Minicursos do SSCAD 2024

XXV Simpósio em Sistemas Computacionais de Alto Desempenho

São Carlos – SP

23 a 25 de outubro de 2024

Porto Alegre

Sociedade Brasileira de Computação

2024

Dados Internacionais de Catalogação na Publicação (CIP)

S612 Simpósio em Sistemas Computacionais de Alto Desempenho (25. : 23 – 25 outubro 2024 : São Carlos)

Minicursos do SSCAD 2024 [recurso eletrônico] / coordenação: Arthur F. Lorenzon, Álvaro Luiz Fazenda. – Dados eletrônicos. – Porto Alegre: Sociedade Brasileira de Computação, 2024.

174 p. : il. : PDF; 7.1 MB

Modo de acesso: World Wide Web.
Inclui bibliografia
ISBN 978-85-7669-610-0 (e-book)

1. Computação – Brasil – Evento. 2. Sistemas computacionais. 3. Alto desempenho. I. Lorenzon, Arthur F. II. Fazenda, Álvaro Luiz. III. Sociedade Brasileira de Computação. VI. Título.

CDU 004(063)

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

Índices para catálogo sistemático:

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

Prefácio

Esta edição do Livro de Minicursos do SSCAD traz seis minicursos apresentados durante o XXV Simpósio em Sistemas Computacionais de Alto Desempenho, realizado entre os dias 23 e 25 de outubro de 2024 em São Carlos–SP. O primeiro capítulo versa sobre conceitos essenciais de processamento e análise de volumes massivos de dados, fazendo uso de algoritmos de aprendizado de máquina de forma prática na plataforma HPCC *(High Performance Computing Cluster)*. Já no segundo capítulo, é apresentada ao leitor a possibilidade de compreender o *Parallel Scalability Suite* para avaliar o comportamento de aplicações paralelas mediante perfilamento e visualização da escalabilidade. O terceiro capítulo, por sua vez, apresenta técnicas de programação paralela híbridas aderentes ao padrão MPI e OpenMP *Offloading*, com ênfase nos modelos de paralelismo em aceleradores. No quarto capítulo, conceitos fundamentais de simulação arquitetural com o simulador gem5 são introduzidos. O capítulo também explora como a simulação habilita os projetistas a explorar, verificar e otimizar arquiteturas através da modelagem de seu comportamento e interação com componentes chaves do sistema. Considerando os avanços da computação quântica, o capítulo 5 demonstra como desenvolver algoritmos para uma arquitetura de computação quântica através do kit de desenvolvimento usando o *IBM/Qiskit*. Por fim, o sexto capítulo estuda as definições sobre análise de desempenho, as principais técnicas utilizadas e algumas das ferramentas para analisar o desempenho de aplicações paralelas.

Esperamos que esse livro permita que usuários, entusiastas e pesquisadores da área de Alto Desempenho, tanto os que estiveram presente quanto os que não puderam participar do SSCAD 2024, possam aproveitar e desfrutar dos conhecimentos aqui apresentados.

Uma ótima leitura a todos.

Arthur F. Lorenzon (UFRGS) e Álvaro Luiz Fazenda (UNIFESP) - Coordenadores dos minicursos do SSCAD 2024

Capítulo

1

Processamento e análise de *Big Data* para aplicação de algoritmos de *Machine Learning* com interface *Myriad* através da utilização da plataforma *HPCC Systems*

Mauro Donato Marques (LexisNexis Risk Solutions)

Abstract

Throughout the mini-course, participants will have the opportunity to learn the essential concepts of processing and analyzing massive volumes of data (Big Data) and the process of developing a query service using the open-source platform High Performance Computing Cluster (HPCC Systems) and also, the application of Machine Learning algorithms with Myriad interface, as well as having the possibility to apply the knowledge acquired in a training environment made available in the classroom.

Resumo

Ao longo do minicurso, os participantes terão a oportunidade de conhecer os conceitos essenciais de processamento e análise de volumes massivos de dados (Big Data) e o processo de desenvolvimento de um serviço de consulta através da utilização da plataforma open-source composta por um Cluster Computacional de Alto Desempenho (HPCC Systems) e, também, a aplicação de algoritmos de Aprendizado de Máquina com interface Myriad, bem como terão a possibilidade de aplicar os conhecimentos adquiridos em um ambiente de treinamento disponibilizado em sala de aula.

Informações Técnicas: Curso de nível básico. O curso requer apenas um computador com acesso à Internet e uma conta no GitHub.

GitHub repositório: https://github.com/mauromarx/SSCAD_2024

1.1. A Plataforma HPCC Systems

HPCC Systems (High Performance Computing Cluster) é uma plataforma para solução de desafios de Big Data com as seguintes características:

- Supercomputação: processamento paralelo e dados distribuídos;
- Open source: código aberto e gratuita;
- Completa: gestão compreensiva e simplificada do fluxo de dados.

HPCC Systems oferece o melhor de ambos os mundos no que tange ao processamento e análise de volumes massivos de dados (Big Data), ou seja, combina o rápido desempenho de um "Data Warehouse" para a entrega de informações com a capacidade de tratar os dados como se estivessem em um "Data Lake". HPCC Systems usa arquitetura de dados distribuída e uma metodologia de processamento paralelo para trabalhar com grandes conjuntos de dados.

1.1.1. Um breve histórico

A plataforma de Big Data que se tornaria HPCC Systems foi desenvolvida em 2001 por uma equipe interna de engenharia da LexisNexis Risk Solutions. Os primórdios da base tecnológica para a plataforma HPCC Systems foi desenvolvida pela primeira vez na Seisint em 1999, a fim de gerenciar um número significativo de conjuntos de dados. Em 2000, a equipe da Seisint precisava coletar e analisar grandes quantidades de informações brutas de consumo de dados financeiros de uma ampla variedade de fontes para um cliente responsável por determinação da "score" de crédito ao consumidor nos Estados Unidos. Isto levou à criação da linguagem de programação, conhecida como ECL (Enterprise Control Language), que HPCC Systems usa atualmente. Em 2004, a LexisNexis Risk Solutions (LNRS) adquiriu Seisint junto com sua tecnologia, incluindo a linguagem ECL usada para programação em clusters Thor e Roxie. Durante esse período, a plataforma HPCC Systems era empregada, principalmente, como uma ferramenta interna da LexisNexis Risk Solutions, e ainda não havia atingido o seu pleno potencial. Em 2008, a LexisNexis Risk Solutions adquiriu a ChoicePoint, uma seguradora e provedora de análise e, nos próximos três anos, o portfólio de produtos da ChoicePoint foi integrado à plataforma HPCC Systems. Combinando a plataforma HPCC Systems com os produtos de seguros ChoicePoint criou-se negócios poderosos e vantajosos - uma solução de processamento de Big Data extremamente eficiente, capaz de gerenciar grandes quantidades de dados, gerando produtos de tratamento de dados muito mais eficientes no já volumoso mercado de seguros.

Em 2011, a LexisNexis decidiu lançar a plataforma HPCC Systems sob uma licença de código aberto. Desde o seu lançamento, a HPCC Systems desenvolveu uma rica comunidade de usuários e uma rede global de clientes. Além disso, a HPCC Systems continua a inovar a plataforma, adicionando suporte para as arquiteturas baseadas em nuvem; em desenvolvimento de novas ferramentas de administração, governança e orquestração de dados; e adicionando novos recursos de dashboard.

Os usuários atuais da plataforma HPCC Systems que podem ser mencionados publicamente incluem a Quod, um Bureau de Crédito no Brasil, o qual a LNRS ajudou a criar, e muitas universidades proeminentes, como: Clemson University, Florida Atlantic University, Kennesaw State University, RV College of Engineering, Universidade de São

Paulo, Universidade Federal de Santa Catarina, Universidade Federal do Pará e várias outras universidades em todo o mundo.

Figura 1.1: A evolução da plataforma HPCC Systems.

1.1.2. O que é a plataforma HPCC Systems?

HPCC Systems é uma plataforma de "Data Lake" de código aberto (open source) projetada para obter dados de diversas fontes de dados em formatos estruturados e não estruturados. Os dados geralmente são armazenados em arquivos simples, como arquivos básicos de disco, ou em armazenamentos de objetos como Amazon S3 e armazenamento BLOB do Azure. Em outras palavras, não há um esquema de formatação pré-definida, na qual os dados precisam ser ajustados antes do armazenamento.

A implementação de uma típica plataforma HPCC Systems começa com apenas algumas fontes de dados, alguns processos de análises iniciais e ferramentas de relatório, mas o tamanho, a complexidade e a capacidade do "Data Lake" pode crescer rapidamente. Depois que os dados são ingeridos no "Data Lake" e, refinados através de limpeza e padronização dos dados, começa o processo de enriquecimento desses dados. Esse enriquecimento de dados é um processo iterativo e evolutivo que extrai tanto conhecimento quanto possível das fontes de dados. Uma vez que esse conhecimento é extraído, ele fica disponível para outros usuários do "Data Lake", que precisam dos mesmos por meio de um processo conhecido como entrega de dados. Durante a entrega de dados, a plataforma HPCC Systems garante que os dados sejam transferidos aos usuários do "Data Lake" de maneira responsiva e segura. Uma analogia pode ajudar a ilustrar o que acontece com os dados à medida que eles entram em um sistema de "Data Lake" do HPCC Systems. Nesta analogia, a água (dados brutos) é coletada em um reservatório (Data Lake) onde em seguida, é processado para torná-lo adequado ao consumo do público.

Para continuar entregando água à população, a usina de beneficiamento não pode parar a coleta ou processamento de água; o processo deve fornecer água de forma confiável 24 horas por dia, sete dias por semana para acompanhar a demanda do consumidor. Um "Data Lake" deve oferecer o mesmo nível de disponibilidade. Não importa quantos dados sejam adicionados ao sistema, o processo de catalogação e análise desses dados deve operar continuamente em níveis de serviço de "cinco noves" (o "Data Lake" deve estar ativo e operacional pelo menos 99,999 por cento do tempo).

A figura abaixo registra o ciclo de vida dos dados em um sistema real de “Data Lake” do HPCC Systems atualmente em uso por um cliente da plataforma. Movendo da esquerda para a direita, as fontes de dados entregam os dados ao “Data Lake” do HPCC Systems para ingestão, refinamento, enriquecimento, indexação e análise. O HPCC Systems pode gerar relatórios ou dashboards sobre os dados em qualquer etapa do processo, dependendo de qual informação o consumidor necessita. Todos esses processos ocorrem dentro do ambiente de “Data Lake” para produzir resultados consistentes.

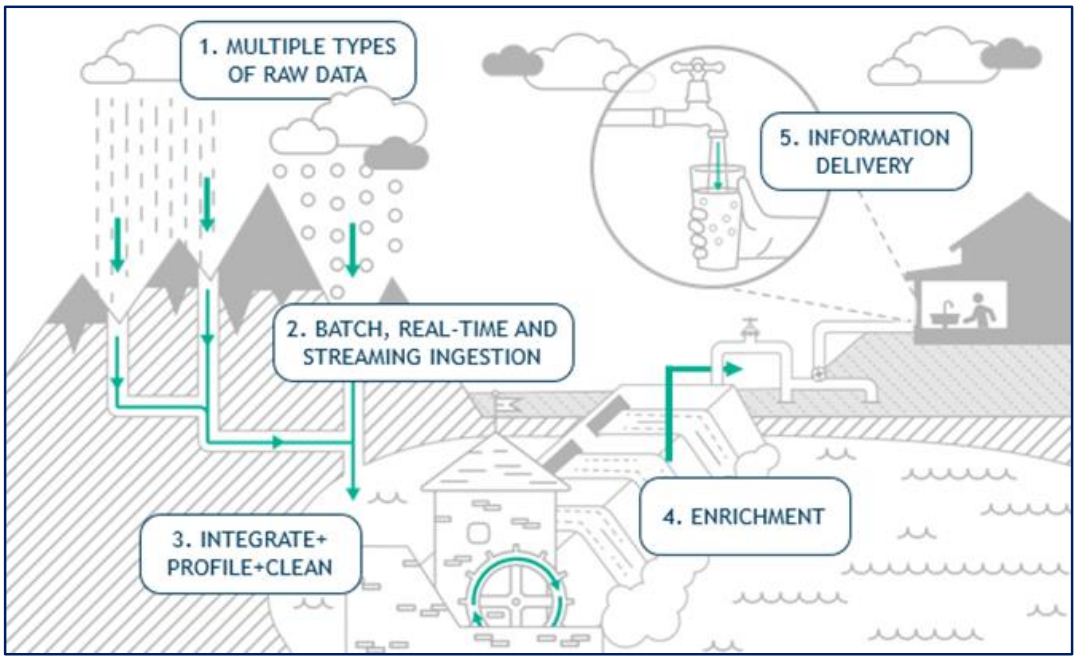


Figura 1.2: Um “Data Lake” deve ser capaz de ingerir, formatar e enriquecer dados com disponibilidade 24 horas por dia, 7 dias por semana.

1.1.3. O Pipeline de enriquecimento de dados no HPCC Systems

O pipeline de dados no HPCC Systems segue os dados desde a origem até sua ingestão no cluster do HPCC Systems, onde é formatado, enriquecido e depois disponibilizado para aplicações hospedadas no Cluster.

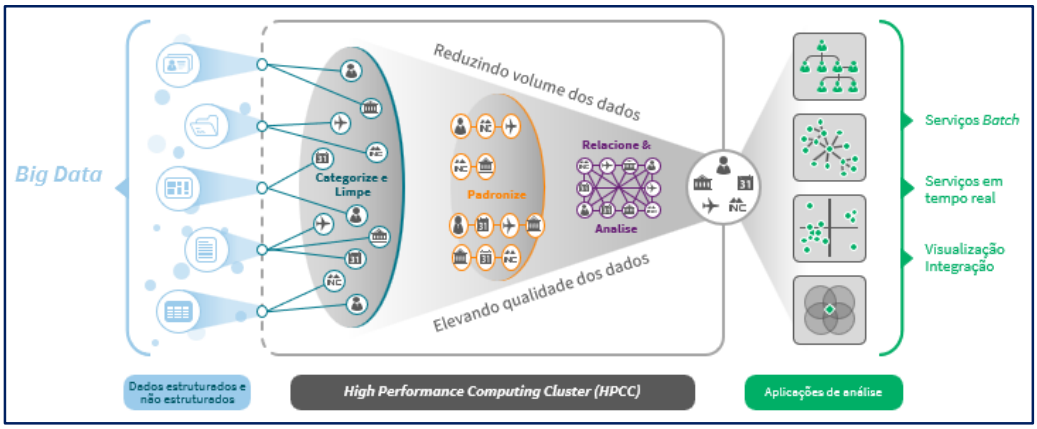


Figura 1.3: Fluxo de dados no HPCC Systems – “Funil” de dados.

1.1.4. Os componentes da plataforma HPCC Systems

O HPCC Systems é composto pelos seguintes componentes:

- A linguagem de programação ECL (Enterprise Control Language) é uma linguagem de programação declarativa e orientada a dados desenvolvida para uso em "Data Lakes" na plataforma HPCC Systems;
- Thor é um cluster de processamento de dados em massa, o qual limpa, padroniza e indexa dados de entrada para uso pelo "Data Lake". Depois que os dados forem refinados pelo Thor, podem, então, serem usados pelo cluster Roxie;
- Roxie é um cluster de API/consulta (query) em tempo real para consultar dados após refinamento por Thor. As consultas Roxie são executadas em menos de um segundo e fornecem resultados de forma concorrente.

Os clusters Thor e Roxie apresentam objetivos específicos e, assim, fazendo uma analogia simplista com o mundo oceânico seria algo como mostrado na figura abaixo:

Figura 1.4: Objetivos dos clusters Thor e Roxie.

1.1.5. A plataforma HPCC Systems

Um diagrama da plataforma HPCC Systems apresentando um cluster Thor (para processamento de dados em massa) e um cluster Roxie (para lidar com consultas de dados) e, ainda, o chamado Power Trio:

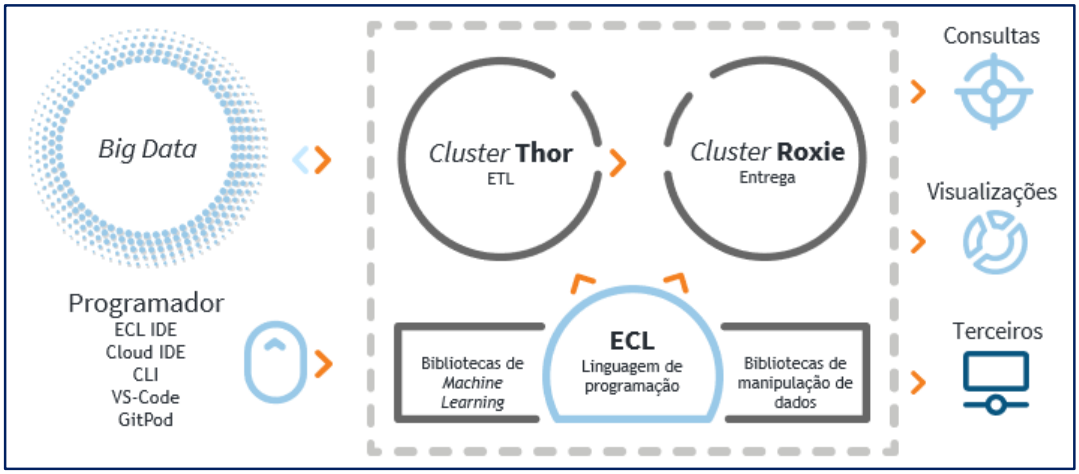


Figura 1.5: Clusters Thor e Roxie.

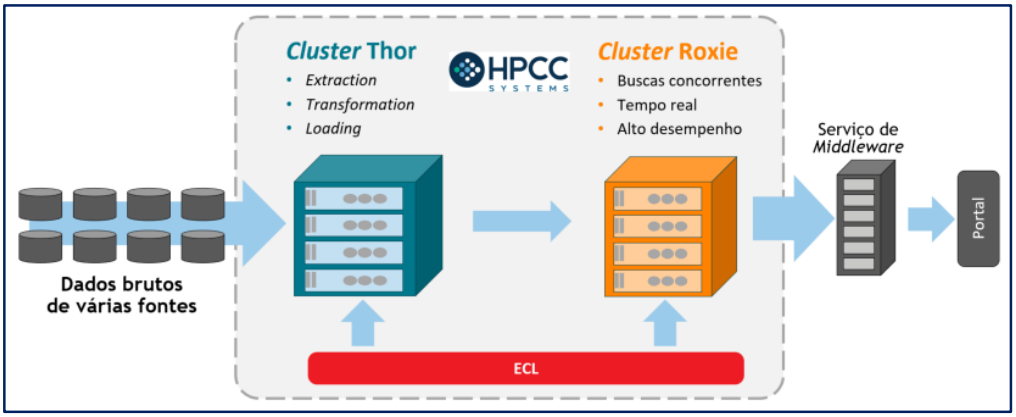


Figura 1.6: Power Trio: A) Thor, B) Roxie e C) ECL.

A) Thor: Arquitetura

Thor é um cluster projetado especificamente para executar processos de manipulação massiva de dados (ETL). Thor é um cluster de preparação de dados de back-office e não se destina a consultas de nível de produção do usuário final.

Os clusters Thor são usados para fazer todo o trabalho "pesado" de preparação de dados para processar dados brutos em formatos padrão. Uma vez concluído o processo, os usuários finais podem consultar esses dados padronizados para coletar informações reais.

No entanto, os usuários finais geralmente desejam ver os seus resultados “imediatamente” e, ainda, geralmente mais de um usuário final deseja obter seus resultados ao mesmo tempo. O cluster Thor só funciona em uma consulta por vez, o que o torna inviável para o usuário final, por isso foi criado o cluster Roxie.

B) Roxie: Arquitetura

Roxie é um cluster projetado especificamente para atender as consultas padrão, fornecendo uma taxa de transferência de mais de mil respostas por segundo (a taxa de resposta real para qualquer consulta, logicamente, depende de sua complexidade). Roxie é um cluster de nível de produção projetado para aplicações de missão crítica.

Os clusters Roxie podem lidar com milhares de usuários finais simultâneos e fornecer a todos eles a percepção dos resultados "imediatamente". Ele faz isso permitindo apenas que os usuários finais executem consultas padrão pré-compiladas que foram desenvolvidas especificamente para uso do usuário final no cluster Roxie.

Normalmente, essas consultas usam índices e, portanto, fornecem desempenho extremamente rápido. No entanto, o cluster Roxie é inviável para uso como ferramenta de desenvolvimento, pois todas as suas consultas devem ser pré-compiladas e os dados utilizados devem ter sido implantados anteriormente.

C) A linguagem ECL

ECL é uma linguagem de programação declarativa, que apresenta inúmeras vantagens sobre o modelo de programação mais convencional. A programação declarativa permite ao programador expressar a lógica de uma computação sem descrever seu controle de

fluxo. Em termos leigos, a linguagem ECL permite que os desenvolvedores digam ao sistema o que eles precisam, mas deixa para o sistema determinar a melhor maneira para fazer isso.

Além de simplificar o design e a implementação de algoritmos complexos, também melhora a qualidade do código, minimizando ou eliminando efeitos colaterais no código do "Data Lake", o que facilita o teste de código e simplifica a manutenção do código. O código ECL é mais fácil de entender e verificar, mesmo por pessoas que não estão familiarizadas com o design original, o que ajuda encurtar a curva de aprendizado para novos programadores.

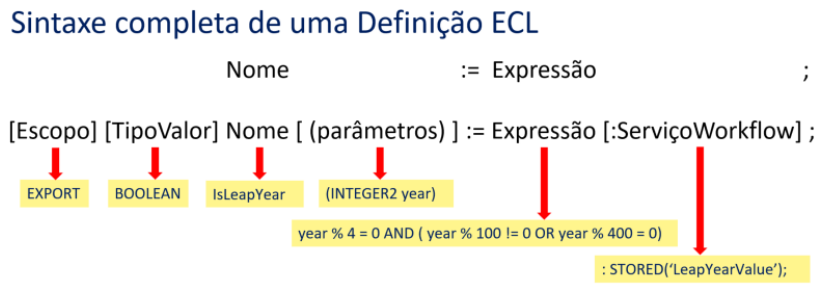
ECL é a única linguagem necessária para expressar algoritmos de dados em toda a plataforma HPCC Systems. No Thor, a linguagem ECL expressa "workflows" de dados que consistem em carregamento dados, transformação, vinculação, indexação etc. No Roxie, a linguagem ECL define consultas de dados. Isso significa que os analistas de dados e programadores da plataforma HPCC Systems só precisam aprender uma linguagem para definir o ciclo de vida completo dos dados.

A linguagem ECL é implicitamente paralela, portanto, o mesmo código ECL desenvolvido para ser executado em um cluster de um nó pode ser executado com a mesma facilidade em um cluster com milhares de nós. O programador não precisa se preocupar em implementar a paralelização, e a linguagem ECL possui uma função otimizadora que garante o melhor desempenho para uma arquitetura específica.

A linguagem ECL foi projetada desde o início para ser uma linguagem de programação orientada a dados. Ao contrário de outras linguagens, funções primitivas de alto nível para dados, como JOIN, TRANSFORM, PROJECT, SORT, DISTRIBUTE, MAP, NORMALIZE etc.; são funções de primeira classe, então operações básicas de dados podem ser implementadas em uma única linha de código. Isto torna a linguagem ECL uma linguagem de programação ideal para análise de dados, pois pode ser usada para expressar algoritmos de dados diretamente, eliminando a necessidade de escrever as especificações do software. Em essência, com o uso da linguagem ECL, os "Data Lakes" no HPCC Systems precisam de menos programadores para entregar mais projetos em menor quantidade de tempo.

- Conceitos básicos de ECL
  - Paradigma declarativo (não-procedural);
  - ECL "não" é sensível a caixa alta/baixa (uppercase/lowercase)
  - Espaço em branco é ignorado para uma melhor leitura;
  - Comentários em linha (//) e em bloco (/* e */);
  - ECL utiliza sintaxe Objeto.Propriedade:
    - Dataset.Campo // um campo em um dataset
    - NomedoDiretorio.Definicao // uma definição em outro módulo
- Definições vs. Ações
  O código ECL é constituído de "Definições" e "Ações".
  - Definições estabelecem o que as coisas são (arquivos de definição ECL):
    - MyDef := 'Hello World'; // não inicia uma WorkUnit

- Ações em ECL resultam em compilação e execução (arquivos BWR):
  - OUTPUT(MyDef); // inicia uma WorkUnit
  - OUTPUT('Hello World, again...'); // inicia uma WorkUnit



1.1.6. Outras ferramentas da plataforma HPCC Systems

A plataforma HPCC Systems fornece para os desenvolvedores um ambiente de desenvolvimento integrado (IDE) denominado como "ECL IDE", a fim de facilitar o desenvolvimento de código ECL. O "ECL IDE" é uma aplicação para o sistema operacional Windows. Há também uma extensão de linguagem ECL disponível para VS Code que alguns desenvolvedores preferem usar.

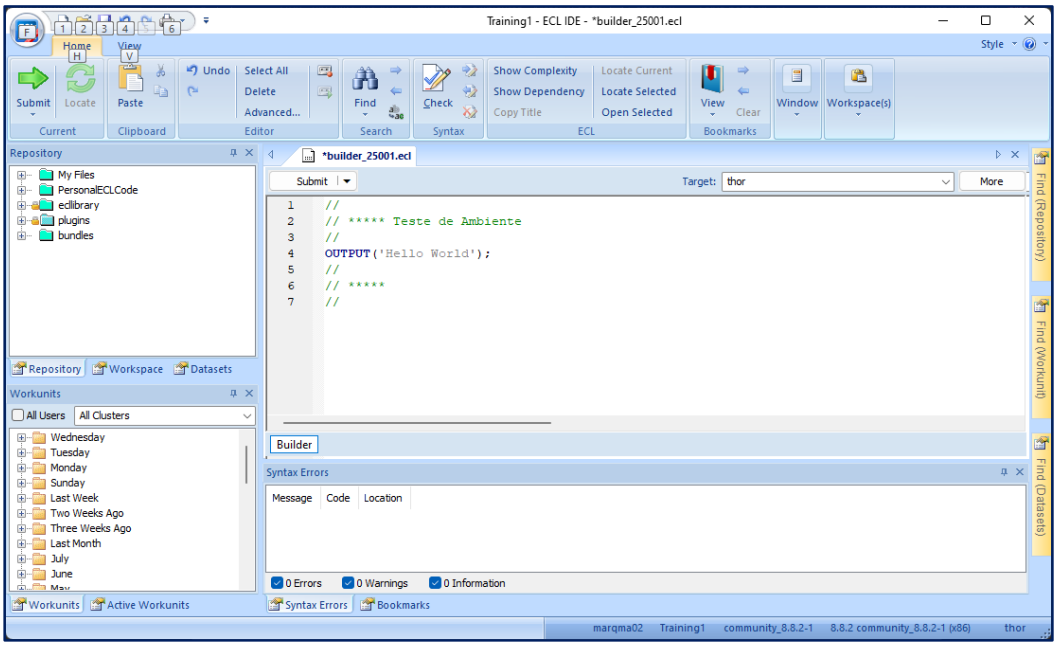


Figura 1.7: ECL Integrated Development Environment (IDE).

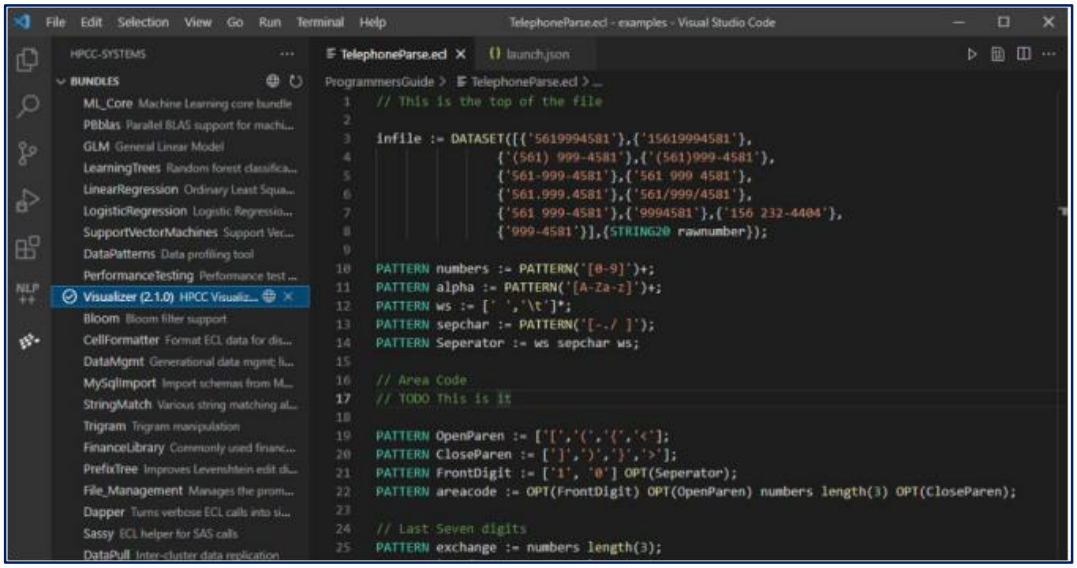


Figura 1.8: VS Code IDE.

1.2. Algoritmos de Aprendizado de Máquina para a plataforma HPCC Systems

A característica principal do Aprendizado de Máquina (Machine Learning) é a capacidade de inferir sobre relacionamentos e, visa prever uma resposta razoável quando apresentado com dados nunca vistos.

