MINICURSOS

Minicursos da XXII Escola Regional de Alto Desempenho da Região Sul (ERAD/RS)

18 a 20 de abril de 2022
Curitiba – PR, Brasil

Organização e Edição
Arthur Lorenzon, Márcio Castro e Mauricio Pillon

Realização

Organização

Apoio

Instituições colaboradoras

PATROCINADORES

DIAMANTE

OURO

ARTHUR LORENZON
MÁRCIO CASTRO
MAURICIO PILLON

MINICURSOS DA XXII ESCOLA REGIONAL DE ALTO DESEMPENHO DA REGIÃO SUL (ERAD/RS)

Porto Alegre
Sociedade Brasileira de Computação – SBC
2022

Dados Internacionais de Catalogação na Publicação (CIP)

E74 Escola Regional de Alto Desempenho da Região Sul (22. : 2022 : Curitiba, PR)

Minicursos da XXII Escola Regional de de Alto Desempenho da Região Sul (ERAD-RS 2022) [recurso eletrônico] – Organizadores: Arthur Lorenzon, Márcio Castro, Mauricio Pillon. – Porto Alegre : SBC, 2022.

ISBN 978-65-87003-87-0

1. Computação. 2. Alto desempenho. I. Lorenzon, Arthur, org. II. Castro, Márcio, org. III. Pillon, Mauricio, org. IV. Sociedade Brasileira de Computação. V. Universidade Federal do Paraná. VI. Universidade do Vale do Rio dos Sinos. VII. Título.

CDU 004

Catalogação elaborada por Francine Conde Cabral
CRB-10/2606

ERAD/RS 2022

XXII Escola Regional de Alto Desempenho da Região Sul
18 a 20 de abril de 2022
Universidade Federal Paraná (UFPR), Curitiba, Paraná, Brasil
https://web.inf.ufpr.br/erad2022/

A XXII Escola Regional de Alto Desempenho da Região Sul (ERAD/RS 2022) foi realizada entre os dias 18 a 20 de abril de 2022, na cidade de Curitiba, no campus da Universidade Federal do Paraná (UFPR). A ERAD/RS é promovida 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 2022 são alunos, profissionais e professores/pesquisadores que atuam direta ou indiretamente na computação de alto desempenho e em áreas correlatas. O evento engloba a 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 2022 contou com palestras, minicursos e fóruns de iniciação científica e pós-graduação com apresentação de trabalhos científicos. O Fórum de Iniciação Científica deste ano foi coordenado pelos professores Charles Christian Miers (UDESC) e Guilherme Galante (UNIOESTE). O Fórum de Pós-Graduação foi coordenado pelos professores Márcia Pasin (UFSM) e Márcio Seiji Oyamada (UNIOESTE). Os minicursos foram coordenados pelos professores Arthur Lorenzon (UNIPAMPA), Márcio Castro (UFSC) e Mauricio Pillon (UDESC). A Maratona de Programação Paralela foi coordenada por Matheus Serpa (UFRGS), Lucas Nesi (UFRGS) e Lucas Schnorr (UFRGS). A equipe organizadora foi coordenada pelos professores Luis Carlos de Bona (UFPR) Marco Zanata Alves (UFPR) e Rodrigo Righi (UNISINOS). A coordenação local ficou à cargo dos professores Carlos Maziero (UFPR), Daniel Gonçalves de Oliveira (UFPR) e Daniel Pigatto (UTFPR).

Índice

Mensagem da Coordenação Geral

Saudamos e damos as boas vindas à vigésima segunda edição da Escola Regional de Alto Desempenho da Região Sul, a ERAD/RS 2022, que neste ano acontece na cidade de Curitiba, capital do estado do Paraná, entre os dias 18 e 20 de abril. A ERAD/RS é um evento anual, promovido pela Sociedade Brasileira de Computação (SBC), por meio da Comissão Regional de Alto Desempenho da Região Sul (CRAD-RS), desde 2001.

O objetivo deste encontro, de caráter essencialmente regional, abrangendo a região sul do Brasil, é de qualificar profissionais da região nas áreas que compõem o Processamento de Alto Desempenho (PAD) e de proporcionar um fórum regular no qual se possa tanto apresentar os avanços recentes nessas áreas quanto discutir as formas de ensino de processamento de alto desempenho nas Instituições de Ensino Superior (IES) do sul do Brasil.

Além disso, as atividades da ERAD/RS aproximam academia, sociedade e indústria. A troca de experiências e conhecimento de novidades é estimulada através da intercalação de palestras com pessoas de renome na área, apresentações acadêmicas e relatos do mercado/indústria. Essa sinergia é ampliada pela interação de professores, estudantes e profissionais de diversas regiões do Brasil, a qual possibilita analisar questões por um viés diferente da sua realidade regional.

Para potencializar seus resultados, A ERAD/RS é itinerante, sendo abrigada, a cada ano, por uma instituição diferente. Neste ano, coube à Universidade Federal do Paraná (UFPR) gerenciar a execução deste encontro, que será realizado em Curitiba – PR, com o apoio da Universidade do Vale do Rio dos Sinos (Unisinos). A região sul do Brasil é, há muitos anos, referência no país na área de PAD. A ERAD/RS reflete este fato e, mais ainda, reforça esta posição de destaque quando se preocupa em formar novos pesquisadores e em manter atualizados os pesquisadores que criaram suas bases nos estados do sul.

Portanto, é com satisfação que nos envolvemos neste já tradicional evento, nos tornando parte dessa história e auxiliando o desenvolvimento científico do país. Assim, é nesse momento que reconhecemos a presença de instituições que acompanham a evolução desta história colaborando de forma efetiva para sua realização. Neste ano, a ERAD/RS contou com o apoio do instituto Serrapilheira, além de patrocínios das empresas Olist, SDC, Fineasy Tech e HP Enterprise.

Agradecemos, em especial, a todos os autores que submeteram trabalhos às sessões técnicas, às empresas e aos convidados diversos que aceitaram nosso convite e engrandecem a ERAD/RS com as suas participações. Por fim, agradecemos a prontidão com que diversos colegas do corpo multi-institucional deste evento, que tomaram para si diversos encargos e os conduziram a termo com pleno sucesso.

Obrigado pela presença de todos e aproveitem a ERAD/RS 2022.

Marco Antonio Zanata Alves (UFPR), Luis Carlos Erpen de Bona (UFPR) e Rodrigo da Rosa Righi (UNISINOS)
Coordenadores da ERAD/RS 2022

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 dois anos, foi realizada entre os dias 18 e 20 de abril de 2022 na cidade de Curitiba/RS, no campus da Universidade Federal do Paraná (UFPR). Este evento marca a primeira edição presencial da escola fora do estado do Rio Grande do Sul.

Um dos objetivos da ERAD/RS é qualificar profissionais da região sul nas diversas áreas do Processamento de Alto Desempenho (PAD). Com este intuito, todo o ano, são selecionados minicursos introdutórios e avançados em tópicos de interessa à comunidade. Nesta edição, foram selecionados seis minicursos, dos quais cinco viraram capítulos para este livro. Os minicursos aqui representados, apresentam tópicos de ponta da área de PAD, os quais irão certamente agradar os participantes do evento.

Os coordenadores dos minicursos agradecem aos autores, por compartilharem seus conhecimentos através da submissão de minicursos de alto nível para esta edição da escola, aos coordenadores e organizadores da ERAD/RS, pelo apoio dado na seleção dos minicursos e na realização do evento, além de desejar uma boa ERAD/RS 2022 a todos!

Arthur Lorenzon (UNIPAMPA), Márcio Castro (UFSC) e Mauricio Pillon (UDESC)
Coordenadores dos Minicursos da ERAD/RS 2022

Comitês Organizadores

Coordenação Geral

- Luis Carlos de Bona (UFPR)
- Marco Antonio Alves (UFPR)
- Rodrigo Righi (UNISINOS)

Coordenação Local

- Carlos Maziero (UFPR)
- Daniel Alfonso de Oliveira (UFPR)
- Daniel Pigatto (UTFPR)

Fórum de Pós-Graduação

- Márcia Pasin (UFSM)
- Márcio Oyamada (UNIOESTE)

Fórum de Iniciação Científica

- Charles Christian Miers (UDESC)
- Guilherme Galante (UNIOESTE)

Minicursos

- Arthur Lorenzon (UNIPAMPA)
- Márcio Castro (UFSC)
- Maurício Pillon (UDESC)

Coordenação da Maratona Paralela

- Matheus Serpa (UFRGS)
- Lucas Nesi (UFRGS)
- Lucas Schnorr (UFRGS)

Coordenação do WHPC

- Andrea Charão (UFSM)
- Patrícia Barcelos (UFSM)

Minicursos

Capítulo 1

Desenvolvimento de Aplicações Paralelas Adaptativas: Uma Visão de Duas Décadas (2001-2021)

Guilherme Galante
Universidade Estadual do Oeste do Paraná
Cascavel, Brasil

Rodrigo da Rosa Righi
Universidade do Vale do Rio dos Sinos
São Leopoldo, Brasil

Resumo

A evolução das arquiteturas paralelas aponta para ambientes dinâmicos nos quais o número de recursos ou configurações disponíveis pode variar durante a execução das aplicações. Isso pode ser facilmente observado em grades e nuvens, mas também pode ser explorado em clusters e arquiteturas de multiprocessadores. Ao longo de mais de duas décadas, várias iniciativas de pesquisa abordaram a exploração dessa característica por aplicações paralelas, permitindo o desenvolvimento de sistemas adaptativos que podem reconfigurar o número de processos/threads e seu mapeamento para processadores para lidar com cargas de trabalho variáveis e mudanças no disponibilidade de recursos no sistema. Nesse contexto, o objetivo deste documento é apresentar o estado da arte sobre adaptabilidade em pontos de vista de recursos e aplicações que vão desde arquiteturas de memória compartilhada, clusters e grades, até recursos baseados em névoa e nuvem. Uma análise abrangente das principais iniciativas de pesquisa em aplicações paralelas adaptativas pode fornecer aos leitores compreensão dos conceitos essenciais sobre a evolução da área. Além disso, esta revisão pode ajudar estudantes e pesquisadores a identificar as lacunas essenciais e oportunidades de tendências nesta área de investigação.

1.1. Introdução

A evolução das arquiteturas paralelas aponta para ambientes dinâmicos nos quais o número de recursos ou configurações disponíveis pode variar durante a execução das aplicações.

Essa variação de recursos pode ser observada em arquiteturas como grades computacionais e nuvens, assim como no nível de arquiteturas com múltiplos núcleos. Além disso, esses aspectos dinâmicos também podem aparecer no contexto das aplicações, nas quais processos (ou threads) obtêm cargas de trabalho desbalanceadas ou não uniformes ao longo do tempo. O mesmo pode acontecer com a comunicação de rede e acesso à memória, onde os padrões de uso podem mudar com o tempo, sendo muitas vezes dependente de arquivos de entrada ou entradas do usuário. Neste contexto, o desenvolvimento aplicações paralelas precisa acompanhar esta evolução e as novas tendências [Cera 2011].

[Feitelson and Rudolph 1996] classificam as aplicações paralelas de acordo com seu nível de adaptabilidade, categorizando-as em *rígidas, moldáveis, evolutivas e maleáveis*. Nas aplicações *Rígidas*, o número de recursos permanece constante ao longo da execução. Assim, eles devem ser executados usando nem mais nem menos recursos do que o especificado. As aplicações *moldáveis* têm alguma flexibilidade em termos dos recursos necessários. Os recursos são definidos no início da execução e a aplicação se adapta a esses recursos. Por sua vez, nas aplicações *evolutivas* e *maleáveis*, a quantidade de recursos pode variar durante as execuções. A diferença entre essas duas classes é que, no caso de aplicações *evolutivas*, a reconfiguração dos recursos começa a partir da própria aplicação. Em contraste, nas *maleáveis*, os recursos são controlados por um sistema subjacente (por exemplo, planejador, gerenciador externo ou tempo de execução). Considerando que a distinção entre aplicações evolutivas e maleáveis pode ser às vezes imprecisa, o termo *aplicações adaptativas* é comumente usado na literatura para se referir às duas classes acima mencionadas [Kalé et al. 2002].

Permitir que aplicações paralelas operem com alocação dinâmica de recursos pode apresentar vários benefícios. Dentre eles, destacam-se as melhorias no desempenho e na eficiência de aplicações [Galante and da Rosa Righi 2017]. Paralelamente a essas métricas, o custo e a tolerância a falhas também podem ser considerados na abordagem de aplicações de HPC dinâmicas. Por exemplo, um aplicação pode iniciar com uma quantidade padrão de recursos e, durante sua execução, novos recursos podem ser alocados a ela de acordo com suas demandas. Quando os recursos não são mais necessários, eles podem ser liberados, disponibilizando-os para outras tarefas na fila (em um cluster ou nuvem). Esses mecanismos de alocação flexível também podem melhorar o rendimento do sistema, o tempo de espera e o tempo de resposta de outras tarefas [Sudarsan and Ribbens 2007]. A adaptabilidade também é adequada para aplicações cujos requisitos de recursos não podem ser determinados com precisão com antecedência e no balanceamento de carga [Prabhakaran et al. 2014]. Outro exemplo pode ser apresentado no contexto da computação em nuvem e névoa, em que a alocação dinâmica de recursos é chamada de *elasticidade* [Galante and Bona 2012, Herbst et al. 2013]. As aplicações podem usar várias máquinas virtuais (VMs) ou contêineres que são instanciados ou redimensionados dinamicamente para moldar-se aos requisitos da aplicação por meio do modelo de pagamento pelo uso. Observando esses cenários, os benefícios de adaptabilidade para a área de computação paralela são claros. De acordo com *The International Exascale Software Project Roadmap* [Dongarra et al. 2011], o provisionamento dinâmico de recursos é considerado um aspecto essencial para a exploração da computação em exaescala.

Neste contexto, propõem-se uma revisão do estado da arte no âmbito das aplicações paralelas adaptativas. Mais especificamente, apresenta-se como a adaptabilidade pode ser explorada em aplicações e arquiteturas com vários núcleos de processamento, clusters e grades, incluindo também computação na névoa e em nuvem. Essa análise abrangente das principais iniciativas de pesquisa em aplicações paralelas adaptativas pode fornecer aos leitores uma compreensão dos conceitos essenciais da evolução da área. Além disso, essa pesquisa ajudará indivíduos e pesquisadores (experientes ou não) a identificar as questões cruciais e as lacunas para uma pesquisa mais aprofundada.

1.2. Aplicações Adaptativas: Estado da Arte

Esta seção tem o objetivo de apresentar e discutir as soluções propostas no estado da arte para o desenvolvimento de aplicações adaptativas em diferentes arquiteturas paralelas.

1.2.1. Arquiteturas de Memória Compartilhada

Em um sistema multiprocessador de memória compartilhada, vários processadores (ou núcleos) acessam um único espaço de endereçamento de memória. Esta é uma configuração muito conveniente do ponto de vista da programação, já que os dados gerados por um processador e armazenados na memória principal são imediatamente acessíveis pelos outros processadores/núcleos [Kale 2020]. Uma aplicação paralela executada em uma arquitetura de memória compartilhada geralmente usa várias threads. Um programa *multithread* pode ser escrito em diferentes linguagens de programação usando muitas bibliotecas e frameworks diferentes, que devem fornecer instruções para iniciar threads, atribuir trabalho a elas e coordenar seus acessos aos dados. Exemplos de bibliotecas e frameworks populares incluem Pthreads, OpenMP, CILK e Threading Building Blocks.

Normalmente, o número de threads usado é fornecido na execução da aplicação por meio de um parâmetro de linha de comando ou uma variável de ambiente, e permanece durante seu tempo de vida. A adaptabilidade geralmente não é explorada [Grelck 2015]. No entanto, as aplicações multithread podem ser convertidas ou implementadas como aplicações adaptativas para obter algum benefício. Nesse sentido, alguns trabalhos abordam essa questão na literatura.

[Hungershöfer and Wierum 2002] apresentam um gerenciador de paralelismo da aplicação (*application parallelism manager* - APM) para determinar a atribuição dinâmica de processadores a tarefas com base no desempenho estimado. As arquiteturas alvo são SMPs multiusuário executando várias aplicações simultaneamente que podem enfrentar conflitos de recursos entre os usuários. O objetivo geral é garantir que o sistema não fique ocioso nem sobrecarregado. O APM é implementado como um único servidor que coleta informações de aplicações, calcula uma distribuição de recursos adequada e envia recursos atribuídos a cada aplicação conectada. Uma biblioteca com métodos de comunicação deve ser vinculada à aplicação para permitir a comunicação com o servidor via sockets TCP. Presume-se que os aplicativos são maleáveis.

O trabalho de [Utrera et al. 2004] objetiva fornecer maleabilidade a programas MPI em multiprocessadores. Os autores apresentam uma técnica chamada *Folding by JobType* (FJT), na qual os jobs paralelos são executados com uma pequena quantidade de recursos (*folded* - dobrados) e são expandidos quando os recursos ficam disponíveis.

As operações de dobramento são gerenciadas pelo runtime do sistema, sendo transparente para as aplicações. [Suleman et al. 2008] propõem o Feedback-Driven Threading (FDT), um framework para controlar o número de threads usando informações coletadas em tempo de execução. O FDT amostra uma pequena fração dos kernels paralelizados para estimar o comportamento da aplicação (sincronização de dados e saturamento do barramento off-chip). Com base nessas informações, ele estima o número de threads nos quais o desempenho do kernel satura o barramento, e então, ajusta o número de threads da aplicação.

Em sua tese, [McFarland 2011] descreve o sistema RSM, que suporta aplicações OpenMP em plataformas de memória compartilhada. O RSM inclui um runtime que coleta informações de desempenho e toma decisões de redimensionamento, e uma biblioteca de comunicação para permitir que as aplicações se comuniquem com o daemon RSM para realizar o redimensionamento de recursos. Além desse trabalho, [Gordon and Lu 2011] apresentam o Elastic Phoenix, uma implementação MapReduce para sistemas de memória compartilhada com suporte a trabalhos maleáveis. A estrutura permite que threads sejam adicionados ou removidos durante a execução de uma tarefa. Com o Elastic Phoenix, se mais recursos estiverem disponíveis (como em um sistema de computação em nuvem elástica), eles poderão ser adicionados dinamicamente a uma tarefa existente. Se os recursos forem necessários para uma outra aplicação, eles também poderão ser removidos de uma tarefa existente.

[Galante and Bona 2014] propõem o Elastic OpenMP, um mecanismo para fornecer alocação dinâmica de processadores virtuais (VCPUs) para aplicações OpenMP executadas em máquinas virtuais com múltiplos núcleos. A API OpenMP original foi estendida recebendo modificações para permitir o provisionamento dinâmico de recursos. As extensões de alocação dinâmica de recursos são baseadas em adaptações de diretivas OpenMP e na adição de um conjunto de rotinas de nível de usuário à API OpenMP. A Figura 1.1 ilustra a interação API-Runtime-Cloud. Pode-se observar no código fonte um diretiva *parallel* que cria as threads e solicita a alocação das VCPUs, enviando uma mensagem para o runtime. O runtime trata a mensagem e interage com a plataforma de nuvem solicitando os recursos por meio de uma interface fornecida pelo provedor de nuvem.

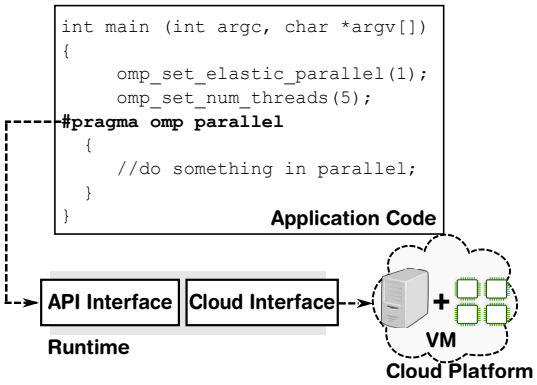


Figura 1.1. Elastic OpenMP: Interações entre a API, Runtime e Plataforma da Nuvem

[Grelck 2015] descreve um sistema de tempo de execução orientado a filas, no qual tarefas S-NET refinadas são mapeadas automaticamente para o número dinamicamente variável de threads de trabalho (ou núcleos) efetivamente disponíveis. O runtime aloca dinamicamente os recursos de execução para uma aplicação em execução e monitora continuamente a adequação dos recursos considerando as demandas do aplicativo e a carga do sistema.

[Creech 2015] apresenta o SCAF, um runtime para suportar a alocação dinâmica de recursos para aplicações maleáveis OpenMP com base na eficiência. O SCAF calcula no início de uma seção paralela a eficiência de cada aplicação para estimar a alocação de recursos. O objetivo geral é melhorar a soma de speedups das aplicações que compartilham recursos de máquina. [Georgakoudis et al. 2017] apresentam uma solução semelhante, SCALO, cuja intenção é otimizar o *throughtput* das aplicações paralelas que compartilham os recursos de um nó. O SCALO monitora as aplicações em tempo de execução para avaliar sua escalabilidade e adaptar o paralelismo de cada programa (número de threads). [Cho et al. 2018] desenvolveram o NuPoCo, um framework para gerenciar automaticamente o paralelismo de aplicações paralelos OpenMP co-alocados em sistemas NUMA. O modelo de desempenho da estrutura prevê a utilização de núcleos de CPU e memória para aplicações paralelas co-alocados; em seguida, essas informações são usadas para determinar o grau de paralelismo para todas as cargas de trabalho em execução com o objetivo de maximizar a utilização dos recursos do sistema.

Além dos esforços para fornecer runtimes e frameworks, alguns trabalhos abordam a alocação dinâmica de recursos para aplicações específicas em arquiteturas de memória compartilhada. [Pagani et al. 2016] e [Dominico et al. 2018] focam em mecanismos elásticos de alocação de núcleos de processamentos e computação para bancos de dados. [Catalán et al. 2019] introduz uma implementação maleável em nível de thread para uma biblioteca de álgebra linear (BLAS). [Libutti et al. 2020] fornecem algumas modificações no TensorFlow para introduzir maleabilidade.

1.2.2. Clusters

[Stallings 2017] define um cluster como um conjunto de computadores interconectados trabalhando juntos como um recurso de computação unificado que pode criar a ilusão de ser uma entidade única. Em um cluster, cada computador é chamado de nó. Para construir um cluster de computador, os nós individuais devem ser conectados a uma rede para permitir a comunicação entre entre esses nós. Uma camada de software pode então ser usado para unir os nós e formar um cluster.

O modelo de programação para esse sistema de memória distribuída geralmente é um modelo de troca de mensagens no qual as mensagens são usadas para a troca de informações entre os computadores. Message Passing Interface (MPI) é o método *de facto* para desenvolver aplicações em clusters computacionais. O padrão MPI original (versão 1) exigia que o número de processos fosse fixado na inicialização do programa. O suporte a processos dinâmicos foi incluído na versão 2.0 do padrão. Devido aos custos de desempenho e limitações dos recursos, o suporte a processos dinâmicos do MPI não tem sido amplamente utilizado pelos desenvolvedores de aplicações [Comprés et al. 2016].

Nesse contexto, vários esforços de pesquisa tem como foco a adaptabilidade de programas MPI.

[Huang et al. 2004] definem o Adaptive MPI (AMPI), uma implementação do MPI sobre o runtime do Charm++. AMPI implementa processos MPI como threads de nível de usuário (objetos Charm++). A maleabilidade é alcançada iniciando as aplicações com uma granularidade de processo muito fina e contando com um balanceador de carga para atribuir os objetos aos recursos físicos. Os programas AMPI recebem informações sobre a disponibilidade de processadores de um agendador de tarefas adaptável; em seguida, o runtime usa a migração de objetos para adaptar a aplicação a novos recursos.

ReSHAPE [Sudarsan and Ribbens 2007] é um framework para o desenvolvimento de aplicações MPI iterativas e maleáveis que usa dados de desempenho coletados em tempo de execução para dar suporte a ações de reconfiguração. As aplicações executadas usando o ReSHAPE podem ser expandidos para aproveitar os processadores livres ou reduzidos para acomodar uma aplicação de alta prioridade sem serem suspensos. A reconfiguração dinâmica é fornecida por um módulo de remapeamento que usa a interface de gerenciamento de processos dinâmicos do MPI-2 para gerar novos processos. O ReSHAPE é mais adequado para aplicações em que dados e processamento são distribuídos de maneira relativamente uniforme entre os processadores.

A PCM (Process Checkpointing and Migration) [El Maghraoui et al. 2009] é uma biblioteca/runtime que permite que programas SPMD MPI se reconfigurem para se adaptarem aos processadores disponíveis. A estratégia empregada adota procedimentos para dividir/mesclar processos para diminuir/aumentar a granularidade e migrar esses processos de acordo com a disponibilidade de recursos. A disponibilidade do processador é gerenciada por um agente que monitora o hardware. As ações adaptativas são realizadas sem intervenção do usuário, mas é necessário que o programador instrumente o código-fonte com primitivas PCM. [Kim et al. 2011] exploram a maleabilidade em modelos acoplados paralelos. Modelos acoplados paralelos evoluem seu estado resolvendo cada módulo de aplicação em seu respectivo conjunto de processadores (cohorts). Nesse contexto, os autores apresentam o Maleable Model Coupling Toolkit (MMCT) para construir modelos acoplados paralelos cuja configuração dos cohorts dos módulos é ajustada automaticamente em tempo de execução por meio de decisões tomadas por um gerenciador de balanceamento de carga. Recursos do MPI-2 são usados para gerenciar a criação de processos dinâmicos.

[Cera 2011] apresenta uma abordagem para fornecer maleabilidade em aplicações paralelas usando a interface de gerenciamento de processo dinâmico do MPI-2. O trabalho se concentra na criação de processos dinâmicos para fornecer adaptabilidade a processadores voláteis (aplicativos maleáveis), bem como necessidades imprevisíveis (aplicativos de tarefas explícitas). Além disso, foram descritas as interações necessárias entre as aplicações MPI e o Sistema de gerenciamento de recursos (OAR) para troca de informações sobre a disponibilidade dos processadores. Ainda na ideia de estender o modelo MPI, cita-se o Flex-MPI [Martín et al. 2015], uma extensão MPI para permitir a maleabilidade e a reconfiguração dinâmica de aplicações iterativas SPMD. O Flex-MPI reconfigura automaticamente a aplicação para ser executada com o número de processos necessários para aumentar o desempenho de modo que a aplicação seja concluída dentro

de um intervalo de tempo especificado. Um modelo de previsão computacional é usado para avaliar múltiplos cenários de reconfigurações potenciais e escolher aquele que é previsto para satisfazer melhor o objetivo de desempenho.

[Comprés et al. 2016] apresentam o framework de maleabilidade Elastic MPI. Neste trabalho, o gerenciador de recursos Slurm e a biblioteca MPICH são estendidos com novas funcionalidades para lidar com a reconfiguração de aplicações paralelas. O Slurm foi estendido para gerenciar a criação e exclusão de processos MPI enquanto lida com a alocação de recursos. Novas funções também foram adicionadas à biblioteca MPI. Quando essas funções são utilizadas, a aplicação é inicialmente definida como maleável, e seus processos verificarão periodicamente se o Slurm iniciou uma reconfiguração. [Lemarinier et al. 2016] abordam a maleabilidade usando duas abordagens diferentes: na primeira, os autores usam um mecanismo tradicional de checkpoint/restart, aproveitando a biblioteca Scalable Checkpoint/Restart for MPI (SCR), para reexecutar um trabalho com um novo número de processos após salvar seu estado. A segunda abordagem é baseada no User Level Failure Migration (ULFM), um conjunto de funções e estruturas de dados para adicionar recursos de tolerância a falhas a programas MPI existentes. As aplicações são interrompidas, o checkpoint é realizado e, em seguida, reinicia-se as aplicações com uma nova quantidade de recursos.

[Iserte et al. 2018] propõem um framework, denominado Dynamic Management of Resources (DMR), para fornecer maleabilidade ao MPI, baseando-se no MPI, OmpSs e Slurm. O DMR é implementado aproveitando o modelo de programação OmpSs, que foi estendido para lidar com reconfiguração dinâmica. Ele interage com o Slurm por meio de chamadas de API. Em seguida, o gerenciador de recursos inspeciona o status global do sistema para decidir se deve iniciar a ação de redimensionamento. O DMR expande e reduz os trabalhos rapidamente, reatribuindo os recursos subjacentes, gerando novos processos MPI, redistribuindo os dados entre os processos e retomando a execução. Posteriormente, os autores apresentam um estudo sobre o projeto, implantação e avaliação de aplicações maleáveis em GPU usando o DMR [Iserte and Rojek 2020]. O Dynamic Resource Ownership Management (DROM) [D'Amico et al. 2018] também é baseado na integração da API e Slurm. Ele fornece um canal de comunicação para permitir que o escalonador se comunique com as aplicações para adaptar os recursos de computação. Além de suportar MPI para aplicações em memória distribuída, DROM permite maleabilidade no nível do nó computacional, alterando também o número de threads que as aplicações OpenMP ou OmpSs estão usando.

Além dos esforços para desenvolver frameworks e arquiteturas com foco em MPI, alguns trabalhos abordam ainda a adaptabilidade em outros modelos e níveis de aplicação. [Batheja and Parashar 2003] desenvolveram um framework para computação em cluster adaptável e oportunista. O framework visa aplicações master-slave e é construído nas tecnologias Java e JavaSpaces. [Gupta et al. 2014] apresentam um mecanismo implementado sobre CHARM++ para permitir a maleabilidade em um runtime paralelo usando migração de tarefas, balanceamento de carga, checkpoint/restart e memória compartilhada Linux. O trabalho de [Fox et al. 2017] apresenta o Elastic-HPC (E-HPC), um framework elástico para gerenciamento de recursos para workflows científicos em sistemas HPC. A estrutura usa a checkpoint/restart como o mecanismo subjacente para migrar a execução do workflow no conjunto dinâmico de recursos.

Para tirar proveito do uso de tarefas maleáveis em ambientes de cluster compartilhado, os escalonadores e os gerenciadores de recursos também precisam ser adaptativos. Nesse contexto, alguns trabalhos abordam esse assunto. [Klein and Perez 2011] propõem o CooRMv2, um gerenciador de recursos que fornece escalonamento eficiente de aplicações maleáveis e evolutivas. [Prabhakaran et al. 2014] apresentam uma extensão do sistema Torque/Maui que permite alocações dinâmicas. O Elastic Job Bundling (EJB) é uma camada de software que opera entre usuários finais de aplicações paralelas e sistemas de batch HPC [Liu and Weissman 2015]. O objetivo do EJB é reduzir o tempo de resposta de aplicações paralelas. O EJB decompõe dinamicamente uma tarefa grande em tarefas menores para reduzir o tempo de espera e permite que a aplicação se expanda em várias subtarefas enquanto obtém progresso contínuo.

Também é possível encontrar na literatura o projeto e desenvolvimento de aplicações adaptativas em clusters. [Leopold et al. 2006] descrevem uma versão maleável do WaterGAP, uma aplicação científica que simula a disponibilidade global de água usando recursos do MPI-2. [Sudarsan et al. 2009] investigam o uso do ReSHAPE para redimensionar códigos de computação científica. Como caso de teste, o código de simulação de dinâmica molecular (MD) LAMMPS é usado. [Mo-Hellenbrand et al. 2017] apresentam um simulador de tsunami maleável usando o Elastic MPI. [Mascagni et al. 2019] descrevem o caso particular de usar a API DMR para gerar uma versão maleável do HPG, um sequenciador genômico não iterativo de memória distribuída. O DMR também é usado para incluir maleabilidade no Algoritmo de Transporte de Advecção Definida Positiva Multidimensional (MPDATA) [Iserte and Rojek 2020]. [Spenke et al. 2019] mostram uma implementação adaptativa do OGOLEM, um framework para otimização global de problemas químicos, baseado em Java-RMI.

1.2.3. Grids

Um grid, ou grade computacional, pode ser definido como um sistema unificado de memória distribuída composto por uma coleção de muitos nós/processadores distribuídos em uma ampla área geográfica com o objetivo de obter computação de alto desempenho e compartilhamento de recursos [Wilkinson 2009]. Considerando que a Internet ou redes de baixa velocidade acoplam essas infraestruturas distribuídas, as grades são mais adequadas para problemas paralelos de granularidade grossa, compostos principalmente por tarefas independentes, que são gerenciadas e executadas por meio de sistemas de workflow ou ambientes de programação paralela [Foster et al. 2008]. Grids são sistemas dinâmicos que podem acomodar nós que entram e saem ao longo do tempo. Assim, é altamente desejável reconfigurar as aplicações em execução em resposta às mudanças no ambiente (deliberadas ou causadas por falhas).

No framework GrADS [Kennedy et al. 2002], cada aplicação tem um gerenciador que monitora o desempenho para alcançar a QoS desejada. O componente Monitor de Contrato identifica violações e inicia o reescalonamento da aplicação (checkpoint/restart). As ações de reecalonamento incluem substituir recursos específicos, redistribuir as tarefas da aplicação nos recursos atuais e adicionar ou remover recursos. Além do referido trabalho, [Vadhiyar and Dongarra 2003] apresentam uma biblioteca de checkpoint chamada SRS (Stop Restart Software) e um Runtime Support System (RSS), visando aplicações MPI iterativas. A maleabilidade é implementada por um mecanismo de checkpoint/restart

que permite que uma aplicação seja verificada e, em seguida, reiniciada com um número diferente de processadores. Nesse trabalho, o objetivo da maleabilidade é garantir tolerância a falhas para aplicações executando em uma grade e ainda permitir o uso de recursos livres adicionais.

Outra solução baseada em checkpoint/restart é o Performance Control System (PerCo) [Mayes et al. 2005]. O sistema monitora o progresso das aplicações e as realoca para otimizar o desempenho. O alvo são aplicações fracamente acopladas que podem ser realocadas e reiniciadas com um número diferente de processos. Ainda no cenário de tolerância a falhas, [Wrzesinska et al. 2005] apresentam um sistema que suporta tolerância a falhas, maleabilidade e migração para aplicações de divisão e conquista. Os mecanismos são implementados usando Satin, um sistema de divisão e conquista baseado em Java projetado para ambientes de grade [Van Nieuwpoort et al. 2010]. No Satin, o trabalho é distribuído entre os processadores por meio de roubo de trabalho (*workstealing*). Adicionar uma nova máquina implica roubar trabalhos de outras máquinas. Quando uma máquina é removida ou para, os trabalhos são colocados de volta nas filas de trabalho para que eventualmente sejam recomputados. O Satin também pode adaptar aplicativos automaticamente para reagir a mudanças nas condições de recursos, como CPUs sobrecarregadas ou links de comunicação lentos.

No ambiente ASSIST [Aldinucci et al. 2006] aplicações paralelas são implementadas usando parmods (módulos paralelos). Quando um parmod é executado, um processo de monitoramento é executado para coletar informações de desempenho. Se um desempenho insatisfatório for detectado, um processo gerenciador é informado e uma ação corretiva é planejada. Neste caso, a política que pode ser adotada requer o incremento do número de recursos empregados na execução do parmod. Assim, o parmod é parado assim que atinge um ponto de sincronização e é reiniciado usando a adição de recursos. Além da iniciativa mencionada, o Dynaco (Dynamic Adaptation for Components) é um framework que ajuda no desenvolvimento e implementação de componentes dinamicamente adaptáveis [Buisson et al. 2007]. Os aplicativos Dynaco realizam adaptações baseadas em eventos, de acordo com a dinamicidade dos processadores. Quando uma aplicação observa que novos processadores estão disponíveis, ela pode aumentar o grau de paralelismo gerando e inicializando novos processos e redistribuindo dados.

[Klemm et al. 2009] apresentam uma abordagem para paralelizar e migrar aplicações OpenMP (mais precisamente, OpenMP para Java) entre clusters de uma grade. Um usuário inicia uma aplicação em um cluster arbitrário na grade. Um ponto de verificação é criado quando a fatia de tempo (alocada nesse cluster) está prestes a ser excedida. A aplicação pode migrar para outro cluster ou reiniciar o sistema atual com uma nova reserva. A paralelização permite que a próxima reserva de recurso local solicite menos ou mais CPUs, dependendo da carga geral do sistema. Assim, a aplicação deve explorar totalmente os recursos livres e adaptar as threads OpenMP aos elementos de processamento disponíveis.

[Ribeiro et al. 2013] estendem o EasyGrid de modo de transformar aplicativos MPI moldáveis tradicionais em versões autonômicas maleáveis. A aplicação maleável EasyGrid é capaz de, em alguns pontos de reconfiguração, ajustar automaticamente sua granularidade e alocação de processo de acordo com o poder computacional disponível.

1.2.4. Computação em Nuvem

No âmbito da nuvem, a elasticidade aparece como principal característica para auxiliar no desempenho de aplicações HPC. A Figura 1.2 ilustra os dois mecanismos de funcionamento: (a) é mais simples e usado principalmente em provedores de nuvem pública como Amazon e Google; (b) precisa de modelos de previsão para entender antecipadamente quando os limites podem ser ultrapassados. Além disso, em (b), a temática de reconhecimento de padrões pode ser usada para detectar o comportamento da aplicação HPC, iniciando novos contêineres ou máquinas virtuais de forma que sejam totalmente entregues e integrados antes de haver qualquer problema de carregamento da aplicação.

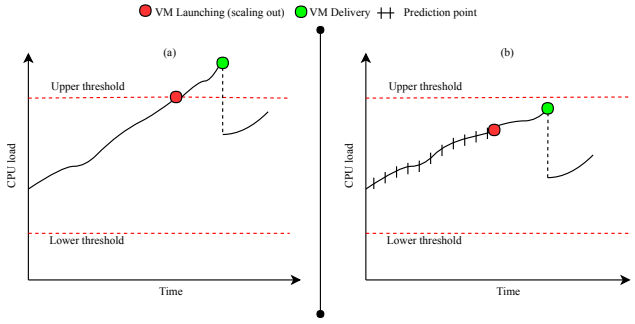


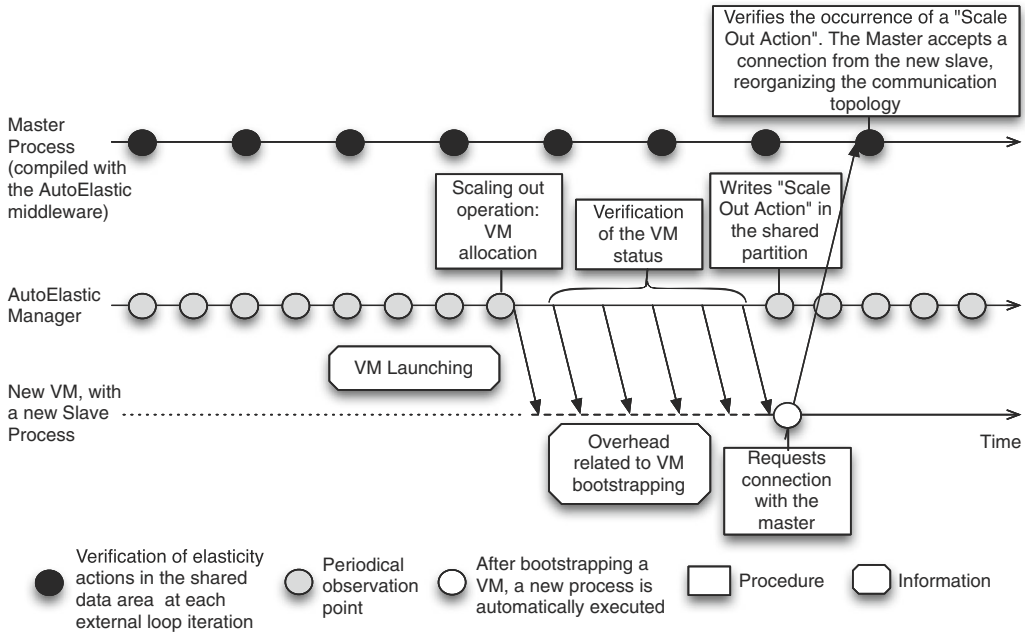
Figura 1.2. Tratamento da elasticidade em duas possibilidades: (a) reativa, uma vez que uma ação é tomada quando um threshold predefinido é atingido; (b) proativa, usando dados de predição para antecipar eventuais problemas ou irregularidades quando executando aplicações HPC.

