Capítulo 1

Boas Práticas para Experimentos Computacionais em Clusters de Alto Desempenho

Lucas Mello Schnorr[1], Vinícius Garcia Pinto[2]
Instituto de Informática, Universidade Federal do Rio Grande do Sul
Porto Alegre, Brasil

Resumo

A temática deste minicurso é em análise de desempenho de aplicações paralelas em clusters de alto desempenho. O minicurso se propõe a sensibilizar os participantes aos fatores que afetam a coleta de medidas, para tornar os experimentos mais confiáveis. O minicurso tem três partes: (a) motivar cuidados essenciais em medidas computacionais para controlar a variabilidade experimental; (b) apresentação de formas de controlar parâmetros em sistemas Linux; e (c) como analisar de maneira reprodutível os dados coletados com linguagens de programação e ferramentas modernas de manipulação de dados.

1.1. Introdução

Um dos pilares do método científico é o uso de experimentos para validar ou refutar hipóteses e teorias que tentam explicar fenômenos naturais. Para ser con-

[1]Lucas Mello Schnorr possui graduação em Ciência da Computação pela Universidade Federal de Santa Maria (2003), mestrado em Computação pela Universidade Federal do Rio Grande do Sul (2005), doutorado em Computação pela Universidade Federal do Rio Grande do Sul com um acordo de cotutela com o Institut National Polytechnique de Grenoble (2009), pós-doutorado pelo Centre National de la Recherche Scientifique (2011) e pós-doutorado pelo Institut National de Recherche en Informatique et en Automatique (2017). Desde 2013 é Professor Adjunto da Universidade Federal do Rio Grande do Sul e orientador do Programa de Pós-Graduação em Computação. Conduz pesquisas em ambiente internacional. Tem experiência na área da Computação, com ênfase em Sistemas de Computação e Processamento de Alto Desempenho.

[2]Vinícius Garcia Pinto possui graduação em Ciência da Computação pela Universidade Federal de Santa Maria (2010), mestrado em Computação pelo Universidade Federal do Rio Grande do Sul (2013) e doutorado em Computação pela Universidade Federal do Rio Grande do Sul em cotutela com a Université Grenoble Alpes / França (2018). Foi professor da Faculdade São Francisco de Assis e da Universidade de Caxias do Sul. Tem experiência em Programação Paralela e Computação de Alto Desempenho

fiável, tais experimentos devem ser reprodutíveis, de forma que outros pesquisadores possam refazer as observações independentemente. A reprodutibilidade se torna possível no momento que se exerce um controle sobre a maior quantidade possível de variáveis que possam afetar o fenômeno sob investigação. A prática consiste também em registrar o valor das variáveis não controladas (ou não controláveis) de forma que elas possam servir de contexto observado do fenômeno.

Experimentos computacionais na grande área de sistemas de computação não são diferentes. O controle e registro de variáveis dos sistemas computacionais é passo obrigatório para tornar qualquer observação computacional mais confiável. Em um cenário com um único computador, esse controle se exerce através do estabelecimento de configurações de todas as camadas do computador, das configurações de hardware (por exemplo, da BIOS) até as de software (sistema operacional e arcabouços). Na área de processamento paralelo de alto desempenho onde múltiplos computadores são utilizados conjuntamente, esse controle deve ser exercido sob os elementos da rede de interconexão.

Tal controle experimental tem desvantagens e vantagens. Por um lado, os experimentos se tornam um processo mais burocrático, envolvendo preparação adicional, cuidado maior no antes, durante e depois dos experimentos, e disciplina reforçada. Tais procedimentos podem fazer com que o avanço da investigação seja mais lento. Por outro lado, um controle reforçado permite que a investigação seja conduzida a partir de uma efeito real, claro, fazendo com que as conclusões delineadas a respeito do fenômeno sejam mais perenes e significativas. Nesta mesma linha, tal controle torna o relato das observações realizadas, na forma de artigos científicos ou relatórios de pesquisa, possa ser enriquecido e também facilmente reprodutível. Por exemplo, os dados coletados podem ser retrabalhados sem a necessidade de realizar uma nova longa bateria experimental.

As boas práticas para experimentos computacionais em clusters de alto desempenho se estendem portanto ao relato das conclusões. Os dados coletados devem ser trabalhados através de programas de computadores (*scripts*) de maneira a retirar o humano da preparação de estatísticas, gráficos e tabelas. Toda transformação de dados deve ser então realizada de maneira automática através de programas preparados pelo analista. Assumindo que um cuidado elevado seja empregado pelo analista na criação destes programas, isso garante que a transformação dos dados reflita exatamente a mensagem que se deseja transmitir.

Levando-se em conta este contexto, este minicurso tem por objetivo sensibilizar os participantes a estas questões. Apresenta portanto um conjunto de boas práticas a serem aplicadas desde a realização de experimentos computacionais em clusters de alto desempenho até a transformação e análise dos dados. Caracterizando-se como uma atividade multidisciplinar, o minicurso envolve conceitos de sistemas operacionais, redes, programação, análise de dados, e processamento paralelo. Ele está estruturado em duas partes:

- **Parte 1: Controle e Coleta** – apresentação de uma lista não exaustiva com as principais formas de controlar sistemas computacionais, focados em ambien-

tes para processamento de alto nível (SO Linux, múltiplos nós, redes de baixa latência), e de projeto experimental [6], para realização de baterias de experimentos com relevância estatística.

- **Parte 2: Análise de Dados** – como analisar os dados coletados com linguagens de programação e ferramentas modernas de transformação de dados que habilitam a reprodutibilidade desta análise, envolvendo conceitos como programação literária [8] e análise de dados com a linguagem R [11].

Esta separação em duas partes reflete um processo metodológico de duas fases. Primeiro, os experimentos são realizados através de mecanismos automáticos enriquecidos com coleta de dados, guiados por um projeto experimental onde constam as variáveis controladas e observadas. Segundo, os dados registrados são analisados pelo analista através de mecanismos, também automáticos, de tratamento de dados. Há portanto uma clara divisão onde a interpretação dos dados observados é realizada *à posteriori*. Considera-se tal divisão importante pois permite um isolamento da fase de coleta. Os dados coletados podem ser analisados sob múltiplas facetas, permitindo interpretações diversas. Essa separação também traz a vantagem de forçar o experimentador e se preocupar com o registro da maior quantidade de informações do sistema computacional. A preocupação é induzida propositalmente pois uma vez finalizada a primeira parte, ao perceber que variáveis não controláveis não foram registradas, o experimentador deve realizar os experimentos com todos os custos de tempo e recurso associados. Portanto, somente uma reexecução completa garante que as configurações observadas refletem a máquina utilizada no experimento.