O "aprendizado" do Machine Learning (ML) tem várias categorias:

- Supervisionado - O tipo mais comum de ML. Esse método envolve o treinamento do sistema em que os recordsets, juntamente com o padrão de saída de destino, são fornecidos ao sistema para executar uma tarefa;
- Não Supervisionado - Este método não envolve a saída de destino, o que significa que nenhum treinamento é fornecido ao sistema. O sistema precisa aprender por meio da determinação e adaptação de acordo com as características estruturais nos padrões de entrada.
- Deep Learning - Move-se para a área dos métodos ML de Redes Neurais (Neural Networks - NNs). O Deep Learning implica várias camadas maiores que ‘2’ e, também, implica em técnicas utilizadas com dados complexos, como análise de vídeo ou áudio.

1.2.1. Aprendizado Supervisionado

Essa será a categoria abordada nesse estudo com foco no método de Árvores de Decisão (Learning Trees - Random Forests).

A premissa básica do ML Supervisionado é que, dado um conjunto de “amostras de dados” (registros) e um conjunto de “valores-alvo” em campo e formato de registro, que ele aprenda como prever “valores-alvo para novas amostras”.

- Amostras de dados: conhecidas como variáveis “Independentes”, porque são as informações fornecidas e não dependem de nenhum outro dado. Variáveis Independentes também são conhecidas como ‘Características’ (features) dos dados.

- Valores-alvo: conhecidos como variáveis “Dependentes”, porque eles são de alguma forma dependentes das amostras de dados.

As variáveis ‘Independentes’ e ‘Dependentes’ juntas são conhecidas como “conjunto de treinamento”.

Dois tipos básicos de Modelos de Análise em Aprendizado Supervisionado:

- Quantitativo - conhecido como “Regressão” - implica um valor numérico;
- Qualitativo - conhecido como “Classificação” - implica uma categoria ou, às vezes, um resultado binário.

1.3. Bundles de Machine Learning da plataforma HPCC Systems

Os bundles de produção (excluindo os bundles de suporte ML_Core e PBblas) fornecem uma interface principal muito semelhante para o Machine Learning. No entanto, cada algoritmo tem suas próprias peculiaridades (suposições e restrições) que devem ser levadas em consideração. Portanto, é importante ler a documentação que acompanha cada bundle para usá-lo efetivamente.

➢ Link definitivo para todos os bundles de produção, sua documentação e tutoriais, quando aplicável:
https://hpccsystems.com/download/free-modules/machine-learning-library

a) Bundles Principais:

- ML_Core - Machine Learning Core
  Fornece as principais definições de dados para ML. É um pré-requisito para todos os outros pacotes configuráveis de produção.
  Mais informações: https://github.com/hpcc-systems/ML_Core

- PBblas - Parallel Block Basic Linear Algebra Subsystem
  Fornece operações de matriz escalonáveis e distribuídas usadas por vários dos outros pacotes configuráveis. Também pode ser usado diretamente sempre que as operações da matriz estiverem em ordem. Essa é uma dependência para vários dos outros pacotes configuráveis.
  Mais informações: https://github.com/hpcc-systems/PBblas

b) Bundles de Aprendizado Supervisionado:

- LearningTrees (baseado no algoritmo clássico “ML RandomForest”)
  Classificação e Regressão baseadas em Árvores de Decisão. Um dos melhores métodos de ML "prontos para uso", pois faz poucas suposições sobre a natureza dos dados e é resistente ao “overfitting” (*). Capaz de lidar com muitas variáveis independentes. Cria uma “floresta” de diversas Árvores de Decisão e calcula a média das respostas das diferentes Árvores.
  Mais informações: https://github.com/hpcc-systems/LearningTrees

(*) Superajuste (overfitting): Muitos algoritmos tendem a superestimar o conjunto de treinamento. Isso significa que ele está reduzindo o erro de previsão ajustando-se ao ruído que ocorre nesse conjunto de dados. Um modelo de excesso de ajuste tratará esse ruído como sinal e usará o que aprendeu para prever os próximos dados apresentados. Infelizmente, esse próximo conjunto de dados estará sujeito a um conjunto de ruído completamente diferente, o que não fornecerá bons resultados.

1.4. Interface Myriad

Todos os bundles de ML de produção suportam a interface Myriad, que é uma maneira de executar muitas ações semelhantes em diferentes conjuntos de dados, com uma única invocação da interface. Por exemplo, pode-se querer criar um modelo separado para cada grande área metropolitana (cidade/estado) e, então, usar esse conjunto de modelos para prever dados para cada área usando o seu próprio modelo exclusivo. A interface Myriad permite o processamento dessas atividades em paralelo.

Este material pressupõe que se tenha um conhecimento básico dos conceitos e terminologia de Machine Learning, que se esteja familiarizado com os conceitos básicos de uso dos bundles de ML nativos na plataforma HPCC Systems e que se tenha alguma experiência com programação em ECL.

1.4.1. Conceito

Conceitualmente, a interface Myriad...

- É um conjunto de padrões incorporados aos bundles de ML de produção nativos na plataforma HPCC Systems;
- É desenhada para executar várias atividades de Aprendizado de Máquina em uma única invocação da interface: Isso significa que se pode invocar um único processo de treinamento, o qual poderá retornar vários modelos, possibilitando fazer uma única previsão em dados desses vários modelos. Logo, pode-se avaliar a precisão de todos os modelos com uma única chamada para a função "Accuracy(...)";
- É uma forma de maximizar a utilização dos recursos do cluster da plataforma HPCC Systems: Ao executar as atividades ao mesmo tempo, permite aproveitar totalmente o paralelismo do cluster, realizando todas as atividades em uma única invocação, contrastando com a execução serial, onde muitos nós podem ficar ociosos para qualquer atividade;
- Permite que todas as atividades sejam balanceadas em todos os nós do cluster;
- Fornece sincronização mais eficiente, necessária para muitos algoritmos de Machine Learning.

1.4.2. Usando a Interface Myriad

Como um exemplo de uso da interface Myriad, conforme já mencionado, será considerado que se deseje criar um modelo separado para cada grande área metropolitana (cidade/estado), a fim de que se possa fazer previsões mais granulares baseadas em dados de cada área.

Com a interface Myriad pode-se invocar um único treinamento, aqui processado através da função "GetModel(…)", de maneira que retorne "n" modelos diferentes compactados em um só. Então, com uma única invocação de "Predict(…)", pode-se fazer previsões sobre dados de todas as "n" áreas metropolitanas, cada uma em relação ao modelo aprendido apropriado para essa área. Pode-se, então, avaliar a precisão dos "n" modelos com uma única chamada para "Accuracy(…)". Isso é conveniente do ponto de vista da programação, mas, mais importante ainda é poder fazer o melhor uso dos recursos do cluster da plataforma HPCC Systems.

Executar as atividades ao mesmo tempo permite aproveitar mais completamente o paralelismo do cluster da plataforma HPCC Systems realizando todas as atividades em uma invocação. Compare isso com a invocação serial de cada atividade. Se cada atividade não puder usar o conjunto completo de nós da plataforma, porque é de tamanho moderado, ou o algoritmo específico não pode utilizar todos os nós de uma vez, muitos dos nós ficarão ociosos para cada atividade. Executar as atividades uma de cada vez, portanto, permite que muitos nós sejam subutilizados durante toda a operação. Executar as atividades em paralelo com a interface Myriad, possibilita que as atividades sejam balanceadas em todos os nós do cluster, maximizando assim o desempenho (ou seja, minimizando o tempo de execução).

Além disso, muitos dos algoritmos de ML exigem várias etapas sequenciais para concluir uma atividade. Cada etapa requer sincronização entre os nós no cluster. Quanto mais sincronizações forem necessárias, menos otimizado será o processamento paralelo dentro do cluster. Executar várias atividades em paralelo não aumenta o número de sincronizações em relação a uma única atividade, embora cada sincronização se torne maior (ou seja, mais dados). Embora as sincronizações maiores afetem o desempenho, o impacto é pequeno em comparação ao aumento do número de etapas de sincronização. Isso ocorre devido à latência (atrasos) na comunicação de rede. Ao esperar por respostas de um nó para outro, nenhum trabalho pode ser feito. Requisições maiores não criam atrasos tão improdutivos porque não há espera, exceto pelo tempo real da transferência de dados na rede.

A estratégia da interface Myriad está incorporada nas principais estruturas de registro usadas pelo ML. Lembrando que, o ML utiliza vários tipos principais de registro em suas interfaces:

- NumericField é usado para fornecer dados de valor real em forma de matriz;
- DiscreteField é usado para transmitir dados discretos (de valor inteiro); e
- Layout_Model é usado para codificar o Modelo que armazena tudo o que é aprendido sobre os dados.

Cada um desses layouts tem um campo conhecido como “work-item-id” (‘wi’). É esse parâmetro “id” que fornece isolamento das várias atividades independentes na interface Myriad.

Quando se fornece os dados independentes, pode-se usar um ‘wi’ diferente para os dados de cada área metropolitana. O mesmo pode ser feito para os dados de treinamento dependentes. Quando for chamado o “GetModel(…)”, o conjunto de dados retornado conterá todos os modelos separados, cada um identificado pelo seu respectivo ‘wi’. Por exemplo, o modelo que foi treinado por dados independentes e dependentes com ‘wi=1’ será rotulado com ‘wi=1’. O mesmo ocorre para cada ‘wi’. Agora, usando o(s) modelo(s) para prever um novo valor dependente, por meio de “Predict(…)” (para Regressão) ou “Classify(…)” (para Classificação), as previsões serão baseadas no modelo com o mesmo ‘wi’ que os itens de dados independentes. O mesmo é verdadeiro quando se avalia os modelos usando “Accuracy(…)”.

Na verdade, cada interface dentro dos bundles de ML de produção pode lidar com várias atividades independentes por meio do uso de “work-item-ids”.

A figura mostrada abaixo, ilustra esse fluxo de dados.

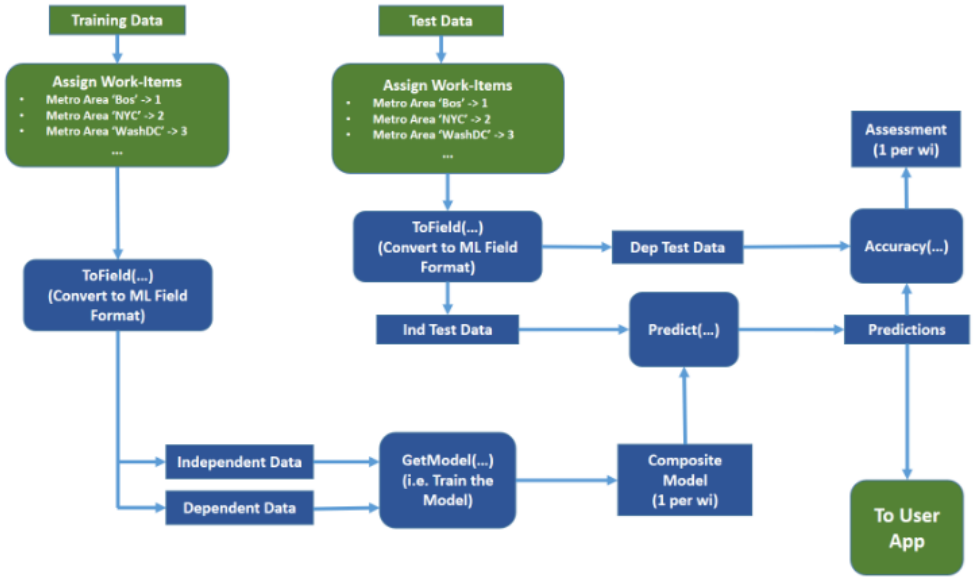


Figura 1.9: Como a interface Myriad funciona.

Mais adiante será apresentado um estudo de caso como exemplo da aplicação de algoritmos de Machine Learning fazendo uso da interface Myriad.

1.5. ML Supervisionado: Árvores de Decisão (Learning Trees - Random Forests)

As Árvores de Decisão têm sido utilizadas, pelo menos, desde a década de 1930 como uma forma de estruturar o conhecimento usando um conjunto de regras em cascata. Elas são conceitualmente simples e razoavelmente fáceis de entender e interpretar.

O LearningTrees bundle fornece uma implementação eficiente e escalável dos métodos Learning Trees. Atualmente, fornece algoritmos de "Decision Trees", "Random Forest", "Gradient Boosted Trees" e "Boosted Forest“.

Diante dos diversos algoritmos disponíveis, qual algoritmo se deve escolher?

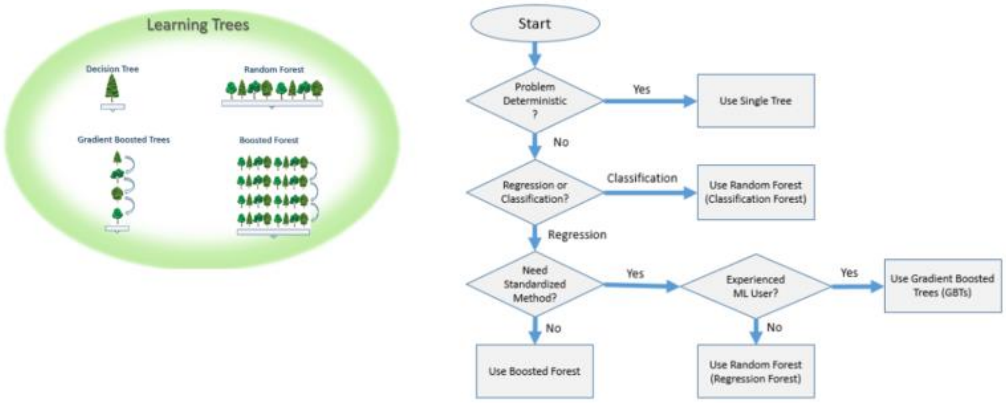


Figura 1.10: Fluxograma a ser usado para auxiliar nessa escolha.

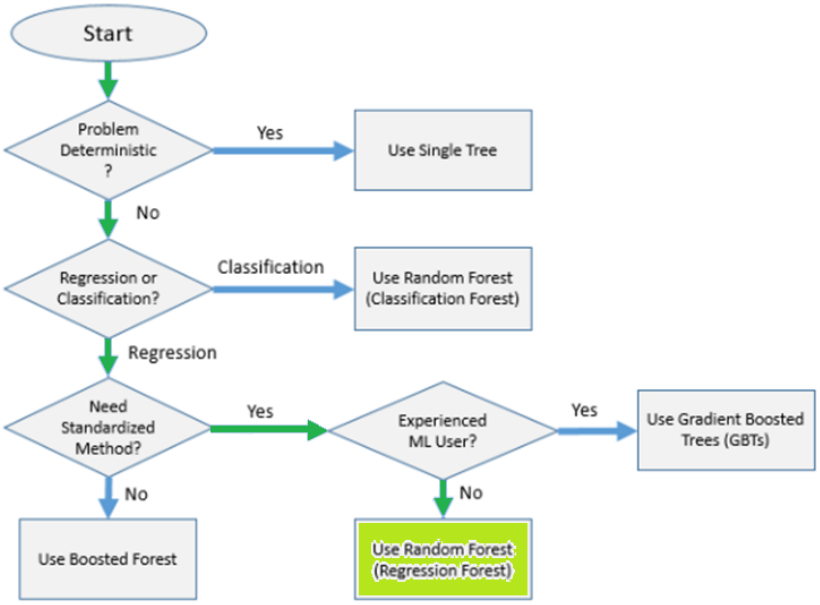


Figura 1.11: No caso em estudo "preço de imóveis" – Escolha: Random Forest - Regression Forest.

Isso funciona muito bem desde que o problema seja 'determinístico' - ou seja, as mesmas variáveis (features) sempre produzem os mesmos resultados e, há dados de treinamento suficientes, desencadeando no uso de uma Árvore Simples. Se o problema for 'estocástico' - incorpora aleatoriedade nos dados ou resultados - uma Árvore de Decisão provavelmente não "generalizará" bem. Quanto ao conceito de "generalização" é importante entender a natureza da 'População' versus 'Amostra' (sample). Os dados de treinamento para um modelo de ML são quase sempre uma pequena amostra da população-alvo:

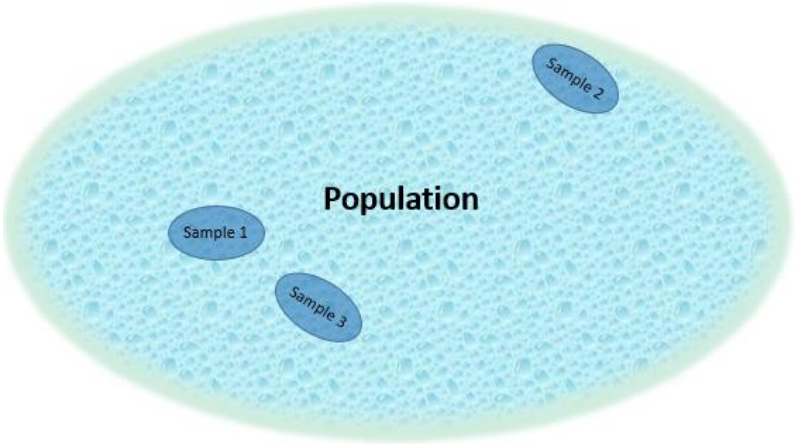


Figura 1.12: Generalização.

Em uma Random Forest, uma série de Árvores de Decisão são geradas, com alguma aleatoriedade adicionada ao processo de geração para garantir que cada árvore use um processo de decisão diferente na separação dos pontos. Então, para cada novo ponto de dados, todas as árvores são consultadas para sua previsão. Essas previsões individuais são agregadas para formar a previsão final.

Se o objetivo da Random Forest é escolher uma das várias classes (ou seja, uma ClassificationForest), então a agregação é feita por votação: a “classe prevista pelo maior número de árvores” torna-se a previsão final. Se o objetivo é prever um valor numérico (como o preço de imóveis), então temos uma RegressionForest, que forma uma previsão final pela “média das previsões das várias árvores”.

Um dos aspectos interessantes do Random Forest é que há muito poucos parâmetros para ajustar, e a escolha desses parâmetros geralmente tem muito pouca influência na precisão dos resultados. Bons resultados geralmente podem ser alcançados usando os parâmetros "default", tornando-o um dos algoritmos de ML mais fáceis de usar.

Outra grande característica é que o Random Forest pode lidar com muitas variáveis. Não é incomum usá-lo com milhares de variáveis.

Por fim, Random Forest são facilmente “paralelizáveis”, sendo, portanto, um algoritmo ideal para uso em clusters do HPCC Systems.

1.5.1. O Fluxo de Aprendizado de Máquina Supervisionado

No estudo de caso foi selecionado o bundle “LearningTrees ML” pelos seguintes motivos:

1. Fácil de usar:
   - Faz muito pouca suposição sobre a distribuição dos dados ou seus relacionamentos;
   - Ele pode lidar com muitos registros e muitos campos;
   - Ele pode lidar com relações não lineares e descontínuas;
   - Quase sempre funciona bem usando os parâmetros padrão e sem nenhum ajuste.
2. Escalabilidade
   - Escala bem em clusters HPCC Systems de quase qualquer tamanho.
3. Sua precisão de previsão é competitiva com os melhores algoritmos de última geração.

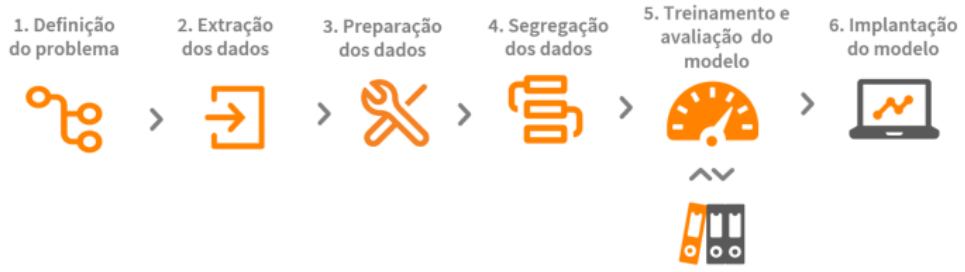
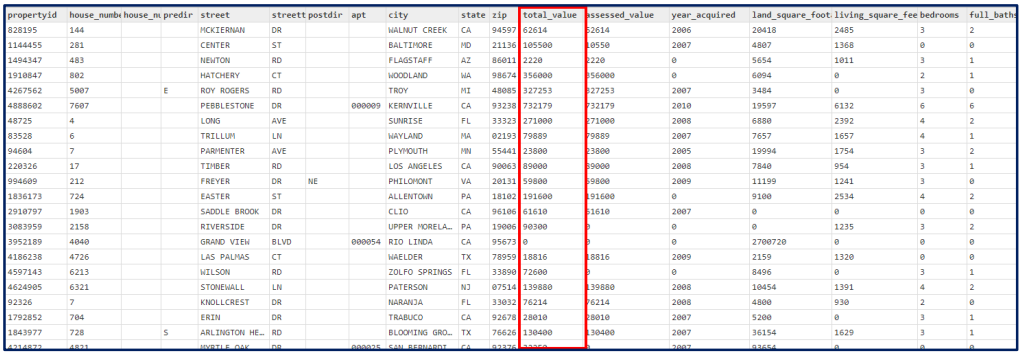


Figura 1.13: Fluxo de ML Supervisionado.

1.5.1.1. Definição do problema

“Dado um conjunto de atributos de uma propriedade (localização, metragem quadrada, ano de construção/aquisição, número de cômodos, etc.), como predizer o seu valor real de venda?"



propertyid	house_numb	house_nu	predir	street	streett	postdir	apt	city	state	zip	total_value	assessed_value	year_acquired	land_square_foot	living_square_fee	bedrooms	full_baths
828195	144			MCKIERNAN	DR			WALNUT CREEK	CA	94597	62614	62614	2006	20418	2485	3	2
1144455	281			CENTER	ST			BALTIMORE	MD	21136	105500	10550	2007	4807	1368	0	0
1494347	483			NEWTON	RD			FLAGSTAFF	AZ	86011	2220	2220	0	5654	1011	3	1
1910847	802			HATCHERY	CT			WOODLAND	WA	98674	356000	356000	0	6094	0	2	1
4267562	5007		E	ROY ROGERS	RD			TROY	MI	48085	327253	327253	2007	3484	0	3	0
4888602	7607			PEBBLESTONE	DR		000009	KERNVILLE	CA	93238	732179	732179	2010	19597	6132	6	6
48725	4			LONG	AVE			SUNRISE	FL	33323	271000	271000	2008	6880	2392	4	2
83528	6			TRILLUM	LN			WAYLAND	MA	02193	79889	79889	2007	7657	1657	4	1
94604	7			PARMENTER	AVE			PLYMOUTH	MN	55441	23800	23800	2005	19994	1754	3	2
220326	17			TIMBER	RD			LOS ANGELES	CA	90063	89000	89000	2008	7840	954	3	1
994609	212			FREYER	DR	NE		PHILOMONT	VA	20131	59800	59800	2009	11199	1241	3	0
1836173	724			EASTER	ST			ALLENTOWN	PA	18102	191600	191600	0	9100	2534	4	2
2910797	1903			SADDLE BROOK	DR			CLIO	CA	96106	61610	51610	2007	0	0	0	0
3083959	2158			RIVERSIDE	DR			UPPER MORELA…	PA	19006	90300	0	0	0	1235	3	2
3952189	4040			GRAND VIEW	BLVD		000054	RIO LINDA	CA	95673	0	0	0	2700720	0	0	0
4186238	4726			LAS PALMAS	CT			WAELDER	TX	78959	18816	18816	2009	2159	1320	0	0
4597143	6213			WILSON	RD			ZOLFO SPRINGS	FL	33890	72600	0	0	8496	0	3	1
4624905	6321			STONEWALL	LN			PATERSON	NJ	07514	139880	139880	2008	10454	1391	4	2
92326	7			KNOLLCREST	DR			NARANJA	FL	33032	76214	76214	2008	4800	930	2	0
1792852	704			ERIN	DR			TRABUCO	CA	92678	28010	28010	2007	5200	0	3	1
1843977	728		S	ARLINGTON HE…	RD			BLOOMING GRO…	TX	76626	130400	130400	2007	36154	1629	3	1
4214872	4821			HYDTLE OAK	DR		000025	SAN BERNARDI	CA	92376	22850	0	2007	93654	0	0	0

Figura 1.14: Preço do imóvel – Valor real de venda (total_value).

1.5.1.2. Extração dos dados

Será utilizado o dataset "Property.csv" no formato "CSV" contendo 1.662.959 registros. Usando os campos selecionados, tentaremos prever o preço da propriedade com base em outros dados (por exemplo: localização, metragem quadrada, ano de aquisição/construção, número de cômodos, código postal, etc.).

Arquivo lógico: "~CLASS::XYZ::ML::Property".

- Códigos ECL utilizados:
  modProperty.ecl
  BWR_BrowseData.ecl

1.5.1.3. Preparação dos dados

A preparação dos dados é a etapa mais importante ao usar qualquer algoritmo ML. Como em qualquer processo de programação, a qualidade dos dados recebidos terá um grande efeito na qualidade dos resultados.

Aqui estão algumas regras importantes a serem consideradas:

- Os dados devem conter todos os valores numéricos;
- O primeiro campo no registro deve ser um identificador exclusivo (geralmente um ID de registro sem sinal - UNSIGNED);
- Para facilitar a implementação, mova seu campo "dependente" para o final da estrutura RECORD;
- Randomize seus dados para criar um recordset de treinamento e um teste mais preciso. Isso pode ser feito adicionando um campo com um número aleatório (RANDOM);
- Faça previamente a limpeza dos dados (cleasing);
- Durante a preparação dos dados, atribua "work-item-ids" com base no campo 'state' através da função MAP, a fim de criar um modelo de ML separado para cada estado norte-americano existente no dataset "Property";
- Usando o campo aleatório gerado, classifique e segregue os dados nos dados iniciais de Treinamento e de Teste. Os dados de Treinamento serão usados para treinar seu modelo de ML e os dados de Teste serão usados para avaliar (ou analisar) a eficácia do modelo. É fundamental que se reserve alguns dos dados para teste, pois é uma péssima ideia testar o modelo com os mesmos dados nos quais se treinou.

- Códigos ECL utilizados:
  isCleanFilter.ecl
  CleanProperty.ecl
  modPrepData.ecl
  BWR_ViewData1.ecl

##	propertyid	zip	assessed_value	year_acquired	land_square_footage	living_square_feet	bedrooms	full_baths	half_baths	year_built	total_value
1	79784	33424	76440	2015	4299	1255	3	2	0	2010	76440
2	3924129	20601	95900	2013	11224	1468	3	2	1	2007	95900
3	413843	8803	76000	2015	57000	1858	3	2	0	1970	76000
4	608224	98370	39340	2012	7405	1066	3	1	1	1967	39340
5	942963	72032	278400	2008	9600	2459	3	2	0	1963	278400
6	2237271	79935	143600	2011	8430	1008	2	1	1	1961	143600
7	4443742	84065	166934	2013	9317	1700	4	2	0	1991	166934
8	3834707	66227	348350	2012	15300	2663	4	2	1	2002	348350
9	3592739	19606	54000	2015	15060	2292	4	2	1	1980	90000
10	2916349	34639	119050	2015	6947	1709	3	2	0	2009	140950

Figura 1.15: Limpeza, padronização e consolidação de registros.

1.5.1.4. Segregação dos dados

Ao segregar os dados é importante coletar amostras aleatoriamente do seu dataset, ao invés de usar os primeiros 'N' registros no conjunto, porque pode existir alguma ordem oculta no processo que originalmente gerou os dados. A segregação dos dados de Treinamento e de Teste é feita facilmente usando ECL.

Como regra geral, cerca de 20 a 30% dos seus dados devem ser reservados para Teste. Entretanto, para fins didáticos e, principalmente, objetivando uma melhor performance durante a realização desse estudo, esse percentual será reduzido, considerando uma proporção de 5.000 registros para a amostra de Treinamento e 2.000 registros para a amostra de Teste:

```
// Considerando os primeiros 5.000 registros como amostra de Treinamento
MyriadTrainData := PROJECT(MyriadPrepDataSort[1..5000],
        $.modPrepData.ML_Prop)  :PERSIST('~CLASS::XYZ::ML::MyriadTrain');
// Considerando os 2.000 registros seguintes como amostra de Teste
MyriadTestData  := PROJECT(MyriadPrepDataSort[5001..7000],
        $.modPrepData.ML_Prop)  :PERSIST('~CLASS::XYZ::ML::MyriadTest');
```

O próximo passo é converter os dados para o formato usado pelos pacotes configuráveis ML. Para operar genericamente com qualquer dado, o ML requer que os dados estejam em um "layout de matriz orientado a célula" conhecido como "NumericField", considerando o parâmetro "wifield" associado ao novo campo 'wi_id'. O bundle configurável ML_Core facilita essa tarefa:

```
IMPORT ML_Core;
// Conversão Matricial dos campos numéricos
ML_Core.ToField(MyriadTrainData, MyriadTrainDataNF, wifield := wi_id);
        // "idfield" parâmetro não fornecido, assume o 1º campo 'propertyid'
ML_Core.ToField(MyriadTestData, MyriadTestDataNF, wifield := wi_id);
        // "idfield" parâmetro não fornecido, assume o 1º campo 'propertyid'
```

A etapa final antes de aplicar o modelo ML é separar os dados “independentes” dos dados “dependentes”, definindo os dados do ML RECORD para colocar o campo de dados dependente no “final” do layout de registro. Um filtro simples nos dados de treinamento e de teste conclui esta etapa.

Sempre deve ser excluído o ID do registro exclusivo e, conte apenas seus campos independentes e dependentes. No caso, os dados de treinamento do dataset “Property”, tem nove (9) campos independentes e o último campo (10º) é o campo dependente (o valor que será previsto).