Em [Kehrer and Blochinger 2020], os autores abordam os desafios de elasticidade para aplicações de pesquisa em árvore paralela, fornecendo um modelo denominado Equilibrium. Os autores mostram que aplicações como branch-and-bound e backtracking search não são executados de forma otimizada com recursos rígidos, de modo que a elasticidade pode fornecer desempenho e eficiência. Eles discutem como construir um controlador de elasticidade que raciocina o comportamento de dimensionamento de um sistema paralelo em tempo de execução e adapta dinamicamente o número de unidades de processamento de acordo com o custo definido pelo usuário e os limites de eficiência. Os autores reconhecem que uma compreensão detalhada do comportamento de dimensionamento de uma aplicação é uma questão fundamental sobre o qual os mecanismos de controle de elasticidade devem ser construídos. Eles se concentraram no modelo de pool de tarefas distribuídas, onde a adaptabilidade da aplicação é direta, ou seja, eles não precisam ser reestruturados para se beneficiar com a elasticidade dos recursos da nuvem.

[Rodrigues et al. 2018, Rodrigues et al. 2017] descrevem o SelfElastic, uma solução que combina abordagens de elasticidade pró-ativa e reativa para executar aplicações HPC. Eles argumentam que temos pelo menos um problema ao usar apenas uma abordagem ativa ou proativa: a necessidade de uma experiência anterior do usuário, falta de manipulação de picos de carga, conclusão de parâmetros ou design para infraestrutura específica e conjunto de carga de trabalho. O SelfElastic apresenta uma arquitetura de elasticidade de loop de controle fechado que adapta os valores dos limites inferior e superior em tempo de execução como um modelo sem parâmetros. O SelfElastic oferece

elasticidade híbrida por meio da técnica Live Thresholding, portanto, valores de limite de auto-organização e alocação de recursos para oferecer uma solução competitiva em níveis de desempenho e custo.

[da Rosa Righi et al. 2016] apresentam o AutoElastic, um modelo de elasticidade de nível de PaaS para HPC na nuvem. Sua abordagem diferencial consiste em fornecer elasticidade para aplicações de alto desempenho sem intervenção do usuário ou modificação do código-fonte. As contribuições científicas do AutoElastic são: (i) uma abordagem baseada no envelhecimento para alocação de recursos e ações de desalocação para evitar reconfigurações desnecessárias de VMs (*thrashing*) e (ii) assincronismo na criação e encerramento de VMs de forma que a aplicação não precisa aguardar a conclusão desses procedimentos. Em (i), os autores usaram a técnica de média móvel exponencial para suavizar os dados de carga da CPU rada, desencadeando ações de reorganização de recursos mais decisivas. Em (ii), eles fornecem um protocolo de interação entre o processo mestre e o gerenciador de elasticidade (ver Figura 1.3). O gerenciador informa o processo mestre quando novos recursos estão disponíveis. Além disso, ao detectar uma situação de subcarga, o gerenciador informa o IP de uma máquina virtual ao processo mestre, que desconecta os escravos dessa VM da aplicação. Depois disso, o gerenciador está livre para consolidar a VM de destino, matando todos os escravos que executam nela.



/2/2

Figura 1.3. Elasticidade assíncrona através da orquestração de aplicação (com o processo mestre) e o gerente de elasticidade. Ações de elasticidade podem ser realizadas ser que seja necessária a parada da aplicação até a entrega de novos recursos.

Historicamente, os projetistas de aplicações ajustam aplicações simultâneas para hardware e plataformas específicos. No entanto, essas abordagens não são viáveis em plataformas de nuvem, pois as aplicações podem ser implantadas em várias plataformas e os ambientes operacionais podem variar em cada implantação. Nesse contexto,

[Rajan and Thain 2017] argumentam e demonstram que aplicações concorrentes em plataformas de nuvem devem ser autoajustáveis. Primeiro, eles mostram que os aplicações devem incorporar um modelo de overheads de operação. Em segundo lugar, eles mostram que as aplicações devem determinar seus requisitos de recursos e ajustar sua operação às condições operacionais usando estimativas do modelo. Para testar as ideias mencionadas, os autores construíram duas aplicações de autoajuste, E-Sort e E-MAKER. Eles demonstram sua capacidade de obter alta eficiência de custo determinando a escala correta de partições e recursos a serem usados para operação e adaptando seu comportamento de acordo com as características do ambiente implantado.

[Galante and Erpen De Bona 2015] argumentam que a maioria das soluções são ineficientes em fornecer elasticidade para aplicações científicas, uma vez que não podem considerar a estrutura interna e o comportamento das aplicações. Eles apresentam uma abordagem para explorar a elasticidade em aplicações científicas, em que o controle de elasticidade é embutido no código-fonte da aplicação e construído usando primitivas de elasticidade. Essa abordagem permite que a aplicação solicite ou libere seus recursos, levando em consideração o fluxo de execução e os requisitos de tempo de execução. Para apoiar a construção de aplicações elásticas usando a abordagem apresentada, eles desenvolveram o framework Cloudine. O Cloudine fornece todos os componentes necessários para construir e executar aplicações científicas elásticas. A eficácia do framework é demonstrada nos experimentos em que a plataforma é usada com sucesso para incluir novos recursos para aplicações existentes, estender as funcionalidades de outras estruturas de elasticidade e adicionar suporte de elasticidade a bibliotecas de programação paralela.

[Raveendran et al. 2011] argumentaram em 2011 que havia uma tendência clara de usar recursos de nuvem na comunidade científica ou de HPC, com uma atração crítica da nuvem sendo a elasticidade que ela oferece. Segundo eles, na execução de aplicações HPC em um ambiente de nuvem, será desejável explorar a elasticidade dos ambientes de nuvem e aumentar ou diminuir o número de instâncias que uma aplicação é executada durante a execução da aplicação para atender às restrições de tempo e/ou custo. Em [Raveendran et al. 2011], descreve-se um trabalho inicial com o objetivo de tornar os aplicações MPI existentes elásticos para uma estrutura de nuvem. Considerando as limitações das implementações MPI atualmente disponíveis, eles suportam a adaptação encerrando uma execução e reiniciando um novo programa em várias instâncias. Os componentes de seu sistema idealizado incluem uma camada de decisão que considera as restrições de tempo e custo, uma estrutura para modificar programas MPI e suporte de tempo de execução baseado em nuvem que pode permitir a redistribuição de dados salvos e apoiar a alocação automatizada de recursos e reinicialização de aplicações em um número diferente de nós.

1.2.5. Computação em Névoa

Observa-se que a computação em névoa é amplamente explorada no escopo IoT (Internet of Things), onde o desempenho é visto na maioria das vezes como redução da latência da rede. Os serviços essenciais podem ser colocados na névoa para fornecer uma resposta pontual aos aplicações do usuário final, em vez de acessar a nuvem para computar insights de IA. Desta forma, [Yin et al. 2018] propuseram o uso de um avaliador de solicitação como o primeiro módulo nos nós de Fog Computing, permitindo avaliar a complexidade

de uma tarefa com base em seu tempo de computação calculado para decidir onde tal tarefa devem ser endereçados para atender aos seus requisitos. O avaliador da solicitação decide se a tarefa não pode ser endereçada localmente no dispositivo de borda. Caso contrário, eles são mapeados para a névoa ou nuvem. O escalonador de tarefas avalia a demanda de recursos de uma tarefa e o período de névoa é o limite de recursos calculado para determinar a alocação. Seu estudo se concentrou na alocação de recursos adequada para aderência de QoS e comunicação reduzida para servidores remotos, minimizando os riscos de congestionamento de rede. [Naha et al. 2020] usou uma abordagem de classificação de recursos - com base no processamento disponível, latência e largura de banda - para alocação dinâmica de recursos em arquiteturas Cloud-Fog como um meio de obter tempos de resposta ideais. O método de provisionamento de recursos incluiu a transmissão de dados como uma métrica para a decisão de provisionamento e uma abordagem hierárquica para minimizar a latência, os tempos de resposta e o congestionamento da rede priorizando os membros mais baixos da hierarquia sempre que possível.

[Chen et al. 2017], por outro lado, propôs uma arquitetura de quatro camadas para lidar com aplicações HPC baseados em IoT - consistindo em IoT, Middleware, Fog e Cloud - onde a maioria dos serviços são fornecidos pela nuvem inicialmente, e o mesmo é alocado para a névoa sob demanda. O middleware é responsável por receber tarefas de dispositivos IoT e realizar a classificação de trabalhos e agendamento de recursos em seu trabalho. A classificação do trabalho é baseada na privacidade de dados e nos requisitos de QoS. Tarefas sujeitas à privacidade de dados são atribuídas diretamente a um nó Fog local para garantir a segurança. Em contraste, tarefas não sensíveis à privacidade têm seus requisitos de QoS avaliados para determinar quais nós do sistema podem atender a QoS. Com base nos resultados da classificação, o middleware escalona tarefas para a Fog ou Cloud com base em uma avaliação de custo operacional realizada pelo agendador de recursos usando custos predefinidos. [Small et al. 2017] apresenta uma solução de middleware para orquestração baseada em microsserviços de aplicações em infraestruturas de IoT multicamadas como um meio de permitir a comunicação reduzida com servidores remotos. Sua solução orquestra a implantação de serviços nas camadas Cloud, Fog ou Mist com base nos requisitos dos serviços e nos recursos disponíveis nas camadas. A camada de nuvem usa VMs hospedadas em OpenStack, tendo a capacidade de inicializar VMs antecipadamente, reduzindo o tempo de implantação. Ao mesmo tempo, os serviços na névoa são fornecidos por contêineres instanciados sob demanda.

Muitos autores propõem diferentes tipos de configurações de hardware para permitir diferentes casos de uso do Fog até hoje. [He et al. 2018] descreve um modelo de arquitetura Fog multicamadas para permitir que tarefas analíticas complexas ocorram no nível Fog e reduzir a carga de solicitações direcionadas aos servidores Cloud para cenários de analítica de dados. O modelo multicamadas proposto é composto por duas camadas de nós Fog, A-Fog (ad-hoc) e D-Fog (dedicado), que operam em tarefas analíticas com base em sua complexidade [He et al. 2018]. O A-Fog era composto por dispositivos de baixo poder de computação, enquanto o D-Fog era formado por um cluster de servidores, permitindo a execução de análises de dados complexas sem um servidor em nuvem. A alocação de recursos é baseada no custo da utilidade como uma medida de decisão, e não na aderência ao QoS, como em alguns estudos anteriores.

Em vez de focar na alocação de recursos em Fog ou Cloud, Alsaffar propôs uma estratégia de colaboração entre Fog Computing e Cloud Computing para permitir a execução de tarefas mais complexas [Choi et al. 2016]. Em seu trabalho, um broker de Fog avalia as solicitações recebidas. Se seu nó não pude processá-lo dentro de seus SLAs, ele contata um servidor monitor de serviço localizado na nuvem, que verifica a disponibilidade da VM em todos os ambientes Cloud e Fog. Com base na disponibilidade de informações recuperadas do servidor do monitor de serviço, o intermediário Fog divide o trabalho em vários blocos para distribuição nas VMs disponíveis. Após a conclusão do processamento, cada VM remota retorna os dados para agregação ao intermediário Fog inicial.

[Al-khafajiy et al. 2019] propõem um sistema de colaboração Fog-2-Fog para permitir uma melhor utilização de recursos e distribuição de carga entre os nós Fog. O estudo projetou algoritmos para decisões de realocação de recursos usando múltiplas métricas de atraso, como atraso de serviço, atraso de propagação e atraso computacional [Al-khafajiy et al. 2019]. A decisão de quando descarregar o trabalho de um nó Fog para outro foi baseada em duas condições: 1. Avaliação se um ou mais serviços na fila perderiam seu prazo; 2. Comparação da taxa de chegada de serviço com a saída de serviço no nó. Nguyen et al. [Nguyen et al. 2020] aponta que meramente monitorar as métricas do servidor, como carga de CPU e consumo de memória, não é uma solução apropriada para lidar com a heterogeneidade de aplicações compreendidos pela IoT. Nessa nota, Nguyen propôs ElasticFog, uma estrutura construída em cima do Kubernetes para alocação dinâmica de recursos de aplicações baseados em contêiner em Fog Computing. ElasticFog monitora o tráfego de rede em cada local do nó Fog e usa isso como uma regra de afinidade no Kubernetes para melhorar as decisões de alocação de recursos. A avaliação da solução mostrou melhorias significativas na taxa de transferência e latência em favor do ElasticFog em comparação com o mecanismo de escalonamento padrão do Kubernetes. A Figura 1.4 ilustra essa arquitetura, onde pode-se ampliar o número de recursos de névoa de acordo com a demanda de IoT.

1.2.6. Discussão

Nesta seção, discute-se o estado da arte apresentado, abordando as principais características das soluções propostas para cada arquitetura. Inicialmente, discute-se aqui as soluções de adaptabilidade em arquiteturas de memória compartilhada. Naturalmente, com exceção de três delas, as soluções visam o multithreading, uma vez que o uso de threads é a forma trivial de explorar o paralelismo neste tipo de arquitetura. As estruturas geralmente permitem que as aplicações ajustem o número de threads para melhorar o uso de recursos, desempenho ou maior eficiência. A maioria das soluções é baseada em uma biblioteca e um Runtime/Middleware. A biblioteca fornece a interface para gerenciamento de encadeamentos e comunicação com o tempo de execução, que coleta informações sobre o uso de recursos ou execução de aplicações. Essas informações são usadas para determinar a alocação de recursos e o grau de paralelismo. Em geral, o uso desses mecanismos exige alguns ajustes no código-fonte original e, em alguns casos, nenhuma modificação é necessária, uma vez que a biblioteca modificada faz o trabalho pesado.

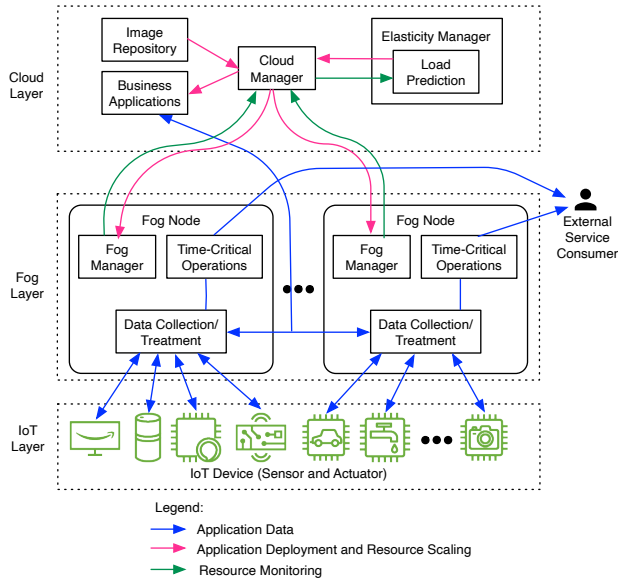


Figura 1.4. Arquitetura de fog baseada em elasticidade para tratar um alto volume de dados provenientes de dispositivos IoT.

Em relação aos esforços em prover adaptabilidade para clusters, pode-se observar um predomínio de soluções voltadas para MPI, uma vez que a maioria das aplicações que executam em clusters de computação de alto desempenho (HPC) são implementadas usando este modelo de programação. A execução de aplicações MPI dinâmicas depende do aumento ou diminuição do número de processos de acordo com as variações na disponibilidade dos recursos. As operações de maleabilidade são manipuladas por uma API que interage com um runtime/middleware para receber informações sobre a disponibilidade de recursos ou desempenho da aplicação. De acordo com essas informações, a API lança as ações de reconfiguração adequadas para adaptar a aplicação ao novo cenário. Como parte das operações crescentes, a API/runtime garante que novos processos sejam alocados nos recursos recém-alocados. Em operações de redução, as soluções devem liberar os recursos não utilizados.

Um desafio na reconfiguração dinâmica de aplicações MPI é escolher o número correto de processos/recursos a serem usados. As ações de reconfiguração só devem ser acionadas se a adição ou remoção do processo puder beneficiar o desempenho da aplicação. Para algumas classes de aplicações, aumentar o número de processadores além de um certo ponto pode não melhorar o tempo de execução (não escalável). Outro problema crítico é a necessidade de redistribuir os dados após a reconfiguração da aplicação. É fundamental, por exemplo, para aplicações fortemente acopladas, uma vez que o próprio processo apresenta altas demandas de comunicação, pode causar desbalanceamento de carga e modificar os padrões de comunicação.

No contexto da computação em grade, temos soluções que visam diferentes objetivos, como aumentar a robustez, garantir um determinado nível de QoS, melhorar o desempenho, fornecer tolerância a falhas e uso eficiente de recursos. Em termos de mecanismos usados, a maioria das soluções é baseada em técnicas de checkpoint/restart. As

aplicações são verificadas, reconfiguradas, reiniciadas e reescalonadas em um conjunto diferente de recursos. A principal vantagem do uso de checkpoints é que é uma técnica muito geral que pode ser aplicada a aplicações paralelas. As soluções propostas não são transparentes, exigindo algum esforço de programação, seja para adicionar anotações ou modificar o código-fonte para instrumentar as ações de adaptabilidade.

Ao abordar a adaptabilidade em ambientes de nuvem, observa-se que as estratégias de elasticidade de recursos e balanceamento de carga estão se integrando ao lidar com o desempenho. Em particular, o principal desafio aqui consiste em alterar o código da aplicação, que geralmente é escrito em MPI. O MPI foi proposto na década de 90 e hoje temos uma variedade de aplicações que usam essa API. A ideia de inserir elasticidade e chamadas de API dentro de um código MPI não é trivial, pois o desenvolvedor deve ter conhecimento profundo sobre a aplicação e das funcionalidades da nuvem. Por outro lado, observamos que cada vez mais iniciativas de HPC baseadas em nuvem estão explorando a elasticidade sem esforço (do ponto de vista do usuário). Aqui, o AutoElastic e o SelfElastic são sistemas que inserem elasticidade no momento da compilação, livrando o usuário de qualquer decisão sobre a reorganização de recursos e processos. Além disso, outro tópico importante é o comportamento da aplicação quando novos recursos estão disponíveis. Normalmente, temos a abordagem stop-reconfigure-and-go para usar mais recursos. Embora essa abordagem possa ser eficiente para as demandas corporativas, ela não se encaixa nas demandas de HPC. Nesses casos, a proposta de elasticidade assíncrona aparece como uma solução que orquestra o gerenciador de nuvem e a aplicação para evitar que a aplicação execute ações de elasticidade de recursos.

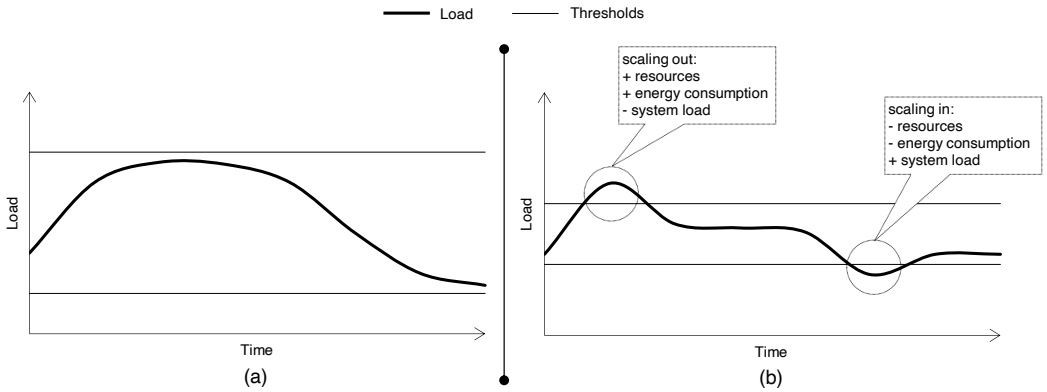
Por fim, no escopo da computação em névoa, percebe-se que a maioria dos trabalhos está focada nas demandas de IoT. Sensores e atuadores IoT são conectados aos recursos de névoa, que são então conectados à nuvem. As aplicações são então conectadas diretamente à névoa para obter resultados mais rapidamente e à nuvem para obter insights de IA mais robustos. A computação de alto desempenho é necessária para fornecer respostas oportunas às aplicações e atuadores IoT. As demandas vêm do número de sensores e atuadores, além do número de aplicações conectadas. Portanto, as preocupações com a escalabilidade são essenciais e percebe-se que a adaptação é tratada usando a elasticidade dos recursos nas camadas de névoa e nuvem para abordar essa temática. Além do desempenho, a elasticidade também é pertinente para reduzir os custos financeiros relacionados à manutenção de contêineres ou máquinas virtuais ligadas, já que seu número corresponde às demandas de um determinado momento.

1.3. Perspectivas e Tendências

Levando em consideração o atual estado da arte, apresenta-se a seguir algumas tendências futuras e oportunidades de pesquisa:

- Soluções leves baseadas em IA - Prevê-se o uso de mais e mais soluções de IA nas soluções de escalonamento e balanceamento de carga de forma que a previsão, o reconhecimento de padrões, a classificação instantânea e a correlação de eventos possam ajudar a otimizar as decisões de desempenho. O desafio aqui consiste em abordar procedimentos demorados relacionados à IA. Assim, é possível usar a ideia de treinar uma vez, usar várias vezes e a combinação de IA com abordagens heurísticas

rápidas. Os artigos recentes publicados por Jiang et al. [Jiang et al. 2021] e Yadav et al. [Yadav et al. 2021] confirmam essa tendência, em que técnicas de aprendizado de máquina são usadas para fornecer provisionamento de recursos eficiente. A Figura 1.5 mostra um exemplo em que podemos usar soluções de IA para combinar vários objetivos na execução de aplicações HPC em arquiteturas de névoa e nuvem.



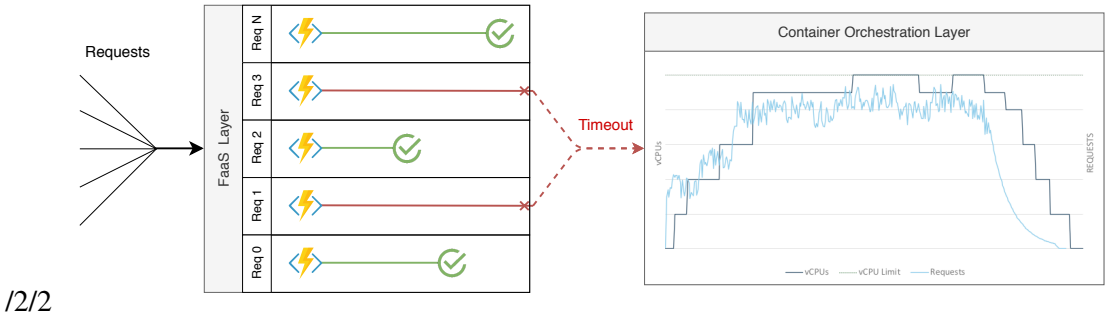
/2/2

Figura 1.5. Possibilidade de usar soluções baseadas em IA para melhorar não só o desempenho, mas também a energia e o orçamento na execução de aplicações HPC na nuvem. Em (a), temos uma aplicação que executa dentro de limites predefinidos. Em (b), temos o controle baseado em IA para manter a aplicação dentro dos limites levando em consideração diversos critérios objetivos.

- Combinação de metodologias de elasticidade de nuvem - É possível combinar elasticidade vertical e horizontal em arquiteturas de névoa e nuvem. Primeiramente, a vertical pode ser usada até o limite de uma máquina física. Se mais desempenho for realmente necessário, podemos usar a elasticidade horizontal para alocar mais recursos virtuais. Essa combinação pode ser interessante para o consumo de energia e redução de custos financeiros, uma vez que primeiro aloca-se as demandas a uma máquina física específica, usando-a de forma eficiente antes de alocar outras. Além disso, a combinação de Serverless ou FaaS (Function as a Service) e as abordagens tradicionais de elasticidade da nuvem podem ser interessantes para executar aplicações irregulares, uma vez que FaaS é mais adequado para tarefas de execução curta. Ao mesmo tempo, a alocação de servidores pode lidar com demandas de longa duração de uma maneira melhor, conforme ilustrado na Figura 1.6.

- Uso eficiente de arquiteturas heterogêneas - Com tecnologias de aceleradores (por exemplo, GPU e FPGA), as arquiteturas de sistema entraram em uma tendência clara para o aumento do paralelismo e da heterogeneidade. No entanto, as estruturas de alocação de recursos que visam arquiteturas heterogêneas não atingiram seu potencial total, que ocorre ao orquestrar todos os diferentes recursos juntos ou dinamicamente, selecionando a configuração de recurso mais adequada para cada aplicação (ou estágio da aplicação).

- Linguagens de programação promissoras - É previsto o uso de linguagens de programação como Go e Elixir/Erlang como uma tendência HPC. Essas linguagens possuem recursos integrados para lidar com computação de alto desempenho, incluindo



/2/2

Figura 1.6. Combinação de Function as a Service (FaaS) e elasticidade tradicional reativa para executar aplicações HPC.

diretivas de exclusão mútua, mecanismos eficientes de envio/recebimento de dados, replicação de dados, suporte de escalabilidade e compatibilidade de arquiteturas baseadas em memória distribuída e compartilhada. Além disso, observamos o uso crescente dessas linguagens de programação com ferramentas de orquestração de contêineres, como K3S e Kubernetes, para lidar com propostas eficientes de programação e elasticidade.

- Exploração da computação de alto desempenho de forma automática - Como tendência, observa-se a exploração de abordagens automáticas para impulsionar aplicações HPC já desenvolvidas a executar ainda mais rápido. A ideia aqui é explorar bibliotecas de programação e compiladores que alteram o código de funções para inserir transparentemente nas diretivas de gerenciamento de recursos do ponto de vista do usuário. Assim, é possível transformar uma aplicação não elástica MPI em uma elástica, apenas computando-a com uma biblioteca baseada em elasticidade MPI particular, permitindo que um código explore os benefícios da computação em nuvem.

- Mecanismos de redistribuição de dados - O desafio de projetar técnicas de adaptabilidade para algumas classes de aplicações (por exemplo, SPMD e decomposição de domínio) não é simplesmente modificar o número de processos em que a aplicação está rodando de acordo com a disponibilidade de recursos. As ações de reconfiguração envolvem redistribuir os dados pelos novos processos (o que pode causar desbalanceamento de carga) e modificar os padrões de comunicação. Nesse sentido, mecanismos de redistribuição de dados transparentes para o usuário são necessários para permitir o uso eficiente de recursos dinâmicos por uma ampla classe de aplicações.

1.4. Conclusão

Este minicurso apresentou como o estado da arte está tratando da adaptabilidade em aplicações paralelos nas últimas duas décadas. Para os próximos anos, prevê-se o uso crescente da computação em nuvem para executar demandas de HPC, permitindo o uso de elasticidade e a alocação de hardware específico, incluindo requisitos de cache e dispositivos de GPU, definições de QoS e especificações de bibliotecas e dependências. Além

disso, ressalta-se dois tópicos importantes para os próximos anos. Em primeiro lugar, a exploração da elasticidade sem esforço, automática e transparente são os principais motivadores para explorar HPC em arquiteturas virtualizadas, permitindo que os desenvolvedores executem suas demandas mais rapidamente com mudanças mínimas de código. Em segundo lugar, alinhado às tendências do Top500.org, se tornará necessário cada vez mais que programas extraiam o poder simultâneo de múltiplas arquiteturas. Assim, novos middleware e bibliotecas devem permitir que um código possa ser executado, de forma automática, em uma gama de arquiteturas (multicomputadores, multiprocessadores e aceleradores, por exemplo) com o mínimo de esforço.

Referências

[Al-khafajiy et al. 2019] Al-khafajiy, M., Baker, T., Al-Libawy, H., Maamar, Z., Aloqaily, M., and Jararweh, Y. (2019). Improving fog computing performance via fog-2-fog collaboration. *Future Generation Computer Systems*, 100:266 – 280.

[Aldinucci et al. 2006] Aldinucci, M., Coppola, M., Danelutto, M., Tonellotto, N., Vanneschi, M., and Zoccolo, C. (2006). High level grid programming with ASSIST. *Computational Methods in Science and Technology*, 12(1):21–32.

[Batheja and Parashar 2003] Batheja, J. and Parashar, M. (2003). A framework for adaptive cluster computing using javaspaces. *Cluster Computing*, 6(3):201–213.

[Buisson et al. 2007] Buisson, J., Andre, F., and Pazat, J.-L. (2007). Supporting adaptable applications in grid resource management systems. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, GRID '07, page 58–65, USA. IEEE Computer Society.

[Catalán et al. 2019] Catalán, S., Herrero, J. R., Quintana-Ortí, E. S., Rodríguez-Sánchez, R., and Van De Geijn, R. (2019). A case for malleable thread-level linear algebra libraries: The lu factorization with partial pivoting. *IEEE Access*, 7:17617–17633.

[Cera 2011] Cera, M. C. (2011). *Providing adaptability to MPI applications on current parallel architectures*. PhD thesis, Universidade Federal do Rio Grande do Sul. Instituto de Informática.Programa de Pós-Graduação em Computação.

[Chen et al. 2017] Chen, Y., Chang, Y., Chen, C., Lin, Y., Chen, J., and Chang, Y. (2017). Cloud-fog computing for information-centric internet-of-things applications. In *2017 International Conference on Applied System Innovation (ICASI)*, pages 637–640.

[Cho et al. 2018] Cho, Y., Guzman, C. A. C., and Egger, B. (2018). Maximizing system utilization via parallelism management for co-located parallel applications. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, pages 1–14, New York, NY, USA. Association for Computing Machinery.

[Choi et al. 2016] Choi, Y., Alsaffar, A. A., , Pham, H. P., Hong, C., Huh, E., and Aazam, M. (2016). An Architecture of IoT Service Delegation and Resource Allocation Based

on Collaboration between Fog and Cloud Computing. *Mobile Information Systems*, 2016.

[Comprés et al. 2016] Comprés, I., Mo-Hellenbrand, A., Gerndt, M., and Bungartz, H.-J. (2016). Infrastructure and api extensions for elastic execution of mpi applications. In *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016, page 82–97, New York, NY, USA. Association for Computing Machinery.

[Creech 2015] Creech, T. M. (2015). Efficient multiprogramming for multicores with scaf. Master's thesis, Faculty of the Graduate School of the University of Maryland.

[da Rosa Righi et al. 2016] da Rosa Righi, R., Rodrigues, V. F., da Costa, C. A., Galante, G., Bona, L. C. E. D., and Ferreto, T. C. (2016). Autoelastic: Automatic resource elasticity for high performance applications in the cloud. *IEEE Trans. Cloud Comput.*, 4(1):6–19.

[D'Amico et al. 2018] D'Amico, M., Garcia-Gasulla, M., López, V., Jokanovic, A., Sirvent, R., and Corbalan, J. (2018). Drom: Enabling efficient and effortless malleability for resource managers. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, ICPP '18, pages 1–10, New York, NY, USA. Association for Computing Machinery.

[Dominico et al. 2018] Dominico, S., de Almeida, E. C., Meira, J. A., and Alves, M. A. (2018). An elastic multi-core allocation mechanism for database systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 473–484.

[Dongarra et al. 2011] Dongarra, J., Beckman, P., Moore, T., Aerts, P., Aloisio, G., Andre, J.-C., Barkai, D., Berthou, J.-Y., Boku, T., Braunschweig, B., Cappello, F., Chapman, B., Chi, X., Choudhary, A., Dosanjh, S., Dunning, T., Fiore, S., Geist, A., Gropp, B., Harrison, R., Hereld, M., Heroux, M., Hoisie, A., Hotta, K., Jin, Z., Ishikawa, Y., Johnson, F., Kale, S., Kenway, R., Keyes, D., Kramer, B., Labarta, J., Lichnewsky, A., Lippert, T., Lucas, B., Maccabe, B., Matsuoka, S., Messina, P., Michielse, P., Mohr, B., Mueller, M. S., Nagel, W. E., Nakashima, H., Papka, M. E., Reed, D., Sato, M., Seidel, E., Shalf, J., Skinner, D., Snir, M., Sterling, T., Stevens, R., Streitz, F., Sugar, B., Sumimoto, S., Tang, W., Taylor, J., Thakur, R., Trefethen, A., Valero, M., Van Der Steen, A., Vetter, J., Williams, P., Wisniewski, R., and Yelick, K. (2011). The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60.

[El Maghraoui et al. 2009] El Maghraoui, K., Desell, T. J., Szymanski, B. K., and Varela, C. A. (2009). Malleable iterative mpi applications. *Concurrency and Computation: Practice and Experience*, 21(3):393–413.

[Feitelson and Rudolph 1996] Feitelson, D. G. and Rudolph, L. (1996). Toward convergence in job schedulers for parallel supercomputers. In Feitelson, D. G. and Rudolph, L., editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag.

[Foster et al. 2008] Foster, I., Zhao, Y., Raicu, I., and Lu, S. (2008). Cloud computing and grid computing 360-degree compared. In *2008 Grid Computing Environments Workshop*, pages 1–10. IEEE.

[Fox et al. 2017] Fox, W., Ghoshal, D., Souza, A., Rodrigo, G. P., and Ramakrishnan, L. (2017). E-hpc: A library for elastic resource management in hpc environments. In *Proceedings of the 12th Workshop on Workflows in Support of Large-Scale Science*, WORKS '17, pages 1–11, New York, NY, USA. Association for Computing Machinery.

[Galante and Bona 2012] Galante, G. and Bona, L. C. E. (2012). A survey on cloud computing elasticity. In *Proceedings of the International Workshop on Clouds and eScience Applications Management*, CloudAM'12, pages 263–270. IEEE.

[Galante and Bona 2014] Galante, G. and Bona, L. C. E. (2014). Supporting elasticity in openmp applications. In *Proceedings of the 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, PDP '14, page 188–195, USA. IEEE Computer Society.

[Galante and da Rosa Righi 2017] Galante, G. and da Rosa Righi, R. (2017). Exploring cloud elasticity in scientific applications. In Antonopoulos, N. and Gillam, L., editors, *Cloud Computing - Principles, Systems and Applications, Second Edition*, Computer Communications and Networks, pages 101–125. Springer.

[Galante and Erpen De Bona 2015] Galante, G. and Erpen De Bona, L. C. (2015). A programming-level approach for elasticizing parallel scientific applications. *Journal of Systems and Software*, 110:239–252.

[Georgakoudis et al. 2017] Georgakoudis, G., Vandierendonck, H., Thoman, P., Supinski, B. R. D., Fahringer, T., and Nikolopoulos, D. S. (2017). Scalo: Scalability-aware parallelism orchestration for multi-threaded workloads. *ACM Trans. Archit. Code Optim.*, 14(4).

[Gordon and Lu 2011] Gordon, A. W. and Lu, P. (2011). Elastic phoenix: Malleable mapreduce for shared-memory systems. In Altman, E. R. and Shi, W., editors, *Network and Parallel Computing - 8th IFIP International Conference, NPC 2011*, volume 6985 of *Lecture Notes in Computer Science*, pages 1–16. Springer.

[Grelck 2015] Grelck, C. (2015). Moldable applications on multi-core servers: Active resource management instead of passive resource administration. In *Proceedings of the 18. Kolloquium Programmiersprachen und Grundlagen der Programmierung*, KPS 2015, pages 1–10. TU Wien.

[Gupta et al. 2014] Gupta, A., Acun, B., Sarood, O., and Kalé, L. V. (2014). Towards realizing the potential of malleable jobs. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10.

[He et al. 2018] He, J., Wei, J., Chen, K., Tang, Z., Zhou, Y., and Zhang, Y. (2018). Multitier fog computing with large-scale iot data analytics for smart cities. *IEEE Internet of Things Journal*, 5(2):677–686.

[Herbst et al. 2013] Herbst, N. R., Kounev, S., and Reussner, R. (2013). Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing*, ICAC'13, pages 23–27. USENIX.

[Huang et al. 2004] Huang, C., Lawlor, O., and Kalé, L. V. (2004). Adaptive mpi. In Rauchwerger, L., editor, *Languages and Compilers for Parallel Computing*, pages 306–322, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Hungershöfer and Wierum 2002] Hungershöfer, J. and Wierum, J. (2002). On the quality of partitions based on space-filling curves. In Sloot, P. M. A., Tan, C. J. K., Dongarra, J. J., and Hoekstra, A. G., editors, *Computational Science - ICCS 2002, International Conference, Amsterdam, The Netherlands, April 21-24, 2002. Proceedings, Part III*, volume 2331 of *Lecture Notes in Computer Science*, pages 36–45. Springer.

[Iserte et al. 2018] Iserte, S., Mayo, R., Quintana-Ortí, E. S., Beltran, V., and Peña, A. J. (2018). Dmr api: Improving cluster productivity by turning applications into malleable. *Parallel Computing*, 78:54–66.

[Iserte and Rojek 2020] Iserte, S. and Rojek, K. (2020). An study of the effect of process malleability in the energy efficiency on gpu-based clusters. *J. Supercomput.*, 76(1):255–274.

[Jiang et al. 2021] Jiang, Y., Kodialam, M., Lakshman, T. V., Mukherjee, S., and Tassiulas, L. (2021). Resource allocation in data centers using fast reinforcement learning algorithms. *IEEE Transactions on Network and Service Management*.

[Kalé et al. 2002] Kalé, L. V., Kumar, S., and DeSouza, J. (2002). A malleable-job system for timeshared parallel machines. In *Proceedings of the 2Nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '02, pages 230–, Washington, DC, USA. IEEE Computer Society.

[Kale 2020] Kale, V. (2020). *Parallel computing architectures and APIs : IoT big data stream processing*. CRC Press, Taylor & Francis Group, Boca Raton, FL.

[Kehrer and Blochinger 2020] Kehrer, S. and Blochinger, W. (2020). Equilibrium: an elasticity controller for parallel tree search in the cloud. *J. Supercomput.*, 76(11):9211–9245.

[Kennedy et al. 2002] Kennedy, K., Mazina, M., Mellor-Crummey, J. M., Cooper, K. D., Torczon, L., Berman, F., Chien, A. A., Dail, H., Sievert, O., Angulo, D., Foster, I. T., Aydt, R. A., Reed, D. A., Gannon, D., Johnsson, S. L., Kesselman, C., Dongarra, J., Vadhiyar, S. S., and Wolski, R. (2002). Toward a framework for preparing and executing adaptive grid programs. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, page 322, USA. IEEE Computer Society.

[Kim et al. 2011] Kim, D., Larson, J. W., and Chiu, K. (2011). Toward malleable model coupling. *Procedia Computer Science*, 4:312–321. Proceedings of the International Conference on Computational Science, ICCS 2011.

[Klein and Perez 2011] Klein, C. and Perez, C. (2011). An rms for non-predictably evolving applications. In *2011 IEEE International Conference on Cluster Computing*, pages 326–334.

[Klemm et al. 2009] Klemm, M., Bezold, M., Gabriel, S., Veldema, R., and Philippsen, M. (2009). Reparallelization techniques for migrating openmp codes in computational grids. *Concurrency and Computation: Practice and Experience*, 21(3):281–299.

[Lemarinier et al. 2016] Lemarinier, P., Hasanov, K., Venugopal, S., and Katrinis, K. (2016). Architecting malleable mpi applications for priority-driven adaptive scheduling. In *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016, page 74–81, New York, NY, USA. Association for Computing Machinery.

[Leopold et al. 2006] Leopold, C., Süß, M., and Breitbart, J. (2006). Programming for malleability with hybrid mpi-2 and openmp: Experiences with a simulation program for global water prognosis. In *Proceedings of the European Conference on Modelling and Simulation*, pages 665–670.