Este texto está organizado da seguinte forma. A Seção 1.2 apresenta as principais formas de controlar sistemas computacionais, focados em ambientes para processamento de alto nível (SO Linux, múltiplos nós, redes de baixa latência), e de projeto experimental. A Seção 1.3 apresenta métodos de análise com linguagens de programação e ferramentas modernas de manipulação de dados que habilitam a reprodutibilidade desta análise. Enfim, a Seção 1.4 conclui este texto com um sumário do que foi descrito e ponteiros para aprofundar os conceitos apresentados.

1.2. Controle e Coleta

A parte de controle e coleta envolve a fase da realização do experimento computacional. No âmbito do processamento de alto desempenho, consideramos que os experimentos são realizados em um *cluster* de computadores interconectados através de uma rede de interconexão específica para a comunicação de mensagens da aplicação paralela. Um exemplo com quatro nós computacionais está ilustrado na esquerda da Figura 1.1. Um conjunto de processos será executado sobre os vários nós deste tipo de plataforma. Habitualmente, executa-se um processo por *core* disponível nos nós computacionais.

Tal execução paralela envolve características que impactam o controle e coleta de dados: o indeterminismo da execução paralela, a aparição de anomalias,

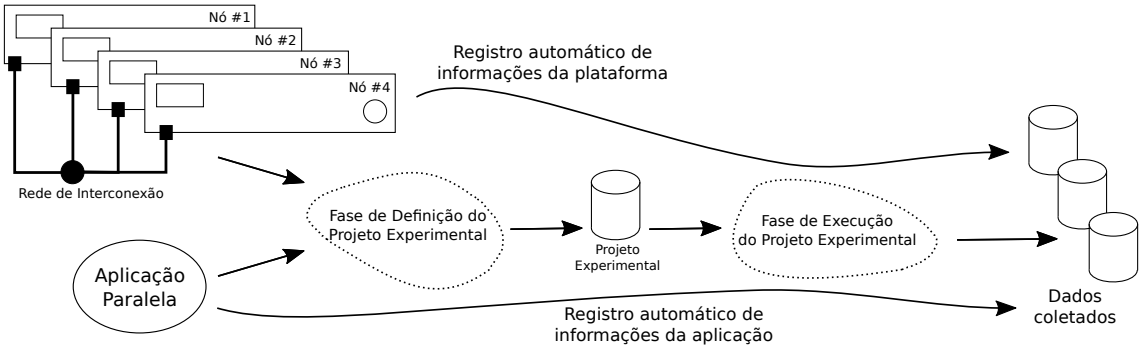

Figura 1.1. Panorama geral do controle e coleta em experimentos computacionais: um *cluster* de computadores com quatro nós e sua rede de interconexão combinado com uma aplicação paralela são sujeitos da definição de um projeto experimental que depois é executado para se coletar os dados para análise.

e a complexidade do sistema computacional. É natural a existência do **indeterminismo** na ordem que as operações são de fato executadas, devido ao caráter concorrente da execução paralela. O aparecimento de **anomalias** inesperadas, em qualquer camada do sistema computacional, pode causar tempos maiores na execução de uma determinada sequência de operações. Enfim, temos a **complexidade** do sistema computacional, com inúmeras camadas em nível de hardware e software. Essa complexidade torna difícil considerar todas as possíveis facetas configuráveis de um *cluster* de computadores.

A combinação dessas características aumenta a *variabilidade* dos experimentos computacionais. Ou seja, o efeito combinado do indeterminismo, da aparição de anomalias, da complexidade, torna o comportamento de qualquer experimento mais sujeito a variações nas medições. Um exemplo disso é a avaliação do tempo de execução de uma aplicação paralela: além de calcular a média de um determinado conjunto de execuções, o experimentador também calcula a variabilidade da média através de um intervalo de confiança calculado, por exemplo, a partir do desvio padrão. Quando maior a dispersão, menos confiável é a média e por consequência qualquer conclusão que possa se tirar do experimento. Qualquer ação do experimentador para reduzir tal dispersão é benéfico para melhor estudar determinado sistema computacional.

Das três características listadas, pouco pode ser feito em relação ao indeterminismo e ao aparecimento de anomalias. O indeterminismo é de certa forma desejado pois ele permite a execução concorrente, paralela, grande objetivo do processamento de alto desempenho. O aparecimento de anomalias inesperadas é natural em qualquer sistema computacional devido a grande quantidade de camadas de controle existentes, desde o baixo nível do hardware até a aplicação sendo executada. Enfim, para diminuir a variabilidade dos experimentos computacionais, resta controlar manualmente a maior quantidade de configurações possíveis do sistema computacional, diminuindo a sua complexidade.

Esta seção apresenta um resumo de conceitos e boas práticas para controlar um sistema computacional e obter medidas mais significativas. A Seção 1.2.1

apresenta conceitos a respeito da metodologia experimental separada em duas fases. A Seção 1.2.2 apresenta uma *checklist* com boas práticas para controle da complexidade de sistemas computacionais. A Seção 1.2.3 apresenta uma discussão e formas de registrar informações sobre a plataforma e ambiente de execução automaticamente. A Seção 1.2.4 apresenta ferramentas para instalação de dependências para a pilha de software sobre o sistema operacional. A Seção 1.2.5 lista ferramentas de virtualização através de *containers* do Linux para controlar também o sistema operacional. A Seção 1.2.6 apresenta formas de integrar todo o mecanismo de controle e coleta em *scripts* para gerenciadores de *jobs* tais como Slurm [15] e OAR [3]. Enfim, a Seção 1.2.7 apresenta um estudo de caso que mostra como tais conceitos e práticas podem ser operacionalizados.