O uso do PROJECT definindo o campo numérico = 1 não é estritamente necessário. Isso indica que é o primeiro campo dos dados dependentes. Como existe apenas um campo dependente, ele foi numerado de acordo:

```
EXPORT MyriadIndTrainDataNF := MyriadTrainDataNF(number < 10);
EXPORT MyriadDepTrainDataNF := PROJECT(MyriadTrainDataNF(number =
10),
        TRANSFORM(RECORDOF(LEFT), SELF.number := 1, SELF := LEFT));
EXPORT MyriadIndTestDataNF := MyriadTestDataNF(number < 10);
EXPORT MyriadDepTestDataNF := PROJECT(MyriadTestDataNF(number = 10),
        TRANSFORM(RECORDOF(LEFT), SELF.number := 1, SELF := LEFT));
```

- Códigos ECL utilizados:
  modSegConvData.ecl
  BWR_ViewData2.ecl

##	wi	id	number	value
1	22	2292044	1	6751.0
2	22	2292044	2	75837.0
3	22	2292044	3	2013.0
4	22	2292044	4	3484.0
5	22	2292044	5	763.0
6	22	2292044	6	1.0
7	22	2292044	7	1.0
8	22	2292044	8	1.0
9	22	2292044	9	2002.0
10	16	3675939	1	12546.0

##	wi	id	number	value
1	22	2292044	1	75837.0
2	16	3675939	1	131500.0
3	3	4346206	1	78954.0
4	3	4168683	1	65000.0
5	18	2956615	1	168821.0
6	41	2622288	1	65568.0
7	18	1196844	1	36100.0
8	43	114945	1	19796.0
9	6	961534	1	156893.0
10	41	882691	1	71786.0

Figura 1.16: Nove (9) campos independentes e o último campo (10º) é o campo dependente, com o work-item-id (wi) associado aos estados norte-americanos.

1.5.1.5. Treinamento e avaliação do modelo

O primeiro passo é selecionar o modelo "learner”. Para Regressão (quantitativa), será usado o módulo RegressionForest:

```
IMPORT $;
IMPORT LearningTrees AS LT;
/* Selecionando o algoritmo...
   Sintaxe:
RegressionForest(UNSIGNED numTrees=100, UNSIGNED featuresPerNode=0,
UNSIGNED maxDepth=100, SET OF UNSIGNED nominalFields=[]);
*/
```

```
MyriadLearnerR := LT.RegressionForest();          // parâmetros default
```

Em seguida, o "learner" será usado para treinar e recuperar o modelo:

```
MyriadModelR :=
        MyriadLearnerR.GetModel($.modSegConvData.MyriadIndTrainDataNF,
                $.modSegConvData.MyriadDepTrainDataNF);
```

Observe que o modelo obtido é uma estrutura de dados opaca (interpretável apenas pelo bundle) que encapsula todos os resultados do treinamento. O primeiro parâmetro fornecido para a função "GetModel" é o dataset de treinamento "independente" e o segundo parâmetro é o dataset de treinamento "dependente".

Em seguida, o modelo será usado para fazer previsões, prevendo o campo dependente com base nos campos de teste independentes:

```
MyriadPredictedDeps := MyriadLearnerR.Predict(MyriadModelR,
$.modSegConvData.MyriadIndTestDataNF);
```

Portanto, aplicando o modelo de treinamento e prevendo os resultados dependentes com base nos dados independentes, permitirá realmente produzir e visualizar esses resultados para análise.

Após o treinamento, será testado o modelo de previsão RegressionForest comparando-o com o dataset de teste dependente [MyriadDepTestDataNF], que foi extraído anteriormente dos registros de teste segregados. ML_Core tem uma ótima maneira de fazer isso no módulo "Analysis". Esse módulo fornece uma avaliação genérica do poder preditivo do modelo. Também há uma grande variedade de outras métricas de avaliação fornecidas pelos pacotes individuais, específicas para os algoritmos desse pacote. No Modelo de Regressão do LearningTrees, basta uma linha de ECL para realizar essa análise:

```
MyriadAssessmentR :=
        ML_Core.Analysis.Regression.Accuracy(MyriadPredictedDeps,
                $.modSegConvData.MyriadDepTestDataNF);
```

A função "Accuracy" usa o resultado da função "Predict" como primeiro parâmetro, comparando-o com os dados de teste dependentes que foi criado anteriormente. Aqui estão os resultados dos dados de treinamento para o dataset "Property" (vide Nota):

- Métricas chaves:
  - r2 (R-Quadrado - O "coeficiente de determinação". Aproximadamente, a proporção da variação nos dados originais que foram capturados pela regressão. O R-Quadrado pode variar ligeiramente de negativo a 1. Um R-Quadrado 1 significa que as previsões correspondem perfeitamente às reais. R-Quadrado zero ou abaixo significa que o modelo não tem valor preditivo. R-Quadrado 0,5 significa que metade da variação dos dados foi explicada pelo modelo.
  - MSE (Mean Squared Error - Erro quadrático médio) - O desvio médio quadrático entre os valores previstos e reais.
  - RMSE (Root Mean Squared Error - Raiz do Erro quadrático médio) - A raiz quadrada do MSE. Essa é uma expectativa do erro médio em uma determinada amostra.

##	wi	regressor	r2	mse	rmse
1	48	1	1.0	7899654400.0	88880.0
2	40	1	0.9311376324090863	86100656.52500002	9279.043944555928
3	6	1	0.8407434005563046	1188713430.574167	34477.72368608703
4	53	1	0.8065354715574288	2619422585.198289	51180.29489166987
5	37	1	0.8051321121617373	3681390045.728787	60674.45958332705
6	41	1	0.8033767773856457	2630432563.605217	51287.74282033883
7	32	1	0.8003100933540183	1700812118.14	41240.90345930846
8	22	1	0.798261028979762	953955898.739406	30886.17649919468
9	34	1	0.7728380339814155	522134100.9433332	22850.25384855348
10	17	1	0.7696693188373955	2411024752.625271	49102.18684157836

Figura 1.17: Métricas chaves – Ordenação pelo Coeficiente de Determinação (r2).

Nota: Para o conjunto de treinamento aplicado em "Property" (dataset bruto), a precisão foi um pouco abaixo de 30% na primeira tentativa, onde para melhorar a precisão foi necessário revisar os campos independentes. Após uma análise mais aprofundada, pode-se verificar que o campo 'zip' não é realmente quantitativo, mas qualitativo (variável categórica). Então, a seguinte modificação no modelo de aprendizado foi realizada:

MyriadLearnerR := LT.RegressionForest(,,,[1]);

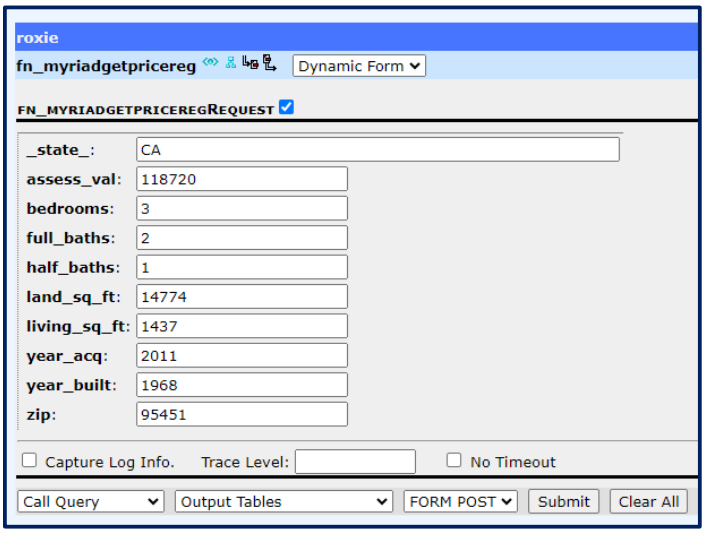
A etapa final foi revisar as amostras e remover os registros com valores de campo "nulos" das amostras de treinamento e de teste, tendo sido usado para tal procedimento o código [CleanProperty].

- Códigos ECL utilizados:

  BWR_TrainReg.ecl

1.5.1.6. Implantação do modelo

Essa etapa corresponde ao desenvolvimento de um serviço de consulta através da codificação de uma "função" (estrutura Function) da linguagem ECL, seguida da compilação do código no cluster Roxie:



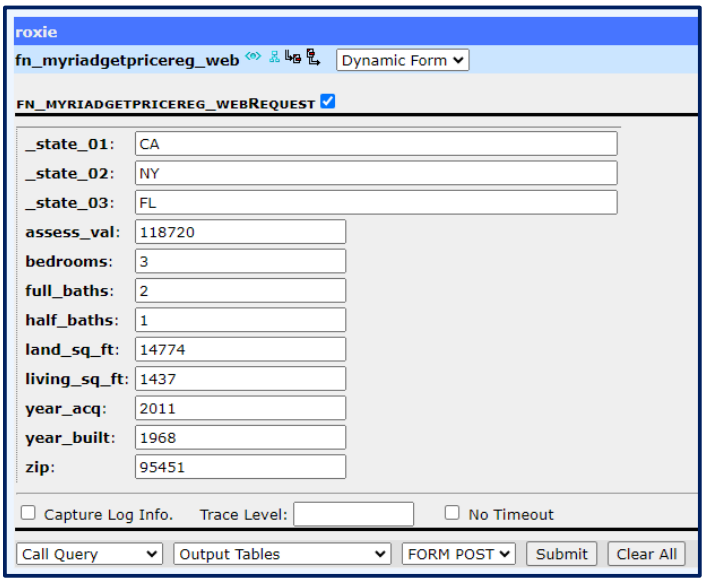


Figura 1.18: Carregamento de dados e disponibilização de serviços de consulta através de ESP (Enterprise Service Platform).

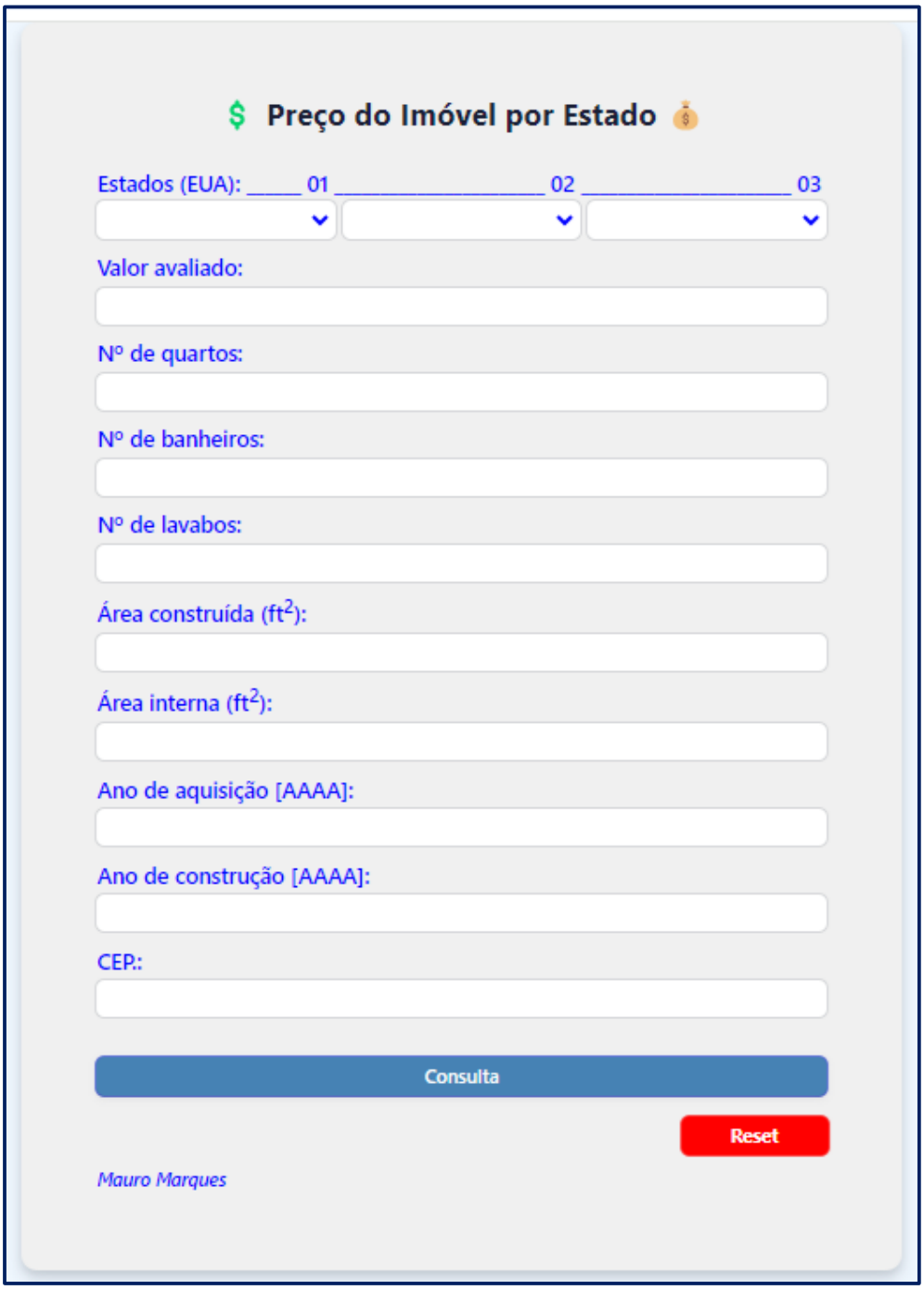


Figura 1.19: Carregamento de dados e disponibilização de serviços de consulta através de interface Web.

- Códigos ECL utilizados:

  fn_MyriadGetPriceReg.ecl
  fn_MyriadGetPriceReg_Web.ecl (código usado na interface Web)
  BWR_TestQueriesReg.ecl

1.5.2. Conclusão

Algoritmos baseados em Árvores de Decisão são provavelmente os algoritmos mais poderosos e fáceis de usar no arsenal de ML. Se os dados forem numéricos, houver uma quantidade razoável de dados e o objetivo for “Previsão” ao invés de “Explicação”, será difícil encontrar um melhor algoritmo do que Learning Trees.

Através desse estudo foi possível saber tudo o que se precisa saber para criar, treinar, prever, avaliar e testar modelos de Machine Learning fazendo uso da interface Myriad com base nos dados disponíveis e usá-los para prever valores quantitativos (Regressão). Caso se deseje usar um bundle de ML diferente, se concluirá que todos os bundles operam de maneira muito semelhante, com algumas variações menores.

No exemplo, foi treinado um número de modelos associados a quantidade de estados norte-americanos existentes no dataset “Property”, correspondente à cardinalidade de 44 atributos distintos para o campo ‘state’, sendo que todos os modelos eram homogêneos — com base no mesmo layout de registro e relacionados aos mesmos dados dependentes (preço de imóveis). Poderia ter sido usado datasets completamente diferentes que não tinham nada a ver um com o outro, ou poderia ter sido treinado modelos em dados dependentes completamente diferentes se assim fosse desejado

A simplicidade conceitual de ML é um tanto enganadora. Cada algoritmo tem suas próprias peculiaridades (ou premissas e restrições) que precisam ser levadas em consideração para maximizar a precisão preditiva. Além disso, quantificar a eficácia das previsões requer habilidades em ciência de dados e estatística que a maioria não possui. Por esses motivos, é muito importante que não seja utilizada a exploração de ML para produzir produtos ou reivindicar habilidades sem antes consultar especialistas no campo.

1.5.3. Apêndice

- Myriad RoadMap
  - Procedimento: Sequência de criação, execução (submit) e compilação (compile) dos códigos:
    modProperty.ecl
    BWR_BrowseData.ecl (submit)
    isCleanFilter.ecl
    CleanProperty.ecl
    modPrepData.ecl
    BWR_ViewData1.ecl (submit)
    modSegConvData.ecl
    BWR_ViewData2.ecl (submit)
    BWR_TrainReg.ecl (submit)
    fn_MyriadGetPriceReg.ecl (compile em hThor & Roxie)
    fn_MyriadGetPriceReg_Web.ecl (compile em hThor & Roxie)
    BWR_TestQueriesReg.ecl (submit)
- Códigos ECL

[modProperty.ecl]

```
//
EXPORT modProperty := MODULE
  EXPORT Layout := RECORD
   UNSIGNED8 personid;
```

```
  INTEGER8  propertyid;
  STRING10  house_number;
  STRING10  house_number_suffix;
  STRING2   predir;
  STRING30  street;
  STRING5   streettype;
  STRING2   postdir;
  STRING6   apt;
  STRING40  city;
  STRING2   state;
  STRING5   zip;
  UNSIGNED4 total_value;
  UNSIGNED4 assessed_value;
  UNSIGNED2 year_acquired;
  UNSIGNED4 land_square_footage;
  UNSIGNED4 living_square_feet;
  UNSIGNED2 bedrooms;
  UNSIGNED2 full_baths;
  UNSIGNED2 half_baths;
  UNSIGNED2 year_built;
 END;
 EXPORT File := DATASET('~CLASS::XYZ::ML::Property',Layout,CSV);
END;
//
```

[BWR_BrowseData.ecl]

```
IMPORT $;
//
// Visualização da extração dos dados
OUTPUT($.modProperty.File);
COUNT($.modProperty.File);          // 1.662.959 registros
//
```

[isCleanFilter.ecl]

```
IMPORT $;
//
Property := $.modProperty.File;
//
// Limpando os dados...
EXPORT isCleanFilter := Property.zip <> '' AND
                        Property.assessed_value <> 0 AND
                        Property.year_acquired <> 0 AND
                        Property.land_square_footage <> 0 AND
                        Property.living_square_feet <> 0 AND
                        Property.bedrooms <> 0 AND
                        Property.full_baths <> 0 AND
                        Property.year_Built <> 0;
//
```

[CleanProperty.ecl]

```
IMPORT $;
//
Property := $.modProperty.File;
//
EXPORT CleanProperty := Property($.isCleanFilter);
//
// CleanProperty := Property($.isCleanFilter);
```

```
// OUTPUT(CleanProperty);
// COUNT(CleanProperty);          // 575.814 registros
//
```

[modPrepData.ecl]

```
IMPORT $;
//
Property := $.modProperty.File;
//
EXPORT modPrepData := MODULE
EXPORT ML_Prop := RECORD
UNSIGNED8 propertyid;          // corresponde ao campo "idfield" usado na conversão matricial pelo
ML_Core.ToField
UNSIGNED3 zip;             // variável Categórica
UNSIGNED4 assessed_value;
UNSIGNED2 year_acquired;
UNSIGNED4 land_square_footage;
UNSIGNED4 living_square_feet;
UNSIGNED2 bedrooms;
UNSIGNED2 full_baths;
UNSIGNED2 half_baths;
UNSIGNED2 year_built;
UNSIGNED4 total_value;       // variável Dependente (a ser determinada)
UNSIGNED4 wi_id;               // Work-Item ID usado na Interface Myriad
END;
//
  EXPORT ML_PropExt := RECORD(ML_Prop)
    UNSIGNED4 rnd;                // número randômico
  END;
//
EXPORT MyriadPrepData := PROJECT($.CleanProperty, TRANSFORM(ML_PropExt,
                                    SELF.rnd   := RANDOM(),
                                    SELF.zip   := (UNSIGNED3)LEFT.zip,
                                    SELF.wi_id := MAP(LEFT.state = 'AE' =>  1, LEFT.state = 'WY' =>  2,
                                                  LEFT.state = 'TN' =>  3, LEFT.state = 'MI' =>  4,
                                                  LEFT.state = 'RI' =>  5, LEFT.state = 'SC' =>  6,
                                                  LEFT.state = 'VA' =>  7, LEFT.state = 'NH' =>  8,
                                                  LEFT.state = 'AP' =>  9, LEFT.state = 'NM' => 10,
                                                  LEFT.state = 'IL' => 11, LEFT.state = 'KS' => 12,
                                                  LEFT.state = 'ME' => 13, LEFT.state = 'AL' => 14,
                                                  LEFT.state = 'WA' => 15, LEFT.state = 'NY' => 16,
                                                  LEFT.state = 'MO' => 17, LEFT.state = 'PA' => 18,
                                                  LEFT.state = 'HI' => 19, LEFT.state = 'GU' => 20,
                                                  LEFT.state = 'VT' => 21, LEFT.state = 'CT' => 22,
                                                  LEFT.state = 'NC' => 23, LEFT.state = 'OR' => 24,
                                                  LEFT.state = 'IA' => 25, LEFT.state = 'DE' => 26,
                                                  LEFT.state = 'VI' => 27, LEFT.state = 'ID' => 28,
                                                  LEFT.state = 'MT' => 29, LEFT.state = 'AR' => 30,
                                                  LEFT.state = 'MS' => 31, LEFT.state = 'UT' => 32,
                                                  LEFT.state = 'NE' => 33, LEFT.state = 'IN' => 34,
                                                  LEFT.state = 'GA' => 35, LEFT.state = 'WV' => 36,
                                                  LEFT.state = 'NJ' => 37, LEFT.state = 'LA' => 38,
                                                  LEFT.state = 'WI' => 39, LEFT.state = 'AK' => 40,
                                                  LEFT.state = 'CA' => 41, LEFT.state = 'NV' => 42,
                                                  LEFT.state = 'FL' => 43, LEFT.state = 'MA' => 44,
                                                  LEFT.state = 'CO' => 45, LEFT.state = 'AZ' => 46,
                                                  LEFT.state = 'SD' => 47, LEFT.state = 'DC' => 48,
                                                  LEFT.state = 'KY' => 49, LEFT.state = 'MP' => 50,
```

```
                                                LEFT.state = 'ND' => 51, LEFT.state = 'AS' => 52,
                                                LEFT.state = 'TX' => 53, LEFT.state = 'PR' => 54,
                                                LEFT.state = 'MD' => 55, LEFT.state = 'OH' => 56,
                                                LEFT.state = 'MN' => 57, LEFT.state = 'OK' => 58,
                                                LEFT.state = 'AA' => 59, 0),
                                        SELF       := LEFT))
                    :PERSIST('~CLASS::XYZ::ML::MyriadPrepData');
//
END;
//
```

[BWR_ViewData1.ecl]

```
IMPORT $;
//
// Preparação dos dados
OUTPUT($.modPrepData.MyriadPrepData);
COUNT($.modPrepData.MyriadPrepData);          // 575.814 registros
//
```

[modSegConvData.ecl]

```
IMPORT $;
IMPORT ML_Core;
//
MyriadPrepData := $.modPrepData.MyriadPrepData;
//
// Torne os dados aleatórios, ordenando os registros pelo número randômico
MyriadPrepDataSort := SORT(MyriadPrepData(wi_id <> 0), rnd);
//
//
// Segregação dos dados - Considerando os primeiros 5.000 registros como amostra de Treinamento
MyriadTrainData := PROJECT(MyriadPrepDataSort[1..5000], $.modPrepData.ML_Prop)
                :PERSIST('~CLASS::XYZ::ML::MyriadTrain');             // layout sem o campo rnd
//
// Segregação dos dados - Considerando os 2.000 registros seguintes como amostra de Teste
MyriadTestData  := PROJECT(MyriadPrepDataSort[5001..7000], $.modPrepData.ML_Prop)
                :PERSIST('~CLASS::XYZ::ML::MyriadTest');              // layout sem o campo rnd
//
//
// Conversão Matricial dos campos numéricos
ML_Core.ToField(MyriadTrainData, MyriadTrainDataNF, wifield := wi_id);  // "idfield" não fornecido,
assume 1º campo = propertyid
ML_Core.ToField(MyriadTestData,  MyriadTestDataNF,  wifield := wi_id);  // "idfield" não fornecido,
assume 1º campo = propertyid
//
//
EXPORT modSegConvData := MODULE
EXPORT MyriadIndTrainDataNF := MyriadTrainDataNF(number < 10);    // excluindo o campo
propertyid
//
EXPORT MyriadDepTrainDataNF := PROJECT(MyriadTrainDataNF(number = 10),
TRANSFORM(RECORDOF(LEFT),
                                                SELF.number := 1,
                                                SELF := LEFT));
//
EXPORT MyriadIndTestDataNF  := MyriadTestDataNF(number < 10);      // excluindo o campo
propertyid
//
```

```
EXPORT MyriadDepTestDataNF  := PROJECT(MyriadTestDataNF(number = 10),
TRANSFORM(RECORDOF(LEFT),
                                                SELF.number := 1,
                                                SELF := LEFT));
END;
//
```

[BWR_ViewData2.ecl]

```
IMPORT $;
//
// Segregação dos dados e Conversão matricial dos campos numéricos
OUTPUT($.modSegConvData.MyriadIndTrainDataNF, NAMED('MyriadIndTrainData'));
OUTPUT($.modSegConvData.MyriadDepTrainDataNF, NAMED('MyriadDepTrainData'));
OUTPUT($.modSegConvData.MyriadIndTestDataNF,  NAMED('MyriadIndTestData'));
OUTPUT($.modSegConvData.MyriadDepTestDataNF,  NAMED('MyriadDepTestData'));
//
```

[BWR_TrainReg.ecl]

```
IMPORT $;
IMPORT ML_Core;
IMPORT LearningTrees AS LT;
//
// Selecionando o algoritmo
//   Sintaxe: RegressionForest(UNSIGNED numTrees=100, UNSIGNED featuresPerNode=0,
//                         UNSIGNED maxDepth=100, SET OF UNSIGNED nominalFields=[])
// MyriadLearnerR := LT.RegressionForest();          // parâmetros default
// MyriadLearnerR := LT.RegressionForest(,,,[1]);    // zip não é realmente quantitativo, mas qualitativo
MyriadLearnerR := LT.RegressionForest(10,,10,[1]);
//
//
// Obtendo o modelo composto (composto por 45 modelos = números de estados existentes no dataset)
MyriadModelR :=
MyriadLearnerR.GetModel($.modSegConvData.MyriadIndTrainDataNF,$.modSegConvData.MyriadDep
TrainDataNF);
OUTPUT(MyriadModelR,,'~CLASS::XYZ::ML::MyriadModelR',
NAMED('Myriad_Modelo_Treinado'),OVERWRITE);
OUTPUT(COUNT(DEDUP(SORT((MyriadModelR), wi), wi)), NAMED('Numero_de_Modelos'));
//
//
// Testando o modelo
MyriadPredictedDeps :=
MyriadLearnerR.Predict(MyriadModelR,$.modSegConvData.MyriadIndTestDataNF);
OUTPUT(MyriadPredictedDeps, NAMED('Myriad_Valores_Previstos'));
//
//
// Avaliando o modelo
MyriadAssessmentR :=
ML_Core.Analysis.Regression.Accuracy(MyriadPredictedDeps,$.modSegConvData.MyriadDepTestData
NF);
OUTPUT(MyriadAssessmentR, NAMED('Myriad_Avaliacao_dos_Modelos'));
OUTPUT(SORT(MyriadAssessmentR, -r2), NAMED('Ordenacao_pela_acuracia_dos_Modelos'));
//
```

[fn_MyriadGetPriceReg.ecl]

```
IMPORT $,STD,Visualizer;
IMPORT ML_Core;
IMPORT LearningTrees AS LT;
```

```
//
// Função de predição de preços de imóveis
EXPORT fn_MyriadGetPriceReg(zip,
                        assess_val,
                        year_acq,
                        land_sq_ft,
                        living_sq_ft,
                        bedrooms,
                        full_baths,
                        half_baths,
                        year_built,
                        // state_code = 0) := FUNCTION
                        STRING2 _state_ = 'XX') := FUNCTION
//
WUID := WORKUNIT; // obter o Id da WorkUnit
//
// Transformação dos parâmetros de entrada no formato de ML data frame
  MyriadInSet := [zip, assess_val, year_acq, land_sq_ft, living_sq_ft, bedrooms, full_baths, half_baths,
year_built];
  MyriadInDS  := DATASET(MyriadInSet, {REAL8 MyriadInValue});
ML_Core.Types.NumericField PrepDataReg(RECORDOF(MyriadInDS) Le, INTEGER cnt) :=
TRANSFORM
// SELF.wi     := state_code,
    SELF.wi     := CASE(STD.Str.ToUpperCase(_state_), 'AE' => 1,'WY' => 2,'TN' => 3,'MI' => 4,'RI' =>
5,'SC' => 6,
                                        'VA' => 7,'NH' => 8,'AP' => 9,'NM' => 10,'IL' => 11,'KS' => 12,
                                        'ME' => 13,'AL' => 14,'WA' => 15,'NY' => 16,'MO' => 17,'PA' => 18,
                                        'HI' => 19,'GU' => 20,'VT' => 21,'CT' => 22,'NC' => 23,'OR' => 24,
                                        'IA' => 25,'DE' => 26,'VI' => 27,'ID' => 28,'MT' => 29,'AR' => 30,
                                        'MS' => 31,'UT' => 32,'NE' => 33,'IN' => 34,'GA' => 35,'WV' => 36,
                                        'NJ' => 37,'LA' => 38,'WI' => 39,'AK' => 40,'CA' => 41,'NV' => 42,
                                        'FL' => 43,'MA' => 44,'CO' => 45,'AZ' => 46,'SD' => 47,'DC' => 48,
                                        'KY' => 49,'MP' => 50,'ND' => 51,'AS' => 52,'TX' => 53,'PR' => 54,
                                        'MD' => 55,'OH' => 56,'MN' => 57,'OK' => 58,'AA' => 59,0),
SELF.id    := 1,
SELF.number := cnt,
SELF.value  := Le.MyriadInValue;
END;
MyriadNewIndDataReg := PROJECT(MyriadInDS, PrepDataReg(LEFT, COUNTER));
//
// Predição e retorno do valor do imóvel consultado
MyriadModelR      :=
DATASET('~CLASS::XYZ::ML::MyriadModelR',ML_Core.Types.Layout_Model2,FLAT,PRELOAD);
MyriadLearnerR    := LT.RegressionForest(10,,10,[1]);
MyriadPredictDeps := MyriadLearnerR.Predict(MyriadModelR, MyriadNewIndDataReg);
//
//
Action01 := OUTPUT(MyriadPredictDeps,{preco:=ROUND(value)}, NAMED('Preco_Imovel'));
Action02 := OUTPUT(MyriadPredictDeps,{_state_, preco:=ROUND(value)},
NAMED('Choropleth_USStates'));
Action03 := Visualizer.Choropleth.USStates('Myriad',, 'Choropleth_USStates');
Action04 := OUTPUT(WUID, Named('WUID'));
//
//
// RETURN OUTPUT(MyriadPredictDeps,{preco:=ROUND(value)});
  RETURN SEQUENTIAL(Action01,PARALLEL(Action02,Action03),Action04);
END;
//
```

