicksort é por meio de uma árvore. Uma forma de visualizar esta árvore é apresentada na Figura 2.2: cada nível da árvore representa uma divisão do arranjo em uma chamada recursiva; em cada nível da árvore estão circulados os *pivôs* em cor vermelha; a altura da árvore está anotado com a letra h, sendo o *nó raiz* indicado por $h = 0$.

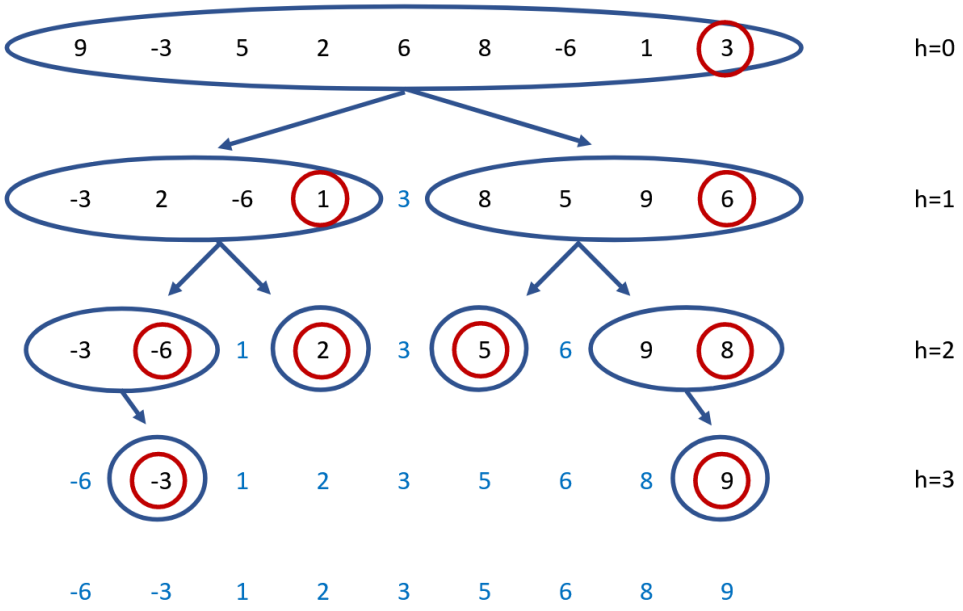

Figura 2.2. Árvore de recursão do Quicksort.

Sabe-se que a escolha do *pivô* é fundamental para o comportamento estável do Quicksort. Considerando uma distribuição ideal do arranjo, pode-se dizer que em cada chamada recursiva, o arranjo é dividido ao meio. Uma solução clássica do algoritmo recursivo é apresentada no Exemplo 2.17, onde cada divisão do arranjo é dividido em aproximadamente $(baixo + alto)/2$ elementos [van der Pas et al., 2017][Beukman, 2021].

```
1  // Função Quicksort simples
2  void quicksort(int arr[], int baixo, int alto) {
3    if (baixo < alto) {
4      // Particionar o array e obter o índice do pivô
5      int pi = particionar(arr, baixo, alto);
6
7      // Executar uma chamada recursiva para cada metade do array
8      quicksort(arr, baixo, pi - 1);
9      quicksort(arr, pi + 1, alto);
10   }
11 }
```

Exemplo 2.17. Código exemplo do algoritmo de ordenação Quicksort

Uma solução paralela trivial desse algoritmo seria anotar cada chamada recursiva do algoritmo com uma diretiva **omp task**. Assim, cada ramo da árvore de recursão seria executada por uma tarefa diferente, permitindo uma independência entre cada uma delas. O Exemplo 2.18 apresenta a estrutura dessa solução.

Veja que na função *main()* foi adicionada a região paralela (**omp parallel**) e, dentro dela, uma diretiva **single**. Ela garante que somente uma thread execute o bloc se código seguinte que, neste caso, é a primeira chamada para a função *quicksort_paralelo()*. Também foi usada uma cláusula **nowait** para que não haja barreira implícita nesta cláusula (veja mais detalhes sobre essa construção em [Silva et al., 2022].

A função *quicksort_paralelo()* praticamente mantém sua estrutura original - neste caso, optou-se por trocar o nome da função apenas para identificá-la mais facilmente. nas chamadas recursivas foi adicionada a diretiva **omp task**. É importante lembrar sobre o controle das variáveis utilizadas. Uma boa prática é alterar o comportamento padrão de **shared** para **none** e identificar as variáveis utilizadas individualmente. Neste exemplo, foram utilizadas as cláusulas **firstprivate** para as variáveis de controle de índice e **shared** para o compartilhamento do ponteiro do arranjo de dados.

Esta solução funciona bem, pois existe a criação das tarefas e suas execuções são independentes, e que cada uma delas ordena sua parte do arranjo. Porém, para um arranjo suficientemente grande, ela pode apresentar uma sobrecarga na criação de tarefas.

Observe, por exemplo, o nível $h = 2$ de altura da árvore na Figura 2.2. Nela, são criadas mais quatro tarefas devido as chamadas recursivas existente no algoritmo, além das tarefas que já foram criadas até o nível anterior (com $h = 1$ foram criadas duas tarefas, e com $h = 0$, foi criada uma tarefa). Podemos perceber que a quantidade de tarefas criadas e enfileiradas poderá ser suficientemente grande devido ao resultado de todas as chamadas recursivas, para todos os níveis da árvore gerada, derivada do número de elementos no arranjo. Por isso, esta solução pode sofrer uma degradação de desempenho pela sobrecarga de tarefas criadas e o seu gerenciamento.

A solução adotada para resolver o problema de sobrecarga é apresentado no Exemplo 2.19. Nele, é introduzido um valor denominado *CORTE*, que é resultado de uma análise experimental sobre o tamanho mínimo que um arranjo deve ter para que, a partir desse ponto, seja optado pela chamada recursiva da função sem a criação de novas tarefas.

```
1  // Função Quicksort paralela usando OpenMP
2  void quicksort_paralelo(int arr[], int baixo, int alto) {
3    if (baixo < alto) {
4      // Particionar o array e obter o índice do pivô
5      int pi = particionar(arr, baixo, alto);
6
7      // Executar a chamada recursiva para cada metade do array em
       ↪ paralelo
8      #pragma omp task default(none) shared (arr) firstprivate(baixo
       ↪ , pi)
9      quicksort_paralelo(arr, baixo, pi - 1);
10     #pragma omp task default(none) shared (arr) firstprivate(alto,
       ↪  pi)
11     quicksort_paralelo(arr, pi + 1, alto);
12   }
13 }
14
15 // Função principal
16 int main()
17 {
18   // Seed para a função rand
19   srand(time(NULL));
20
21   int tamanho = TAMANHO; // Tamanho do vetor
22
23   // Alocar memória dinamicamente para o vetor
24   int *arr = (int *)malloc(tamanho * sizeof(int));
25
26   // Preencher o vetor com valores aleatórios
27   for (int i = 0; i < tamanho; i++)   {
28     arr[i] = rand() % tamanho; // Gera valores aleatórios entre 0
       ↪ e tamanho
29   }
30
31   printf("Array original (%d): \n", tamanho);
32   imprimir_array(arr);
33
34   // Iniciar região paralela
35   #pragma omp parallel default(none) shared(arr, tamanho)
36   {
37     // Executar Quicksort paralelo
38     #pragma omp single nowait
39     quicksort_paralelo(arr, 0, tamanho - 1);
40   }
41
42   printf("\nArray ordenado (%d): \n", tamanho);
43   imprimir_array(arr);
44   if(validar_array_ordenado(arr, tamanho)) {
45     fprintf(stderr, "ERRO ao ordenar\n");
46   }
47   return 0;
48 }
```

Exemplo 2.18. Código exemplo trivial do algoritmo de ordenação Quicksort com *tasks*

```
1  // Função Quicksort paralela usando OpenMP
2  void quicksort_paralelo(int arr[], int baixo, int alto) {
3    if (baixo < alto) {
4      // Particionar o array e obter o índice do pivô
5      int pi = particionar(arr, baixo, alto);
6
7      // Executar a chamada recursiva para cada metade do array em
       ↪ paralelo
8      #pragma omp task final((pi - baixo + 1) < CORTE) mergeable
       ↪ default(none) shared(arr) firstprivate(baixo, pi)
9      quicksort_paralelo(arr, baixo, pi - 1);
10     #pragma omp task final((alto - pi + 1) < CORTE) mergeable
       ↪ default(none) shared(arr) firstprivate(alto, pi)
11     quicksort_paralelo(arr, pi + 1, alto);
12   }
13 }
```

Exemplo 2.19. Código exemplo com CORTE do algoritmo Quicksort com *tasks* paralelas

Nesta nova versão, ajustada para resolver o problema da sobrecarga, usamos a cláusula **final** nas criação das tarefas (revise o assunto na Seção 2.3.3) que avalia o tamanho do arranjo nas chamadas recursivas. Se esse tamanho for pequeno o suficiente ($< CORTE$), deseja-se não criar uma nova tarefas, mas forçar a *thread* vigente executar sequencialmente a chamada recursiva até finalizar toda a árvore de recursão desse ramo. Essa estratégia é utilizada nos dois ramos da recursão do algoritmo do Quicksort.