1.2.1. Metodologia experimental

Segundo Jain [6], um experimento computacional se inicia através da definição de um projeto experimental. Ele deve ser constituído levando-se em conta os objetivos da investigação, definindo quais são as variáveis de controle – os **fatores** – e quais são as **variáveis de resposta**, ou seja, o que será observado. O objetivo é entender como os diferentes valores dos fatores – os **níveis** –influenciam a resposta. Como exemplo, podemos utilizar uma aplicação paralela. Uma variável de resposta pode ser simplesmente o tempo de execução ou a aceleração obtida com a paralelização. Como fatores, podemos considerar que o número de processos (seguindo a quantidade de núcleos de processamento), a quantidade de nós computacionais (de acordo com a disponibilidade do *cluster*), a frequência do processador (na praia de valores aceita pelo hardware) e a rede de interconexão (configurações alternativas de largura de banda) podem ter uma influência direta nas variáveis de resposta.

Existem vários tipos de projetos experimentais. Na sua versão mais simples, um projeto é capaz de estudar o impacto dos valores de um único fator, sendo que os valores dos demais fatores se mantém fixos. Tal tipo de projeto não permite o estudo da interação que possa existir entre os fatores. No exemplo anterior, seria inviável estudar a relação entre a quantidade de processos e a rede de interconexão. Tais fatores tem possivelmente uma relação devido a contenção da rede, mais fácil de ser atingida com um número maior de processos comunicantes. Por outro lado, o exemplo mais representativo de um projeto experimental mais complexo é o fatorial completo. Ele permite estudar a influência de todas as combinações de valores de todos os fatores nas variáveis de resposta. Tal projeto é bastante caro de ser executado, pois sua natureza combinatória o torna proibitivo de ser executado com um número já moderado de valores e fatores. Um exemplo intermediário é o projeto fatorial fracionário, onde alguns fatores se mantém fixos enquanto os demais são estudados com todas as combinações. A escolha do projeto experimental depende do recurso de tempo e de plataforma que se deseja investir para entender o fenômeno que se estuda.

A esquerda da Figura 1.1 ilustra a fase de *definição do projeto experimental* culminando na definição do **Projeto Experimental** no centro da imagem. Na prática, este projeto experimental pode consistir em uma tabela onde as colunas

representam os fatores, e cada linha representa uma determinada configuração a ser executada na plataforma. Os valores das células nas colunas representam os níveis dos fatores que devem ser adotados por aquela execução específica. A ordem aleatória dos itens do projeto experimental é fundamental, pois permite absorver anomalias inesperadas durante a execução da bateria experimental.

Definido o projeto experimental, passa-se a fase de *execução do projeto experimental*, como ilustrado na direita da Figura 1.1. Essa fase pode ser representada através de programa de computador (idealmente escrito em linguagem de *script)* que lê o projeto experimental e executa a aplicação paralela na plataforma alvo de acordo com os valores de fatores preestabelecidos. É portanto fundamental que tal script tenha controle das configurações da plataforma e da aplicação. Embora existem arcabouços que possam tornar genérica tal fase de execução, frequentemente são construídos procedimentos específicos para cada combinação de plataforma e aplicação, dada a especialização obrigatória desta fase. Um conjunto de dados observados, incluindo as variáveis de resposta, é registrado em arquivos de dados. Tais arquivos contém também todas as informações de variáveis não controladas e configurações de sistema.

1.2.2. Boas práticas para controle da complexidade

Como anteriormente discutido, o controle da complexidade da plataforma computacional permite diminuir a variabilidade dos fenômenos sendo estudados. Esse controle visa a reduzir a quantidade de variáveis controláveis, fixando e registrando suas configurações para valores conhecidos de forma que possam ser utilizados mais tarde para a análise dos dados. Quais configurações devem ser realizadas depende bastante de qual tipo de experimento está se conduzido. A listagem a seguir não é exaustiva e se propõem a simplesmente dar uma noção de quais configurações são úteis em determinados contextos.

- □ Vinculação (*binding*) fixa de fluxos de execução (*threads*), permite evitar a migração automática pelos algoritmos de balanceamento de carga embutidos no sistema operacional. Embora esses algoritmos tenham sido concebidos para eventualmente melhorar o desempenho, a migração de *threads* acontece de maneira não explícita, ou seja, a aplicação não fica sabendo e é relativamente difícil rastrear em qual núcleo de processamento (*core*) ela efetivamente está sendo executada em cada intervalo de tempo.

- □ Controle de frequência dos núcleos de processamento (*cores*) do processador, permite evitar que o HW ou o SW (neste caso, o sistema operacional), realize mudanças da frequência de processamento. Esse tipo de controle pode ser executado através da fixação de uma política de frequência por usuário, estabelecendo o uso da frequência máxima. Deve-se ter atenção ao fato que o HW, por possuir diversos elementos fechados, pode adotar uma política de frequência inadvertidamente.

- □ Desativar *turboboost* (em processadores Intel) pois este faz com que, sob altas demandas de processamento, a frequência seja a máxima possível para aquele

processador. Como a ativação deste recurso é de maneira não transparente ao sistema operacional ou à aplicação, cabe desativá-lo para evitar que tal variabilidade afete o entendimento dos fenômenos sendo investigados.

- □ Desativar *hyperthreading* (em processadores Intel), ou seja, desativar os núcleos de processamento lógicos, tendo em vista que seus recursos são mais limitados que os núcleos físicos (*cores*). Esse tipo de recurso computacional, por mais que dobre a quantidade de *cores* visíveis em nível de sistema operacional, aumenta a variabilidade experimental. Isso acontece principalmente em aplicações limitadas pela CPU, embora aplicações limitadas pelo acesso à memória possam ter algum ganho de desempenho.
- □ Detectar a configuração NUMA do nó computacional e estabelecer uma política fixa de distribuição de fluxos de execução, em processadores com múltiplos processadores.
- □ Configurar uma política TCP/Ethernet adequada para a rede de interconexão, sabendo que por *default* o *kernel* do Linux vem configurado com tamanhos de pacote e outras configurações relacionadas específicas para redes 100 Mbit Ethernet. Esse tipo de configuração pode impactar negativa nas redes de interconexão de alto desempenho tais como 10 GBit Ethernet ou Infiniband.

Outras informações, incluindo outras configurações passíveis de verificação específicas para o sistema operacional Linux, podem ser obtidas em um trabalho relacionado [14].