[Libutti et al. 2020] Libutti, L. A., Igual, F. D., Piñuel, L., De Giusti, L., and Naiouf, M. (2020). Towards a malleable tensorflow implementation. In Rucci, E., Naiouf, M., Chichizola, F., and De Giusti, L., editors, *Cloud Computing, Big Data & Emerging Topics*, pages 30–40, Cham. Springer International Publishing.

[Liu and Weissman 2015] Liu, F. and Weissman, J. B. (2015). Elastic job bundling: An adaptive resource request strategy for large-scale parallel applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 1–12, New York, NY, USA. Association for Computing Machinery.

[Martín et al. 2015] Martín, G., Singh, D. E., Marinescu, M.-C., and Carretero, J. (2015). Enhancing the performance of malleable mpi applications by using performance-aware dynamic reconfiguration. *Parallel Comput.*, 46(C):60–77.

[Mascagni et al. 2019] Mascagni, M., Iserte, S., Martínez, H., Barrachina, S., Castillo, M., Mayo, R., and Peña, A. J. (2019). Dynamic reconfiguration of noniterative scientific applications: A case study with hpg aligner. *Int. J. High Perform. Comput. Appl.*, 33(5):804–816.

[Mayes et al. 2005] Mayes, K., Luján, M., Riley, G., Chin, J., Coveney, P., and Gurd, J. (2005). Towards performance control on the grid. *Phil. Trans. R. Soc.*, 363(1833):1793–1805.

[McFarland 2011] McFarland, D. J. (2011). Exploiting malleable parallelism on multicore systems. Master's thesis, Faculty of the Virginia Polytechnic Institute and State University.

[Mo-Hellenbrand et al. 2017] Mo-Hellenbrand, A., Comprés, I., Meister, O., Bungartz, H.-J., Gerndt, M., and Bader, M. (2017). A large-scale malleable tsunami simulation realized on an elastic mpi infrastructure. In *Proceedings of the Computing Frontiers Conference*, CF'17, page 271–274, New York, NY, USA. Association for Computing Machinery.

[Naha et al. 2020] Naha, R. K., Garg, S., Chan, A., and Battula, S. K. (2020). Deadline-based dynamic resource allocation and provisioning algorithms in fog-cloud environment. *Future Generation Computer Systems*, 104:131 – 141.

[Nguyen et al. 2020] Nguyen, N. D., Phan, L. A., Park, D. H., Kim, S., and Kim, T. (2020). Elasticfog: Elastic resource provisioning in container-based fog computing. *IEEE Access*, 8:183879–183890.

[Pagani et al. 2016] Pagani, D. H., Bona, L. C. E. D., and Galante, G. (2016). Uma abordagem baseada em níveis de estresse para alocação elástica de recursos em sistema de bancos de dados. In *Anais do XIV Workshop em Clouds e Aplicações*, WCGA 2016, pages 1–14, Porto Alegre. SBC.

[Prabhakaran et al. 2014] Prabhakaran, S., Iqbal, M., Rinke, S., Windisch, C., and Wolf, F. (2014). A batch system with fair scheduling for evolving applications. In *Proceedings of the 2014 Brazilian Conference on Intelligent Systems*, BRACIS '14, page 351–360, USA. IEEE Computer Society.

[Rajan and Thain 2017] Rajan, D. and Thain, D. (2017). Designing self-tuning split-map-merge applications for high cost-efficiency in the cloud. *IEEE Transactions on Cloud Computing*, 5(2):303–316.

[Raveendran et al. 2011] Raveendran, A., Bicer, T., and Agrawal, G. (2011). A framework for elastic execution of existing mpi programs. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 940–947.

[Ribeiro et al. 2013] Ribeiro, F., Rebello, V., Nascimento, A., Boeres, C., and Sena, A. (2013). Autonomic malleability in iterative mpi applications. In *Proceedings of the 2013 25th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '13, page 192–199, USA. IEEE Computer Society.

[Rodrigues et al. 2018] Rodrigues, V. F., da Rosa Righi, R., da Costa, C. A., Singh, D., Muñoz, V. M., and Chang, V. (2018). Towards combining reactive and proactive cloud elasticity on running HPC applications. In Muñoz, V. M., Wills, G. B., Walters, R. J., Firouzi, F., and Chang, V., editors, *Proceedings of the 3rd International Conference on Internet of Things, Big Data and Security, IoTBDS 2018*, pages 261–268. SciTePress.

[Rodrigues et al. 2017] Rodrigues, V. F., da Rosa Righi, R., Rostirolla, G., Barbosa, J. L. V., da Costa, C. A., Alberti, A. M., and Chang, V. I. (2017). Towards enabling live thresholding as utility to manage elastic master-slave applications in the cloud. *J. Grid Comput.*, 15(4):535–556.

[Small et al. 2017] Small, N., Akkermans, S., Joosen, W., and Hughes, D. (2017). Niflheim: An end-to-end middleware for applications on a multi-tier iot infrastructure. In *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, pages 1–8.

[Spenke et al. 2019] Spenke, F., Balzer, K., Frick, S., Hartke, B., and Dieterich, J. M. (2019). Malleable parallelism with minimal effort for maximal throughput and maximal hardware load. *Computational and Theoretical Chemistry*, 1151:72–77.

[Stallings 2017] Stallings, W. (2017.). *Computer organization and architecture*. Pearson Education, Inc., Hoboken, New Jersey, 10th ed. edition.

[Sudarsan and Ribbens 2007] Sudarsan, R. and Ribbens, C. J. (2007). Reshape: A framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment. In *Proceedings of the 2007 International Conference on Parallel Processing*, ICPP '07, page 44, USA. IEEE Computer Society.

[Sudarsan et al. 2009] Sudarsan, R., Ribbens, C. J., and Farkas, D. (2009). Dynamic resizing of parallel scientific simulations: A case study using lammps. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, page 175–184, Berlin, Heidelberg. Springer-Verlag.

[Suleman et al. 2008] Suleman, M. A., Qureshi, M. K., and Patt, Y. N. (2008). Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps. *SIGARCH Comput. Archit. News*, 36(1):277–286.

[Utrera et al. 2004] Utrera, G., Corbalan, J., and Labarta, J. (2004). Implementing malleability on mpi jobs. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, page 215–224, USA. IEEE Computer Society.

[Vadhiyar and Dongarra 2003] Vadhiyar, S. S. and Dongarra, J. J. (2003). Srs: A framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters*, 13(02):291–312.

[Van Nieuwpoort et al. 2010] Van Nieuwpoort, R. V., Wrzesińska, G., Jacobs, C. J. H., and Bal, H. E. (2010). Satin: A high-level and efficient grid programming model. *ACM Trans. Program. Lang. Syst.*, 32(3).

[Wilkinson 2009] Wilkinson, B. (2009). *Grid Computing: Techniques and Applications*. CRC Press, Boca Raton, FL, 1st ed. edition.

[Wrzesinska et al. 2005] Wrzesinska, G., van Nieuwpoort, R., Maassen, J., and Bal, H. (2005). Fault-tolerance, malleability and migration for divide-and-conquer applications on the grid. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 10 pp.–.

[Yadav et al. 2021] Yadav, M. P., Rohit, and Yadav, D. K. (2021). Resource provisioning through machine learning in cloud services. *Arabian Journal for Science and Engineering*.

[Yin et al. 2018] Yin, L., Luo, J., and Luo, H. (2018). Tasks scheduling and resource allocation in fog computing based on containers for smart manufacturing. *IEEE Transactions on Industrial Informatics*, 14(10):4712–4721.

Capítulo
2

High Performance Computing in Julia

Roberto Machado Velho
Sarpen Quant Investments, São Paulo, Brazil

Rafael Benchimol Klausner
PSR Energy Consulting and Analytics, Rio de Janeiro, Brazil

Matheus da Silva Serpa
Parallel and Distributed Processing Group (GPPD)
Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil

Adriano Maurício de Almeida Côrtes
Applied Mathematics Department (DMA) and
High-Performance Computing Center (NACAD)
Federal University of Rio de Janeiro, Rio de Janeiro, Brazil

Abstract

These are a set of notes designed for a short course addressing the use of the Julia Programming Language for High Performance Computing (HPC). They target the reader that has some basic knowledge of HPC and distributed systems. First goal is to show how to use Julia from zero on a typical HPC cluster environment. Second goal is showing which changes are necessary for parallelizing a code using Julia. The main purpose of the use of Julia is that it is easy to install, easy to code, and it can still deliver great computational performance on a HPC system.

2.1. Introduction

Julia is a modern language that is easy to use and to learn. It was designed to be used in scientific computing while being fast and allow multiple types of parallelization. The main goals of the notes is to enable the reader to prototyping simple parallel applications in Julia, and to run already existing complex applications, under multicore, distributed CPUs, or GPU architectures, the typical environment of nowadays clusters found in a High Performance Computing (HPC) Center. Our examples use the Process Manager SLURM.

These notes assume some familiarity of the reader with HPC systems. We will describe how easy is to prototyping a parallel code in Julia. Later, via examples, we discuss which kind of modifications are necessary when converting some application (eventually composed of multiple functions) from its serial version to a parallel one.

We start the notes by presenting Julia and how to install its binary file and its packages on a HPC system in section 2.2. We then pass to analyze some typical aspects of HPC programming, on how to measure performance of an application and how to organize code for maximum efficiency, on section 2.3. Section 2.4 discusses how the use of Julia for multicore programming, while Section 2.5 goes through distributed computing. Finally, in Section 2.6 we bring some simple analysis of speed up for the examples presented in the multicore and distributed sections, while in Section 2.7 we present graphics processing unit (GPU) programming with Julia.

Since these notes target an intermediary course on HPC, we avoided complex discussions on Input/Output (I/O) aspects in Julia. One of the reasons is the already not mature capabilities of the language to perform I/O for many forms of parallelizing a code. For more information about Parallel Computing in Julia, you can access the parallel computing documentation [1].

2.2. Basics of Julia

Our goal along this section is presenting the language Julia and how to install it and its packages on a HPC system.

2.2.1. Why Julia?

Julia is a high-level, dynamic, general-purpose programming language that is specifically aimed for numerical computing and computational science. Julia benefits from a technique of compiling in advance the necessary libraries a piece of code will run. This can bring high speed on running codes. It is also easy to programming, resembling Python or Matlab syntaxes. Its ecosystem has already a large amount of libraries for solving from basic problems in computer science to specific industrial applications.

Julia allows the use of Unicode characters, giving the programmer access to a large extension of symbols to construct variables. Among them are Greek characters, very widely used in science, in particular in mathematics, what makes the process of writing a code with variable names that reflect the scientific problem easier. To input such characters, use the Latex code and press Tab. A list of permitted characters is in here:

[1] https://docs.julialang.org/en/v1/manual/parallel-computing/

https://docs.julialang.org/en/v1/manual/unicode-input/

Julia permits calls to pieces of code of other languages. This is a great advantage on prototyping code in HPC systems, as you can call a performative code from a library in C or Fortran, and at the same time assemble the results and visualize it with the modern libraries from Julia's ecosystem. This is very beneficial for non-experts on parallel programming, that can assemble pieces of code in Julia while using previously assembled performative libraries written in other languages by experts on HPC systems and in the domain of knowledge in question, e.g., biology, chemistry, geophysics, or mathematics. There are several Julia case studies available at https://juliacomputing.com/case-studies/.

2.2.2. Installing Julia and its packages

In order to download Julia, go to the website https://julialang.org/downloads/. All code examples run in the current stable release v1.7.2 (at the time of writing). The Julia Language has a thorough documentation in https://docs.julialang.org/en/v1/. Julia can be used in an interactive command-line interface, called REPL (read-eval-print loop), by calling `julia` in the command-line.

```
$ julia

julia>
```

Also, you can run a code from a julia script.

```
$ julia script.jl
```

To download a specific package in Julia, open Julia's terminal and press the button closed square bracket (]) to go to the Package Manager (Pkg).

```
$ julia

julia>

(@v1.7) pkg>
```

To install an specific package use the keyword add:

```
(@v1.7) pkg> add Example
```

It is also possible to specify multiple packages at once.

```
(@v1.7) pkg> add JSON StaticArrays
```

To remove a package, use the keyword rm.

```
(@v1.7) pkg> rm JSON StaticArrays
```

Use update to update an installed package.

```
(@v1.7) pkg> update Example
```

To update all installed packages, use update without any arguments.

```
(@v1.7) pkg> update
```

Julia has packages environments just like Python language, but it is a lot easier to use. To activate an environment use the keyword activate.

```
(@v1.7) pkg> activate my_env
  Activating environment at `/home/user/my_env/`

(my_env) pkg>
```

The Pkg let you know which environment you are using in the brackets. To see what packages you have already installed, use status.

```
(my_env) pkg> status
  Status `/home/user/my_env/` (empty project)
```

When a environment is just started, its condition is empty. The packages metadata are stored in the `Project.toml` file in the folder `/home/user/my_env/`, so it is possible to use the same environment in different computers. This is important for two or more persons working on the same project, or a single person working on different computers, e.g., a personal laptop and the HPC system where a parallel code will run. Let us add a package.

```
(my_env) pkg> add StatsBase
...

(my_env) pkg> status
    Status `/home/user/my_env/`
  [2913bbd2] StatsBase v0.33.16
```

The environment now has `StatsBase` in its dependencies. You can start Julia with a specific project with the `--project` flag.

```
  $ julia --project=my_env

julia>
```

```
(my_env) pkg>
```

For more information about Package Manager access https://pkgdocs.julialang.org/v1/.

This course uses a specific `Project.toml` to guarantee the code will be reproducible in all computers.

Listing 2.1. Project.toml

```
[deps]
BenchmarkTools = "6e4b80f9-dd63-53aa-95a3-0cdb28fa8baf"
CUDA = "052768ef-5323-5732-b1bb-66c8b64840ba"
ColorSchemes = "35d6a980-a343-548e-a6ea-1d62b119f2f4"
Images = "916415d5-f1e6-5110-898d-aaa5f9f070e0"
Plots = "91a5bcdd-55d7-5caf-9e0b-520d859cae80"
StatsBase = "2913bbd2-ae8a-5f71-8c99-4fb6c76f3a91"
```

It is highly recommended to create different environments for each Julia repository, because the package has a set of dependencies and, if you have a high number of different packages (about 20 or more), there is a high probability of dependency conflict. More on this topic is discussed in https://pkgdocs.julialang.org/v1/managing-packages/#conflicts.

2.3. High Performance Computing aspects in Julia

This section is dedicated to describe some aspects of HPC in Julia. We start by presenting how to measure time of a Julia application. Later, we go through some specificities of Julia interpreter and how to organize a serial for good performance even before an attempt to parallelize it.

2.3.1. Analyzing Julia Performance

As a high performance language, Julia has several tools to analyze and benchmark code performance. The most simple and yet the most useful tool is the `@time` macro.

```
julia> @time sqrt(rand())
  0.000001 seconds
  0.5857267559291746
```

The `@time` macro evaluates how much time Julia took to give aresult, in this case 0.000001 seconds, and provides the output of the function `sqrt(rand())` right below. Let us see more examples:

```
julia> @time sqrt.(rand(10))
  0.000002 seconds (2 allocations: 320 bytes)
10-element Vector{Float64}:
 0.7798392052070687
 0.5408866082279589
 0.994391535760473
 0.5254650346521715
 0.805928150879407
 0.4298229174632851
```

```
 0.5083361019219698
 0.5073599184347386
 0.8721449074432472
 0.7945992718819369
```

In the previous case we used Julia broadcasting attribute in the function `sqrt` by adding the dot after the function call, so that the function is applied to all elements in the array. Without the dot, Julia generates an error.

There is a more verbose version of `@time`, the `@timev` macro.

```
julia> @timev sqrt.(rand(1000));
  0.000005 seconds (2 allocations: 15.875 KiB)
elapsed time (ns): 5500
bytes allocated:   16256
non-pool GC allocs:2
```

Along the examples in this notes we will often run a piece of code twice. On the first call, the code gets compiled. This call should not be taken into account for the total time, just the second one. It is possible to speedup this latency with PackageCompiler [2].

For a more complete benchmarking, Julia provides a package called BenchmarkTools. Details are available at https://juliaci.github.io/BenchmarkTools.jl/stable/. On it, there is a macro called `@btime`, that provides the same information as `@time`.

Along our examples, we have made use of a macro called `@my_time`. The reason we used this macro, instead of the original @time, is to retrieve both the information about the result and the time elapsed. For more information about macros, look at the Julia Manual [3]. Our macro is defined as following:

Listing 2.2. My time

```
macro my_time(ex)
    return quote
        local t0 = time_ns()
        local val = $ex
        local t1 = time_ns()
        Δt = (t1-t0)/1e9
        val, Δt
    end
end
```

2.3.2. Organizing your serial code even before parallelization

As we already know, Julia is a dynamic language that balances the best of two worlds: the high-level flexibility of a pure interpreted language and the speed of compiled language. To do so, a JIT (Just-in-Time) Compiler amalgamates several design decisions and features of Julia.

[2] https://github.com/JuliaLang/PackageCompiler.jl
[3] https://docs.julialang.org/en/v1/manual/metaprogramming

In this section, we elaborate on some of these features that should be taken into account by the programmer before going to parallelization, that is, some guidelines to achieve performance in the serial version of the code. We start with a description of the four main steps of the JIT compilations process, from the user's source code (script) to the machine-level code. At each step, we touch upon some of the features the programmer should be aware of.

Julia's JIT compilation process can be disassembled in four main stages. The first ones are promoted by Julia, and generates intermediate representations, already implementing some code optimization, and the last one is target-specific, and is promoted by the LLVM[4] compiler framework, promoting final optimizations targeting the binary machine code. Julia provides code macros for the programmer be able to inspect the intermediate representations (IR) of the main steps, as we see bellow. Despite generally being hard to read them all in detail, important performance tips and checks can pop up inspecting them. We will describe them in sequence, as summarized in Figure 2.1.

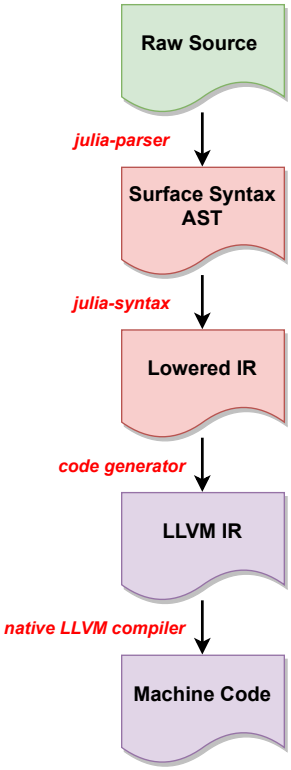


Figure 2.1. Julia's JIT compilation process diagram.

As a first step, the raw source code is parsed by Julia, producing an abstract syntax tree (AST) called *Surface Syntax AST*. In this tree representation, for example, operators are replaced by their respective function call. We can inspect this step by using the functions `dump` and `Meta.parse`. Try on the REPL `dump(:(1+2))`.

[4]originally an acronym for Low Level Virtual Machine

```
julia> dump(:(1+2))
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 1
    3: Int64 2
```

As we can see, we have an expression with three nodes being the *head* node the call to addition. Now try on the REPL `dump(:(1+2-3))`.

```
julia> dump(:(1+2-3))
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol -
    2: Expr
      head: Symbol call
      args: Array{Any}((3,))
        1: Symbol +
        2: Int64 1
        3: Int64 2
    3: Int64 3
```

We can see now two expressions in a hierarchical (tree) organization. First, the lower head node is the addition operation call, which results is a operand node for the upper head node: the subtraction operation call. Let us try a bit more complex code, now making use of the function `Meta.parse`. Try on the REPL `Meta.parse("sum = 0; for i in 1:5 sum+=i end")`.

```
julia> Meta.parse("sum = 0; for i in 1:5 sum+=i end")
:($(Expr(:toplevel, :(sum = 0), :(for i = 1:5 #= none:1 =# sum += i end))))
```

The output now is an expression, where the *head* node is the top level (tree root), and the child nodes are expressions (the ones preceded by :). We can now use it as an input to the `dump` function.

```
julia> dump(Meta.parse("sum = 0; for i in 1:5 sum+=i end"))
Expr
  head: Symbol toplevel
  args: Array{Any}((2,))
    1: Expr
      head: Symbol =
      args: Array{Any}((2,))
        1: Symbol sum
        2: Int64 0
    2: Expr
      head: Symbol for
      args: Array{Any}((2,))
        1: Expr
          head: Symbol =
          args: Array{Any}((2,))
            1: Symbol i
            2: Expr
              head: Symbol call
```

```
            args: Array{Any}((3,))
              1: Symbol :
              2: Int64 1
              3: Int64 5
        2: Expr
          head: Symbol block
          args: Array{Any}((2,))
            1: LineNumberNode
              line: Int64 1
              file: Symbol none
            2: Expr
              head: Symbol +=
              args: Array{Any}((2,))
                1: Symbol sum
                2: Symbol i
```

The Julia AST is then parsed by the *julia-syntax* producing an intermediate representation, known as *Lowered IR*. In this representation, the high-level Julia syntax is replaced by a smaller set of instructions, from the standard modules for example. Additionally, all control flow is converted to explicit branches (go to) and sequences of statements, and all macros are expanded. We can inspect this lowered IR using some functions and macros. The first macro is `Meta.@lower`. When applied to the **`for`** code snippet above we obtain:

```
julia> Meta.@lower begin sum = 0; for i in 1:5 sum+=i end end
:($(Expr(:thunk, CodeInfo(
    @ REPL[228]:1 within `top-level scope'
1 —       sum = 0
│   %2  = 1:5
│         #s564 = Base.iterate(%2)
│   %4  = #s564 === nothing
│   %5  = Base.not_int(%4)
└──       goto #4 if not %5
2 — %7  = #s564
│         i = Core.getfield(%7, 1)
│   %9  = Core.getfield(%7, 2)
│         sum = sum + i
│         #s564 = Base.iterate(%2, %9)
│   %12 = #s564 === nothing
│   %13 = Base.not_int(%12)
└──       goto #4 if not %13
3 —       goto #2
4 —       return nothing
))))
```

Another option is to encapsulate the code snippet in a function and use the macro `@code_lowered`.

```
julia> function my_sum()
       sum = 0; for i in 1:5 sum+=i end
       end
my_sum (generic function with 2 methods)

julia> @code_lowered my_sum()
CodeInfo(
1 —       sum = 0
│   %2  = 1:5
│         @_2 = Base.iterate(%2)
│   %4  = @_2 === nothing
│   %5  = Base.not_int(%4)
```

```
└──       goto #4 if not %5
2 ─ %7  = @_2
│         i = Core.getfield(%7, 1)
│   %9  = Core.getfield(%7, 2)
│         sum = sum + i
│         @_2 = Base.iterate(%2, %9)
│   %12 = @_2 === nothing
│   %13 = Base.not_int(%12)
└──       goto #4 if not %13
3 ─       goto #2
4 ─       return nothing
)
```

We see that the **`for`** statement is transformed to a sequence of `go to` together with `Base.iterate` function calls. Lowered IR is an important link in the Julia's compilation chain. It has a significant impact on Julia's performance, since it is used for type inference, optimizations like inlining, and code generation. Types and type inference, together with Multiple Dispatch, are some language design features that guarantees Julia's performance. The type inference outcome can be inspected by the macro `@code_type`. For example, try the commands bellow:

```
julia> function my_sum(n)
           sum = 0; for i in 1:n sum+=i end
           return sum
       end
my_sum (generic function with 2 methods)

julia> @code_typed my_sum(5)
CodeInfo(
1 ── %1  = Base.sle_int(1, n)::Bool
│    %2  = Base.ifelse(%1, n, 0)::Int64
│    %3  = Base.slt_int(%2, 1)::Bool
└──       goto #3 if not %3
2 ──       goto #4
3 ──       goto #4
4 ── %7  = φ (#2 => true, #3 => false)::Bool
│    %8  = φ (#3 => 1)::Int64
│    %9  = φ (#3 => 1)::Int64
│    %10 = Base.not_int(%7)::Bool
└──       goto #10 if not %10
5 ── %12 = φ (#4 => %8, #9 => %21)::Int64
│    %13 = φ (#4 => %9, #9 => %22)::Int64
│    %14 = φ (#4 => 0, #9 => %15)::Int64
│    %15 = Base.add_int(%14, %12)::Int64
│    %16 = (%13 === %2)::Bool
└──       goto #7 if not %16
6 ──       goto #8
7 ── %19 = Base.add_int(%13, 1)::Int64
└──       goto #8
8 ── %21 = φ (#7 => %19)::Int64
│    %22 = φ (#7 => %19)::Int64
│    %23 = φ (#6 => true, #7 => false)::Bool
│    %24 = Base.not_int(%23)::Bool
└──       goto #10 if not %24
9 ──       goto #5
10 ─ %27 = φ (#8 => %15, #4 => 0)::Int64
└──       return %27
) => Int64
```

Note that all the annotated types and the final line indicating the returned type `Int64` of

the function. Now we suggest to run the line command `@code_typed my_sum(5.0)` in the REPL and compare the result with the output above. Notice that we obtain `Union{Float64, Int64}`, depicting an important feature, related to type stability, that we describe later. By now, let us return to the description of the compilation process chain.

The lowered IR passes through the code generator, see Figure 2.1, to produce the LLVM IR. Actually, beginning in Julia 0.7, parts of the compiler use a new SSA-form (*static single assignment form*) intermediate representation. The code above already brings elements of this representation, like the ϕ functions. They also appear in the LLVM IR, that can be inspected by the `@code_llvm` macro, for example,

```
julia> @code_llvm my_sum(5)
;  @ REPL[231]:1 within `my_sum'
define i64 @julia_my_sum_4154(i64 signext %0) {
top:
;  @ REPL[231]:2 within `my_sum'
; ┌ @ range.jl:5 within `Colon'
; │┌ @ range.jl:287 within `UnitRange'
; ││┌ @ range.jl:292 within `unitrange_last'
     %.inv = icmp sgt i64 %0, 0
     %1 = select i1 %.inv, i64 %0, i64 0
; └└└
  br i1 %.inv, label %L12.preheader, label %L27

L12.preheader:                                    ; preds = %top
  %2 = shl nuw i64 %1, 1
  %3 = add nsw i64 %1, -1
  %4 = zext i64 %3 to i65
  %5 = add nsw i64 %1, -2
  %6 = zext i64 %5 to i65
  %7 = mul i65 %4, %6
  %8 = lshr i65 %7, 1
  %9 = trunc i65 %8 to i64
  %10 = add i64 %2, %9
  %11 = add i64 %10, -1
;  @ REPL[231]:3 within `my_sum'
  br label %L27

L27:                                              ; preds = %L12.preheader, %top
  %value_phi9 = phi i64 [ 0, %top ], [ %11, %L12.preheader ]
  ret i64 %value_phi9
}
```

Finally, the last representation, target-specific, is the *LLVM assembly code*, hereafter called the *machine code*. Following our recurrent example, we have

```
 julia> @code_native my_sum(5)
        .section        __TEXT,__text,regular,pure_instructions
; ┌ @ REPL[231]:2 within `my_sum'
; │┌ @ range.jl:5 within `Colon'
; ││┌ @ range.jl:287 within `UnitRange'
; │││┌ @ range.jl:292 within `unitrange_last'
        testq   %rdi, %rdi
; │└└└
        jle     L43
; │ @ REPL[231] within `my_sum'
        movq    %rdi, %rax
        sarq    $63, %rax
        andnq   %rdi, %rax, %rax
; │ @ REPL[231]:2 within `my_sum'
```

```
        leaq    -1(%rax), %rdx
        leaq    -2(%rax), %rcx
        mulxq   %rcx, %rcx, %rdx
        shldq   $63, %rcx, %rdx
        leaq    (%rdx,%rax,2), %rax
        decq    %rax
; | @ REPL[231]:3 within `my_sum'
        retq
; | @ REPL[231] within `my_sum'
L43:
        xorl    %eax, %eax
; | @ REPL[231]:3 within `my_sum'
        retq
        nop
; L
```

Julia is a language targeting scientific computing applications, as such, vector, matrices, and, more generally, arbitrary dimension arrays, a data container provided by `Base` library. They are parameterized by the data type `T` (for example `Int64, Float64`, etc) and dimension `N` in `Array{T,N}`. For vectors, `N=1`. In fact, the definition `Vector{T}` is an alias for `Array{T,1}`. For matrices, `N=2`, the alias `Matrix{T}` is an alias for `Array{T,2}`. The indexing rule for `N`-dimensional arrays is from the last index to the first. In terms of matrices, this means that the memory layout of data storage is column-wise. To exemplify this and propose some gradual performance improvements we will resort to an important Linear Algebra kernel, the matrix-vector product, hereafter referred as `matvec`.

The code in 2.3 implements a row-wise `matvec`, that is, taking inner products of the matrix row with the column vector, whiel code in 2.4 implements a column-wise `matvec`, that is, forming a linear combination of the matrix columns, with the vector entries as coefficients.

Listing 2.3. Matvec row-wise (row-column inner products).

```
function matvec_row!(y, A, x)
    for i in 1:size(A,1)
        for j in 1:size(A,2)
            y[i] += A[i,j]*x[j]
        end
    end
end
```

Listing 2.4. Matvec column-wise (linear combination of the column)

```
function matvec_col!(y, A, x)
    for j in 1:size(A,2)
        for i in 1:size(A,1)
            y[i] += A[i,j]*x[j]
        end
    end
end
```

As one would expect, by the matrix data layout in memory, `matvec_col` is faster than `matvec_row`. Indeed, using the `@btime` macro, we obtain

```
julia> @btime matvec_row!(y,$A,$x) setup=(y = zero(x))
  2.682 ms (0 allocations: 0 bytes)

julia> @btime matvec_col!(y,$A,$x) setup=(y = zero(x))
  688.710 μs (0 allocations: 0 bytes)
```

Julia has bounds checking, and if the programmer can guarantee to have full control of it, as in our case, since we have a simple loop by using the function `size`), one can make use of the macro `@inbounds`. This macro implies checking is skipped, impacting positively to the performance. In listing 2.5, we make use of it to reduced the time to run `matvec_col`.

Listing 2.5. Matvec column-wise with `@inbounds` macro.

```
function matvec_col_inbounds!(y, A, x)
    @inbounds for j in 1:size(A,2)
        for i in 1:size(A,1)
            y[i] += A[i,j]*x[j]
        end
    end
end
```

And we observe an expressive gain in performance:

```
julia> @btime matvec_col_inbounds!(y,$A,$x) setup=(y = zero(x))
  261.954 μs (0 allocations: 0 bytes)
```

But as one can note, we are not using any high-level syntax, aside from that, our code looks very similar to a C/C++ vanilla implementation. The listing 2.6 implements a naive column-wise `matvec` using the array slicing `A[:,j]` to get the matrix columns.

Listing 2.6. Matvec column-wise using naive high-level syntax.

```
function matvec_col_hl_naive!(y, A, x)
    for j in 1:size(A,2)
         y += A[:,j]*x[j]
    end
end
```

In this case, we obtain a timing that is even worse than the `matvel_row`. Indeed, we get

```
julia> @btime matvec_col_hl_naive!(y,$A,$x) setup=(y = zero(x))
  3.057 ms (3000 allocations: 23.25 MiB)
```

Notice that the number of allocations increased from 0 to 3000, causing a considerable degradation in performance when compared to `matvec_col`. The reason for that is the array slicing `A[:,j]`, that promotes a copy of the matrix column. For such case, one can use the macro `@view`, as in the Listing 2.7

Listing 2.7. Matvec column-wise using high-level syntax fixing slicing allocations.

```
function matvec_col_hl_fix1!(y, A, x)
    for j in 1:size(A,2)
         y += @view(A[:,j])*x[j]
    end
end
```

The macro `@view` prevents these copies, then increasing performance:

```
julia> @btime matvec_col_hl_fix1!(y,$A,$x) setup=(y = zero(x))
  2.140 ms (2000 allocations: 15.50 MiB)
```

But we still note a non-negligible number of allocations, 2000. The reason now is that we are still allocating temporary vectors to perform the scalar-vector multiplication and the vector-vector addition. The fixing now is by use of one of the performance gain players of Julia, namely, the broadcasting. Broadcasting is achieved by using the dot operator, `.`, as in the Listing 2.8

Listing 2.8. Matvec column-wise using high-level syntax still allocations with broadcast.

```
function matvec_col_hl_fix2!(y, A, x)
    for j in 1:size(A,2)
         y .+= @view(A[:,j]).*x[j]
    end
end
```

We finally zeroed the number of allocations, and obtained a comparable performance, as we can see from

```
julia> @btime matvec_col_hl_fix2!(y,$A,$x) setup=(y = zero(x))
  262.494 μs (0 allocations: 0 bytes)
```

In this simple progressive exercise, we were able to show the most important aspects related to working with arrays, in such a way to obtain a performative serial code. We will pass then to aspects of parallelization in Julia.

2.4. Multicore Computing

Modern computer CPUs have architectures that allow multiple cores to operate simultaneously. While one of the possibilities is that different applications use different cores, in scientific computing we would like that a single application of interest use part or all the available cores of a single CPU. This is achieved via the so-called threading. Along this section we describe the basic aspects of threading in Julia and we include examples.

Two remarks must be addressed. First is about taking care while using Input/Output operations when using code containing multithread calls. The second is the use of nested threading, that must be avoided. This last issue may appear even when using basic scientific computing libraries and one thread implementation in Julia. Several basic libraries,

as for linear algebra, include already parallelized versions within it. An attempt of using the basic threading in Julia while making calls for functions of such libraries may lead to an unexpected nested threading and the computing time may end up getting worse when compared with the serial version call of the same function. This effect happens because the code may try to use twice the maximum number of threads possible.

2.4.1. Threading in Julia

Threads are independent parts computation that run on a CPU core. As known in HPC techniques, we should use at maximum the number of threads as the number of cores available. One point of attention is when the CPU you work with has the hyper-threading option. Intel's CPUs have such option and operational systems may see the number of cores available as the double of the real (physical) CPU cores. For intense CPU computation, you must use at maximum the number of physical cores in the CPU. One form of embedding in the Julia code the information about the number of physical cores of the machine being used is through the package **Hwloc**, and then with the use of the command `Hwloc.num_physical_cores()`.

In Julia the number of threads is fixed at startup. Thus, to set it, you can export the environment variable `JULIA_NUM_THREADS` setting the number the threads you would like or you can call Julia with the flags `-t` or `--threads`. The command `nthreads()` shows the current number of threads in use.

```
$ export JULIA_NUM_THREADS=4
$ julia

julia> using Base.Threads

julia> nthreads()
4
```

In Julia the use of threads is done via the `@threads` macro, as we can see in the Hello World file example 2.9.

Listing 2.9. Hello World

```
using Base.Threads

Threads.@threads for thread = 1:nthreads()
    println("$(Threads.threadid()) of $(nthreads()) - Hello World!")
end
```

When in the command-line, we call Julia using 4 threads to run such code, obtaining as result:

```
$ julia -t 4 .\hello-world.jl
  1 of 4 - Hello World!
  2 of 4 - Hello World!
  3 of 4 - Hello World!
  4 of 4 - Hello World!
```

The iteration space is split among the threads, each one having its thread ID, and each thread prints "Hello World!". The macro `@threads` is similar to the use of OpenMP pragmas in the C++ language [5].

2.4.2. Synchronization with threads

Since multiple cores can access simultaneously the same memory, we may have different cores changing the same piece of information in memory at the same time. This is often called *race condition*. In order to avoid it, Julia offers the possibility of changing values atomically, similarly to the SYNCHRONIZATION Pragmas in OpenMP. Example 2.10 demonstrates the use.

Listing 2.10. Vector sum

```
using Base.Threads

N = 1000
acc1 = Ref(0) # Pointer with value 0
@threads for i in 1:N
    acc1[] += 1 # Add 1 with race condition
end
println("Sum wrong is $(acc1[])")

acc2 = Atomic{Int64}(0) # Atomic pointer with value 0
@threads for i in 1:1000
    atomic_add!(acc2, 1) # Add 1 thread safely
end
println("Sum atomic is $(acc2[])")
```

When the previous code is run, we obtain:

```
 $ julia -t 4 .\vsum.jl
Sum wrong is 989
Sum atomic is 1000
```

Such strange behavior occurs because our use of a pure function[6]. If a function modify its arguments, we might end up with a wrong answer. To avoid such issue, it suffices to encapsulate the sum in a function, then obtaining the right answer.

2.4.3. Selection Sort Algorithm via threads

We now pass to an example of the use of threads in Julia. We implement a serial and a parallel version of the algorithm *selection sort*. We can the observe several aspects on the following code: show how we can adapt a serial Julia code to its parallel counterpart, regard how to prepare the time measurements of both codes running, and finally regard the speed up achieved. Both versions of the algorithm are in the Listing 2.11, and the experiment of speed up was performed using 88 threads (more about speed up of this

[5] https://www.intel.com/content/www/us/en/develop/documentation/oneapi-dpcpp-cpp-compiler-dev-guide-and-reference/top/optimization-and-programming-guide/openmp-support/openmp-pragmas-summary.html

[6] https://en.wikipedia.org/wiki/Pure_function

code and other examples is presented in Section 2.6.

Listing 2.11. Selection sort

```
using Base.Threads

macro my_time(ex)
    return quote
        local t0 = time_ns()
        local val = $ex
        local t1 = time_ns()
        Δt = (t1-t0)/1e9
        val, Δt
    end
end

function selectionsort!(arr::Vector{<:Real})
    len = length(arr)
    if len < 2 return arr end
    for i in 1:len-1
        lmin, j = findmin(arr[i+1:end])
        if lmin < arr[i]
            arr[i+j] = arr[i]
            arr[i] = lmin
        end
    end
    return arr
end

function selectionsortparallel!(arr::Vector{<:Real})
    len = length(arr)
    if len < 2 return arr end
    @threads for i in 1:len-1
        lmin, j = findmin(arr[i+1:end])
        if lmin < arr[i]
            arr[i+j] = arr[i]
            arr[i] = lmin
        end
    end
    return arr
end

v = rand(50000)
val_ser, Δt_ser = @my_time selectionsort!(v)
val_ser, Δt_ser = @my_time selectionsort!(v)
println("Serial:   time = ",Δt_ser, " seconds")
val_par, Δt_par = @my_time selectionsortparallel!(v)
val_par, Δt_par = @my_time selectionsortparallel!(v)
println("Parallel: time = ",Δt_par, " seconds")
@assert val_ser == val_par
```

The computing times were:

```
  $ julia -t 88 .\selection_sort.jl
Serial:   time = 3.065008172 seconds
Parallel: time = 0.374435835 seconds
```

showing the parallel version was more than 8 times faster compared to the serial version.

2.4.4. Final Remarks on Threads

The first remark concerns the use of I/O operations inside threads, what is advisable to avoid. In particular, the use of the command `println` for debugging multithreaded code. A safer way is the command `Core.println`.

The second remark is about Julia's garbage collector. It does not run in threaded mode and an attempt to use it will block all other threads, slowing down the performance.

The final remark is about Threaded libraries. Many libraries already use threads inside its code to speed up calculations. An example is the linear algebra library *OpenBLAS*. Thus, using it in your code already gives you a multicore performance automatically. However, since nested threads are not safe in Julia, it is not advisable to use calls to such threaded libraries inside your own threaded code. An attempt to do it, could in fact reduce the performance compared to a simple serial call to a function of the threaded library.

We will pass now to aspects of parallelization of Julia code on distributed computing.

2.5. Distributed Computing or Multi-Machine Computing in Julia

Most of HPC systems have many computer nodes. In this context, scientific applications can be distributed across all system processors/cores aiming to achieve high performance and meet the computational demands of accurate simulations. Using the Julia language, we can take advantage of this approach, dividing the work of an application between several computational nodes, each one possessing multiple CPUs.

