XXVI Escola Regional de Alto Desempenho da Região Sul

XXVI ERAD/RS 2026

Bagé, RS – 6, 7 e 8 de Maio de 2026

ERAD/RS 2026

MINICURSOS

Realização

Organização

Fomento

Apoio

Patrocinadores

Prata

Bronze

André Rauber Du Bois e Sandro da Silva Camargo
(Editores)

Minicursos da XXVI Escola Regional de Alto Desempenho da Região Sul

Porto Alegre

Sociedade Brasileira de Computação – SBC

2026

Dados Internacionais de Catalogação na Publicação (CIP)

E74 Escola Regional de Alto Desempenho da Região Sul (26. : 06 – 08 maio 2026 : Bagé)
Minicursos da XXVI ERAD-RS 2026 [recurso eletrônico] / editores: André Rauber Du Bois, Sandro da Silva Camargo. Dados eletrônicos. – Porto Alegre: Sociedade Brasileira de Computação, 2026.
38 p. : il. : PDF

Modo de acesso: World Wide Web.
Inclui bibliografia
ISBN 978-85-7669-667-4 (e-book)

1. Computação – Brasil – Evento. 2. Alto desempenho. 3. Sistemas computacionais. I. Du Bois, André Rauber. II. Camargo, Sandro da Silva. III. Sociedade Brasileira de Computação. VI. Título.

CDU 004(063)

Ficha catalográfica elaborada por Annie Casali – CRB-10/2339
Biblioteca Digital da SBC – SBC OpenLib

Sociedade Brasileira de Computação
Av. Bento Gonçalves, 9500
Setor 4 | Prédio 43.412 | Sala 219 | Bairro
Agronomia Caixa Postal 15012 | CEP 91501-970
Porto Alegre - RS
(51) 99252-6018
sbc@sbc.org.br

Sumário

Capítulo

1

Concorrência sem Memória Compartilhada

Gerson Geraldo H. Cavalheiro, André Rauber Du Bois, Alexandro Baldassin

Resumo

Este texto apresenta alternativas ao modelo tradicional de programação concorrente baseado em memória compartilhada, explorando mecanismos que deslocam o foco do controle explícito de threads para a coordenação por meio de dados e comunicação. Inicialmente, são revisados os mecanismos básicos de criação de threads em C++, Rust, Go e Elixir, estabelecendo uma base comum para comparação. Em seguida, discute-se a coordenação por espera explícita, evidenciando suas limitações, e introduz-se a abstração de futures como forma de integrar produção de resultados e sincronização. Na sequência, é abordado o modelo de comunicação por troca de mensagens, com ênfase no papel dos canais como mecanismo estruturador da interação entre unidades de execução. Exemplos nas linguagens consideradas ilustram como essas abstrações se manifestam em diferentes contextos. Ao final, destaca-se a mudança de perspectiva que caracteriza modelos baseados em comunicação, nos quais a sincronização emerge da troca de dados e a coordenação deixa de ser centrada no gerenciamento direto de threads. O texto serve como material complementar para uma apresentação sobre concorrência além do modelo de memória compartilhada.

1.1. Introdução

Multithreading é um modelo de programação no qual um processo é estruturado como um conjunto de múltiplos threads de execução, cada um representando um fluxo de controle independente dentro de um mesmo espaço de endereçamento e compartilhando os recursos do processo [Silberschatz et al. 2018, Tanenbaum and Bos 2022]. Esse modelo permite concorrência intraprocesso, na medida em que múltiplos threads podem progredir de forma intercalada ao longo do tempo, e, quando suportado pelo hardware e pelo escalonamento do sistema operacional, pode possibilitar execução paralela em múltiplas unidades de processamento [Silberschatz et al. 2018, Tanenbaum and Bos 2022].

Em contraposição ao modelo de execução sequencial tradicional associado à arquitetura de von Neumann, no qual as instruções são concebidas como uma única sequência ordenada de operações, o multithreading introduz a existência de múltiplos fluxos de

controle que podem evoluir de forma concorrente dentro de um mesmo programa. Nesse contexto, a noção de ordem de execução deixa de ser global e passa a ser parcialmente definida, exigindo mecanismos adicionais para coordenar a interação entre os diferentes fluxos e garantir a consistência dos resultados produzidos [Silberschatz et al. 2018, Tanenbaum and Bos 2022].

A disponibilização do multithreading nas ferramentas de programação atuais ocorre, em geral, por meio de abstrações que se organizam segundo dois modelos principais: compartilhamento de memória ou troca de mensagens.

O modelo de implementação baseado em compartilhamento de memória é, possivelmente, o mais explorado no contexto acadêmico. Entre seus representantes mais difundidos destacam-se o Pthreads [Kleiman et al. 1996] e o OpenMP [Chandra et al. 2001]. Já o modelo de troca de mensagens consolidou-se em diferentes ambientes e linguagens, como Elixir [Gospodinov 2021] e Go [Cox-Buday 2017], sendo também suportado por linguagens multiparadigma como Rust [Team 2021].

Neste texto, o foco recai sobre mecanismos de programação em ambiente multithreading que vão além do uso convencional de seções críticas em implementações baseadas em memória compartilhada. As linguagens utilizadas como base são C++ [Williams 2019], Rust, Go e Elixir. Caso o leitor deseje maiores informações sobre essas e outras linguagens, inclusive cobrindo recursos relacionados ao compartilhamento de dados em memória, também podem ser consultado o material divulgado em [Cavalheiro et al. 2025, Cavalheiro 2009, Cavalheiro and Santos 2007].

O restante do texto está organizado como segue. Na Seção 1.2 são apresentados os mecanismos básicos de criação de threads nas linguagens consideradas, estabelecendo uma base comum para comparação. Na Seção 1.3 discute-se a coordenação de execução concorrente, iniciando pela espera explícita e avançando para a abstração de futures. Na Seção 1.4 aborda-se a comunicação por troca de mensagens, incluindo o uso de canais e sua realização nas linguagens. Por fim, a Seção 1.5 apresenta uma síntese das abordagens discutidas.

1.2. Criando Threads

Nesta seção são apresentados mecanismos básicos de criação de threads em C++, Rust, Go e Elixir. É importante observar que este texto não cobre todos os mecanismos oferecidos pelas linguagens, tampouco esgota a discussão sobre os mecanismos apresentados. No entanto, o que é apresentado é suficiente para compreensão dos demais conceitos que serão desenvolvidos.

Antes de avançar, é necessário estabelecer uma distinção conceitual. Termos como processo, thread e tarefa referem-se a unidades de execução, porém em níveis distintos de abstração. Um processo caracteriza-se como uma unidade de isolamento, com espaço de endereçamento próprio. Um thread é uma unidade de execução associada a um processo, compartilhando seus recursos. Já uma tarefa representa uma unidade lógica de trabalho, frequentemente gerenciada por um runtime e que pode ou não corresponder diretamente a um thread ou processo. As linguagens e ferramentas adotam nomenclaturas próprias para essas unidades, sendo necessário observar o significado atribuído em cada contexto.

1.2.1. C++

A biblioteca padrão de C++ oferece suporte a multithreading por meio da classe `std::thread`. Um thread é criado pela construção de um objeto dessa classe, ao qual se associa um objeto invocável, como uma função, um functor ou uma expressão lambda. A execução do thread inicia imediatamente após a construção do objeto.

O Código 1 ilustra essas três formas de criação. Cada thread executa de forma concorrente ao fluxo principal do programa.

```cpp
#include <iostream>
#include <thread>

void foo() {
  std::cout << "Executando foo\n";
}

class Functor {
public:
  void operator()() {
    std::cout << "Executando functor\n";
  }
};

int main() {
  Functor f;

  std::thread t1(foo);
  std::thread t2(f);
  std::thread t3([]() {
    std::cout << "Executando lambda\n";
  });

  t1.join();
  t2.join();
  t3.join();
}
```

Código 1: Criação de threads em C++

O controle do thread é realizado por meio do próprio objeto `std::thread`. A chamada ao método `join()` bloqueia o fluxo chamador até a finalização do thread. Alternativamente, o método `detach()` permite que o thread execute de forma independente. Caso o objeto `std::thread` seja destruído enquanto ainda estiver associado a um thread em execução, ocorre a chamada a `std::terminate()`, encerrando o programa.

Esse modelo associa explicitamente o ciclo de vida do thread ao objeto que o representa, exigindo que o programador gerencie sua finalização de forma adequada.

1.2.2. Rust

Rust oferece suporte a criação de threads nativos por meio da função `std::thread::spawn`. Essa função recebe como argumento uma expressão invocável, tipicamente uma closure, que será executada concorrentemente em um novo thread. A execução do thread inicia imediatamente após a chamada a `spawn`.

O Código 2 ilustra a criação de um thread em Rust. A closure utiliza a palavra-chave `move`, que transfere a posse das variáveis capturadas para o novo thread, garantindo segurança no acesso aos dados.

```rust
use std::thread;

fn main() {
  let x = 8;

  let handle = thread::spawn(move || {
    x * x
  });

  let r = handle.join().unwrap();
  println!("{}", r);
}
```

Código 2: Criação de threads em Rust

A função `spawn` retorna um objeto do tipo `JoinHandle`, que representa o thread criado. Esse objeto permite sincronização explícita por meio do método `join()`, que bloqueia o fluxo chamador até a conclusão do thread. O valor retornado pela closure é disponibilizado como resultado da chamada a `join()`, encapsulado em um `Result`.

Diferentemente de C++, Rust não oferece um mecanismo explícito de `detach`. Caso o `JoinHandle` não seja utilizado para sincronização, o thread continuará sua execução de forma independente, sem garantia de conclusão antes do término do programa. O modelo de ownership da linguagem impõe restrições ao compartilhamento de dados entre threads, exigindo transferência de posse ou o uso de abstrações seguras, o que contribui para evitar erros comuns em programação concorrente.

1.2.3. Go

Em Go, a unidade básica de execução concorrente é a *goroutine*. Uma goroutine é criada de forma simples, prefixando a chamada de uma função com a palavra-chave `go`. A execução da função é então agendada pelo runtime da linguagem, ocorrendo de forma concorrente ao fluxo principal do programa.

O Código 3 ilustra a criação de goroutines em Go. Duas tarefas são lançadas a partir da função `main`, uma utilizando função nomeada e outra utilizando uma função anônima.

```go
package main

import (
  "fmt"
  "sync"
)

func f(nome string, wg *sync.WaitGroup) {
  defer wg.Done()
  fmt.Println("Executando:", nome)
}

func main() {
  var wg sync.WaitGroup

  wg.Add(2)

  go f("goroutine 1", &wg)

  go func() {
    defer wg.Done()
    fmt.Println("Executando: goroutine 2")
  }()

  wg.Wait()
  fmt.Println("Função main retornou.")
}
```

Código 3: Criação de goroutines em Go

O runtime de Go é responsável por gerenciar a execução das goroutines, abstraindo detalhes de criação e escalonamento de threads do sistema operacional. Por padrão, o programa é encerrado assim que a função `main` retorna, mesmo que existam goroutines em execução. Por esse motivo, mecanismos de sincronização, como `sync.WaitGroup`, são utilizados para garantir que as goroutines concluam sua execução antes do término do programa.

1.2.4. Elixir

Em Elixir, a unidade básica de execução concorrente é o processo leve, gerenciado pela máquina virtual BEAM. Esses processos são independentes entre si, não compartilham memória e executam de forma concorrente.

A criação de um processo é realizada por meio da função `spawn`, que recebe como argumento uma função a ser executada. A execução ocorre de forma assíncrona em relação ao fluxo que realizou a chamada.