1.2.3. Registro automático de informações sobre a plataforma

Usualmente os usuários registram manualmente informações sobre a plataforma na qual os experimentos estão sendo executados. Tais informações, de forma geral, compreendem apenas características básicas do *hardware*, tais como modelo da CPU, tamanho da memória e tipo da interface de rede, e do software como sistema operacional, versão do compilador e distribuição MPI. Além de ser pouco prática, tal estratégia pode incorrer em informações incompletas e até mesmo incorretas. Podemos imaginar uma situação na qual dados referentes a CPU são coletados antes do início do experimento, neste momento a CPU está operando com uma frequência de 1200 MHz, entretanto o controle de frequência (`governor`) está configurado na opção `ondemand`, o que provavelmente fará com que, durante a execução do experimento, o processador passe a operar em uma frequência bem mais alta (e.g., 2300 MHz).

Para evitar tais situações, é conveniente adotar uma estratégia de registro automático de informações sobre a plataforma. É recomendável que estas informações sejam coletadas toda vez que uma execução for realizada e que sejam armazenadas juntamente com os resultados. Para coletar informações sobre o *hardware*, podemos partir do seguinte conjunto de ferramentas, assumindo um ambiente baseado em Linux. Cabe ressaltar que, além dos comandos abaixo, outros específicos podem ser necessários caso os experimentos envolvam outros recursos de *hardware* como GPUs, FPGAs ou interfaces de rede proprietárias.

Sistema operacional, topologia de hardware e frequência do processador (HW)

lstopo – fornecido pela ferramenta `hwloc`, permite obter a topologia do sistema, incluindo hierarquia de memória cache, nós NUMA, núcleos físicos e lógicos bem como dispositivos PCI conectados. **cpufreq-info** – fornecido pela ferramenta `cpufrequtils`, permite obter a frequência atual, mínima e máxima para cada núcleo do processador, de maneira genérica independente do fabricante do processador. Informações sobre a política de controle de frequência atual (`governor`) e as demais disponíveis também podem ser obtidas. **pstate driver** – trata-se de um módulo de *kernel* específico para processadores Intel com funcionalidade semelhante àquela fornecida pelo cpufreq. **lspci** – lista todos os dispositivos PCI conectados ao sistema. **ifconfig** (ou **ip** em um Linux moderno) – este comando permite obter as configurações da interface de rede.

Informações associadas à aplicação paralela (SW)

Quanto ao software além das informações básicas como versão do sistema operacional e do compilador, pode-se obter algumas informações adicionais com os seguintes comandos. **ompi-info** – assumindo que a aplicação paralela faz uso da implementação OpenMPI da especificação MPI, este comando permite listar todas as configurações que controlam o comportamento interno da implementação, como *buffers* e protocolos de envio/recepção. **ldd** – mostra as bibliotecas compartilhadas requeridas por um executável e o onde elas se encontram (`PATH`) na árvore de diretórios. **env** (assumindo um *shell* baseado em `sh`) – lista as variáveis de ambiente no *shell* corrente. **nm** – lista todos os símbolos de arquivos objeto, um programa útil para se utilizar como informação de assinatura de binários executáveis.

1.2.4. Ferramentas para instalação de dependências

Aplicações paralelas que executam em *clusters* de alto desempenho frequentemente apresentam uma pilha de software extensa, incluindo diversos níveis de dependências e parâmetros opcionais. Dessa forma, a configuração do ambiente experimental implica em obter, compilar e ligar dezenas de bibliotecas. Tal cenário, motiva a utilização de gerenciadores de pacotes. Entretanto, em ambientes compartilhados como *clusters*, não é viável que os usuários tenham permissão para utilizar o gerenciador de pacotes do sistema (e.g. `dpkg`, `apt`, `rpm`).

Spack [5] é um gerenciador de pacotes que permite aos usuários obter, compilar e instalar programas e bibliotecas em seus próprios diretórios sem fazer uso de privilégios de administrador nem de comandos específicos do sistema operacional. Em oposição a gerenciadores similares de uso geral como o `homebrew`, o Spack oferece funcionalidades específicas para ambientes de computação de alto desempenho, entre elas, configurações e dependências personalizadas, instalações não-destrutivas e coexistência de múltiplas instalações. Tais funcionalidades permitem testar e avaliar uma ampla gama de configurações. Os comandos abaixo, ilustram como o Spack pode ser usado para gerenciar diferen-

tes configurações da biblioteca `Boost` que podem coexistir simultaneamente na mesma árvore de diretórios.

- Configuração da biblioteca Boost na versão 1.69.0 com compilador padrão (`gcc`) ligado com a distribuição OpenMPI para prover suporte à interface MPI:

```
spack install -v boost@1.69.0+mpi^openmpi
```

- Configuração da biblioteca Boost na versão 1.68.0 com compilador padrão (`gcc`) ligado com a distribuição MPICH para prover suporte à interface MPI:

```
spack install -v boost@1.68.0+mpi^mpich
```

- Configuração da biblioteca Boost na versão 1.69.0 com compilador `clang` ligado com a distribuição OpenMPI com compilador `gcc` para prover suporte à interface MPI:

```
spack install -v boost@1.69.0+mpi%clang^openmpi%gcc
```

1.2.5. Controle em nível de sistema operacional

Embora o Spack seja uma ferramenta que permita um controle preciso da instalação de bibliotecas e arcabouços, ele não é capaz de gerenciar totalmente a pilha de software. Por exemplo, a forma como as chamadas de sistema são realizadas no sistema operacional, tanto em nível de usuário (através da `libc`), quando em nível de superusuário, se mantém sem controle. Existem alternativas para controlar também o sistema computacional, através de métodos nativos ou virtualizados.

Métodos nativos exigem algum suporte de hardware, tal como a necessidade de gerenciar e utilizar perfis PXE em/de servidores TFTP, disparar comandos de reinicialização através de IPMI ou uma PDU gerenciável, etc. Ferramentas como Kadeploy3 [7] utilizam tal infraestrutura de hardware para manter em cada nó computacional um sistema operacional principal em uma partição, ao mesmo tempo que possibilita a instalação completa de outros sistemas operacionais em outras partições. O usuário do cluster pode então instalar seu próprio sistema operacional em todos os nós gerenciados por Kadeploy3, se tornando superusuário, com controle completo da pilha de software.

Métodos virtualizados, principalmente aqueles baseados em *containers* Linux, permitem obter o mesmo tipo de controle sem a necessidade de reinicializar a máquina ou de se tornar superusuário. Não exigem também nenhum tipo de hardware específico pois são baseados em recursos do sistema operacional Linux. Exemplos de ferramentas que permitem essa alternativa incluem CharlieCLoud [10] ou Singularity [9]. Estudos [1] baseados em *containers* identificaram que esse tipo de controle impacta minimamente o desempenho de aplicações paralelas quando estas são executadas nativamente.