Por outro lado, se o tamanho do arranjo ainda for grande o suficiente para a criação de uma nova tarefa, permitimos que o OpenMP crie a tarefa e coloque-a no *pool* de execuções.

Por fim, além das considerações tradicionais sobre o algoritmo de Quicksort, como a escolha e localização do pivô, é importante analisar os fatores do paralelismo usando tarefas. Vale ressaltar, novamente, que um desses fatores é derivado da técnica de divisão-e-conquista aplicado no algoritmo clássico. Conforme a divisão acontece, novas tarefas são cridas e, por isso, pode acontecer uma sobrecarga no ambiente de execução.

Outro fator importante seria entender o comportamento de cada solução proposta em uma arquitetura, não só para compreender os fatores de sobrecarga, mas também analisar a escalabilidade da solução. Essa análise envolveria o aumento da quantidade de processadores (ou *cores*, como são chamados atualmente), mantendo o mesmo tamanho da entrada, como previsto na escalabilidade forte de Amdahl. Uma outra análise consideraria um aumento proporcional do tamanho da entrada comparado à quantidade de processadores utilizados, como discutido na escalabilidade fraca de Gustafson – veja mais detalhes em [Silva et al., 2022].

Um último fator importante é a busca por um valor limite de *CORTE*. Sendo ele descoberto de forma experimental, seria necessário realizar diversos experimentos em diferentes tipos de arquiteturas para encontrar um melhor valor.

Referências

[Beukman, 2021] Beukman, M. (2021). Parallel quicksort using openmp. Medium article at https://mcbeukman.medium.com/parallel-quicksort-using-openmp-9d18d7468cac.

[Bianchini et al., 2019] Bianchini, C. P., Vilabôas, F. G., and Castro, L. N. (2019). Paralelismo de tarefas utilizando openmp 4.5. In *Minicurso da ERAD/RS 2019*. https://setrem.com.br/erad2019/data/pdf/minicursos/mc06.pdf.

[Kumar, 2002] Kumar, V. (2002). *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

[OpenMP ARB, 2015] OpenMP ARB (2015). *OpenMP Application Programming Interface*. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[OpenMP ARB, 2016] OpenMP ARB (2016). *OpenMP Application Programming Interface Examples*. https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf.

[Silva et al., 2022] Silva, G., Bianchini, C., and Costa, E. B. (2022). *Programação Paralela e Distribuída*. Ed. Casa do Código. https://www.casadocodigo.com.br/pages/sumario-programacao-paralela.

[van der Pas et al., 2017] van der Pas, R., Stotzer, E., and Terboven, C. (2017). *Using OpenMP—The Next Step: Affinity, Accelerators, Tasking, and SIMD*. Scientific and Engineering Computation. MIT Press.

Capítulo

3

DinD-Bench: Impacto de Contêineres Docker em Docker para a Programação Paralela[1]

Claudio Schepke - claudioschepke@unipampa.edu.br[2]
Felipe Bedinotto Fava - felipefava.aluno@unipampa.edu.br[3]
Diego Luis Kreutz - diegokreutz@unipampa.edu.br[4]

[1]Este capítulo foi parcialmente financiado pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico – CNPq (Processo número 407827/2023-4), Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul - FAPERGS (Edital PqG 07/2021 - Projeto Nº 21/2551-0002055-5) e Programa Hackers do Bem da Rede Nacional de Ensino e Pesquisa - RNP (GT - Malware DataLab)

[2]Claudio Schepke possui graduação em Ciência da Computação pela Universidade Federal de Santa Maria (2005) e mestrado (2007) e doutorado (2012) em Computação pela Universidade Federal do Rio Grande do Sul, sendo este feito na modalidade sanduíche na Technische Universität Berlin, Alemanha (2010-2011). É professor associado da Universidade Federal do Pampa (UNIPAMPA), campus Alegrete/RS desde 2012. Tem experiência na área de Ciência da Computação, com ênfase em Processamento de Alto Desempenho, atuando principalmente nos seguintes temas: interfaces de programação paralela, aplicações científicas e computação em nuvem.

[3]Felipe Bedinotto Fava é graduado em Ciência da Computação pela Universidade Federal do Pampa (UNIPAMPA), Campus Alegrete (2024) e técnico em Informática para Internet pelo Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense (IFSUL), Campus Santana do Livramento (2017). Atualmente é mestrando do Programa de Pós-Graduação em Engenharia de Software (PPGES) da UNIPAMPA. Possui experiência profissional no ramo do desenvolvimento de aplicações para múltiplas plataformas com ênfase em dispositivos móveis. Apresenta afinidade e interesse nas áreas de Computação Ubíqua, Computação em Nuvem, Computação Paralela, Computação de Alto Desempenho e Arquitetura de Computadores.

[4]Diego Kreutz possui doutorado pela Universidade de Luxemburgo (Luxemburgo) e mestrado em Informática, mestrado em Engenharia de Produção e graduação em Ciência da Computação pela UFSM. Realizou estágio de pós-doutorado na Universidade de Monash (Austrália) e possui experiências internacionais em pesquisa, desenvolvimento e inovação no LaSIGE/ULisboa/Portugal e CritiX/UNI/Luxemburgo. Durante essas experiências, atuou em projetos internacionais com pesquisadores e especialistas de instituições estrangeiras como CMU/USA, UPorto/Portugal, UCoimbra/Portugal, Portugal Telecom/Portugal, TUM/Alemanha, Infineon/Alemanha, Ether Trust/França, UPMC/França e SnT/Luxemburgo. Atualmente é professor associado na UNIPAMPA. Possui experiência na área de Ciência da Computação, com ênfase em Segurança Cibernética (ou Cibersegurança), Inteligência Artificial, Sistemas Distribuídos, Redes de Computadores, Tolerância a Falhas e Intrusões e Desenvolvimento de Software.

Resumo

Contêineres oferecem versatilidade permitindo o desenvolvimento de sistemas com arquiteturas monolítica e de microsserviços. Na primeira abordagem, um sistema inteiro opera dentro de um único contêiner, enquanto no segundo, um ou mais processos são encapsulados dentro de contêineres. Existem dois modelos predominantes para implementação baseada em contêineres : o modelo mestre-escravo e o modelo de contêiner aninhado. Neste minicurso introduzimos a ferramenta denominada DinD-Bench, publicamente disponível, para sistematizar a avaliação de desempenho entre o Docker padrão e o Docker dentro do Docker (DinD ou contêineres aninhados). A DinD-Bench incorpora componentes e códigos projetados especificamente para instanciar e executar benchmarks conhecidos, como SysBench, Stress, IOzone e iPerf, no Docker e sua variante de contêiner aninhado. Os recursos da DinD-Bench viabilizam uma execução automatizada, sistemática e reprodutível, podendo ser aplicada em outras arquiteturas computacionais e adaptada para outros sistemas operacionais, além de prover a coleta de valores de performance e permitir a incorporação de outros benchmarks. Para demonstrar a aplicação prática dos recursos introduzidos, apresentamos uma avaliação de desempenho considerando duas distribuições GNU Linux (Alpine e Debian) para os contêineres Docker, diversas cargas de trabalho e diferentes plataformas de computação, como nós da nuvem da Google e máquinas locais. Através das avaliações experimentais realizadas, podemos concluir que (a) os contêineres aninhados exigem até 7 segundos para serem inicializados, enquanto os contêineres padrão Docker exigem menos de 0,5 segundos para os sistemas operacionais Debian e Alpine; (b) Debian exibe desempenho superior de CPU e menor latência comparado ao Alpine; (c) Imagens Docker baseadas em Debian podem exigir 20% (ou mais) de tamanho de memória do que aquelas construídas em Alpine; e (d) As operações de disco e rede não tiveram diferenças significativas entre contêineres Docker aninhados e Docker. Além disso, vale ressaltar que algumas das disparidades decorrentes da combinação de contêineres e sistemas de hospedagem parecem ser influenciadas pela pilha de software em uso, incluindo diferentes versões de kernel, bibliotecas e outros pacotes de software essenciais.

3.1. Introdução

Contêineres revolucionaram o modo como aplicações podem ser instanciadas. Um contêiner é extremamente leve, provendo velocidade e eficiência de processamento, e garante isolamento e segurança na computação, sem afetar o software de gerenciamento de base de uma determinada arquitetura. Assim, por exemplo, um único servidor x86 pode hospedar facilmente centenas de contêineres, sendo, neste caso, a memória muitas vezes a principal restrição para o número de contêineres executados simultaneamente. Notavelmente, contêineres têm tempos de inicialização rápidos, demorando normalmente entre 1 a 2 segundos para serem instanciados.

Duas razões fundamentais destacam-se para o ressurgimento dos contêineres do ponto de vista técnico. Em primeiro lugar, as melhorias no suporte de *namespaces* e de *cgroups* do *kernel* Linux, desempenharam um papel fundamental no isolamento e limitação de recursos. Em segundo lugar, Docker, uma implementação específica de contêineres, transformou o cenário. Docker não apenas introduziu um formato de encapsulamento

atraente, mas também forneceu ferramentas indispensáveis e promoveu um ecossistema diversificado, tornando-se fundamental para este renascimento dos contêineres.