O Código 4 ilustra a criação de um processo em Elixir.

```
defmodule Demo do
  def saudacao do
    IO.puts("Olá do processo!")
  end
end

spawn(Demo, :saudacao, [])
```

Código 4: Criação de processos em Elixir

A função `spawn` retorna um identificador do tipo `pid`, que representa o processo criado. Diferentemente das linguagens apresentadas anteriormente, não há um mecanismo direto equivalente a `join` para aguardar a finalização de um processo. A coordenação entre processos é realizada por meio de troca de mensagens, tratada em seções posteriores.

O runtime da BEAM é responsável pelo gerenciamento e escalonamento dos processos, permitindo a criação de um grande número de unidades concorrentes com baixo custo.

1.3. Coordenação de Execução Concorrente

Nos mecanismos apresentados na Seção 1.2, a coordenação entre fluxos de execução ocorre de forma explícita, por meio de operações que suspendem o fluxo chamador até a conclusão de outro. Esse modelo, embora simples ainda que efetivo, exige controle manual da ordem de execução e não integra diretamente a produção e o consumo de resultados. Recursos de mais alto nível são oferecidos para que o programador tenha sua tarefa simplificada. Importante um lembrete ao leitor: seções críticas (mutex, semáforo) e execução condicional (variáveis de condição) preveem compartilhamento de memória e este texto não trata esse assunto.

1.3.1. Coordenação por Espera Explícita

A forma mais direta de realizar essa coordenação é por meio de operações de espera explícita. Nesses casos, o fluxo chamador é suspenso até que o fluxo concorrente finalize sua execução. Em C++ e Rust, essa coordenação é realizada por meio da chamada ao método `join()`, que bloqueia até o término do thread. Em Go, um comportamento equivalente pode ser obtido com o uso de `sync.WaitGroup`, que permite aguardar a conclusão de múltiplas goroutines. Já em Elixir, não há um mecanismo direto equivalente a `join`. A coordenação entre processos é realizada por meio de troca de mensagens, tipicamente com o uso de `send` e `receive`, permitindo que um processo aguarde explicitamente por uma mensagem que sinalize a conclusão de outra atividade.

O modelo de coordenação baseado em espera explícita, exemplificado pelo uso de `join()` em C++ e Rust e por `WaitGroup` em Go, apresenta uma característica comum: a sincronização é externa ao resultado da computação. O programador deve explicitamente controlar quando aguardar a conclusão de uma atividade concorrente, separando o fluxo de execução da obtenção dos resultados produzidos.

Embora simples, essa abordagem impõe uma estrutura de controle rígida, na qual a ordem de execução deve ser gerenciada manualmente. Essa limitação motiva a introdução de abstrações que integrem a produção e o consumo de resultados em um único mecanismo.

1.3.2. Futures como Abstração

Uma alternativa ao modelo de coordenação por espera explícita consiste em representar o resultado de uma computação concorrente como um valor que será disponibilizado no futuro. Essa ideia é formalizada pela abstração de *future*.

Um *future* representa o resultado de uma computação que pode ainda não ter sido concluída. Em vez de suspender explicitamente o fluxo de execução para aguardar a finalização de uma tarefa, o programa passa a manipular esse resultado de forma indireta, podendo acessá-lo quando necessário.

Essa abordagem integra a produção e o consumo de resultados em um único mecanismo. A sincronização deixa de ser uma operação externa e passa a estar associada ao próprio valor, que encapsula tanto o resultado quanto o estado da computação. Dessa forma, a coordenação entre fluxos de execução torna-se mais declarativa, reduzindo a necessidade de controle manual da ordem de execução.

1.3.3. Futures nas Linguagens

A abstração de *futures* é disponibilizada de forma explícita em C++ e Rust, ainda que com diferenças na forma de uso e no modelo de execução subjacente.

Em C++, a biblioteca padrão oferece os tipos `std::future` e `std::promise`, que permitem representar e manipular resultados produzidos por tarefas concorrentes. Um `future` pode ser obtido, por exemplo, a partir da função `std::async`, que executa uma função de forma concorrente e retorna imediatamente um objeto associado ao resultado futuro da computação. O valor pode ser obtido posteriormente por meio do método `get()`, que bloqueia até que o resultado esteja disponível.

Embora Go e Elixir não ofereçam um tipo explícito de *future* em suas bibliotecas padrão, a ideia pode ser representada por meio de seus mecanismos nativos de comunicação. Em Go, canais podem ser utilizados para transportar o resultado de uma goroutine, permitindo que o consumidor aguarde o valor quando necessário. Em Elixir, o mesmo papel é desempenhado pela troca de mensagens entre processos, em que o recebimento de uma mensagem pode ser interpretado como a obtenção de um resultado futuro.

Dessa forma, ainda que a abstração de *futures* não seja uniformemente exposta em todas as linguagens, sua ideia fundamental, a associação entre resultado e sincronização, pode ser observada em diferentes modelos de concorrência.

```cpp
#include <future>
#include <iostream>

int f() {
  return 313;
}

int main() {
  std::future<int> fut = std::async(f);
  int r = fut.get();
  std::cout << r << std::endl;
}
```

Código 5: Uso de `std::future` em C++

```rust
async fn f() -> i32 {
  313
}

#[tokio::main]
async fn main() {
  let r = f().await;
  println!("{}", r);
}
```

Código 6: Uso de `async/await` em Rust

1.4. Comunicação por Troca de Mensagens

Nesta seção, o foco passa do modelo de coordenação baseado na espera por resultados para um modelo no qual a interação entre fluxos de execução ocorre por meio da troca de dados. Em vez de apenas aguardar a disponibilização de um resultado, considera-se a comunicação explícita entre unidades de execução. Esse modelo elimina a necessidade de compartilhamento direto de memória e estabelece a comunicação como mecanismo central de coordenação em sistemas concorrentes.

1.4.1. Comunicação por Mensagens

Uma alternativa à coordenação baseada em compartilhamento de memória é a comunicação por troca de mensagens, na qual unidades de execução interagem por meio do envio e recebimento explícito de dados. Nesse modelo, em vez de múltiplos threads acessarem diretamente uma região compartilhada da memória, os dados são transferidos entre as unidades de execução por meio de mecanismos de comunicação, evitando o acesso concorrente direto a estruturas compartilhadas.

Esse modelo favorece um estilo de programação no qual a sincronização ocorre como parte do próprio processo de comunicação. Ao enviar ou receber uma mensagem, estabelece-se implicitamente uma relação de coordenação entre as unidades envolvidas, o que reduz os riscos de condições de corrida e simplifica o raciocínio sobre o comportamento concorrente do programa.

Em linguagens como Go, a comunicação é realizada por meio de canais, que permitem a troca segura de dados entre goroutines, promovendo tanto a transmissão de valores quanto a sincronização entre os participantes da comunicação. Operações de envio e recebimento são utilizadas para transferir dados, podendo inclusive bloquear a execução até que a contraparte esteja pronta, estabelecendo um ponto de coordenação entre as tarefas concorrentes.

Já em Elixir, a comunicação ocorre por meio de troca de mensagens entre processos leves, utilizando primitivas como `send` e `receive`. Nesse modelo, processos são criados de forma independente e interagem exclusivamente por mensagens, sendo comum a construção de soluções nas quais múltiplos processos cooperam por meio desse mecanismo para produzir um resultado final.

Assim, a comunicação por mensagens estabelece um modelo no qual a interação entre unidades de execução é explícita e estruturada, deslocando o foco do controle de acesso à memória para a troca organizada de informações entre componentes concorrentes.

1.4.2. Canais como Abstração de Comunicação

Canais constituem uma abstração para comunicação entre unidades de execução concorrentes, permitindo a troca de dados de forma segura sem a necessidade de acesso direto a memória compartilhada. Por meio dessa abstração, valores são explicitamente enviados e recebidos, estabelecendo uma forma estruturada de interação entre tarefas.

Além de viabilizar a transmissão de dados, os canais também promovem sincronização entre as unidades de execução envolvidas. Em particular, operações de envio e recebimento podem impor bloqueio até que a contraparte esteja pronta para prosseguir, fazendo com que o próprio mecanismo de comunicação atue como ponto de coordenação entre tarefas concorrentes.

Os canais podem ser definidos com ou sem buffer. Em canais sem buffer, o envio de um valor exige que haja simultaneamente uma operação de recebimento correspondente, estabelecendo uma sincronização direta entre emissor e receptor. Já canais com buffer permitem que um número limitado de valores seja armazenado temporariamente, possibilitando certo desacoplamento entre produção e consumo de dados.

Outro aspecto relevante é a possibilidade de especialização do uso dos canais, restringindo sua direção para envio ou recebimento. Essa característica permite explicitar o papel das diferentes partes do programa na comunicação, contribuindo para uma organização mais clara das interações entre tarefas.

Dessa forma, os canais estabelecem uma forma de comunicação tipada e estruturada, na qual a troca de dados e a sincronização são tratadas de maneira integrada, favorecendo a construção de programas concorrentes que evitam os problemas associados ao

uso de memória compartilhada.

1.4.3. Comunicação nas Linguagens

A comunicação por troca de mensagens manifesta-se de forma explícita nas linguagens consideradas, sendo possível observar diferentes formas de estruturar essa interação entre unidades de execução.

Em Go, a comunicação ocorre por meio de canais, que permitem a troca direta de valores entre goroutines. No exemplo do cálculo da sequência de Fibonacci (Código 7), uma goroutine é responsável por gerar os valores e enviá-los por um canal, enquanto outra goroutine realiza o consumo desses valores. O envio é realizado com o operador $<-$ aplicado ao canal, enquanto o recebimento utiliza o mesmo operador em sentido inverso. Esse mecanismo estabelece uma relação direta entre produtor e consumidor, na qual a comunicação também atua como forma de sincronização, uma vez que o envio e o recebimento podem bloquear até que ambas as partes estejam prontas para prosseguir.

```go
func fib(n, ch chan <- uint64) {
  if n <= 1 {
    ch <- uint64(n)
    return
  }
  ch1 := make(chan uint64)
  ch2 := make(chan uint64)
  go fib(n-1, ch1)
  go fib(n-2, ch2)
  res1 := <-ch1
  res2 := <-ch2
  ch <- res1 + res2
}
```

```go
func main() {
  n
  resultCh := make(chan uint64)
  go fib(n, resultCh)
  result := <-resultCh
}
```

Código 7: Fibonacci em Go com sincronização via canais.

Já em Elixir, a comunicação é realizada por meio do envio e recebimento explícito de mensagens entre processos (Código 8). No exemplo do cálculo de Fibonacci, um processo envia mensagens contendo resultados intermediários, enquanto outro processo utiliza a construção `receive` para aguardar e tratar essas mensagens. Cada processo possui sua própria fila de mensagens, e a interação ocorre exclusivamente por meio desse mecanismo, sem qualquer compartilhamento direto de memória.

```elixir
defmodule Fib do
  def compute(n) when n <= 1, do: n
  def compute(n) do
    parent = self()
    spawn(fn ->
      send(parent, {:fib1, compute(n - 1)})
    end)
    spawn(fn ->
      send(parent, {:fib2, compute(n - 2)})
    end)
    receive do
      {:fib1, res1} ->
        receive do
          {:fib2, res2} ->
            res1 + res2
        end
    end
  end
end
defmodule Main do
  def main(_) do
    n = 10
    result = Fib.compute(n)
  end
end
Main.main([])
```

Código 8: Fibonacci em Elixir com troca de mensagens explícitas.

Esse modelo pode ser observado também no padrão produtor-consumidor imple-

mentado em Elixir, no qual um processo produtor envia mensagens representando dados produzidos, enquanto um processo consumidor as recebe e processa. A coordenação entre esses processos ocorre inteiramente por meio da troca de mensagens, evidenciando um modelo no qual a comunicação é o elemento central de organização do fluxo concorrente.

Assim, tanto em Go quanto em Elixir, a comunicação por mensagens não apenas viabiliza a troca de dados, mas também estrutura a coordenação entre unidades de execução, integrando comunicação e sincronização em um único mecanismo.