1.2.6. Integração com gerenciadores de *jobs*

Os nós computacionais de clusters de computadores são frequentemente gerenciados por ferramentas específicas que permitem a alocação e reserva desses nós. Ferramentas como Slurm [15] e OAR [3] fazem este ofício. O usuário do cluster que pretende executar uma aplicação paralela codifica um *script* que contém informações essenciais para a alocação como o tempo de alocação pedido, a quantidade de nós, qual tipo de recurso, etc.

O restante do *script* é executado no *frontend* do cluster de maneira que ele pode ser utilizado para implementar o controle dos nós computacionais (conforme a listagem discutida na Seção 1.2.2), realizar o registro automático de informações sobre a plataforma (Seção 1.2.3), adoção de uma versão específica da pilha de software (Seções 1.2.4 e 1.2.5) e, enfim, a execução das diretivas impostas pelo projeto experimental previamente preparado. Essas diretivas incluem mudanças de configuração em nível de usuário ou do sistema operacional, como troca de frequência, limitações de memória, de banda passante da rede, ou de qualquer outro parâmetro que esteja sendo estudado pelo experimentador. Tais mudanças devem ser executadas no próprio *script* de execução do experimento, de forma automática. Recomenda-se inclusive que após a mudança das configurações, essas sejam reobtidas para que fiquem registrados possíveis erros de parametrização. Isso é importante para, antes de iniciar o processo de análise, confirmar que todos os valores de fatores foram de fato aplicados conforme dito no projeto experimental.

O *script* que realiza a reserva pode, no final, concentrar todas as informações e medidas realizadas em um único diretório, que é independente. Os *logs* de execução registrados pelo gerenciador de recursos pode ficar no próprio diretório que contém as outras informações do experimento. Assim, o processo de análise pode ser padronizado para aqueles dados.

O *script* da Listagem 1.1 traz um exemplo que resume as atividades a serem realizadas de maneira integrada para executar o projeto experimental. Sendo não exaustiva, esse exemplo traz as quatro principais etapas. Cada uma dessas etapas é implementada através de códigos *bash* ou outra linguagem de *script* (como *python*, por exemplo). Recomenda-se enfim que este *script* seja arquivado junto com as demais informações coletadas, de maneira que o analista dos dados saiba em que momento as informações foram coletadas.

```
#!/bin/bash
#SBATCH --nodes=16
#SBATCH --time=02:00:00
#SBATCH --partition=gppd-hpc

# 1. Controle inicial dos nós computacionais (HW e SW)
# 2. Registro das condições iniciais
# 3. Ler o projeto experimental, e para cada experimento
# 3.1 Aplicar os parâmetros (fatores e valores) específicos
```

```
# 3.2 Registro das condições iniciais do experimento
# 3.3 Executar o experimento
# 3.4 Coletar os dados do experimento em um diretório
# 4. Centralizar os dados em um único diretório
```

Listagem 1.1. Exemplo de script Slurm para execução de um projeto experimental.

1.2.7. Estudo de caso com um estudo de balanceamento de carga no Alya

Para ilustrar como um experimento de coleta de dados é realizada na prática, utilizaremos um estudo de caso baseado no uso de *Space Filling Curves* (SFC) para melhorar o balanceamento de carga da aplicação Alya [2]. Neste estudo, foi investigado se o particionamento obtido com uma técnica simples e iterativa do tipo SFC pode gerar um balanceamento de carga computacional melhor que uma abordagem via particionamento Metis para malhas grandes. A vantagem de SFC é que o particionamento é significativamente mais rápido que via Metis. A plataforma alvo foi o MareNostrum4 (MN4), do Barcelona Supercomputing Center.

O projeto experimental envolvia fatores com um único nível, ou seja, variáveis de entrada com configurações fixas para servir de base da análise. Elas incluem a entrada (a malha com o problema a ser utilizado como base), a quantidade de passos de tempo de simulação, a quantidade de passos de balanceamento de carga, e a quantidade de processos e de nós computacionais. Essas configurações fixas fazem parte do início do *script* Slurm, ilustrado na Listagem 1.2, onde o nome da tarefa é `EXP20`, a quantidade de nós é 16 e a quantidade de processos é 768, sendo que duas horas são solicitadas para a execução do experimento. Além desses parâmetros, também são definidas as saídas padrão (`--output`) e de erro (`--error`) seguindo a parametrização do Slurm. Os demais parâmetros que serão utilizados no *script* são definidos como variáveis de ambiente.

```
#!/bin/bash
#SBATCH --job-name="EXP20"
#SBATCH --output=%x_%j.out
#SBATCH --error=%x_%j.err
#SBATCH --nodes=16
#SBATCH --ntasks=768
#SBATCH --time=02:00:00

export CASENAME=fensap    # A entrada que será utilizada
export TIMESTEPS=5        # Quantidade de passos de simulação
export STEPS=20           # Quantidade de passos de balanceamento
export NP=${SLURM_NTASKS} # Quantidade de processos

# continua...
```

Listagem 1.2. Configurações iniciais em exemplo para experimento `EXP20` com Alya.

A Listagem 1.3 apresenta a parte principal da execução do projeto experimental, guiada pelo laço principal de passos de balanceamento. Em cada laço, é gerado uma chave única (`RUNKEY`) que identificará a integralidade dos dados gerados naquele experimento. Em seguida, as chamadas para o programa `module`