Uma alternativa para o desenvolvimento que tem sido explorada é o uso de contêineres aninhados, também conhecidos como *Docker in Docker* (DinD) ou *Docker Inside Docker*. DinD envolve a execução do Docker dentro de um contêiner Docker. Em vez de interagir com o *daemon* Docker do *host*, um novo mecanismo Docker é gerado dentro de um contêiner, fornecendo um ambiente isolado para gerenciar contêineres e imagens. DinD oferece uma solução elegante para criar contêineres Docker reprodutíveis e confiáveis. Ao lançar o Docker dentro do Docker, os desenvolvedores e administradores de sistema podem agilizar seus fluxos de trabalho de desenvolvimento, aumentar a segurança e simplificar seus pipelines de *Continuous Integration/Continuous Deployment* (CI/CD).

Em relação ao desempenho, pelo menos cinco pontos-chave podem ser destacados nos ambientes DinD para o ano de 2015 Amaral et al. (2015). Primeiro, as tarefas com uso intensivo de CPU em contêineres não apresentam diferença significativa de desempenho em comparação com configurações *bare-metal*. Em segundo lugar, os contêineres regulares são os mais rápidos, seguidos pelos contêineres aninhados e pelas máquinas virtuais, durante a criação de uma instância. Terceiro, a criação de contêineres aninhados envolve mais sobrecarga, mas ainda é mais rápida que as máquinas virtuais. Camadas extras de abstração impõem a execução de mais código, a fim de garantir a confiabilidade necessária, o que impacta na performance da aplicação. Quarto, a criação de vários pais com um único filho leva mais tempo devido a gargalos na inicialização do contêiner pai. Por último, configurações variadas de rede para contêineres aninhados (*host network*, *Linux bridge* e *Open vSwitch*) não apresentam impacto significativo no desempenho quando a rede física torna-se o gargalo.

No entanto, o panorama da tecnologia Docker evoluiu substancialmente de 2015 à 2024. Além disso, é importante destacar que avaliações passadas (e.g., (AMARAL et al., 2015)) concentraram-se exclusivamente em tarefas com uso intensivo de CPU, isto é, não exploraram cargas de trabalho levando em consideração quesitos como desempenho da memória e E/S de disco e de rede. Diferentemente de outras avaliações e soluções existentes, a DinD-Bench (FAVA et al., 2023, 2024) leva em consideração também os impactos potenciais de diferentes distribuições GNU/Linux e explora diversas configurações de servidores em ambientes de nuvem.

Considerando esses fatores, este capítulo introduz a DinD-Bench e apresenta uma avaliação de desempenho atualizada e que permite uma compreensão mais abrangente do comportamento do desempenho do Docker em contextos contemporâneos. Para além da DinD-Bench, é apresentada também uma metodologia de avaliação de desempenho de DinD e Docker em diferentes tipos de hardware e distribuições GNU/Linux usando *benchmarks* estabelecidos como SysBench[5], Stress[6], IOzone[7] e iPerf[8] (DIAZ et al., 2014; BACHIEGA et al., 2020; POKHARANA; GUPTA, 2023). O impacto de DinD foi considerado em plataformas de infraestrutura como serviço (IaaS) em nuvem, como *Google*

[5]<https://github.com/akopytov/sysbench>
[6]<https://github.com/resurrecting-open-source-projects/stress>
[7]<https://www.iozone.org/>
[8]<https://iperf.fr>

Compute Engine (GCE) e em ambientes de servidores locais tradicionais disponíveis.

O restante do capítulo está organizado da seguinte forma. A Seção 3.2 introduz Docker e DinD no cenário atual da computação. A DinD-Bench é apresentada na Seção 3.3, incluindo aspectos de arquitetura, implementação, funcionamento e utilização. A Seção 3.4 discrimina a metodologia dos experimentos, caracterizando os *benchmarks*, métricas e ambientes de execução. A Seção 3.5 mostra o resultados experimentais para CPU, memória, disco e rede, considerando quatro *benchmarks* distintos em cinco máquinas com diferentes configurações de hardware. Finalmente, as considerações finais são apresentadas na Seção 3.6.

3.2. Docker

Docker emergiu como uma tecnologia essencial em ambientes de desenvolvimento e produção, especialmente para implementação e implantação de arquiteturas baseadas em microsserviços (BOGNER et al., 2019; TAPIA et al., 2020; Karabey Aksakalli et al., 2021). No entanto, sua prevalência é enfatizada pela ampla adoção em diversos domínios e aplicações. Por exemplo, o Docker tornou-se uma ferramenta crucial na engenharia de software moderna, facilitando a instrumentação de testes automatizados e a implementação de pipelines de *Continuous Integration/Continuous Deployment* (CI/CD). Isto demonstra a sua versatilidade e eficácia no atendimento de demandas complexas do desenvolvimento de software contemporâneo.

Em termos de programação paralela, Docker oferece um desempenho similar às execuções em ambiente físico e superior ao uso de máquinas virtuais. Por outro lado, o grande tempo de inicialização de microsserviços, não é algo tolerável para aplicações com tempo de execução na ordem de poucos segundos. A grande vantagem está no fato de que contêineres permitem versatilidade de gerenciamento, o que provê uma forma de avaliar a instanciação de um grande número de nós, por exemplo. Uma outra vantagem é que existem aplicações com muitas dependências de software, como os modelos de previsão do tempo. O uso de uma versão distinta de um pacote ou a versão diferente de uma biblioteca pode gerar erros de compilação. Conteinerização pode ser a solução para que esses tipos de erros não ocorram.

Existem dois modelos predominantes para implementação baseada em contêiner Docker dentro da estrutura de microsserviços: o modelo mestre-escravo e o modelo de contêiner aninhado (DinD) (AMARAL et al., 2015). O modelo mestre-escravo compreende um contêiner mestre orquestrando outros como escravos, onde esses operam os processos da aplicação. O mestre é responsável por monitorar os contêineres subordinados e facilitar sua comunicação. Em contraste, a abordagem de contêiner aninhado envolve a criação hierárquica de contêineres subordinados (filhos) dentro de um contêiner principal (pai). O contêiner pai pode ser agnóstico e simplesmente existir, enquanto os filhos executam os processos do aplicativo, confinados dentro dos limites definidos pelo pai.

A abordagem DinD simplifica o gerenciamento em arquiteturas de microsserviços. Isso reduz o número de etapas de reprodução de experimentos e isola os ambientes de desenvolvimento e teste, aumentando a segurança e o isolamento, simplificado implementação de pipelines de CI/CD e gerenciamento de recursos em cenários de multi-locação, onde múltiplas equipes ou usuários exigem ambientes isolados em infraestrutura com-

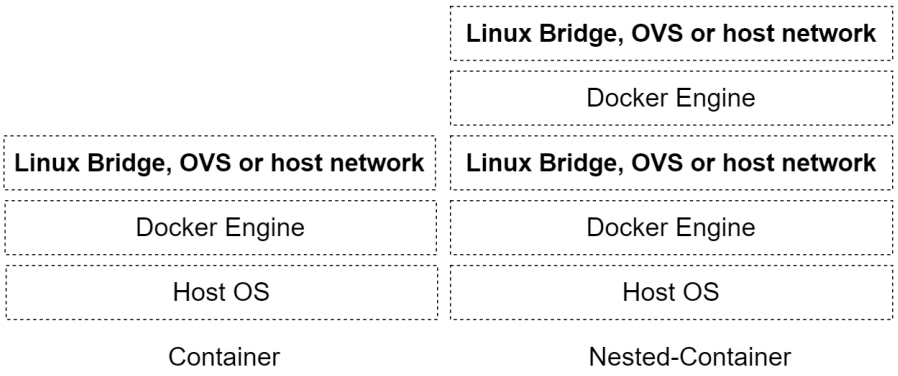

Figura 3.1. Duas camadas de *daemon* Docker (adaptado de (AMARAL et al., 2015), Figura 2)

partilhada (KUSHWAH, 2024; PETAZZONI, 2024). DinD facilita a comunicação entre processos (IPC), garante o compartilhamento de destino e permite o compartilhamento de memória, disco e recursos de rede. No entanto, esta abordagem pode introduzir custos adicionais devido às duas camadas do *daemon* Docker, conforme ilustrado na Figura 3.1.

Com relação aos aspectos práticos, a Figura 3.2 ilustra a interação entre os componentes Docker (*Docker Workflow*). O Dockerfile é considerado o arquivo padrão que especifica como a imagem Docker irá ser construída e qual será o seu conteúdo. Um contêiner Docker representa uma instância de uma imagem Docker em execução. O Docker Hub pode ser utilizado para armazenar de maneira centralizada e organizada as imagens Docker, que serão utilizadas tanto no Docker padrão quanto no DinD.