1.4.4. Modelos de Concorrência Baseados em Comunicação

Os mecanismos apresentados nas seções anteriores evidenciam uma mudança de perspectiva na forma de estruturar programas concorrentes. Em contraste com o modelo centrado em threads, no qual a coordenação é realizada por meio do controle explícito da execução, as abordagens baseadas em comunicação deslocam o foco para a interação entre unidades de execução.

Nesse contexto, a concorrência deixa de ser organizada em torno da gestão de threads e passa a ser estruturada a partir do fluxo de dados entre entidades independentes. A comunicação não atua apenas como meio de troca de informações, mas também como mecanismo de sincronização, uma vez que a disponibilidade dos dados passa a determinar o progresso da execução.

Abstrações como futures e canais materializam essa mudança. Em vez de bloquear explicitamente a execução até o término de uma atividade, o programa passa a reagir à disponibilidade de resultados ou mensagens, permitindo uma composição mais direta entre produção e consumo de dados.

Como consequência, o controle da execução torna-se menos centralizado, e a coordenação emerge da própria estrutura de comunicação do programa. Esse deslocamento reduz a necessidade de gerenciamento explícito de estados compartilhados e favorece a construção de programas concorrentes mais modulares.

1.5. Discussão Final

Neste texto foram apresentadas diferentes formas de estruturar programas concorrentes, variando desde modelos centrados no controle explícito de threads até abordagens baseadas na comunicação entre unidades de execução. Mecanismos baseados em compartilhamento de memória não foram tratados.

Para auxiliar uma visão do conteúdo, os principais conceitos tratados estão sumarizados na Tabela 1.1. A apresentação desta tabela busca sintetiza essas diferenças entre C++, Rust, Go e Elixir no tema.

Tabela 1.1. Comparação entre modelos de concorrência nas linguagens consideradas

Linguagem	Unidade	Coordenação	Comunicação	Threads explícitas	Futures
C++	Thread	Espera (join)	Secundária	Sim	Sim
Rust	Thread/Tarefa	Espera e futures	Secundária	Sim	Sim
Go	Goroutine	Comunicação	Central	Não	Implícito
Elixir	Processo	Comunicação	Central	Não	Implícito

A tabela apresentada evidencia duas abordagens distintas para a construção de programas concorrentes. Em C++ e, pelo menos na sua forma nativa, em Rust, a estruturação da concorrência permanece centrada no thread, com coordenação realizada por mecanismos de espera explícita ou por abstrações que ainda mantêm relação direta com o fluxo de execução. Nesses casos, a comunicação desempenha papel secundário, sendo utilizada como complemento ao controle do programa.

Por outro lado, Go e Elixir adotam uma abordagem na qual a comunicação assume papel central. Nessas linguagens, as unidades de execução são tratadas como entidades independentes, e a coordenação ocorre por meio da troca de mensagens ou do uso de canais. Como consequência, a sincronização deixa de ser uma operação explicitamente controlada sobre threads e passa a emergir da própria interação entre os componentes do programa. Esse deslocamento caracteriza uma mudança de modelo, na qual o foco deixa de estar na gestão de execução e passa a privilegiar o fluxo de dados.

Referências

[Cavalheiro 2009] Cavalheiro, G. G. H. (2009). Programação com pthreads. In Mattos, J. C. B., Da Rosa Junior, L. S., and Pilla, M. L., editors, *Desafios e Avanços em Computação: O Estado da Arte*, pages 137–151. Editora e Gráfica Universitária - PREC UFPel, Pelotas.

[Cavalheiro et al. 2025] Cavalheiro, G. G. H., Baldassin, A., and Bois, A. R. D. (2025). Programação multithread: Modelos e abstrações em linguagens contemporâneas. In Musse, S. R. and dos Santos, A. P., editors, *44a Jornada de Atualização em Informática (JAI 2025)*. Sociedade Brasileira de Computação (SBC), Porto Alegre.

[Cavalheiro and Santos 2007] Cavalheiro, G. G. H. and Santos, R. R. (2007). Multiprogramação leve em arquiteturas multi-core. In Kowaltowski, T. and Breitman, K. K., editors, *Atualizações em Informática 2007*, pages 327–379. PUC-Rio, Rio de Janeiro.

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

[Cox-Buday 2017] Cox-Buday, K. (2017). *Concurrency in Go: Tools and Techniques for Developers*. O'Reilly Media, Inc., 1st edition.

[Gospodinov 2021] Gospodinov, S. (2021). *Concurrent Data Processing in Elixir: Fast, Resilient Applications with OTP, GenState, Flow, and Broadway*. The Pragmatic Bookshelf, Raleigh, North Carolina.

[Kleiman et al. 1996] Kleiman, S., Shah, D., and Smaalders, B. (1996). *Programming with Threads*. Sun Soft Press ; Prentice Hall, Mountain View, Calif. ; Upper Saddle River, NJ.

[Silberschatz et al. 2018] Silberschatz, A., Galvin, P. B., and Gagne, G. (2018). *Operating System Concepts*. Wiley, 10 edition.

[Tanenbaum and Bos 2022] Tanenbaum, A. S. and Bos, H. (2022). *Modern Operating Systems*. Prentice Hall, 5 edition.

[Team 2021] Team, T. R. (2021). *The Rust Programming Language*. Rust Core Team. Official language guide.

[Williams 2019] Williams, A. (2019). *C++ Concurrency in Action, Second Edition*. Manning, 2 edition.

Capítulo

2

Análise de Desempenho de Sistemas Computacionais: Fundamentos e Metodologia Científica para HPC

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

Resumo

A análise de desempenho é fundamental para o desenvolvimento e otimização de sistemas computacionais, especialmente em aplicações de Alto Desempenho (HPC), porém a área é frequentemente negligenciada na formação acadêmica, resultando em estudos com metodologia inadequada, resultados não reprodutíveis e conclusões equivocadas. Este minicurso tem como objetivo geral capacitar os participantes a planejar, executar e reportar experimentos de avaliação de desempenho de sistemas computacionais de forma metodologicamente rigorosa e reprodutível, combinando os fundamentos consolidados de livros clássico de análise de desempenho com princípios modernos de metodologia científica e reprodutibilidade na área de HPC.

2.1. Introdução

Um pilar do método científico é o uso de experimentos para validar ou refutar hipóteses e teorias. Para ser confiável, um experimento deve ser reprodutível, de forma que outros pesquisadores possam refazer as observações de maneira independente. Na computação de alto desempenho (HPC), essa exigência se traduz em um conjunto de práticas metodológicas, desde o controle experimental, a escolha das métricas corretas, até a apresentação estatística adequada de resultados. A análise de desempenho constitui, portanto, a base do avanço científico em HPC. Sem ela, não é possível comparar sistemas, validar otimizações ou garantir que os resultados relatados em uma publicação possam ser compreendidos e avaliados por outros pesquisadores. Esse papel central, no entanto, convive com um problema estrutural: a análise de desempenho é frequentemente negligenciada na formação acadêmica em computação. Por exemplo, disciplinas de probabilidade e estatística, quando presentes, raramente abordam casos práticos de sistemas computacionais HPC como distribuições não-normais, métricas derivadas como aceleração e eficiência, ou até a clássica questão de quantas repetições são necessárias para defender uma conclusão. O resultado são metodologias inadequadas que culminam em conclusões equivocadas.

A Lacuna entre as Referências Clássicas e a Prática HPC. Existem referências fortes sobre avaliação de desempenho de sistemas computacionais, como os livros de Jain [Jain 1991], Lilja [Lilja 2000] e Le Boudec [Le Boudec 2011]. Estes oferecem bases metodológicas sólidas. Nenhum deles, no entanto, foi escrito tendo considerando especificidades de HPC. Não abordam a variabilidade introduzida por aceleradores heterogêneos nem o ruído de sistema em máquinas grandes. Não abordam as dificuldades de reprodutibilidade em supercomputadores de acesso restrito. Não discutem as armadilhas de métricas paralelas como speedup, weak e strong scaling, ou eficiência de comunicação MPI. O movimento inverso é igualmente problemático. A literatura de HPC é rica em contribuições técnicas sobre os mais variados assuntos, mas raramente trata a avaliação de desempenho com o rigor que os clássicos ensinam. Métricas são escolhidas sem justificativa, experimentos são reportados sem medidas de variabilidade, e comparações são feitas sem qualquer teste estatístico [Hoefler and Belli 2015]. Para contextualizar essa interseção, percorremos quatro aspectos complementares do estado da arte. O diagnóstico empírico do problema [Hoefler and Belli 2015]. A resposta institucional como a SC Reproducibility Initiative [SC Repr. Initiative 2015]. O aprofundamento conceitual para melhores práticas [Bruel et al. 2023]. A síntese que emerge dessa cronologia e que justifica a existência e o design deste minicurso.

Diagnóstico: o Estado da Prática de Análise de Desempenho em HPC é Deficiente. Um diagnóstico sistemático do problema foi realizado por [Hoefler and Belli 2015]. A partir de uma amostra estratificada de 120 artigos publicados no ACM HPDC, SC e ACM PPoPP entre 2011 e 2014, os autores documentaram que o ambiente de software é negligenciado sistematicamente, que nenhum dos 95 artigos aplicáveis usa argumentos estatísticos para comparar resultados, que apenas 2 reportam intervalos de confiança, e que apenas 1 usa corretamente a média harmônica para sumarizar taxas. Como resposta a esse diagnóstico, os autores propõem o conceito de interpretabilidade como meta realista para HPC, tendo em vista que a reprodutibilidade exata é frequentemente impossível em supercomputadores de acesso restrito, e codificam 12 regras que constituem o conjunto mínimo de boas práticas para benchmarking interpretável em computação paralela, as quais serão retomadas ao longo deste capítulo.

Resposta Institucional: Iniciativas de Reprodutibilidade. A comunidade HPC respondeu ao diagnóstico com iniciativas institucionais concretas, sendo a mais relevante a SC Reproducibility Initiative [SC Repr. Initiative 2015], cujo apêndice "Artifact Description" tornou-se obrigatório em 2019 e que hoje concede badges ACM nas categorias *Available*, *Functional* e *Reproduced*. Uma survey conduzida por [Plale et al. 2021] revelou que 90% dos respondentes estão cientes dos problemas de reprodutibilidade e que 77% mudaram sua forma de interpretar resultados publicados, evidência de uma mudança cultural real, ainda que limitada ao plano da documentação e da transparência, sem endereçar os problemas metodológicos de análise de dados identificados por [Hoefler and Belli 2015].

Persistência do Problema: Desempenho como Distribuição. Bruel [Bruel et al. 2023] propõem uma reorientação fundamental na forma como o desempenho é concebido. Ao invés de ver desempenho como número, vê-lo como distribuição. Executando benchmarks da suíte Rodinia [Che et al. 2009], os autores demonstram empiricamente quatro armadilhas recorrentes: uso de resumos inadequados para distribuições multimodais ou de cauda longa; cálculo de intervalos de confiança sob pressuposição incorreta de norma-

lidade; descarte de *outliers* informativos, como os 13% de medições do benchmark needle que excedem 3 segundos; e adoção de tamanho fixo de amostra, sendo que o número necessário varia por uma ordem de grandeza entre benchmarks da própria suíte. A proposta central é que o alvo da avaliação seja a distribuição observacional subjacente, não uma estatística dela, o que implica o uso de testes de comparação de distribuições, regras de parada adaptativas e familiaridade com ajuste de modelos, constituindo potencialmente uma atualização natural das 12 regras de [Hoefler and Belli 2015] para o contexto dos sistemas HPC modernos.