[fn_MyriadGetPriceReg_Web.ecl]

```
IMPORT $,STD,Visualizer;
IMPORT ML_Core;
IMPORT LearningTrees AS LT;
//
// Função de predição de preços de imóveis
EXPORT fn_MyriadGetPriceReg_Web(zip,
                        assess_val,
                        year_acq,
                        land_sq_ft,
                        living_sq_ft,
                        bedrooms,
                        full_baths,
                        half_baths,
                        year_built,
                        // state_code = 0) := FUNCTION
                        STRING2 _state_01 = 'XX',
                        STRING2 _state_02 = 'YY',
                        STRING2 _state_03 = 'ZZ') := FUNCTION
//
WUID := WORKUNIT; // obter o Id da WorkUnit
//
// Transformação dos parâmetros de entrada no formato de ML data frame
  MyriadInSet := [zip, assess_val, year_acq, land_sq_ft, living_sq_ft, bedrooms, full_baths, half_baths,
year_built];
  MyriadInDS  := DATASET(MyriadInSet, {REAL8 MyriadInValue});
//
ML_Core.Types.NumericField  PrepDataReg01(RECORDOF(MyriadInDS)  Le,  INTEGER  cnt01)  :=
TRANSFORM
// SELF.wi     := state_code,
   SELF.wi     := CASE(STD.Str.ToUpperCase(_state_01), 'AE' => 1,'WY' => 2,'TN' => 3,'MI' => 4,'RI' =>
5,'SC' => 6,
                                        'VA' => 7,'NH' => 8,'AP' => 9,'NM' => 10,'IL' => 11,'KS' => 12,
                                        'ME' => 13,'AL' => 14,'WA' => 15,'NY' => 16,'MO' => 17,'PA' => 18,
                                        'HI' => 19,'GU' => 20,'VT' => 21,'CT' => 22,'NC' => 23,'OR' => 24,
                                        'IA' => 25,'DE' => 26,'VI' => 27,'ID' => 28,'MT' => 29,'AR' => 30,
                                        'MS' => 31,'UT' => 32,'NE' => 33,'IN' => 34,'GA' => 35,'WV' => 36,
                                        'NJ' => 37,'LA' => 38,'WI' => 39,'AK' => 40,'CA' => 41,'NV' => 42,
                                        'FL' => 43,'MA' => 44,'CO' => 45,'AZ' => 46,'SD' => 47,'DC' => 48,
                                        'KY' => 49,'MP' => 50,'ND' => 51,'AS' => 52,'TX' => 53,'PR' => 54,
                                        'MD' => 55,'OH' => 56,'MN' => 57,'OK' => 58,'AA' => 59,0),
SELF.id    := 1,
SELF.number := cnt01,
SELF.value  := Le.MyriadInValue;
END;
MyriadNewIndDataReg01 := PROJECT(MyriadInDS, PrepDataReg01(LEFT, COUNTER));
//
ML_Core.Types.NumericField  PrepDataReg02(RECORDOF(MyriadInDS)  Le,  INTEGER  cnt02)  :=
TRANSFORM
// SELF.wi     := state_code,
   SELF.wi     := CASE(STD.Str.ToUpperCase(_state_02), 'AE' => 1,'WY' => 2,'TN' => 3,'MI' => 4,'RI' =>
5,'SC' => 6,
                                        'VA' => 7,'NH' => 8,'AP' => 9,'NM' => 10,'IL' => 11,'KS' => 12,
                                        'ME' => 13,'AL' => 14,'WA' => 15,'NY' => 16,'MO' => 17,'PA' => 18,
                                        'HI' => 19,'GU' => 20,'VT' => 21,'CT' => 22,'NC' => 23,'OR' => 24,
                                        'IA' => 25,'DE' => 26,'VI' => 27,'ID' => 28,'MT' => 29,'AR' => 30,
                                        'MS' => 31,'UT' => 32,'NE' => 33,'IN' => 34,'GA' => 35,'WV' => 36,
```

```
                                    'NJ' => 37,'LA' => 38,'WI' => 39,'AK' => 40,'CA' => 41,'NV' => 42,
                                    'FL' => 43,'MA' => 44,'CO' => 45,'AZ' => 46,'SD' => 47,'DC' => 48,
                                    'KY' => 49,'MP' => 50,'ND' => 51,'AS' => 52,'TX' => 53,'PR' => 54,
                                    'MD' => 55,'OH' => 56,'MN' => 57,'OK' => 58,'AA' => 59,0),
SELF.id    := 1,
SELF.number := cnt02,
SELF.value  := Le.MyriadInValue;
END;
MyriadNewIndDataReg02 := PROJECT(MyriadInDS, PrepDataReg02(LEFT, COUNTER));
//
ML_Core.Types.NumericField PrepDataReg03(RECORDOF(MyriadInDS) Le, INTEGER cnt03) :=
TRANSFORM
// SELF.wi     := state_code,
   SELF.wi     := CASE(STD.Str.ToUpperCase(_state_03), 'AE' => 1,'WY' => 2,'TN' => 3,'MI' => 4,'RI' =>
5,'SC' => 6,
                                    'VA' => 7,'NH' => 8,'AP' => 9,'NM' => 10,'IL' => 11,'KS' => 12,
                                    'ME' => 13,'AL' => 14,'WA' => 15,'NY' => 16,'MO' => 17,'PA' => 18,
                                    'HI' => 19,'GU' => 20,'VT' => 21,'CT' => 22,'NC' => 23,'OR' => 24,
                                    'IA' => 25,'DE' => 26,'VI' => 27,'ID' => 28,'MT' => 29,'AR' => 30,
                                    'MS' => 31,'UT' => 32,'NE' => 33,'IN' => 34,'GA' => 35,'WV' => 36,
                                    'NJ' => 37,'LA' => 38,'WI' => 39,'AK' => 40,'CA' => 41,'NV' => 42,
                                    'FL' => 43,'MA' => 44,'CO' => 45,'AZ' => 46,'SD' => 47,'DC' => 48,
                                    'KY' => 49,'MP' => 50,'ND' => 51,'AS' => 52,'TX' => 53,'PR' => 54,
                                    'MD' => 55,'OH' => 56,'MN' => 57,'OK' => 58,'AA' => 59,0),
SELF.id    := 1,
SELF.number := cnt03,
SELF.value  := Le.MyriadInValue;
END;
MyriadNewIndDataReg03 := PROJECT(MyriadInDS, PrepDataReg03(LEFT, COUNTER));
//
//
// Predição e retorno do valor do imóvel consultado - State 01
MyriadModel01                                                                                   :=
DATASET('~CLASS::XYZ::ML::MyriadModelR',ML_Core.Types.Layout_Model2,FLAT,PRELOAD);
MyriadLearner01     := LT.RegressionForest(10,,10,[1]);
MyriadPredictDeps01 := MyriadLearner01.Predict(MyriadModel01, MyriadNewIndDataReg01);
//
// Predição e retorno do valor do imóvel consultado - State 02
MyriadModel02                                                                                   :=
DATASET('~CLASS::XYZ::ML::MyriadModelR',ML_Core.Types.Layout_Model2,FLAT,PRELOAD);
MyriadLearner02     := LT.RegressionForest(10,,10,[1]);
MyriadPredictDeps02 := MyriadLearner02.Predict(MyriadModel02, MyriadNewIndDataReg02);
//
// Predição e retorno do valor do imóvel consultado - State 03
MyriadModel03                                                                                   :=
DATASET('~CLASS::XYZ::ML::MyriadModelR',ML_Core.Types.Layout_Model2,FLAT,PRELOAD);
MyriadLearner03     := LT.RegressionForest(10,,10,[1]);
MyriadPredictDeps03 := MyriadLearner03.Predict(MyriadModel03, MyriadNewIndDataReg03);
//
//
Action01a := OUTPUT(MyriadPredictDeps01,{preco:=ROUND(value)}, NAMED('Preco_Imovel_01'));
Action01b := OUTPUT(MyriadPredictDeps02,{preco:=ROUND(value)}, NAMED('Preco_Imovel_02'));
Action01c := OUTPUT(MyriadPredictDeps03,{preco:=ROUND(value)}, NAMED('Preco_Imovel_03'));
//
Action02a       :=      OUTPUT(MyriadPredictDeps01,{states:=_state_01,      preco:=ROUND(value)},
NAMED('Choropleth_USStates'),EXTEND);
Action02b       :=      OUTPUT(MyriadPredictDeps02,{states:=_state_02,      preco:=ROUND(value)},
NAMED('Choropleth_USStates'),EXTEND);
```

```
Action02c := OUTPUT(MyriadPredictDeps03,{states:=_state_03, preco:=ROUND(value)},
NAMED('Choropleth_USStates'),EXTEND);
//
Action03  := Visualizer.Choropleth.USStates('Myriad',, 'Choropleth_USStates');
Action04  := OUTPUT(WUID, Named('WUID'));
//
//
// RETURN OUTPUT(MyriadPredictDeps,{preco:=ROUND(value)});
  RETURN SEQUENTIAL(Action01a,Action01b,Action01c,
                    Action02a,Action02b,Action02c,
                    Action03,
                    Action04);
END;
//
```

[BWR_TestQueriesReg.ecl]

```
IMPORT $;
//
// Teste da Função
// Zip | Assess_val | Year_acq | Land_sq_ft | Living_sq_ft | Bedrooms | Full_baths | Half_baths | Year_built
| States 01/02/03
//
// $.fn_MyriadGetPriceReg(95451,118720,2011,14774,1437,3,2,1,1968,'CA');
$.fn_MyriadGetPriceReg_Web(95451,118720,2011,14774,1437,3,2,1,1968,'CA','NY','FL');
//
/*
_state_01:    CA
_state_02:    NY
_state_03:    FL
assess_val:  118720
bedrooms:    3
full_baths:  2
half_baths:  1
land_sq_ft:  14774
living_sq_ft:1437
year_acq:    2011
year_built:  1968
zip:         95451
*/
//
```

1.6. Referências

Introduction to HPCC Systems Open Source Big Data Platform. Disponível em: https://cdn.hpccsystems.com/whitepapers/wp_introduction_HPCC.pdf Acesso em: 25 set. 2024.

Machine Learning Demystified. Disponível em: https://hpccsystems.com/resources/machine-learning-demystified/ Acesso em: 25 set. 2024.

HPCC Systems Machine Learning Library. Disponível em: https://hpccsystems.com/download/free-modules/hpcc-systems-machine-learning-library/ Acesso em: 25 set. 2024.

Using HPCC Systems Machine Learning. Disponível em:
https://hpccsystems.com/resources/using-hpcc-systems-machine-learning/
Acesso em: 25 set. 2024.

Introducing the new, improved HPCC Systems Machine Learning Library | HPCC Systems. Disponível em:
https://hpccsystems.com/resources/introducing-the-new-improved-hpcc-systems-machine-learning-library/
Acesso em: 25 set. 2024.

ECL-ML Machine Learning Module. Disponível em:
https://hpccsystems.com/resources/ecl-ml-machine-learning-module/
Acesso em: 25 set. 2024.

ML_Core Documentation. Disponível em:
https://cdn.hpccsystems.com/pdf/ml/ML_Core.pdf
Acesso em: 25 set. 2024.

Source code: HPCC Systems ML_Core repository on GitHub. Disponível em:
https://github.com/hpcc-systems/ML_Core
Acesso em: 25 set. 2024.

Introduction to using PBblas on HPCC Systems. Disponível em:
https://hpccsystems.com/resources/introduction-to-using-pbblas-on-hpcc-systems/
Acesso em: 25 set. 2024.

Documentation: PBblas Documentation. Disponível em:
https://cdn.hpccsystems.com/pdf/ml/PBblas.pdf
Acesso em: 25 set. 2024.

Source code: HPCC Systems PBblas repository on GitHub. Disponível em:
https://github.com/hpcc-systems/PBblas
Acesso em: 25 set. 2024.

Learning Trees — A guide to Decision Tree based Machine Learning. Disponível em:
https://hpccsystems.com/resources/learning-trees-a-guide-to-decision-tree-based-machine-learning/
Acesso em: 25 set. 2024.

Documentation: LearningTrees Documentation. Disponível em:
https://cdn.hpccsystems.com/pdf/ml/LearningTrees.pdf
Acesso em: 25 set. 2024.

Source code: LearningTrees repository on GitHub. Disponível em:
https://github.com/hpcc-systems/LearningTrees
Acesso em: 25 set. 2024.

Understanding the Myriad Interface feature of HPCC Systems Machine Learning | HPCC Systems. Disponível em:
https://hpccsystems.com/resources/understanding-the-myriad-interface-feature-of-hpcc-systems-machine-learning/
Acesso em: 25 set. 2024.

Capítulo 2

Perfilamento e Visualização da Escalabilidade de Aplicações Paralelas com o Parallel Scalability Suite

Felipe Santos-da-Silva, Anderson B. N. da Silva, Vitor R. G. da Silva, Carlos A. Valderrama e Samuel Xavier-de-Souza

2.1. Desempenho vs. Escalabilidade

O principal objetivo ao criar um programa paralelo é, em geral, melhorar o **desempenho**, ou seja, reduzir o tempo necessário para realizar uma tarefa. No entanto, apenas o desempenho não é suficiente para avaliar a qualidade de um programa paralelo. É igualmente importante considerar a sua **escalabilidade**, que está diretamente relacionada a **eficiência** do programa à medida que variamos o número de recursos computacionais disponíveis e o tamanho do problema que ele resolve.

Para esclarecer melhor a diferença entre desempenho e escalabilidade, considere a seguinte analogia: imagine que em um restaurante há uma equipe responsável por lavar pratos. O desempenho é medido pela rapidez com que a equipe executa essa tarefa. Se apenas uma pessoa está lavando os pratos, o desempenho será avaliado pela quantidade de pratos que essa pessoa consegue lavar em um determinado período. Se ela trabalha rapidamente e sem cometer erros, dizemos que o desempenho é alto, ou seja, estamos analisando a eficiência do trabalho individual, sem considerar outros fatores.

Agora, imagine que o número de pratos para lavar aumenta significativamente, e para lidar com essa demanda extra, mais pessoas são adicionadas à equipe. A escalabilidade se refere à capacidade da equipe de continuar lavando os pratos de maneira eficiente à medida que mais pessoas são colocadas para ajudar. Se essas novas pessoas conseguirem dividir o trabalho de forma equilibrada, sem atrapalhar umas às outras, a equipe tem boa escalabilidade.

Por outro lado, se ao adicionar mais pessoas elas começarem a se esbarrar, disputar o espaço na pia ou esperar pelas mesmas ferramentas, a eficiência da tarefa será prejudicada. Isso acontece porque o aumento de pessoas não torna necessariamente o processo mais rápido, mas sim mais confuso. Da mesma forma, em um programa paralelo, a sobrecarga causada pela necessidade de coordenar o trabalho entre diferentes núcleos de processamento pode reduzir o benefício esperado de adicionar mais recursos.

2.1.1. Speedup

Um dos métodos mais comuns para medir o desempenho de um programa paralelo é através do conceito de ***speedup***, que compara o tempo de execução sequencial com o tempo de execução paralelo. Idealmente, se conseguirmos dividir o trabalho de maneira equilibrada entre p núcleos e minimizar a sobrecarga adicional, o programa paralelo deveria ser p vezes mais rápido que o programa sequencial. Se o tempo de execução sequencial for $T_{\text{sequencial}}$ e o tempo de execução paralelo for T_{paralelo}, o ***speedup*** pode ser expresso como:

$$S = \frac{T_{\text{sequencial}}}{T_{\text{paralelo}}}.$$

Quando o *speedup* é exatamente igual ao número de núcleos, ou seja, $S = p$, dizemos que o programa apresenta *speedup* linear. No entanto, na prática, obter essa aceleração ideal é raro, devido a fatores como sobrecarga de comunicação, sincronização entre *threads* e seções críticas que exigem mecanismos como *mutexes*. Esses elementos introduzem uma forma de execução sequencial dentro do programa paralelo, diminuindo o *speedup* real.

2.1.2. Eficiência e Escalabilidade

A **eficiência** de um programa paralelo está relacionada a como os recursos (como os núcleos) são utilizados em comparação com a aceleração obtida. Formalmente, podemos definir a eficiência como:

$$E = \frac{S}{P}.$$

Uma eficiência de 100% (ou 1) indicaria que cada núcleo está contribuindo de maneira ideal para o desempenho, algo que raramente ocorre devido à sobrecarga mencionada anteriormente.

Na analogia dos pratos 2.1, em um dos cenários, ao adicionar mais pessoas para lavar pratos, a eficiência da equipe diminuía devido à falta de espaço ou à sobrecarga de coordenação. Isso reflete o que ocorre em sistemas computacionais: muitas vezes, aumentar apenas os recursos (como mais núcleos ou máquinas) pode prejudicar a eficiência se não houver um aumento proporcional no tamanho do problema a ser resolvido. Máquinas maiores, como supercomputadores, precisam geralmente de problemas maiores para manter níveis elevados de eficiência. A eficiência, nesse caso, pode variar conforme os recursos e o tamanho do problema escalam, sendo necessário um equilíbrio cuidadoso para garantir que o ganho de desempenho compense a sobrecarga adicional de coordenação.

Além disso, a **escalabilidade** é um conceito essencial ao se avaliar o desempenho de programas paralelos. Ela refere-se à capacidade de um sistema ou algoritmo de sustentar ou melhorar sua eficiência à medida que o número de recursos disponíveis, como processadores ou *threads*, aumenta[1].

A análise de escalabilidade exige a avaliação de diversas configurações de uma aplicação, enquanto a análise de desempenho se concentra em uma avaliação detalhada de uma única configuração. Embora seja possível utilizar ferramentas de criação de perfis de desempenho para a análise de escalabilidade, essa abordagem apresenta certas limitações. Devido à natureza da análise de escalabilidade, o usuário precisa executar e coletar manualmente informações de múltiplas configurações. Além disso, essas ferramentas

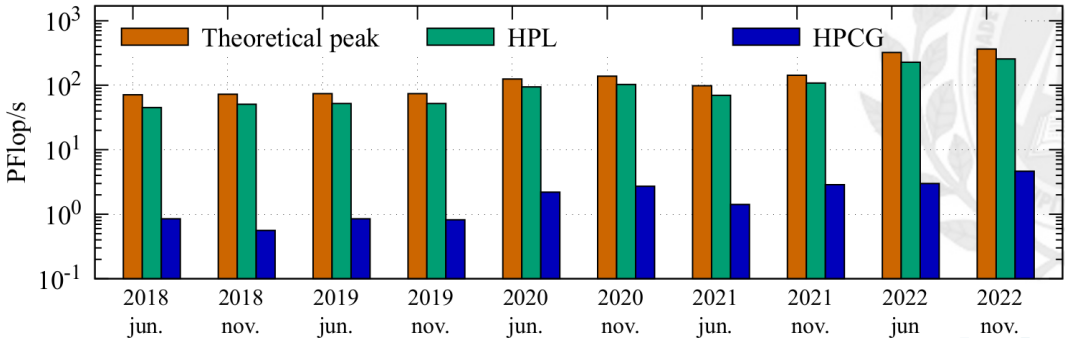


Figura 2.1. Comparação do desempenho médio dos 10 principais supercomputadores da lista Top500 (2018-2022) em termos de pico teórico, HPL e HPGC. O desempenho teórico de pico é consistentemente maior que os resultados medidos pelos benchmarks HPL e HPGC

fornecem dados excessivamente detalhados, muitas vezes além do necessário para avaliar escalabilidade. Outro ponto relevante é a ausência de artefatos visuais nativos que facilitem a interpretação dos resultados relacionados à escalabilidade.

A Figura 2.1 ilustra a comparação entre o desempenho teórico de pico e os resultados obtidos através dos benchmarks *HPL* (*High Performance Linpack*) e *HPGC* (*High Performance Conjugate Gradients*) para os 10 principais supercomputadores da lista Top500, no período de 2018 a 2022. Observa-se uma diferença significativa entre o pico teórico, representado em laranja, e os desempenhos efetivos, medidos pelo *HPL* (em verde) e *HPGC* (em azul), com o último apresentando os menores resultados. Isso evidencia que, embora grandes máquinas sejam capazes de resolver muitas tarefas simultaneamente, seu desempenho em problemas de grande escala, particularmente os mais complexos, como os medidos pelo *HPGC*, é consideravelmente inferior ao esperado. Além disso, o gráfico demonstra que, ao longo dos anos, essa tendência permanece constante, indicando limitações na eficiência dos sistemas em explorar todo o potencial teórico disponível.

A escalabilidade paralela tem se tornado uma preocupação crescente em razão do aumento da demanda por desempenho em sistemas computacionais. Com o avanço de processadores *multicore*, a eficiência no uso de múltiplos núcleos tornou-se crucial para garantir que aplicações possam lidar com volumes de dados cada vez maiores e tarefas mais complexas.

Adicionalmente, à medida que sistemas paralelos se tornam mais comuns em áreas como inteligência artificial, big data e simulações científicas, a capacidade de escalar eficientemente recursos paralelos afeta diretamente a produtividade e o custo-benefício das operações. No entanto, observa-se uma escassez de ferramentas dedicadas especificamente à análise e otimização da escalabilidade paralela, o que aumenta a complexidade desse processo.

As principais fontes de ineficiência em sistemas paralelos estão relacionadas à comunicação e sincronização entre os núcleos. A comunicação excessiva entre diferentes núcleos pode gerar um *overhead* significativo, diminuindo a eficiência global da aplicação. Além disso, a necessidade de sincronização frequente entre as *threads* pode causar

períodos de inatividade enquanto se espera a finalização de tarefas em diferentes unidades de processamento.

Outro fator importante é o desbalanceamento de carga, que ocorre quando diferentes partes da aplicação recebem quantidades desiguais de trabalho, levando a um uso desigual dos recursos. Isso resulta em computação extra para alguns processos, enquanto outros permanecem ociosos, aguardando a sincronização. Essas ineficiências aumentam o tempo total de execução e diminuem o *speedup*, dificultando a escalabilidade paralela. Esse ciclo de ineficiências, quando identificadas, podem ser mitigadas por técnicas de balanceamento de carga, que distribuem o trabalho de maneira uniforme, e técnicas de ocultação de latência, que permitem que a computação prossiga enquanto ocorrem transferências de dados.

2.1.3. Lei de Amdahl

Em 1967 Gene Amdahl fez uma observação que ficaria conhecida como a **Lei de Amdahl**[2]. A lei estabelece um limite teórico no *speedup* que pode ser alcançado com a paralelização, considerando que uma fração do código de um programa é "inerentemente sequencial"e, portanto, não pode ser paralelizada. De acordo com Amdahl, mesmo que uma grande parte do código seja paralelizada, o tempo de execução total será limitado pela parte que permanece sequencial. Isso significa que o *speedup* total não pode crescer indefinidamente, independentemente do número de processadores adicionados.

Suponhamos que 90% de um programa possa ser paralelizado e que o tempo de execução em um único processador seja $T_{\text{sequencial}} = 20$ segundos. Usando a Lei de Amdahl, podemos calcular o tempo de execução paralelizado da seguinte forma:

$$T_{\text{paralelo}} = 0,9 \times \frac{T_{\text{sequencial}}}{p} + 0,1 \times T_{\text{sequencial}} = \frac{20}{\frac{18}{p} + 2},$$

onde p representa o número de processadores. E o *speedup* será:

$$S = \frac{20}{0,9 \times \frac{T_{\text{sequencial}}}{p} + 0,1 \times T_{\text{sequencial}}} = \frac{20}{\frac{18}{p} + 2}.$$

Conforme p aumenta, o *speedup* se aproxima de um valor máximo. Neste exemplo, mesmo com 1000 processadores, o *speedup* não excede 10, evidenciando o limite imposto pela parte sequencial do código. Mais genericamente, se uma fração r do programa for intrinsecamente sequencial, o limite máximo de *speedup* será dado por $S \leq \frac{1}{r}$.

2.1.4. Lei de Gustafson

Embora a **Lei de Amdahl** seja amplamente utilizada para modelar o speedup em sistemas paralelos, ela pressupõe que o tamanho do problema permanece constante à medida que o número de processadores aumenta, o que pode limitar o potencial de paralelização. Para abordar essa limitação, John Gustafson propôs, em 1988, uma alternativa chamada **Lei de Gustafson**, que considera o aumento no tamanho do problema conforme mais processadores são adicionados.

A Lei de Gustafson sugere que, ao invés de focar na fração sequencial do código que não pode ser paralelizada, deveríamos considerar o fato de que o tamanho total do problema pode crescer com o aumento do número de processadores, fazendo com que a parte paralelizável do programa se torne mais significativa. Isso implica que, para problemas maiores, a fração do tempo gasto nas partes sequenciais do código diminui, possibilitando um *speedup* maior do que o previsto pela Lei de Amdahl.

A Lei de Gustafson é expressa pela seguinte fórmula para o speedup S:

$$S = p - (p-1) \times r,$$

onde p é o número de processadores e r, a fração sequencial do programa. Diferente da Lei de Amdahl, que impõe um limite rígido ao *speedup*, a Lei de Gustafson[3] sugere que o *speedup* cresce linearmente com o número de processadores, assumindo que o tamanho do problema cresce proporcionalmente à quantidade de recursos computacionais disponíveis.

Suponhamos que $r = 0,1$ ou seja, 10% do código é sequencial e o restante pode ser paralelizado. Para $p = 100$ processadores, o *speedup* S pode ser calculado como:

$$S = 100 - (100-1) \times 0,1 = 100 - 99 \times 0,1 = 100 - 9,9 = 90,1.$$

Esse resultado mostra que o *speedup* é de $90,1$, significativamente maior do que o limite previsto pela Lei de Amdahl para o mesmo número de processadores.

2.1.5. Tipos de Escalabilidade

Em 2.1.2 fomos apresentados rapidamente ao conceito de escalabilidade de maneira mais informal. Embora o termo "escalável"seja frequentemente usado dessa forma, aqui ele será definido de maneira mais precisa, conforme a literatura de computação paralela.

Suponha que um programa paralelo seja executado com um número fixo de *threads* e um problema de tamanho fixo, e que tenha uma eficiência inicial E. Agora, se aumentarmos o número de *threads* e também ajustarmos o tamanho do problema de maneira adequada, podemos verificar se a eficiência E é mantida. Caso seja possível encontrar uma razão correspondente de aumento do tamanho do problema para que a eficiência permaneça constante, o programa é considerado escalável.

A eficiência E de um programa paralelo pode ser descrita pela seguinte equação:

$$E = \frac{n}{\frac{n}{p} + 1} = \frac{n}{n+p}.$$

onde n é o tamanho do problema, p é o número de *threads*, $T_{\text{sequencial}} = n$ e $T_{\text{paralelo}} = \frac{n}{p} + 1$ são os tempos de execução.

Para determinar se o programa é escalável, aumentamos o número de processadores por um fator k e verificamos o fator x pelo qual o tamanho do problema deve aumentar para manter a eficiência E constante. Assim, o número de processadores se torna kp e o tamanho do problema se torna xn. A equação que devemos resolver para x é:

$$E = \frac{n}{n+p} = \frac{xn}{xn+kp}.$$

Quando $x = k$, a eficiência E permanece inalterada, ou seja, programa é escalável. Podemos simplificar a equação para:

$$\frac{xn}{xn+kp} = \frac{kn}{k(n+p)} = \frac{n}{n+p}.$$

Ou seja, se aumentarmos o tamanho do problema n na mesma proporção que aumentarmos o número de *threads* p, a eficiência se mantém constante, indicando que o programa é escalável.

2.1.6. Análise de Eficiência e Escalabilidade Paralela

Um programa é dito **fortemente escalável** quando consegue manter a eficiência E mesmo sem aumentar o tamanho do problema. Isso significa que, ao adicionar mais processadores ou I, o programa mantém uma alta eficiência sem precisar aumentar proporcionalmente o tamanho da tarefa a ser processada. Em um cenário de escalabilidade forte ideal, a eficiência E não se altera mesmo com o aumento número de processos p.

Por outro lado, um programa é considerado **fracamente escalável** quando consegue manter a eficiência E somente quando o tamanho do problema aumenta na mesma proporção que o número de processadores ou *threads*. Isso quer dizer que embora mais recursos sejam adicionados, o problema precisa crescer proporcionalmente para que a eficiência não decresça.

2.1.7. Avaliando a Escalabilidade com Tabelas

Uma forma de avaliar a escalabilidade de um programa é utilizar tabelas que correlacionam o número de núcleos de processamento com o tamanho do problema. O primeiro passo é coletar os tempos de execução do programa em diversas configurações de número de núcleos e tamanhos do problema como ilustrado na tabela abaixo.