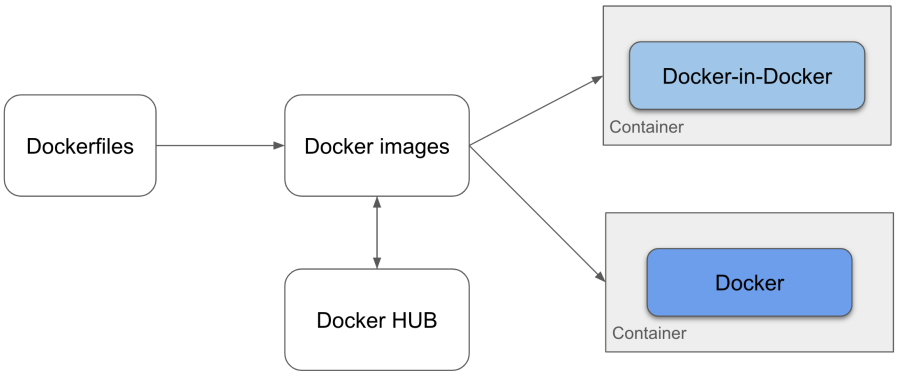

Figura 3.2. Relação entre Dockerfile, imagens, contêineres e hub docker

No contexto do Docker, o fluxo de trabalho pode ser definido através de 7 etapas (OZOR, 2023), como discriminado a seguir.

1. **Desenvolvimento e construção da aplicação:** O desenvolvimento de uma aplicação pode ser realizado em qualquer linguagem de programação. Posteriormente, são definidos o conjunto de instruções para construir uma imagem Docker, tipicamente em um arquivo de texto estruturado conhecido como Dockerfile. O Doc-

kerfile define a imagem base (e.g., Debian 12.0, Ubuntu 22.04), as dependências necessárias, o caminho do código da aplicação que será copiado para dentro da imagem e os comandos a serem executados automaticamente quando o contêiner for iniciado.

2. **Construção de uma imagem Docker:** Através da CLI (*Command Line Interface*) do Docker, ou de uma ferramenta de construção como *Docker Compose*, o usuário pode construir uma imagem Docker baseada na especificação contida no Dockerfile. A imagem Docker resultante é um pacote de software independente, encapsulando tudo o que é essencial para executar a aplicação, o que inclui o código-fonte/executável, o ambiente de execução e quaisquer dependências necessárias, como ferramentas de sistema, bibliotecas e configurações específicas.

3. **Execução de contêineres Docker:** A partir de uma imagem previamente criada, o usuário pode instanciar um contêiner, via CLI ou outras ferramentas, utilizando a *Engine* do Docker. Contêineres são instâncias de imagens Docker que podem ser iniciadas, interrompidas e gerenciadas de forma independente. Os contêineres garantem funcionalidade uniforme em diferentes ambientes, ao isolar o software do seu entorno. Pode-se executar contêineres localmente em uma máquina de desenvolvimento ou implantá-los em um ambiente de produção.

4. **Gerenciamento de contêineres Docker:** O *Docker Engine* fornece um conjunto de recursos e comandos para gerenciar contêineres em execução. Pode-se visualizar contêineres em execução, interrompê-los, iniciar contêineres interrompidos e remover aqueles que não forem mais necessários.

5. **Utilização do *Docker Compose* para aplicações em vários contêineres:** *Docker Compose* é uma ferramenta para definir e executar aplicativos Docker de vários contêineres. Os serviços são definidos através de arquivos de configuração YAML, onde são especificados recursos de rede e volumes necessários para a aplicação / serviço. É possível iniciar e finalizar diversos contêineres simultaneamente através do *Docker Compose*.

6. **Publicação e distribuição imagens Docker:** As imagens do Docker podem ser publicadas em repositórios privados ou públicos, como é o caso do *Docker Hub*. A publicação das imagens facilita o gerenciamento e a implantação e também potencializa a re-utilização por outras pessoas em outros ambientes e contextos.

7. **Atualização e iteração:** À medida que novas versões da aplicação continuam a ser desenvolvidas, pode-se iterar na imagem do Docker, atualizando o Dockerfile e reconstruindo a imagem. Isso permite distribuir e implantar facilmente novas versões do seu aplicativo.

3.3. DinD-Bench

A Figura 3.3 apresenta a visão geral da DinD-Bench. Ela é composta por cinco componentes principais: (1) módulo principal; (2) conjunto de *benchmarks*; (3) repositório de dados; (4) módulo de análise de dados; e (5) recursos gerais. A função do módulo

principal é permitir que o usuário selecione o *benchmark* que será executado. Um aspecto importante do módulo principal é o fato de ele estar preparado para incorporar novos *benchmarks*. A inclusão de um novo *benchmark* requer apenas poucas ações do usuário, como inclusão de um novo diretório e a configuração do arquivo padrão de inicialização e execução do *benchmark*. A partir desse ponto, o *benchmark* passa a estar automaticamente disponível na DinD-Bench.

O arquivo de configuração e execução do *benchmark* está programado para receber entradas (i.e., parâmetros de execução) e gerar saídas (e.g., arquivos com as estatísticas de saída) que estarão disponíveis no repositório de dados ao final da execução. Na configuração de um novo *benchmark*, o usuário necessita também realizar ajustes de parâmetros de entrada e saída para a execução e geração de dados de saídas compatíveis com a estrutura e organização da DinD-Bench.

Ao final da execução do(s) *benchmark(s)*, o usuário pode utilizar o módulo de análise de dados para gerar estatísticas e gráficos da(s) execução(ões). Para cada novo *benchmark*, o usuário pode também incluir novos recursos de processamento e análise de dados no respectivo módulo. Isto torna a solução flexível e ajustável a diferentes ferramentas e contextos.

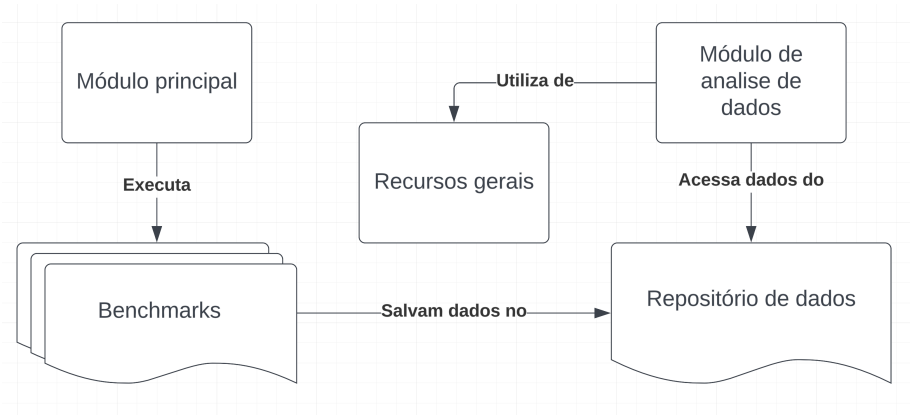

Figura 3.3. Arquitetura organizacional DinD-Bench

A Figura 3.4 ilustra o fluxo de execução da DinD-Bench. O usuário inicializa o módulo principal escolhendo o(s) *benchmark(s)* que serão executados e a instância que será executada (i.e., Docker ou DinD). No momento que o módulo principal inicializa o(s) *benchmark(s)* especificado(s) pelo usuário, é executada a instância definida pelo usuário, ou seja, Docker ou DinD. No primeiro caso é instanciado um contêiner Docker através do comando `docker run`. Já no segundo caso são preparadas as configurações que serão repassadas para o contêiner aninhado. A execução do DinD é realizada através da execução do comando `docker run –privileged`.

Para entender os detalhes da DinD-Bench, é importante compreender o que são *nested* contêineres e o que os difere de contêineres normais. Resumidamente, quando um contêiner é executado dentro de outro, cunha-se o nome de *nested* contêiner ou contêiner aninhado. Uma das estratégias mais conhecidas e utilizadas é a Docker-in-Docker (DinD). Na DinD é instanciado um contêiner Docker *daemon* dentro de outro contêiner

Docker. Para executar contêineres aninhados, é necessário utilizar a *flag* `-privileged`, que irá elevar os privilégios do contêiner para viabilizar o acesso a recursos da máquina hospedeira (e.g., diretório de dados, dispositivos de E/S).

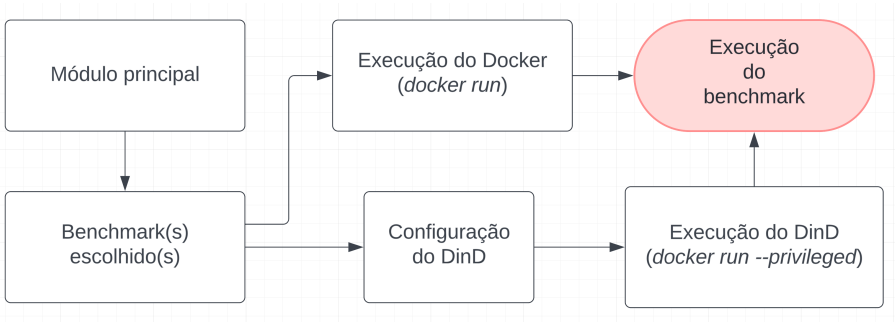

Figura 3.4. Fluxo de execução da DinD-Bench

A Figura 3.5 ilustra visualmente a organização dos contêineres quando utilizada a abordagem DinD. Na máquina hospedeira (*host*) é executado um contêiner Docker, dentro do qual é instanciado um segundo contêiner Docker, identificado na figura por *Docker-in-Docker*. As setas presentes na imagem indicam onde são executados os comandos passados pelo usuário para cada um dos ambientes.