2.5.1. Introduction to Julia Clusters

We discuss now how to operate with such architectures using Julia. The abstraction used in Julia for such tasks is called *Clusters*. This abstraction can also be used on a single computer for calculations with multiple cores, without concerning about synchronization issues. This is similar to the use of the MPI abstraction in C++ Language on a single machine, what is largely described in the literature of HPC systems. Also, there is a interface of MPI in Julia called `MPI.jl`[7].

One of the forms of working with distributed computing in Julia is via the **Distributed** package. If we wish to use it, we first need to start Julia on a local cluster using the flag `-p`. To start it with 2 workers, load the package, and check how many workers we have set, proceed as:

```
 $ julia -p 2

julia> using Distributed

julia> workers()
2-element Vector{Int64}:
 2
 3
```

Alternatively, we can use the function `addprocs` from the `Distributed` package:

[7] https://github.com/JuliaParallel/MPI.jl

```
 $ julia

julia> using Distributed

julia> addprocs(2)
2-element Vector{Int64}:
 2
 3
```

The advantage of the method `addprocs()` is that, being it a Julia's pure function, it can be called from the REPL or from another Julia code.

On both previous cases, 2 additional processes were created besides the master process. Communication in Julia is one-sided. So, a master process is used to receive inputs from the user and this master process controls all the other processes. In the example above, there is one master process and two workers' processes.

Since we now have the option to perform calculations on multiple machines, we need to have access to them. A simple way of doing that is with the help of cluster managers. They are tools allowing us from a single machine to perform calculations on other machines located in the physical cluster (containing multiple CPUs). There are different ways of organizing machines inside a cluster, the usual one is via *partitions*. They are simply abstractions on how to group different machines. You can also think of that as an alias to a set of machines where calculations will be performed. Cluster managers control also the time a single user can have on one or several partitions, the priority of an user compared to others, the amount of time that can be allocated on a partition, etc.

Along the examples in the notes, we have made use of SLURM as a cluster manager. We quickly introduce SLURM and how to work with it in what follows.

2.5.2. Slurm Workload Manager

Slurm [Yoo, Jette e Grondona 2003] (Simple Linux Utility for Resource Management) is an Open-source job scheduler for Linux-based systems created by LLNL (Lawrence Livermore National Laboratory). Currently, Slurm is deployed at many supercomputers worldwide, including the Brazilian super computer SDumont. Slurm allows users to allocate and deploy batch jobs to a series of resources, ensuring that detailed resources, like CPUs, memory, or full nodes, are accurately shared among users and jobs without wrong interaction. Primarily, each set of equal resources is aggregated on a division of resources with an individual queue for submissions of jobs. Users can send jobs and allocate entire nodes of different partitions.

2.5.2.1. Interactive Jobs on Slurm

Interactive jobs is a manner of working directly at the operational system of each machine and having a direct screen feedback of the operations being performed. We will decide for such jobs in Julia while using the REPL.

Aiming to submit interactive jobs, it is necessary to use the salloc command, requesting the resources to be used. When salloc allocates the requested resources to the job, it

informs the user, who can then access the node (via ssh), perform its tasks locally, and run the application. See examples on how to use salloc below.

Example 1 Request to allocate any machine present in the *draco* partition for 5 hours:

```
salloc -p draco -J NAME-JOB -t 05:00:00
```

Example 2 Request to allocate two computational nodes present in the *hype* partition for 24 hours:

```
salloc -p hype -N 2 -J NAME-JOB -t 24:00:00
```

where `NAME-JOB` is the name given to such job and that it will be listed on the queue of jobs, visible to all other users on the cluster.

Besides salloc, there are two other relevant SLURM commands: **sinfo** and **squeue**. Command sinfo presents information about SLURM nodes and partitions, while squeue is used to view jobs in queue or running.

2.5.2.2. Non-Interactive Jobs on SLURM

Non-interactive jobs are those in which the user submits a script that runs on the machines. SLURM permits this kind of scripting using directives #SBATCH. Every line starting with this command will have its content passed straight to SLURM at allocation time. Those non-interactive jobs are the ones we will use to run an entire piece of Julia code saved in a file.

A simple non-interactive script to a Julia code for the *selection_sort.jl* file is in the Listing 2.12. Line 1 is used to instruct the operating system to use bash as a command interpreter. Lines 2 and 3 set the partition as Blaise and ask for one node, respectively. Line 5 sets the number of tasks to 88, which is the number of threads of the Blaise node. Line 6 request the amount of time to run the job, here as 10 minutes. Finally, in the last line, there is the call to run the code contained in the Julia file.

Listing 2.12. Non-interactive Julia Example

```
#!/bin/bash

#SBATCH --partition=blaise
#SBATCH --nodes=1
#SBATCH --ntasks=88
#SBATCH --time=0:10:00

julia-1.7.2/bin/julia -t 88 ./selection_sort.jl
```

Once the SBATCH script is ready, for example with the name run_julia_selection_sort.slurm, you can request a non-interactive job to run it as:

```
$ sbatch run_julia_selection_sort.slurm
```

You must be aware that you can not request a longer computing time than the maximum allowed for a given partition, a value predefined by the cluster administrator. Also, if your computation is not done by the time it was requested for computing, you may loose the results.

2.5.3. Back to Julia Clusters

In order to start all workers with a certain project it is necessary to set the system environment variable `JULIA_PROJECT` as the directory path of the `Project.toml` file. Once this is done, we can check which packages and versions were loaded.

```
$ export JULIA_PROJECT="/home/users/rbklausner/Julia_HPC"

$ julia -p 2

julia> using Pkg

julia> Pkg.status()
      Status `~/Julia_HPC/Project.toml`
  [6e4b80f9] BenchmarkTools v1.2.2
  [052768ef] CUDA v3.8.0
  [35d6a980] ColorSchemes v3.16.0
  [916415d5] Images v0.25.1
  [91a5bcdd] Plots v1.25.7
  [2913bbd2] StatsBase v0.33.14
```

Julia can perform distributed computing on a high-level and on a low-level of control and communication to workers. Using at high-level we simply distribute computational effort along workers without worrying about the specific calculation being performed at a specific worker. At low-level programming, we are able to designate which computation will be performed at which worker in our cluster. The high-level option is easier while writing parallel code.

2.5.3.1. High-level distributed computing

The main tools for high-level distributed programming in Julia are the `@distributed` macro and the parallel map called `pmap`. To illustrate the use of both we first show how a simple function can be defined in Julia. This will be useful for understanding the behavior of `pmap`. We define a simple function that calculates the square of a number.

```
julia> f = i -> i^2

julia> f(12)
       144
```

In the example of high-level computing below we exemplify on how to call a distributed piece of code using `@distributed` and `pmap`. We start by calling Julia with 4 workers besides the master process. Then, using `nprocs()` we check the total number of processes, 5. The method `workers()` retrieves the ID of the workers, while `nworkers()` retrieves the number of available workers.

```
 $ julia -p 4

julia> nprocs()
5

julia> procs()
5-element Vector{Int64}:
 1
 2
 3
 4
 5

julia> workers()
4-element Vector{Int64}:
 2
 3
 4
 5

julia> nworkers()
4

julia> myid()
1

julia> pmap(i -> println("Worker $(myid()), working on i=$i"), 1:10)
      From worker 4:    Worker 4, working on i=1
      From worker 4:    Worker 4, working on i=5
      From worker 4:    Worker 4, working on i=6
      From worker 4:    Worker 4, working on i=7
      From worker 4:    Worker 4, working on i=8
      From worker 4:    Worker 4, working on i=9
      From worker 4:    Worker 4, working on i=10
      From worker 3:    Worker 3, working on i=2
      From worker 2:    Worker 2, working on i=3
      From worker 5:    Worker 5, working on i=4

julia> @sync @distributed for i in 1:10
         println("Worker $(myid()), working on i=$i")
       end
      From worker 3:    Worker 3, working on i=4
      From worker 3:    Worker 3, working on i=5
      From worker 3:    Worker 3, working on i=6
      From worker 4:    Worker 4, working on i=7
      From worker 4:    Worker 4, working on i=8
      From worker 2:    Worker 2, working on i=1
      From worker 5:    Worker 5, working on i=9
      From worker 5:    Worker 5, working on i=10
      From worker 2:    Worker 2, working on i=2
      From worker 2:    Worker 2, working on i=3
```

The behavior of `@distributed` and `pmap` seems to be very similar. `pmap` works applying a function on a one-dimensional list or array. It passes each element of the list/array to each worker node, one at a time. Once a worker finishes the computation and retrieves the result, a new element of the list/array is sent to this worker. The process goes until the last element of the list is sent and computation performed.

Now, the `@distributed` macro works on a different manner. It splits the list/array in a set of blocks, with the number of blocks equal to the number of workers available. Then, each entire block is sent to each worker, at one shot, at the beginning of computation. Since `@distributed` executes the code asynchronously, it is necessary to wait for completion before going to the next instruction in Julia. For that purpose, we can use `@sync`.

Not waiting for a process to complete may generate sync errors.

Since the `pmap` requires a larger amount of communication steps, it will perform better if the calculation on a single element consumes a good amount of time.

We must introduce another macro, the `@everywhere`, that will help us on the distributed computing. Is is used to run the same piece of code over all processes in the cluster. An example of this use is calculating the realization of a stochastic function for different seeds.

```
julia> @everywhere println(rand())
0.856476480924345
      From worker 2:    0.3882556897855819
      From worker 5:    0.4721670866061788
      From worker 3:    0.22741379804937245
      From worker 4:    0.21971772218390695
```

On the output above, the master process produces the first result, followed by the workers printing their results.

The example below presents the use of `map`, `pmap`, and `@distributed` for two functions, one of fast calculation with simple evaluation, the other one, a slow one, composed of a nested loop.

Listing 2.13. Pmap example

```
@everywhere function fast(x::Float64)
    return x^2+1.0
end

@everywhere function fast_distributed(arr::StepRangeLen)
    @sync @distributed for i in arr
        fast(i)
    end
end

@everywhere function slow(x::Float64)
    a = 1.0
    for i in 1:1000
        for j in 1:5000
            a+=asinh(i+j)
        end
    end
    return a
end

@info "Precompilation"
map(fast,range(1,1000,1000))
pmap(fast,range(1,1000,1000))
map(slow,range(1,1000,10))
pmap(slow,range(1,1000,10))
fast_distributed(range(1,1000,1000))

@info "Testing slow function"
@time map(slow,range(1,1000,10))
@time pmap(slow,range(1,1000,10))
@info "Testing fast function"
@time map(fast,range(1,1000,1000))
@time pmap(fast,range(1,1000,1000))
@info "Testing @distributed fast function"
```

```
@time fast_distributed(range(1,1000,1000))
```

We can regard the different running times below.

```
  $ julia -p 32 .\pmap.jl
[ Info: Precompilation
[ Info: Testing slow function
  2.411691 seconds (1 allocation: 144 bytes)
  0.350282 seconds (1.28 k allocations: 59.188 KiB)
[ Info: Testing fast function
  0.000011 seconds (1 allocation: 7.938 KiB)
  0.077671 seconds (46.37 k allocations: 1.568 MiB)
[ Info: Testing @distributed fast function
  0.018515 seconds (5.49 k allocations: 201.750 KiB)
```

And we can conclude, as expected, that, for the slow function, `pmap` has gains over the use of a simple `map`, more than 7 times faster. However, for the fast function, `pmap` has a worse behavior, being more than 7000 times slower, while `@distributed` is about 1600 times slower than `map`.

All such degrading behavior is due to the unnecessary distributed computing that includes communication performed. This also shows that `@distributed` is a better fit for simple computations per node, as `pmap` will be the successful one when intense (long) computations are done per node. Long here meaning that the amount of time spent on communication from the master process to the worker and back is negligible compared to the time spent on the computation.

2.5.3.2. Estimation of Pi via Monte Carlo

The next example calculates an approximation for π using the Monte Carlo technique. The idea is to sample N_{total} 2D points in a square of side 1 unit. From the sampled points, $N_{inner} < N_{total}$ will be inside the unit circle. Since the circle has area $\pi/4$, the ratio N_{inner}/N_{total} will be approximately $\pi/4$.

We make use of `pmap` for the computation on a distributed system with 88 processes. At this example we can see how easy is to transform a serial version to a parallel one using a high-level distributed computing technique of Julia. This requires one extra line of code and one use of `@everywhere`. At the end of computation we compare the speed of both codes.

Listing 2.14. Monte Carlo pi estimative

```
using Distributed

macro my_time(ex)
    return quote
        local t0 = time_ns()
        local val = $ex
        local t1 = time_ns()
        Δt = (t1-t0)/1e9
        val, Δt
```

```
    end
end

@everywhere function darts_in_circle(N)
    n = 0
    for i in 1:N
        if rand()^2 + rand()^2 < 1
            n += 1
        end
    end
    return n
end

function pi_distributed(N, loops)
    n = sum(pmap((x)->darts_in_circle(N), 1:loops))
    4 * n / (loops * N)
end

function pi_serial(n)
    return 4 * darts_in_circle(n) / n
end

N = 20_000_000
loops = 100
pi_approx_ser, t_ser = @my_time pi_serial(N*loops)
pi_approx_ser, t_ser = @my_time pi_serial(N*loops)
pi_approx_par, t_dist = @my_time pi_distributed(N, loops)
pi_approx_par, t_dist = @my_time pi_distributed(N, loops)
println("Serial = $pi_approx_ser in $t_ser seconds")
println("Distributed = $pi_approx_par in $t_dist seconds")
```

Has the result :

```
  $ julia -p 88 .\monte_carlo.jl
Serial = 3.141549988 in 8.427463928 seconds
Distributed = 3.141601936 in 0.307187824 seconds
```

We can see that the parallel version was about 28 times faster. Similar calculations, for different number of processes are analyzed in Section 2.6. We pass now to a brief discussion of the low-level aspects of distributed computing with Julia.

2.5.3.3. Low-level distributed computing

The manner Julia implements low-level control on distributed computing is via **remote calls** and **remote references**. A remote call is a request to execute a function on a particular process. You need to wait (with `wait()`) the process to complete in order to fetch (with `fetch()`) the result into the master process. A remote reference is a reference to data living on a specific process.

The remote call is an asynchronous process. The remote call finishes almost immediately and it returns a **Future** object, which is the remote reference for the result of such call. The way we carry out a remote call is specifying, in sequence, which function we will apply, the number of the process where this function will be calculated, and the argument(s) to such function. The caller then waits (`wait()`) on the reference and once the call is completed, it can `fetch()` the result of the computation and use it. A example of such

use is below, where we call the function `sqrt` with the argument `4.0` in the process 2.

```
julia> a = remotecall(sqrt, 2, 4.0)
Future(2, 1, 6, ReentrantLock(nothing, Base.GenericCondition{Base.Threads.
    SpinLock}(Base.InvasiveLinkedList{Task}(nothing, nothing), Base.Threads.
    SpinLock(0)), 0), nothing)

julia> wait(a)
Future(2, 1, 6, ReentrantLock(nothing, Base.GenericCondition{Base.Threads.
    SpinLock}(Base.InvasiveLinkedList{Task}(nothing, nothing), Base.Threads.
    SpinLock(0)), 0), nothing)

julia> fetch(a)
2.0
```

Another way of low-level control on distributed computing is via the use of `spawnat`. As an example, we create first a remote call **r**. In the example we do not specify in which node it will be computed, since we used as second argument of `@spawnat` the option `:any`. The computation will be then performed on one of the nodes. We then create the variable **s** that adds one to the result of the first remote call. As described before, we need to use `fetch(r)` to pick the result of this first call, and then we can use this result for further calculations. While creating the variable `s`, we called `@spawnat`, again with the option `:any`, meaning that the operation of adding 1 to the result of fetch will be performed on any node. It is possible to specify different nodes for the two different calculations. To select a specific node, we would need just to define the desired node instead of using `:any`. On an example with 4 nodes, we could run the first calculation on node 3, while the second one on node 2.

```
julia> r = @spawnat :any rand(2,2)
Future(2, 1, 25, ReentrantLock(nothing, Base.GenericCondition{Base.Threads.
    SpinLock}(Base.InvasiveLinkedList{Task}(nothing, nothing), Base.Threads.
    SpinLock(0)), 0), nothing)

julia> s = @spawnat :any 1 .+ fetch(r)
Future(3, 1, 26, ReentrantLock(nothing, Base.GenericCondition{Base.Threads.
    SpinLock}(Base.InvasiveLinkedList{Task}(nothing, nothing), Base.Threads.
    SpinLock(0)), 0), nothing)

julia> fetch(s)
2×2 Matrix{Float64}:
 1.12129  1.28538
 1.75886  1.34516
```

Such level of control on distributed computing may be necessary on some applications. In the following, we present a large algorithm that makes use of `@spawnat`.

2.5.3.4. Application - Ant Colony Optimization Algorithm

Ant colony optimization (ACO) is a probabilistic technique to find minimum paths. Its use range from solving mathematical problems to concrete industrial applications. The algorithm can be seen as part of metaheuristic optimization algorithms.

The purpose of this example is to illustrate how a more sophisticated problem, whose solution makes use of several predefined functions, can be simply parallelized using Julia. Example 2.15 presents, first, an adaptation of the serial code available in https://github.com/schiegl/AntColony.jl. Then, it depicts a parallel version written by one of the authors of these notes, R.B. Klausner. The main changes to the serial code are one call of `@spawnat`, creating a dictionary, and coordinate calls using `@sync` and `@async`, besides calling the necessary packages for the computations and the **Distributed** one for the parallel computing.

Listing 2.15. Ant Colony Optimization

```
using Distributed

macro my_time(ex)
    return quote
        local t0 = time_ns()
        local val = $ex
        local t1 = time_ns()
        Δt = (t1-t0)/1e9
        val, Δt
    end
end

@everywhere using LinearAlgebra, Random, StatsBase

# Make a path based on the Probability Matrix
@everywhere function travel(
        P::Matrix{Float64},
        start_node::Int
    )
    n_nodes = size(P,1)
    not_visited = collect(1:n_nodes)
    path = [start_node]
    P_copy = copy(P)

    node = start_node

    for i in 2:n_nodes
        # Sample a new node
        next_node = sample(1:n_nodes,Weights(P_copy[node,:]))
        # You can't revisit the node
        P_copy[:,1:end .== node] .= 0.0
        # Normalize the other nodes probabilities, to sum 1
        P_copy ./= sum(P_copy,dims=2)
        # Go to the next node
        node = next_node
        push!(path, node)
    end
    return path
end

# Calculate the distances from a path
@everywhere function edge_distances(
        dist_mat::AbstractMatrix{<:Number},
        path::AbstractArray{Int}
    )
    [dist_mat[to, from] for (from, to) in edges(path)]
end

# Return a array of all edges in a path
@everywhere function edges(path::AbstractArray{Int})
    path_len = length(path)
    [(path[i], path[mod1(i + 1, path_len)]) for i in 1:path_len]
end
```

```
# Calculate the Serial Ant Colony Optimization Algorithm
function ACO(dist_mat,max_iter=50,nants=5;
    β=2,α=1,Q=1,ρ = 0.5,verbose = true)

    # Attractiveness of the move, calculated a priori
    η = (1 ./dist_mat) .^ β
    # Trail level
    τ = ones(size(dist_mat)) .^ α

    best_path = nothing
    best_cost = Inf
    no_improv = 0
    n_nodes = size(dist_mat,1)

    for i in 1:max_iter

        solutions = []
        # Create the probability matrix
        P = η.*τ
        P ./= sum(P,dims=2)

        # Run all ants in serial
        for k in 1:nants
            # Start randomly in a node
            start_node = rand(1:n_nodes)
            # Need to return to the same node
            end_node = start_node
            # Calculate path and costs
            path = travel(P, start_node)
            cost = sum(edge_distances(dist_mat, path))
            push!(solutions, (cost, path))
        end

        # Sort solutions
        sort!(solutions)
        best_local_cost, best_local_path = solutions[1]
        # Check if found a better cost path
        if best_local_cost < best_cost
            if verbose
                println("Better solution found with cost
                $(best_local_cost) at iteration $(i)")
            end
            best_path = best_local_path
            best_cost = best_local_cost
            no_improv = 0
        else
            no_improv += 1
        end

        # deposit pheromones
        τ .*= (1-ρ)
        for (cost, path) in solutions
            Δτ = Q / cost
            for (from, to) in edges(path)
                τ[to, from] += Δτ
            end
        end
    end
    best_path,best_cost
end

# Calculate the Parallel Ant Colony Optimization Algorithm
function ACO_parallel(dist_mat,max_iter=50,nants=5;
    β=2,α=1,Q=1,ρ = 0.5,verbose = true)

    # Attractiveness of the move, calculated a priori
    η = (1 ./dist_mat) .^ β
    # Trail level
    τ = ones(size(dist_mat)) .^ α
```

```
    best_path = nothing
    best_cost = Inf
    no_improv = 0
    n_nodes = size(dist_mat,1)

    for i in 1:max_iter

        solutions = []
        # Create the probability matrix
        P = η.*τ
        P ./= sum(P,dims=2)

        # Create a dictionary with Process and Future
        # to fetch later
        par_results = Dict{Int,Future}()
        @sync for k in 1:nants
            # Choose which process will calculate this path
            w = mod1(k, nprocs())
            # Spawn the call at `w` process
            @async par_results[k] = @spawnat w begin
                start_node = rand(1:n_nodes)
                end_node = start_node
                path = travel(P, start_node)
                cost = sum(edge_distances(dist_mat, path))
                path,cost
            end
        end

        for k in 1:nants
            w = mod1(k, nprocs())
            # Retrieve the call from `w` process calculated earlier
            path,cost = fetch(par_results[k])
            push!(solutions, (cost,path))
        end

        # Sort solutions
        sort!(solutions)
        best_local_cost, best_local_path = solutions[1]
        # Check if found a better cost path
        if best_local_cost < best_cost
            if verbose
                println("Better solution found with cost
                $(best_local_cost) at iteration $(i)")
            end
            best_path = best_local_path
            best_cost = best_local_cost
            no_improv = 0
        else
            no_improv += 1
        end

        # deposit pheromones
        τ .*= (1-ρ)
        for (cost, path) in solutions
            Δτ = Q / cost
            for (from, to) in edges(path)
                τ[to, from] += Δτ
            end
        end
    end
    best_path,best_cost
end

# Fix Random seed
Random.seed!(123)

# Create weights of the graph
n = 300
dist_mat = rand(n, n)
```

```
for i in 1:n
    # No self loop (1/Inf = 0.0)
    dist_mat[i,i] = Inf
end
max_iter = 50;
nants = 100;
verbose = false

(best_path_ser,best_cost_ser),Δt_ser = @my_time ACO(dist_mat,max_iter,nants,
    verbose=false)
(best_path_ser,best_cost_ser),Δt_ser = @my_time ACO(dist_mat,max_iter,nants,
    verbose=false)
println("Serial:   best_cost = ",best_cost_ser)
println("Serial:   time = ",Δt_ser)
(best_path_par,best_cost_par),Δt_par = @my_time ACO_parallel(dist_mat,max_iter,
    nants,verbose=false)
(best_path_par,best_cost_par),Δt_par = @my_time ACO_parallel(dist_mat,max_iter,
    nants,verbose=false)
println("Parallel: best_cost = ",best_cost_par)
println("Parallel: time = ",Δt_par)
```

Running the serial versus the parallel version we can observe an speed up of more than 12 times, as shown in Listing 2.16.

Listing 2.16. Ant Colony Speed up

```
  $ julia -p 88 .\aco_parallel.jl
Serial:   best_cost = 59.01948498666938
Serial:   time = 208.416440638
Parallel: best_cost = 57.180082172457624
Parallel: time = 16.431587189
```

2.5.4. Final remarks on distributed computing with Julia

We have seen that Julia permits high and low-level control on distributed computed and how to perform calculations on both cases.

We would like to bring attention to the fact that, a typical **Cloud service** (as AWS, Azure, etc) for computation is just a cluster that can allow distributed and multicore computations. It is then possible to run a Julia application on such services. You can use the `Distributed` package or specific interfaces like `MPI.jl` to parallelize the code. In order to make your code reproducible in a cluster that does not have Julia installed, you can create an executable app built by `PackageCompiler`[8].

Julia has integration with cloud machine learning platforms like Amazon Sagemaker [9] and Azure (with a preconfigured Julia installation). Also, there is a specific Julia platform for cloud computing called JuliaHub [10].

2.6. Multicore and distributed speed up analysis

Once we have seen how to adapt serial code to its parallel version under the multicore and distributed settings, and we have slightly touched on their speed up performance for

[8] https://github.com/JuliaLang/PackageCompiler.jl
[9] https://d1.awsstatic.com/whitepapers/julia-on-sagemaker.pdf
[10] https://juliahub.com/lp/

one single setup of workers/threads/processes, we decided to illustrate how three of the previously described algorithms behave under a variable number of processes.

The table in Figure 2.2 presents the running time in seconds for 8 different number of processes. Then, figure 2.3 depicts the speed up obtained under such settings. The experiments were conducted using the partition *Blaise* on the **PCAD** cluster (see Section 2.8.1), a computational node with the following specifications: 2*x* Intel Xeon E5-2699 v4 Broadwell (Q1'16) 2.2 GHz, each one with 44 physical cores, adding up to 88 threads and 256 GB DDR4 RAM. The speed up analysis described here must not be taken as the best possible behavior Julia could provide us. The goal here is simply illustrate the behavior of the described techniques along these notes while using Julia for parallelizing code.

	Number of Processes							
Instance	1	2	4	8	16	32	64	88
Selection Sort (Threads)	2.987	2.233	1.385	0.966	0.596	0.456	0.328	0.298
Pi Monte Carlo (Distributed)	8.446	4.225	2.185	1.197	0.721	0.411	0.312	0.309
Ant Colony Optimization (Distributed)	208.4	141.0	85.9	52.8	30.7	20.2	17.1	16.4

Figure 2.2. Table with execution time of the instances in seconds.

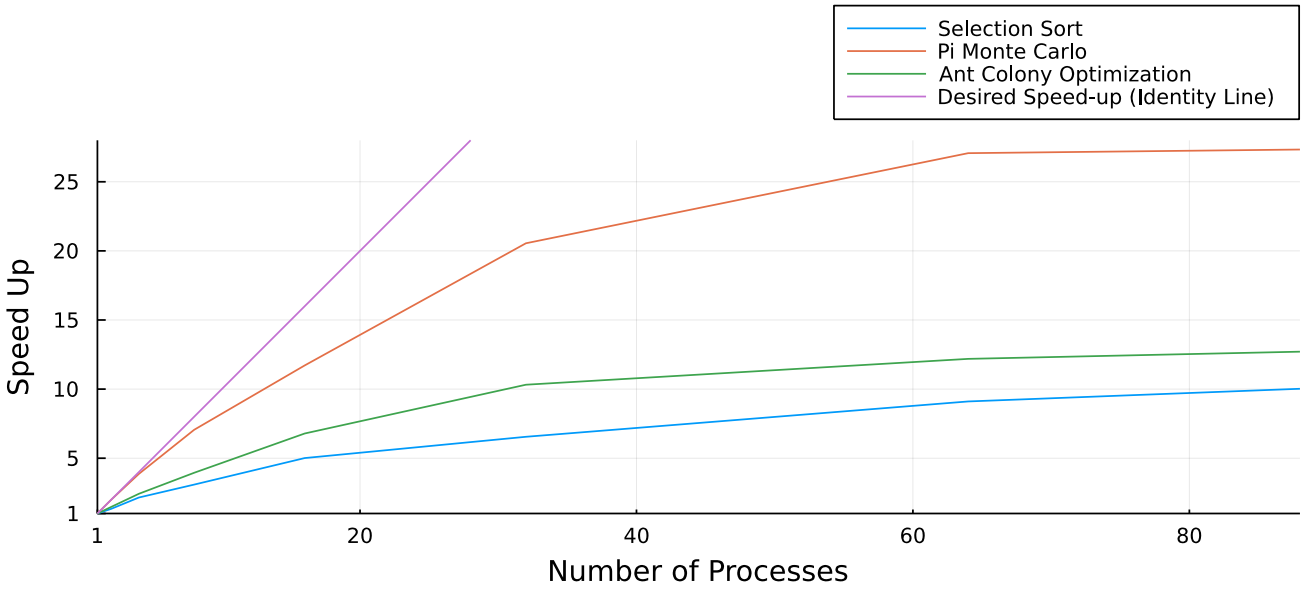


Figure 2.3. Speed Up

2.7. High-level GPU programming in Julia

High Performance and significant energy savings can be achieved by combining multicore processors and Graphics Processing Units (GPUs) [Nasciutti, Panetta e Lopes 2019, Serpa et al. 2019, Serpa et al. 2021]. It has become a popular choice for the top-ranked

HPC systems and for the yet to come exascale platforms [Dongarra e Strohmaier 2021]. GPUs have thousands of processing elements and different memory levels. Julia has support for programming in GPUs and can deliver high performance on these architectures.

Along this section, you need to have a computer with a compatible GPU and have installed CUDA [11]. We present here examples on how to use one single GPU only for computation with Julia, but the same package allows computations on multiple GPUs.[12]

Since we are restricting ourselves to work with one single GPU, GPU programming with Julia resembles multicore programming with CPUs. All examples of this section were computed using a NVidia Tesla P100, Pascal GPU with 3584 CUDA threads.

On the first example, Listing 2.17, we demonstrate GPU computations through high-level programming using the `CUDA.@sync` macro. The function of interest is just a simple sequential addition. We depict a serial version, a parallel one using threads, and a CUDA one. The only difference on the CUDA version of the function is the use of `@Cuda.@sync` . Also, at the moment of calling the function with CUDA version, we make use of `CUDA.fill` instead of the regular `fill` in Julia.

Desiring to measure the computational performance of these three versions, we use the macro `@benchmark`, from the **BenchmarkTools** package. This outputs on screen the behavior of the application. We can see in Figure 2.4 how such three versions performed.

Listing 2.17. Benchmarks

```
using CUDA
using Base.Threads
using BenchmarkTools


function sequential_add!(y, x)
    for i in eachindex(y, x)
        @inbounds y[i] += x[i]
    end
    return nothing
end

function parallel_add!(y, x)
    Threads.@threads for i in eachindex(y, x)
        @inbounds y[i] += x[i]
    end
    return nothing
end

function gpu_add!(y, x)
    CUDA.@sync y .+= x
    return
end


N = 2^23
x = fill(1.0f0, N)
y = fill(2.0f0, N)

println("Sequential Add:")
bench1 = @benchmark sequential_add!(x,y)
```

[11] https://developer.nvidia.com/cuda-downloads

[12] You can check also https://arxiv.org/pdf/1604.03410.pdf for further analysis on GPU programming with Julia, comparisons with C++ language and speed up analyses.

```
display(bench1)
println()

x = fill(1.0f0, N)
y = fill(2.0f0, N)

println("Parallel Add:")
bench2 = @benchmark parallel_add!($x,$y)
display(bench2)
println()

x_d = CUDA.fill(1.0f0, N)  # a vector stored on the GPU filled with 1.0 (Float32
    )
y_d = CUDA.fill(2.0f0, N)  # a vector stored on the GPU filled with 2.0

println("GPU Add:")
bench3 = @benchmark gpu_add!($x_d,$y_d)
display(bench3)
println()
```

The parallel version using threads is about 21 times faster, while the GPU version is about 28 times faster, all compared to the serial version. Also, the GPU version is about 30% faster than the parallel CPU version.

2.7.1. Example: Mandelbrot set

The next example constructs a plot of the Mandelbrot set[13]. This is the set of all complex numbers c for which the sequence defined by the quadratic recurrence equation

$$z_{n+1} = z_n + c,$$

with $z_0 = c$, is bounded. The plot is constructed by a simple escape time algorithm: the recurrence is evaluated for each (x,y) point in the plot area and depending on how fast it diverges, a color is given to the point. We present both the serial and the GPU versions of functions for such calculations. We also use the GPU code to produce the final picture of the set, seen in Figure 2.5. The analysis is inspired on https://github.com/vinifda/Mandelbrot-julia/blob/main/src/Mandelbrot_gpu.ipynb.

The main differences between the serial and the GPU version is the use of the `CUDA.CuArray` type, instead of the simple `Array` type Julia has by default, and specific CUDA programming fuctions, that are intuitive, and calculate the same as their homologue ones, as `CUDA.abs2`, `CUDA.Complex`, `CUDA.@time`. As in the first example, we also need to make use of `CUDA.@sync`.

Listing 2.18. MandelBrot CUDA

```
using CUDA, Images

function coloring(n)
    a = 0.1;
    return RGB(0.5 * sin(a * n) + 0.5,
            0.5 * sin(a * n + 2.094) + 0.5,
            0.5 * sin(a * n + 4.188) + 0.5)
end
```

[13] https://mathworld.wolfram.com/MandelbrotSet.html

```
/home/users/rbklausner/Julia_HPC
Sequential Add:
BenchmarkTools.Trial: 878 samples with 1 evaluation.
 Range (min … max):  5.638 ms …   7.765 ms  ┊ GC (min … max): 0.00% …
0.00%
 Time  (median):     5.654 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   5.680 ms ± 158.604 µs  ┊ GC (mean ± σ):  0.00% ±
0.00%
```

```
  5.64 ms      Histogram: log(frequency) by time      6.73 ms <

 Memory estimate: 0 bytes, allocs estimate: 0.
Parallel Add: 88 threads
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  226.814 µs …  16.809 ms  ┊ GC (min … max): 0.00% …
93.12%
 Time  (median):     251.220 µs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   270.542 µs ± 336.821 µs  ┊ GC (mean ± σ):  2.90% ±
2.32%
```

```
  227 µs           Histogram: frequency by time          336 µs <

 Memory estimate: 42.05 KiB, allocs estimate: 441.
GPU Add:
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  201.902 µs … 251.993 µs  ┊ GC (min … max): 0.00% …
0.00%
 Time  (median):     205.429 µs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   205.669 µs ±   1.730 µs  ┊ GC (mean ± σ):  0.00% ±
0.00%
```

```
  202 µs           Histogram: frequency by time          210 µs <

 Memory estimate: 1.61 KiB, allocs estimate: 23.
```

Figure 2.4. Benchmarks

```
# Image definitions
width, height = 301*2,241*2
width -= 1
height -= 1
xmin = -2.2
xmax = 0.8
ymin = -1.2
ymax = 1.2
h_x = (xmax-xmin)/width
h_y = (ymax-ymin)/height
xs = xmin:h_x:xmax
ys = ymin:h_y:ymax
xsize = length(xs)
ysize = length(ys)
maxiter = 1024;

# Serial function to compute the number of iterations
# necessary to the recurrence diverge
function get_steps(c::Complex, max_steps::Int64)
    z = Complex(0.0, 0.0) # 0 + 0im
    for i=1:max_steps
        # Calculate the recurrence
        z = z^2+c
        # Diverged, return the number of iterations
        if abs2(z) >= 400
            return i
        end
    end
    # The recurrence is bounded
    return max_steps+1
end

mandel(x,y) = get_steps(Complex(x, y), maxiter)
@time Z = [mandel(x,y) for y in ys, x in xs];
@time Z = [mandel(x,y) for y in ys, x in xs];

# GPU function to compute the number of iterations
# necessary to the recurrence diverge
function get_steps_gpu(c::CUDA.Complex, max_steps::Int64)
    z = CUDA.Complex(0.0, 0.0) # 0 + 0im
    for i=1:max_steps
        z = z^2+c
        if CUDA.abs2(z) >= 400
            return i
        end
    end
    return max_steps+1
end

Z = CUDA.CuArray([Complex(x,y) for y in ys, x in xs]);
cu_steps = CuArray(zeros(Int, (length(ys), length(xs))));
CUDA.@time CUDA.@sync cu_steps .= get_steps_gpu.(Z, maxiter);
CUDA.@time CUDA.@sync cu_steps .= get_steps_gpu.(Z, maxiter);
values = Array(cu_steps);
img = coloring.(collect(values))
save("mandelbrot_CUDA.png", img);
```

```
  $ julia --project=.. .\mandelbrot.jl
  0.312863 seconds (389.37 k allocations: 22.755 MiB, 31.05% compilation time)
  0.258803 seconds (87.25 k allocations: 6.455 MiB, 16.52% compilation time)
 19.873111 seconds (35.58 M CPU allocations: 1.870 GiB, 4.33% gc time)
  0.000968 seconds (23 CPU allocations: 1.281 KiB)
```

The GPU version is about 270 times faster than the serial one. The CUDA first call takes a long time to compile and should not be taken into account to make a time comparison.

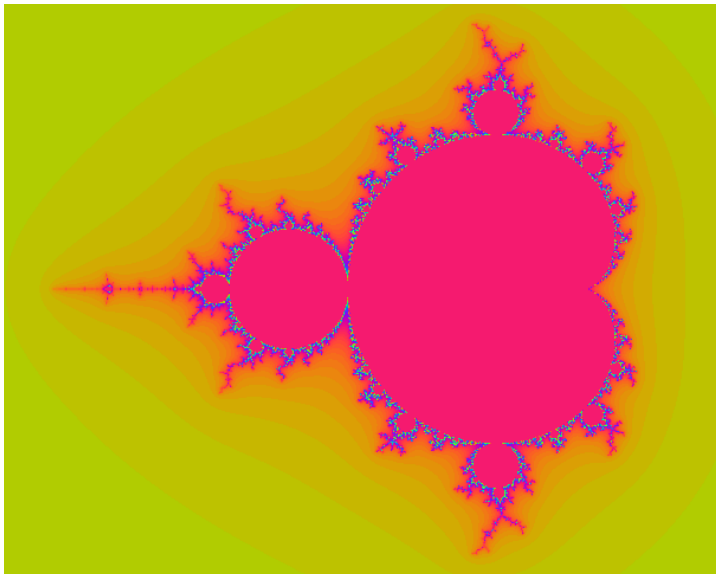


Figure 2.5. Mandelbrot plot

2.8. Conclusion

Along these notes we presented the use of the Julia language for HPC systems. We covered from its installation on a typical HPC cluster to parallel programming via multicore, distributed, and GPU architectures. Our motivation for the notes was the lack of a good reference with easy to access and updated to the current version of Julia for students and enthusiasts of HPC systems. Also, the complete absence of Julia on courses for HPC systems. We expect that these notes can serve as a first step for the HPC user that wishes to use Julia or for the Julia user that would like to speed up his/her applications running in parallel.

2.8.1. Acknowledgements

R.M. Velho would like to acknowledge Prof. Eduardo S. L. Gastal from the Institute of Informatics of the Federal University of Rio Grande do Sul (UFRGS) - Brazil for the long transfer of knowledge on Julia language and its use for R&D

A.M.A. Côrtes would like to acknowledge Prof. Ricardo Rosa from the Applied Mathematics Department of the Federal University of Rio de Janeiro (UFRJ) - Brazil for helpful discussions regarding Julia's performance.

All the authors would like to thank the High Performance Computing Group (PCAD) http://gppd-hpc.inf.ufrgs.br/ at the Federal University of Rio Grande do Sul - Brazil for making available the use of its multiple clusters while the codes used in these notes were written and tested.

Several examples were inspired on the book [Sengupta 2019]. One of the few published book on the topic of HPC with Julia. We thank the author for the effort on gathering all that information.