são responsáveis por descarregar e carregar as bibliotecas necessárias (um recurso semelhante aquele disponibilizado pelo Spack). O comando `mpirun` é então utilizado para executar a aplicação nos 16 nós computacionais, lançando 768 processos (48 por nó, sendo que cada nó tem dois processadores cada um com 24 *cores* físicos). O parâmetro `-x` fornecido ao `mpirun` serve para repassar aquelas variáveis de ambiente para todos os nós computacionais, tendo em vista que as variáveis que começam por `SCOREP` servem para controlar o registro de informações importantes da execução. Os demais parâmetros para o comando MPI servem para configurar aspectos relacionados a vinculação dos processos aos *cores* e relatar qual vinculação foi executada, de forma que isso possa ser considerado no processo de análise posterior. Antes do final do laço, e após a execução da aplicação, os dados são centralizados no diretório do experimento e em seguida movidos para o diretório final que contém o resultado de todas as execuções.

```
# ... continuação

# Diretório geral para contar todos os resultados
export EXPEDIR=EXP20
rm -rf $EXPEDIR; mkdir -p $EXPEDIR

for RUN in $(seq 1 ${STEPS}); do
   RUNKEY="${SLURM_JOB_NAME}_${SLURM_JOB_ID}_${NP}_STEP_${RUN}_of_${STEPS}"
   rm -rf $RUNKEY; mkdir -p $RUNKEY

   module unload mkl
   module unload intel
   module unload impi
   module load gcc/7.2.0
   module load openmpi/3.0.0

   $(which mpirun) \
        --mca btl_base_warn_component_unused 0 \
        --bind-to core:overload-allowed \
        --report-bindings \
        -x SCOREP_TOTAL_MEMORY=3900MB \
        -x SCOREP_MPI_ENABLE_GROUPS=ALL \
        -x SCOREP_ENABLE_TRACING=FALSE \
        -x SCOREP_ENABLE_PROFILING=TRUE \
        -x SCOREP_EXPERIMENT_DIRECTORY=$SCOREPDIR \
        $ALYA $CASENAME

   # Copiar todos os arquivos registrados para o diretório $RUNKEY

   # Mover o diretório deste experimento para o diretório geral
   mv $RUNKEY $EXPEDIR
done
```

Listagem 1.3. Parte central do experimento `EXP20` com Alya.

1.3. Análise de dados

A etapa de análise de dados envolve a fase pós-execução do experimento. Usualmente, esta etapa é executada no computador pessoal do usuário que, em geral, não é a mesma plataforma na qual os experimentos foram executados. A análise de dados consiste em um processo iterativo e reflexivo, no qual o usuário parte de uma análise em alto nível dos dados inicialmente coletados e a partir desta aprofunda-se em pontos específicos. Frequentemente, a análise de dados

desenrola-se de maneira iterativa, onde uma análise anterior permite identificar e delimitar cenários e configurações que alimentam uma nova execução da etapa de controle e coleta como detalhado na Seção 1.2. Esta nova execução gerará mais dados, que implicarão em uma nova iteração do processo de análise.

As características do processo de análise de dados motivam a adoção de uma estratégia sistematizada, que permita, facilmente, reexecutar algumas etapas do processo de análise bem como revisar o fluxo de experimentos, reflexões e conclusões que levou a uma determinada suposição ou resultado. A adoção de tal estratégia é benéfica, não apenas ao usuário durante o desenvolvimento do trabalho, mas também às outras partes envolvidas no processo científico e que não, necessariamente, estarão próximas temporal ou fisicamente do usuário, tais como orientadores, revisores de publicações, autores de trabalhos relacionados ou até mesmo o próprio autor em momento futuro. Dessa forma, esta seção tem por objetivo ilustrar ferramentas e conceitos que permitam sistematizar e disponibilizar uma análise de desempenho.

O restante desta seção apresentará conceitos e ferramentas que podem ser empregados no desenvolvimento de uma análise de dados reprodutível. A Seção 1.3.1 ilustra como a programação literária pode ser usada no processo de análise de dados experimentais. Já a Seção 1.3.2 discute conceitos relacionados a reprodutibilidade da análise de desempenho em si tais como formato e plataforma de distribuição.

1.3.1. Programação Literária

A Programação Literária proposta por Donald Knuth [8] tem por base duas operações (*weave* e *tangle*) que permitem converter um documento fonte em duas representações distintas, um formato legível para humanos e outro apto para execução em computadores. Esta funcionalidade é bastante útil na análise de resultados experimentais pois permite manter em um mesmo documento tanto as anotações preliminares como expectativas, suposições e reflexões acerca do experimento quanto os comandos usados para colocá-lo em execução e posteriormente processar e visualizar seus resultados.

A extensão Org-mode [4] do editor de texto Emacs [13] oferece, entre outros recursos, funcionalidades de programação literária. Arquivos criados com esta extensão (arquivos `org`) são arquivos em texto puro que podem ser abertos e lidos em qualquer editor de texto, embora seja conveniente o uso do editor Emacs para aproveitamento de todas as funcionalidades. O pacote `Babel` possibilita a definição de blocos ativos de código e de dados dentro de documentos `org`, tais blocos podem ser avaliados e a saída correspondente é capturada e incluída no documento. Os blocos podem ser escritos em diferentes linguagens de programação, e podem ser encadeados de forma que os dados de saída produzidos por um bloco sejam usados como entrada de outro. Os trechos de código abaixo ilustram o comportamento desta funcionalidade. O bloco a seguir é escrito em `shell script`, com a possível saída produzida representa na tabela abaixo.

```
for n in `seq 5`; do printf "%d $RANDOM \n" $n ; done
```

Tabela 1.1. Possível saída produzida pelo trecho de código em `shell script`

1	21020
2	20873
3	7597
4	19882
5	30785

A avaliação do trecho de código acima produzirá uma saída, que será encadeada como entrada do código abaixo escrito na linguagem R. Quando avaliado, o código abaixo produzirá como saída, uma imagem contendo um gráfico construído a partir dos dados gerados pelo primeiro trecho de código.

```
library(ggplot2)
library(tidyverse)
dados %>%
    ggplot(aes(V1, V2)) +
    geom_point() +
    theme_bw()
```

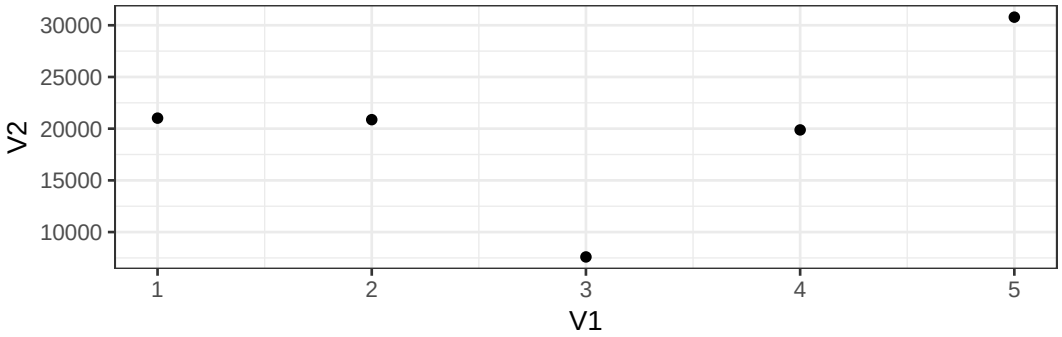

Figura 1.2. Gráfico gerado pelo código `R` utilizando a saída do código `shell` como entrada