A instanciação da DinD-Bench inicia com a execução de um comando na máquina hospedeira (física ou virtual). Na sequência, é executado um comando para a criação do contêiner aninhado dentro da primeira instância Docker. Finalmente, é executado o *benchmark* selecionado dentro do contêiner aninhado (DinD).

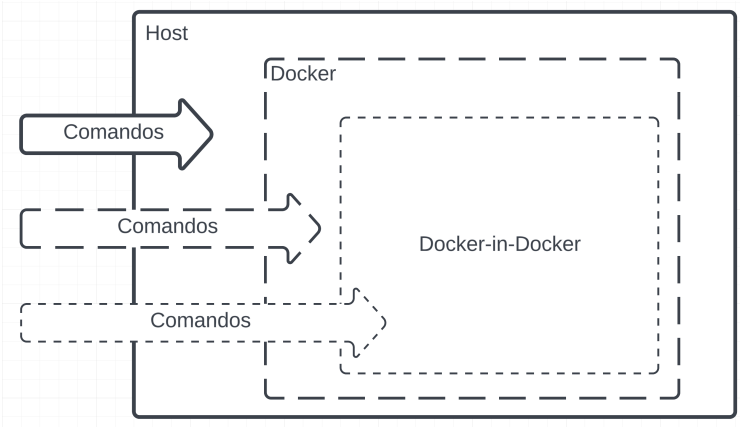

Figura 3.5. Ambiente e comandos no contexto da DinD-Bench

Observando a arquitetura e o fluxo de funcionamento, a DinD-Bench pode ser considerada também um automatizador da execução de aplicações dentro de contêineres normais ou aninhados (i.e., DinD). Na prática, a DinD-Bench é agnóstica às aplicações que são executadas dentro dos contêineres. Consequentemente, o usuário pode adaptar a solução para executar quaisquer outros tipos de aplicações.

O código da DinD-Bench, bem como dados coletados nas medições de desempenho apresentadas na Seção 3.5, pode ser encontrado no repositório Github[9]. Os *benchmarks* disponíveis estão configurados para execução em contêineres Docker baseados em Debian e Alpine. É importante destacar também que novas distribuições GNU/Linux podem ser facilmente acrescentadas para ampliar ainda mais as análises e avaliações. Atualmente, a avaliação de desempenho produzida pelos *benchmarks* já incorporados à DinD-Bench permitem a avaliação sistemática de CPU, memória, latência e taxa de transferência de E/S (para rede e disco).

3.4. Metodologia

Uma visão geral da metodologia é apresentada na Figura 3.6. Para avaliar o desempenho de quatro componentes cruciais do computador (CPU, memória, disco e rede) em dois ambientes de contêiner (Docker e Docker-in-Docker) em duas distribuições GNU/Linux (Debian 12.1 e Alpine 3.18), conduziu-se a execução de quatro *benchmarks* em cinco máquinas com configurações distintas. Cada *benchmark* é executado 20 vezes, resultando em um total de 1.600 execuções. A Tabela 3.1 resume as principais configurações. Para uma descrição detalhada e configurações dos experimentos, disponibilizamos as informações publicamente acessíveis no GitHub (FAVA et al., 2023).

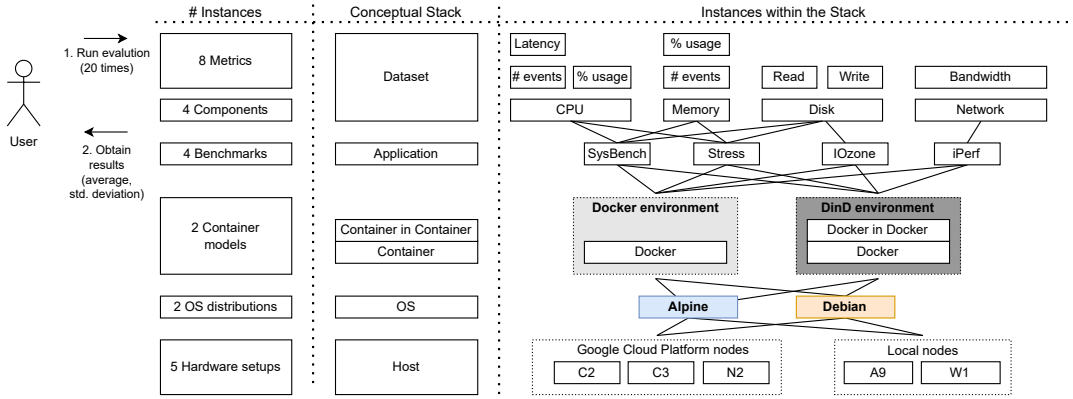

Figura 3.6. Metodologia aplicada (conforme (FAVA et al., 2024), Figura 2)

Conforme mostra a Tabela 3.1, o ambiente de execução compreende três nós do Google Cloud Platform (GCP): C3, N2 e E2, e dois nós locais, A9 (Intel i7) e W1 (AMD Ryzen). Especificamente, os nós GCP C3 (tipo c3-standard-4), N2 (tipo n2-standard-2) e E2 (tipo e2-medium) apresentam 2, 1 e 1 núcleos/threads, respectivamente. Em contraste, os nós locais A9 e W1 estão equipados com 4 e 8 núcleos físicos, respectivamente. Vale ressaltar que os nomes das máquinas locais não remetem nenhum padrão especifico de nomenclatura de nenhum ambiente de *cloud*.

Os *benchmarks* selecionados são amplamente reconhecidos nas comunidades de profissionais e pesquisadores para avaliar a performance de aspectos de hardware (UpCloud, 2018; QIAN, 2019; Dell Technologies, 2021; DIAZ et al., 2014; XAVIER et al., 2016; BACHIEGA et al., 2020; POKHARANA; GUPTA, 2023). Esses *benchmarks* são

[9]<https://github.com/DinDperf/DinD-Bench>

Tabela 3.1. Configurações dos benchmarks, contêineres, sistemas computacionais e computadores ((FAVA et al., 2024), Tabela 1))

Item	Descrição
SysBench	Emprego de uma sequência padrão de testes, incluindo threads, memória, CPU e FILEIO.
Stress	Um teste com tempo limite de 100s para cada parâmetro de CPU, HDD, E/S e VM.
IOzone	Utilização de 1 GB de dados para leitura e gravação em blocos de 4 KB.
iPerf	Uso dos parâmetros cliente-servidor padrões (locais) para medir a taxa de transferência e a latência.
Contêiner	Docker e Docker in Docker (DinD)
Sist. Oper.	Alpine 3.18 e Debian 12.1
C2 (GCP)	Debian 12, Docker Engine 24.0.7, Intel Xeon 3.10GHz, 32GB NVMe de Disco, 16GiB DIMM RAM
C3 (GCP)	Debian 12, Docker Engine 24.0.7, Intel Xeon Platinum 8481C, 32GB NVMe de Disco, 16GiB DIMM RAM
N2 (GCP)	Debian 12, Docker Engine 24.0.7, Intel Xeon 2.80GHz, 32GB PersistentDisk, 8GiB DIMM RAM
A9 (Local)	Debian 11, Docker Engine 24.0.7, Intel i7-9700 3.00GHz, 1TB WDC WD10EZEX-08W, 16GiB DIMM DDR4
W1 (Local)	Ubuntu 22.04.3 LTS, Docker Engine 24.0.5, AMD Ryzen 7 5800X, 512GB NVMe de Disco, 64GiB DIMM DDR4

abrangentes para mensurar processamento, memória e desempenho de E/S, abrangendo operações de rede e disco. A seguir, são descritas as especificações das ferramentas de referência escolhidas para a avaliação.

- **SysBench**: é uma ferramenta versátil de *benchmark* amplamente utilizada em sistemas do tipo Unix. SysBench avalia o desempenho do sistema em áreas como CPU, memória, threads, E/S de disco e bancos de dados (KOPYTOV, 2023). Esta ferramenta auxilia gerentes de sistema e desenvolvedores a testar e analisar o desempenho de sistemas e aplicações em diversos cenários.

- **Stress**: é uma ferramenta projetada para aplicar uma variedade de testes em um sistema. Stress pode avaliar CPU, memória, E/S (sincronização) ou estresse de disco (WATERLAND, 2023). A ferramenta é comumente utilizada por programadores de núcleo do sistema, permitindo-lhes avaliar a escalabilidade do sistema e examinar minuciosamente os recursos de desempenho percebidos. Além disso, a ferramenta ajuda os programadores de sistemas a revelar classes específicas de falhas que muitas vezes surgem apenas sob cargas pesadas, oferecendo percepções críticas sobre o desempenho do sistema sob condições de estresse.

- **IOzone**: é uma ferramenta essencial no domínio de avaliação de sistemas de arquivos (IOZONE, 2023). IOzone é um utilitário de *benchmark* versátil que gera e

avalia uma diversidade de operações de arquivos. Ele foi portado para várias máquinas e é compatível com vários sistemas operacionais. IOzone é inestimável para conduzir análises abrangentes de sistemas de arquivos em diversas plataformas de computador. O *benchmark* avalia rigorosamente o desempenho de E/S de arquivos em um espectro de operações, incluindo leitura, gravação, releitura, reescrita, leitura reversa, leitura circular, leitura aleatória, utilizando as funções *fread*, *fwrite*, *pread*, *aio_read* e *aio_write* e *mmap*.