References

[Carrijo Nasciutti et al. 2019] Carrijo Nasciutti, T., Panetta, J., and Pais Lopes, P. (2019). Evaluating optimizations that reduce global memory accesses of stencil computations in gpgpus. *Concurrency and Computation: Practice and Experience*, 31(18):e4929. páginas

[J. Dongarra and Strohmaier 2021] J. Dongarra, H. M. and Strohmaier, E. (2021). Top500 supercomputer: November 2021. https://www.top500.org/lists/2021/11/. [Accesed in: 3 Mar. 2022]. páginas

[Sengupta 2019] Sengupta, A. (2019). *Julia High Performance Optimizations, distributed computing, multithreading, and GPU programming with Julia 1.0 and beyond*. Packt Publishing. páginas

[Serpa et al. 2019] Serpa, M. S., Moreira, F. B., Navaux, P. O., Cruz, E. H., Diener, M., Griebler, D., and Fernandes, L. G. (2019). Memory performance and bottlenecks in multicore and gpu architectures. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 233–236. IEEE. páginas

[Serpa et al. 2021] Serpa, M. S., Pavan, P. J., Cruz, E. H., Machado, R. L., Panetta, J., Azambuja, A., Carissimi, A. S., and Navaux, P. O. (2021). Energy efficiency and portability of oil and gas simulations on multicore and graphics processing unit architectures. *Concurrency and Computation: Practice and Experience*, 33(18):e6212. páginas

[Yoo et al. 2003] Yoo, A. B., Jette, M. A., and Grondona, M. (2003). Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer. páginas

Capítulo 3

Introdução à Programação com Memória Persistente

Alexandro Baldassin
Universidade Estadual Paulista (UNESP)
Rio Claro, Brasil

Emilio Francesquini
Universidade Federal do ABC (UFABC)
Santo André, Brasil

Resumo

Este minicurso é uma introdução à programação com memória persistente (PM). Nele apresentaremos a motivação para uso dessa nova tecnologia; o suporte atual disponibilizado por processadores e sistemas operacionais; os problemas de consistências de dados que podem acontecer; e como utilizar abstrações de mais alto nível (como transações) para resolvê-los. O minicurso também apresenta uma série de exemplos práticos utilizando o Intel PMDK para programação de estruturas de dados persistentes.

3.1. Introdução

As tecnologias para memória persistente endereçada a byte (PM)[1] têm sido exaustivamente pesquisadas nos últimos anos [3]. Como resultado, em 2019, a Intel colocou no mercado o primeiro dispositivo dessa linha, conhecido como *Intel Optane DC* [33].

Quando comparadas à tradicional tecnologia de memória DRAM, essas novas memórias possuem maior densidade, menor custo por bit e um desempenho ligeiramente inferior (principalmente quanto às operações de escrita) [3]. O fato destas novas tecnlogias disponibilizarem acessos na granularidade de byte permite que programadores acessem tal memória diretamente, sem o uso de interfaces convencionais como sistemas de arquivos ou um banco de dados. No entanto, uma falha (como queda de energia) pode acontecer a qualquer momento e tornar os dados inconsistentes. Dessa forma, programar com PM requer cuidados extras quando comparado à forma usual empregada com

[1]Também conhecidas como memórias não voláteis (NVM) ou *Storage Class Memory* (SCM).

memória volátil. O grande desafio atual é como prover abstrações para a programação desses novos dispositivos persistentes [31].

Considerando a recente introdução de PM no mercado e os desafios relacionados à sua programação, argumentamos que esse tema é de extrema relevância e que o suporte atual para sua programação deva ser difundido, uma vez que tais dispositivos tendem a ser utilizados em sistemas de alto desempenho [26]. Estudos recentes mostram uma acentuada quantidade de trabalhos científicos com foco em modelos de programação para PM [2], ressaltando ainda mais a importância do tema. Apesar do minicurso ter um cunho bem prático, também serão mencionados tópicos de pesquisas mais recentes, o que pode abrir oportunidades para os interessados em começar algum estudo sobre o tema, tanto no nível de graduação quanto de pós-graduação.

Atualmente já é possível adquirir máquinas nas quais a PM divide espaço com a DRAM no papel de memória de trabalho. Neste minicurso, vamos explorar o uso de uma máquina com essas características, e nos concentraremos na tecnologia 3D Xpoint da Intel e no Intel PMDK (*Persistent Memory Development Kit*) que são o padrão *de facto* do mercado. O restante do documento está organizado como descrito a seguir. Um detalhamento mais aprofundado sobre os conceitos de persistência são descritos na Seção 3.2. Em seguida, na seção Seção 3.3, descrevemos alguns dos desafios envolvidos com o uso dessa tecnologia. Exemplos práticos do uso do Intel PMDK são mostrados na Seção 3.4, incluindo o desenvolvimento de uma lista ligada simples persistente. O minicurso é encerrado na Seção 3.5.

3.2. Conceitos de Persistência

Nesta seção descrevemos brevemente algumas tecnologias de memória para em seguida falarmos mais especificamente de conceitos de persistência relevantes à utilização de uma máquina com PM.

3.2.1. Tecnologias de memória

Static Random-access Memory (SRAM), *Dynamic Random-access Memory* (DRAM) e Flash são as tecnologias não mecânicas para armazenamento de memória que atualmente desfrutam das mais amplas aplicações no mercado. Essas tecnologias são empregadas desde dispositivos móveis, como telefones celulares, até computadores para processamento científico de alto desempenho. Além de serem amplamente utilizadas, são tecnologias muito maduras que já estão disponíveis há décadas (a tecnologia Flash, a mais recente dentre as três, surgiu há pelo menos 35 anos [20]). Cada uma destas tecnologias conta com certas vantagens e desvantagens dependendo da aplicação. Por exemplo, a tecnologia SRAM é rápida porém possui um gasto energético superior àquele da DRAM; Flash, ao contrário das demais, não é volátil; e DRAM permite maiores densidades do que SRAM (por utilizar apenas um transistor e um capacitor por célula enquanto SRAM utiliza ao menos seis). No entanto, em arquiteturas de hardware reais, usuários procuram por um bom desempenho, por um gasto de energia aceitável e também por um preço competitivo [1]. Portanto, em plataformas de hardware reais, há um compromisso entre as limitações e vantagens de cada tipo de memória aliado às necessidades práticas para a sua fabricação. Tal conjunto de restrições e compromissos motivou os projetistas de

hardware a utilizar soluções híbridas onde cada um destes tipos de memória é empregado para desempenhar papéis nos quais o seu desempenho é o melhor. Esta solução hibrida é claramente evidenciada pelos níveis hierárquicos de memória presentes nos atuais computadores.

Nas modernas arquiteturas de hardware pode ocorrer uma considerável variação no tempo de acesso aos dados dependendo da distância hierárquica entre os núcleos de processamento e a localização dos dados. Por esta razão memórias menores (mais rápidas porém com um sobrecusto energético significativo) são utilizadas para mascarar o tempo de acesso às memórias com maiores capacidades (com menor gasto energético porém maior tempo de acesso). Atualmente, processadores típicos possuem algumas dezenas de registradores baseados em SRAM, alguns níveis de memória cache localizadas no próprio processador (SRAM), a memória principal de trabalho baseada em DRAM e, finalmente, um dispositivo de armazenamento não volátil que normalmente se apresenta na forma de um disco rígido ou *Solid-state Drive* (SSD) baseado em memória Flash. Em algumas plataformas de hardware, *e.g. Non-Uniform Memory Access* (NUMA), pode existir ainda um nível hierárquico adicional entre os núcleos de processamento e algumas regiões da memória.

Não apenas os usuários destes sistemas computacionais estão interessados em desempenhos cada vez mais expressivos (que se traduzem, em parte, por tempos reduzidos de acesso à memória), como também estão interessados em memórias com capacidades cada vez maiores. Esta é uma tendência que não dá sinais de mudança a curto prazo [19]. No entanto, as tecnologias de hardware atuais para a memória principal baseadas em DRAM, por mais maduras que estejam, estão alcançando os seus limites em termos de escalabilidade [14, 15] por fatores intrínsecos à maneira pela qual operam. Os problemas enfrentados pelos engenheiros de hardware envolvem desde a criação de células de memória cada vez menores, até taxas de *refresh* que atualmente são impraticáveis.

A necessidade por maiores capacidades e desempenho dos mais modernos sistemas computacionais levaram os arquitetos de hardware a buscar alternativas para substituir as já consagradas tecnologias de armazenamento de memória. Entre as principais novas tecnologias alternativas competidoras estão versões mais modernas da memória Flash [5, 23, 32], *Ferroelectric RAM* (FeRAM) [4], *Spin-Torque Transfer RAM* (STT-RAM) [9, 24], *Phase-Change Memory* (PCM) [18, 27] e, mais recentemente, Intel 3D XPoint [11][2] lançada sob o nome comercial Intel Optane DC. Diferentemente da tecnologia DRAM, essas tecnologias alternativas têm como característica a não volatilidade dos dados, ou seja, não dependem de um fluxo contínuo de alimentação elétrica para a manutenção dos dados.

Estas tecnologias são genericamente classificadas como *Storage Class Memory* (SCM), *Non-Volatile Memory* (NVM), ou simplesmente *Persistent Memory* (PM). PMs, principalmente aquelas baseadas em Flash, têm sido utilizadas para a fabricação de dispositivos de armazenamento secundários, um território (ainda) dominado por dispositivos de armazenamento mecânicos como discos rígidos. A principal vantagem vinculada ao

[2] A tecnologia 3D XPoint é, de fato, uma tecnologia PCM. Contudo ela tem diversas características como ter várias camadas empilhadas [16] (daí o nome 3D) que justificam a sua classificação em uma categoria diferente das PCMs tradicionais.

Características	DRAM	XPoint	FeRAM	STT-RAM	PCM	NOR Flash	NAND Flash
Tamanho da célula (F^2)	6–8	4	15–32	36	4–12	10	4
Tamanho do bloco/latência	N.A.	N.A.	N.A.	N.A.	N.A.	64–256KB 900ms	8–64KB 2ms
Tamanho da página	64b	N.A.	64b	64b	64b	64b	512b–4KB
Latência de leitura	10's ns	10's ns	10's ns	10's ns	10's ns	100's ns	10's μs
Latência de escrita	10's ns	10's μs	10's ns	10's ns	10's μs	10's μs	100's μs
Não-volatilidade	não	sim	sim	sim	sim	sim	sim
Durabilidade (Nº. escritas)	10^{16}	10^8	10^{14}	10^{15}	10^8–10^9	10^5	10^5

Tabela 3.1: Comparação entre DRAM [22, 13, 14], XPoint [11, 16], FeRAM [6, 7, 17], STT-RAM [9, 24, 35], PCM [18, 27] e Flash [5, 23, 32].

uso de PMs é uma capacidade potencialmente muito maior do que as que são possiveis com o uso de DRAM (apesar de possuírem velocidades de acesso inferiores) e velocidades muito maiores de acesso do que dispositívos de bloco (apesar de ter uma capacidade de armazenamento inferior). Em outras palavras, PMs se encaixam como uma tecnologia intermediária entre os atuais dispositivos de bloco e a as memórias de trabalho que são tipicamente representadas pela tecnologia DRAM.

Algumas dessas novas tecnologias, como por exemplo PCM, já possuem desempenhos comparáveis à DRAM. A Tabela 3.1 apresenta um resumo das características de algumas dessas tecnologias de memória. Apesar de Flash estar entre as tecnologias de memória com o maior amadurecimento atualmente, ela tem uma durabilidade limitada. No entanto, algumas projeções [14, 15] já indicam que as mais novas tecnologias alternativas de memória superarão o desempenho e escalabilidade de tecnologias tradicionais como Flash em alguns anos, como já é, inclusive, o caso da 3D XPoint.

O ritmo de desenvolvimento destas novas tecnologias indica que o desempenho dos dispositivos de armazenamento baseados nessas novas tecnologias cada vez mais deve se aproximar do desempenho das memórias de trabalho, hoje baseadas quase que exclusivamente[3] em DRAM. Quando levamos em consideração a natureza hierárquica dos computadores atuais e o desempenho das mais novas tecnologias de memória, nos damos conta que um passo natural após a substituição da tecnologia dos dispositivos de armazenamento secundários por memórias persistentes é a utilização de tecnologias de memória com maior escalabilidade no próximo nível hierárquico: a memória de trabalho. Com exceção da memória Flash[4], novas tecnologias de memória persistente já podem ser consideradas como possíveis substitutas à tecnologia tradicional.

[3]Em alguns dispositivos embarcados o uso de memória persistente como memória de trabalho já é uma realidade.

[4]Sua baixa durabilidade a torna inadequada para uma memória ativa de trabalho.

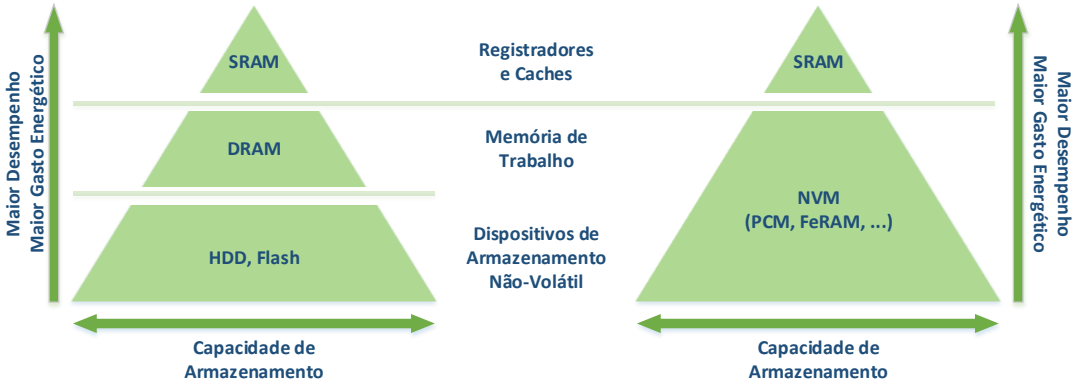


Figura 3.1: Diagrama simplificado dos níveis hierárquicos de memória. Nesta figura são comparados os níveis hierárquicos de memória e suas funções entre uma arquitetura tradicional (esquerda) com uma arquitetura baseada em memória universal (direita). Em uma arquitetura com memória universal, a memória de trabalho é a mesma utilizada para o armazenamento permanente de dados.

3.2.2. O papel das PMs na hierarquia de memória

Pesquisas sobre os estágios de adoção de PM têm sido feitas considerando tanto substituições parciais [27, 36, 29, 28] quanto totais [18, 35, 37] da DRAM por PMs. Uma arquitetura com *memória universal* é, simplificadamente, uma arquitetura na qual os níveis de memória de armazenamento secundário e de trabalho foram condensados em apenas um [1]. A Figura 3.1 ilustra simplificadamente as diferentes funções de cada um dos níveis hierárquicos de memória comparando uma arquitetura tradicional a uma arquitetura com memória universal.

Ainda assim, é muito provável que leve alguns anos até que uma máquina comercial com estas características chegue ao mercado [21]. Em 2014 a Hewlett-Packard anunciou [10, 34, 8] ter dado início um a um projeto de pesquisa para a construção de um protótipo com tal arquitetura (apesar de, até hoje, não existir um produto amplamente disponível no mercado com essas características).

As atuais limitações de desempenho, durabilidade, disponibilidade e custo das PMs quando comparadas à DRAM (uma tecnologia madura e já produzida em larga escala) fazem com que a adoção dessas novas tecnologias seja gradual, primeiramente com PMs lado a lado com DRAM, em seguida substituindo-a completamente para finalmente estar presente em todos os níveis hierárquicos de memória.

A Figura 3.2 mostra algumas das possíveis combinações. No caso *(a)*, vemos o uso tradicional de memória DRAM como memória de trabalho exclusiva, sem a presença de PMs e com dispositivos de armazenamento secundários tradicionais. Em *(b)*, contudo, vemos a presença de PM em total substituição à memória volátil. Este cenário é o mais próximo de uma máquina com arquitetura universal de memória. No entanto, tal máquina ainda possui as caches e os registradores do processador utilizando tecnologias de memória voláteis como, por exemplo, SRAM. Este cenário, onde há a substituição completa de DRAM por PM, deve ainda levar alguns anos para ocorrer de maneira ampla

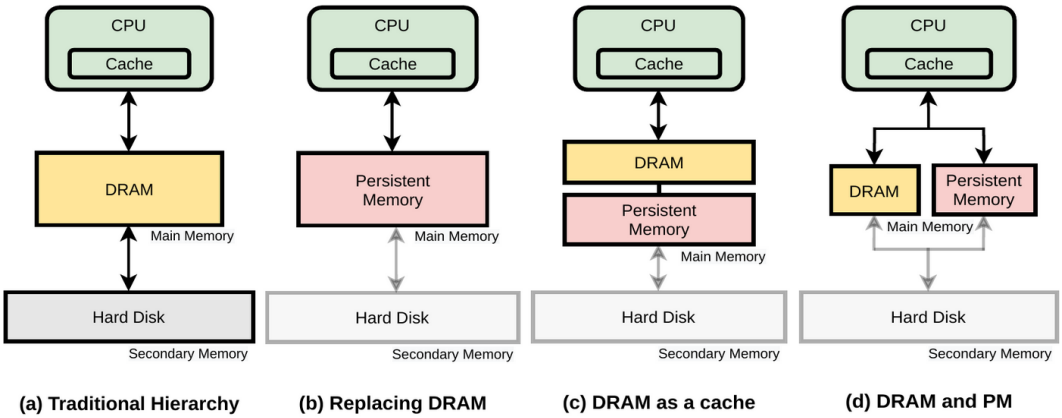


Figura 3.2: Possíveis papeis da PM na hierarquia da memória principal. (a) Arquitetura tradicional. (b) Como uma substituta para DRAM (c) Utilizando DRAM como uma cache (d) DRAM e PM lado a lado. Adaptado de [30].

simplesmente porque a tecnologia de DRAM ainda possui um melhor desempenho do que as PMs. O caso *(c)* tenta combinar o desempenho superior de DRAM com a capacidade superior das PMs, assim servindo como uma cache para acelerar os acessos. De fato, neste cenário pode-se pensar na DRAM como um nível adicional de cache (externo) ao processador. Finalmente, no caso *(d)*, a PM trabalha lado a lado com a DRAM. Logo, a escolha de qual memória deve ser utilizada (e quando) fica a cargo do software (do sistema operacional e das aplicações). Embora as máquinas atuais equipadas com Optane DC possam trabalhar nas configurações *(c)* e *(d)*, as pesquisas tendem a se concentrar na opção *(d)*, já que essa configuração expõe a PM diretamente ao programador.

3.3. Desafios na Programação

A Figura 3.3 apresenta os modos de acesso a dispositivos não-voláteis. O primeiro modo de uso (a) emprega uma API de arquivos, ou seja, o dispositivo é exposto para as aplicações como se fosse um arquivo (tipicamente localizado em `/dev/pmem` em sistemas Linux). A segunda interface de programação *(b)* é através de uma API de consultas, por exemplo fornecida por um banco de dados, que enxerga a dispositivo persistente como um arquivo armazenado em um dispositivo de armazenamento em blocos. Finalmente, a terceira interface *(c)* enxerga a PM como uma área de memória, a ser utilizada diretamente como se fosse memória de trabalho convencional, fazendo acessos de leitura e escrita diretamente da aplicação.

Os modos de acesso *(a)* e *(b)*, embora tradicionais, foram construídos para dispositivos acessados em blocos. Eles requerem duas representações dos dados para uma estrutura de dados: uma para a versão em DRAM e outra serializada para ser armazenada no dispositivo de armazenamento em bloco. Por esta razão é necessário converter entre um formato e outro durante o carregamento (de-serialização) e salvamento (serialização). A utilização desses dois modos não possui um mapeamento direto e óbvio com o uso da DRAM pelas aplicações. O modo *(c)*, que é o foco desde minicurso, acessa a PM diretamente, como se fosse a memória DRAM que estamos acostumados. Esta maneira

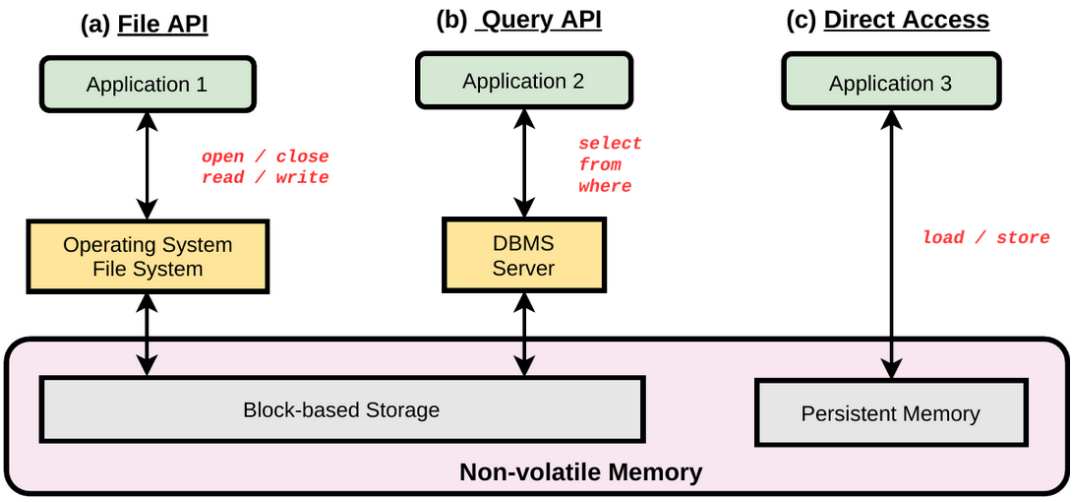


Figura 3.3: Principais modos de acesso à PM. (a) através de uma API de arquivos. (b) através de uma API fornecida, por exemplo, por um banco de dados que acessa a PM como um dispositivo de armazenamento de blocos. (c) Através de acesso direto, usando operações de load/store diretamente da aplicação. Adaptado de [30].

de acesso evita gargalos impostos pela pilha de software presente nos modos *(a)* e *(b)* (sistema de arquivos, sistema gerenciador do banco de dados, paginação do sistema operacional, etc). Enquanto os overheads associados com este tipo de acesso, em geral, são desprezíveis quando estamos lidando com dispositivos lentos como um disco rígido ou com um SSD, eles passam a ser um gargalo importante de desempenho quando utilizamos uma PM cujo desempenho supera muito (em termos de latência e banda) aqueles de SSDs e HDs típicos.

O uso de PM como uma memória de trabalho traz consigo desafios, alguns novos e outros bem conhecidos, para os desenvolvedores. O primeiro e mais óbvio é que torna-se preciso assegurar que acessos concorrentes aos dados sejam sincronizados e, assim, sejam evitadas condições de corrida. Outros, contudo, são desafios que aparecem devido às interações da PM com os demais níveis de memória. Nós os exploramos na próxima seção.

3.3.1. Interações com as *caches*

Suponha que tenhamos um programa simples, que apenas quer armazenar um valor em uma posição de memória. Um possível código que efetua esta tarefa é mostrado abaixo (à esquerda) e o que será efetivamente executado pelo processador será algo semelhante ao que está à direita:

```
x = 1;
```

```
STORE X, 1
```

Após a execução deste código, desejamos que como resultado o valor `1` seja armazenado na PM, conforme a Figura 3.5a. Porém, na verdade, o que provavelmente ocorrerá é o cenário mostrado na Figura 3.5b.

(a) Comportamento desejado. (b) Comportamento provável.

Figura 3.5: Interações da cache com a PM. Uma operação de escrita não altera imediatamente a PM, mas sim a cache do processador.

A situação mostrada na Figura 3.5b pode ser perigosa pois, apesar da visão dos processos em execução que têm acesso à essa posição de memória ser consistente e todos verem o valor `1` (garantido pelo modelo de memória do processador/protocolos de coerência de cache), uma possível falha no fornecimento de energia neste ponto da execução implicaria a perda do valor armazenado já que as caches são voláteis.

De fato existem instruções específicas que ignoram a cache e que permitem fazer *loads* e *stores* diretamente na memória de trabalho. Em arquiteturas Intel tais instruções são chamadas de *Non-temporal*. Essas instruções, contudo, trazem consigo uma penalidade considerável de desempenho além de forçarem o programador a trocar um programa relativamente simples, como uma atribuição a uma variável, pela chamada de uma função especial do compilador para a geração das instruções relevantes, incluindo barreiras de memória para tornar os resultados visíveis para todos os processadores.

Uma possível correção para o problema acima usando apenas barreiras seria a seguinte:

```
x = 1;
```

```
STORE X, 1
FLUSH
```

A instrução do tipo *flush* força o dado a sair da cache e ir para o próximo nível de memória. Há diferentes versões de flushes, como veremos adiante, mas é possível que o flush não envie o dado imediatamente para o próximo nível. Podemos imaginar que o flush age como se fosse uma chamada não bloqueante. O que significa que mesmo após executar o flush, em caso de falhas, pode ocorrer de alguns dados serem perdidos. Assim, para garantir que tudo está corretamente salvo, precisamos de uma barreira de memória (*fence*) após o flush.

Outro cuidado a ser tomado é durante otimizações do código que podem inadvertidamente introduzir bugs. Por exemplo, o uso de flushes e barreiras a cada operação de store pode deixar o programa lento. Logo é razoável supor que sua consolidação seja feita como uma possível otimização:

```
// x = 1;
// y = 1;

store X, 1
flush X
fence

store Y, 1
flush Y
fence
```

(a) Original

```
// x = 1;
// y = 1;

store X, 1
store Y, 1

flush X
flush Y
fence
```

(b) Otimizado

Contudo o código otimizado acima pode não ser válido caso exista uma necessidade (definida pela aplicação em questão) de que o valor de `X` seja armazenado antes do valor de `Y`. Em outras palavras, o código acima pode não obedecer a ordem dos stores em caso de falhas e é possível que apenas o valor de `Y` seja escrito, e portanto salvo, na PM.

A lista abaixo apresenta algumas instruções de flush e acesso não-temporal à memória disponíveis na ISA x86. Note que nem todas as instruções podem estar disponíveis em todos os processadores e que o comportamento de algumas pode variar dependendo da versão do processador.

- `CLFLUSH` - Faz o flush e invalida uma linha de cache. Comportamento serializado.
- `CLFLUSHOPT` - O mesmo que CLFLUSH mas não serializa (o que permite um certo nível de concorrência). Exige o uso de uma barreira após uma sequência de instruções deste tipo.
- `CLWB` - Comportamento semelhante ao CLFLUSHOPT, contudo não invalida (obrigatoriamente) a linha da cache. Em alguns modelos de processadores a linha é invalidada e em outros não. Exige uso de uma barreira.
- `Non-Temporal Stores` - Ignoram as caches e escrevem diretamente na PM. Exigem uso de barreira.
- `SFENCE` - Barreira de escrita. Garante que todas as operações que apareceram antes da barreira (*program order*) têm seus efeitos globalmente visíveis antes de efeitos de qualquer outra instrução de store após a barreira.
- `WBINVD` - (*kernel mode*) Faz o flush de todas as linhas de todas as caches e as invalida.

3.3.2. Domínios de persistência

Para tentar especificar (um pouco) melhor o modelo de memória oferecido pelas diferentes máquinas com suporte a PM e facilitar o trabalho dos desenvolvedores, a Intel divide as máquinas com este suporte em duas categorias chamadas domínios de persistência (*persistence domains*) ou domínios de proteção contra falhas de alimentação (*power-fail protection domain*). Esses domínios são mostrados na Figura 3.8.

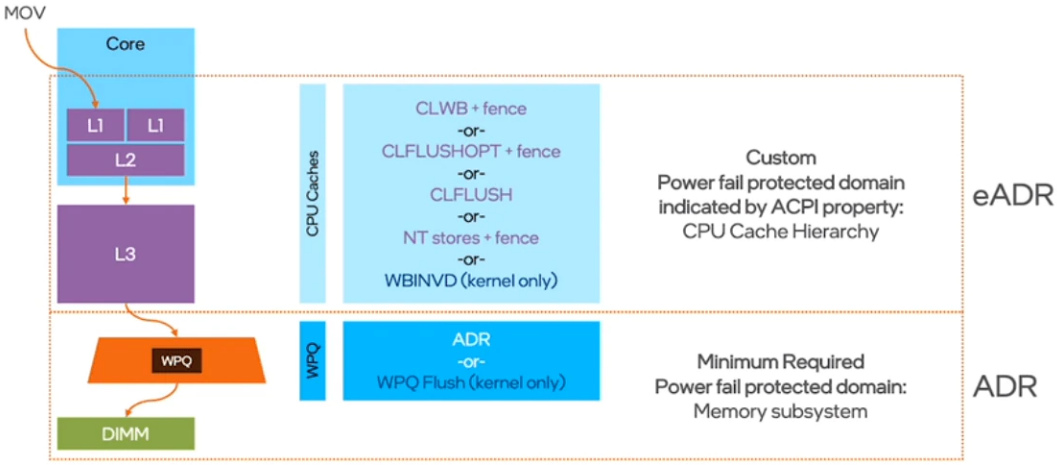


Figura 3.8: Domínios de persistência. Com ADR, os dados são persistentes assim que atingirem a WPQ. A eADR garante que os dados se tornem persistentes assim que atingirem as caches. Adaptado de [12].

O primeiro domínio de persistência chamado de *Asynchronous DRAM Refresh* (ADR) garante que dados que já chegaram até a PM, ou ainda, que estejam no controlador de memória (representado pela fila de escritas pendentes, *Write Pending Queue* - WPQ, na Figura 3.8) serão persistidos, ainda que haja uma falha. Para garantir que esses dados estejam em uma dessas localizações podemos utilizar uma combinação de flushes e barreiras de memória. As combinações corretas de quais instruções de flush e de barreira devem ser utilizadas dependem da máquina em questão.

O segundo domínio de persistência, chamado *Enhanced Asynchronous DRAM Refresh* (eADR), oferece todas as garantias oferecidas pelo domínio ADR, além de oferecer garantias que dados presentes nas caches dos processadores também serão escritos para a PM. Neste caso apenas os dados que ainda estejam nos registradores do processador seriam perdidos em caso de falhas. Essa garantia exige, contudo, uma fonte de energia auxiliar que seja capaz de manter o processador funcionando por alguns instantes para que as linhas sujas das caches possam ser enviadas para o controlador de memória e, em seguida, para a PM. Apesar deste domínio de persistência já ter suporte oficial de alguns processadores e placas-mães [12], ainda não está claro como eADR funcionaria já que não existem, até onde pudemos averiguar, soluções disponíveis no mercado para fornecer a energia que seria necessária para essa operação.

Atualmente há, de fato, apenas suporte de processadores da Intel para o uso de PM. A biblioteca PMDK, apoiada pela Intel, foi criada como uma solução para que os desenvolvedores que desejam utilizar PM não tenham que, repetidamente, lidar com problemas semelhantes aos descritos acima além de não precisarem estar, necessariamente, cientes das idiossincrasias de cada processador. Em particular, determinar que tipos de operações de memória (barreiras, flushing, logging, ...) estão disponíveis em cada processador é algo que foge do domínio da maior parte das aplicações que empregam PM. Some-se a isso o fato de que alguns comportamentos de instruções não são exatamente especificados pela documentação e que variam de acordo com a versão de processadores

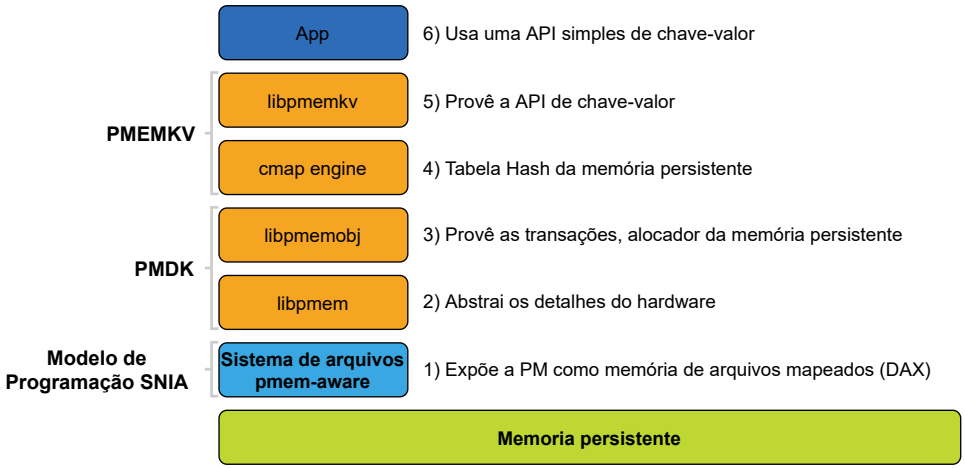


Figura 3.9: Pilha de software para programação com PM fornecida pela Intel. Este Adaptado de [30].

com a mesma ISA! Por estas razões recomendamos o uso do PMDK (ou de uma biblioteca semelhante). Neste curso concentraremo-nos no uso do PMDK apenas.

3.4. Programando com o Intel PMDK

Como apresentado nas seções anteriores, a programação para PM diverge de várias maneiras do modelo tradicional com DRAM. Esta seção apresenta de forma prática os principais pontos necessários para garantir a persistência dos dados usando o Intel PMDK (*Persistent Memory Development Kit*) [30]. Como a tecnologia de PM é relativamente recente, a pilha de software ainda é incipiente e há uma grande variedade de modelos sendo propostos na Academia (veja [2]). O Intel PMDK tem sido desenvolvido há uma década e a Intel optou por abrir o código fonte do projeto [5]. O projeto é atualizado com bastante frequência e é notável o interesse da Intel em prover essa infraestrutura de software dado que lançou recentemente no mercado dispositivos com PM (Intel Optane DC).

A Figura 3.9 mostra as camadas típicas fornecidas pela Intel para a programação com PM, separadas em vários níveis de abstração. Como o propósito desta seção é apresentar uma visão prática dos conceitos já discutidos, optou-se por focar na camada conhecida como `libpmemobj` (camada 3). Essa é a camada mais básica que provê os recursos essenciais para programação de estruturas de dados persistentes, em particular *transações*. O `libpmemobj` é uma biblioteca para linguagem C mas atualmente há alguns *bindings* para outras linguagens, como C++ e Java. Os exemplos utilizados neste texto usam a sintaxe original para linguagem C.

Nesta parte serão discutidos os conceitos chaves empregados pelo PMDK para a programação com PM. Estes podem ser divididos em: i) acesso à PM; ii) atualização dos dados persistentes; e iii) gerenciamento de memória. As próximas seções apresentam com um maior detalhe esses conceitos, seguido de um exemplo prático com a implementação

[5] Veja https://pmem.io/

de um estrutura de lista ligada persistente.

3.4.1. Acesso à PM

A PM é acessada por meio de um *pool de memória*, representado no PMDK pelo tipo `PMEMobjpool`. O pool de memória é uma abstração para a área de memória persistente que será gerenciada. Do ponto de vista do sistema operacional, o pool de memória é tratado como um arquivo convencional e pode ser criado pela ferramenta `pmemtool` disponibilizada pelo PMDK (recomendado) ou programaticamente. Uma vez criado, programadores devem explicitamente abrir o pool através de uma chamada específica fornecida pelo PMDK antes de trabalhar com seu conteúdo. Essa chamada retornará um identificador, comumente chamado de *pool object pointer* (pop), utilizado para realizar diversas operações sobre o pool. Enquanto que na programação tradicional com memória volátil a memória é alocada a partir do *heap*, com PM os dados devem ser alocados no pool. Diferentemente do heap, os dados na PM são acessados a partir de um *objeto raiz*. Todo objeto armazenado em um pool persistente deve ser acessado a partir do objeto raiz. Todo pool tem exatamente um objeto raiz e ele sempre existe.

Uma vez o pool aberto, o objeto raiz pode ser recuperado através de uma chamada oferecida pelo PMDK, tipicamente a `pmemobj_root()`. Essa chamada retorna um ponteiro para o objeto raiz. É importante ressaltar que esse ponteiro aponta para uma área de memória persistente e portanto requer alguns cuidados especiais. Quando um pool é aberto, ele é mapeado para o espaço de endereçamento do processo. Geralmente não é possível prever em qual faixa desse espaço o pool será mapeado. Ou seja, usar um ponteiro convencional não é suficiente para endereçar um objeto persistente porque os endereços podem mudar de uma execução para outra. Usa-se o termo *ponteiro persistente* para diferenciar os ponteiros para PM dos ponteiros convencionais para memória volátil. No PMDK, um ponteiro persistente é representado pelo tipo `PMEMoid` de 128 bits, sendo composto por um identificador único de pool, de 64 bits, e um deslocamento, também de 64 bits. Para encontrar o endereço de um objeto no pool persistente é necessário somar o endereço virtual base, no qual o pool foi mapeado, com o deslocamento. Isso geralmente é feito por meio da chamada `pmemobj_direct()`.

Como um resumo, eis os passos necessários para acessar a PM: 1) criar o pool (se ainda não existir); 2) abrir o pool; 3) recuperar o objeto raiz; 4) efetuar operações na PM a partir deste objeto.

3.4.2. Atualização dos dados persistentes

O conceito principal utilizado pelo PMDK para proporcionar alteração do estado persistente é o de *transação*. Uma transação garante que todas as alterações realizadas dentro dela ou são efetivadas (*committed*) e consideradas persistentes, ou então abortadas (*aborted*) e revertidas (*rollback*) – nesse caso é como se nada tivesse acontecido. Essa propriedade é geralmente conhecida como *tudo ou nada* (*all or nothing*) e é a principal razão de se usar transações no contexto de PM. Desta forma, se uma queda de energia ou falha do sistema acontece enquanto uma transação está sendo executada, todas as alterações realizadas até aquele ponto serão revertidas. As transações fornecidas pelo PMDK podem ser aninhadas, mas é importante ressaltar que não garantem atomicidade entre threads,

ou seja, as modificações realizadas por uma transação de uma determinada thread são visíveis para todas as outras. Portanto é necessário a utilização de um mecanismo de sincronização específico no caso de aplicações multithreading. Por simplicidade, este texto não discute o suporte da PMDK para programação multithreading.

Toda operação efetuada sobre a PM que possa causar inconsistências deve ser colocada dentro de uma transação. No PMDK, uma transação pode ser especificada através do conjunto de macros `TX_BEGIN` e `TX_END`. Ao começar a transação é necessário especificar o pool de memória sobre o qual ela atuará. A maneira como as transações do PMDK garantem atomicidade a falhas é através do *versionamento de dados*, similar ao que acontece em sistemas de Banco de Dados. A técnica mais comum é conhecida como *Write-Ahead Logging*, ou simplesmente WAL [25]. Por exemplo, até recentemente o PMDK utilizava o *undo-logging* [6], no qual uma cópia do dado é salvo em um log interno antes do valor original ser alterado na PM. Apesar do programador não precisar conhecer exatamente como o esquema de logging é implementado, ele precisa informar para o PMDK, por meio de chamadas específicas da biblioteca (e.g., `TX_ADD`), quais os blocos de memória persistente serão alterados. Discutimos com mais detalhes como isso é feito ao implementar uma estrutura de dados persistente na Seção 3.4.4.

3.4.3. Alocação de memória

A interface para alocação de memória persistente também se difere da interface convencional para memória volátil. Para entender o porquê, considere que exista um alocador de memória persistente e que ela pudesse ser alocada através da seguinte chamada:

```
persistent_ptr = persistent_malloc(100);
```

Agora pense nos problemas que podem acontecer durante a execução desse trecho de código. Em particular, o que acontece se a memória é reservada internamente pelo alocador mas uma falha qualquer (e.g., queda de energia) acontece antes da atribuição para o ponteiro persistente? Qual é o estado do sistema depois que a falha for tratada (e.g., o sistema for religado)? Do ponto de vista do alocador, a alocação aconteceu. Porém, do ponto de vista da aplicação, o ponteiro ainda é inválido. O resultado da falha é um vazamento persistente de memória!

A forma mais comum e recomendada para resolver essa inconsistência é realizar a alocação de memória persistente dentro de uma transação. Pelo fato da chamada estar sendo realizada dentro de uma transação, uma falha antes da atribuição faz com que todo o trecho de código (inclusive a memória persistente alocada) seja revertido. Ou seja, o vazamento de memória não pode mais acontecer. O PMDK fornece ainda outras duas formas para alocação de memória persistente: alocação atômica e a interface *reserve/-publish*. Pelo fato da usabilidade dessas interfaces ser restrita a casos específicos, este texto se concentra no uso de alocação através de transações. O leitor pode encontrar mais detalhes sobre esses métodos alternativos no livro da Intel [30].