Nó do Problema: uma Questão de Formação, não de Política. Os três aspectos anteriores convergem para uma conclusão desconfortável. As iniciativas institucionais melhoraram a transparência e a documentação do ambiente experimental, o que representa um avanço real e necessário. Mas os problemas mais profundos identificados por Hoefler [Hoefler and Belli 2015] e aprofundados por Bruel [Bruel et al. 2023] permanecem largamente sem solução. A razão é estrutural: esses são problemas de formação e não de política de conferência. Nenhum *badge* corrige um pesquisador que não distingue média aritmética de harmônica. Nenhum apêndice obrigatório ensina a verificar normalidade antes de calcular um intervalo de confiança. A consciência do problema cresceu, tendo em vista que 90% dos pesquisadores que participam do SC declaram estar cientes da reprodutibilidade [Plale et al. 2021], mas consciência não equivale a prática. Saber que se deve reportar um intervalo de confiança não basta se não se sabe construí-lo corretamente para dados com distribuição não-normal, por exemplo.

Objetivos e Organização do Capítulo. Este minicurso está na interseção entre o rigor metodológico dos livros clássicos de análise de desempenho e os desafios concretos do ambiente HPC. O objetivo geral é capacitar os participantes a planejar, executar e reportar experimentos de avaliação de desempenho de forma metodologicamente rigorosa, reprodutível e interpretável. A Seção 2.2 apresenta os fundamentos conceituais da avaliação de desempenho, cobrindo o processo, as métricas, as técnicas disponíveis e os modelos analíticos de desempenho. A Seção 2.3 trata da análise estatística para avaliação de desempenho. A Seção 2.4 aborda o planejamento de experimentos. A Seção 2.5 trata dos erros comuns e das melhores práticas em HPC. A Seção 2.6 apresenta a caracterização de carga de trabalho. A Seção 2.7 apresenta ferramentas de monitoramento e rastreamento. A Seção 2.8 aborda a apresentação visual de resultados. Enfim, a Seção 2.9 conclui com um checklist prático e direções futuras.

2.2. Fundamentos Conceituais da Avaliação de Desempenho

Vamos primeiro estabelecer os conceitos que fundamentam qualquer avaliação de desempenho, independentemente do sistema sob análise.

O objetivo da avaliação de desempenho deve ser determinado. Podemos, por exemplo, escolher um dentre estes quatro [Jain 1991]. 1/ A comparação entre alternativas, como escolher entre dois algoritmos de comunicação ou entre duas configurações de hardware. 2/ A caracterização de um sistema existente, buscando entender seu comportamento sob diferentes condições de carga. 3/ O dimensionamento, ou *capacity planning*, que consiste em prever o comportamento do sistema sob cargas futuras antes que elas ocorram. 4/ A otimização, voltada para identificar e eliminar gargalos de desempenho. O objetivo esco-

lhido determina tudo o que vem depois: a escolha das métricas, a técnica de avaliação, a carga de trabalho, o critério de sucesso e a forma de apresentação dos resultados. Ao confundir objetivos, pode-se produzir conclusões tecnicamente corretas e conceitualmente equivocadas ao mesmo tempo.

Toda avaliação de desempenho é uma modelagem. Uma distinção conceitual fundamental, enfatizada por [Le Boudec 2011], é que toda avaliação de desempenho é necessariamente uma modelagem. Mesmo quando se realiza uma medição direta sobre o sistema real, o que se observa não é o sistema em si, mas uma projeção do sistema através das métricas escolhidas. O tempo de execução de um programa, por exemplo, é uma abstração que agrega em um único número todo o comportamento do processador, do sistema de memória, do sistema operacional e da aplicação ao longo de um intervalo de tempo. Essa cadeia de abstrações pode ser representada como uma progressão do sistema real para o modelo do sistema, do modelo para a métrica e da métrica para o número que aparece na tabela de resultados. Cada elo dessa cadeia é uma fonte potencial de erro e de incompreensão. A consequência prática é que os resultados de uma avaliação têm validade apenas dentro do escopo do modelo adotado. Qualquer extrapolação para condições não cobertas pelo modelo requer justificativa explícita. Afirmar que um sistema é 30% mais rápido sem especificar em qual carga, em qual configuração e com qual métrica é uma afirmação sem conteúdo científico.

O processo de avaliação como sequência metodológica. Jain [Jain 1991] insiste na descrição do processo de avaliação de desempenho como uma sequência de etapas: definição do sistema e escopo, identificação dos objetivos, listagem de métricas e parâmetros, escolha da técnica de avaliação, planejamento dos experimentos, coleta de dados, análise e interpretação, e apresentação dos resultados. A sequência não é rígida, e iterações são esperadas, especialmente entre a fase de análise e o planejamento de novos experimentos. O desvio mais perigoso é definir métricas após a coleta de dados, prática conhecida como HARKing (*Hypothesizing After Results are Known*) [Kerr 1998], que produz conclusões que parecem confirmatórias mas são de fato exploratórias, comprometendo a validade científica dos resultados.

Técnicas de Avaliação: Medição, Simulação e Modelagem Analítica. Jain [Jain 1991] organiza as técnicas de avaliação de desempenho em três categorias. 1/ A medição direta opera sobre o sistema real ou um protótipo e produz resultados de alta fidelidade, mas está sujeita a todos os fatores de variabilidade de sistemas reais e requer que o sistema exista e esteja acessível. 2/ A modelagem analítica, baseada em teoria de filas e redes de filas [Le Boudec 2011], permite análise em estágios iniciais do projeto quando o sistema ainda não existe, e produz resultados em forma fechada que facilitam a compreensão dos trade-offs, ao custo de abstrações que podem não capturar o comportamento real. 3/ A simulação enfim ocupa uma posição intermediária: permite modelar sistemas complexos com mais fidelidade que modelos analíticos, mas requer validação cuidadosa contra medições reais e pode consumir recursos computacionais significativos. A escolha entre essas abordagens depende de quatro critérios: custo, que inclui tempo e recursos necessários; precisão requerida pelas decisões a serem tomadas; generalidade, ou seja, a capacidade de responder a variações nos parâmetros do sistema; e o estágio do projeto, pois modelagem analítica e simulação são as únicas opções quando o sistema ainda não existe. Na prática, as abordagens são complementares: medições calibram modelos analíticos, modelos

analíticos guiam o design dos experimentos, e simulação permite explorar o espaço de parâmetros além do que seria viável medir diretamente. Tipicamente se empregam dois métodos para confirmar comportamentos observados.

Os três elementos irredutíveis de qualquer avaliação. Jain [Jain 1991] identifica três elementos que estão presentes em qualquer avaliação de desempenho e cuja definição precisa é indispensável antes que qualquer medição seja realizada. O primeiro é o sistema sob avaliação, que deve ter seus limites claramente estabelecidos: o que é considerado interno ao sistema e o que é tratado como estímulo externo. O segundo é a carga de trabalho, ou workload, que é o conjunto de estímulos aplicados ao sistema durante a avaliação. O terceiro é a métrica, que é o aspecto do comportamento do sistema que se deseja quantificar. Vamos abordar primeiro o papel da carga de trabalho para depois delinear as métricas.

O papel da carga de trabalho na validade da avaliação. Um resultado de desempenho é sempre condicional à carga usada na avaliação. Essa observação, aparentemente simples, tem consequências metodológicas profundas. Jain [Jain 1991] dedica um capítulo inteiro à caracterização de carga de trabalho precisamente porque a escolha do workload é uma das decisões mais determinantes para a validade e a generalidade das conclusões. Dois critérios essenciais devem orientar essa escolha. O primeiro é a representatividade: a carga deve refletir o uso real do sistema de forma que os resultados sejam relevantes para o contexto de interesse. O segundo é a controlabilidade: a carga deve ser reprodutível e parametrizável de forma que o experimento possa ser repetido e variações possam ser introduzidas de maneira sistemática. Esses dois critérios estão em tensão permanente. Benchmarks sintéticos são facilmente controláveis, mas podem não capturar o comportamento real das aplicações de interesse. Rastros de execuções reais são representativos por definição, mas são difíceis de reproduzir e não permitem parametrização simples. A escolha entre essas alternativas, ou a combinação de ambas, deve ser justificada explicitamente em função do objetivo da avaliação. A Seção 2.6 aprofunda a caracterização de cargas de trabalho em HPC, incluindo benchmarks consagrados, mini-aplicações e o uso de rastros reais.

A hierarquia de métricas. As métricas de desempenho podem ser organizadas em três níveis [Jain 1991] [Le Boudec 2011]. 1/ As métricas de nível de sistema, como tempo de resposta, vazão e utilização, são observáveis externamente e independem dos detalhes de implementação, sendo portanto as mais adequadas para comparações entre sistemas distintos. 2/ As métricas de nível de componente, como taxa de *cache miss*, instruções por ciclo (IPC) e taxa de transferência de memória, requerem acesso interno ao sistema e são mais adequadas para diagnóstico e otimização. 3/ As métricas derivadas, como aceleração, eficiência e escalabilidade, são razões entre métricas de nível de sistema e sua interpretação depende criticamente de como a baseline é definida [Hoefler and Belli 2015]. A aceleração linear ideal segue da Lei de Amdahl, que estabelece o limite imposto pela fração serial da aplicação: se uma fração f do código não pode ser paralelizada, a aceleração máxima é limitada a $1/f$ independentemente do número de processos. A Lei de Gustafson reorienta essa análise para o regime de *weak scaling*, onde o problema cresce com o número de processos, e é mais adequada para avaliar aplicações científicas de larga escala. As implicações metodológicas de strong e weak scaling para o design de experimentos são discutidas na Seção 2.4. Para sistemas modernos, métricas específicas

incluem FLOPS efetivos, largura de banda de memória, eficiência de comunicação MPI e energia por operação (FLOPS/Watt). Uma boa métrica, segundo [Le Boudec 2011], deve satisfazer três propriedades. A linearidade garante que dobrar o desempenho real dobre o valor da métrica. A monotonicidade garante que melhorias no sistema se traduzam em melhorias na métrica, sem ambiguidade de direção. A interpretabilidade direta garante que o valor da métrica tenha significado concreto sem necessidade de transformações adicionais.

Métricas de balanceamento de carga. O Load Imbalance Factor (LIF) definido como a razão entre a carga máxima e a carga média entre processos, quantifica o potencial de melhoria disponível: LIF igual a um indica balanço perfeito, e LIF maior que um indica que a execução é limitada pelo processo mais sobrecarregado. Olga [Pearce et al. 2012] mostra, no entanto, que métricas simples como o LIF capturam a magnitude do desbalanceamento mas não sua causa, pois nenhuma métrica disponível até então considerava simultaneamente a carga computacional por processo e a estrutura do domínio simulado. Métricas complementares incluem o tempo de idle por rank MPI, o desvio padrão do tempo de execução entre ranks e a eficiência de balanceamento, definida como $E_{lb} = T_{avg}/T_{max}$. Uma armadilha comum é reportar apenas o tempo do rank mais lento como representativo da execução. Em sistemas heterogêneos com aceleradores, o LIF deve considerar as capacidades distintas de CPU e GPU, e o desbalanceamento espacial entre nós e o desbalanceamento temporal ao longo da execução são dimensões independentes que requerem métricas separadas.

Métricas de mascaramento de comunicação. Operações não-bloqueantes MPI, incluindo `MPI_Isend`, `MPI_Irecv` e as coletivas não-bloqueantes introduzidas no padrão MPI-3, permitem em princípio sobrepor comunicação com computação. Se esse mascaramento efetivamente ocorre depende do mecanismo de progressão da biblioteca MPI e do hardware de rede. [Denis et al. 2022] propõem duas métricas para avaliar coletivas não-bloqueantes: o *overlap ratio*, que mede a fração do tempo de comunicação efetivamente sobreposto com computação, e a *interference metric*, que quantifica o impacto do mecanismo de progressão sobre a computação, ou seja, os ciclos de CPU consumidos pelo progresso da comunicação em background. A distinção é essencial: não basta medir se há overlap; é necessário medir também o custo que o mecanismo de progressão impõe à computação, pois um overlap aparente pode degradar o desempenho total se a interferência for alta. Em sistemas MPI+GPU, o custo de comunicação inclui lançamento de kernel pelo CPU, sincronização CPU-GPU, transferência PCIe ou NVLink e comunicação de rede; a decomposição desses componentes é necessária para identificar o gargalo real. A armadilha comum é medir apenas o tempo total de uma iteração sem decompor as contribuições de computação, comunicação e sincronização, o que impede avaliar o efeito de qualquer otimização. Com os fundamentos conceituais e os modelos analíticos de desempenho estabelecidos, a próxima seção desenvolve as ferramentas estatísticas necessárias para extrair conclusões rigorosas das medições.