- **iPerf**: é uma ferramenta de código aberto frequentemente utilizada para medir o desempenho de redes de computadores baseadas em IP (IPERF.FR, 2024). iPerf é eficaz para avaliações essenciais, incluindo testes de largura de banda, latência e perda de pacotes. A ferramenta designa um dispositivo como servidor, que escuta ativamente as conexões, operando em um modelo cliente-servidor. A segunda instância do iPerf inicia as conexões, funcionando como cliente, para realizar testes de desempenho.

3.5. Resultados Experimentais

Nesta seção são discutidos os principais resultados obtidos relativos a CPU, memória, disco e componentes de rede, em suas respectivas subseções. São apresentados uma série de gráficos obtidos pela execução dos *benchmarks*, para cada item considerado. Nos gráficos são apresentados resultados para as diferentes configurações de hardware, as duas distribuições GNU/Linux utilizadas nos experimentos (Alpine preenchido em azul e Debian preenchido em laranja) e os dois modelos de contêiner (DinD preenchido com cores mais escuras e Docker preenchido com cores mais claras).

3.5.1. CPU

Na Figura 3.7 são apresentados os resultados de uso da CPU pelo *benchmark* SysBench. O número de eventos e a latência estão sendo levados em consideração. Observando a tecnologia da CPU e a frequência de operação (consulte Tabela 3.1), percebe-se que o número de eventos da CPU se correlacionam diretamente com a capacidade da CPU. Em particular, o sistema W1 demonstra o mais alto desempenho, enquanto N2 apresenta o mais baixo. Como esperado, há uma relação quase inversa, ou seja, a melhor CPU apresenta a menor latência.

Os resultados também indicam que os contêineres Docker baseados em Debian superam consistentemente os resultados das configurações equivalentes baseados em Alpine, obtendo menor latência de CPU. Notavelmente, estudos recentes também destacaram disparidades de desempenho entre imagens Docker construídas em diferentes distribuições Linux para vários sistemas, como bancos de dados (por exemplo, Cassandra, Mongo e MySQL) e servidores web (por exemplo, Nginx) (IBRAHIM; SAYAGH; HASSAN, 2020). Uma distinção importante entre essas imagens Docker está no número variável de bibliotecas instaladas (por exemplo, de 13 a 119) em diferentes distribuições Linux para o mesmo sistema (por exemplo, MySQL). Além disso, o número de eventos e a latência da CPU mostrados na Figura 3.7 indicam que a influência do DinD sobre o Docker é insignificante para ambos os sistemas operacionais.

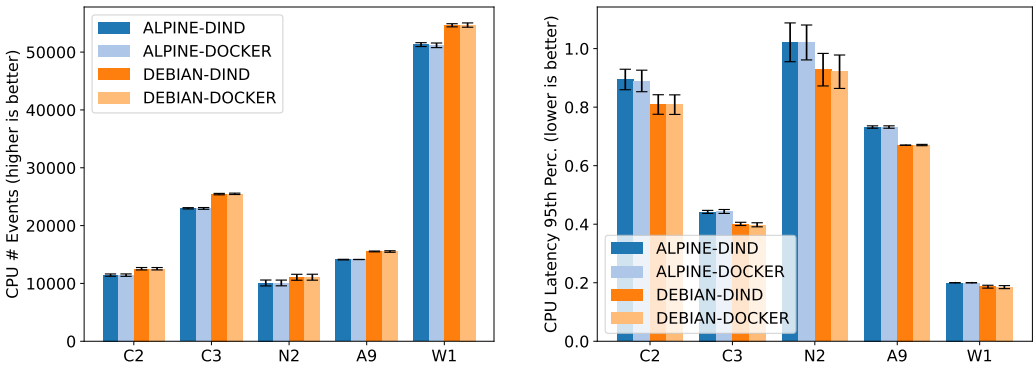

Figura 3.7. Estatísticas de CPU usando SysBench incluindo número de eventos e valores de latência (FAVA et al., 2024), Figura 3

Na Figura 3.8, são apresentados os dados sobre o número de eventos e porcentagem dos resultados de SysBench para o subsistema de testes usando threads. Notavelmente, as instâncias do Docker exibem uma contagem maior de eventos com latência menor. A maior latência do DinD permanece consistente em diversos sistemas de hospedagem. No entanto, é importante observar que essa latência varia entre as máquinas. Por exemplo, em cenários como W1, a latência DinD pode ultrapassar a latência do Docker em mais de 10%.

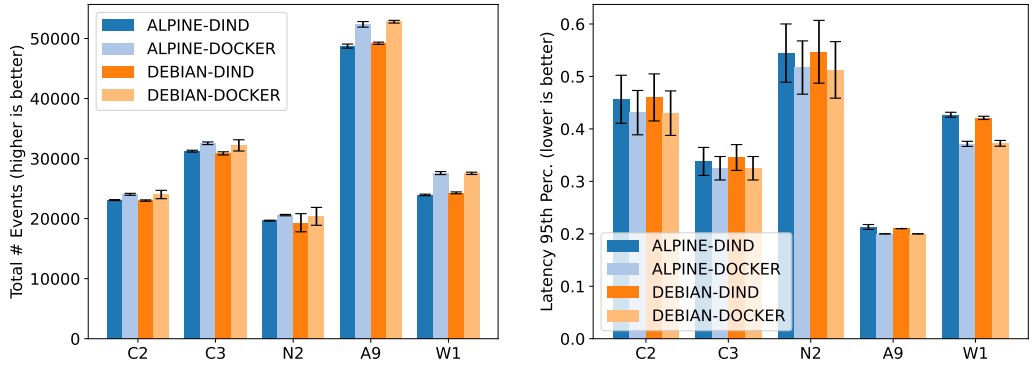

Figura 3.8. Estatísticas de threads usando SysBench, incluindo o número de eventos e porcentagem (FAVA et al., 2024), Figura 4

Ao examinar o comportamento de execução de Stress em CPU para DinD e Docker em 1024 amostras (segundos), conforme ilustrado na Figura 3.9 e na Figura 3.10, torna-se evidente uma notável latência de inicialização (*bootstrap*) em instâncias DinD. Por exemplo, embora o *benchmark* Stress inicie sua execução imediatamente no Alpine Docker, isso pode levar mais de 20 segundos em um sistema de hospedagem C2 ao usar DinD. Esse padrão de latência persiste em diferentes *benchmarks* e ambientes de execução, bem como também é observado consistentemente tanto em instâncias Debian como Alpine. Notavelmente, a latência de *bootstrap* de DinD pode variar significativamente de uma máquina para outra. O sistema N2 exige mais tempo de inicialização. Já W1 apresenta desempenho mais rápido.

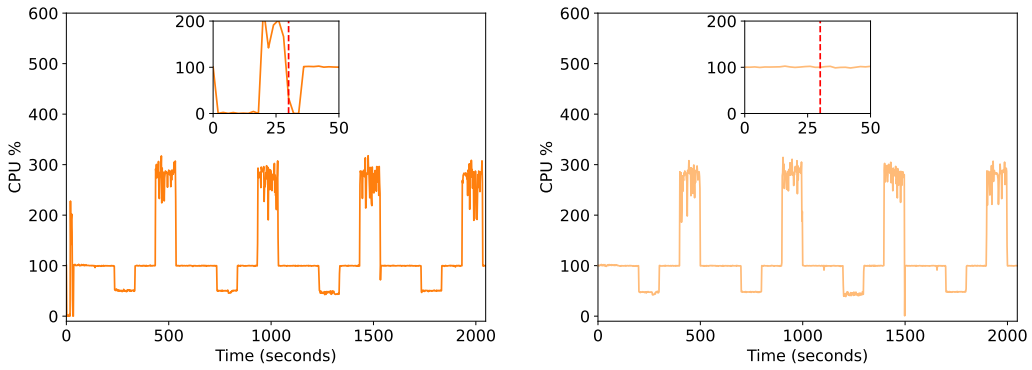

Figura 3.9. Comparação de DinD e Docker usando Stress para Debian no sistema C3

Para investigar melhor a latência de *bootstrap*, mostrada na Figura 3.9 e na Figura 3.10, foram analisados os tempos de inicialização e desligamento dos contêineres DinD, conforme valores apresentados na Figura 3.11. Enquanto os contêineres Docker iniciam em menos de 500 ms, DinD leva em média até 6 segundos para iniciar. Isso significa que DinD pode ser 12 vezes mais lento que os contêineres Docker para se tornar operacional para a execução de aplicações.