# Núcleos	1x	2x	4x	8x	16x	32x
1	100.00	400.00	1600.00	6400.00	25600.00	102400.00
2	51.00	201.00	803.00	3201.00	12801.00	51201.00
4	27.00	102.00	402.00	1602.00	6402.00	25602.00
8	15.50	53.00	203.00	803.00	3203.00	12803.00
16	10.25	29.00	104.00	404.00	1604.00	6404.00
32	8.13	17.50	55.00	205.00	805.00	3205.00
64	7.56	12.25	31.00	106.00	406.00	1606.00
128	7.78	10.13	19.50	57.00	207.00	807.00

Tabela 2.1. Tempos de execução para diferentes números de núcleos e tamanhos de problema.

A equação que representa o tempo de execução sequencial é $T_{\text{sequencial}} = n$, on n é o tamanho do problema. Já o tempo de execução paralelo $T_{\text{paralelo}} = \frac{n}{p} + log_2 p$, onde p representa o número de núcleos utilizados.

Uma vez que os tempos de execução foram obtidos, como descrito na Tabela 2.1, podemos agora calcular o *speedup* pata cada combinação de número de núcleos e tamanho

do problema, O *speedup* é definido como a razão entre o tempo de execução sequencial $T_{\text{sequencial}}$ e o tempo de execução paralelo T_{paralelo}:

$$S = \frac{T_{\text{sequencial}}}{T_{\text{paralelo}}}.$$

Segue a tabela que representa os valores para *speedup*

# Núcleos	1x	2x	4x	8x	16x	32x
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.96	1.99	2.00	2.00	2.00	2.00
4	3.70	3.92	3.98	4.00	4.00	4.00
8	6.45	7.35	7.80	7.97	7.99	8.00
16	9.76	13.79	15.38	15.84	15.96	16.00
32	12.31	22.86	29.09	31.22	31.80	31.96
64	13.22	32.65	51.61	60.48	63.05	63.76
128	12.85	39.51	82.05	112.28	123.67	126.89

Tabela 2.2. Speedups obtidos para diferentes números de núcleos e tamanhos de problema.

Após o cálculo do *speedup* para cada valor p e n, podemos calcular a eficiência E, que definida como a razão entre o *speedup* S e o número de núcleos:

$$E = \frac{S}{p}.$$

A tabela abaixo representa o cálculo de eficiência para cada combinação possível de configuração.

# Núcleos	1x	2x	4x	8x	16x	32x
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.98	1.00	1.00	1.00	1.00	1.00
4	0.93	0.98	1.00	1.00	1.00	1.00
8	0.81	0.92	0.98	1.00	1.00	1.00
16	0.61	0.86	0.96	0.99	1.00	1.00
32	0.38	0.71	0.91	0.98	0.99	1.00
64	0.21	0.51	0.81	0.94	0.98	1.00
128	0.10	0.31	0.64	0.88	0.97	0.99

Tabela 2.3. Eficiências obtidas para diferentes números de núcleos e tamanhos de problema.

Após a obtenção da Tabela 2.3, podemos avaliar a escalabilidade do programa. Observa-se que, para tamanhos de problema menores, como $1x$, a eficiência apresenta uma queda acentuada à medida que o número de núcleos aumenta. Esse comportamento vai contra o conceito de escalabilidade forte.

No entanto, ao analisar a escalabilidade de forma linear, isto é, aumentando o tamanho do problema proporcionalmente ao número de núcleos, percebe-se que a eficiência

se mantém mais estável, próxima de 1. Isso indica que o programa consegue distribuir melhor a carga de trabalho em cenários com maior demanda de processamento, mostrando uma capacidade de escalabilidade sob essas condições.

Portanto, podemos concluir que o programa exibe escalabilidade fraca para problemas de menor dimensão, mas aproxima-se de uma escalabilidade forte quando o tamanho do problema cresce na mesma proporção que o número de núcleos.

A análise por tabelas ainda é viável com uma pequena contagem de núcleos e taxas crescentes quadraticamente, agora para análises mais detalhadas, o uso de tabelas começa a ser um problema, a Tabela 2.4 ilustra uma parte da Tabela 2.3 agora com taxas lineares.

# Núcleos	1x	2x	3x	4x	5x	6x	7x	8x	9x	10x
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
3	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
4	0.93	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
5	0.87	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
6	0.81	0.95	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
7	0.84	0.94	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
8	0.81	0.92	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00
9	0.84	0.91	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
10	0.75	0.91	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
11	0.72	0.91	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
12	0.72	0.96	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
13	0.75	0.95	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
14	0.63	0.95	0.97	0.99	1.00	1.00	1.00	1.00	1.00	1.00
15	0.63	0.94	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
16	0.61	0.86	0.96	0.99	1.00	1.00	1.00	1.00	1.00	1.00
17	0.57	0.85	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
18	0.57	0.94	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00
19	0.83	0.92	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
20	0.85	0.92	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
21	0.83	0.92	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
22	0.50	0.81	0.96	0.99	1.00	1.00	1.00	1.00	1.00	1.00
23	0.43	0.79	0.94	0.99	1.00	1.00	1.00	1.00	1.00	1.00
24	0.48	0.78	0.93	0.99	1.00	1.00	1.00	1.00	1.00	1.00
25	0.46	0.78	0.94	0.99	1.00	1.00	1.00	1.00	1.00	1.00
26	0.71	0.88	0.93	0.99	1.00	1.00	1.00	1.00	1.00	1.00
27	0.44	0.76	0.93	0.97	1.00	1.00	1.00	1.00	1.00	1.00
28	0.43	0.75	0.92	0.98	0.99	1.00	1.00	1.00	1.00	1.00
29	0.42	0.74	0.86	0.95	0.98	1.00	1.00	1.00	1.00	1.00
30	0.43	0.72	0.92	0.98	1.00	1.00	1.00	1.00	1.00	1.00
31	0.72	0.81	0.91	0.98	0.99	1.00	1.00	1.00	1.00	1.00
32	0.38	0.71	0.85	0.94	0.96	0.97	0.98	0.99	1.00	1.00

Tabela 2.4. Eficiências obtidas para diferentes números de núcleos e tamanhos de problema.

A medida que o detalhamento nas análises crescem, maiores ficam as tabelas, que vale ressaltar, não são geradas por nenhuma ferramenta de perfil de desempenho, essas ferramentas fornecem os dados utilizados para uma única célula dessa tabela. O desenvolvedor que se preocupa com a escalabilidade é deixado sozinho para rodar, coletar e de alguma forma visualizar os pontos críticos e gargalos da aplicação.

Embora tabelas sejam frequentemente utilizadas para essa tarefa, elas podem ocultar tendências importantes de escalabilidade. Além disso, gráficos de linha ou barras não são adequados para visualizar esse tipo de dado, já que muitas vezes disfarçam as tendên-

cias. No caso de perfis voltados para escalabilidade, seria necessário criar uma tabela para cada região de código de interesse, o que, manualmente, se torna ainda mais demorado e, com ferramentas de perfilamento, pode ser ainda mais trabalhoso.

2.2. Introdução ao PaScal Suite

O *Parallel Scalability Suite* (PaScal Suite) é um conjunto de ferramentas para avaliar as tendências de escalabilidade de programas paralelos. Ela simplifica a execução, medição e comparação de várias execuções de um programa paralelo, permitindo a análise de tendências de escalabilidade em ambientes de configuração com diferentes quantidades de elementos de processamento e diferentes cargas de trabalho, com elementos visuais que ajudam o usuário a entender o comportamento do programa e reconhecer gargalos de escalabilidade que podem exigir uma análise de otimização mais aprofundada. O conjunto de ferramentas pode ser utilizado para auxiliar o desenvolvimento de programas paralelos executados em um único nó computacional de memória compartilhada.

2.2.1. PaScal Analyzer

O *Parallel Scalability Analyzer* [4], ou **PaScal Analyzer**, é uma ferramenta projetada para perfilar aplicações, permitindo a medição e a comparação de execuções com diferentes configurações alternativas. Seu objetivo é facilitar o entendimento da capacidade de escalabilidade de uma aplicação paralela, observando como ela utiliza os recursos computacionais disponíveis. Além disso, a ferramenta se destaca por seu baixo nível de intrusão nos programas analisados, um aspecto fundamental para compreender com precisão o comportamento e a capacidade de escalabilidade da aplicação. Após sua execução, o *Analyzer* organiza os dados coletados para uma futura análise visual. Esses dados podem incluir tempo de execução, potência e outros medidores de desempenho; contudo, neste trabalho, vamos nos concentrar em tempo de execução.

No *Analyzer*, a instrumentação, seja manual ou automática, permite que os desenvolvedores observem, de forma incremental, como diferentes partes da aplicação paralela utilizam os recursos computacionais, medindo a eficiência e o desempenho conforme diferentes configurações são aplicadas. Essa flexibilidade é útil para localizar gargalos ou otimizar o uso dos recursos disponíveis, sem adicionar sobrecarga excessiva ao código em análise.

A instrumentação automática permite que o desenvolvedor execute a ferramenta sem a necessidade de modificar o código manualmente, facilitando o perfilamento. Já a instrumentação manual envolve a adição explícita de chamadas para marcar regiões de interesse no código-fonte, controlando exatamente quais partes da execução devem ser monitoradas.

O Algoritmo 2.1 ilustra a instrumentação manual com o *Analyzer*. Para isso, é necessário incluir a biblioteca `pascalops.h` no cabeçalho do seu código em C ou C++ e delimitar a zona perfilada com as funções `pascal_start(id)` e `pascal_stop(id)`, onde `id` é um número inteiro utilizado para identificar a região medida.

A biblioteca `pascalops` fica visível para o *GCC* após a instalação do *Analayzer* e após delimitar as zonas a serem medidas, basta compilar o código com o argumento

```
#include "pascalops.h"
...
int main(){
    pascal_start(1);
    #pragma omp parallel
    { ... }
    pascal_stop(1);
    ...
}
```

Listing 2.1. Exemplo de uso do PaScal Analyzer com OpenMP

`-lmpascalops`:

```
g++ main.cpp -fopenmp -lmpascalops
```

Para realizar a instrumentação automática, basta executar o *pascalanalyzer* com o argumento opcional `-t aut`:

```
    pascalanalyzer -t aut -c 8,4,2,1 -i 100,200,400,800 -o out.
       json <binario_executavel>
```

No exemplo acima, além de utilizarmos o argumento `-t` para escolher o tipo de instrumentação, foram empregados outros argumentos como `-c`, para selecionar a quantidade de núcleos, e `-i`, para fornecer os inputs que serão usados no programa. O argumento `-o` é utilizado para especificar o nome do arquivo de saída, sendo que, caso não seja fornecido, o *Analyzer* salvará a saída com um nome padronizado.

2.2.2. PaScal Viewer

O *Parallel Scalability Viewer*[5], ou **PaScal Viewer**, é uma aplicação web projetada para visualizar métricas de desempenho em programas paralelos em sistemas de memória compartilhada. Ele aceita como entrada o arquivo de saída do *Analyzer*, fornecendo uma maneira simplificada de analisar e otimizar o código para escalabilidade paralela, introduzindo uma abordagem inovadora por meio de diagramas de cores que lembram mapas de calor. Esses diagramas são um suporte visual para identificar tendências de eficiência paralela do programa inteiro ou de partes específicas de um código paralelo. A interface do Viewer também possibilita a análise de zonas paralelas em hierarquia, permitindo o perfilamento de um código paralelo. O Viewer se encontra disponível no domínio: `https://pascalsuite.imd.ufrn.br/`.

A Figura 2.2 apresenta uma captura de tela da interface do *Viewer*, a qual exibe quatro diagramas que auxiliam na análise de desempenho de programas paralelos. Cada um desses diagramas tem uma função específica para avaliar diferentes aspectos da execução da aplicação.

O primeiro diagrama, localizado no canto superior esquerdo, mede a eficiência

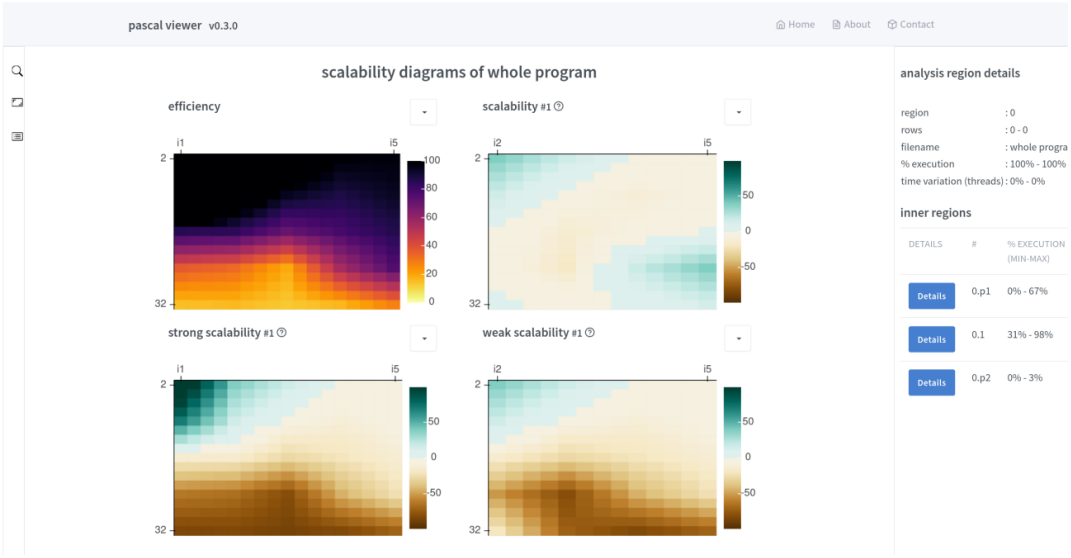


Figura 2.2. Captura de tela da interface da ferramenta web PaScal Viewer

global da aplicação paralela. Ele relaciona o tempo útil de execução ao tempo total gasto. Regiões com cores mais escuras indicam uma maior eficiência, enquanto áreas mais claras sugerem perda de desempenho.

O segundo diagrama, no canto superior direito, avalia a escalabilidade global da aplicação. Cores mais próximas do azul indicam trechos do código que são escaláveis, ou seja, que se beneficiam do aumento de recursos (como núcleos de processamento). Já os tons de marrom apontam ineficiência, sugerindo que a aplicação não está aproveitando de maneira adequada os recursos disponíveis.

Os dois diagramas no canto inferior da interface medem a escalabilidade forte (esquerda) e a escalabilidade fraca (direita). A escalabilidade forte avalia como o tempo de execução da aplicação é reduzido à medida que o número de núcleos aumenta, mantendo-se o mesmo tamanho de problema. Por outro lado, a escalabilidade fraca verifica se a aplicação consegue manter o mesmo tempo de execução ao se aumentar proporcionalmente tanto o número de núcleos quanto o tamanho do problema. Em ambos os diagramas, o esquema de cores segue o mesmo padrão do diagrama de escalabilidade global.

2.3. Estudo de Casos e Aplicabilidade

Agora, vamos avaliar a escalabilidade de um programa paralelo, o material utilizado nesta seção estará disponível no repositório público do PaScal Suite: `https://gitlab.com/lappsufrn/pascal-suite-tutorial/-/tree/minicurso2024`.

O nosso objeto de estudo principal é a análise de escalabilidade em um problema clássico de suavização de matriz utilizando o método de *Red-Black Gauss-Seidel* em um ambiente paralelo. Esse tipo de abordagem é comum em problemas de simulação numérica, como difusão de calor, simulação de fluidos e outros sistemas que utilizam equações diferenciais parciais (EDPs). O código apresentado a seguir é uma versão paralela do

algoritmo utilizando *OpenMP* para paralelização das operações sobre a matriz.

```
#include <omp.h>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <iostream>

#define MAX_ITER 1000

void initialize(double **A, int n) {
  // inicialize uma matriz de tamanho n x n
}

double matrix_calculation(double **A, int n) {
  double tmp;
  double diff = 0;
  int i, j;

#ifdef _OPENMP
#pragma omp parallel private(tmp, i, j)
  {
#pragma omp for reduction(+ : diff)
    for (i = 1; i <= n; ++i) {
      for (j = 1; j <= n; ++j) {
        if ((i + j) % 2 == 1) {
          tmp = A[i][j];
          A[i][j] = 0.2 * (A[i][j] + A[i][j - 1] + A[i - 1][j] +
              A[i][j + 1] +
                          A[i + 1][j]);
          diff += fabs(A[i][j] - tmp);
        }
      }
    }
#pragma omp single
    {
      for (i = 1; i <= n; ++i) {
        for (j = 1; j <= n; ++j) {
          if ((i + j) % 2 == 0) {
            tmp = A[i][j];
            A[i][j] = 0.2 * (A[i][j] + A[i][j - 1] + A[i - 1][j]
                + A[i][j + 1] +
                            A[i + 1][j]);
            diff += fabs(A[i][j] - tmp);
          }
        }
      }
    }
  }
```

```
#endif // _OPENMP
  return diff;
}

void solve_parallel(double **A, int n) {
  int iters;
  double diff = 0;
  for (iters = 1; iters < MAX_ITER; ++iters) {
    diff = matrix_calculation(A, n - 1);
  }
}

int main(int argc, char *argv[]) {
  int i;
  double **A;
  int n;

  try {
    n = std::stoi(argv[1]);
} catch (const std::invalid_argument& e) {
    std::cerr << "Invalid argument: " << argv[1] << " is not a
       valid integer." << std::endl;
    return 1;
}
  A = new double *[n + 2];
  for (i = 0; i < n + 2; i++) {
    A[i] = new double[n + 2];
  }

  initialize(A, n);

  solve_parallel(A, n - 1);
}
```

Para conduzir o estudo de caso, configuramos um ambiente com diferentes quantidades de núcleos de processamento, variando entre 1, 2, 4 e 8 núcleos, utilizando matrizes de diferentes dimensões (tamanho do problema) para observar o comportamento da escalabilidade forte e fraca da aplicação.

Primeiro compilamos o código:

```
    g++ -Wall -o RB -fopenmp RB.cpp
```

E com ele compilado executamos o *Analyzer*:

```
    pascalanalyzer ./RB -c 8,4,2,1 -i 1001,1501,2001,2501 -r 5 -
       t aut -o RB.json
```

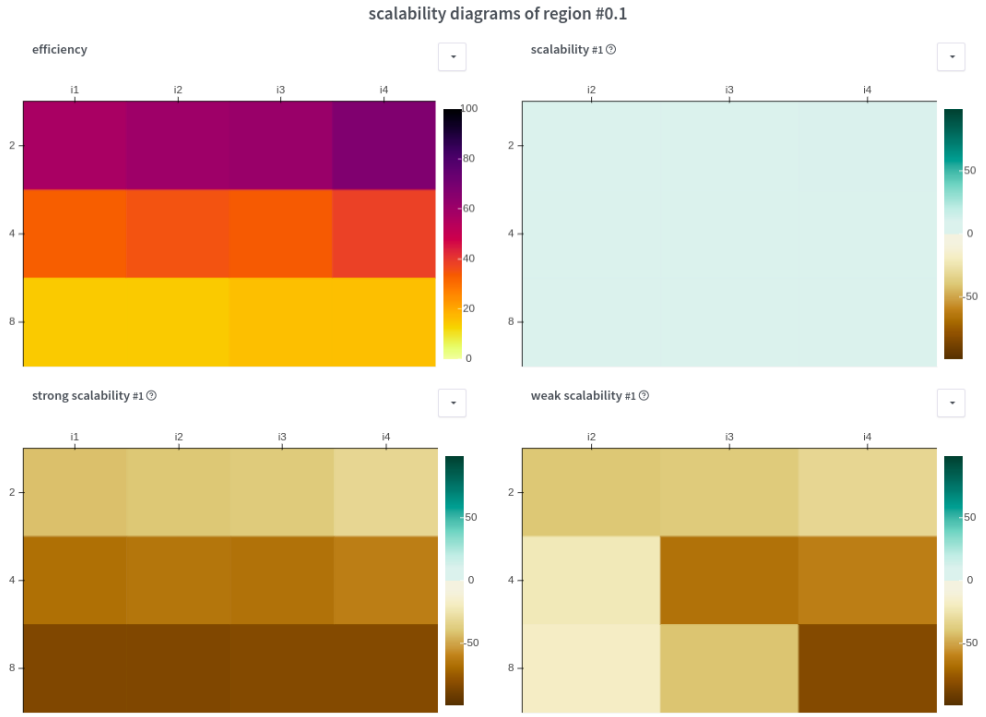


Figura 2.3. Captura de tela do PaScal Viewer com diagramas para nosso objeto de estudo na primeira leva de testes. A quantidade de núcleos varia em 2,4,8 no eixo Y enquanto o tamanho do problema varia em i1,i2,i3 e i4 no eixo X

Definimos o tamanho do problema crescendo quadraticamente similar a quantidade de núcleos, o argumento `-r` define o número de repetições para cada configuração, caso não seja definido, o valor padrão é 1.

A Figura 2.3 apresenta uma captura de tela do *Viewer* contendo o arquivo de saída gerado pelo *Analyzer*. O diagrama de eficiência global demonstra uma queda de eficiência à medida que o tamanho do problema cresce proporcionalmente ao número de recursos alocados. Além disso, observa-se também uma perda de eficiência ao se manter o tamanho do problema fixo e apenas aumentar a quantidade de recursos.

Adicionalmente, o gráfico de escalabilidade global revela uma baixa escalabilidade em todos os pontos analisados. Tanto o gráfico de escalabilidade forte quanto o de escalabilidade fraca indicam que o programa não atingiu os critérios de nenhuma das duas formas de escalabilidade.

Para obter uma visão mais precisa dos possíveis gargalos em nosso código, podemos utilizar a instrumentação manual para perfilar as regiões paralelas identificadas na primeira rodada de testes com a instrumentação automática. Convenientemente, o nosso objeto de estudo possui apenas uma região paralela. Como demonstrado na Seção 2.2.1,

para realizar a instrumentação manual, precisaremos incluir a biblioteca `pascalops` em nosso código e delimitar as regiões de interesse utilizando `pascal_start(id)` e `pascal_stop(id)`.

O Algoritmo 2.2 ilustra a instrumentação manual. A Região 1 delimita toda a zona paralela, a Região 2 o primeiro laço de repetição, e, por fim, a Região 3 delimita o último laço de repetição dentro da Região 1.

Com o código devidamente modificado, vamos compilá-lo inserindo o argumento `-lmpascalops` e repetir os testes com o Analyzer, dessa vez indicando a instrumentação manual na linha de comando:

```
1    g++ -Wall -o RB -fopenmp RB.cpp -lmpascalops
```

Listing 2.3. Compilar o código com o argumento necessario para instrumentação manual

```
1    pascalanalyzer ./RB -c 8,4,2,1 -i 1001,1501,2001,2501 -r 5 -
        t man -o RB.json
```

Listing 2.4. Executar o PaScal Analyzer com -t man"para indiicar instrumentação manual

No *Viewer* é possível acessar os diagramas de cor da Região 2 e 3 como Sub-Regiões da Região 1 (Figura 2.4. Além da identificação das regiões, a iterface do *Viewer* nos fornece quanto esse trecho representa em porcentagem do tempo de execução. Essas informações indicam que Sub-Região 3 representa até 76% do tempo de execução total de nosso programa.

A Figura 2.5 apresenta uma captura de tela do *Viewer* da Sub-Região 2. Embora o diagrama de eficiência global demonstre determinada queda de eficiência a medida que os núcleos e tamanho do problema escalam, é possível concluir que esse trecho é mais eficiente que a execução por inteira (veja a Figura 2.3. Além disso, observa-se também uma perda de eficiência ao se manter o tamanho do problema fixo e apenas aumentar a quantidade de recursos.

O gráfico de escalabilidade global também revela uma baixa escalabilidade em todos os pontos analisados dessa zona.

A Figura 2.6 apresenta uma captura de tela do *Viewer* da Sub-Região 3. Diferentemente da Sub-Região 2, esse trecho apresenta uma baixa eficiência em todos os pontos do diagrama de eficiência global, o que irá refletir nos outros diagramas. Apesar da ineficiência, esse trecho também representa boa parte da porcentagem da execução de nossa aplicação. Por isso, acreditamos que o gargalo se encontre nessa zona.

Avaliando o trecho de código delimitado por `pascal_start(3)` e `pascal_stop(3)`, identificamos que o laço aninhado responsável por calcular o valor de `diff` nos índices pares de nossa matriz, está em uma diretiva `omp single` que indica que, aquele trecho deverá ser executado por somente uma thread. É possível que, se subtrairmos essa diretiva desnecessária desse trecho e adicionarmos uma diretiva `omp for` com a cláusula `reduction(+ : diff)`, vamos conseguir otimizar essa operação de redução e garantir que o laço alvo seja paralelizado.

Aplicando as modificações pensadas, a região 3 ficaria da seguinte forma:

```
pascal_start(3);
#pragma omp for reduction (+ : diff)
      for (i = 1; i <= n; ++i) {
        for (j = 1; j <= n; ++j) {
          if ((i + j) % 2 == 0) {
            tmp = A[i][j];
            A[i][j] = 0.2 * (A[i][j] + A[i][j - 1] + A[i - 1][j]
                + A[i][j + 1] +
                              A[i + 1][j]);
            diff += fabs(A[i][j] - tmp);
          }
        }
      }
    pascal_stop(3);
  }
```

Após isso, recompilamos o código e refizemos mais uma leva de testes com o *Analyzer* agora para identificar os possíveis ganhos de nossa solução.

A 2.7 mostra que após nossa implementação, a Região 1 apresentou ganhos de eficiência em todos os pontos, embora ainda apresente baixa escalabilidade, abrindo portas para uma futura investigação mais profunda.

A 2.8 mostra Sub-Região 3 após a identificação de um possível gargalo e a implementação de nossa solução. É possível observar que, tivemos um ganho eficiência em todos os pontos de nosso diagrama de eficiência global. Os valores também melhoram para os gráficos de escalabilidade forte e fraca, embora nosso código ainda não tenha atingido um nível de escalabilidade desejável.

2.4. Conclusão

Neste capitulo, discutimos a diferença entre desempenho e escalabilidade no contexto da computação paralela e foi apresentada o PaScal Suite, um conjunto de ferramentas que auxiliam na automatização de testes e na visualização de tendências de escalabilidade paralela.

Foi feito um estudo de caso com o PaScal Suite, onde identificamos um gargalo de desempenho em um código de teste. Com as otimizações, conseguimos melhorar a eficiência, mas o programa ainda não apresenta escalabilidade ideal. O próximo passo seria uma análise mais profunda para entender melhor os motivos da baixa escalabilidade em outros pontos do código.

Referências

[1] Pacheco, P. S. (2011) *An introduction to parallel programming*. Morgan Kaufmann.

[2] Amdahl, G. M. (1967) *Validity of the single processor approach to achieving large scale computing capabilities*. In *Proceedings of the April 18-20, 1967, Spring Joint*

Computer Conference (pp. 483–485). Association for Computing Machinery.

[3] John L. Gustafson. 1988. Reevaluating Amdahl's law. Commun. ACM 31, 5 (May 1988), 532–533. https://doi.org/10.1145/42411.42415

[4] Silva, Vitor, Nóbrega-da-Silva, Anderson, Valderrama Sakuyama, C., Manneback, Pierre, and Xavier-de-Souza, Samuel (2022). "A Minimally Intrusive Approach for Automatic Assessment of Parallel Performance Scalability of Shared-Memory HPC Applications". *Electronics*, vol. 11, no. 5. DOI: 10.3390/electronics11050689.

[5] Nóbrega-da-Silva, Anderson, Cunha, Daniel, Silva, Vitor, Araújo Furtunato, Alex Fabiano, and Xavier-de-Souza, Samuel (2019). "PaScal Viewer: A Tool for the Visualization of Parallel Scalability Trends". In: *Handbook of Research on Emerging Developments and Applications of High Performance Computing*. pp. 250-264. ISBN: 978-981-13-6209-5. DOI: 10.1007/978-3-030-17872-7_15.

```
\\ outras bibliotecas
#include "pascalops.h"

double matrix_calculation(double **A, int n) {
double tmp;
double diff = 0;
int i, j;

  pascal_start(1);
#ifdef _OPENMP
#pragma omp parallel private(tmp, i, j)
  {
    pascal_start(2);

#pragma omp for reduction(+ : diff)
    for (i = 1; i <= n; ++i) {
      for (j = 1; j <= n; ++j) {
        if ((i + j) % 2 == 1) {
          tmp = A[i][j];
          A[i][j] = 0.2 * (A[i][j] + A[i][j - 1] + A[i - 1][j] +
              A[i][j + 1] +
                          A[i + 1][j]);
          diff += fabs(A[i][j] - tmp);
        }
      }
    }

    pascal_stop(2);
pascal_start(3);
#pragma omp single
    {
      for (i = 1; i <= n; ++i) {
        for (j = 1; j <= n; ++j) {
          if ((i + j) % 2 == 0) {
            tmp = A[i][j];
            A[i][j] = 0.2 * (A[i][j] + A[i][j - 1] + A[i - 1][j]
                + A[i][j + 1] +
                            A[i + 1][j]);
            diff += fabs(A[i][j] - tmp);
          }
        }
      }
    }

    pascal_stop(3);
  }
  pascal_stop(1);
#endif // _OPENMP

  return diff;
}
```

Listing 2.2. Instrumentação manual da região paralela 1 de nosso objeto de estudo.