2.3. Análise Estatística para Avaliação de Desempenho

A análise estatística não é uma etapa opcional que vem depois dos experimentos: ela determina como os experimentos devem ser planejados, quantas amostras coletar e como as conclusões podem ser enunciadas com rigor.

Medidas de tendência central e de dispersão. A escolha da medida de tendência central depende da natureza do dado [Jain 1991]. A média aritmética é adequada para custos como tempo e energia; a média harmônica é a medida correta para taxas como flop/s e largura de banda, pois preserva a relação inversa entre tempo e desempenho; a média geométrica é adequada quando se deseja agregar razões adimensionais. Para distribuições assimétricas, frequentes em dados de desempenho HPC, a mediana poder ser mais representativa que a média aritmética. As medidas de dispersão são tão importantes quanto as de tendência central. O desvio padrão e a variância descrevem a dispersão absoluta; o coeficiente de variação, razão entre desvio padrão e média, permite comparar a consistência de desempenho entre configurações com magnitudes diferentes. Para aplicações sensíveis a eventos raros, os percentis altos (P95, P99) capturam comportamentos de cauda que a média e a mediana ignoram completamente [Hoefler and Belli 2015].

Distribuições de dados HPC e o mito das 30 amostras. As distribuições de tempo de execução e latência em sistemas HPC são raramente normais. Assimetria à direita, caudas pesadas, multimodalidade e picos estreitos são padrões comuns [Bruel et al. 2023]. Para dados com distribuição log-normal, a transformação logarítmica e o uso da média geométrica são mais adequados que a média aritmética sobre os valores originais. A recomendação informal de que 30 a 40 amostras são suficientes para invocar o teorema central do limite não tem fundamento para esses dados. Bruel [Bruel et al. 2023], por exemplo, demonstra empiricamente que o número de amostras para caracterizar adequadamente a distribuição varia por uma ordem de grandeza entre benchmarks da mesma suíte.

Intervalos de confiança (IC). Para dados com distribuição normal ou aproximadamente normal, o intervalo de confiança para a média é construído com a distribuição t de Student com n-1 graus de liberdade [Le Boudec 2011]. Para dados não-paramétricos, o IC para a mediana é construído a partir de ranks ordenados, sem pressupor qualquer forma distribucional; nesses casos o IC pode ser assimétrico, refletindo a assimetria dos dados subjacentes. Uma regra prática útil é continuar coletando amostras até que o IC de 99% esteja dentro de 5% da média ou mediana estimada. Quando outliers forem removidos, o método de Tukey deve ser aplicado e o número de observações removidas deve ser sempre reportado: omitir essa informação é uma forma de manipulação silenciosa dos resultados.

Comparação estatística de configurações. A não-sobreposição de ICs é uma condição suficiente para afirmar diferença entre duas configurações, mas não é uma condição necessária: dois ICs podem se sobrepor e a diferença ainda ser estatisticamente significativa [Hoefler and Belli 2015]. Para comparação formal de médias sob normalidade, a ANOVA e o teste F fornecem o arcabouço correto. Para dados não-normais, o teste de Kruskal-Wallis compara medianas sem pressupor forma distribucional. Nenhum dos 95 artigos analisados por [Hoefler and Belli 2015] usou qualquer argumento estatístico formal para comparar resultados entre configurações. Além da significância estatística, o tamanho do efeito, expresso como diferença relativa ao desvio padrão combinado, deve ser reportado: uma diferença pode ser estatisticamente significativa e praticamente irrelevante ao mesmo tempo. Para análises de comportamento de cauda, a regressão de quantis permite modelar o efeito de um fator em percentis arbitrários; sua importância prática é que o sistema com melhor latência mediana pode ter comportamento de cauda pior que o concorrente, conclusão invisível a qualquer análise baseada em médias [Hoefler and Belli 2015]. A análise estatística fornece os critérios de rigor para interpretar os dados; o planejamento

de experimentos, tratado a seguir, determina quais dados coletar e em que condições.

2.4. Planejamento de Experimentos

Um experimento mal planejado produz dados que não respondem à pergunta de interesse, independentemente da qualidade da análise estatística aplicada posteriormente. O planejamento define quais fatores serão variados, em quais níveis, com qual número de repetições, e como o ambiente será controlado ou aleatorizado. Essa decisão é anterior e superior a qualquer escolha de ferramenta ou técnica de análise.

Conceitos fundamentais de design de experimentos. Um experimento de desempenho é estruturado em torno de fatores, que são as variáveis controladas pelo experimentador, e de respostas, que são as métricas observadas [Jain 1991]. Cada fator assume um conjunto discreto de valores chamados níveis. Um design fatorial completo 2^k varia k fatores em dois níveis cada e permite estimar tanto os efeitos principais de cada fator quanto as interações entre eles, ou seja, os casos em que o efeito de um fator depende do valor de outro. Para k fatores, o número de experimentos cresce exponencialmente, o que torna o design fatorial completo inviável para k grande. O design fatorial fracionado 2^{k-p} reduz o número de experimentos por um fator de 2^p ao custo de confundir certos efeitos de interação de ordem alta com efeitos principais, troca aceitável quando as interações de ordem alta são presumivelmente desprezíveis. A ANOVA quantifica a contribuição relativa de cada fator à variância total da resposta, permitindo identificar quais fatores têm impacto prático relevante e quais podem ser fixados em experimentos subsequentes.

Documentação do setup experimental. Hoefler [Hoefler and Belli 2015] estabelece que todos os fatores variantes e o setup completo devem ser documentados. Na prática isso significa registrar, além do hardware, as versões exatas de compiladores e flags de compilação, versões de bibliotecas e middleware, configurações do sistema de arquivos e do gerenciador de jobs, parâmetros de entrada e seus geradores. Um erro recorrente é assumir que mencionar o nome de um sistema famoso, como um supercomputador ou uma suíte de benchmarks conhecida, descreve o setup de forma suficiente. Atualizações regulares de software e firmware mudam os resultados, e parâmetros implícitos não são universais. O código-fonte e os dados de entrada devem estar disponíveis em repositório público, condição necessária para que outros possam verificar, reproduzir ou estender os resultados.

Aspectos específicos de HPC no design experimental. O ambiente de execução em sistemas paralelos introduz fontes de variabilidade ausentes em experimentos monoprocesso [Hoefler and Belli 2015]. A alocação de nós pelo gerenciador de jobs pode variar entre execuções, afetando a topologia de comunicação. O mapeamento processo-nó impacta a localidade de memória e a contagem de saltos na rede. A interferência de outros jobs no sistema compartilhado é, em geral, não controlável. Quando o controle direto dessas variáveis não é possível, a randomização é uma alternativa válida: executar experimentos em ordem aleatória absorve parte da variabilidade sistemática e impede que efeitos de aprendizado ou de aquecimento do sistema sejam confundidos com efeitos dos fatores de interesse. Na seleção dos níveis dos fatores, potências de dois para número de processos são casos especiais que frequentemente ativam otimizações específicas em algoritmos de comunicação coletiva; resultados obtidos exclusivamente nesses valo-

res podem não ser representativos do comportamento geral. Uma abordagem de refinamento adaptativo de níveis, por exemplo como aquela empregada no benchmark SKaMPI [Reussner et al. 2002], concentra medições nas regiões de maior incerteza ou variação, evitando tanto a subamostragem em regiões de comportamento complexo quanto o desperdício em regiões de comportamento monótono.

Strong e weak scaling. Experimentos de escalabilidade devem declarar explicitamente qual modalidade é utilizada [Hoefler and Belli 2015]. No strong scaling, o problema é fixo e o número de processos varia; a métrica natural é o speedup em relação à execução com um único processo. No weak scaling, o tamanho do problema cresce proporcionalmente ao número de processos; a métrica natural é a eficiência de scaling, mas a função de escalonamento do problema deve ser especificada, pois diferentes escolhas, como escalar apenas a dimensão espacial ou escalar todas as dimensões simultaneamente, produzem resultados incomparáveis entre si. Apresentar resultados de weak scaling sem documentar quais dimensões foram escaladas e com qual função torna a comparação com outros trabalhos impossível. Com o experimento planejado e o instrumental estatístico disponível, a próxima seção sistematiza os erros mais comuns na prática de HPC e as formas de evitá-los.

2.5. Erros Comuns e Melhores Práticas Específicas para HPC

Com a base estatística e o instrumental de planejamento das seções anteriores, é possível identificar com precisão os erros mais comuns e as melhores práticas específicas para HPC a serem consideradas em uma avaliação de desempenho.

O Estado da Prática em HPC: Evidências. A evidência empírica sobre o estado da prática em HPC é desconfortável. Como vimos, Hoefler et. al [Hoefler and Belli 2015] analisaram uma amostra estratificada de 120 artigos publicados e documentaram sistematicamente o que se pratica nas melhores conferências da área. No que diz respeito ao projeto experimental, o hardware é razoavelmente documentado, mas o ambiente de software, incluindo versões de compiladores, flags de compilação e configuração de filesystem, é negligenciado de forma quase universal. O setup de medição foi descrito em apenas 30 dos 95 artigos aplicáveis. Na análise de dados, 38% dos artigos que reportam aceleração não incluem o desempenho absoluto do caso base; há confusão sistemática entre unidades de contagem e de taxa (MFLOP versus MFLOP/s); e é comum o cherry-picking de benchmarks e configurações favoráveis. A conclusão dos autores é direta: a maioria dos resultados publicados em HPC não é reprodutível nem sequer interpretável por terceiros.

Erros na formulação e no design do experimento. O erro mais custoso é o que ocorre antes de qualquer medição: objetivos mal definidos ou ausentes determinam tudo o que vem depois. Uma consequência direta é a escolha de métricas erradas ou não alinhadas ao objetivo, como reportar aceleração sem especificar o caso base em termos absolutos [Hoefler and Belli 2015]. A distinção é essencial: aceleração em relação a um processo serial único é diferente de aceleração em relação à melhor implementação serial disponível, e a omissão do desempenho absoluto torna a comparação entre sistemas distintos impossível. Outra fonte recorrente de ambiguidade é o uso de unidades imprecisas: o comitê PARKBENCH recomenda reservar *flop* para contagem de operações, *flop/s* para

taxa, B para bytes e b para bits, com prefixos binários conforme IEC 60027-2. Por fim, reportar apenas o subconjunto de benchmarks ou configurações que favorece as conclusões, sem justificativa explícita, invalida qualquer comparação.

Erros na coleta de dados. Um erro estrutural comum é não verificar se as amostras coletadas são independentes e estacionárias antes de aplicar qualquer análise. Associado a isso está o uso de um número arbitrário de repetições: a recomendação informal de que 30 a 40 amostras [Jain 1991] garantem normalidade pela via do teorema central do limite é falsa para dados de desempenho em HPC, cujas distribuições raramente satisfazem as condições necessárias [Hoefler and Belli 2015] [Le Boudec 2011]. Outros erros frequentes incluem ignorar o período de aquecimento, especialmente crítico em sistemas de comunicação que estabelecem estado sob demanda, e não controlar o estado de cache entre execuções. Em medições com temporizadores, a sobrecarga de instrumentação deve ser inferior a 5% do intervalo medido e a resolução do timer deve ser pelo menos dez vezes maior que esse intervalo. Em sistemas paralelos, há armadilhas adicionais: barreiras MPI e OpenMP não oferecem garantias de temporização, relógios entre processos derivam ao longo do tempo, e as nP medições coletadas nos diferentes ranks precisam ser verificadas por ANOVA antes de serem agregadas, para confirmar que provêm da mesma população.