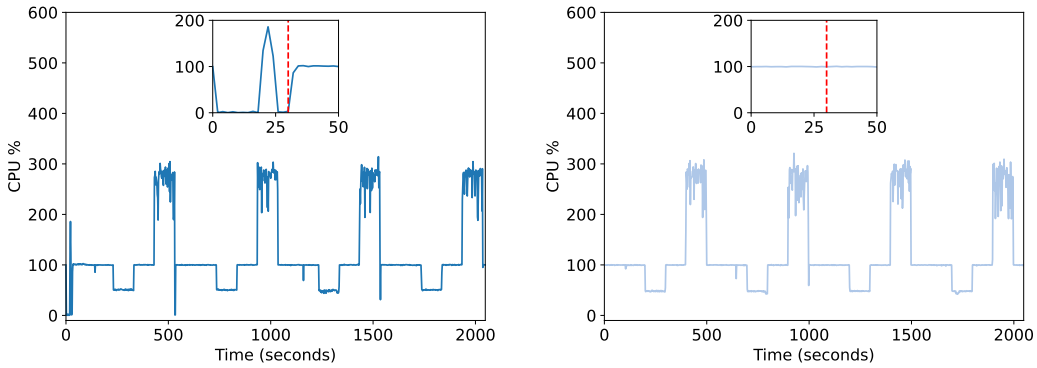

Figura 3.10. Comparação de DinD e Docker usando Stress para Alpine no sistema C3

De forma similar, embora os contêineres Docker terminam em menos de 100 ms, DinD pode levar mais de 1,5 segundos para parar completamente. Também é importante notar que não há diferença significativa nos tempos de inicialização e desligamento entre contêineres DinD baseados em Alpine e Debian. Curiosamente, o sistema A9 exibe um tempo de inicialização e desligamento um pouco mais lento, apesar de sua CPU Intel i7-9700 com clock de 3GHz. Atribui-se o alto desvio padrão à natureza não determinística inerente aos tempos de paralisação dos contêineres. No entanto, é importante observar que a pilha de software em uso pode potencialmente influenciar essa variação. Ao contrário dos outros sistemas de hospedagem que operam em Debian 12 ou Ubuntu 22.04, o sistema A9 roda em Debian 11, conforme indicado na Tabela 3.1. Esta distinção nos sistemas operacionais pode contribuir para as disparidades observadas no desempenho.

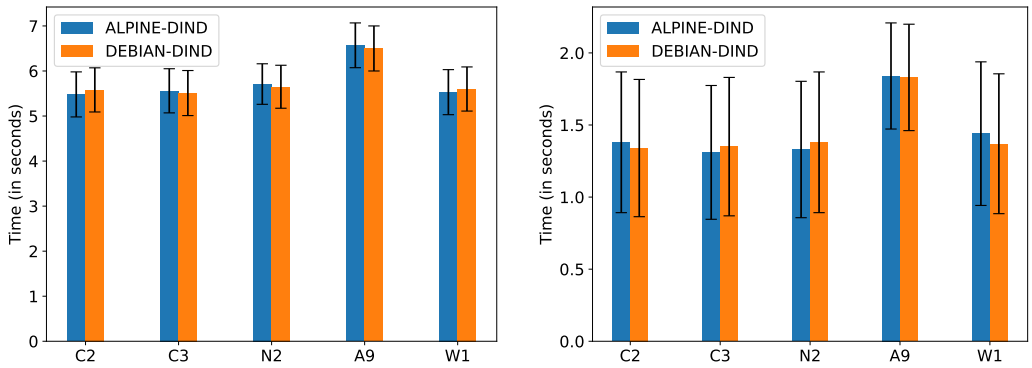

Figura 3.11. Tempo necessário para inicializar e parar contêineres DinD (FAVA et al., 2024), Figura 6

3.5.2. Memória

Na Figura 3.5.2, são apresentadas as contagens de eventos de memória para SysBench e a utilização de memória (por amostragem) para Stress. Os resultados tornam evidente que o número total de eventos de memória permanece consistente nas configurações DinD e Docker, os quais empregam a mesma distribuição Linux. Atribui-se essa consistência à metodologia uniforme de coleta de dados do SysBench em ambos os ambientes. No entanto, há uma disparidade notável nas contagens de eventos de memória, ao comparar as distribuições Debian e Alpine. Em particular, a imagem Docker baseada em Debian incorre em uma sobrecarga significativa, ultrapassando 20% do tamanho da memória em todos os ambientes de execução. Essa discrepância ressalta o impacto da distribuição Linux subjacente na geração de eventos de memória.

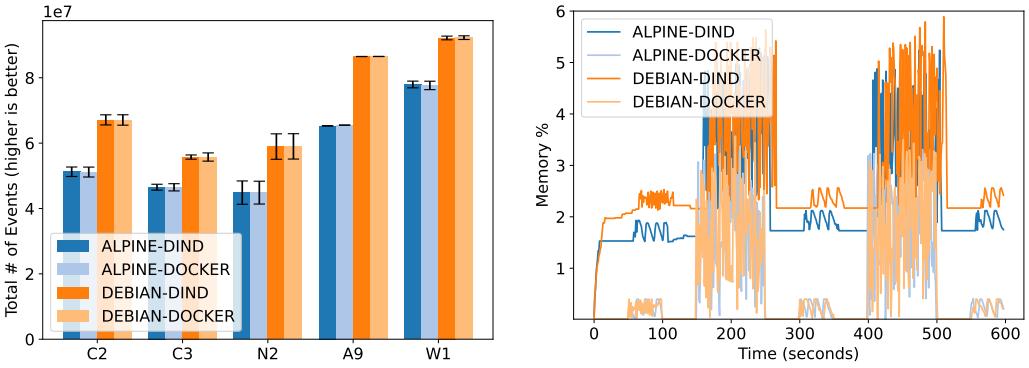

Figura 3.12. Consumo de memória de SysBench e Stress (FAVA et al., 2024), Figura 7

Além disso, pode-se fazer pelo menos três observações a partir da Figura 3.5.2. Em primeiro lugar, é evidente que o consumo geral de memória de DinD é significativamente maior do que o Docker em ambas as distribuições Linux. Esta medida abrange o consumo de memória inerente do *benchmark*, como a sobrecarga de memória adicional introduzida pelo DinD. Vale a pena enfatizar que as amostras de utilização de memória

são coletadas em tempo real, fornecido por *docker stats* para um contêiner específico em execução.

Em segundo lugar, surge uma diferença visível entre DinD baseado em Debian (representado pela linha amarela) e DinD baseado em Alpine (representado pela linha azul). Debian DinD exibe consistentemente um consumo de memória significativamente maior do que seu equivalente Alpine (no gráfico, é fácil visualizar considerando o par DinD). Mais uma vez, essa variação ressalta que as imagens Docker baseadas no Debian podem exigir 20% a mais memória do que aquelas construídas no Alpine.

Além disso, a Figura 3.5.2 revela um padrão cíclico no comportamento do *benchmark*. Esta ocorrência cíclica é comum em todos os programas ao capturar estatísticas de utilização de memória usando `docker stats`. Atribui-se às disparidades ilustradas aqui às diferenças nas imagens Docker enraizadas em distribuições Linux distintas (IBRAHIM; SAYAGH; HASSAN, 2020). Dependendo das bibliotecas específicas da distribuição usadas para suportar e executar os *benchmarks*, o consumo de memória pode variar significativamente, levando às discrepâncias observadas no consumo de memória.

3.5.3. Disco

Na Figura 3.13 são mostrados os resultados do *benchmark* IOzone em GB/s. Os testes incluíram as execuções nas opções *read*, *write*, *reread* e *rewrite*. Observa-se diferenças mínimas entre as quatro abordagens, tanto para DinD quanto para Docker usando Alpine e Debian, em nossos sistemas de hospedagem. Geralmente, as operações de releitura apresentam uma ligeira vantagem nos contêineres Docker em comparação ao DinD. Em ambientes específicos, como A9 e W1, as distinções entre contêineres Docker e DinD podem variar para *benchmarks* ou aplicativos com uso intensivo de disco de curto prazo. Seria interessante uma exploração mais abrangente dessas diferenças, através da execução de *benchmarks* de E/S prolongados e robustos, como a avaliação de uma aplicação real de cargas de trabalho.

Enquanto os sistemas C3, C2 e N2 usam Google PersistentDisk com especificações como revisão 1, em conformidade com SPC-4 e dispositivo de estado sólido, W1 alcança desempenho superior utilizando discos SSD (modelo: SM2P32A8-512GC1, versão de firmware: VC0S032V, NVMe Versão: 1.4). Por outro lado, A9 também deveria alcançar maior desempenho usando seus discos SSD de alta velocidade (modelo: SKHynix_HFS256GD9TNI-L2B0B, versão de firmware: 11710C10, versão NVMe: 1.3). No entanto, a pilha docker em A9 automaticamente reduz a taxa de transferência de E/S do disco de até 62 MB/s para entre 10 MB/s e 15MB/s.

3.5.4. Rede

A Figura 3.14 apresenta os resultados de execução de desempenho de rede para iPerf com medidas expressas em Gbits/s. Fica evidente pelos dados que existem variações insignificantes de desempenho entre DinD e Docker nos nós C2, C3 e N2, todos hospedados no Google Cloud Platform. Porém, em sistemas de hospedagem local, A9 e W1, os contêineres Docker apresentam desempenho até 7% superior em relação ao DinD.

Um dos principais fatores que contribuem para essa discrepância é o uso de diferentes versões de distribuições e sistemas operacionais Linux (ou seja, versões de *kernel*).