Figura 2.4. Acesso a Sub-Regiões 2 e 3 após instrumentação manual

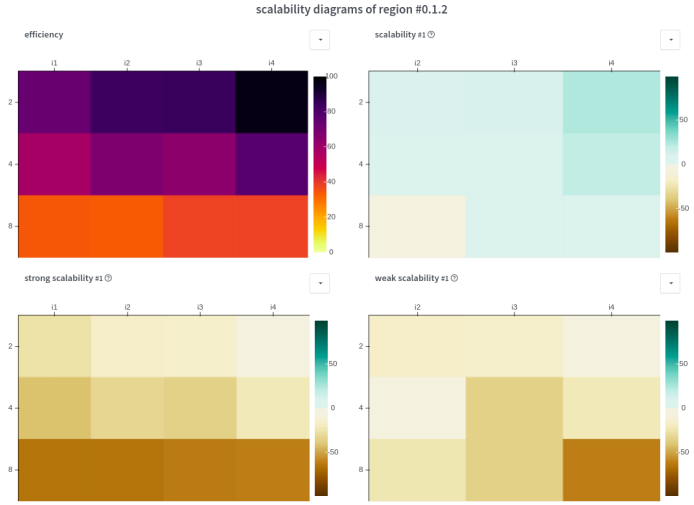


Figura 2.5. Captura de tela do PaScal Viewer com diagramas de eficiência e escalabilidade da Sub-Região 2 na segunda leva de testes. A quantidade de núcleos varia em 2,4,8 no eixo Y enquanto o tamanho do problema varia em i1,i2,i3 e i4 no eixo X

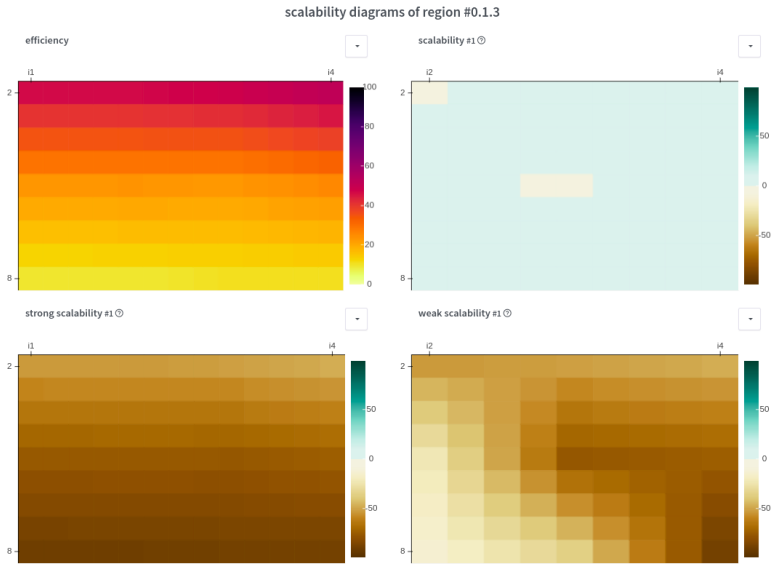


Figura 2.6. Captura de tela do PaScal Viewer com diagramas de eficiência e escalabilidade da Sub-Região 3 na segunda leva de testes. A quantidade de núcleos varia em 2,4,8 no eixo Y enquanto o tamanho do problema varia em i1,i2,i3 e i4 no eixo X

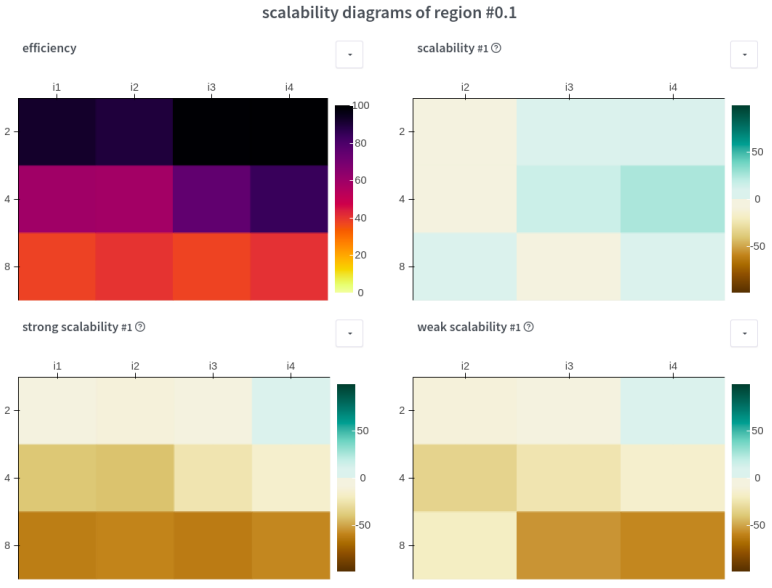


Figura 2.7. Captura de tela do PaScal Viewer com diagramas de eficiência e escalabilidade da Região 1 após nossa solução. A quantidade de núcleos varia em 2,4,8 no eixo Y enquanto o tamanho do problema varia em i1,i2,i3 e i4 no eixo X

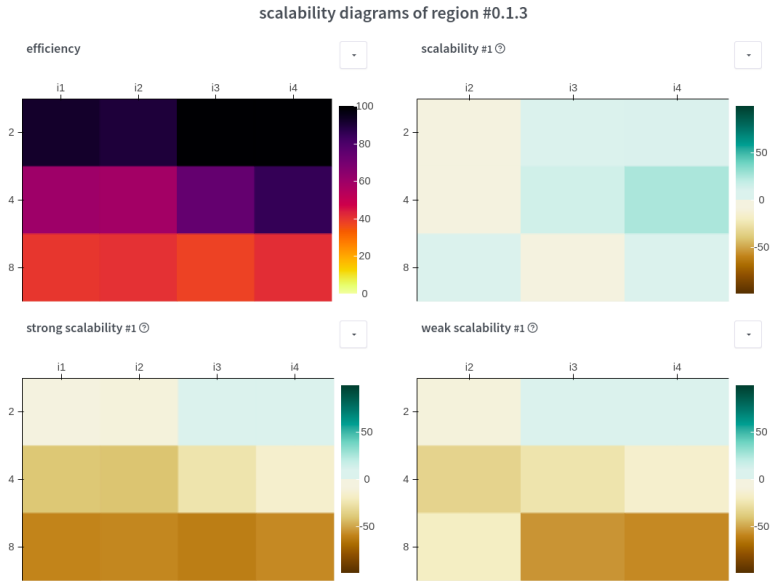


Figura 2.8. Captura de tela do PaScal Viewer com diagramas de eficiência e escalabilidade da Sub-Região 3 após nossa solução. A quantidade de núcleos varia em 2,4,8 no eixo Y enquanto o tamanho do problema varia em i1,i2,i3 e i4 no eixo X

Capítulo 3

Programação Paralela Híbrida: MPI + OpenMP Offloading

Calebe P. Bianchini, Evaldo B. Costa, Gabriel P. Silva

Abstract

This mini-course aims to introduce hybrid parallel programming techniques using MPI and OpenMP offloading directives, with a focus on parallelism models for accelerators. Necessary modifications to the source code to implement these models will be discussed with pratical examples.

Resumo

Este minicurso tem como objetivo apresentar técnicas de programação paralela híbridas utilizando MPI e diretivas de Offloading do OpenMP, com ênfase nos modelos de paralelismo em aceleradores. Serão abordadas as modificações necessárias no código-fonte para implementar esses modelos com uso de exemplos práticos.

3.1. Introdução

A programação paralela tem se tornado fundamental em diversas áreas, como ciência da computação, engenharia e biologia, devido à crescente complexidade dos problemas e à disponibilidade de *hardware* mais avançado, como processadores multinúcleo e aceleradores, incluindo GPUs [8]. Essas plataformas exigem o uso de técnicas de paralelismo que maximizem o desempenho e a escalabilidade.

Nos aceleradores gráficos de propósito geral (*GPGPU - General-Purpose Graphics Processing Unit*), o paralelismo é explorado por meio de multiprocessadores de fluxo (*streaming multiprocessors - SM*), que executam simultaneamente trechos computacionalmente intensivos chamados *kernels* [4]. Esse modelo de execução maciçamente paralelo é ideal para resolver problemas que exigem alta capacidade de processamento.

O MPI (*Message Passing Interface*) é a principal biblioteca de programação paralela utilizada em computadores de alto desempenho [14]. Ele é amplamente adotado em

sistemas de memória distribuída, permitindo a comunicação entre processos localizados em diferentes nós de um *cluster*, através da troca de mensagens. O MPI facilita a divisão de tarefas entre processadores que podem, por sua vez, gerenciar múltiplos aceleradores, criando uma hierarquia eficiente de processamento.

Por outro lado, o OpenMP, amplamente utilizado para paralelismo em sistemas de memória compartilhada, foi expandido para incluir diretivas de *offloading*. Essas diretivas permitem que trechos de código intensivos sejam transferidos para aceleradores, como GPUs, enquanto os processadores coordenam o fluxo geral de execução.

A maioria dos sistemas de alto desempenho (HPC) na lista Top500[1] são *clusters* de nós de memória compartilhada com aceleradores do tipo GPU. Para a utilização eficiente desses sistemas, há necessidade de se recorrer a vários modelos de programação: troca de mensagens, memória compartilhada e aceleradores. Portanto, a programação híbrida oferece a combinação da paralelização de memória distribuída na interconexão dos nós (com uso do MPI) com a paralelização de memória compartilhada (com uso do OpenMP) dentro de cada nó, além de possibilitar o uso eficiente de aceleradores (com as diretivas de *offloading* do OpenMP) [1].

Neste minicurso, exploraremos como combinar MPI, OpenMP e suas diretivas de *offloading* para otimizar a execução de códigos em sistemas heterogêneos. O MPI será utilizado para coordenar a comunicação entre nós distribuídos, enquanto o OpenMP com *offloading* permitirá que partes específicas do código sejam executadas em aceleradores. A seguir, apresentamos os principais tópicos abordados:

- **Arquitetura de computadores paralelos**: uma revisão sobre as arquiteturas paralelas modernas.
- **Arquitetura dos aceleradores**: com ênfase na estrutura interna dos aceleradores gráficos.
- **Modelos de programação paralela**: MPI puro, MPI combinado com OpenMP e, finalmente, MPI com OpenMP com diretivas *offloading*.
- Um estudo de caso baseado no cálculo de Pi.

Ao final do minicurso, espera-se que os participantes sejam capazes de:

- Entender os diversos paradigmas de exploração de paralelismo e as arquiteturas subjacentes;
- Compreender o funcionamento das aplicações paralelas híbridas através da utilização combinada de MPI e OpenMP;
- Avaliar as vantagens e desvantagens do uso das técnicas de paralelismo híbrido apresentadas.

[1] https://www.top500.org

Essa combinação de técnicas permitirá que os participantes avancem no desenvolvimento de soluções que aproveitem o melhor das arquiteturas modernas de processamento paralelo e distribuído.

3.2. Arquitetura de Computadores Paralelos

As arquiteturas paralelas são aquelas capazes de explorar o paralelismo existente nas aplicações e podem ser organizadas de diversas maneiras. Vamos apresentar, para melhor entendimento dos paradigmas de programação ilustrados neste cursos, algumas das formas de organizar essas arquiteturas paralelas [14].

Ao falarmos sobre paralelismo, devemos considerar os seguintes níveis de paralelismo possíveis na execução de uma aplicação:

- **No nível de instrução (granulosidade fina)**: encontrado em processadores com pipeline, superescalares ou VLIW (*Very Long Instruction Word*). Esse tipo de paralelismo é explorado de forma transparente ao usuário, normalmente pelo compilador ou diretamente pelo *hardware*, pela execução de diversas instruções em paralelo em diversas unidades funcionais existentes em um processador.

- **No nível de *thread* (granulosidade média)**: encontrado em diversos tipos de processadores e arquiteturas, destacando-se as *multithreading*, de *multithreading* simultâneo (SMT), *multicore* e aceleradores. Nesse caso, é necessário que tenhamos várias *threads* executando em paralelo, necessitando da intervenção do programador ou do compilador para o aproveitamento desse tipo de paralelismo.

- **No nível de processo (granulosidade grossa)**: encontrado em multiprocessadores (memória compartilhada) e multicomputadores (memória distribuída), e requerem também a intervenção explícita do programador e o suporte de bibliotecas e/ou ambientes de execução paralelos.

As arquiteturas que exploram o paralelismo no nível de instrução não irão receber nossa atenção, pois a exploração do paralelismo é feita diretamente pelo *hardware* ou compilador e não necessitam da intervenção do programador.

As **arquiteturas de memória compartilhada** têm como característica principal diversos processadores compartilhando um único espaço de endereçamento, permitindo assim a comunicação entre os diferentes fluxos de execução por meio de variáveis na memória compartilhada. O paralelismo no nível de processo em multiprocessadores com memória compartilhada, incluídos também o multiprocessamento simétrico (SMPs), não é usual. A comunicação via memória entre processos, que têm espaços de endereçamento distintos, requer esquemas mais sofisticados e lentos de comunicação via páginas compartilhadas pelo sistema operacional. Nesse caso é preferível a utilização de *threads* e,

dentro os diversos tipos de biblioteca disponíveis para exploração desse tipo de paralelismo, neste capítulo, vamos apresentar o uso do OpenMP.

Dentre as arquiteturas que exploram o paralelismo no nível de *thread*, temos dois tipos de arquiteturas que iremos estudar mais detalhadamente: os processadores *multicore* (SMP) e os aceleradores.

As **arquiteturas *multicore***, também conhecidas como *chip multiprocessing*, são caracterizadas pela existência de diversos processadores (núcleos) em um mesmo encapsulamento, compartilhando uma memória cache e a memória principal. Eventualmente esse tipo de arquitetura pode ser expandido para vários *chips*, cada um com vários núcleos, compartilhando uma mesma memória global, no que é chamado multiprocessamento simétrico (SMP). Notem que, em ambos os casos, a comunicação entre as *threads* é feita através de variáveis na memória compartilhada (ou na memória cache no caso dos *multicore*), necessitando de intervenção explícita do programador para a coordenação do paralelismo. Neste capítulo apresentaremos o OpenMP para a exploração de uma maneira mais fácil e eficientes esse tipo de paralelismo.

As **arquiteturas com aceleradores** são um tipo particular de exploração de paralelismo no nível de *thread*, onde trechos computacionalmente intensivos do programa, chamados de *kernels* são enviados para execução nos aceleradores, que tem memória distinta da memória do hospedeiro. Isso requer que tanto o código como os dados, sejam transferidos da memória do hospedeiro para a memória do acelerador, através de barramentos dedicados de alta velocidade. Os tipos de aceleradores mais utilizados nesse tipo de paralelismo são as GPUs (*Graphics Processing Unit*), cujos detalhes iremos apresentar mais adiante, junto com as diretivas de *offloading* do OpenMP.

Um outro modelo importante são as **arquiteturas com memória distribuída**, ou multicomputadores, onde cada processador possui seu próprio espaço de endereçamento, que não é compartilhado com os demais processadores. Dessa forma, a comunicação entre os diferentes programas em execução ocorre por meio de troca de mensagens, transmitidas por uma rede de comunicação que conecta esses processadores, caracterizando o que também chamamos de sistema distribuído. O paralelismo é explorado no nível dos processos, que se comunicam através de rotinas de troca de mensagens pela rede, como o *MPI* (Message Passing Interface), que também será abordado neste texto.

Um exemplo típico dessa categoria de computadores são os ***clusters***, que representam um tipo de sistema de processamento paralelo composto por uma coleção de computadores independentes, interconectados por meio de uma rede de comunicação de alto desempenho, trabalhando de forma cooperativa como um único recurso computacional integrado. Cada nó do *cluster* pode conter um ou mais processadores *multicore* com memória compartilhada, e, muitas vezes, um ou mais aceleradores, como as GPUs, que desempenham um papel importante, em termos de desempenho paralelo, nesse ambiente.

3.3. Aceleradores GP-GPUs

As arquiteturas dos aceleradores gráficos (GPUs) são bem diferenciadas das arquiteturas dos processadores convencionais. O paralelismo nos aceleradores gráficos é explorado através de um conjunto maciço de multiprocessadores de fluxo (*streaming multiproces-*

sors – SM), executando em paralelo e de forma sincronizada trechos computacionalmente intensivos, chamados de *kernels*, das diversas aplicações.

Para o melhor entendimento dos aceleradores gráficos (GPUs) vamos estudar, sem perda de generalidade, a arquitetura de um tipo de acelerador gráfico desenvolvido pela NVIDIA, a arquitetura Kepler [10].

Figura 3.1: Arquitetura NVIDIA Kepler

Na Figura 3.1 verificamos que o acelerador gráfico possui uma arquitetura distinta, com diversos níveis de hierarquia de memória, algumas delas compartilhadas, outras exclusivas de cada multiprocessador de fluxo (SM). Analisamos esses e outros detalhes a seguir.

Principais características da arquitetura Kepler

Cada unidade de multiprocessador de fluxo possui 192 núcleos CUDA de precisão simples e 64 de precisão dupla, onde cada núcleo tem unidades lógicas de aritmética inteira de ponto flutuante capazes de operar em modo "*pipeline*", incluindo operações do tipo *fused multiply-add* (FMA), conforme mostra a Figura 3.2. As 32 unidades de função especial (SFU) dentro de cada SM são utilizadas para calcular aproximações de operações transcendentais como raiz quadrada, seno, cosseno e recíproco $(1/x)$. O projeto dessa arquitetura está focado no desempenho/consumo energético, fundamental na computação de alto desempenho moderna.

O escalonador do multiprocessador de fluxo (SM) dispara as *threads* em grupos de 32 *threads* chamadas de *warps*. Cada SM possui quatro escalonadores de *warp*, permitindo um máximo de quatro *warps* disparadas e executadas concorrentemente. O número de registradores pode chegar até 255 registradores utilizados simultaneamente por cada *thread*.

Figura 3.2: Multiprocessador de Fluxo (SM)

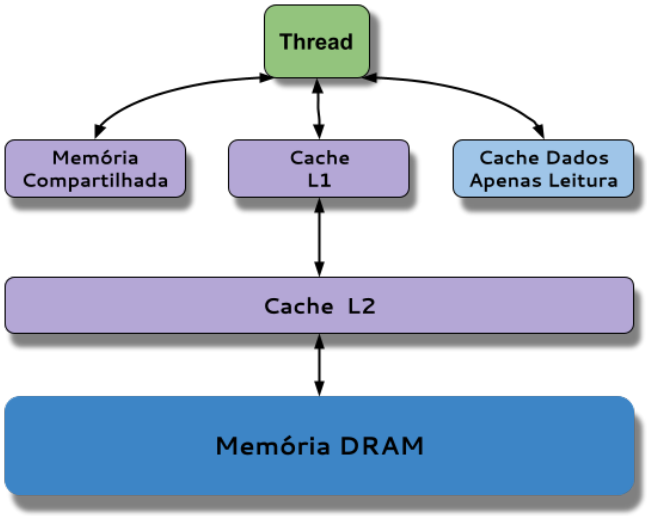
Para melhorar ainda mais o desempenho, a arquitetura Kepler apresenta uma instrução de *shuffle* que permite às *threads* dentro de uma mesma *warp* compartilhar dados. Anteriormente, o compartilhamento de dados entre *threads* demandava o acesso à memória compartilhada, com operações de *load* e *store*, impactando em muito o desempenho de aplicações como, por exemplo, a transformada de Fourier (FFT).

Outro tipo de instrução disponível são operações atômicas em memória, permitindo que as *threads* realizem adequadamente operações de *read-modify-write*, como soma, máximo, mínimo e compare-e-troque em estruturas de dados compartilhadas. Operações atômicas são amplamente utilizadas para ordenação em paralelo e para o acesso em paralelo a estruturas de dados compartilhadas sem a necessidade de travas que serializam a execução do código.

A arquitetura de memória do acelerador está organizada em diversos níveis, possuindo uma cache L1 para cada SM, além de uma cache apenas de leitura, como visto na Figura 3.3.

A quantidade de memória de cada SM é configurável. Por exemplo, a memória

Figura 3.3: Hierarquia de Memória



local (64 ou 128 KB) pode ser dividida nas seguintes proporções: 75% x 25%, 25% x 75% ou 50% x 50% entre uma memória compartilhada e uma cache L1.

Além da cache L1, a arquitetura Kepler introduz uma cache apenas de leitura de 48 KB. O gerenciamento dessa cache pode ser feito automaticamente pelo compilador ou explicitamente pelo programador. O acesso a uma variável ou estrutura de dados que o programador identifica como apenas de leitura, pode ser declarada com a palavra chave *const __restrict*, permitindo ao compilador carregá-la na cache apenas de leitura.

Essa arquitetura possui também um cache de nível 2 (L2) com 1,5 MB de capacidade. A cache L2 é o ponto primário de unificação de dados entre os diversos SMs, servindo operações de *load, store* e de textura, provendo um compartilhamento de dados eficiente e de alta velocidade.

Algoritmos onde o endereço dos dados é conhecido previamente, tais como solucionadores de física, *ray tracing* e multiplicação esparsa de matrizes, se beneficiam especialmente das hierarquia de cache. Os *kernels* de filtro e convolução onde é necessário que diversos SMs leiam os mesmos dados, também se beneficiam dessa hierarquia.

A arquitetura possui uma série de outras facilidades como código de correção de erro, paralelismo dinâmico, gerenciamento de filas de trabalho e unidade de gerenciamento de *grids*, que servem para melhorar o desempenho e a confiabilidade do acelerador. Maiores detalhes podem ser vistos na referência [10].

3.4. Modelos de programação

Ao explorar a combinação de MPI, OpenMP e suas diretivas de *offloading*, é importante conhecer o modelo de programação relacionado ao MPI, neste caso, utilizado para coordenar a comunicação entre nós de um *cluster*; e o modelo relacionado ao OpenMP, que

é utilizado para realizar *offloading* de partes específicas do código para serem executadas em aceleradores [12].

3.4.1. Paralelismo com MPI

O objetivo central do MPI é oferecer uma interface padrão para o desenvolvimento de programas baseados no paradigma de troca de mensagens. Para isso, é necessário que as funções definidas sejam portáteis, flexíveis e permitam implementações eficientes de acordo com as características do ambiente de execução escolhido. Outros objetivos gerais do MPI são [9][14]:

- Definir uma interface de programação de aplicações (API) — ao invés de apenas uma interface para uso pelos compiladores ou uma implementação de biblioteca de sistema.
- Permitir uma comunicação eficiente evitando cópias de memória para memória, com superposição de comunicação e computação, e possibilidade do uso de co-processadores de comunicação, quando disponíveis.
- Permitir o uso do padrão MPI em ambientes heterogêneos.
- Facilitar o uso da interface por linguagens como "C" e Fortran
- Assumir que a interface de comunicação é confiável: as falhas de comunicação devem ser tratadas pelo subsistema de comunicação da plataforma.
- Definir uma interface que possa ser implementada em diversas plataformas, sem mudanças significativas no sistema de comunicação subjacente ou *software* de sistema.
- Tornar a semântica da interface independente de linguagem.
- Permitir o uso seguro de *threads* (fluxos de execução independentes dentro de um mesmo processo).

O padrão MPI é uma forma eficiente de implementação de programas paralelos. Embora esteja voltado para máquinas de memória distribuída, permite a sua utilização também em máquinas com memória compartilhada, com desempenho equivalente.

Cálculo de Pi somente com MPI

O Exemplo 3.1 apresenta um exemplo do uso de MPI com múltiplos processos [13][14]. Essa solução apresentada uma implementação simples para o cálculo de Pi a partir da integração numérica usando o método do ponto médio (ou dos retângulos) conforme a Equação 1, que é derivada da equação trigonométrica apresentada na Equação 2.

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \simeq \frac{1}{N}\sum_{i=0}^{N-1} \frac{4}{1+x_i^2} \tag{1}$$

$$arctan(1) = \pi/4 \tag{2}$$

Exemplo 3.1: Cálculo do PI em diversos processos com MPI

```c
#include <stdio.h>
#include <mpi.h>
static long num_steps = 10000000000;
double step;
int main(int argc, char *argv[])
{
  long int i;
  int rank, size, provided;
  double x, pi, sum = 0.0, global_sum = 0.0;
  double start_time, run_time;
  // Inicia o MPI com suporte para threads
  MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
  if (provided < MPI_THREAD_FUNNELED) {
    printf("Nível de suporte para threads não é suficiente!\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
  }
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // O rank do processo
  MPI_Comm_size(MPI_COMM_WORLD, &size); // O número de processos
  step = 1.0 / (double)num_steps;
  start_time = MPI_Wtime(); // Tempo de início da execução
  for (i = rank; i < num_steps; i += size) { // Saltos de acordo com
   ↪  o número de processos
    x = (i + 0.5) * step;
    sum += 4.0 / (1.0 + x * x);
  }
  // MPI_Reduce para somar os resultados de todos os processos MPI
  MPI_Reduce(&sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, 0,
   ↪ MPI_COMM_WORLD);
  if (rank == 0) {
    pi = step * global_sum; // Só o processo 0 calcula o valor final
   ↪  de Pi
    run_time = MPI_Wtime() - start_time;
    printf("pi = %2.15f, %ld passos, computados em %lf segundos\n",
   ↪ pi, num_steps, run_time);
  }
MPI_Finalize(); // Finaliza o MPI
  return 0;
}
```

Cada processo é responsável por executar uma parte da aproximação da integral para encontrar o número Pi e, para isso, utiliza-se o número do ranque de cada processo para uma divisão balanceada das iterações. Ao final, todos os processos enviam suas somas parciais para o processo com ranque 0 que, por meio de uma operação de redução realizada pela função *MPI_Reduce*, calcula o valor final do Pi.

3.4.2. Paralelismo com OpenMP

O OpenMP foi projetado para a programação de computadores paralelos com memória compartilhada. A facilidade principal é a existência de um único espaço de endereçamento através de todo o sistema de memória, assim cada processador pode ler e escrever em todas as posições de memória. É possível a utilização do OpenMP em diversos tipos de arquitetura, quais sejam:

- Arquiteturas com memória compartilhada centralizada
- Arquiteturas com memória compartilhada distribuída
- Arquiteturas *manycore*
- Aceleradores

O paralelismo no OpenMP é obtido pela execução simultânea de diversas *threads* dentro das regiões paralelas. Haverá ganho real de desempenho se houver processadores disponíveis na arquitetura para efetivamente executar essas regiões em paralelo. Por exemplo, as diversas iterações de um laço podem ser compartilhadas entre as diversas *threads* e, se não houver dependências de dados entre as iterações do laço, poderão também ser executadas em paralelo.

Os laços são a principal fonte de paralelismo em muitas aplicações. Se as iterações de um laço são independentes (podem ser executadas em qualquer ordem), então podemos compartilhar as iterações entre *threads* diferentes. Por exemplo, se tivermos duas *threads* e o laço no trecho de código a seguir, as iterações 0-49 podem ser feitas em uma *thread* e as iterações 50-99 na outra.

```
for (i = 0; i<100; i++)
  a[i] = a[i] + b[i];
```

O OpenMP faz uso combinado de diretivas passadas para o compilador, assim como de funções definidas na sua biblioteca, para explorar o paralelismo no código em linguagem C, C++ ou Fortran. É possível também modificar o comportamento da execução de uma aplicação a partir de informações passadas pelas variáveis de ambiente de um sistema operacional.

Cálculo Pi somente com OpenMP

Mais uma vez, apresentamos a solução do cálculo do Pi a partir da integração numérica no Exemplo 3.2, derivado também das Equações 1 e 2. Nesta nova solução, a divisão das iterações, que são independentes, é feita pelo ambiente de execução do OpenMP para cada *thread* disponível e a soma realizada por cada uma delas é armazenada em uma variável privada *x*. Ao término do laço de repetição, há uma operação de redução dos valores de cada *thread* para a variável *sum*, conforme indica a diretiva *reduction*.