[6] Outra forma comum é o *redo-logging*, no qual as alterações são salvas em um log interno e aplicadas durante a fase de efetivação da transação.

```
1 struct Node {
2   int data;
3   struct Node *next;
4 };
5
6 typedef struct Node NODE;
7
8
```

(a) Convencional

```
struct Node {
  int data;
  PMEMoid p_next;
};

struct my_root {
  PMEMoid p_head;
};
```

(b) PMDK

Figura 3.10: Definição da estrutura para lista ligada simples. A versão com o PMDK (direita) utiliza ponteiros persistentes (`PMEMoid`) e declara explicitamente o objeto raiz.

3.4.4. Uma lista ligada persistente

Esta seção discute como os principais conceitos do PMDK descritos anteriormente podem ser usados para implementar uma estrutura de dados do tipo lista ligada persistente. Para fins de comparação, uma versão convencional (memória volátil) dessa estrutura de dados também é usada[7]. A Figura 3.10a mostra a forma convencional, enquanto a Figura 3.10b apresenta uma versão utilizando o PMDK. Há duas diferenças principais. Primeiro, note que os ponteiros precisam usar o tipo `PMEMoid`, já que agora a estrutura residirá na PM. Em segundo lugar, é comum declarar uma estrutura para representar o objeto raiz, no caso do exemplo a estrutura `my_root` (linhas 6–8). Neste exemplo, o objeto raiz é apenas um ponteiro persistente para o primeiro elemento da lista.

Há diversos procedimentos que operam sobre a lista ligada, entre eles: inserção, remoção, busca e impressão. Para começar a operar com a estrutura, a versão convencional provavelmente vai declarar um ponteiro (volátil) que será passado para os diferentes procedimentos. A versão persistente necessita primeiramente abrir o pool e recuperar o objeto raiz. A Figura 3.11 mostra um pool já existente (`exemplo.pool`) sendo aberto (linha 2)[8]. O pool é simplesmente um arquivo localizado na PM. Para que a aplicação possa confirmar se o conteúdo do pool é o esperado, geralmente um *layout* (basicamente, uma string) pode ser associado ao pool. Desta forma, no momento da abertura do pool, a aplicação pode exigir que o pool sendo aberto tenha um layout específico. No exemplo, o layout `pm-linkedlist` é usado. Assim, se o pool possuir um outro layout, a rotina vai falhar. Quando o pool persistente é aberto, um ponteiro para o objeto do tipo `PMEMobjpool` é retornado. Este ponteiro deve ser usado em outras chamadas do PMDK que lidam com o pool.

Uma vez o pool aberto, é possível recuperar o ponteiro para o objeto raiz (linha 5). Lembre-se que todo pool tem um único objeto raiz e ele é sempre válido. Caso seja a primeira vez que o objeto esteja sendo recuperado, a chamada `pmemobj_root` cria um novo objeto (vazio, por padrão), cujo tamanho é dado pelo segundo parâmetro. Para acessar o conteúdo do objeto raiz (no caso, um ponteiro para a lista ligada persistente) é

[7] O código fonte deste exemplo está disponível em `https://github.com/baldas/minicurso-memoria-persistente`

[8] O código para checagem de erro é omitido para simplificar a exposição do tema.

```
1  /* Abre o pool persistente e retorna um identificador */
2  PMEMobjpool *pop = pmemobj_open("exemplo.pool", "pm-linkedlist");
3
4  /* Recupera um ponteiro persistente para o objeto raiz */
5  PMEMoid p_root = pmemobj_root(pop, sizeof(struct my_root));
6
7  /* Transforma em um ponteiro convencional para acesso ao objeto raiz */
8  struct my_root *root = pmemobj_direct(p_root);
9
10 /* Imprime o conteudo da lista */
11 display(root->p_head);
```

Figura 3.11: Trecho de código necessário na versão persistente: abertura do pool de memória e acesso ao objeto raiz.

necessário primeiro transformar o ponteiro persistente (`PMEMoid`) para um ponteiro convencional. Isto pode ser feito adicionando ao endereço base do pool persistente (endereço no qual o pool for mapeado no espaço de endereçamento do processo) o deslocamento do objeto raiz. A chamada `pmemobj_direct` desempenha essa função (linha 8). Em geral, essa transformação é necessária toda vez que o conteúdo de um objeto apontado por um ponteiro persistente (PMEMoid) precisar ser acessado. Uma dica para evitar a confusão entre ponteiros convencionais e persistentes é sempre nomear as variáveis do tipo `PMEMoid` com o prefixo "`p_`". Finalmente, um exemplo com a chamada que imprime o conteúdo da lista ligada persistente pode ser visto na linha 11.

Para compreender melhor a diferença da versão volátil da lista ligada para a persistente, considere a Figura 3.12. O lado esquerdo mostra o espaço de endereçamento (endereços de memória aumentam de baixo para cima) de um processo que usa apenas memória volátil. A alocação dinâmica usa memória do heap (área verde). No exemplo, o ponteiro para o começo da lista é armazenado na seção de dados. Agora compare com a versão persistente (lado direito). Neste caso, a memória persistente também foi mapeada para o espaço de endereçamento do processo (área azul). É possível, portanto, acessar tanto a memória DRAM quanto PM. Os ponteiros persistentes (`PMEMoid`) estão representados graficamente por hexágonos. A figura assume que um pool persistente (azul mais escuro) já foi aberto. O ponteiro `root` (alocado em DRAM) contém o endereço de 128 bits (`PMEMoid`) para o objeto raiz. É importante que os ponteiros persistentes sempre apontem para algo dentro do pool.

Um fator complicador para quem está migrando do modelo convencional para o persistente é trabalhar de forma consistente com os ponteiros persistentes. Considere a implementação do procedimento para impressão da lista apresentado na Figura 3.13. A Figura 3.13a mostra uma implementação típica para a versão volátil da estrutura de dados. Comparando com a versão persistente, da Figura 3.13b, é possível notar que o ponteiro convencional foi substituído pelo persistente (`PMEMoid`). Qualquer operação não trivial sobre esse ponteiro requer algum tipo de chamada para o PMDK. Por exemplo, para checar se o ponteiro é nulo, geralmente a macro `OID_IS_NULL` é usada (linha 3). Como já dito, sempre que o conteúdo do objeto apontado por um ponteiro persistente deve ser acessado, primeiro é necessário convertê-lo para a versão convencional através

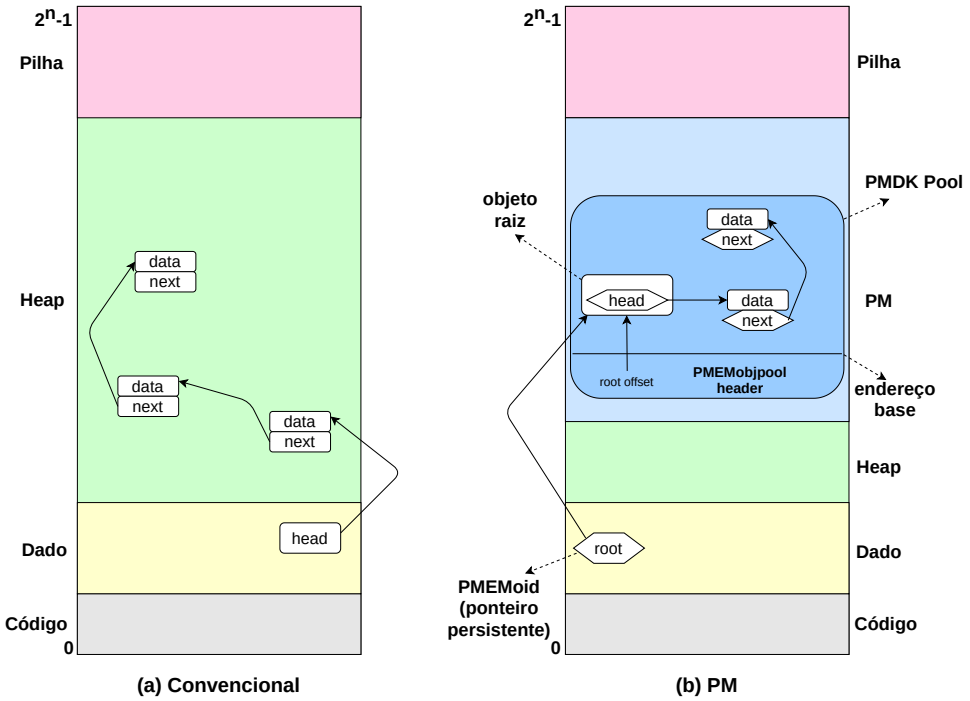


Figura 3.12: Uma visão do espaço de endereçamento com a lista ligada volátil (esquerda) e persistente (direita). A PM é mapeada no espaço de endereçamento do processo e pode ser acessada assim que um pool (`PMEMobjpool`) é aberto.

da chamada `pmemobj_direct` (linhas 4 e 5).

As operações realizadas diretamente sobre os objetos `PMEMoid` podem ser confusas e suscetíveis a erros. Por exemplo, a chamada `pmemobj_direct` retorna ponteiros sem tipo (void *). Portanto, é necessário realizar uma coerção para o tipo adequado. Os desenvolvedores do PMDK logo notaram que isso poderia ser um problema e implementaram uma abordagem baseada em macros para facilitar o manuseio dos ponteiros persistentes. Esta estratégia provê checagem estática de tipo, ou seja, o compilador pode acusar erros se um ponteiro para um objeto de um tipo for atribuído para um ponteiro de um tipo diferente.

Uma explicação completa sobre o suporte para checagem de tipos do PMDK está fora do escopo deste texto. No entanto, será apresentado o básico para a implementação da lista ligada persistente. A primeira etapa consiste da declaração do layout do pool, como mostrado na Figura 3.14. O layout é declarado nas linhas 1 a 4. A primeira linha marca o início da declaração e, a quarta linha, seu final. É necessário informar um nome para o layout (no caso, `linkedlist`). A declaração é composta do tipo para o objeto raiz (linha 2) e de todas as estruturas auxiliares; no caso do exemplo, apenas o tipo do nodo da lista ligada (linha 3). As linhas 6–13 mostram a declaração das estruturas, similar àquelas já apresentadas na Figura 3.10b. Porém, note que agora o tipo `PMEMoid` é substituído pela macro `TOID`. O parâmetro da macro é o tipo do ponteiro persistente.

A Figura 3.15 mostra como o procedimento para impressão da lista persistente

```
1 void display(NODE *head) {
2   NODE *current = head;
3   while (current != NULL) {
4     printf("␣%d␣", current->data);
5     current = current->next;
6   }
7 }
```

(a) Convencional

```
1 void display(PMEMoid p_head) {
2   PMEMoid p_current = p_head;
3   while (!OID_IS_NULL(p_current)) {
4     printf("␣%d␣", ((NODE *)pmemobj_direct(p_current))->data);
5     p_current = ((NODE *)pmemobj_direct(p_current))->p_next;
6   }
7 }
```

(b) PMDK

Figura 3.13: Implementação do procedimento para impressão da lista ligada.

pode ser codificada usando as macros. Compare essa versão com a apresentada anteriormente na Figura 3.13b. Note que, novamente, o tipo `PMEMoid` é substituído pela macro `TOID` (linhas 1 e 2). Além disso, ao invés de usar a chamada `pmemobj_direct`, é possível usar a macro `D_RO` para acessar o ponteiro direto para o objeto persistente (linhas 4 e 5). Quando o conteúdo do objeto é somente lido, em geral a macro `D_RO` é usada. Para os casos onde o conteúdo é escrito, necessita-se usar a macro `D_RW`.

O procedimento para a criação de um novo nodo para a lista ligada persistente é apresentado na Figura 3.16. Como discutido previamente, a forma mais comum de alocar memória do pool persistente é através de uma transação. A transação é iniciada na linha 5 e termina na linha 10. O parâmetro `pop` passado para a transação é o identificador do pool aberto. Um novo objeto é alocado através de `TX_NEW`, na linha 6. As linhas 8

```
1  POBJ_LAYOUT_BEGIN(linkedlist);
2  POBJ_LAYOUT_ROOT(linkedlist, struct my_root);
3  POBJ_LAYOUT_TOID(linkedlist, struct Node);
4  POBJ_LAYOUT_END(linkedlist);
5
6  struct Node {
7    int data;
8    TOID(struct Node) p_next;
9  };
10
11 struct my_root {
12   TOID(struct Node) p_head;
13 };
```

Figura 3.14: Lista ligada usando o suporte para declaração de layout do PMDK.

```
1  void display(TOID(struct Node) head) {
2    TOID(struct Node) p_current = head;
3    while (!TOID_IS_NULL(p_current)) {
4      printf("␣%d␣", D_RO(p_current)->data);
5      p_current = D_RO(p_current)->p_next;
6    }
7  }
```

Figura 3.15: Procedimento para impressão usando o suporte de macros do PMDK.

e 9 inicializam os campos do novo nodo (dado e ponteiro). Note que aqui é necessário utilizar a macro `D_RW` ao invés da `D_RO`, já que os campos estão sendo alterados. Se a transação for abortada por alguma falha em qualquer ponto antes de ser finalizada, todas as operações realizadas até aquele ponto serão descartadas.

Finalmente, a operação para inserir um novo elemento na cabeça da lista persistente é apresentada na Figura 3.17. A primeira grande diferença para uma versão volátil do mesmo procedimento é que é necessário o uso de uma transação (linhas 3–9). Isto ocorre porque as operações de alocação do nodo (linha 4, código da Figura 3.16) e atualização dos ponteiros (linhas 5 e 8) devem ser atômicas, já que uma falha em qualquer desses pontos pode deixar a estrutura de dados em um estado inconsistente. Note também que o código que cria o novo nodo é executado dentro de uma transação (Figura 3.16). Nesse caso, o PMDK faz um "achatamento" (*flattening*) das transações, ou seja, é como se existisse apenas uma. Uma outra diferença para a versão volátil é que, como discutido anteriormente, é necessário explicitamente versionar todos os dados que são alterados dentro da transação para que eles possam ser salvos no undo-log e a transação consiga se recuperar corretamente em caso de falhas. Isso é feito através da chamada da linha 7. Note que o único dado que precisa ser versionado é a cabeça da lista, já que o nodo em si (alterado na linha 5) já é versionando automaticamente pelo alocador de memória.

```
1  TOID(struct Node) createNewNode(PMEMobjpool *pop, int data)
2  {
3    TOID(struct Node) p_newNode;
4
5    TX_BEGIN(pop) {
6      p_newNode = TX_NEW(struct Node);
7
8      D_RW(p_newNode)->data = data;
9      D_RW(p_newNode)->p_next = TOID_NULL(struct Node);
10   } TX_END
11
12   return p_newNode;
13 }
```

Figura 3.16: Procedimento para criação de um novo nodo usando o pool persistente.

```
1  void insertAtHead(PMEMobjpool *pop, TOID(struct Node) *head, int data)
2  {
3    TX_BEGIN(pop) {
4      TOID(struct Node) p_newNode = createNewNode(pop, data);
5      D_RW(p_newNode)->p_next = *head;
6
7      TX_ADD_DIRECT(head);
8      *head = p_newNode;
9    } TX_END
10 }
```

Figura 3.17: Procedimento para inserção de novo nodo na lista persistente.

3.5. Conclusão

Neste minicurso apresentamos os principais conceitos relacionados à memória persistente, suas tecnologias e seu uso. Para exemplificar o uso de PM, utilizamos a biblioteca Intel PMDK e mostramos como uma estrutura de dados simples, como uma lista ligada persistente, pode ser implementada para fazer uso das vantagens da PM em relação à memória de trabalho tradicional. Já neste exemplo simples é fácil de perceber que ainda há um longo caminho de pesquisa a ser percorrido para que o uso de PM seja tão simples, direto e menos propenso a falhas. Em particular, eventuais erros na programação utilizando PM frequentemente só se manifestam quando há efetivamente uma falha no sistema computacional em questão, o que dificulta a localização e a eventual correção de erros de implementação. Neste minicurso nosso propósito também foi mostrar que dispositivos com memória persistentes já são uma realidade e que a programação de tais dispositivos vem ganhando bastante interesse tanto por parte da academia quanto pela iniciativa privada.

Agradecimentos

Este minicurso teve apoio dos processos nº 2019/26702-8 e 2018/15519-5 da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP).

Referências

[Åkerman 2005] Åkerman, J. (2005). Toward a universal memory. *Science*, 308(5721):508–510.

[Baldassin et al. 2021] Baldassin, A., Barreto, J. a., Castro, D., and Romano, P. (2021). Persistent memory: A survey of programming support and implementations. *ACM Comput. Surv.*, 54(7).

[Boukhobza et al. 2018] Boukhobza, J., Rubini, S., Chen, R., and Shao, Z. (2018). Emerging NVM: A Survey on Architectural Integration and Research Challenges. *ACM Trans. Des. Autom. Electron. Syst.*, 23(2):1–32.

[Burr et al. 2008] Burr, G., Kurdi, B., Scott, J., Lam, C., Gopalakrishnan, K., and Shenoy, R. (2008). Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development*, 52(4.5):449–464.

[Chen and Yen 2009] Chen, E. and Yen, T. (2009). Comparing SLC and MLC Flash Technologies and Structure.

[Doh et al. 2010] Doh, I. H., Kim, Y. J., Park, J. S., Kim, E., Choi, J., Lee, D., and Noh, S. H. (2010). Towards Greener Data Centers With Storage Class Memory: Minimizing Idle Power Waste Through Coarse-Grain Management In Fine-Grain Scale. In *Conference On Computing Frontiers*.

[Eshraghian 2010] Eshraghian, K. (2010). Evolution of Nonvolatile Resistive Switching Memory Technologies: The Related Influence on Hetrogeneous Nanoarchitectures. In *Transactions on Electrical and Electronic Materials*.

[Firth 2014] Firth, S. (2014). The machine – HP labs launches a bold new research initiative to transform the future of computing.

[Guo et al. 2010] Guo, X., Ipek, E., and Soyata, T. (2010). Resistive Computation: Avoiding the Power Wall with Low-Leakage, STT-MRAM Based Computing. In *International Symposium on Computer Architecture (ISCA)*.

[HP Labs 2014] HP Labs (2014). The Machine: A New Kind of Computer.

[Intel 2018] Intel (2018). Intel optane technology, press kit.

[Intel 2021] Intel (2021). eadr: New opportunities for persistent memory applications.

[International Technology Roadmap for Semiconductors (ITRS) 2007] International Technology Roadmap for Semiconductors (ITRS) (2007). Emerging Research Devices.

[International Technology Roadmap for Semiconductors (ITRS) 2009] International Technology Roadmap for Semiconductors (ITRS) (2009). Emerging Research Devices.

[International Technology Roadmap for Semiconductors (ITRS) 2010] International Technology Roadmap for Semiconductors (ITRS) (2010). 2010 Update Overview.

[Jeongdong Choe 2017] Jeongdong Choe, T. I. (2017). Intel 3d xpoint memory die removed from intel optane™ pcm (phase change memory).

[Kim et al. 2007] Kim, E.-k., Shin, H., Jeon, B.-g., Han, S., Jung, J., and Won, Y. (2007). FRASH: hierarchical file system for FRAM and flash. In *International Conference on Computational Science and Its Applications (ICCSA)*.

[Lee et al. 2009] Lee, B. C., Ipek, E., Mutlu, O., and Burger, D. (2009). Architecting Phase Change Memory as a Scalable DRAM Architecture. In *International Symposium on Computer Architecture (ISCA)*.

[Lim et al. 2009] Lim, K., Chang, J., Mudge, T., Ranganathan, P., Reinhardt, S. K., and Wenisch, T. F. (2009). Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 267–278, New York, NY, USA. ACM.

[Masuoka et al. 1984] Masuoka, F., Asano, M., Iwahashi, H., Komuro, T., and Tanaka, S. (1984). A new flash e2prom cell using triple polysilicon technology. In *Electron Devices Meeting, 1984 International*, volume 30, pages 464–467.

[McGrath 2008] McGrath, D. (2008). 'universal memory' race still on the starting block. EE Times.

[Micron 2006] Micron (2006). 1Gb: x4, x8, x16 DDR3 SDRAM Features.

[Micron 2006] Micron (2006). TN-29-19: NAND Flash 101 NAND vs. NOR Comparison.

[Mishra et al. 2011] Mishra, A. K., Dong, X., Sun, G., Xie, Y., Vijaykrishnan, N., and Das, C. R. (2011). Architecting on-chip interconnects for stacked 3D STT-RAM caches in CMPs. In *International Symposium on Computer Architecture (ISCA)*.

[Mohan et al. 1992] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. (1992). ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.*, 17(1):94–162.

[Peng et al. 2020] Peng, I., Wu, K., Ren, J., Li, D., and Gokhale, M. (2020). Demystifying the Performance of HPC Scientific Applications on NVM-based Memory Systems. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 916–925.

[Qureshi et al. 2009] Qureshi, M. K., Srinivasan, V., and Rivers, J. A. (2009). Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *International Symposium on Computer Architecture (ISCA)*.

[Ramos and Bianchini 2012] Ramos, L. and Bianchini, R. (2012). Exploiting phase-change memory in cooperative caches. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 227–234.

[Ramos et al. 2011] Ramos, L. E., Gorbatov, E., and Bianchini, R. (2011). Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 85–95, New York, NY, USA. ACM.

[Scargall 2020] Scargall, S. (2020). *Programming Persistent Memory - A Comprehensive Guide for Developers*. Apress, 1st edition.

[Solihin 2019] Solihin, Y. (2019). Persistent Memory: Abstractions, Abstractions, and Abstractions. *IEEE Micro*, 39(1):65–66.

[Toshiba 2006] Toshiba (2006). NAND vs. NOR Flash memory.

[Tyson 2019] Tyson, M. (2019). Intel Optane DC Persistent Memory launched. Retrieved from https://hexus.net/tech/news/storage/129143-intel-optane-dc-persistent-memory-launched/.

[Vance 2014] Vance, A. (2014). With 'the machine,' HP may have invented a new kind of computer. *BusinessWeek: technology*.

[Wu et al. 2009] Wu, X., Li, J., Zhang, L., Speight, E., Rajamony, R., and Xie, Y. (2009). Hybrid Cache Architecture with Disparate Memory Technologies. In *International Symposium on Computer Architecture (ISCA)*.

[Zhang and Li 2009] Zhang, W. and Li, T. (2009). Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[Zhou et al. 2009] Zhou, P., Zhao, B., Yang, J., and Zhang, Y. (2009). A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *International Symposium on Computer Architecture (ISCA)*.

Capítulo
4

Coisas para Saber Antes de Fazer o Seu Próprio *Benchmarks Game* [1]

Alfredo Goldman, **Elisa Uhura**
Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, Brasil

Sarita Mazzini Bruschi
Instituto De Ciências Matemáticas e de Computação, Universidade de São Paulo, São Paulo, Brasil

Resumo

Esse texto apresenta diversas técnicas para medir a analisar o desempenho de programas usadas no minicurso da ERAD-RS de 2022. O nosso objetivo é que ao final do texto os alunos possam analisar o desempenho dos seus programas e também criar os seus próprios benchmarks.

4.1. Introdução

Uma das formas de se melhorar o desempenho de aplicações é por meio da paralelização que é muito usada em computação de alto desempenho (do inglês, *HPC - High Performance Computing*). Mas, antes de se pensar em como fazer com que partes de um programa possam ser executadas de forma concorrente é essencial saber avaliar o seu desempenho. Nesse texto, o nosso objetivo é exatamente esse, entender melhor como avaliar e comparar diferentes variações de um mesmo pseudo-código.

Inspirado no *Computer Language Benchmarks Game* [2], este minicurso revisa conceitos essenciais para a avaliação e comparação de programas, incluindo exemplos e demonstrações ao vivo de como aplicá-los. Dividido em duas partes, a primeira parte aborda os aspectos gerais enquanto a segunda foca na análise individual de um programa.

[1]Esse minicurso teve apoio do processo nº 2019/26702-8, Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP)

[2]`https://benchmarksgame-team.pages.debian.net/benchmarksgame/`

Na primeira parte utilizaremos um conjunto de programas que implementam um mesmo algoritmo (pseudo-código) em linguagens diferentes. Em seguida, será construído um `Makefile` para comparar o desempenho dos programas. Vamos:

- apresentar algumas métricas para avaliação de um programa (tempo, espaço, impacto energético, memória, ...);
- detalhar do que é composto o tempo de execução (tempo real, em espaço de usuário e espaço do SO);
- mostrar formas de tabular os dados coletados, meios de avaliar, interpretar e visualizar os as medições;
- e por fim discutir fatores básicos que afetam o desempenho com base nas medições (linguagem de programação, sistema operacional, hardware e estado de carga do sistema).

Na segunda parte vamos trabalhar nos programas de forma individual para explicar tópicos de análise de desempenho usando o `Makefile` desenvolvido na parte anterior. Vamos abordar:

- os diferentes tipos de relógios que um sistema pode oferecer (incluindo relógios disponibilizados pela arquitetura) usando C;
- o impacto de *flags* no tempo de execução e compilação usando o GCC;
- uso de compiladores/interpretadores alternativos e considerações sobre compilação JIT usando o PyPy;
- Profiling usando a versão em Go;
- verificação de saída e soma de verificação usando Go.

Os códigos apresentados no texto estão disponíveis em `https://github.com/elisauhura/ERAD-benchmarks`.

4.2. Conceitos importantes para avaliar o desempenho

Esta seção apresenta alguns conceitos importantes para que seja possível avaliar o desempenho de programas.

Primeiramente, um ponto importante é definir o que é desempenho. Em termos computacionais, o desempenho pode ser considerado como a quantidade de trabalho útil realizado por um computador. O desempenho deve ser mensurável e para isso, algumas métricas são utilizadas.

O desempenho pode também ser absoluto ou relativo. Em termos absolutos, temos como exemplo as métricas: tempo de execução, latência, vazão (*throughput*), consumo energético, operações por segundo. Outras métricas podem ser computadas por meio de operações aritméticas considerando várias métricas absolutas. Nesse caso, essas métricas

fazem mais sentido quando são utilizadas para comparação entre sistemas, pois isoladamente elas não possuem significado. Por exemplo, podemos combinar métricas de tempo de execução e de gasto de energia simultaneamente, isso é feito em uma famosa lista de HPC verde [3].

Quando se deseja comparar o desempenho, é necessário que, de todos os fatores que possam influenciar no desempenho, somente um ou poucos fatores sofram variação. Com isso, é possível avaliar a influência da alteração daquele fator (através de simples comparação dos valores obtidos) ou dos poucos fatores que foram alterados (utlizando técnicas de análise de variância ou outras técnicas estatísticas).

Em computação, um *bechmark* é um programa (ou um conjunto de programas), definido especificamente com o objetivo de avaliar o desempenho, podendo ser executado em diferentes configurações. As métricas coletadas nas execuções podem então ser comparadas, sendo possível afirmar qual configuração tem o melhor desempenho sob algum aspecto.

Vários são os fatores que influenciam o desempenho dos programas, incluindo fatores em nível de hardware (por exemplo, hierarquia de memória e níveis de cache, velocidade e conjunto de instruções do processador) e software (por exemplo, estrutura do código, compilador/interpretador e sistema operacional).

Como a análise de desempenho está intimamente ligada a medidas, é essencial também se levar em conta a precisão. Variações em medições de tempo de execução são usuais em computação, e existem várias formas de se procurar encontrar relevância estatística. A regra mais comum, que infelizmente não é usada de forma adequada é a de se repetir a medição 30 vezes, e apresentar a média e o desvio padrão encontrado. Mas, em muitos casos isso não é suficiente, pois quando o desvio padrão é alto, a média pode deixar de ser significativa.

O tema de relevância estatística foge um pouco do escopo do texto, mas vamos mostrar um exemplo de como esse processo pode ser bem feito, ou seja em uma medição, vamos mostrar a distribuição dos valores. Com isso podemos verificar se a distribuição obedece um padrão (se espera uma distribuição em forma de sino), e nesse caso, podemos falar em média e desvio padrão com segurança. Para o leitor interessado sugerimos as leituras de [Jain 1991, Kalibera e Jones 2013].

Vejamos os tempos de execução do programa a seguir, ele aloca um vetor de tamanho SIZE, o preenche com números aleatórios e depois calcula a média dos números:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE  2000000

int media(){
    int v[SIZE], sum = 0;
    for (int i = 0; i < SIZE; i++)
        v[i] = rand() % 100; // Random number between 0 and 99
    for (int i = 0; i < SIZE; i++)
```

[3] https://www.top500.org/lists/green500/

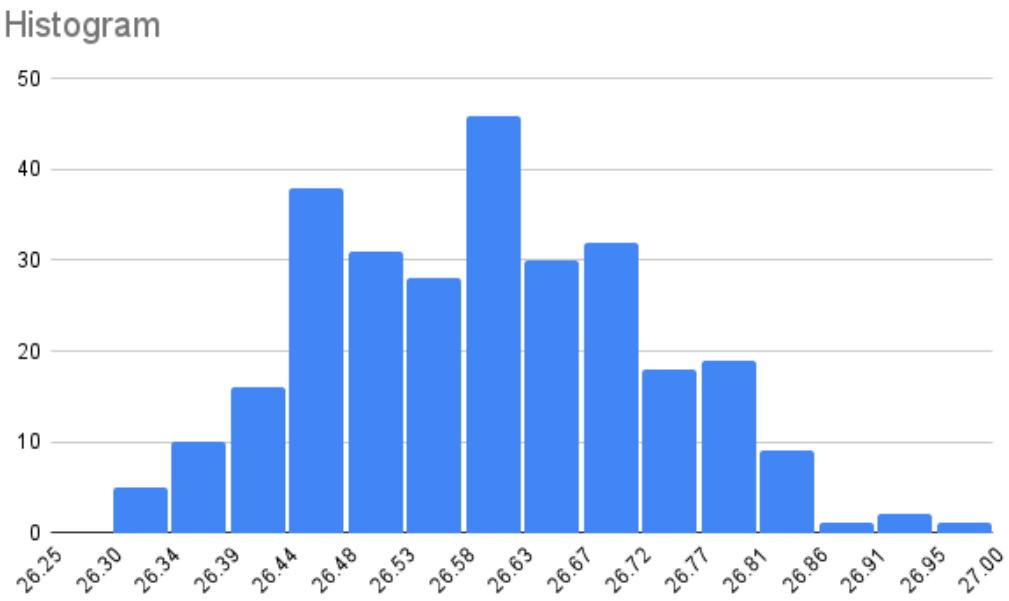


Figura 4.1. Histograma com os valores da execução em mili segundos

```
        sum = sum + v[i];
    return sum/SIZE;
}

void main(){
    clock_t start, end;
    double cpu_time_used;
    int val;
    for (int i = 0; i < 300; i++){
        start = clock();
        val = media();
        end = clock();
        cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
        printf("%g \n",  cpu_time_used);
    }
}
```

Ao fazer um histograma com os valores obtidos em um processador i7 de nona geração (tirando *outliers* os valores muito altos, sim, uma das execuções levou mais de 59 mili segundos), podemos observar na figura 4.1 que a média sim, faz sentido. Dois indícios são importantes nessa distribuição, a forma de sino e o desvio padrão pequeno. Nesse caso, sim podemos dizer que a média foi: 26.6 mili segundos com desvio padrão 0.3.

4.2.1. Ferramentas utilizadas para medir o desempenho

Esta seção apresenta alguns programas utilizados para medir o desempenho de programas e as métricas que podem ser obtidas a partir deles. Além de programas, também é possível

utilizar dispositivos físicos, como medidores de consumo de energia ou processadores auxiliares que obtém informação fora do dispositivo.

4.2.1.1. Time

O utilitário `time`[4], especificado na ISO/IEC 9945-2:1993, é utilizado para medir o a duração da execução de um programa qualquer em sistemas POSIX. Ele pode ser disponibilizado como um programa individual ou como um comando embutido em *shell*s e sua saída padrão é dependente da implementação, assim, o argumento `-p` pode ser utilizado para emitir o resultado em um formato padronizado.

São mostrados três valores, o **real** que corresponde ao tempo total, ou de relógio entre o início e o final do programa. O tempo **user** que é o tempo gasto pelas instruções do programa, e finalmente o **sys** que o tempo usado em chamadas de sistema (*kernel*) do programa. A vantagem de se usar o `time` é que ele não é invasivo, mas só mede o programa como um todo.

4.2.1.2. Perf

O `perf`[5] é um comando que utiliza o subsistema de contadores de desempenho do *kernel Linux* para gerar métricas detalhadas do sistema. O `perf` suporta uma quantidade de eventos que podem ser obtidos a partir de diferente fontes. Algumas métricas são do próprio kernel do linux (troca de contexto, *page-fault*) e outras são obtidas a partir dos contadores disponíveis na *Performance Monitoring Unit* - (PMU), tais como ciclos de processador utilizados e referências à memória cache.

Apesar de ser específico para sistema com o *kernel Linux*, outros sistemas possuem ferramentas com funcionalidades similares como o *Instruments*[6] para *MacOS*.

4.2.1.3. Valgrind

O *Valgrind*[7] é um arcabouço para análise de programas em tempo de execução. Apesar de ser conhecido como ferramenta para a detecção de vazamento de memória, ele também pode ser utilizado para a geração de árvores de execução e análise de uso do *cache* do processador. Devido a utilização de uma técnica de virtualização para a análise dos programas, ele aumenta consideravelmente a duração da execução do programa.

[4] `https://pubs.opengroup.org/onlinepubs/009604499/utilities/time.html`
[5] `https://perf.wiki.kernel.org/index.php/Main\_Page`
[6] `https://developer.apple.com/xcode/features/`
[7] `https://valgrind.org`

4.2.1.4. Vtune™ Profiler

O *Intel® Vtune™ Profiler*[8] é um ambiente integrado para análises de programas exclusivo para plataformas da *Intel*. Ele permite a análise de diversos tipos de programas, incluindo a análise de programas que rodam em *clusters MPI*, gerando um conjunto diverso de métricas, como contadores de desempenho de hardware, utilização do *cache* e consumo de energia.

4.2.2. Tabulação e análise dos dados

Esta seção apresenta informações sobre a tabulação dos dados coletados e algumas ferramentas para análise deles.

4.2.2.1. Resultados, parâmetros e ambiente

Além do registro dos resultados, é importante registrar informação sobre os parâmetros utilizados e o ambiente de execução. Os parâmetros compreendem as informações passadas que alteram o funcionamento do programa, como variáveis de ambiente que determinam número de *threads* a serem utilizadas ou os argumentos de execução. O ambiente de execução compreende dados sobre o *hardware* e o *software* em que o *benchmark* está sendo executado.

4.2.2.2. Tabulação

Para experimentos onde poucos dados são gerados, o registro manual dos dados é uma opção viável e pode ser feita utilizando uma planilha eletrônica. Para quantidades maiores de dados, o registro automático através de *script*s passam a ser um caminho mais vantajoso pela economia de tempo e menor chance de erros no processo de registro.

Ao automatizar o processo é necessário definir o formato de registro utilizado, como arquivos CSV ou bancos SQLite. Além da facilidade de registro, deve ser considerado o suporte de leitura do formato escolhido pelas ferramentas de análise. No caso de campos não estruturados, representações como JSON podem apresentar a flexibilidade necessária.

4.2.2.3. Análise

Após o registro dos dados, a análise pode ser realizada por uma vasta gama de programas. Além de planilhas eletrônicas, pode ser empregado o uso de programas como *Jupyter notebooks*[9] com bibliotecas como a *Matplotlib*[10], ou programas como o *Jamovi*[11].

[8] https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[9] https://jupyter.org

[10] https://matplotlib.org/

[11] https://www.jamovi.org

4.3. Parte 1

Nosso objetivo nesta primeira parte do minicurso é apresentar como obter algumas métricas e utilizá-las para comparação de um mesmo algoritmo implementado em várias linguagens.

Para isso, utilizaremos o algoritmo *binary-trees* como *benchmark*, o qual é uma adaptação do benchmark de Hans Boehm para analisar *Garbage Collection* (GC) [12]. Esse *benchmark* aloca milhões de *short-lived trees* e percorre-as.

As linguagens que iremos comparar são: C, Python e Go. Os três códigos estão disponíveis no *Benchmarks Game* e são as versões que possuem o melhor desempenho em suas respectivas linguagens.

4.3.1. Compilação e Execução

Arquivos `makefile` são utilizados para definir regras de modo que o processo de compilação seja mais automático. Neste minicurso serão definidos dois arquivos de tt makefile para a compilação dos programas em C e Go.

O código em C utilizará o compilador GCC[13], o código em Go utilizará o compilador Go[14] e o código em Pyhton utilizará o interpretador Python[15].

O `makefile` para o programa em C define o compilador, as flags de compilação, os diretórios para os arquivos de *header* e das bibliotecas, o caminho onde é encontrada a biblioteca que será linkada ao arquivo executável, e o nome dos arquivos fonte, objeto e executável:

```
1  # the compiler: gcc for C program
2  CC=gcc
3
4  # compiler flags:
5  CFLAGS=-pipe -Wall -O3 -fomit-frame-pointer -march=ivybridge -fopenmp
6
7  # directory containing hearder files
8  INCLUDES=-I/usr/local/apr/include/apr-1
9
10 # library paths:
11 LFLAGS=-L/usr/local/apr/lib -fopenmp
12
13 # library to link into executable
14 LIBS=-lapr-1
15
16 # C source file
17 SRCS=binary-trees.c
18
19 # C object file
20 # chance .c in SRSC by .o
21 OBJS=$(SRCS:.c=.o)
22
```

[12] https://hboehm.info/gc/gc_bench/
[13] https://gcc.gnu.org/
[14] https://go.dev/
[15] https://www.python.org/

```
# executable file
MAIN=binary-trees

.PHONY: depend clean

all:  $(MAIN)
  @echo File compiled

$(MAIN): $(OBJS)
  $(CC) $(OBJS) -o $(MAIN)  $(LFLAGS) $(LIBS)

$(OBJS): $(SRCS)
  $(CC) $(CFLAGS) $(INCLUDES) -c $(SRCS)

clean:
  $(RM) *.o *~ $(MAIN)
```

Nesse exemplo do *benchmark binary-trees*, é necessário instalar o *Apache Portable Runtime*[16], o qual irá fazer o gerenciamento de alocação de memória do programa.

Para compilação do código em Go, basta ter instalado o compilador e executar o seguinte `Makefile`:

```
build:
  go build -o binary-trees.go_run binarytrees.go-2.go
```

No caso da linguagem Python, como se trata de uma linguagem interpretada, não é necessário compilar o código, sendo que o comando será executado diretamente na execução.

O esforço de deixar claros todos os parâmetros usados, como ambiente, compilador e *flags* é essencial para permitir a reprodutibilidade dos experimentos.

Para facilitar a repetição da execução dos programas, como descrito em 4.2, foi elaborado um *script shell* que repete a execução dos três programas que serão comparados, coletando o tempo de execução de cada repetição e armazenando um arquivo texto:

```
# ====================
# ||     CLEANUP     ||
# ====================

rm results/*.out

# ====================
# ||    EXECUTION    ||
# ====================

SOURCE="binary-trees"
PARAM="15"


for i in $(seq 1 30)
do
  (time C/$SOURCE $PARAM) 2>> results/C.out
```

[16] `https://apr.apache.org/`

```
  (time python3 -OO Python/$SOURCE.py $PARAM) 2>> results/Python.out
  (time Go/$SOURCE.go_run $PARAM) 2>> results/Go.out
done
```

4.3.2. Métricas

Em termos de métricas a serem analisadas, as ferramentas descritas em 4.2.1 podem fornecer muitas métricas, tais como: tempo de execução, quantidade de troca de contexto, *page-fault*, quantidade de ciclos de máquina utilizados, quantidade de instruções, *branchs* (desvios) e *branch-misses* (previsão errada no desvio). A saída abaixo é resultado da execução do `Perf` com o seguinte comando:

```
sudo perf stat -o ../results/perf.out ./binary-trees 15
      ../results/saida.out
```

```
 Performance counter stats for './binary-trees 15':

            268,19 msec task-clock       #    0,960 CPUs utilized
                30      context-switches #    0,112 K/sec
                 0      cpu-migrations   #    0,000 K/sec
               731      page-faults      #    0,003 M/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

       0,279237198 seconds time elapsed

       0,264584000 seconds user
       0,003949000 seconds sys
```

No resultado descrito acima, a informação `<not supported>` é devido ao teste ter sido executado em uma máquina virtual com o sistema operacional Linux.

Para essa parte do minicurso, utilizaremos a ferramenta `time`, como já observado na Seção 4.3.1 e é com base nesses valores que faremos a comparação das linguagens.

4.3.3. Tabulando os dados

Para a tabulação dos dados, foi elaborado um programa em Python, que lê os arquivos de saída gerados (C.out, Python.out e Go.out), trata os valores e insere-os em um *dataframe*, salva o *dataframe* em um arquivo .csv e gera os gráficos para cada linguagem e cada métrica analisada (tempos real, user e sys).