Erros na análise estatística. Com a base estabelecida na Seção 2.3, é possível identificar os erros mais frequentes em sua aplicação ao contexto de HPC. O mais disseminado é usar a média aritmética como resumo universal. Distribuições de desempenho em HPC raramente são simétricas: assimetria à direita, caudas pesadas e multimodalidade são a regra [Bruel et al. 2023]. A média aritmética é apropriada apenas para custos como tempo e energia; para taxas como flop/s a medida correta é a média harmônica; e médias de razões devem ser evitadas [Hoefler and Belli 2015]. A normalidade não deve ser presumida sem diagnóstico: o teste de Shapiro-Wilk combinado com um gráfico Q-Q é o mínimo necessário antes de aplicar qualquer estatística paramétrica. Comparações válidas requerem testes adequados ao tipo de dado: t-test ou ANOVA para dados normais, Wilcoxon ou Kruskal-Wallis para dados não-paramétricos, sempre com correção para múltiplas comparações quando aplicável. Identificados os erros a evitar, a seção seguinte aprofunda um dos elementos irredutíveis da avaliação: a caracterização da carga de trabalho em HPC.

2.6. Caracterização de Carga de Trabalho

A carga de trabalho é o conjunto de estímulos aplicados ao sistema durante a avaliação, identificada na Seção 2.2 como um dos três elementos irredutíveis de qualquer avaliação ao lado do sistema e das métricas. Sua escolha é uma das decisões mais determinantes para a validade das conclusões. Uma avaliação rigorosa requer que a carga seja ao mesmo tempo representativa do uso real e controlável o suficiente para que os experimentos sejam reprodutíveis [Jain 1991].

Parâmetros e dimensões da caracterização. Jain [Jain 1991] identifica três parâmetros fundamentais de qualquer carga de trabalho: a intensidade, que determina o nível de demanda imposto ao sistema; o mix de operações, que define a proporção relativa de diferentes tipos de requisição; e os padrões de acesso, que descrevem como os dados são referenciados no espaço e no tempo. Em sistemas computacionais gerais, esses parâme-

tros são suficientes para capturar o comportamento relevante. Em HPC, no entanto, a carga tem dimensões adicionais: o padrão de comunicação entre processos MPI, a hierarquia de memória acessada por cada thread, e o padrão de acesso ao sistema de arquivos paralelo. A caracterização incompleta de qualquer uma dessas dimensões compromete a validade dos resultados obtidos.

Workload real versus benchmark: o trade-off central. O workload real, obtido por captura de rastros de execuções da aplicação de interesse, é representativo por definição, mas é de difícil reprodução em outro ambiente e não permite parametrização simples. O benchmark sintético, por outro lado, é facilmente reprodutível e controlável, mas pode não capturar o comportamento real das aplicações de interesse. Essa tensão não tem resolução universal: a escolha depende do objetivo da avaliação. Para estudos de configuração de sistema, um benchmark sintético calibrado pelos parâmetros extraídos de rastros reais frequentemente oferece o melhor compromisso [Stanisic et al. 2017]. Para estudos de otimização de aplicação específica, o rastro real ou uma reprodução fiel dele é indispensável. O que não é aceitável é usar benchmarks sintéticos sem verificar que seus padrões de acesso são compatíveis com os da aplicação que motivou o estudo.

Caracterização de padrões de comunicação MPI. A análise de rastros de comunicação MPI revela a estrutura da carga em sistemas paralelos [Hager and Wellein 2010]. As distribuições de tamanho de mensagem determinam qual regime de comunicação domina: mensagens pequenas são dominadas pela latência da rede, enquanto mensagens grandes são dominadas pela largura de banda. A proporção entre operações coletivas e ponto-a-ponto define a escala do problema de sincronização. As operações coletivas, em particular, exibem comportamentos muito distintos dependendo do tamanho do comunicador, do hardware de rede e da implementação MPI. Uma caracterização adequada identifica essas distribuições e as usa para selecionar ou construir benchmarks que exercitem os mesmos regimes.

Caracterização de acessos à memória e I/O. A localidade espacial e temporal determina as taxas de acerto nos diferentes níveis de cache. Os contadores de hardware fornecem cache miss rates por nível de cache, número de transações de memória e taxa de reuso das linhas de cache. Para I/O paralelo, os padrões de acesso determinam a eficiência do sistema de arquivos: acesso sequencial com *stripe size* adequado atinge a largura de banda de pico, enquanto acesso aleatório de pequeno porte satura os metadados e colapsa o desempenho. A contenção em operações coletivas de I/O, quando múltiplos processos acessam a mesma região do arquivo, é uma fonte frequente de gargalo não identificada sem instrumentação de I/O.

Benchmarks consagrados em HPC. A comunidade HPC possui um conjunto de benchmarks muito utilizados para caracterização de desempenho. O HPL (*High Performance LINPACK*) [Petitet et al. 2004] mede a capacidade de ponto flutuante de precisão dupla resolvendo um sistema linear denso, sendo esta a métrica do ranking Top500 [TOP500 Project 2024]. O HPCG [Dongarra et al. 2016] (*High Performance Conjugate Gradient*) complementa o HPL com padrões de acesso esparso e comunicação irregular, mais representativos de aplicações científicas reais. Os NAS Parallel Benchmarks [Bailey et al. 1991] cobrem os principais núcleos computacionais de simulações científicas (BT, CG, FT, IS, LU, MG, SP) e permitem avaliação de strong e weak scaling. O

Graph500 [Murphy et al. 2010] mede o desempenho em análise de grafos, com padrões de acesso altamente irregulares e baixa razão computação/comunicação. Uma advertência crítica de [Hoefler and Belli 2015] é que, ao usar suítes existentes, o pesquisador deve rodar todos os benchmarks ou justificar explicitamente a omissão de qualquer um deles. Selecionar apenas os benchmarks que exibem os melhores resultados para o sistema em avaliação é uma forma de cherry-picking que invalida qualquer comparação posterior com outros sistemas.

Mini-aplicações como alternativa. As mini-aplicações são programas compactos que capturam os padrões computacionais essenciais de aplicações reais, sem a complexidade de código de produção. LULESH (*Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics*) [Karlin et al. 2013] representa os kernels de simulações de hidrodinâmica; miniMD [Sandia National Laboratories 2011] representa dinâmica molecular; AMG [Lawrence Livermore National Laboratory 2013] representa multigrid algébrico para resolução de sistemas lineares esparsos. As mini-aplicações combinam a controlabilidade dos benchmarks sintéticos com maior fidelidade aos padrões de acesso das aplicações reais, tornando-as adequadas para estudos de otimização de arquitetura e de compiladores. Sua limitação é que abstraem aspectos de gerenciamento de memória e de estruturas de dados que podem ser determinantes no desempenho da aplicação completa. O projeto SMPI Proxy Apps [Degomme et al. 2017] consolida diversas mini-aplicações instrumentadas para avaliação do comportamento comunicacional com o simulador SimGrid/SMPI [Casanova et al. 2014]. O SMPI reimplementa a interface MPI sobre o mecanismo de simulação do SimGrid, permitindo executar mini-aplicações sem modificação de código-fonte em plataformas virtuais calibradas a partir de modelos analíticos do hardware alvo. Essa abordagem é valiosa quando o sistema real é de acesso restrito ou ainda está em estágio de projeto: o comportamento comunicacional da aplicação pode ser avaliado e comparado entre arquiteturas alternativas sem necessidade de acesso físico ao supercomputador. Definida e caracterizada a carga de trabalho, o passo seguinte é instrumentar o sistema para coletar as métricas planejadas com a precisão requerida.

2.7. Monitoramento e Rastreamento em HPC

Medir desempenho em sistemas HPC exige instrumentação especializada. A instrumentação introduz sobrecarga que pode alterar o comportamento que se pretende medir, e a escolha das ferramentas deve equilibrar granularidade de informação, sobrecarga de medição e facilidade de uso.

Contadores de hardware. Os contadores de hardware, expostos pela unidade de monitoramento de desempenho (PMU) do processador, fornecem visibilidade sobre o comportamento interno do hardware sem instrumentar o código-fonte. PAPI (*Performance Application Programming Interface*) [Mucci et al. 1999] oferece uma interface portável para leitura de contadores de hardware através de múltiplas arquiteturas, permitindo medir cache misses, branch mispredictions, instruções por ciclo (IPC) e outros eventos de hardware dentro de regiões específicas do código. O utilitário `perf` do Linux e o LIKWID (*Like I Knew What I'm Doing*) [Treibig et al. 2010] fornecem acesso similar com menor custo de integração, sendo este último especialmente adequado para medições precisas de largura de banda de memória e eficiência de vetorização. Uma precaução essencial é verificar se os contadores de hardware disponíveis em determinado sistema cobrem os

eventos de interesse. Arquiteturas distintas definem eventos distintos, e a correspondência entre nomes portáveis (PAPI) e eventos nativos do hardware deve ser validada. A sobrecarga de leitura dos contadores é tipicamente de algumas dezenas de nanosegundos por acesso, tornando-os adequados para instrumentar regiões de código com duração superior a alguns microssegundos.

Monitoramento de comunicação MPI e de I/O. Para aplicações MPI, a instrumentação da camada de comunicação revela onde o tempo é gasto em operações de rede. Score-P [Knüpfer et al. 2012] e TAU [Shende and Malony 2006] são frameworks de instrumentação automática que interceptam chamadas MPI e OpenMP, registrando tempos de entrada e saída de cada operação e gerando rastros que podem ser analisados posteriormente. mpiP [Vetter and Mueller 2001] é uma alternativa mais leve baseada em amostragem estatística, com sobrecarga tipicamente inferior a 1% e adequada para execuções em produção em que instrumentação pesada é inaceitável. Para análise de I/O paralelo, o Darshan [Carns et al. 2009] é a ferramenta de referência: registra de forma não-intrusiva todos os acessos ao sistema de arquivos (POSIX, MPI-IO, HDF5), capturando distribuições de tamanho de operação, sequencialidade, alinhamento e tempo de abertura de arquivo. A sobrecarga do Darshan é suficientemente baixa para ser habilitada por padrão em vários supercomputadores de produção. Para monitoramento de energia, o Intel RAPL (*Running Average Power Limit*) [David et al. 2010] fornece estimativas de consumo por package de processador e memória DRAM com resolução temporal de milissegundos; o NVML da NVIDIA [NVIDIA Corporation 2024] fornece consumo de GPU; e o IPMI (*Intelligent Platform Management Interface*) [Intel Corporation 2013] fornece consumo de nó completo, incluindo armazenamento e rede.

Sobrecarga versus amostragem: o trade-off fundamental. A instrumentação por rastreamento registra cada evento individualmente, produzindo um rastro completo da execução, mas com sobrecarga proporcional ao número de eventos. A instrumentação por amostragem interrompe periodicamente a execução e registra o contexto atual (call stack, contadores de hardware), produzindo uma visão estatística com sobrecarga controlada e independente do número de eventos. Para diagnóstico detalhado de comportamento de comunicação, o rastreamento é necessário; para identificação de *hotspots* em código de computação de longa duração, a amostragem é suficiente e menos intrusiva. A sobrecarga de instrumentação deve ser inferior a 5% do intervalo medido; a própria sobrecarga deve ser medida antes de coletar os dados experimentais. Um sistema de monitoramento com 20% de sobrecarga não mede o sistema original: mede o sistema instrumentado, que pode ter comportamento qualitativamente diferente do sistema de interesse.