Exemplo 3.2: Cálculo do PI em diversas *threads* com OpenMP

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

int main() {
  static long num_steps = 10000000000;
  double step = 1.0 / (double) num_steps;
  double sum=0.0, pi = 0.0, begin, end;
  begin=omp_get_wtime();
  #pragma omp parallel shared(sum, step, num_steps) num_threads(8)
  {
   #pragma omp for reduction(+:sum)
    for (long int i = 0; i < num_steps; i++) {
      double x = (i + 0.5) * step;
      sum += 4.0 / (1.0 + x * x);
    }
  }
  pi = sum * step;
  end=omp_get_wtime();
  printf("Valor de Pi calculado: %2.15f. O tempo de execução foi %lf
   ↪  segundos\n", pi, end-begin);
  return 0;
}
```

3.5. Programação Híbrida MPI + OpenMP

A paralelização de um programa com MPI exige, muitas vezes, a replicação e transferência de dados entre os diversos processos participantes da computação [5]. Um dos benefícios de uma implementação híbrida MPI + OpenMP é que apenas uma cópia dos dados replicados é necessária cada nó do *cluster*, que normalmente executará apenas um processo. E, dentro deste processo, os dados podem ser compartilhados por várias *threads* sem nenhuma (ou substancialmente menos) replicação.

A programação híbrida se adapta perfeitamente às arquiteturas atuais baseadas em *clusters* de multiprocessadores e/ou processadores multinúcleo, pois induz menos comunicação entre diferentes nós e aumenta o desempenho de cada nó sem a necessidade de aumentar os requisitos de memória da aplicação.

Aplicações com dois níveis de paralelismo podem utilizar processos MPI para explorar paralelismo de maior granularidade, ocasionalmente trocando mensagens para sincronizar informações e/ou compartilhar trabalho. Já o uso de *threads* ou GPUs exploram bem paralelismo de granularidade média e pequena fazendo uso do espaço de endereçamento compartilhado ou capacidade de processamento dos aceleradores.

Aplicações com restrições ou requisitos que limitem o número de processos MPI utilizáveis (por exemplo, algoritmo de multiplicação de matrizes de Fox), podem se beneficiar do OpenMP para explorar os recursos computacionais restantes. Aplicações para as quais o balanceamento de carga é difícil de alcançar somente com processos MPI, podem se beneficiar do OpenMP para balancear o trabalho, atribuindo um número diferente de *threads* a cada processo MPI em função de sua carga [7].

A seguir listamos algumas das possíveis vantagens do uso de programação híbrida MPI + OpenMP:

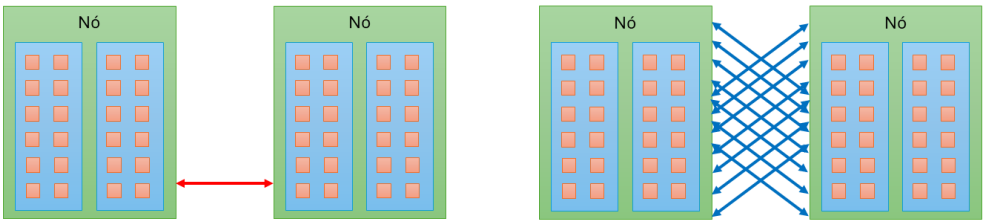
- **Redução do uso de memória**, já que haverá uma diminuição no número de cópias de estruturas de dados replicadas, além de uma menor quantidade de dados em regiões "halo", que são necessárias em algumas operações de comunicação entre processos.
- **Exploração de níveis adicionais de paralelismo**, pois é mais fácil adicionar paralelismo através de OpenMP em regiões críticas do código do que tentar aumentar o paralelismo em MPI puro, que exige mais esforços na divisão de tarefas e na comunicação entre processos.
- **Redução da quantidade de computação**, visto que alguns códigos MPI podem acabar replicando partes da computação, especialmente quando diferentes processos precisam realizar as mesmas operações em seus respectivos dados.
- **Redução do desequilíbrio de carga**, porque é mais fácil e econômico equilibrar a carga de trabalho entre *threads* OpenMP do que entre processos MPI, devido ao menor custo de comunicação e sincronização entre *threads*.
- **Redução dos custos de comunicação**, uma vez que os dados desnecessários não são transmitidos entre processos. Além disso, o número de processos (ranques) envolvidos em operações coletivas é menor, e há um número reduzido de mensagens ponto-a-ponto, embora possam ser maiores em tamanho.

Resumindo, os aplicativos que podem se beneficiar potencialmente da programação híbrida MPI + OpenMP são:

- Códigos com escalabilidade MPI limitada (devido, por exemplo, ao uso da primitiva de comunicação *MPI_Alltoall*);
- Códigos limitados pela capacidade de memória dos nós, mas com uma grande quantidade de dados replicados em cada processo MPI;
- Códigos cujo desempenho é prejudicado pela implementação ineficiente da comunicação dentro no mesmo nó do MPI;
- Códigos limitados pela escalabilidade de seus algoritmos (número de processos MPI).

É importante notar as formas de exploração de programação híbrida MPI + OpenMP, que vamos procurar esclarecer a seguir [2].

Na Figura 3.4, cada nó em um par de nós é ocupado por uma única tarefa MPI, identificada por seu ranque. São mostradas duas situações diferentes. À esquerda (Figura 3.4a), uma única *thread* em um dos ranques MPI troca mensagens com uma única *thread* no outro ranque — talvez até em circunstâncias em que múltiplas *threads* estejam



(a) Ranque por ranque (b) Ranque + *ThreadID* por Ranque + *ThreadID*

Figura 3.4: Comunicação de Programação Hibrida

sendo executadas. À direita (Figura 3.4b), algumas ou todas as *threads* em um ranque MPI podem trocar mensagens simultaneamente com *threads* parceiras em outro ranque MPI. As mensagens nesta segunda situação provavelmente precisarão ser identificadas tanto pelo ID da *thread* quanto pelo ranque.

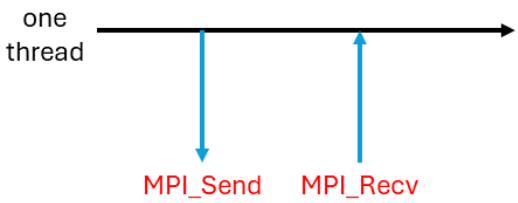
É importante manter essa imagem em mente ao explorar este tópico. Lembre-se de que apenas que na Figura 3.4b à direita as mensagens MPI estão sendo enviadas em paralelo entre múltiplas *threads* em dois (ou mais) ranques MPI.

- **Mensagens de *thread* única**: apenas uma *thread* em cada processo realiza a comunicação. Existem várias opções para este caso:
  - As chamadas MPI são feitas a partir de processos com *thread* única (na realidade não é híbrido; não exige que MPI tenha suporte para *threads*).
  - As chamadas MPI são feitas apenas da *thread* principal — seja em uma região serial, ou após mudar para a *thread* principal em uma região paralela.
  - As chamadas MPI são feitas a partir de múltiplas *threads* — mas de forma sincronizada, para que apenas uma *thread* faça chamadas MPI por vez.
- **Mensagens *multithread***: neste caso as chamadas MPI podem ser feitas a partir de múltiplas múltiplas *threads* em qualquer lugar dentro de uma região paralela; o MPI envia e recebe mensagens em paralelo. Essa opção requer uma implementação de MPI totalmente segura para *threads* (*thread-safe*).

Além dos cuidados essenciais com a programação, é necessária a chamada da rotina **MPI_Init_thread()** para determinar/selecionar o nível de suporte a *threads* do MPI. Ou seja, deve-se substituir a chamada usual para **MPI_Init()** por **MPI_Init_thread()**, que vai informar ao MPI o nível de suporte a *threads* que o programa requer. Ao retornar, a rotina **MPI_Init_thread()** inclui informações sobre o nível real de suporte a *threads* que o MPI utilizará, podendo ser diferente do solicitado. Ou seja, se o suporte a *threads* for inadequado, o programa poderá precisar tomar as medidas apropriadas. O suporte a threads é identificado/controlado pelos tipos fornecidos pelo MPI:

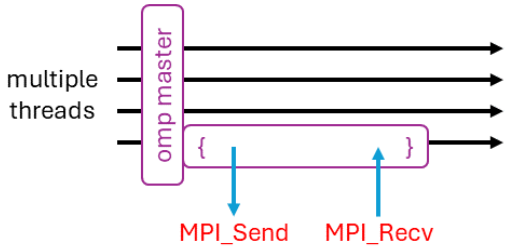
- **MPI_THREAD_SINGLE** (*single*): apenas uma *thread* existe em cada processo MPI — sem *multithreading* - veja a Figura 3.5. Em termos práticos, o código não pode conter diretivas OpenMP com regiões paralelas (*#pragma omp parallel*).

Figura 3.5: Modelo *Single*



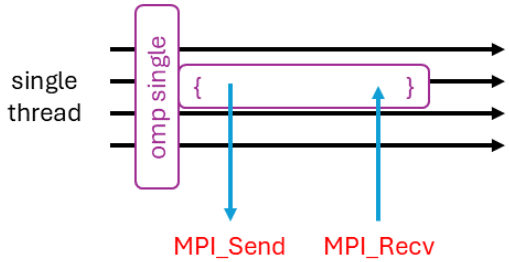
- **MPI_THREAD_FUNNELLED** (*funneled*): apenas a *thread* principal pode fazer chamadas MPI; em regiões *multithread*, todas as chamadas MPI devem ser canalizadas através da *thread* principal - veja a Figura 3.6. Ou seja, o programa pode conter diretivas OpenMP com regiões paralelas, mas apenas a *thread master* (*#pragma omp master*) pode realizar chamadas MPI. Como a construção **omp master** não cria barreiras implícitas, pode ser necessário o uso de barreiras explícitas (*#pragma omp barrier*) antes e depois da região *master*.

Figura 3.6: Modelo *Funneled*



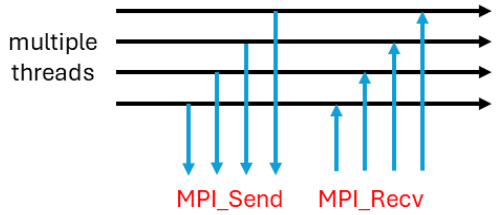
- **MPI_THREAD_SERIALIZED** (*seralized*): múltiplas *threads* podem chamar o MPI, mas de forma sincronizada, de modo que apenas uma chamada MPI esteja em progresso por vez - veja a Figura 3.7. Para evitar que chamadas MPI de uma *thread* sejam sobrepostas com chamadas de outras *threads*, as construções **omp critical** e **omp single** devem ser usadas. Uma barreira explícita (*#pragma omp barrier*) pode ser necessária antes da construção **omp single**, mas não depois, já que há uma barreira implícita no final da construção **omp single**.

Figura 3.7: Modelo *Serialized*



- **MPI_THREAD_MULTIPLE** (*multiple*): o MPI é seguro para *threads* (*thread-safe*), ou seja, funciona corretamente quando chamado por várias *threads* simultaneamente - veja a Figura 3.8. Em termos de implementação, o "thread id" de cada *thread*, obtido com a rotina *omp_get_thread_num()*, pode ser enviado como rótulo (*tag*) da mensagem, como forma de garantir que seja recebida pela *thread* correspondente no outro processo. Eventualmente, pode ser contraproducente permitir que a comunicação seja realizada por todos os núcleos do processador, e seja interessante limitar a comunicação a um número menor de núcleos, de modo a garantir melhor desempenho.

Figura 3.8: Modelo *Multiple Threads*



Apenas implementações de MPI com a capacidade MPI_THREAD_MULTIPLE poderão realizar o tipo totalmente *multithread* de comunicação conforme ilustrado na Figura 3.4b.

Se a rotina **MPI_Init()** for chamado em vez de **MPI_Init_thread()**, o nível de suporte a *threads* será automaticamente definido como MPI_THREAD_SINGLE. Para usar o **MPI_Init_thread()**, conforme descrito abaixo, *rqd*, ou "required" (*integer*) é o parâmetro de entrada, e que indica o nível desejado de suporte de *thread*; e *pvd*, ou "provided" (*integer*, passado por referência em 'C'), indica o nível de suporte de *thread* disponível para o programa. É importante observar que não há nenhuma garantia que *pvd* será maior ou igual a *rqd*.

```
int  MPI_Init_thread (int *argc, char ***argv, int rqd, int *pvd)
```

O código do Exemplo 3.3 mostra o nível mais alto de suporte de *thread* para uma dada implementação MPI. E, embora a maioria das implementações de MPI de hoje seja totalmente segura para *threads*, é importante fazer uma verificação de segurança logo após chamar *MPI_Init_thread()*. Um risco particular ocorre quando a implementação do MPI utilizada é construída a partir do código fonte.

Exemplo 3.3: Nível de suporte de *threads* em um programa MPI

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
  int* thread_support;
  thread_support = malloc(sizeof(int));
  MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, thread_support)
   ↪ ;

  if (*thread_support == MPI_THREAD_SINGLE) {
    printf("Executando com MPI_THREAD_SINGLE\n");
  }
  if (*thread_support == MPI_THREAD_FUNNELED) {
    printf("Executando com MPI_THREAD_FUNNELED\n");
  }
  if (*thread_support == MPI_THREAD_SERIALIZED) {
    printf("Executando com MPI_THREAD_SERIALIZED\n");
  }
  if (*thread_support == MPI_THREAD_MULTIPLE) {
    printf("Executando com MPI_THREAD_MULTIPLE\n");
  }

  MPI_Finalize();
}
```

O nível MPI_THREAD_FUNNELED deve ser suficiente para a maioria dos códigos com MPI e OpenMP, onde as chamadas MPI só ocorrem nas regiões seriais.

Para um código híbrido mais complexo que é projetado para enviar mensagens MPI entre várias *threads* individuais, o nível MPI_THREAD_MULTIPLE pode ser utilizado inicialmente. No entanto, se todas as chamadas *multithread* MPI foram serializadas colocando-as, por exemplo, em seções críticas OpenMP, pode haver uma vantagem usar apenas o nível MPI_THREAD_SERIALIZED. Isso notifica o MPI do nível de suporte que é realmente necessário, possibilitando à implementação MPI pular algumas das inicializações de *thread* e bloqueios internos que seriam necessários de outra forma, o que pode melhorar o desempenho do envio e recepção de mensagens.

Cálculo de Pi com MPI + OpenMP

No exemplo de cálculo de Pi usando uma abordagem híbrida com MPI + OpenMP, como mostra o Exemplo 3.4, cada processo MPI dividirá o trabalho entre as *threads* de uma região paralela (*#pragma omp parallel for*). Espera-se que cada *thread* esteja mapeada para um processador diferente dentro de cada nó do *cluster*, de modo que a execução paralela possa ter ganho real.

Embora diversas formas de mapeamento da execução das *threads* nos processadores sejam possíveis, não serão objeto de estudo neste minicurso. Neste exemplo, somente

a *thread master* de cada processo MPI realiza a operação de redução *MPI_Reduce*.

Exemplo 3.4: Cálculo de Pi usando MPI + OpenMP *threads*

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <math.h>
#include <mpi.h>

int main(int argc, char *argv[] ) {
  static long num_steps = 10000000000;
  double step = 1.0 / (double) num_steps;
  double sum=0.0, pi = 0.0, mypi = 0.0, begin, end;
  int rank, size;
  int* thr_status;

  MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, thr_status);
  if (*thr_status != MPI_THREAD_FUNNELED) {
    printf("Erro ao iniciar no modo MPI_THREAD_FUNNELED\n");
    exit(-1);
  }
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  begin = omp_get_wtime();
  sum = 0.0;
  #pragma omp parallel shared(sum, step, num_steps) num_threads(2)
  {
   #pragma omp for reduction(+:sum)
    for (long int i = rank; i <= num_steps; i += size){
      double x = (i + 0.5) * step;
      sum += 4.0 / (1.0 + x * x);
    }
  }
  mypi = step * sum;
  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  end = omp_get_wtime();
  if (rank == 0)
  printf("Valor de Pi calculado: %2.15f. O tempo de execução foi %lf
    ↪  segundos.\n", pi, end-begin);
  MPI_Finalize();
  return 0;
}
```

Considerações sobre o desempenho MPI + OpenMP

A maioria das aplicações híbridas é escrita (por simplicidade) no estilo *master-only*, onde todas as chamadas MPI são feitas fora das regiões paralelas do OpenMP. Nesse cenário, ocorrem os seguintes efeitos na execução dos programas:

- As *threads* OpenMP ficam inativas durante as comunicações MPI, resultando em subutilização dos recursos de processamento.

- A hierarquia de memória, especialmente a cache, pode sofrer aumento de falhas (*cache misses*) quando a *thread* principal (*master*) envia dados que foram lidos ou escritos por outras *threads*, o que degrada o desempenho geral.

A sincronização ponto a ponto implícita, feita via mensagens MPI, pode ser substituída por barreiras explícitas, que são mais custosas. Isso acontece porque uma sincronização mais flexível entre *threads* é difícil de ser realizada de maneira eficiente com OpenMP.

Em um código puramente MPI, as mensagens intra-nó (entre processos dentro de um mesmo nó) tendem a ser naturalmente sobrepostas às mensagens inter-nó (entre diferentes nós). No entanto, no modelo híbrido, é mais difícil sobrepor a comunicação entre *threads* dentro do nó com a comunicação MPI entre nós.

Além disso, o OpenMP pode sofrer com o problema de *false sharing*, que ocorre quando múltiplas *threads* modificam dados na mesma linha de cache, e com os efeitos de NUMA (*Non-Uniform Memory Access*), que resulta em tempos de acesso à memória diferentes dependendo da localização física da memória. Esses problemas são naturalmente evitados com MPI, uma vez que cada processo tem seu próprio espaço de memória.

Finalmente, ao aumentar o número de *threads* por processo MPI, mantendo o número total de núcleos fixo, as seguintes tendências de desempenho são observadas:

- O tempo total gasto nas rotinas MPI diminui, pois há menos processos MPI se comunicando. (+++)
- O desbalanceamento de carga entre os processos MPI também diminui, já que há mais *threads* para executar as tarefas. (+++)
- A quantidade de computação pode ser reduzida, já que menos processos MPI significa menos duplicação de trabalho entre eles. (+++))
- O tempo ocioso das *threads* aumenta, já que há menos núcleos disponíveis para distribuir o trabalho entre as *threads*. (- - -)
- O tempo de sincronização entre as *threads* aumenta, principalmente devido às barreiras de sincronização no OpenMP. Esse tempo perdido nas barreiras aumenta conforme aumenta o número de *threads*. (- - -)
- Paradoxalmente, o desequilíbrio de carga entre as *threads* pode aumentar, já que nem todas as *threads* terão necessariamente a mesma quantidade de trabalho. (- - -)
- O uso do sistema de memória pode se tornar menos eficiente, devido ao *false sharing* e aos efeitos de NUMA, que são mais pronunciados quando muitas *threads* acessam diferentes partes da memória de maneira concorrente. (- - -)

3.6. Programação em GPU com OpenMP *Offloading*

Inicialmente, o suporte no OpenMP para fazer a descarga (*offloading*) da computação para GPUs era experimental e dependia do compilador específico e/ou da biblioteca de

ambiente de execução utilizada. No entanto, o suporte para descarga de computação para GPUs no OpenMP se tornou mais difundido nos últimos anos, com muitos compiladores e bibliotecas de ambiente de execução adicionando suporte para essa funcionalidade [12].

Em 2018, o OpenMP *Architecture Review Board* (ARB) lançou a especificação OpenMP 5.0, que incluiu várias novas funcionalidades e melhorias para descarregar a computação para aceleradores, incluindo GPUs. A especificação OpenMP 5.0 inclui novas diretivas para gerenciar transferências de dados entre o hospedeiro e o acelerador, bem como novas construções para gerenciar dependências entre tarefas (*tasks*) em computações descarregadas para a GPU. A versão atual da especificação OpenMP é a 5.2, lançada em novembro de 2021 [11].

Atualmente, muitos compiladores e bibliotecas de ambiente de execução que são amplamente utilizados, como o GNU Compiler Collection (GCC), Intel oneAPI, NVIDIA HPC SDK e Clang, oferecem suporte para descarregar a computação para GPUs utilizando OpenMP. Além disso, muitos fornecedores de GPUs, incluindo NVIDIA, Intel e AMD, desenvolveram ferramentas e bibliotecas que integram o OpenMP para fornecer suporte otimizado para descarregar a computação para suas GPUs. Outras melhorias de *hardware* também aumentam a sua eficiência, como a introdução de memória de alta largura de banda (HBM) e memória DDR5 para *kernels* que dependem fortemente de memória.

Há uma grande variedade de aceleradores disponíveis no mercado, assim como várias versões de compiladores e ambientes de execução. Na Tabela 3.1 apresentamos os principais compiladores e suas versões [3][6].

Tabela 3.1: Compiladores com suporte para GPU Offload (em Maio 2020)

Compilador	**Comandos C/C++/Fortran**	**Fabricante**	**Acelerador**	**Última versão**
GCC	gcc, g++, gfortran	GNU	NVPTX, AMD GCN	14.2 (01-08-2024)
Clang	clang, clang++	LLVM	NVPTX, AMDGPU, X86_64, Arm e PowerPC	18.1.8 (18-Jun-2024)
XL	xlc, xlc++, xlf	IBM (CLANG)	Power + Nvidia CUDA	17.1.1 (01-08-2022)
ICC	icc, i++, ifort	Intel	Intel Integrated Graphics, NVIDIA, AMDCGN (Codeplay)	2024.2.1 (01-12-2023)
AOMP	clang, clang++	AMD	AMD GCN	11.5-0 (30-04-2020)
CCE	cc, CC, ftn	Cray (CLANG)	Nvidia CUDA, AMD GCN	18.0 (01-08-2024)

3.6.1. Modelo *host/device* do OpenMP

O modelo ***host/device*** no OpenMP é uma solução eficiente para sistemas heterogêneos, permitindo que programadores aproveitem o poder de dispositivos aceleradores, como GPUs, sem a complexidade de programação explícita para esses *hardwares* [12][4].

Através das diretivas OpenMP, é possível paralelizar e transferir partes do código para execução em dispositivos, proporcionando melhorias significativas em desempenho para aplicações que lidam com grandes volumes de dados e operações computacionais intensivas.

O modelo ***host/device*** do OpenMP assume que o hospedeiro é onde a *thread* inicial do programa inicia sua execução, sendo que um ou mais dispositivos estão conectados ao hospedeiro, mas a memória do hospedeiro e do dispositivo estão em espaços de endereçamento distintos.

- ***Host*** **(Hospedeiro — Processador)**: o ***host*** controla o fluxo principal do programa, coordenando a execução do programa, alocando memória, gerenciando a comunicação com dispositivos e executa partes do código que não são descarregadas para os dispositivos. No contexto do OpenMP, o hospedeiro geralmente inicia a computação paralela e decide quando as tarefas devem ser enviadas para os dispositivos aceleradores.

- ***Device*** **(Dispositivo — Acelerador/GPU)**: o ***device*** refere-se a um dispositivo de processamento especializado, como uma GPU ou outro tipo de acelerador, que possui uma grande quantidade de unidades de processamento paralelas, otimizadas para realizar operações em grande escala. No OpenMP, as regiões de código que são paralelizadas para o dispositivo são transferidas para ele para execução, enquanto o hospedeiro aguarda os resultados ou continua com outras tarefas. As GPUs são um exemplo comum de dispositivo, usadas para realizar cálculos maciços em paralelo, com eficiência superior para certos tipos de operações, como processamento de grandes matrizes.

O modelo ***host/device*** no OpenMP permite que o código seja dividido entre o processador central, responsável pela maior parte do controle e gerenciamento, e dispositivos aceleradores, que realizam tarefas computacionalmente intensivas de forma eficiente.

3.6.2. Funcionamento do modelo *host/device* no OpenMP

O OpenMP introduziu suporte para dispositivos externos (como GPUs) a partir da versão 4.0, com a inclusão de diretivas que permitem o ***offloading*** (transferência da computação) de partes do código para esses dispositivos. No modelo *host/device*, a computação ocorre da seguinte forma:

1. **Identificação de Regiões para *Offloading***: o programador marca, através de diretivas, as regiões de código que serão executadas no dispositivo. A principal diretiva utilizada é o '*target*'. Exemplo básico de *offloading*:

```
#pragma omp target
{
  // Codigo que sera executado no dispositivo (GPU)
}
```

O código entre '{ }' será enviado para execução no dispositivo.

2. **Transferência de Dados**: o hospedeiro deve garantir que os dados necessários para o cálculo no dispositivo sejam copiados para a memória do acelerador antes da execução. Isso é feito com diretivas como '*map*', que especifica quais variáveis devem ser copiadas para o dispositivo.

```
#pragma omp target map(to: a, b) map(from: result)
{
  result = a + b;
}
```

Neste exemplo, '*a*' e ' ' são transferidos para o dispositivo e '*result*' é copiado de volta para o hospedeiro após a execução.

3. **Execução no *Device***: uma vez que os dados são transferidos, o dispositivo executa a região marcada para paralelização. O OpenMP utiliza as unidades de processamento massivamente paralelas do dispositivo para realizar as operações de forma eficiente.

4. **Sincronização e Transferência de Resultados**: após a conclusão da tarefa, os dados processados no dispositivo podem ser copiados de volta para o hospedeiro para serem usados no restante do programa. O controle retorna ao hospedeiro, que pode continuar com outras operações ou coordenar novas execuções no dispositivo.

O Exemplo 3.5 apresenta uma solução simples de utilização do modelo ***host/device*** no OpenMP para executar um laço de iterações independentes com operações simples sobre três vetores em uma GPU.

Exemplo 3.5: Exemplo de código OpenMP com offloading

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main(int argc, char *argv[])
5  {
6    int N = 1000;
7    int a[N], b[N], c[N];
8    // Iniciando os vetores no hospedeiro
9    for (int i = 0; i < N; i++) {
10     a[i] = i;
11     b[i] = i * 2;
12   }
13 // Região paralela que será executada no dispositivo (GPU)
```

```
#pragma omp target map(to : a, b) map(from : c)
  {
#pragma omp parallel for
    for (int i = 0; i < N; i++) {
      c[i] = a[i] + b[i];
    }
  }
  // Exibindo os resultados
  for (int i = 0; i < 10; i++) {
    printf("%d ", c[i]);
  }
  printf("\n");
  return 0;
}
```

Vantagens do modelo *host/device* no OpenMP

O OpenMP permite que programadores adicionem paralelismo e *offloading* com mínima intervenção, apenas inserindo diretivas no código. Os programas escritos com OpenMP são portáveis entre diferentes arquiteturas, pois o código é transferido automaticamente para o dispositivo apenas quando disponível. Em caso contrário, é executado pelo hospedeiro.

O modelo ***host/device*** permite que diferentes dispositivos de computação (CPUs, GPUs, FPGAs) trabalhem de forma coordenada, maximizando o desempenho. Entre os desafios desta programação está o fato de que o programador precisa gerenciar corretamente a transferência de dados entre o hospedeiro e o dispositivo, o que pode ser complicado em códigos mais complexos. Ainda, o tempo necessário para transferir dados entre o hospedeiro e o dispositivo pode impactar negativamente o desempenho se não for bem gerenciado.

3.6.3. Diretivas principais do modelo *host/device* no OpenMP

Além da diretiva **target**, as diretivas **teams** e **distribute** no OpenMP são usadas em conjunto no modelo de programação paralelo para dispositivos, como GPUs, no modelo ***host/device***. Elas permitem que o programador organize e distribua o trabalho de forma eficiente em ambientes com muitas unidades de processamento paralelas, como em GPUs.

- **#pragma omp target**: usada para especificar a região de código que será descarregada para o dispositivo.
- **#pragma omp target data**: é usada para especificar o mapeamento de dados entre o hospedeiro (*host*) e o dispositivo (por exemplo, uma GPU) através de múltiplas regiões de código *offload*, minimizando a movimentação de dados entre o hospedeiro e o dispositivo.
- **#pragma omp teams**: cria múltiplas equipes de *threads* que operam de forma independente no dispositivo.

- **#pragma omp distribute**: distribui as iterações de um laço entre essas equipes.
- **#pragma omp parallel for**: paraleliza a execução de laços dentro de cada equipe, permitindo que as *threads* dentro de cada equipe executem as iterações do laço de forma paralela, quando precedido de uma diretiva **target**.

Vamos entender o propósito e a aplicação de cada uma dessas diretivas.

3.6.3.1. Diretiva target – Cláusula map

A cláusula **map** especifica como os dados do hospedeiro (*host*) são transferidos para o dispositivo (GPU) e de volta, facilitando a manipulação de grandes volumes de dados em computações paralelas.

A principal função da cláusula **map** é gerenciar explicitamente a **movimentação de dados** entre o *hospedeiro* (*host*, normalmente um processador) e o *dispositivo* (*device*, normalmente uma GPU ou outro acelerador). No modelo de programação *host/device*, o processador (*host*) controla a execução do código e delega parte do processamento para o dispositivo (*device*). Entretanto, como o processador e o acelerador possuem espaços de memória separados, é necessário transferir os dados entre essas duas áreas. A cláusula **map** permite ao programador controlar como e quando essas transferências ocorrem, otimizando o desempenho da aplicação.

A cláusula **map** é usada com a diretiva **target**, que envia uma região de código para ser executada no dispositivo. A sintaxe básica da cláusula **map** é:

```
#pragma omp target map(tipo: variavel)
{
  // Codigo a ser executado no dispositivo
}
```

Nela,

- **tipo**: o tipo de mapeamento que será realizado (descritos abaixo).
- **variavel**: as variáveis que serão mapeadas entre o hospedeiro e o dispositivo.

O seguintes tipos de mapeamento são admitidos:

- **map(to: variavel)**: transfere os dados da variável do hospedeiro para o dispositivo. Isso significa que a variável será copiada para o dispositivo antes que o código dentro da região **target** seja executado. As alterações feitas no dispositivo não serão refletidas no hospedeiro.
- **map(from: variavel)**: transfere os dados do dispositivo para o hospedeiro após a execução do código no dispositivo. A variável no hospedeiro será atualizada com o valor presente no dispositivo quando a execução da região **target** for concluída.

- **map(tofrom: variavel)**: este tipo de mapeamento faz uma cópia dos dados do hospedeiro para o dispositivo no início da região **target** e, em seguida, faz o caminho inverso, ou seja, do dispositivo para o hospedeiro quando a execução do código no dispositivo é finalizada.
- **map(alloc: variavel)**: aloca memória no dispositivo para a variável, mas não transfere dados do hospedeiro. Esse tipo é útil quando a variável será iniciada ou usada apenas no dispositivo e não precisa ser copiada de ou para o hospedeiro.

O Exemplo 3.6 demonstra uma forma simples de uso da cláusula 'map' no OpenMP.