```
import os, math
import subprocess
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

NUM_EXPERIMENTS = 30


def proccess_file(df, lang):
    data  = { 'language':lang }
```

```
12
13      f     = open('results/{}.out'.format(lang))
14
15      line = f.readline()
16      line = f.readline()
17
18      for i in range (NUM_EXPERIMENTS):
19        if 'real' in line:
20          # Trata a linha com a informacao do tempo real
21          line = line.strip().split()
22          time = line[1].split("m")
23          seg = time[1].split("s")
24          seg = seg[0].replace(',', '.')
25          time2 = float(time[0])*60 + float(seg)
26          data['real'] = time2
27          # Trata a linha com a informacao do tempo user
28          line = f.readline()
29          line = line.strip().split()
30          time = line[1].split("m")
31          seg = time[1].split("s")
32          seg = seg[0].replace(',', '.')
33          time2 = float(time[0])*60 + float(seg)
34          data['user'] = time2
35          # Trata a linha com a informacao do tempo sys
36          line = f.readline()
37          line = line.strip().split()
38          time = line[1].split("m")
39          seg = time[1].split("s")
40          seg = seg[0].replace(',', '.')
41          time2 = float(time[0])*60 + float(seg)
42          data['sys'] = time2
43          line = f.readline()
44          line = f.readline()
45          df = df.append(data, ignore_index=True)
46
47      f.close()
48
49      return df
50
51  def proccess_data():
52      languages = ['C', 'Python', 'Go']
53
54      # Criando o DataFrame a ser populado
55      column_names = ['real', 'user', 'sys']
56      df = pd.DataFrame(columns = column_names)
57
58      # Insere as informacoes de cada arquivo de resultado
59      for lang in languages:
60        df = proccess_file(df,lang)
61
62      return df
63
64
65  def processs_results(df):
66
```

```
    languages = ['C', 'Python', 'Go']
    metrics = ['real', 'user', 'sys']

    # Gerando Box plot dos graficos de maneira simples e calculando o intervalo de confianca
    for lang in languages:
        if lang == 'C':
          str_option = 'C'
        elif lang == 'Python':
          str_option = 'Python'
        elif lang == 'Go':
          str_option = 'Go'

        print("Gerando Imagens para Linguagem {}".format(str_option))

        temp_df = df[df['language'] == lang]
        print(temp_df)
        for c in metrics:
            # Calculando o intervalo de confianca a 95% com t-student
            conf_interval = (2.5096 * temp_df[c].std()) / math.sqrt(float(NUM_EXPERIMENTS))

            plt.title("Laguage {} {}=({:.2f} + {:.2f})".format(str_option, c, temp_df[c].mean(), conf_interval ))
            temp_df.boxplot(column=c)
            plt.savefig('figures/fig_{}_{}'.format(lang, c))
            plt.close('all')


if __name__ == '__main__':

    print("Processamento os arquivos de resutlados")
    df = proccess_data()
    print("Salvando resultados em disco (results/results.csv)...")
    df.to_csv('results/results.csv', index=False, header=True)
    print("Resultados salvos em um Dataframe {}.".format(df.shape))

    print("Processando a Analise dos dados")
    processs_results(df)
    print("Analise finalizada")
```

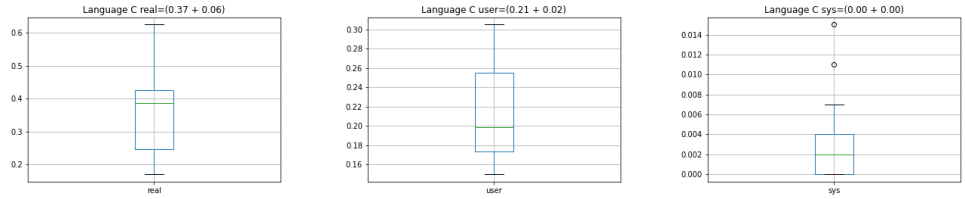
Listing 4.1. Código para ler os resultados e gerar os gráficos

4.3.4. Analisando os resultados

Os gráficos gerados pelo código da Listagem 4.1 podem ser observados nas Figuras 4.2, 4.4 e 4.3, as quais apresentam boxplots com 30 resultados para cada código para cada métrica. Além disso, são apresentados também os valores referentes à média e a metade do valor do intervalo de confiança com nível de confiança de 95%.

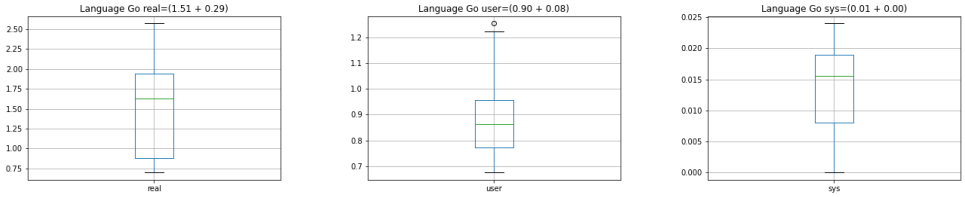
A linguagem que obteve o melhor desempenho (menor tempo de execução) é a linguagem C, seguida pela linguagem Go. Python obteve o pior desempenho. Esses resultados confirmam os resultados apresentados na página com os *benchmarks*.

É interessante também ressaltar que uma outra métrica que pode ser usada é o



(a) Tempo Real (b) Tempo User (c) Tempo Sys

Figura 4.2. Tempos para linguagem C



(a) Tempo Real (b) Tempo User (c) Tempo Sys

Figura 4.3. Tempos para linguagem Go

valor mínimo, ou seja entre todas a medidas a de menor valor. Dessa forma temos que o tempo real em C é pouco menor do que 0.2 segundos, em Go, pouco menor do que 0.75 segundos. O menor tempo em Python foi de mais de 6 segundos.

4.4. Parte 2

Nesta parte são explorado tópicos de análise de desempenho de programas. Utilizaremos o cálculo do conjunto Mandelbrot[17] para realizar as análises. Primeiro será apresentado a definição geral do programa, com implementações disponíveis para as linguagens C[18], Python[19] e Go[20]. Posteriormente são abordados os diferentes tipos de relógios que um sistema pode oferecer utilizando a versão em C; impacto de *flags* no tempo de execução e compilação usando o GCC; uso de compiladores/interpretadores alternativos usando o PyPy e considerações sobre compilação JIT; Profiling usando a versão em Go; e verificação de saída e soma de verificação usando a versão em Go.

4.4.1. Conjunto Mandelbrot

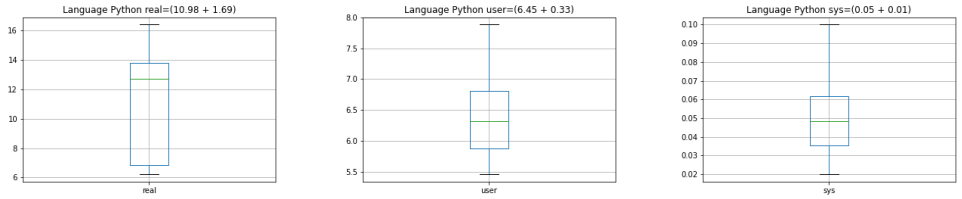
O conjunto Mandelbrot consiste nos números imaginários que não divergem para a função $f_c(z) = z^2 + c$ quando aplicada de forma recursiva: $f_c(f_c(...f_c(0)))$. Para fins computacionais, consideramos que um número imaginário diverge quando, em algum momento

[17] https://en.wikipedia.org/wiki/Mandelbrot_set

[18] https://github.com/elisauhura/ERAD-benchmarks/blob/main/Parte2/mandelbrot.c

[19] https://github.com/elisauhura/ERAD-benchmarks/blob/main/Parte2/mandelbrot.py

[20] https://github.com/elisauhura/ERAD-benchmarks/blob/main/Parte2/mandelbrot.go



(a) Tempo Real (b) Tempo User (c) Tempo Sys

Figura 4.4. Tempos para linguagem Python

da aplicação recursiva da função, a norma euclidiana do valor intermediário é maior que 2, e consideramos que um número imaginário converge quando, após um número pré-estabelecido de iterações, nenhum de seus valores intermediários apresenta norma euclidiana maior que 2. O programa gera uma imagem 2D onde o eixo das abscisas representa a parte real e o eixo das ordenadas a parte imaginária.

O programa recebe como argumentos de entrada a largura (w) e altura (h) da imagem a ser gerada e os valores em ponto flutuante x_r, x_i, y_r e y_i que serão utilizados para definir o conjunto de pontos analisados. Para cada pixel da imagem (x,y), sendo $(0,0)$ o ponto superior esquerdo e $(w-1,h-1)$ o ponto inferior esquerdo, é atribuído o número imaginário $(x/w*(y_r-x_r)+x_r)+(y/h*(x_i-y_i)+y_i)i$. Caso o número atribuído faça parte do conjunto Mandelbrot ele é pintado de branco, caso contrário, de preto. Por fim, o programa deve escrever na saída padrão a imagem utilizando o formato Netpbm P1[21].

4.4.2. Relógios do sistema

Sistemas operacionais modernos possuem diferentes fontes de tempo disponíveis para os processos. Essas fontes avançam com base em critérios diferentes e permitem obter diferentes métricas para análise.

Um modo portátil para ter acesso a um conjunto útil de fontes é por meio da função `clock_gettime`[22], disponível em sistemas POSIX.

Sistemas GNU/Linux costumam disponibilizar 3 fontes de interesse para análise de programas: CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID, CLOCK_THREAD_CPUTIME_ID. Elas representam, respectivamente, uma fonte de tempo que incrementa de forma consistente, uma fonte de tempo que incrementa de forma consistente sempre que o processo está em execução e uma fonte de tempo que incrementa de forma consistente sempre que a *thread* que invocou a função está em execução.

A versão em C do programa faz uso das três fontes de tempo diferentes e escreve na saída padrão de erro a duração de execução para cada *thread*, para o processo como um todo (proc) e o tempo decorrido no mundo real (mono). No caso da *threads*, também é escrito a quantidade de pontos dentre os processados por cada *thread* que convergiram.

A Tabela 4.1 apresenta um exemplo dos dados gerados para uma execução utilizando 4 *threads* em um computador 6-core comum para gerar a Figura 4.5. É possível ob-

[21] `https://en.wikipedia.org/wiki/Netpbm`
[22] `https://linux.die.net/man/3/clock_gettime`

Fonte	Segundos	Pontos Convergentes
thread 1	0.010794s	5406
thread 2	0.010774s	5525
thread 3	0.061591s	219960
thread 4	0.061792s	220777
mono	0.065554s	-
proc	0.148752s	-

Tabela 4.1. Dados de tempo e pontos convergentes para o programa Mandelbrot em C com os argumentos: largura = 1600; altura = 1600; xr = -1.5; xi = -1.5; yr = 1.5; yi = 1.5; máximo de iterações = 50; workers = 4

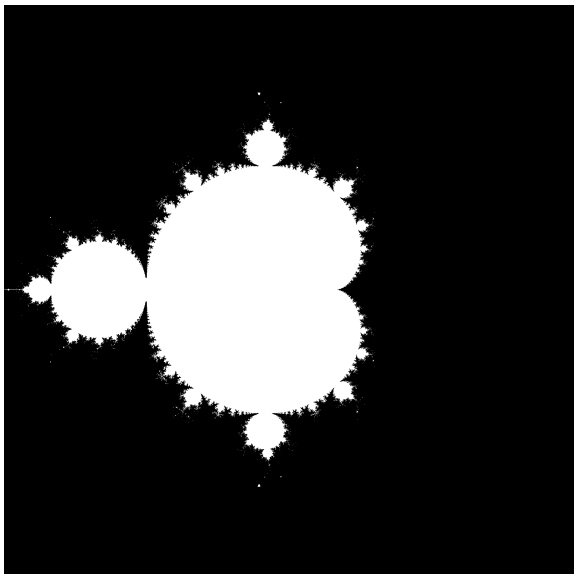


Figura 4.5. Exemplo de imagem gerada pelos programas Mandelbrot

servar que o tempo de execução do programa (mono) é dominado em boa parte por apenas uma das *thread*, enquanto as outras completam suas partes da imagem antes, indicando uma má distribuição da carga de trabalho. Como cada *thread* processa uma quantidade similar de pontos, a diferença de carga está na quantidade de iterações para determinar se um ponto diverge ou não.

4.4.3. Impacto de flags no *runtime* e *compile time*

Um forma de otimizar a execução de um programa é através da alteração das *flags* de otimização passadas ao compilador. Geralmente, ao ativar uma *flag* acontecendo uma troca, onde tenta se reduzir o tempo de execução de um programa por meio de um maior tempo de análise e transformações durante a compilação.

Todavia, a tentativa de reduzir o tempo de execução nem sempre é alcançada e por isso é importante analisar o impacto do uso de *flags* tanto no tempo de execução quanto no de compilação por ser possível que o tempo extra gasto na compilação não seja vantajoso

Otimização	Média (compile)	Média (runtime)
-O0	0.119s	0.124s
-O1	0.132s	0.068s
-O2	0,144s	0,079s
-O3	0,154s	0,276s

Tabela 4.2. Tempos de compilação e execução para a versão em C do programa

em determinadas partes do estudo.

A Tabela 4.2 apresenta o tempo de compilação e execução médio com 10 amostras para as flags -O0, -O1, -O2 e -O3 e com os mesmos argumentos de execução da Tabela 4.1 em um computador 6-core comum utilizando o compilador *Clang* (a variância foi omitida pois todas foram inferiores a 10^{-4}). É possível observar que o entre a *flag* -O1 e -O2, é observada uma piora no tempo de execução e mesmo sendo o primeiro nível de otimizações do clang, a -O1 apresentou o melhor tempo de execução.

É importante ressaltar que o espaço de busca por *flags* adequadas vai muito além das otimizações padrão, pois certas combinações de *flags* fora do padrão -O"podem ser benéficas, ou mesmo levar a erros na execução. Quando se pensa em desempenho uma busca pelas melhores *flags* não pode ser descartada, mas só lembrando o espaço de busca pode ser exponencial. Para o leitor interessado sugerirmos um artigo clássico [Hoos 2012] e um mais recente que mostra o potencial de uso de *flags* para GPUs [Bruel, Gonzalez e Goldman 2017].

4.4.4. Compiladores e interpretadores alternativos

Uma outra alternativa para otimizar o programa é através do uso de compiladores ou interpretadores alternativos. Eles podem explorar otimizações não implementadas no projeto principal ou utilizarem como alvo aceleradores disponíveis no sistema, como GPUs ou TPUs.

Um caso em que é comum observar uma grande diferença nos números observados é em linguagens interpretadas como no caso do Python. Utilizando a versão em Python do programa e os mesmos argumentos da Tabela 4.1 (exceto pela omissão dos workers, já que esta é uma versão sequencial), com o interpretador padrão da linguagem (Python 3.9.10), para 10 amostras, foi observado um tempo de execução médio de 7,421 segundos com variância de 0,088 e com o interpretador alternativo (Pypy 7.3.8), para 10 amostras, foi observado um tempo de execução médio de 0,562 segundos com variância de 0,0008. Esse salto acontece pelo uso da técnica de compilação *just-in-time* utilizada pelo Pypy.

4.4.5. Considerações sobre interpretadores *just-in-time*

Uma forma de se aumentar o desempenho de linguagens interpretadas como Python, ou Julia é o uso da compilação *just-in-time* (JIT). Usualmente, o código é interpretado para ser executado e o tempo gasto com a interpretação pode ser considerável. Algo semelhante acontece em Java, onde após a pré-compilação em byte-code (javac), o código é interpretado (java), dessa forma os processos de compilação e execução ficam explícitos.

Em casos que a perca de tempo interpretando é grande o bastante, a técnica do JIT se torna vantajosa já que o tempo gasto na compilação em tempo de execução é compensado pelo ganho de desempenho na execução do código em si. Porém é sempre bombom lembrar as questões ligadas ao *warm-up*, ou aquecimento, pois ao menos a primeira execução será mais lenta. Logo, assim como em experimentos podemos cortar os valores extremos (*outliers*), os valores iniciais, podem ser ignorados.

Vale lembrar que também pode ocorrer um efeito contrário, de uma aceleração artificial de programas após as primeiras execuções, isso geralmente ocorre, pois os níveis de cache já podem conter dados relevantes para às execuções seguintes. Nesses, casos, para uma análise de desempenho justa é importante limpar as memórias cache. Isso pode ser feito facilmente alternando diferentes programas a serem medidos.

4.4.6. *Profiling*

Profiling é uma forma de análise dinâmica que visa medir diversos aspectos de um programa. Essa análise pode ser realizada com pouco impacto no desempenho do programa ao utilizar funcionalidades disponíveis em processadores. Para análises mais completas ou em sistemas que não possuem suporte de hardware, é possível utilizar técnicas de virtualização que, apesar de reduzirem o desempenho do programa durante o processo, permitem uma análise mais detalhada do comportamento do mesmo.

A linguagem Go provê um ambiente prático por incluir diversas ferramentas para analisar o comportamento do programa, como a *go tool pprof*. A versão `mandelbrot_prof.go`[23] possui as alterações necessárias no código da versão em Go do programa para realizar a coleta dos dados para a ferramenta. Com as alteração, apesar de se comportar como o programa anterior, a versão modificada gera um arquivo `cpu.prof` que contem os detalhes da execução do programa. Como a ferramenta é baseada na coleta de amostras em curtos intervalos do programa, execuções curtas (menos de 5 segundos) podem não trazer todo o detalhe desejado e com isso, uma possibilidade é aumentar a duração da tarefa a ser realizada.

A Tabela 4.3 foi gerada ao utilizar o comando `top` do *go tool pprof*, ela apresenta os 10 nós que representam o maior tempo de execução do programa. A coluna *flat* representa o tempo executando instruções dentro da função (nó) sem contar o tempo executando invocações internas de outras funções e a coluna *cum* conta o tempo dessas invocações internas. Como podemos observar, boa parte do tempo do programa é gasto determinando se os pontos divergem ou não (main.work) e o tempo gasto para escrever a imagem na saída padrão é ínfimo (fmt.(*buffer).writeString). Assim, o programa possui apenas uma área cuja otimizações e melhorias impactariam de forma significativa o desempenho do programa.

Devido a simplicidade do programa utilizado, o *profiling* acaba sendo uma técnica que não traz muito mais valor do que uma simples análise do código, porém em projetos maiores, as métricas geradas podem auxiliar no processo de encontrar as partes mais importantes a serem otimizadas.

[23] `https://github.com/elisauhura/ERAD-benchmarks/blob/main/Parte2/mandelbrot_prof.go`

flat	flat%	sum%	cum	cum%	node
58.11s	96.67%	96.67%	59.04s	98.22%	main.work
0.93s	1.55%	98.22%	0.93s	1.55%	runtime.asyncPreempt
0.73s	1.21%	99.43%	0.73s	1.21%	runtime.memmove
0	0%	99.43%	0.38s	0.63%	fmt.(*buffer).writeString
0	0%	99.43%	0.38s	0.63%	fmt.(*fmt).fmtS
0	0%	99.43%	0.38s	0.63%	fmt.(*fmt).padString
0	0%	99.43%	0.38s	0.63%	fmt.(*pp).doPrintln
0	0%	99.43%	0.38s	0.63%	fmt.(*pp).fmtString
0	0%	99.43%	0.38s	0.63%	fmt.(*pp).printArg
0	0%	99.43%	0.39s	0.65%	fmt.Fprintln

Tabela 4.3. `top` gerado pelo pprof para o programa Mandelbrot em Go adaptado com os argumentos: largura = 28000; altura = 28000; xr = -1.5; xi = -1.5; yr = 1.5; yi = 1.5; máximo de iterações = 100; workers = 12

4.4.7. Soma de verificação

Um ponto crucial que não pode ser esquecido é o fato que desempenho deixa de ser relevante quando o resultado obtido não é o esperado. Ou seja, antes de se falar em melhorar o desempenho é essencial sempre garantir (de preferência por meio de testes automatizados) a corretude do programa. Talvez o exemplo mais conhecido foi o erro de divisão de ponto flutuante dos primeiros processadores Pentium [24].

Mas, claro que para saídas muito grandes, não é viável sempre comparar uma saída completa, para evitar isso, uma forma é usar algo como uma função de *hash*, permitindo que seja necessário guardar e comparar apenas pequenas strings no lugar de de toda a saída do programa. Além de simplificar o processo de validar a saída de um programa, ela pode reduzir o tempo de escrita, pois os algoritmos de *hash* podem funcionar como dispositivos de escrita que não exigem chamadas de sistema. A versão `mandelbrot_sha.go`[25] possui as alterações necessárias para utilizar a função SHA-256[26] como função dispositivo de saída.

Outro ponto importante é que guardar a saída do programa pode ajudar a detectar erros de natureza não determinísticas, como condições de corrida.

4.4.8. Diferença de desempenho linguagens

Como um assunto extra, a Tabela 4.4 apresenta o tempo médio de execução dos diferentes programas para uma mesma entrada (utilizando 4 *workers* nas versões em C e Go). É possível observar uma separação considerável entre as versões de linguagens compiladas e a versão em Python. Mesmo utilizando um interpretador JIT, a versão em Python continua uma ordem de grandeza mais lenta que as versões em C e em Go.

[24] `https://en.wikipedia.org/wiki/Pentium_FDIV_bug`

[25] `https://github.com/elisauhura/ERAD-benchmarks/blob/main/Parte2/mandelbrot_sha.go`

[26] `https://pkg.go.dev/crypto/sha256`

Versão	Tempo médio	Linhas de Código
Go	0,064s	80
C	0,068s	145
Python (Pypy)	0,562s	44
Python (CPython)	7,421s	44

Tabela 4.4. Tempo médio de 10 execuções dos programas com os argumentos: largura = 1600; altura = 1600; xr = -1.5; xi = -1.5; yr = 1.5; yi = 1.5; máximo de iterações = 50; workers = 4

Ao comparar C e Python é possível argumentar que apesar do pior desempenho, o programa possui um menor número de linhas de código. Porém, a versão em Go conseguiu apresentar o mesmo desempenho que a versão em C e quase metade das linhas de código ao mesmo tempo que provê funcionalidades como Coletor de Lixo e CSP[27] na linguagem. Uma possível razão para Go ter apresentado um desempenho um pouco melhor que C se dá pela linguagem fazer uso de *goroutines*[28], que apresentam um overhead menor que as *threads* do *pthreads*.

Os programas executados na parte 2 rodaram em MacBook Pro (15", 2019) com um processador Intel I7-9750H 6-core, 16 GB 2400 MHz DDR4 e rodando o MacOS 12.2.1 com o compilador C Apple clang version 13.0.0, compilador Go go1.17.8 e interpretadores Python 3.9.10 e PyPy 7.3.8.

4.5. Conclusão

Nesse texto apresentamos diversas técnicas de medir desempenho e de mostrar os resultados com relevância estatística. O nosso objetivo foi mostrar que é possível sistematizar a análise de desempenho. Além disso, o desempenho depende de diversos fatores, entre eles a linguagem de programação e/ou o compilador usado. Esperamos que esse texto sirva como um começo de uma jornada!

Referências

[Bruel et al. 2017] Bruel, P., Amaris Gonzalez, M., and Goldman, A. (2017). Autotuning cuda compiler parameters for heterogeneous applications using the opentuner framework. *Concurrency and Computation Practice and Experience*, 29.

[Hoos 2012] Hoos, H. H. (2012). Programming by optimization. *Commun. ACM*, 55(2):70–80.

[Jain 1991] Jain, R. (1991). *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. Wiley New York.

[Kalibera and Jones 2013] Kalibera, T. and Jones, R. (2013). Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 International Symposium on Memory Ma-*

[27] https://en.wikipedia.org/wiki/Communicating_sequential_processes
[28] https://go.dev/doc/effective_go#goroutines

nagement, ISMM '13, page 63–74, New York, NY, USA. Association for Computing Machinery.

Capítulo
5

Apresentação de Resultados Experimentais para Processamento de Alto Desempenho em `R`

Vinícius Garcia Pinto
Centro de Ciências Computacionais, Universidade Federal do Rio Grande
Rio Grande, Brasil

Lucas Leandro Nesi, Lucas Mello Schnorr
Instituto de Informática, Universidade Federal do Rio Grande do Sul
Porto Alegre, Brasil

Resumo

A apresentação de resultados de experimentos que avaliam desempenho é uma etapa onipresente nas pesquisas de processamento de alto desempenho (PAD). Uma apresentação adequada é essencial para compreensão e reprodutibilidade dos resultados relatados. Tal etapa, entretanto, é frequentemente subestimada, sendo conduzida de maneira ad hoc*, utilizando ferramentas inapropriadas e com pouca rastreabilidade, como por exemplo, editores de planilhas eletrônicas. Neste capítulo, apresentaremos uma abordagem baseada na linguagem* `R` *abrangendo desde o processamento dos dados brutos até a representação visual. A metodologia a ser apresentada explora os pacotes* `dplyr`*,* `tidyr`*,* `readr`*,* `patchwork`*,* `ggplot2` *e* `plotly` *para permitir que todas as transformações e manipulações aplicadas aos dados sejam documentadas e reprodutíveis. O conteúdo abordado está organizado em três seções principais: introdução à linguagem* `R`*, processamento de dados brutos e visualização de dados. A discussão de cada seção é acompanhada de exemplos práticos dos tópicos apresentados.*

5.1. Considerações Iniciais

O processo metodológico para realização de experimentos em computação de alto desempenho pode ser organizado em duas grandes etapas: a coleta dos dados e a análise dos resultados. A coleta dos dados é o momento em que ocorre a execução de fato do experimento no ambiente computacional de alto desempenho. O sucesso desta etapa depende dos apropriados controle e coleta dos parâmetros da plataforma computacional e

da definição de um projeto experimental [Pinto et al. 2020]. Já a etapa de análise de resultados, onde acontece o pré-processamento, filtragem, transformação, interpretação e apresentação das conclusões, é realizada *a posteriori*. É nessa etapa que se situa este minicurso.

A reprodutibilidade da análise e da apresentação dos resultados é um aspecto chave do processo científico. Entretanto é frequentemente subestimada ou realizada com ferramentas com pouca rastreabilidade. Em diferentes áreas do conhecimento, existem relatos de problemas [Hutson 2010, Herndon et al. 2014], equívocos ou retratações decorrentes de falhas metodológicas ou de manipulação e interpretação dos dados. Uma forma de fortalecer a reprodutibilidade da apresentação de resultados e reduzir riscos é investir em automatização de processos (*scripting*), na rastreabilidade, no versionamento e na apresentação. Tais conceitos nem sempre são plenamente acessíveis ou praticáveis em ferramentas interativas como editores de planilhas eletrônicas. Entretanto, são largamente utilizados no desenvolvimento de *software*. Neste minicurso, ilustramos como podemos usar a linguagem `R` e seu ambiente para ir desde o processamento dos dados brutos até a representação visual. Tal abordagem é facilmente documentável, rastreável e reprodutível.

Este capítulo está organizado em três grandes seções. Primeiramente, a Seção 5.2 apresenta uma introdução à linguagem `R` discutindo os comandos e funcionalidades essenciais. Nessa seção, são abordados apenas recursos providos pela biblioteca de base. Em seguida, a Seção 5.3 trata do processamento de dados brutos em formato texto. As transformações apresentadas nessa seção são providas pelos seguintes pacotes: `readr` [Wickham and Hester 2021], `magrittr` [Bache and Wickham 2022], `dplyr` [Wickham et al. 2021], `tidyr` [Wickham and Girlich 2022] e, finalmente, o pacote `tibble` [Müller and Wickham 2021]. Esses pacotes são disponibilizados pelo metapacote `tidyverse` [Wickham et al. 2019]. Por fim, a visualização de dados é discutida na Seção 5.4 onde empregamos funcionalidades dos pacotes `ggplot2` [Wickham 2016], `patchwork` [Pedersen 2020] e `plotly` [Sievert 2020].

5.2. Introdução à linguagem `R`

Nas próximas seções discutiremos alguns aspectos fundamentais da linguagem `R`. O objetivo é introduzir a linguagem e suas características a leitores e leitoras que tenham pouca ou nenhuma experiência com a mesma. O conteúdo abordado restringe-se ao necessário no escopo deste minicurso e não abrange outras funcionalidades da biblioteca base da linguagem `R`. Por exemplo, para a leitura de dados e construção de gráficos, utilizaremos pacotes específicos.

5.2.1. Comandos básicos

`R` é uma linguagem interpretada, logo, a interação com o usuário acontece primordialmente por uma interface de linha de comandos (CLI), dispensando comandos específicos para entrada e saída básicas. Os comandos de atribuição (`<-` e `->`) possibilitam o armazenamento de valores na memória, resultando na criação ou alteração de um objeto `R`. Para simplificar, vamos considerar um objeto `R` como equivalente a uma variável em linguagens de propósito geral como `C`. Operações aritméticas básicas podem ser obtidas

com os operadores usuais como −, +, ⋆, / e ^, assim como as relacionais com <, >, ==, !=, >= e <=. O caractere # é usado para inserção de comentários de linha.

```
x <- 100            # x recebe 100
100/3 + 2^2 -> y    # y recebe 37.33333...
z <- "escola regional de alto desempenho"
```

Código 5.1. Exemplo de comandos básicos em R.

5.2.2. Funções e Pacotes

Além dos diversos operadores e funções disponíveis por padrão no R, podemos definir nossas próprias funções atribuindo uma construção `function` a um nome. Os argumentos são opcionais e podem conter valores predefinidos (*default values*). A construção `return` pode ser usada para retornar explicitamente algum valor. Todavia, na ausência desta, por padrão, R retorna o valor da última linha de código avaliada da função.

```
aceleracao <- function(tseq, tpar){
    return(tseq/tpar)
}
```

Código 5.2. Exemplo de definição de função.

Funções R usualmente são agrupadas e distribuídas em pacotes. O CRAN[1] (*Comprehensive R Archive Network*) é o repositório oficial da linguagem e contém milhares de pacotes desenvolvidos pela comunidade. Pacotes do CRAN podem ser instalados com o comando `install.packages` e uma vez instalados podem ser carregados com o comando `library`, atualizados com o comando `update.packages` ou removidos com `remove.packages`. A documentação de um pacote ou de uma função pode ser acessada por meio do comando `?` seguido do respectivo nome (*e.g.*, `?round`).

```
install.packages("ggplot2")
library(ggplot2)

ggplot(mtcars, aes(x = mpg, y = cyl, color = gear)) + geom_point()
```

Código 5.3. Exemplo de instalação e uso do pacote `ggplot2`.

5.2.3. Tipos básicos

Vetores são o tipo mais simples de objeto R e podem armazenar apenas valores de um mesmo tipo básico. Os tipos básicos possíveis são `logical`, `integer`, `numeric`, `complex`, `character` e `raw` [R Core Team 2021]. Objetos com um único valor, como `x` e `z` declarados no Código 5.1, são vetores de tamanho 1. Vetores de tamanho zero também são permitidos. A indexação de vetores R inicia em 1 e estes podem ter mais de uma dimensão, constituindo matrizes ou de maneira mais geral, vetores multidimensionais. Em contrapartida, uma lista é similar a um vetor porém admite valores de diferentes tipos básicos. Tanto listas quanto vetores podem ter seus valores internos nomeados, possibilitando que os nomes dos valores sejam usados para indexação. O código 5.4 ilustra três maneiras para definição de vetores. A função `c` permite combinar valores individuais em um vetor. Enquanto o operador `:` e a função `seq` são usados para geração de sequências.

[1] `https://cran.r-project.org/`

```
d <- c(1, 5, 3)   # d recebe 1 3 5
e <- 2:6          # e recebe 2 3 4 5 6
f <- seq(9, 5)    # f recebe 9 8 7 6 5
```

Código 5.4. Exemplos de definição de vetores.

Data frames são as estruturas mais usadas para armazenamento de dados experimentais representando casos (linhas) por variáveis (colunas). Cada *data frame* é uma lista de vetores, todos com o mesmo tamanho. Cada vetor desta lista possui um nome e é usado para representar uma variável [Grolemund 2014]. No restante deste capítulo, usaremos *Tibbles* [Müller and Wickham 2021] que são uma implementação moderna dos *Data frames*.

5.2.4. Aritmética com vetores

Vetores `R` podem ser usados como operandos em expressões aritméticas. Nestes casos, a operação é dita vetorial, ou seja, a mesma operação é aplicada em cada um dos valores armazenados. O Código 5.5 ilustra este comportamento. O vetor `b` recebe o resultado da expressão `a^3`, isto é, um novo vetor de mesmo tamanho que `a` onde cada valor deste é elevado ao cubo. Por fim, no caso de `c`, cada posição receberá a soma dos valores armazenados nas mesmas posições de `a` e `b`.

```
a <- 1:5      # a recebe 1  2  3  4   5
b <- a ^ 3    # b recebe 1  8 27 64 125
c <- a + b    # c recebe 2 10 30 68 130
```

Código 5.5. Exemplo do uso de vetores em expressões aritméticas.

No exemplo acima, os três vetores possuem o mesmo tamanho. Entretanto, quando vetores diferentes estão envolvidos na mesma expressão, o resultado da avaliação será um vetor com o tamanho do maior dos vetores presentes na expressão. Os vetores menores serão *reciclados* quantas vezes necessário, até atingir o maior tamanho [Venables et al. 2021]. Um aviso é emitido toda vez que o tamanho do maior vetor não é múltiplo do tamanho do menor. O Código 5.6 exemplifica estas situações.

```
i <- 3:8      # i recebe 3 4 5 6 7 8
j <- 0:1      # j recebe 0 1
k <- i * j    # k recebe 0 4 0 6 0 8

m <- 5:7      # m recebe 5 6 7
n <- m - j    # n recebe 5 5 7 (um aviso é emitido)
```

Código 5.6. Exemplo de expressões aritméticas com vetores de tamanhos distintos.

5.2.5. *Scripts* R

Uma maneira apropriada para disponibilização das etapas e procedimentos realizados durante a análise de resultados experimentais é através de um arquivo `.R` contendo todos os comandos necessários. A criação destes arquivos de comandos (*script*) permite automatizar a reexecução da análise dos dados e das etapas posteriores como a geração de gráficos. Tal automatização contribui para melhor reprodutibilidade visto que há um rastreamento integral desde a leitura dos dados brutos até a apresentação final. O Código 5.7

demonstra como os argumentos informados na linha de comando podem ser recuperados no `R` enquanto o Código 5.8 ilustra a invocação do *script* em um `shell` Linux.

```
#!/usr/bin/Rscript
args <- commandArgs(trailingOnly = TRUE)

print(paste("primeiro argumento:", args[1]))
print(paste("segundo argumento:",  args[2]))
print(paste("terceiro argumento:", args[3]))
```

Código 5.7. Exemplo de recuperação de argumentos da linha de comando.

```
./meuscript.R arg1 arg2 arg3
```

Código 5.8. Invocação de um *script* R.

5.3. Manipulação de resultados em texto puro

Nesta seção abordaremos o processamento dos dados brutos, usualmente obtidos em texto puro após a execução de um experimento. Iniciaremos pela leitura dos dados utilizando o pacote `readr`. Em seguida, empregaremos o pacote `dplyr` para ilustrar uma série de manipulações gerais como a seleção de um subconjunto de colunas (variáveis), reordenamento e filtragem de linhas (observações). O pacote `tidyr` é usado para pivotar o conjunto de dados, reorganizando-o com mais ou menos colunas o que facilita algumas operações. Por fim, utilizaremos novamente o pacote `dplyr` para realização de derivações, criando uma nova coluna a partir das outras colunas ou de dados externos, sumarizações, sintetizando dados de um grupo ou ainda combinações, unindo diferentes conjuntos de dados a partir de uma chave em comum.

5.3.1. Leitura

O pacote `readr` pode ser utilizado para a leitura de dados "retangulares". Estes dados estão estruturados em uma tabela, onde cada linha do documento é uma linha da tabela, e as células estão separadas por um delimitador comum. Cada coluna é uma variável, e cada linha é um caso. Um exemplo clássico deste tipo de documento é o CSV (*Comma-separated values* – valores separados por vírgula). O Código 5.9 apresenta um exemplo de documento CSV com quatro variáveis (nas colunas: Maquina, A, B e Tempo) explicitadas na primeira linha do arquivo, e oito casos, da segunda até a oitava linha. Por exemplo, o caso da última linha é a situação onde Máquina é `Mac2`, A é `1`, B é `1` e Tempo é `18.0`.

```
Maquina,A,B,Tempo
Mac1,0,0,20.0
Mac1,0,1,18.0
Mac1,1,0,16.0
Mac1,1,1,12.0
Mac2,0,0,25.0
Mac2,0,1,20.0
Mac2,1,0,25.0
Mac2,1,1,18.0
```

Código 5.9. Exemplo de um arquivo CSV.

A leitura de um arquivo CSV em R pode ser realizada com a função `read_csv`. Esta função recebe como parâmetro obrigatório o caminho do arquivo. Outros parâmetros adicionais úteis são: *col_types* para especificar o nome e o tipo de cada coluna, e *progress* para mostrar ou não o progresso da leitura. Os tipos de coluna comumente utilizados são *col_character* para *strings*, *col_integer* para inteiros, e *col_double* para números decimais. O seguinte bloco de código mostra um exemplo de utilização desta função para a leitura de dados que se encontram no arquivo `exemplo.csv` (com dados do Código 5.9). A função, após leitura dos dados do arquivo, retorna um `tibble`, que é um tipo para dados tabulares sendo atribuído para a variável `df`:

```
read_csv("exemplo.csv",
         col_types = list(Maquina = col_character(),
                          A       = col_integer(),
                          B       = col_integer(),
                          Tempo   = col_double()),
         progress = FALSE) -> df_ex
df_ex
```

```
# A tibble: 8 x 4
  Maquina     A     B Tempo
  <chr>   <int> <int> <dbl>
1 Mac1        0     0    20
2 Mac1        0     1    18
3 Mac1        1     0    16
4 Mac1        1     1    12
5 Mac2        0     0    25
6 Mac2        0     1    20
7 Mac2        1     0    25
8 Mac2        1     1    18
```

Para realizar a leitura de um arquivo retangular estruturado que não seja um CSV, mas que tenha um delimitador comum, pode-se utilizar a função `read_delim`. Um segundo parâmetro obrigatório desta função é *delim*, o delimitador entre células. Basicamente, o `read_csv` é uma chamada para o `read_delim` com *delim=","*. Todas estas funções retornam um objeto do tipo `tibble`.

5.3.2. Encadeamento

O operador `%>%` é um operador provido pelo pacote `magrittr`[2] que possibilita o encadeamento de operações de maneira similar ao mecanismo *pipe* dos sistemas do tipo Unix. Este é um operador binário onde o operando esquerdo é uma variável (incluindo os retornos de funções) e o operando direito é uma função com pelo menos um parâmetro. O operador permite a passagem do operando da esquerda (variável) como o primeiro argumento do operador da direita (função). Desta forma, uma chamada originalmente escrita como `fun(x, y)` pode ser realizada com o operador pipe como `x %>% fun(y)`. Este operador permite uma melhor redação de código, já que as várias operações estarão escritas (concatenadas) na ordem em que elas ocorrem.

[2]Recentemente, na versão R 4.1.0, foi adicionado na linguagem um comando de *pipe* nativo com a sintaxe `|>`.

5.3.3. Transformações

O comando `select` é utilizado para selecionar colunas dos dados. Para tal, basta passar como parâmetro o nome das colunas desejadas para seleção assim:

```
df_ex %>% select(Maquina, Tempo)
```

```
# A tibble: 8 x 2
  Maquina Tempo
  <chr>   <dbl>
1 Mac1       20
2 Mac1       18
3 Mac1       16
4 Mac1       12
5 Mac2       25
6 Mac2       20
7 Mac2       25
8 Mac2       18
```

Outra operação recorrente é a ordenação dos dados. Para isto, pode-se utilizar o comando `arrange`. A ordenação pode ser realizada utilizando múltiplas colunas, onde os dados primeiramente são ordenados pela primeira coluna informada, em seguida pela segunda, e assim por diante. Então, para os casos com a primeira coluna com valor repetido, acontece a ordenação pelos valores das colunas subsequentes.