Identificação de gargalos. O modelo Roofline [Williams et al. 2009] organiza o espaço de desempenho em torno de dois recursos: capacidade de ponto flutuante (flop/s) e largura de banda de memória (B/s). O eixo horizontal é a intensidade aritmética da aplicação, medida em flop/B. Para intensidades baixas, a aplicação é limitada pela memória; para intensidades altas, é limitada pelo processador. A posição da aplicação no diagrama Roofline, combinada com a posição dos tetos empíricos dos recursos, determina qual otimização tem potencial de impacto. O modelo Roofline, porém, tem limitações relevantes para o contexto HPC [Hager et al. 2016]. A intensidade aritmética de uma aplicação não é uma constante portável: seu valor depende dos tamanhos de cache da arquitetura alvo, de modo que a mesma aplicação pode ter intensidades aritméticas distintas em máqui-

nas diferentes. Além disso, o modelo ignora por completo os custos de comunicação MPI, que são centrais em aplicações de larga escala. O modelo ECM (*Execution-Cache-Memory*) [Hager et al. 2016], desenvolvido pelo grupo de Erlangen, endereça a primeira limitação ao decompor o tempo de execução em contribuições de cada nível da hierarquia de cache (L1, L2, L3, memória principal), permitindo prever com precisão o scaling multicore sem depender de medição direta da intensidade aritmética. O custo é uma maior complexidade de construção: o modelo requer microbenchmarks por nível de cache e conhecimento detalhado dos tempos de transferência entre níveis da hierarquia. Para gargalos de comunicação, a decomposição entre latência e largura de banda é essencial: o modelo $t = \alpha + n/\beta$, onde t é o tempo de transferência em segundos, n é o tamanho da mensagem em bytes, α é a latência em segundos e β a largura de banda em bytes por segundo, determina o regime de operação de cada mensagem. Em aplicações com muitas mensagens pequenas, o gargalo é a latência; em aplicações com poucas mensagens grandes, é a largura de banda. O desequilíbrio de carga entre ranks MPI é visível como variação no tempo de execução entre processos. [Hager and Wellein 2010] ilustram que a variação no tempo de `MPI_Reduce` entre nós de um cluster pode atingir um fator de dois mesmo em condições nominalmente idênticas, resultado da interferência de ruído de sistema sobre as sincronizações coletivas. Com os gargalos identificados por instrumentação e modelagem, a seção seguinte discute como comunicar os resultados de forma clara, honesta e cientificamente rigorosa.

2.8. Apresentação de Resultados

A apresentação visual de resultados é a última etapa do ciclo experimental e a mais visível para o leitor. Uma visualização que oculta variabilidade ou distorce a comparação compromete a interpretabilidade dos resultados, independentemente do rigor da coleta e da análise [Tufte 2001] [Few 2012].

Box plots: semântica precisa e uso correto. O box plot é o gráfico mais usado para apresentar distribuições de desempenho, mas sua semântica precisa deve ser sempre declarada explicitamente [Jain 1991]. A caixa representa o intervalo interquartil (IQR), do primeiro ao terceiro quartil; a linha central representa a mediana; os whiskers devem ter sua definição declarada, pois existem pelo menos três convenções distintas: $1{,}5\times$IQR além dos quartis (Tukey), mínimo e máximo excluindo outliers por algum critério, ou percentis fixos como P5 e P95. Pontos além dos whiskers são outliers segundo a convenção adotada e devem ser exibidos individualmente. Os notches (entalhes laterais na caixa, que produzem um estreitamento visual na altura da mediana) representam um intervalo de confiança aproximado para a mediana. A expressão $\pm 1{,}58 \times IQR/\sqrt{n}$, introduzida por [McGill et al. 1978], é derivada de uma aproximação assintótica que relaciona o IQR ao desvio padrão sob normalidade, e deve ser entendida como uma heurística visual, não como um intervalo de confiança exato. A não-sobreposição de notches entre dois box plots indica diferença estatisticamente significativa entre as medianas ao nível aproximado de 5%, oferecendo uma inspeção visual rápida sem necessidade de teste formal. Essa semântica deve ser declarada na legenda; notches sem explicação induzem o leitor a interpretações incorretas.

Violin plots e a distribuição completa. O violin plot combina um box plot com uma estimativa de densidade por kernel (KDE), revelando a forma completa da distribuição:

multimodalidade, assimetria e modos estreitos são visíveis no violin mas invisíveis no box plot. Para dados de desempenho HPC, cuja distribuição raramente é unimodal simétrica [Bruel et al. 2023], o violin plot fornece uma visão mais honesta da variabilidade do sistema. O custo é a maior área visual ocupada e a necessidade de um número razoável de amostras (tipicamente $n \geq 20$) para que a estimativa de densidade seja informativa. A combinação de box plot e violin plot é considerada a melhor prática corrente: o violin exibe a distribuição completa e o box plot interno fornece os quantis de referência. Essa combinação é diretamente suportada por bibliotecas como `ggplot2` em R e `seaborn` em Python.

Histogramas. O histograma revela a distribuição completa dos dados agrupados em intervalos e é especialmente útil para identificar multimodalidade e caudas pesadas que o box plot não exibe [Jain 1991]. A escolha do número de intervalos é crítica: poucos intervalos suavizam estruturas relevantes; muitos intervalos amplificam ruído e tornam o padrão ilegível. A regra de Sturges [Sturges 1926] ($k = \lceil 1 + \log_2 n \rceil$) oferece um ponto de partida, mas para dados de desempenho com distribuições de cauda pesada, a regra de Scott [Scott 1979] ou Freedman-Diaconis [Freedman and Diaconis 1981] produz resultados mais adequados. Para comparação de distribuições de dois sistemas, histogramas sobrepostos com transparência ou histogramas de diferença são mais informativos que dois histogramas lado a lado.

Visualização de rastros. Vampir [Knüpfer et al. 2008] e Paraver [Pillet et al. 1995] são as principais ferramentas de visualização de rastros de execução paralela. Ambas representam o tempo no eixo horizontal e os processos MPI (ou threads OpenMP) no eixo vertical, permitindo identificar visualmente padrões de comunicação, desbalanceamento de carga e janelas de serialização. O Projections [Kale et al. 2006] é a ferramenta correspondente para aplicações Charm++, com suporte à análise de desempenho de tarefas e a granulares objetos paralelos. O ViTE [Coulomb et al. 2012] é uma ferramenta de código aberto voltada à visualização de rastros em formato OTF2 e Paje, com ênfase em heterogeneidade e suporte a runtimes como StarPU. A visualização revela gargalos que estatísticas agregadas ocultam: um rank que inicia uma operação MPI coletiva antes dos demais aparece como idle enquanto aguarda; esse idle time é visível no rastro como uma lacuna no estado do processo, mas seria absorvido na média de tempo de execução dos ranks [Schnorr and Legrand 2013]. A presença de ferramentas de ciência de dados para análise de rastros é crescente. O Pipit [Bhatele et al. 2023] exemplifica essa tendência: trata-se de uma biblioteca Python que expõe rastros de execução paralela como DataFrames, permitindo ao usuário explorar, filtrar e visualizar o comportamento de aplicações com a expressividade de ambientes de análise de dados como pandas e Jupyter. Abordagens desse tipo permitem análises customizadas que ferramentas tradicionais não suportam diretamente, como correlação entre métricas de desempenho e parâmetros da aplicação ao longo de múltiplos experimentos. No contexto de aplicações baseadas em tarefas, [Asch and Schnorr 2025] demonstram a viabilidade dessa abordagem para aplicações Charm++, analisando rastros de simulações de dinâmica molecular com pipelines reprodutíveis baseados em bibliotecas de ciência de dados.

Princípios gerais de visualização para desempenho. Eixos devem começar em zero quando a razão entre valores plotados e escala é relevante para a conclusão; truncar o eixo y para amplificar diferenças pequenas é uma forma recorrente de distorção visual

[Tufte 2001]. Linhas de conexão entre pontos são válidas apenas quando a interpolação entre os valores plotados tem significado: em gráficos de scaling com valores de potências de dois no eixo x, a linha conecta pontos em que experimentos existem, e sua inclinação comunica a taxa de crescimento; conectar pontos com configurações categoricamente distintas produz a falsa impressão de uma tendência contínua. Bounds analíticos, como speedup ideal linear ou o teto do Roofline, devem ser incluídos nos gráficos de desempenho para contextualizar as medições empíricas: um resultado não pode ser avaliado como bom ou ruim sem uma referência [Jain 1991]. A variabilidade deve estar visível em todos os gráficos que apresentam resultados não-determinísticos. Um gráfico de barras com apenas a altura média, sem qualquer indicação de dispersão, é cientificamente inadequado para dados de sistemas HPC. A escolha entre error bars de desvio padrão e de intervalo de confiança deve ser declarada explicitamente: desvio padrão descreve a dispersão dos dados; intervalo de confiança descreve a incerteza sobre a média ou mediana estimada. Confundir as duas representações é um erro recorrente que leva a conclusões sobre a comparação entre sistemas que os dados não suportam.

2.9. Conclusão

Este capítulo partiu de um diagnóstico empírico incômodo: a prática de avaliação de desempenho em HPC é sistematicamente deficiente. O percurso construído a partir desse diagnóstico teve um fio condutor: a tensão entre o rigor metodológico dos clássicos [Jain 1991] [Lilja 2000] [Le Boudec 2011], desenvolvido para sistemas gerais com forte base estatística, e as especificidades do ambiente HPC, com seu não-determinismo, sua variabilidade sistêmica e suas escalas de máquina irreprodutíveis. O capítulo estabeleceu os fundamentos conceituais e os modelos analíticos de desempenho; construiu a base estatística e o instrumental de planejamento de experimentos; sistematizou os erros mais comuns e como evitá-los; e aprofundou três aspectos específicos de HPC: a caracterização de carga de trabalho que conecta benchmarks e mini-aplicações às aplicações reais que motivam o estudo; o monitoramento e rastreamento, com a cadeia de ferramentas desde contadores de hardware até rastreamento de MPI e I/O e a identificação de gargalos via modelos Roofline e ECM; e a apresentação de resultados, com a semântica precisa de box plots, violin plots, histogramas e visualizações de rastros. A resolução da tensão entre rigor e realismo não é um compromisso, mas uma síntese: adotar o rigor dos clássicos adaptado às realidades do HPC moderno. As três principais mensagens que podemos deixar dessa reflexão sobre análise de desempenho são as seguintes.

- **Desempenho não é um número, é uma distribuição.** A mudança de perspectiva mais importante é esta. Enquanto sistemas HPC forem não-determinísticos, qualquer estatística resumida é uma aproximação com grau de incerteza que precisa ser quantificado e reportado. Distribuições multimodais, assimétricas e de cauda longa são a regra em HPC, não a exceção [Bruel et al. 2023], e nenhuma média, por mais bem calculada, captura esse comportamento. A implicação prática é planejar experimentos para medir distribuições, não pontos, e dimensionar o número de amostras dinamicamente, não por regra de bolso.
- **Interpretabilidade é a meta realista em HPC.** Reprodutibilidade exata é frequentemente difícil em supercomputadores de acesso restrito; interpretabilidade, enten-

dida como fornecer informação suficiente para que outros entendam, avaliem e generalizem os resultados, é alcançável e obrigatória [Hoefler and Belli 2015]. Seus três pilares são a documentação completa do ambiente experimental, a análise estatística adequada ao tipo de dado e a visualização honesta da variabilidade.

- **Erros metodológicos têm consequências científicas reais.** Usar média aritmética para taxas produz valores incorretos. Assumir normalidade sem diagnóstico leva a intervalos de confiança errados e comparações inválidas. Não reportar variabilidade torna impossível distinguir melhoria real de ruído experimental. Cherry-picking de benchmarks e configurações invalida qualquer comparação. Esses não são detalhes técnicos: são a diferença entre ciência e impressão.