Um mesmo conjunto de dados pode ser usado inúmeras vezes como entrada para blocos de código diversos. Além do gráfico da Figura 1.2, podemos usar os dados da Tabela 1.1 para calcular valores estatísticos como mínimo, máximo, média, mediana e quartis. O trecho de código abaixo ilustra o comando em R que permite a obtenção destas informações.

```
dados$V2 %>% summary()
```

```
Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
7597   19882   20873   20031   21020   30785
```

A programação literária, por si só, já é uma prática aconselhada para facilitar o registro e análise de resultados. Entretanto, ela não garante que os gráficos gerados na análise em questão sejam claros e diretos. O resultado do processo de criação de gráficos pode ser aprimorado com a aplicação de alguns passos de verificação e controle [6, 12] apresentados na Tabela 1.2.

Tabela 1.2: *Checklist* para gráficos

Dados	✓	O tipo do gráfico é adequado para a natureza do dado (curva, barras, setores, histograma, nuvem de pontos, etc)
	✓	As aproximações/interpolações fazem sentido
	✓	As curvas são definidas com um número suficiente de pontos
	✓	O método de construção da curva é claro: interpolação (linear, polinomial, regressão, etc)
	✓	Os intervalos de confiança são visualizados (ou informados separadamente)
	✓	Os passos do histograma são adequados
	✓	Histogramas visualizam probabilidades (de 0 a 1)
Objetos Gráficos	✓	Os objetos gráficos são legíveis na tela, na versão impressa (P&B), em vídeo, etc
	✓	O intervalo do gráfico é padrão, sem cores muito similares, sem verde (vídeo)
	✓	Os eixos do gráfico estão claramente identificados e rotulados
	✓	Escalas e unidades estão explícitas
	✓	As curvas se cruzam sem ambiguidade
	✓	As grades ajudam o leitor
Anotações	✓	Eixos são rotulados por quantidades
	✓	Rótulos dos eixos são claros e autocontidos
	✓	Unidades estão indicadas nos eixos
	✓	Eixos são orientados da esquerda para a direita e de baixo para cima
	✓	Origem é (0,0), caso contrário deve estar claramente justificada
	✓	Sem buracos nos eixos
Anotações (2)	✓	Para gráficos de barras/histogramas a ordem das barras segue a ordenação clássica (alfabética, temporal, do melhor pro pior)
	✓	Cada curva tem uma legenda
	✓	Cada barra tem uma legenda
Informação	✓	Curvas estão na mesma escala

Continued on next page

Continued from previous page

	✓	Histogramas visualizam probabilidades (de 0 a 1)
	✓	O número de curvas em um mesmo gráfico é pequeno (menor que 6)
	✓	Compare as curvas no mesmo gráfico
	✓	Uma curva não pode ser removida sem redução de informação
	✓	O gráfico fornece informações relevantes ao leitor
	✓	Se o eixo vertical mostra médias, as barras de erro devem estar presentes
	✓	Não é possível remover qualquer objeto sem modificar a legibilidade do gráfico
Contexto	✓	Todos os símbolos são definidos e referenciados no texto
	✓	O gráfico produz mais informação que qualquer outra representação (escolha da variável)
	✓	O gráfico tem um título
	✓	O título é suficientemente autocontido para a compreensão parcial do gráfico
	✓	O gráfico é referenciado no texto
	✓	O texto comenta a figura
	✓	**A representação gráfica deve ser elegante**

Ao aplicarmos as orientações da Tabela 1.2 à Figura 1.2, notamos que o gráfico em questão pode ser aprimorado. Inicialmente, devemos adicionar rótulos aos eixos vertical e horizontal. Dada a natureza dos dados não há unidades a serem indicadas nos mesmos. Em seguida, adicionamos um título ao gráfico, e por fim verificamos que a origem deve ser o ponto (0,1) e não (0,0) pois as observações foram numeradas a partir de 1. O código R abaixo produz a Figura 1.3 que contém a versão aprimorada do gráfico.

```
library(ggplot2)
library(tidyverse)

dados %>%
    ggplot(aes(V1, V2)) +
    theme_bw() +
    geom_point() +
    ylab("Valor Aleatório") +
    xlab("Observação") +
    ggtitle("Geração de Números Aleatórios em shell script") +
    lims(y = c(0, NA), x = c(1, NA))
```

1.3.2. Reprodutibilidade da análise de desempenho

Demonstrabilidade e reprodutibilidade são conceitos-chave no método científico. Entretanto, frequentemente, tais processos ficam limitados ou comprometidos

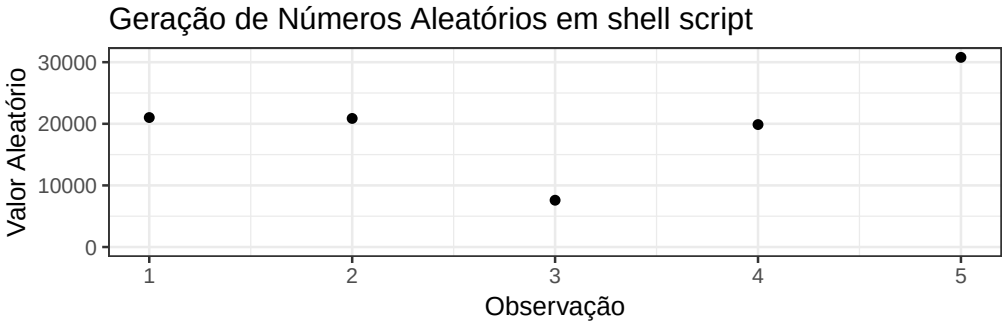

Figura 1.3. Gráfico gerado pelo código `R` utilizando a saída do código `shell` como entrada (versão aprimorada)

devido à falta de informações além do texto científico. No contexto da computação, e em especial da área de análise de desempenho, a disponibilização do código fonte e dos dados de entrada e saída são essenciais para permitir a reprodutibilidade dos experimentos.