```
df_ex %>% arrange(A, B)
```

```
# A tibble: 8 x 4
  Maquina     A     B Tempo
  <chr>   <int> <int> <dbl>
1 Mac1        0     0    20
2 Mac2        0     0    25
3 Mac1        0     1    18
4 Mac2        0     1    20
5 Mac1        1     0    16
6 Mac2        1     0    25
7 Mac1        1     1    12
8 Mac2        1     1    18
```

5.3.4. Filtragem

O comando `filter` é utilizado para realizar a filtragem de casos que satisfazem alguma condição nas suas variáveis. O seu argumento principal é a condição lógica que os casos devem satisfazer para passarem pelo filtro. Por exemplo, considerando o conjunto de dados anterior, gostaríamos de filtrar os dados que foram executados na máquina `Mac1`. Desta forma, pode-se utilizar o seguinte comando:

```
df_ex %>% filter(Maquina == "Mac1")
```

```
# A tibble: 4 x 4
  Maquina     A     B Tempo
  <chr>   <int> <int> <dbl>
1 Mac1        0     0    20
```

```
2 Mac1         0     1    18
3 Mac1         1     0    16
4 Mac1         1     1    12
```

Outros operadores lógicos podem ser utilizados para especificar as necessidades de filtragem. Um exemplo mais elaborado é a filtragem de casos na máquina `Mac1` ou `Mac2` onde o tempo de execução é maior que 18s. Em outras linguagens de programação, a operação lógica de máquinas seria descrita usualmente utilizando `Maquina = "Mac1" OU Maquina = "Mac2"`. Entretanto, em `R` podemos utilizar o operador binário `%in%` , ou seja, `váriavel %in% vetor` onde a operação é verdade se a variável está presente no conjunto de elementos do vetor.

```
df_ex %>% filter(Maquina %in% c("Mac1", "Mac2") & Tempo > 18)
```

```
# A tibble: 4 x 4
  Maquina     A     B Tempo
  <chr>   <int> <int> <dbl>
1 Mac1        0     0    20
2 Mac2        0     0    25
3 Mac2        0     1    20
4 Mac2        1     0    25
```

Ainda é possível selecionar linhas específicas dos dados. Isso é realizado com o comando `slice(linhas)`, onde `linhas` é um vetor que tem os números das linhas que devem ser selecionadas. Por exemplo, para selecionar as linhas de 2 até 4, pode-se utilizar:

```
df_ex %>% slice(2:4)
```

```
# A tibble: 3 x 4
  Maquina     A     B Tempo
  <chr>   <int> <int> <dbl>
1 Mac1        0     1    18
2 Mac1        1     0    16
3 Mac1        1     1    12
```

5.3.5. Pivotagem

A organização de dados recomendada é conhecida como *tidy* [Wickham 2014]. Nesta abordagem, cada coluna é uma variável, cada linha é uma observação, e cada célula contêm um valor único. Entretanto, existem duas abordagens principais de organização dos dados que são chamadas de "longas" e "largas". A organização de dados "larga" utiliza mais colunas e diminui a quantidade de linhas. Nesta abordagem, cada linha contém diversas observações. Na organização "longa", cada coluna é uma variável. A Tabela 5.1 apresenta estas duas organizações.

Pode-se utilizar os comandos `pivot_wider` e `pivot_longer` para transformar dados de uma organização em outra. Com o comando `pivot_longer`, transformamos uma organização larga em longa informando: as colunas que desejamos unificar, a coluna que vai receber o nome destas variáveis (*names_to*), e a coluna que vai receber o valor destas variáveis (*values_to*). No exemplo abaixo, construímos duas tabelas através

Nome	Caso_A	Caso_B	Caso_C
α	1	2	3
β	4	5	6

Nome	Variavel_1	Valor
α	Caso_A	1
α	Caso_B	2
α	Caso_C	3
β	Caso_A	4
β	Caso_B	5
β	Caso_C	6

Tabela 5.1. Dados organizados de maneira larga (esquerda) e longa (direita), sendo esta organização também conhecida como *tidy*.

da função `tibble` que recebe como parâmetro uma lista de elementos nomeados, onde os nomes são as variáveis das tabelas (`Nome` `Caso_A`, e assim por diante) e o conteúdo são vetores de dados.

```
df_wider <- tibble(Nome = c("Alpha", "Beta"),
                   Caso_A = c(1, 4),
                   Caso_B = c(2, 5),
                   Caso_C = c(3, 6))
pivot_longer(df_wider,
             c(Caso_A, Caso_B, Caso_C),
             names_to = "Variavel_1",
             values_to = "Valor") -> df_longer
df_longer
```

```
# A tibble: 6 x 3
  Caso  Variavel_1 Valor
  <chr> <chr>      <dbl>
1 Alpha Caso_A         1
2 Alpha Caso_B         2
3 Alpha Caso_C         3
4 Beta  Caso_A         4
5 Beta  Caso_B         5
6 Beta  Caso_C         6
```

A operação reversa é realizada com a função `pivot_wider` e com os argumentos opostos. Informando a coluna que contêm os nomes das variáveis (*names_from*) e a coluna que contêm os valores (*values_from*).

```
df_longer %>% pivot_wider(names_from = `Variavel_1`,
                          values_from = Valor)
```

```
# A tibble: 2 x 4
  Caso  Caso_A Caso_B Caso_C
  <chr>  <dbl>  <dbl>  <dbl>
1 Alpha      1      2      3
2 Beta       4      5      6
```

5.3.6. Derivações

A criação de uma nova coluna pode ser realizada com a operação `mutate`, informando o nome da nova coluna e seus valores. Os valores podem ser derivados de outras colunas ou utilizando vetores externos ao *dataset*.

```
df_ex %>% mutate(D = A+B)
```

```
# A tibble: 8 x 5
  Maquina     A     B Tempo     D
  <chr>   <int> <int> <dbl> <int>
1 Mac1        0     0    20     0
2 Mac1        0     1    18     1
3 Mac1        1     0    16     1
4 Mac1        1     1    12     2
5 Mac2        0     0    25     0
6 Mac2        0     1    20     1
7 Mac2        1     0    25     1
8 Mac2        1     1    18     2
```

Para sumarizar os dados, i.e., realizar uma transformação sobre um determinado grupo para gerar um valor final, pode-se utilizar a operação `summarize`. Uma função comumente utilizada é a função de média sobre os valores:

```
df_ex %>% summarize(media_a = mean(A))
```

```
# A tibble: 1 x 1
  media_a
    <dbl>
1     0.5
```

5.3.7. Combinando Tabelas

Dados podem ser organizados em diferentes tabelas que compartilhem informações. Estes compartilhamentos de dados são relações entre as tabelas. Por exemplo, podemos ter uma tabela descrevendo as máquinas e uma tabela de experimentos nas máquinas (conforme podemos visualmente observar na Tabela 5.2):

```
maquinas <- tibble(Nome = c("Mac1", "Mac2"),
                   Cores = c(20, 24))
experimentos <- tibble(Maquina = c("Mac1", "Mac1", "Mac2", "Mac2"),
                       Parametro = c(0, 1, 0, 1),
                       Tempo = c(20, 10, 24, 12))
```

Se desejamos adicionar os valores de *Cores* para a tabela de experimentos, podemos realizar a operação de união de tabelas utilizando o comando `inner_join` (Vários outros comandos `*_join` estão disponíveis para necessidades variadas). A operação recebe duas tabelas para serem unidas, e um argumento especificando quais colunas devem ser utilizadas para a combinação. No exemplo de máquinas e experimentos, devemos realizar a combinação da coluna `Maquina` da tabela `experimentos` com a coluna `Nome` da tabela `maquinas`. Informamos esta combinação com o parâmetro *by* passando um vetor nomeado onde cada elemento tem como nome a coluna da tabela e o valor a coluna da segunda tabela. Tais combinações podem ser feitas com mais de uma variável.

Nome	Cores
Mac1	20
Mac2	24

Máquina	Parâmetro	Tempo
Mac1	0	20
Mac1	1	10
Mac2	0	24
Mac2	1	12

Tabela 5.2. Tabela das máquinas (esquerda) e de experimentos (direita).

```
inner_join(experimentos, maquinas, by = c("Maquina" = "Nome"))
```

```
# A tibble: 4 x 4
  Maquina Parametro Tempo Cores
  <chr>       <dbl> <dbl> <dbl>
1 Mac1            0    20    20
2 Mac1            1    10    20
3 Mac2            0    24    24
4 Mac2            1    12    24
```

5.3.8. Grupos

Os dados podem ser organizados em grupos, ou seja, casos que compartilham o mesmo valor em uma ou mais variáveis. Esse agrupamento é realizado com a função `group_by` e ele é importante já que os comandos do `dplyr` podem atuar sobre grupos. Por exemplo, considerando o dataset anterior, podemos criar grupos para cada máquina utilizada com o seguinte comando `group_by`, que recebe dois argumentos: os dados e as variáveis que serão utilizadas para o agrupamento (neste caso, apenas a variável `Maquina`):

```
df_ex %>% group_by(Maquina)
```

```
# A tibble: 8 x 4
# Groups:   Maquina [2]
  Maquina     A     B Tempo
  <chr>   <int> <int> <dbl>
1 Mac1        0     0    20
2 Mac1        0     1    18
3 Mac1        1     0    16
4 Mac1        1     1    12
5 Mac2        0     0    25
6 Mac2        0     1    20
7 Mac2        1     0    25
8 Mac2        1     1    18
```

O *tibble* resultante contêm o seguinte cabeçalho:

```
# A tibble: 8 x 4
# Groups:   Maquina [2]
```

Assim, casos que compartilham o mesmo valor para a variável Maquina estarão logicamente organizados e divididos em grupos. A mensagem `# Groups: Maquina`

`[2]` mostra que existem dois grupos, já que existem dois valores possíveis para máquina. Considerando os grupos, podemos realizar o comando `slice(1)`, onde cada grupo será recortado individualmente:

```
df_ex %>% group_by(Maquina) %>% slice(1)
```

```
# A tibble: 2 x 4
# Groups:   Maquina [2]
  Maquina     A     B Tempo
  <chr>   <int> <int> <dbl>
1 Mac1        0     0    20
2 Mac2        0     0    25
```

5.4. Construção de gráficos

Com uma perspectiva histórica, a linguagem `R` possui um extenso histórico de geração de gráficos estatísticos sendo este suporte majoritariamente nativo, embutido na biblioteca de base da linguagem. Ao longo dos anos, surgiu um pacote para esta linguagem chamado `ggplot2` que incorpora uma importante evolução no que diz respeito a forma como os gráficos são construídos. Este pacote implementa o que é conhecido por uma Gramática de Gráficos [Wilkinson 2012], que foi uma primeira proposta para prover uma teoria formal para criar visualização de dados. Nós veremos aqui a visão em camadas desta gramática de gráficos [Wickham 2010].

A ideia principal por trás do conceito de gramática de gráficos é que, ao construir um gráfico, se realiza de maneira explícita uma relação entre as variáveis dos dados (colunas de tabelas) com os elementos gráficos. Por exemplo, em um gráfico de pontos, o tamanho do ponto será associado a uma determinada coluna da tabela que contém os dados de origem. Esse mesmo tipo de mapeamento acontece para todos os demais elementos estéticos tais como forma, cor, eixos horizontal e vertical, e assim por diante. Frequentemente, as características estéticas obrigatórias dependem do tipo de gráfico. Com pontos, precisamos apenas de duas coordenadas para posicioná-los, para um gráfico de retângulos, no entanto, precisamos de quatro coordenadas.

Não temos por objetivo ser exaustivos na apresentação de todos os tipos de gráficos e estéticas obrigatórias, afinal de contas temos a documentação em mãos[3]. Precisamos deixar claro que, independente do tipo de gráfico que desejamos construir, os dados devem estar organizados de maneira "longa", ou *tidy*, como já apresentado (veja Seção 5.3.3). Na terminologia do pacote `ggplot2`, os tipos de gráficos são chamados de *geoms* e o mapeamento de dados para as estéticas são definidas com a função `aes()`.

No que segue, apresentamos as principais funcionalidades do pacote `ggplot2` para a construção de gráficos. Em seguida, veremos como combinar gráficos diferentes com o pacote `patchwork`. Enfim, veremos como estes objetos gráficos podem ser transformados em gráficos interativos, com a ferramenta `plotly`. Para este percurso, adotamos um conjunto de dados disponível publicamente na internet[4] que consiste em

[3]Disponível em `https://ggplot2.tidyverse.org/reference/`

[4]Disponível em `https://github.com/dlenski/top500/blob/master/TOP500_history.csv`

dados históricos do Top500. Uma vez este arquivo estando localmente disponível, podemos ler seu conteúdo da seguinte forma:

```
FILE <- "TOP500_history.csv"
df_t500 <- read_csv(FILE, progress = FALSE, show_col_types = FALSE)
```

Os dados do Top500 são bastante numerosos. São 29 mil observações e 53 variáveis. Simplificamos os dados por razões pedagógicas selecionando apenas as sete colunas seguintes (percebam o encadeamento de comandos com o `%>%` como explicado na Seção 5.3.2), criando uma nova coluna (utilizando a função `ymd` do pacote `lubridate`) para termos uma única coluna com a data, e filtrando para obter apenas sistemas que tenham uma região definida (onde na coluna `Region` temos um valor diferente de `NA`):

```
library(lubridate)
df_t500 %>%
    filter(!is.na(Region)) %>%
    mutate(Data = ymd(paste(Year, Month, Day))) %>%
    select(Data, Rank, RPeak, Region, Nmax) %>%
    print(n=2) -> df.simple
```

```
# A tibble: 27,000 x 5
  Data        Rank RPeak Region         Nmax
  <date>     <dbl> <dbl> <chr>         <dbl>
1 1993-06-01     1 131   North America 52224
2 1993-06-01     2  69.6 North America 36864
#...with 26,998 more rows
```

Os dados do Top500 contém a posição dos diferentes sistemas computacionais de alto desempenho que foram ranqueados ao longo do tempo. Cada linha da tabela consiste em um sistema, sendo que as colunas representam as diferentes variáveis que o sistema teve naquele instante de tempo.

5.4.1. Gráficos Básicos

Para a construção de gráficos devemos realizar chamadas ao comando `ggplot` do pacote `ggplot2`. Como `ggplot` faz parte do universo de pacotes que integram o `dplyr`, ele pode ser utilizado com encadeamento de comandos (veja abaixo). Primeiro filtramos para obter apenas os sistemas que foram ranqueados em primeiro lugar (`Rank == 1`), para em seguida construir o gráfico. Usando a função `aes()`, nós mapeamos para o eixo horizontal a coluna `Data` (`x = Data`) e para o eixo vertical a coluna `RPeak` (y = `RPeak`). Isso permite definir duas estéticas básicas para o gráfico, sendo estas obrigatórias para um gráfico de pontos `geom_point()`.

```
df.simple %>%
    filter(Rank == 1) %>%
    ggplot(aes(x = Data, y = RPeak)) +
    theme_bw(base_size = 12) +
    geom_point() -> p1
p1
```

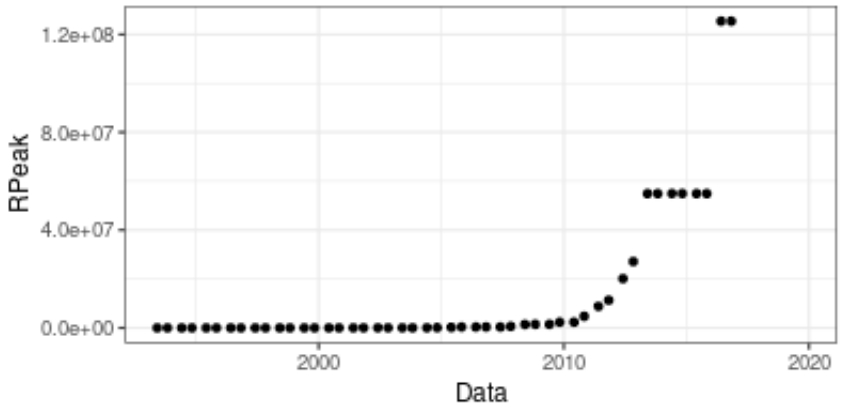
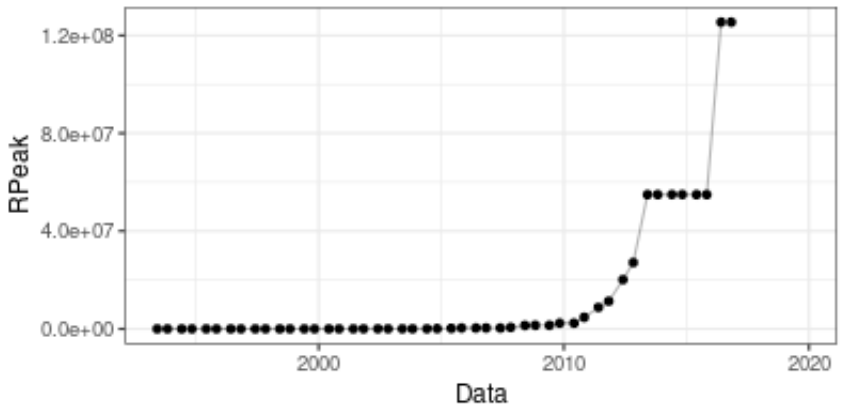


Figura 5.1. Exemplo de gráfico de pontos construído com o pacote `ggplot2`.

Um objeto `ggplot` consiste em camadas sobrepostas de tipos de gráficos potencialmente diferentes. Por exemplo, podemos unir os pontos do gráfico anterior simplesmente adicionando uma chamada `geom_line()`, que herdará a estética definida globalmente na chamada `ggplot()`. A ordem das camadas importa, portanto adicionar `geom_point() + geom_line()` é diferente graficamente do que fazer o contrário. No primeiro caso, as linhas são desenhadas por cima dos pontos. O exemplo abaixo faz o contrário, mantendo as linhas desenhadas por trás com uma certa transparência para distinguirmos a diferença:

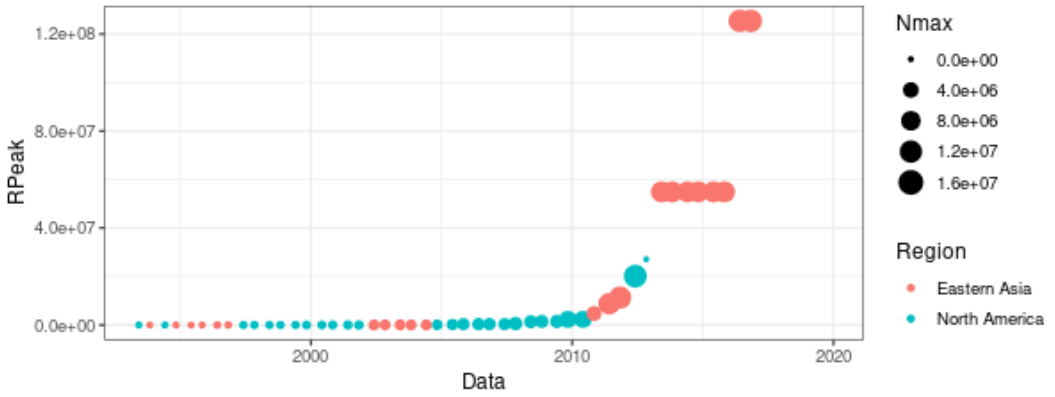
```
df.simple %>%
    filter(Rank == 1) %>%
    ggplot(aes(x = Data, y = RPeak)) +
    theme_bw(base_size = 12) +
    geom_line(alpha = 0.3) +
    geom_point()
```



Outras estéticas incluem `size` e `color`, por exemplo. Elas podem ser empregadas de maneira análoga ao que já fizemos com as coordenadas x e y. No exemplo abaixo,

mapeamos a região para a cor (`color = Region`) e o valor de `Nmax` para o tamanho (`size = Nmax`).

```
df.simple %>%
    filter(Rank == 1) %>%
    ggplot(aes(x = Data, y = RPeak, size = Nmax, color = Region)) +
    theme_bw(base_size = 12) +
    geom_point() -> p2
p2
```



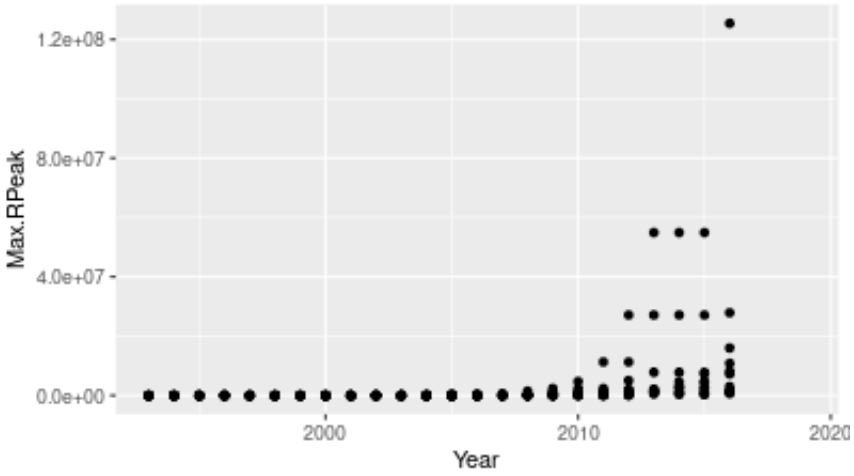
Podemos observar que automaticamente o pacote `ggplot2` define as escalas de cores e de tamanho para `Region` e `NMax` conforme os dados da tabela fornecida como entrada (`df.simple`). Observamos também que alguns sistemas que foram posicionados em primeiro lugar ao longo do tempo tem a região não definida `NA`.

5.4.2. Comparações com Facetas

O emprego de facetas permite a criação de painéis definidos por linha e coluna dependendo de variáveis categóricas que temos em nossos dados. Existem dois tipos de facetas que podem ser criadas, em grade (`facet_grid`) ou em linha (`facet_wrap`). A primeira é utilizada quando desejamos posicionar paineis em uma grade criada a partir de duas variáveis dos dados, enquanto que a segunda é utilizada quando desejamos paineis baseados apenas em uma variável, em cenários onde esta possui vários valores possíveis.

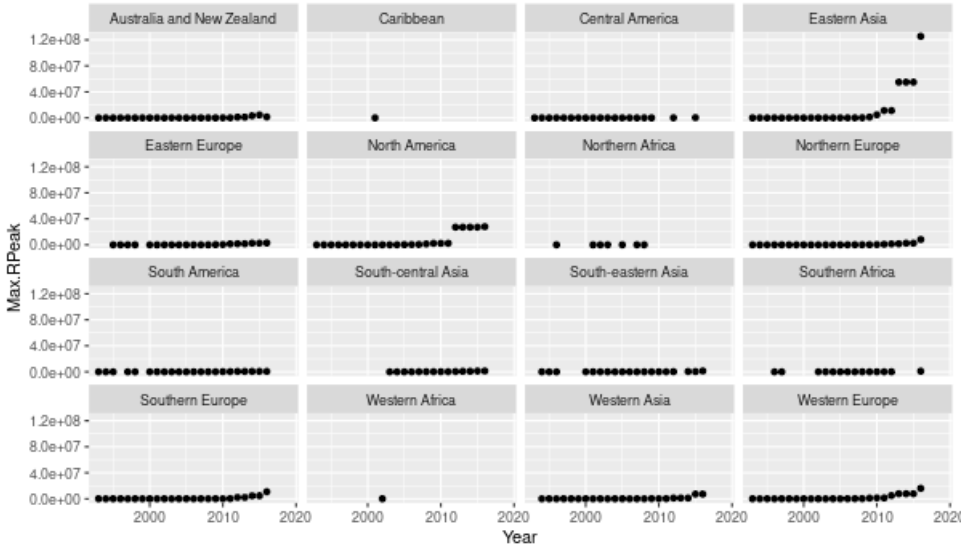
Para exemplificar o emprego de facetas, vamos primeiro agrupar os dados por ano e por região para então calcular o máximo `RPeak` observado em cada um desses grupos. Em seguida, vamos criar um gráfico onde mapeamos o ano para o eixo horizontal (`x = Year`) e o máximo `RPeak` para o eixo vertical (`y = Max.RPeak`). Construindo um gráfico de pontos (com a camada `geom_point()`), observamos que para um mesmo ano temos vários pontos representando as diferentes regiões.

```
df.simple %>%
    mutate(Year = year(Data)) %>%
    group_by(Year, Region) %>%
    summarize(Max.RPeak = max(RPeak), .groups = "keep") %>%
    ggplot(aes(x = Year, y = Max.RPeak)) +
    geom_point()
```



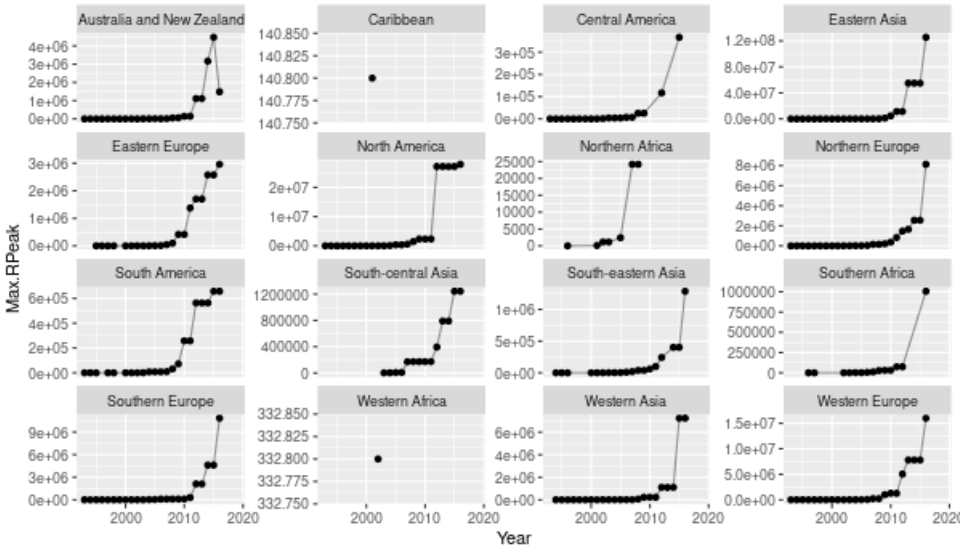
Para empregar facetas para diferenciar as regiões no gráfico acima, basta adicionarmos uma nova camada `facet_wrap()` informando sobre qual variável desejamos que tais facetas sejam criadas:

```
df.simple %>%
    mutate(Year = year(Data)) %>%
    group_by(Year, Region) %>%
    summarize(Max.RPeak = max(RPeak), .groups = "keep") %>%
    ggplot(aes(x = Year, y = Max.RPeak)) +
    geom_point() +
    facet_wrap(~Region)
```



Observamos que os gráficos compartilham as escalas de x (`Year`) e y (`Max.RPeak`). Isso normalmente é o desejado quando se analisam dados através desses paineis, mas por vezes gostaríamos de ter escalas livres para cada uma das facetas. Para fazer isso no eixo y, adicionamos um novo parâmetro ao `facet_wrap` e adicionamos linhas:

```
df.simple %>%
    mutate(Year = year(Data)) %>%
    group_by(Year, Region) %>%
    summarize(Max.RPeak = max(RPeak), .groups = "keep") %>%
    ggplot(aes(x = Year, y = Max.RPeak)) +
    geom_line(alpha = .4) +
    geom_point() +
    facet_wrap(~Region, scales = "free_y") -> p3
p3
```

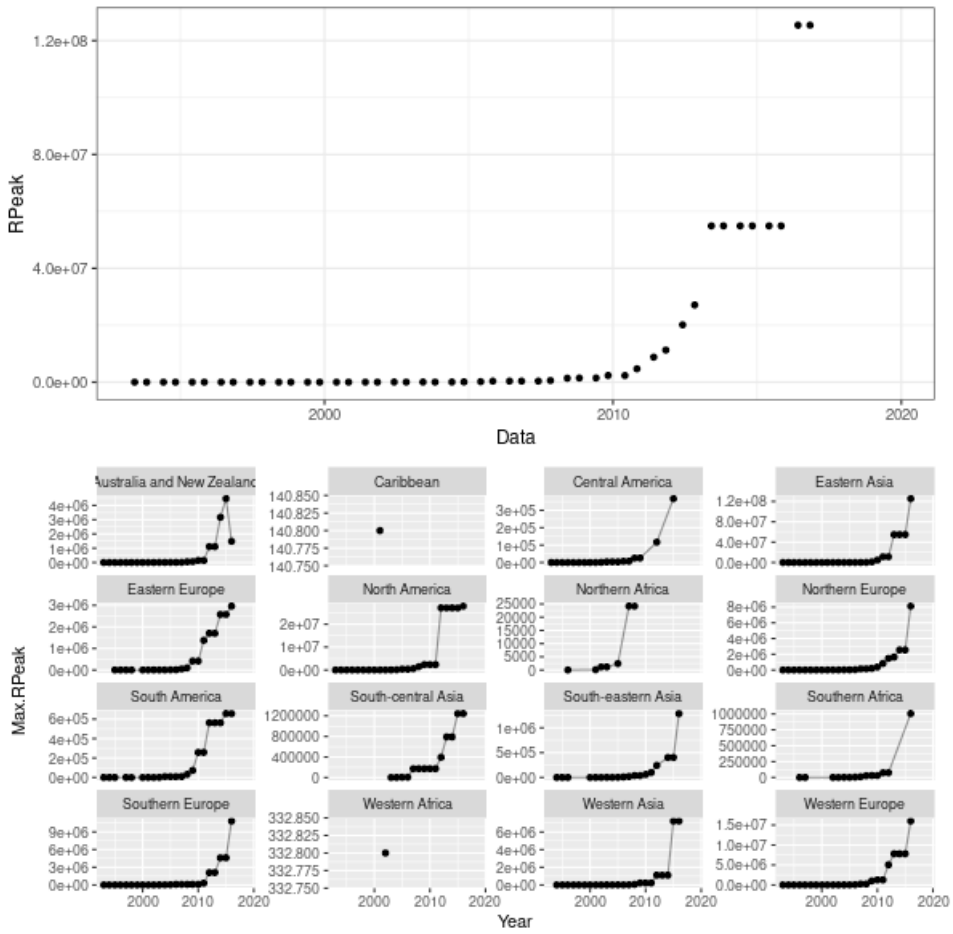


Agora temos gráficos que não são mais comparáveis visualmente entre si, pelo menos no que diz respeito ao eixo Y, mas onde podemos ver a evolução do máximo de RPeak em cada ano ao longo do tempo para cada região.

5.4.3. Composição de Gráficos

O pacote `patchwork` é utilizado para se combinar, ou compor, gráficos que tenha sido criados com `ggplot` mas que tenham características diferentes. Seu objetivo é associar os vários gráficos para compor uma única visualização. Para exemplificar seu uso, faremos uma combinação de dois gráficos gerados na seção anterior, identificados pelas variáveis `p1` e `p3`. No código abaixo, utilizamos o operador `/` para informar ao patchwork que os gráficos devem ficar uma acima do outro. Caso os quiséssemos lado a lado, poderíamos usar o operador `+`.

```
library(patchwork)
p1 / p3
```



5.4.4. Gráficos interativos

Os gráficos apresentados anteriormente são entidades estáticas. Embora suficientes para a maioria das situações, há casos em que é preciso ir além da visão geral. Tomando como exemplo o gráfico da Figura 5.1 podemos observar que é difícil distinguir entre os valores da série histórica no intervalo entre a data inicial até 2010. Tal dificuldade é consequência dos valores mais recentes, que por serem de maior grandeza, acabam por "achatar" os demais pontos. Ainda que possamos gerar novos gráficos estáticos reduzindo a faixa de valores, tal ação ainda será limitada a necessidade momentânea e pouco produtiva.

Uma solução para este e outros problemas é a utilização de gráficos interativos. Tais gráficos não substituem os equivalentes estáticos, porém são úteis para complementar as possibilidades de visualização e análise dos dados. Acrescentar interatividade permite ao usuário explorar o gráfico sob demanda, fazendo *zoom* em algumas partes, rolando o mouse para visualizar os valores absolutos de cada ponto ou ainda desativando temporariamente partes do conjunto de dados que possam estar em sobreposição com outras. O pacote `plotly` possibilita a geração de gráficos interativos que podem ser explorados em um navegador web. A função `ggplotly` realiza a tradução automática de um gráfico `ggplot2` para `plotly`. Desta maneira, podemos obter facilmente versões interativas dos gráficos estáticos criados anteriormente. A Figura 5.2 mostra uma captura de tela dos

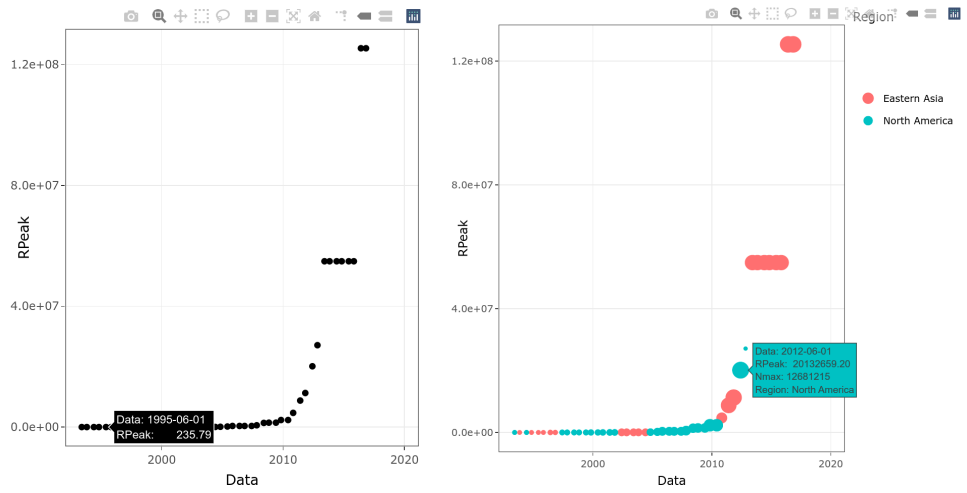


Figura 5.2. Captura de tela de dois gráficos interativos construídos com `plotly`.

dois gráficos interativos criados. As versões interativas podem ser acessadas na internet nos endereços `http://exp-hpc.gitlab.io/erad-2022/plotly-1.html` e `http://exp-hpc.gitlab.io/erad-2022/plotly-2.html`.

```
library(plotly)

ggplotly(p1)
ggplotly(p2)
```

5.5. Considerações Finais

Este capítulo ilustrou uma abordagem em `R` para apresentação de resultados experimentais para processamento de alto desempenho. Acreditamos que esta abordagem contribui para melhor documentação e reprodutibilidade dos resultados uma vez que todas as transformações aplicadas desde o processamento dos dados brutos até a construção dos gráficos pode ser facilmente rastreadas. A título de exemplo, todos os trechos de código apresentados neste texto podem ser acessados em: `http://exp-hpc.gitlab.io/erad-2022/companion.R`. Ao executar o mesmo é possível reproduzir as saídas dos trechos de código e as figuras.

Por fim, gostaríamos de lembrar que o assunto tratado neste capítulo tem estreita relação com outros três minicursos apresentados em edições recentes da ERAD/RS: “Boas Práticas para Experimentos” [Schnorr and Pinto 2019], “Boas Práticas para Experimentos Computacionais de Alto Desempenho” [Pinto et al. 2020] e “Are you root? Experimentos Reprodutíveis em Espaço de Usuário” [Dagostini et al. 2021]. Nós encorajamos as pessoas que desejam empregar essas boas práticas em seus experimentos a entrar em contato conosco para melhoramos esta série de cursos. Materiais complementares destes minicursos estão disponíveis em `https://exp-hpc.gitlab.io`.

Referências

[Bache and Wickham 2022] Bache, S. M. and Wickham, H. (2022). *magrittr: A Forward-Pipe Operator for R*. R package version 2.0.2.

[Dagostini et al. 2021] Dagostini, J. I., Pinto, V. G., Nesi, L. L., and Schnorr, L. M. (2021). Are you root? Experimentos Reprodutíveis em Espaço de Usuário. In Charão, A. and Serpa, M., editors, *Minicursos da XXI Escola Regional de Alto Desempenho da Região Sul*, chapter 3, pages 70–87. Sociedade Brasileira de Computação - SBC, Porto Alegre.

[Grolemund 2014] Grolemund, G. (2014). *Hands-On Programming with R*. O'Reilly Media, 1 edition.

[Herndon et al. 2014] Herndon, T., Ash, M., and Pollin, R. (2014). Does high public debt consistently stifle economic growth? a critique of reinhart and rogoff. *Cambridge Journal of Economics*, 38(2):257–279.

[Hutson 2010] Hutson, S. (2010). Data handling errors spur debate over clinical trial. *Nature Medicine*, 16(6):618–618.

[Müller and Wickham 2021] Müller, K. and Wickham, H. (2021). *tibble: Simple Data Frames*. R package version 3.1.6.

[Pedersen 2020] Pedersen, T. L. (2020). *patchwork: The Composer of Plots*. R package version 1.1.1.

[Pinto et al. 2020] Pinto, V. G., Nesi, L. L., and Schnorr, L. M. (2020). Boas Práticas para Experimentos Computacionais de Alto Desempenho. In du Bois, A. R. and Castro, M. B., editors, *Minicursos da XX Escola Regional de Alto Desempenho da Região Sul*, pages 1–19. Sociedade Brasileira de Computação - SBC.

[R Core Team 2021] R Core Team (2021). *R Language Definition*.

[Schnorr and Pinto 2019] Schnorr, L. M. and Pinto, V. G. (2019). Boas Práticas para Experimentos. In *Anais da XIX Escola Regional de Alto Desempenho da Região Sul*, pages 45–64. Sociedade Brasileira de Computação - SBC.

[Sievert 2020] Sievert, C. (2020). *Interactive Web-Based Data Visualization with R, plotly, and shiny*. Chapman and Hall/CRC.

[Venables et al. 2021] Venables, W. N., Smith, D. M., and R Core Team (2021). An Introduction to R. Technical report.

[Wickham 2010] Wickham, H. (2010). A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28.

[Wickham 2014] Wickham, H. (2014). Tidy data. *The Journal of Statistical Software*, 59.

[Wickham 2016] Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.

[Wickham et al. 2019] Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., Takahashi, K., Vaughan, D., Wilke, C., Woo, K., and Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686.

[Wickham et al. 2021] Wickham, H., François, R., Henry, L., and Müller, K. (2021). *dplyr: A Grammar of Data Manipulation*. R package version 1.0.7.

[Wickham and Girlich 2022] Wickham, H. and Girlich, M. (2022). *tidyr: Tidy Messy Data*. R package version 1.2.0.

[Wickham and Hester 2021] Wickham, H. and Hester, J. (2021). *readr: Read Rectangular Text Data*. R package version 2.0.1.

[Wilkinson 2012] Wilkinson, L. (2012). The grammar of graphics. In *Handbook of computational statistics*, pages 375–414. Springer.