As 12 regras de [Hoefler and Belli 2015] como checklist mínimo. As doze regras cobrem: reportar se a base de aceleração é processo serial único ou melhor implementação serial; usar unidades não-ambíguas (*flop/s*, B para bytes, b para bits, prefixos binários IEC); usar média aritmética apenas para custos e média geométrica para razões; reportar intervalos de confiança; não assumir normalidade sem diagnóstico; comparar com teste estatístico adequado; documentar hardware, software, versões, flags e configuração completa; reportar técnica de medição e sincronização para tempo paralelo; mostrar bounds analíticos de desempenho; e conectar pontos por linhas apenas quando a interpolação tem significado.

Checklist prático. Antes do experimento: objetivos escritos, métricas alinhadas aos objetivos, baseline definida em termos absolutos, workload justificado, critério de parada definido, ambiente de controle planejado e script de automação completo antes de rodar qualquer experimento. Durante a coleta: warm-up excluído, sobrecarga de instrumentação verificado, resolução do timer verificada, número de amostras determinado dinamicamente. Após a coleta: independência e estacionariedade verificadas, distribuição examinada antes de escolher o resumo, normalidade testada se forem usadas estatísticas paramétricas, intervalos de confiança calculados, comparações feitas com teste adequado, outliers investigados. Na apresentação: variabilidade visível em todos os gráficos, unidades não-ambíguas, valores absolutos reportados junto com todas as razões, bounds analíticos incluídos nos gráficos de scaling, setup completo documentado, código e dados brutos disponíveis em repositório público.

Leituras complementares. Para fundamentos gerais: [Jain 1991], [Le Boudec 2011] e [Lilja 2000]. Para metodologia específica para HPC: [Hoefler and Belli 2015] é leitura obrigatória; [Bruel et al. 2023] oferece a perspectiva de distribuições. Para modelos de desempenho: [Hager and Wellein 2010] cobre o modelo Roofline; [Hager et al. 2016] apresenta o modelo ECM (*Execution-Cache-Memory*) para análise de hierarquia de cache. Para design de experimentos: [Montgomery 2012] é a referência para ANOVA e designs fatoriais. Para visualização: [Tufte 2001] e [Few 2012]. Entre as ferramentas práticas, destacam-se R/tidyverse e Python/pandas para transformação de dados por cadernos de anotação. Para a coleta controlada de dados, podemos usar desde PAPI para contadores de hardware até Score-P para instrumentação e rastreamento de aplicação HPC. Enfim, a publicação pode ser acompanhada de dados e artefatos públicos no Zenodo.

Referências

[Asch and Schnorr 2025] Asch, C. and Schnorr, L. M. (2025). Profiling a task-based molecular dynamics application with a data science approach. In *Latin America High Performance Computing Conference (CARLA)*.

[Bailey et al. 1991] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. (1991). The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73.

[Bhatele et al. 2023] Bhatele, A., Dhakal, R., Movsesyan, A., Ranjan, A. K., and Cankur, O. (2023). Pipit: Scripting the analysis of parallel execution traces. *arXiv preprint arXiv:2306.11177*.

[Bruel et al. 2023] Bruel, P., Mittal, V., Milojicic, D., Faloutsos, M., and Frachtenberg, E. (2023). Revisiting performance evaluation in the age of uncertainty. In *2023 IEEE 30th International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW)*, pages 1–8. IEEE.

[Carns et al. 2009] Carns, P., Latham, R., Ross, R., Iskra, K., Lang, S., and Riley, K. (2009). 24/7 characterization of petascale I/O workloads. In *Proceedings of the IEEE International Conference on Cluster Computing and Workshops*.

[Casanova et al. 2014] Casanova, H., Giersch, A., Legrand, A., Quinson, M., and Suter, F. (2014). Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917.

[Che et al. 2009] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *International Symposium on Workload Characterization (IISWC)*, pages 44–54. IEEE.

[Coulomb et al. 2012] Coulomb, K., Degomme, A., Faverge, M., and Trahay, F. (2012). An open-source tool-chain for performance analysis. In *Tools for High Performance Computing 2011: Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing*, pages 37–48, Berlin, Heidelberg. Springer.

[David et al. 2010] David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R., and Le, C. (2010). RAPL: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, pages 189–194.

[Degomme et al. 2017] Degomme, A., Legrand, A., Markomanolis, G., Quinson, M., Stillwell, M., and Suter, F. (2017). Simulating MPI applications: The SMPI approach. *IEEE Transactions on Parallel and Distributed Systems*, 28(8):2387–2400.

[Denis et al. 2022] Denis, A., Jaeger, J., Jeannot, E., and Reynier, F. (2022). A methodology for assessing computation/communication overlap of MPI nonblocking collectives. *Concurrency and Computation: Practice and Experience*, 34(22).

[Dongarra et al. 2016] Dongarra, J., Heroux, M. A., and Luszczek, P. (2016). High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *International Journal of High Performance Computing Applications*, 30(1):3–10.

[Few 2012] Few, S. (2012). *Show Me the Numbers: Designing Tables and Graphs to Enlighten*. Analytics Press, Burlingame, CA, 2 edition.

[Freedman and Diaconis 1981] Freedman, D. and Diaconis, P. (1981). On the histogram as a density estimator: L2 theory. *Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete*, 57(4):453–476.

[Hager et al. 2016] Hager, G., Treibig, J., Habich, J., and Wellein, G. (2016). Exploring performance and power properties of modern multicore chips via simple machine models. *Concurrency and Computation: Practice and Experience*, 28(2):189–210.

[Hager and Wellein 2010] Hager, G. and Wellein, G. (2010). *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Boca Raton, FL.

[Hoefler and Belli 2015] Hoefler, T. and Belli, R. (2015). Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*, pages 1–12, Austin, TX, USA. ACM.

[Intel Corporation 2013] Intel Corporation (2013). Intelligent platform management interface specification, version 2.0. Technical report, Intel Corporation. `https://www.intel.com/content/www/us/en/products/docs/servers/ipmi/ipmi-second-gen-interface-spec-v2-rev1-1.html`.

[Jain 1991] Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, New York.

[Kale et al. 2006] Kale, L. V., Zheng, G., Lee, C. W., and Kumar, S. (2006). Scaling applications to massively parallel machines using projections performance analysis tool. *Future Generation Computer Systems*, 22(3):347–358.

[Karlin et al. 2013] Karlin, I., Keasler, J., and Neely, R. (2013). LULESH 2.0 updates and changes. Technical Report LLNL-TR-641973, Lawrence Livermore National Laboratory.

[Kerr 1998] Kerr, N. L. (1998). HARKing: Hypothesizing after the results are known. *Personality and Social Psychology Review*, 2(3):196–217.

[Knüpfer et al. 2008] Knüpfer, A., Brunst, H., Doleschal, J., Geimer, M., Jurenz, M., Lieber, M., Mickler, H., Otto, S., Rahn, M., Schilling, K., Wylie, B. J. N., and Nagel, W. E. (2008). The Vampir performance analysis tool-set. In *Tools for High Performance Computing*, pages 139–155. Springer.

[Knüpfer et al. 2012] Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., Nagel, W. E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., and Wolf, F. (2012). Score-P: A joint performance measurement runtime infrastructure for Periscope, Scalasca, TAU and Vampir. In *Tools for High Performance Computing 2011*, pages 79–91. Springer.

[Lawrence Livermore National Laboratory 2013] Lawrence Livermore National Laboratory (2013). AMG: Algebraic multigrid benchmark. `https://github.com/LLNL/AMG`.

[Le Boudec 2011] Le Boudec, J.-Y. (2011). *Performance Evaluation of Computer and Communication Systems*. CRC Press, USA.

[Lilja 2000] Lilja, D. J. (2000). *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, Cambridge, UK.

[McGill et al. 1978] McGill, R., Tukey, J. W., and Larsen, W. A. (1978). Variations of box plots. *The American Statistician*, 32(1):12–16.

[Montgomery 2012] Montgomery, D. C. (2012). *Design and Analysis of Experiments*. Wiley, New York, 8 edition.

[Mucci et al. 1999] Mucci, P. J., Browne, S., Deane, C., and Ho, G. (1999). PAPI: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10.

[Murphy et al. 2010] Murphy, R. C., Wheeler, K. B., Barrett, B. W., and Ang, J. A. (2010). Introducing the Graph500. In *Cray User Group (CUG)*.

[NVIDIA Corporation 2024] NVIDIA Corporation (2024). *NVIDIA Management Library (NVML) Reference*. `https://developer.nvidia.com/nvidia-management-library-nvml`.

[Pearce et al. 2012] Pearce, O., Gamblin, T., de Supinski, B. R., Schulz, M., and Amato, N. M. (2012). Quantifying the effectiveness of load balance algorithms. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*, pages 185–194, Venice, Italy. ACM.

[Petitet et al. 2004] Petitet, A., Whaley, R. C., Dongarra, J., and Cleary, A. (2004). HPL – a portable implementation of the high-performance Linpack benchmark for distributed-memory computers. Technical report, Innovative Computing Laboratory, University of Tennessee. Disponível em `http://www.netlib.org/benchmark/hpl/`.

[Pillet et al. 1995] Pillet, V., Labarta, J., Cortes, T., and Girona, S. (1995). PARAVER: A tool to visualize and analyze parallel code performance. In *Proceedings of WoTUG-18: Transputer and occam Developments*, pages 17–31. IOS Press.

[Plale et al. 2021] Plale, B., Malik, T., et al. (2021). Reproducibility practice in high-performance computing: Community survey results. *Computing in Science & Engineering*, 23(5):16–26.

[Reussner et al. 2002] Reussner, R. H., Sanders, P., and Träff, J. L. (2002). SKaMPI: A comprehensive benchmark for public benchmarking of MPI. *Scientific Programming*, 10(1):55–65.

[Sandia National Laboratories 2011] Sandia National Laboratories (2011). miniMD: A simple molecular dynamics proxy application. `https://github.com/Mantevo/miniMD`.

[SC Repr. Initiative 2015] SC Repr. Initiative (2015). SC reproducibility initiative. `https://sc24.supercomputing.org/program/reproducibility-initiative/`. Iniciativa lançada em 2015; AD appendix obrigatório desde 2019.

[Schnorr and Legrand 2013] Schnorr, L. M. and Legrand, A. (2013). Visualizing more performance data than what fits on your screen. In *Tools for High Performance Computing 2012*, pages 149–162. Springer.

[Scott 1979] Scott, D. W. (1979). On optimal and data-based histograms. *Biometrika*, 66(3):605–610.

[Shende and Malony 2006] Shende, S. S. and Malony, A. D. (2006). The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311.

[Stanisic et al. 2017] Stanisic, L., Mello Schnorr, L., Degomme, A., Heinrich, F., Legrand, A., and Videau, B. (2017). Characterizing the Performance of Modern Architectures Through Opaque Benchmarks: Pitfalls Learned the Hard Way. In *IPDPS 2017 - 31st IEEE International Parallel & Distributed Processing Symposium (RepPar workshop)*, Orlando, United States.

[Sturges 1926] Sturges, H. A. (1926). The choice of a class interval. *Journal of the American Statistical Association*, 21(153):65–66.

[TOP500 Project 2024] TOP500 Project (2024). TOP500 supercomputer sites. `https://www.top500.org`.

[Treibig et al. 2010] Treibig, J., Hager, G., and Wellein, G. (2010). LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the 39th International Conference on Parallel Processing Workshops (ICPPW)*, pages 207–216.

[Tufte 2001] Tufte, E. R. (2001). *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, 2 edition.

[Vetter and Mueller 2001] Vetter, J. S. and Mueller, F. (2001). Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In *Proceedings of the 15th Intl. Par. and Distributed Processing Symposium (IPDPS)*.

[Williams et al. 2009] Williams, S., Waterman, A., and Patterson, D. (2009). Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76.