Existem dois aspectos principais que devem ser considerados na disponibilização de anexos de publicações científicas. O primeiro deles se refere ao formato utilizado. Tal deve ser aberto e de estrutura simples. O formato CSV, por exemplo, é um formato adequado para disponibilização de resultados numéricos brutos, pois é simples e de fácil leitura tanto por seres humanos quanto por ferramentas automatizadas. Para disponibilização de resultados qualitativos, que incluam não apenas os dados brutos mas também análises e reflexões, pode-se usar formatos que ofereçam alguma estrutura hierárquica e permitam criar uma espécie de caderno de laboratório, tais como Org-mode (apresentado na Subseção 1.3.1), R Markdown ou IPython.

O segundo aspecto refere-se à plataforma utilizada para disponibilização dos dados. Lamentavelmente, os repositórios de textos científicos, tais como IEEE Xplore[3], ACM DL[4] e Portal de Conteúdo da SBC[5], ainda não oferecem espaço para armazenamento de anexos de artigos. Idealmente, esses dados deveriam estar disponíveis juntamente com o texto científico.

A alternativa passa a ser a disponibilização do material complementar em outras plataformas não necessariamente científicas, incluindo, no texto científico, uma referência (*link*) para o material. Neste caso, os principais pontos a serem considerados são a livre acessibilidade, a perenidade, versionamento e o espaço disponível. Embora de fácil acesso, soluções baseadas em computação em nuvem como Dropbox, Onedrive e Google Drive tendem a ser limitadas em termos de perenidade, versionamento e espaço disponível. O uso de páginas pessoais em servidores institucionais tende a ser mais flexível, porém está sujeito a política de cada instituição. Plataformas de hospedagem e versionamento de código

[3] `https://ieeexplore.ieee.org/`
[4] `https://dl.acm.org`
[5] `https://portaldeconteudo.sbc.org.br/`

fonte como GitHub[6], Bitbucket[7] e GitLab[8] são boas opções em termos de acessibilidade e versionamento porém implicam restrições quando é necessário armazenar dados não-textuais ou ainda em grande volume. Por fim, pode-se citar plataformas voltadas para armazenamento de dados científicos como o Figshare[9] e o Zenodo[10]. Estas plataformas permitem o armazenamento de qualquer formato de arquivo com poucas restrições de tamanho. Cada registro recebe um identificador DOI, o que facilita a busca e a citação dos conjuntos de dados. A principal limitação dessas plataformas está relacionada a impossibilidade de corrigir ou apagar registros já publicados, o que pode ser um limitante para trabalhos em andamento ou sob revisão. Tal limitação, entretanto, pode ser contornada por meio da integração nativa com plataformas como GitHub, o que facilita a publicação de *releases* ou de resultados consolidados.

1.4. Conclusão

Este minicurso sensibiliza os participantes da importância do emprego de boas práticas na realização de experimentos computacionais na área de processamento paralelo de alto desempenho. Após uma breve motivação, o minicurso se divide em duas partes, uma primeira que trata dos procedimentos de controle e coleta de dados experimentais, seguindo de uma segunda parte que trata da análise dos dados de maneira reprodutível.

As boas práticas apresentadas neste minicurso não são exaustivas. O enfoque dado foi em experimentos computacionais limitados pela CPU, levando a verificações relacionadas a vinculação de fluxos de execução aos núcleos de processamento, por exemplo. Caso os experimentos tenham um enfoque na rede, em entrada/saída (disco), em memória `RAM`, em uso de GPUs, ou algum outro aspecto computacional, novas diretivas de controle e verificação devem ser concebidas. Essas novas diretivas podem envolver também elementos de software (bibliotecas, arcabouços, *middlewares*). De maneira colaborativa foi instituído um repositório intitulado "Good Practices for Computational Experiments in High Performance Clusters"[11] para organizar tais diretivas.

Enfim, lembramos que qualquer decisão experimental requer claramente um ponto de vista crítico no seu emprego. Cada experimento tem suas particularidades que devem ser levadas em conta na hora de escolher quais tipos de controle devem ser executados antes, durante o experimento. Espera-se mesmo assim que o texto deste minicurso ressalte a importância de procedimentos sistemáticos na condução do experimentos computacionais, de forma a culminar em resultados credíveis.

---

[6] `http://github.com/`
[7] `https://bitbucket.org/`
[8] `https://gitlab.com/`
[9] `http://figshare.com/`
[10] `https://zenodo.org/`
[11] `https://gitlab.com/schnorr/experiments`

Referências

[1] Guilherme Alles, Lucas Mello Schnorr, and Alexandre Carissimi. Assessing the computation and communication overhead of linux containers for hpc applications. In *Anais do Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*. Sociedade Brasileira de Computação, 2018.

[2] R. Borrell, J.C. Cajas, L. Schnorr, A. Legrand, and G. Houzeaux. SFC based multi-partitioning for accurate load balancing of CFD simulations. In *Tenth International Conference on Computational Fluid Dynamics (ICCFD10)*, Barcelona, 2018.

[3] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard. A batch scheduler with high level components. In *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, volume 2, pages 776–783. IEEE, 2005.

[4] Carsten Dominik. *The Org Mode 7 Reference Manual - Organize Your Life with GNU Emacs*. Network Theory Ltd., 2010.

[5] Todd Gamblin, Matthew LeGendre, Michael R Collette, Gregory L Lee, Adam Moody, Bronis R de Supinski, and Scott Futral. The spack package manager: Bringing order to hpc software chaos. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pages 1–12. IEEE, 2015.

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

[7] Emmanuel Jeanvoine, Luc Sarzyniec, and Lucas Nussbaum. Kadeploy3: Efficient and Scalable Operating System Provisioning. *USENIX ;login:*, 38(1):38–44, February 2013.

[8] D. E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 2 1984.

[9] Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer. Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5):e0177459, 2017.

[10] Reid Priedhorsky and Tim Randles. Charliecloud: Unprivileged containers for user-defined software stacks in hpc. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 36. ACM, 2017.

[11] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2018.

[12] Lucas Mello Schnorr and Jean-Marc Vincent. Literate Programming and Statistics (CMP595), 2018. `https://github.com/schnorr/lps`.

[13] Richard Stallman et al. *GNU Emacs Manual*. Free Software Foundation, Boston, USA, 17 edition, 2017.

[14] Luka Stanisic, Lucas Mello Schnorr, Augustin Degomme, Franz Heinrich, Arnaud Legrand, and Brice Videau. 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, June 2017.

[15] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.