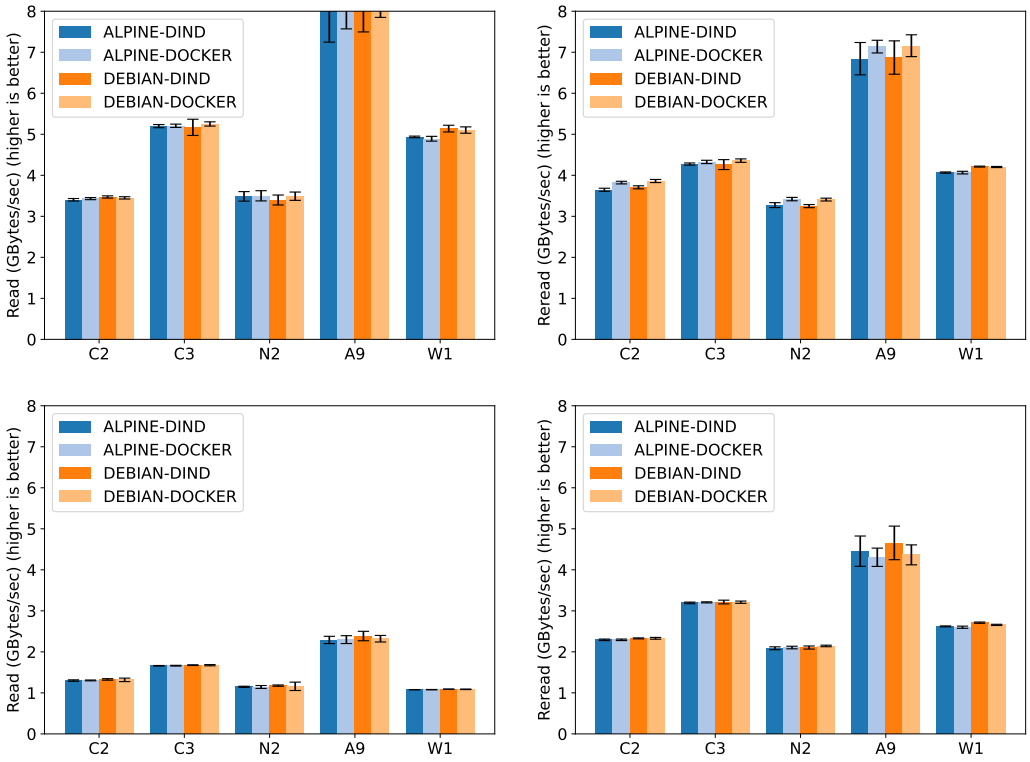

Figura 3.13. Resultados das operações read, reread, write e rewrite usando IOzone

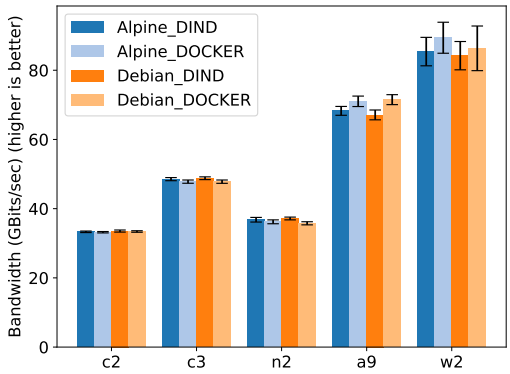

Figura 3.14. Valores de largura de banda da rede coletados por iPerf (FAVA et al., 2024), Figura 9

Embora todos os nós do GCP utilizem o Debian 12, o sistema A9 emprega o Debian 11 e o W1 opera no Ubuntu 22.04. Consequentemente, variações nas versões instaladas de pacotes e bibliotecas contribuem para a diferença de desempenho observada.

3.6. Considerações finais

Neste capítulo introduzimos a ferramenta DinD-Bench e avaliamos o desempenho de contêineres Docker aninhados (DinD) dentro de um contexto de arquiteturas baseadas em microsserviços. A DinD-Bench permite avaliar sistematicamente as capacidades do DinD em uma variedade de ambientes de hospedagem, através da utilização de *benchmarks* bem estabelecidos de CPU, memória, disco e E/S de rede, nomeadamente Sysbench, Stress, IOZone e iPerf. Implantamos esses *benchmarks* em imagens Docker criadas em distribuições Debian e Alpine Linux. Por fim, coletamos as estatísticas dos *benchmarks* em três nós da Google Computing Platform (GCP) e dois nós locais.

As comparações entre as distribuições de Linux distintas (Debian e Alpine), utilizado na execução dos *benchmarks*, revelam que a influência do DinD sobre o Docker em relação ao número de eventos e a latência da CPU são insignificantes para ambos os sistemas operacionais. Além disso, não há diferenças significativas entre DinD e Docker para E/S de disco e rede. No entanto, em termos de consumo de memória, a execução dos contêineres indicam diferenças não negligenciáveis entre Docker e DinD. Além disso, os contêineres DinD necessitam de um tempo de inicialização de até 7 segundos.

Estes fatores, nomeadamente, o consumo de memória e o tempo de inicialização, surgem como potenciais restrições em aplicações paralelas ou arquiteturas de microsserviços que operam sob restrições de tempo ou recursos. Vale a pena enfatizar que algumas das disparidades, como o maior consumo de memória, parecem ser resultado direto da pilha de software em uso, incluindo diferentes versões de *kernel*, bibliotecas e outros pacotes essenciais. Já um tempo de inicialização de 7 segundos pode ser considerado inaceitável para microsserviços de baixa latência e curta duração.

Para facilitar a transparência e a reprodutibilidade, a ferramenta DinD-Bench e os dados coletados estão disponíveis no repositório GitHub (FAVA et al., 2023). Este repositório garante a reprodutibilidade dos experimentos e também pode ser utilizado como um recurso valioso para futuras iniciativas de pesquisa ou experimentação prática.

Referências

AMARAL, M. et al. Performance Evaluation of Microservices Architectures Using Containers. In: *2015 IEEE 14th international Symposium on Network Computing and Applications*. Cambridge, MA, USA: IEEE, 2015. p. 27–34.

BACHIEGA, N. G. et al. Performance Evaluation of Container's Shared Volumes. In: *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Porto, Portugal: IEEE, 2020. p. 114–123.

BOGNER, J. et al. Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality. In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. Hamburg, Germany: IEEE, 2019. p. 187–195.

Dell Technologies. *How to Test Available Network Bandwidth Using 'Iperf'*. 2021. Alibaba Cloud ECS. <https://www.dell.com/support/kbdoc/en-us/000139427/how-to-test-available-network-bandwidth-using-iperf>.

DIAZ, C. O. et al. Performance Evaluation of an IaaS Opportunistic Cloud Computing. In: *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. Chicago, IL, USA: IEEE, 2014. p. 546–547.

FAVA, F. B. et al. Assessing the Performance of Docker in Docker Containers for Microsservice-based Architectures. In: *2024 32nd Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. Dublin, Ireland: IEEE, 2024. v. 1.

FAVA, F. B. et al. *DinD-Bench*. 2023. <https://github.com/DinDperf/DinD-Bench>.

IBRAHIM, M. H.; SAYAGH, M.; HASSAN, A. E. Too many images on dockerhub! how different are images for the same system? *Empirical Software Engineering*, Springer, v. 25, p. 4250–4281, 2020.

IOZONE. *IOzone Filesystem Benchmark*. 2023. <http://iozone.org/>. Last access in February, 2024.

IPERF.FR. *IPerf - The TCP, UDP and SCTP network bandwidth measurement tool*. 2024. <https://iperf.fr/>. Last access in February, 2024.

Karabey Aksakalli, I. et al. Deployment and Communication Patterns in Microservice Architectures: A Systematic Literature Review. *Journal of Systems and Software*, v. 180, p. 111014, 2021. ISSN 0164-1212.

KOPYTOV, A. *Scriptable database and system performance benchmark*. 2023. <https://github.com/akopytov/sysbench>. Last access in February, 2024.

KUSHWAH, S. *Docker Inside Docker*. 2024. Https://medium.com/@shivam77kushwah/docker-inside-docker-e0483c51cc2c.

OZOR, A. T. *Docker Workflow*. 2023. https://medium.com/augustineozor/docker-workflow-b9fe71d32184.

PETAZZONI, J. *Using Docker-in-Docker for your CI or Testing Environment? Think Twice*. 2024. https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/.

POKHARANA, A.; GUPTA, R. Using Sysbench, Analyze the Performance of Various Guest Virtual Machines on A Virtual Box Hypervisor. In: *2023 2nd International Conference for Innovation in Technology (INOCON)*. Bangalore, India: IEEE, 2023. p. 1–5.

QIAN, C. *Testing I/O Performance with Sysbench*. 2019. Alibaba Cloud ECS. <https://www.alibabacloud.com/blog/testing-io-performance-with-sysbench_594709>.

TAPIA, F. et al. From Monolithic Systems to Microservices: A Comparative Study of Performance. *Applied Sciences*, v. 10, n. 17, 2020. ISSN 2076-3417.

UpCloud. *Evaluating cloud server performance with sysbench*. 2018. <https://upcloud.com/blog/evaluating-cloud-server-performance-with-sysbench>.

WATERLAND, A. *stress - Tool to impose load on and stress test a computer system*. 2023. <https://github.com/resurrecting-open-source-projects/stress>. Last access in February, 2024.

XAVIER, M. G. et al. Understanding Performance Interference in Multi-Tenant Cloud Databases and Web Applications. In: *2016 IEEE International Conference on Big Data (Big Data)*. Washington, DC, USA: IEEE, 2016. p. 2847–2852.