Exemplo 3.6: Exemplo de uso da cláusula 'map'

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main(int argc, char *argv[]) {
5    int N = 10;
6    int a[N], b[N], c[N];
7    // Iniciando os vetores no hospedeiro
8    for (int i = 0; i < N; i++)
9    {
10     a[i] = i;
11     b[i] = i * 2;
12   }
13 // Regiao paralela no dispositivo (GPU)
14 #pragma omp target map(to : a, b) map(from : c)
15   {
16     for (int i = 0; i < N; i++) {
17       c[i] = a[i] + b[i];
18     }
19   }
20   // Exibindo os resultados no hospedeiro
21   for (int i = 0; i < N; i++) {
22     printf("%d ", c[i]);
23   }
24   printf("\n");
25   return 0;
26 }
```

Neste exemplo, os vetores '*a*' e '*b*' recebem os valores iniciais no hospedeiro e são transferidos para o dispositivo, ou seja, copiados do hospedeiro para o dispositivo antes da execução do código com '*map(to: a, b)*'. O cálculo da soma '*c[i] = a[i] + b[i]*' é realizado no acelerador (GPU), e o vetor resultante '*c*' é transferido de volta para o hospedeiro após a execução da região **target** com '*map(from: c)*'.

Uso avançado da cláusula 'map'

A cláusula **map** pode ser usada de maneira mais flexível, dependendo da necessidade de otimização do desempenho, especialmente em casos onde queremos evitar transferências

de dados desnecessárias. No Exemplo 3.7, o vetor '*c*' é alocado no dispositivo, mas não é transferido de volta ao hospedeiro com uso de '*map(alloc: c)*'. Isso pode ser útil quando '*c*' for usado apenas no dispositivo.

Exemplo 3.7: Uso avançadao da cláusula 'map'

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main(int argc, char *argv[])
5  {
6    int N = 10;
7    int a[N], c[N];
8    // Iniciando o vetor 'a' no hospedeiro
9    for (int i = 0; i < N; i++) {
10     a[i] = i;
11   }
12 // Regiao paralela no dispositivo (GPU)
13 #pragma omp target map(to : a) map(alloc : c)
14   {
15     // Inicia o vetor 'c' no dispositivo
16     for (int i = 0; i < N; i++) {
17       c[i] = a[i] * 2;
18     }
19   }
20   // Exibindo os resultados no hospedeiro
21   for (int i = 0; i < N; i++) {
22     printf("%d ", c[i]); // Este array nao foi transferido de volta!
23   }
24   printf("\n");
25   return 0;
26 }
```

Mapeamento de Subregiões de Arrays

Você também pode mapear subregiões de vetores para reduzir o volume de dados transferidos, otimizando a comunicação entre o hospedeiro e o dispositivo.

```
#pragma omp target map(to: a[0:N/2]) map(from: c[0:N/2])
{
  for (int i = 0; i < N/2; i++)
    c[i] = a[i] * 2;
}
```

Nesse exemplo, apenas a primeira metade do vetor 'a' é mapeada para o dispositivo e a primeira metade do vetor 'c' é mapeada de volta para o hospedeiro, economizando tempo de transferência e memória.

Boas Práticas ao Usar 'map'

O uso eficiente da cláusula **map** pode otimizar significativamente o desempenho de aplicações paralelas que utilizam dispositivos aceleradores, como GPUs. Para isso deve-se observar o seguinte:

- **Reduzir a movimentação de dados**: evite transferir dados desnecessários entre o hospedeiro e o dispositivo. Transfira apenas o que será efetivamente utilizado no dispositivo.
- **Usar 'alloc' quando possível**: sempre que possível, use **map(alloc:)** para alocar memória no dispositivo sem copiar dados do hospedeiro, quando os dados são usados ou inicializados apenas no dispositivo.
- **Mapear apenas regiões necessárias**: para grandes vetores ou estruturas de dados, mapeie apenas as partes que realmente serão utilizadas no dispositivo, como mostrado no exemplo de subregiões de vetores.
- **Mapeamento implícito**: em alguns casos, o OpenMP pode inferir automaticamente as transferências de dados. No entanto, o uso explícito de **map** permite controle total sobre o comportamento das transferências de dados, e pode ser essencial para otimizações de desempenho.

3.6.3.2. Diretiva target data

A diretiva **#pragma omp target data** permite que você controle explicitamente quais dados são copiados para o dispositivo antes da execução da parte *offload* e quais são trazidos de volta para o hospedeiro depois que o cálculo no dispositivo termina. Sua sintaxe é:

```
#pragma omp target data map(tipo: variavel)
{
  // Região de código que usa variáveis mapeadas para o dispositivo
}
```

A cláusula **map** é utilizada aqui com a mesma sintaxe e parâmetros que na diretiva **target** – o *tipo* de mapeamento que será realizado; e as *variáveis* que serão mapeadas entre o hospedeiro e o dispositivo – para recordar, reveja a Seção 3.6.3.1. O Exemplo 3.8 demonstra um uso simplificado de como a diretiva **target data** funciona.

Exemplo 3.8: Diretiva 'target data'

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int main(int argc, char *argv[])
5 {
6   int N = 100;
```

```
7   double A[N], B[N];
8   // Iniciando as matrizes no hospedeiro
9   for (int i = 0; i < N; i++) {
10    A[i] = i * 1.0;
11    B[i] = 0.0;
12  }
13 // Copiar A para o dispositivo, e depois B de volta ao hospedeiro
14 #pragma omp target data map(to : A[0 : N]) map(from : B[0 : N])
15  {
16 // Executa no dispositivo
17 #pragma omp target teams distribute parallel for
18    for (int i = 0; i < N; i++) {
19      B[i] = A[i] * 2.0;
20    }
21  }
22  // Imprimindo o resultado no hospedeiro
23  for (int i = 0; i < N; i++) {
24    printf("B[%d] = %f\n", i, B[i]);
25  }
26  return 0;
27 }
```

O uso da diretiva **target data** tem as seguintes vantagens:

- **Controle de movimentação de dados**: permite que o desenvolvedor controle exatamente quando os dados são transferidos entre o hospedeiro e o dispositivo, minimizando sobrecarga desnecessária.
- **Persistência dos dados**: ao usar **target data**, você pode manter os dados no dispositivo por várias chamadas **target**, evitando cópias repetidas para o dispositivo entre regiões de código *offload*.
- **Eficiência**: para aplicações que fazem múltiplos *offloads* para o acelerador, **target data** é mais eficiente do que usar várias diretivas **target** individuais que repetem as mesmas transferências de dados.

3.6.3.3. Diretiva teams

A diretiva **teams** é usada para criar equipes de *threads* no dispositivo (como uma GPU). Essas equipes consistem em várias *threads* que podem trabalhar em paralelo, mas que não compartilham *threads* entre si (cada equipe tem suas próprias *threads*). São as seguintes as funções da diretiva **teams**:

- **Criação de equipes paralelas**: quando você usa **teams**, está criando várias equipes de *threads* independentes que podem executar uma tarefa em paralelo no dispositivo. Cada equipe funciona como uma entidade separada e não compartilha *threads* com outras equipes.

- **Execução em dispositivos**: a diretiva é especialmente útil para dispositivos de *hardware* que possuem várias unidades de processamento paralelas, como GPUs, onde as *threads* são agrupadas em blocos ou equipes para otimizar o desempenho.

```
#pragma omp target teams
{
  // Este codigo será executado por várias equipes no dispositivo
}
```

Neste exemplo, múltiplas equipes de *threads* são criadas no dispositivo, e cada uma delas pode executar o código de forma paralela. A diretiva **teams** é normalmente usada em conjunto com **distribute**, que será explicado a seguir, para dividir o trabalho entre as equipes. Dentro de cada equipe, pode-se usar outras diretivas OpenMP, como **parallel**, para paralelizar o trabalho entre as *threads* da equipe.

3.6.3.4. Diretiva distribute

A diretiva **distribute** é usada para distribuir o trabalho entre as diferentes equipes criadas pela diretiva **teams**. Enquanto **teams** cria as equipes, **distribute** divide as iterações de um laço entre essas equipes. A diretiva **distribute** realiza as seguintes funções:

- **Divisão do trabalho entre equipes**: a principal função de **distribute** é atribuir diferentes partes de um laço às diferentes equipes, garantindo que cada equipe trabalhe em uma porção distinta do problema.
- **Compatível com laços**: assim como a diretiva **for** no OpenMP é usada para paralelizar laços entre *threads*, **distribute** faz o mesmo, mas no nível de equipes. Ou seja, distribui o laço entre equipes, e cada equipe pode, por sua vez, paralelizar o trabalho entre de si.

```
#pragma omp target teams distribute
for (int i = 0; i < N; i++) {
  // Cada equipe trabalha em diferentes iterações do laço
}
```

Aqui, o laço '**for**' é dividido entre as equipes criadas por **teams**. Cada equipe será responsável por um subconjunto de iterações do laço.

Uso Combinado de 'teams' e 'distribute'

As diretivas **teams** e **distribute** são frequentemente usadas juntas para distribuir o trabalho de maneira eficiente entre as equipes de *threads* e garantir que todo o potencial de paralelismo do dispositivo seja explorado.

```
#pragma omp target teams distribute parallel for
for (int i = 0; i < N; i++) {
  // Cada equipe paraleliza o laço entre si
  // com suas próprias threads
}
```

O bloco de código dentro da diretiva **target** será enviado para execução no dispositivo (por exemplo, uma GPU). Várias equipes de *threads* são criadas no dispositivo com a diretiva **teams**. Cada equipe funcionará independentemente. O laço '**for**' é distribuído entre as equipes com a diretiva **distribute**. Cada equipe recebe um subconjunto das iterações do laço. Finalmente, dentro de cada equipe, o laço é executado em paralelo pelas *threads* da equipe com uso da diretiva **parallel for**.

Diferença entre 'distribute' e 'for'

- **distribute**: distribui o trabalho entre equipes de *threads* (um nível superior).
- **for**: distribui o trabalho entre *threads* dentro de uma equipe (um nível inferior).

O Exemplo 3.9 demonstra o uso mais detalhado das diretivas **teams** e **distribute** juntos.

Exemplo 3.9: Uso combinado de 'distribute' e 'for'

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main(int argc, char *argv[])
5  {
6    int N = 1000;
7    int a[N], b[N], c[N];
8    // Valores iniciais dos vetores no hospedeiro
9    for (int i = 0; i < N; i++) {
10     a[i] = i;
11     b[i] = i * 2;
12   }
13 // Região paralela no dispositivo (GPU)
14 #pragma omp target teams distribute parallel for
15   for (int i = 0; i < N; i++) {
16     c[i] = a[i] + b[i];
17   }
18   // Exibindo os primeiros 10 resultados
19   for (int i = 0; i < 10; i++) {
20     printf("%d ", c[i]);
21   }
22   printf("\n");
23   return 0;
24 }
```

Cálculo de Pi somente com OpenMP *offloading*

No exemplo de cálculo de Pi usando uma abordagem OpenMP + *offloading*, como mostra o Exemplo 3.10, o trabalho é enviado para o acelerador com a diretiva (*#pragma omp target teams distribute parallel for*).

Neste exemplo em particular, não exploramos os paralelismo adicional entre *threads* executando nos vários núcleos do processador, o que também é possível, mas não ofereceria ganhos quando comparado à execução do mesmo código na GPU, pela enorme diferença de desempenho entre esses tipos de arquiteturas.

Exemplo 3.10: Pi só com OpenMP *offloading*

```
1  #include <stdio.h>
2  #include <omp.h>
3  static long num_steps = 100000000;
4  double step;
5  int main(int argc, char *argv[])
6  {
7    long int i;
8    double x, pi, sum = 0.0;
9    double start_time, run_time;
10   step = 1.0 / (double)num_steps;
11   start_time = omp_get_wtime(); // Tempo de início da execução
12 // Offloading com OpenMP para o acelerador (GPU), com paralelismo
       ↪ dentro do processo
13 #pragma omp target teams distribute parallel for default(none)
       ↪ firstprivate(num_steps, step) reduction(+ : sum) map(tofrom :
       ↪ sum) private(x) device(1)
14   for (i = 0; i < num_steps; i++) {
15     x = (i + 0.5) * step;
16     sum += 4.0 / (1.0 + x * x);
17   }
18   pi = step * sum; // calcula o valor final de Pi
19   run_time = omp_get_wtime() - start_time;
20   printf("pi = %lf, %ld passos, computados em %lf segundos\n", pi,
       ↪ num_steps, run_time);
21   return 0;
22 }
```

3.7. Programação Híbrida com MPI + OpenMP *offloading* para GPU

O uso do MPI (*Message Passing Interface*) em combinação com OpenMP *offloading* é uma abordagem importante para a exploração adequada do paralelismo em *clusters* com múltiplos nós e aceleradores (como GPUs) integrados, principalmente em contextos de computação científica e simulação. Essa combinação permite explorar o paralelismo em diferentes níveis — tanto entre nós quanto dentro de cada nó.

O MPI é uma ferramenta importante para a comunicação entre nós em um *cluster*, sendo que cada nó pode ter um ou mais processadores e aceleradores (GPUs). Com a integração com o OpenMP *offloading*, parte do código pode ser descarregado para GPUs

(ou outros aceleradores) localizados em cada nó, aproveitando o alto desempenho computacional da GPU para tarefas intensivas de processamento.

A comunicação entre nós é baseada em troca de mensagens através de MPI, utilizando um dos modelos apresentados anteriormente, ou seja, apenas uma das *threads* em cada processo é responsável pela comunicação ou múltiplas *threads* em cada processo podem se comunicar. O modelo divide o trabalho entre os nós, e esses processos podem se comunicar via redes de alta velocidade, como Infiniband.

Dentro de cada nó, o OpenMP pode ser utilizado para paralelismo *multithread* entre os vários núcleos ou para *offloading* para aceleradores. Cada processo MPI pode criar *threads* OpenMP para trabalhar simultaneamente ou descarregar partes do trabalho para GPUs.

Uma das maiores preocupações é o balanceamento de carga entre os nós e entre as *threads* dentro de um nó. A combinação MPI + OpenMP requer que o trabalho seja adequadamente distribuído entre os nós (via MPI) e entre as *threads* de processador ou GPUs (via OpenMP *offloading*). Um desequilíbrio pode levar a subutilização de recursos.

A troca de mensagens entre nós pode ser um gargalo, especialmente em aplicações com alta frequência de comunicação de modo que a redução da comunicação entre nós ou a minimização do volume de dados trocados é importante para melhorar a escalabilidade. Adicionalmente, a descarga do trabalho para aceleradores exige a transferência de dados entre o hospedeiro e o dispositivo. Se o código não for otimizado, a latência associada à transferência de dados entre a memória do hospedeiro e a memória do dispositivo pode também prejudicar o desempenho da aplicação.

Uma das maiores dificuldades é a identificação de gargalos e a otimização das aplicações de modo a realizar o melhor aproveitamento dos recursos disponíveis. Esse é um desafio que só pode ser vencido com um conhecimento adequado das estruturas de *hardware* e também dos diversos modelos de programação utilizados. Eventualmente o uso de ferramentas de perfilagem e monitoração dos recursos são importantes nesta tarefa de identificação dos gargalos e melhoria do desempenho, quer em termos de ganho, como de escalabilidade.

Embora a combinação MPI e OpenMP ofereça desempenho significativo, ela também aumenta a complexidade da implementação. Os desenvolvedores precisam ter conhecimento profundo tanto de comunicação distribuída (MPI) quanto de paralelismo compartilhado e *offloading* (OpenMP).

Uma escolha adequada das implementações, tanto do MPI como do OpenMP e do *offloading* é outra questão importante. Embora tanto o MPI como o OpenMP sejam amplamente suportados, o suporte a OpenMP *offloading* ainda depende de combinações específicas entre compilador e do acelerador utilizados. Verificar a compatibilidade e testar em diferentes arquiteturas de aceleradores pode ser necessário para garantir a robustez do código.

O uso de MPI com OpenMP *offloading* para *clusters* com aceleradores pode fornecer ganhos maciços de desempenho, mas requer atenção a vários fatores, como balanceamento de carga, escalabilidade, gerenciamento de memória e otimização de comunicação.

Uma estratégia hibrida eficaz permite aproveitar tanto o paralelismo distribuído quanto o paralelismo compartilhado, oferecendo uma solução escalável para problemas computacionais de grande porte.

Cálculo de Pi com MPI + OpenMP com *offloading*

No exemplo de cálculo de Pi utilizando uma abordagem híbrida de MPI + OpenMP com *offloading*, como mostrado no Exemplo 3.11, as iterações são distribuídas entre os nós de um *cluster*, onde cada nó executa um processo MPI, com base no ranque de cada processo. Dentro de cada nó, as iterações que lhe cabem são enviadas para o acelerador utilizando a diretiva *#pragma omp target teams distribute parallel for*.

Exemplo 3.11: Pi com MPI + OpenMP com offloading

```
#include <stdio.h>
#include <mpi.h>
#include <omp.h>
static long num_steps = 10000000000;
double step;
int main(int argc, char *argv[])
{
  long int i;
  int rank, size, provided;
  double x, pi, sum = 0.0, global_sum = 0.0;
  double start_time, run_time;
  // Inicia o MPI com suporte para threads
  MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
  if (provided < MPI_THREAD_FUNNELED) {
    printf("Nível de suporte para threads não é suficiente!\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
  }
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // O rank do processo
  MPI_Comm_size(MPI_COMM_WORLD, &size); // O número de processos
  step = 1.0 / (double)num_steps;
  start_time = omp_get_wtime(); // Tempo de início da execução
// Offloading com OpenMP para o acelerador (GPU), com paralelismo
   ↪ dentro do processo
#pragma omp target data map(tofrom:sum) map(to:size, num_steps, rank
   ↪ , step) device(1)
#pragma omp target teams distribute parallel for reduction(+:sum)
  for (i = rank; i < num_steps; i += size) { // Saltos de acordo com
   ↪  o número de processos
    x = (i + 0.5) * step;
    sum += 4.0 / (1.0 + x * x);
  }
  // MPI_Reduce para somar os resultados de todos os processos MPI
  MPI_Reduce(&sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, 0,
   ↪ MPI_COMM_WORLD);
  if (rank == 0) {
    pi = step * global_sum; // Só o processo 0 calcula o valor final
   ↪  de Pi
    run_time = omp_get_wtime() - start_time;
```

```
34      printf("pi = %3.15f, %ld passos, computados em %lf segundos\n",
        ↪ pi, num_steps, run_time);
35    }
36  MPI_Finalize(); // Finaliza o MPI
37    return 0;
38  }
```

O ganho de desempenho da aplicação advém do fato de haver múltiplos aceleradores no *cluster*, permitindo a divisão do trabalho entre as diferentes GPUs. Em outras palavras, o aumento na eficiência não resulta da adição de mais nós à computação, mas da capacidade de distribuir o trabalho de maneira eficaz entre os diversos aceleradores do *cluster*.

Neste exemplo, também não exploramos o paralelismo adicional através da execução de múltiplas *threads* nos diversos núcleos de processadores em um mesmo nó, algo que também é possível. No entanto, o ganho desse tipo de paralelismo também seria marginal em comparação com a execução na GPU, dada a diferença substancial de desempenho entre essas arquiteturas, conforme comentado anteriormente.

3.8. Conclusão

Apresentamos os principais conceitos sobre programação paralela híbrida utilizando MPI e OpenMP com *offloading*, visando otimizar a execução de códigos em sistemas heterogêneos. O uso dessas técnicas se justifica pela crescente complexidade dos problemas computacionais e pela disponibilidade de *hardware* avançado, que demandam estratégias eficientes de paralelismo.

Discutimos os principais tipos de arquiteturas e os modelos básicos de programação com MPI e OpenMP, explicando também a integração dessas APIs para criar aplicações paralelas eficientes em sistemas heterogêneos. A combinação de MPI com OpenMP oferece vantagens como a redução do uso de memória, exploração de níveis adicionais de paralelismo, diminuição da quantidade de computação e comunicação, além de melhorar o balanceamento de carga. Abordamos os diferentes níveis de suporte a *threads* no MPI (*single*, *funneled*, *serialized*, *multiple*), discutindo as implicações de cada nível na programação híbrida.

As principais diretivas de descarga de trabalho para aceleradores do OpenMP, segundo o modelo *host/device*, foram apresentadas, considerando a hierarquia de memória e de *hardware* dos aceleradores, assim como os cuidados para uma eficiente movimentação de dados entre o hospedeiro e o acelerador.

Por último, oferecemos um exemplo prático de programação híbrida MPI + OpenMP com *offloading*, demonstrando maneiras simplificadas e eficientes de dividir o trabalho entre os diversos processadores e aceleradores de um *cluster*.

Todos os exemplos deste minicurso, juntamente com as orientações para sua compilação, estão disponíveis no seguinte repositório `https://github.com/Programacao-Paralela-e-Distribuida/SSCAD24-MPI-OpenMP`.

Referências

[1] B N Chandrashekhar and H A Sanjay. Performance analysis of sequential and parallel programming paradigms on cpu-gpus cluster. In *2021 Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV)*, pages 1205–1213, 2021. 10.1109/ICICV50876.2021.9388469.

[2] Cornell Virtual Workshop. MPI Calls Among Threads. Technical report, Cornell University, 2024. https://cvw.cac.cornell.edu/hybrid-openmp-mpi/hybrid-program-types/mpi-threads.

[3] Joshua Hoke Davis, Christopher Daley, Swaroop Pophale, Thomas Huber, Sunita Chandrasekaran, and Nicholas J. Wright. Performance assessment of openmp compilers targeting nvidia v100 gpus, 2020. https://arxiv.org/abs/2010.09454.

[4] Tom Deakin and Timothy G. Mattson. *Programming your GPU with openmp: Performance portability for gpus*, volume 1. The MIT Press, 1 edition, 2023.

[5] CSC – IT Center for Science Ltd. Hybrid CPU programming with OpenMP and MPI. Technical report, CSC – IT Center for Science Ltd, 2022. https://github.com/csc-training/hybrid-openmp-mpi.

[6] Thomas Huber, Swaroop Pophale, Nolan Baker, Michael Carr, Nikhil Rao, Jaydon Reap, Kristina Holsapple, Joshua Hoke Davis, Tobias Burnus, Seyong Lee, David E. Bernholdt, and Sunita Chandrasekaran. Ecp sollve: Validation and verification testsuite status update and compiler insight for openmp. In *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 123–135, 2022. 10.1109/P3HPC56579.2022.00017.

[7] Holly Judge and Mark Bull. Understanding Hybrid MPI + OpenMP Performance. Technical report, EPCC, University of Edinburgh, 2022. https://www.openmp.org/wp-content/uploads/OpenMPBoothTalk-SC22-MPI-OpenMPPerformance.pdf.

[8] Michael Klemm and Jim Cownie. *High performance parallel runtimes: Design and implementation*, volume 1. De Gruyter Oldenbourg, 1 edition, 2021.

[9] MPI Forum. MPI: A Message-Passing Interface Standard Version 2.2. Technical report, MPI Forun, 2009.

[10] NVIDIA. NVIDIA's Next Generation Compute Architecture: Kepler GK110/210. Technical report, NVIDIA, 2014.

[11] OpenMP ARB. OpenMP Application Programming Interface Version 5.0. Technical report, OpenMP ARB, 2018.

[12] Ruud van der. Pas, Eric Stotzer, and Christian Terboven. *Using OpenMP - the next step: affinity, accelerators, tasking, and SIMD*. the MIT Press, 2017.

[13] Hermes Senger and Jaime Freire de Souza. Programe sua GPU com OpenMP. Technical report, ERAD/RS 2022, 2022. https://web.inf.ufpr.br/erad2022/wp-content/uploads/sites/35/2022/08/Minicurso-OpenMP-GPU-V-17_04_22_compressed.pdf.

[14] Gabriel P. Silva, Calebe P. Bianchini, and Evaldo B. Costa. *Programação Paralela e Distribuída com MPI, OpenMP e OpenACC para computação de alto desempenho*. CasaDoCodigo, 2022.

Chapter 4

Architectural Simulation with gem5

Iago Caran Aquino (UNICAMP)
i198921@dac.unicamp.br
http://lattes.cnpq.br/4465472589884134
Lucas Wanner (UNICAMP)
wanner@unicamp.br
http://lattes.cnpq.br/4458409544983212
Sandro Rigo (UNICAMP)
srigo@unicamp.br
http://lattes.cnpq.br/8308517667746974

Abstract

This chapter introduces the fundamental concepts of architectural simulation and explores its critical role in modern computer architecture. It highlights how simulation enables designers to explore, verify, and optimize processor architectures by modeling their behavior and interaction with key system components, such as memory hierarchies and accelerators. The chapter illustrates how gem5's flexibility can be used to easily evaluate different architectural configurations, helping designers make informed decisions and avoid costly physical prototyping. It also discusses the challenges of simulation in increasingly complex systems, strategies for balancing detail and abstraction, and how gem5 can help explore this ample design space.

A case study involving the RISC-V architecture demonstrates how new instructions, particularly matrix-based operations, can be integrated into an existing gem5 model. Matrix multiplication, a key operation in AI workloads, is a practical example of expanding the RISC-V instruction set. The case study illustrates the addition of matrix load, store, and multiply-and-accumulate instructions in gem5, providing readers with hands-on experience in extending architectural models.

Resumo

Este capítulo introduz os conceitos fundamentais de simulação arquitetural e explora seu papel crítico na arquitetura de computadores moderna. Ele destaca como a simulação permite que os projetistas explorem, verifiquem e otimizem arquiteturas de processadores, modelando seu comportamento e interação com componentes chave do sistema como hierarquias de memória e aceleradores. O capítulo ilustra como a flexibilidade do gem5 pode ser usado para avaliar facilmente diferentes configurações arquiteturais, ajudando os projetistas a tomar decisões informadas e evitar a construção de protótipos físicos caros. Ele também discute os desafios da simulação de sistemas cada vez mais complexos, estratégias para equilibrar detalhes e abstração e como 0 gem5 pode ajudar a explorar esse vasto espaço de projeto.

Um estudo de caso envolvendo a arquitetura RISC-V demonstra como novas instruções, particularmente operações baseadas em matrizes, podem ser integradas a um modelo existente no gem5. A multiplicação de matrizes, uma operação essencial em cargas de trabalho de IA, é um exemplo prático de expansão do conjunto de instruções do RISC-V. O estudo de caso ilustra a adição de instruções de carga, armazenamento e multiplicação-acumulação de matrizes no gem5, proporcionando aos leitores uma experiência prática em estender modelos arquiteturais.

4.1. Introduction

Simulation is an indispensable tool across all fields of science, enabling researchers to study and reason about complex systems that would otherwise be too difficult, costly, or impossible to explore through direct experimentation. Whether simulating the behavior of subatomic particles, modeling climate systems, or evaluating the performance of cutting-edge computer architectures, simulations allow scientists to create controlled virtual environments to test hypotheses, analyze interactions, and predict outcomes.

In this chapter, we explore the idea of architectural simulation, which is essential in modern computer architecture because it allows designers to explore, verify, and optimize their designs. Architectural simulation refers to modeling and emulating a processor's behavior and its interaction with key components of a computer system. This includes simulating not just the processor's execution of instructions but also how it communicates with and manages caches, memory hierarchies, buses, and specialized hardware accelerators. For the sake of simplicity, we will refer to architectural simulation as just simulation from now on.

As computing systems become more complex, the importance of simulation grows. Designing hardware involves significant trade-offs between performance, power consumption, and cost. Due to time and financial constraints, physical prototyping for every potential design iteration is impractical. This is where simulation comes into play, offering several benefits:

1. **Design Space Exploration:** Simulation enables the evaluation of multiple design configurations, such as different processor architectures or cache sizes, to determine their impact on performance. This accelerates innovation and enables more informed decision-making early in the design process.
2. **Performance Prediction:** With accurate models, architectural simulations can predict how a system will behave under different workloads, helping designers optimize hardware for specific use cases or applications.
3. **Verification and Validation:** Simulating an architecture allows for early detection of functional errors and design flaws, reducing the risk of costly mistakes during manufacturing. Simulation can serve as a platform for functional validation.
4. **Cost Reduction:** By identifying issues or bottlenecks early, simulation reduces the need for multiple physical prototypes, leading to significant cost savings.

While architectural simulation offers many advantages, it also presents several challenges, particularly when scaling simulations for modern, increasingly complex architectures. Simulation models must balance detail and abstraction:

1. **Detailed Modeling:** In detailed models (e.g., cycle-accurate simulations and RTL), every aspect of the hardware is modeled as accurately as possible, down to the timing of individual components like ALUs or memory controllers. This level of detail ensures high accuracy in predicting performance but drastically increases simulation time and computational resource requirements.

2. **Abstraction in Functional Simulation:** Functional simulations, on the other hand, abstract away the timing details and focus solely on whether the hardware can correctly execute instructions. This abstraction makes functional simulation much faster, which is critical for exploring architectural designs in the early stages. However, these simulations are less suitable for performance analysis, as they don't capture how long operations take, limiting their ability to model real-world execution times.

3. **Hybrid Approaches:** Many modern simulators, such as gem5, offer hybrid approaches where parts of the simulation can be more detailed (for critical components) while others are abstracted. This balance between functional accuracy and timing detail allows for more practical simulations while maintaining reasonable execution times.

4. **Parameterization and Configurability:** Architectural simulators often rely on parameterization to handle the complexity of modern architectures. By adjusting parameters like cache size, memory latency, or core counts, designers can explore various configurations without creating entirely new models for each change. This enables quick experimentation with different design choices.

4.1.1. RISC-V

RISC-V is an open-source, reduced instruction set computing (RISC) architecture that has gained significant traction recently due to its flexibility, scalability, and lack of licensing fees. Unlike traditional proprietary instruction set architectures (ISAs) such as x86 or ARM, RISC-V is designed to be freely available, allowing researchers, developers, and companies to customize it for their own needs without the constraints of intellectual property restrictions. This makes RISC-V particularly appealing for innovation in both academia and industry, as it can be adapted for a wide range of applications, from embedded systems to high-performance computing and AI. Its modular design, with a minimal base instruction set and optional extensions, provides a strong foundation for building general-purpose processors and specialized accelerators.

Simulation plays a critical role in the development of new architectures like RISC-V. Providing a virtual environment for design space exploration accelerates innovation by enabling rapid iteration and evaluation of performance under various workloads, even before physical processors reach the market.

4.1.2. Our Case Study

To make things more interesting, we will introduce gem5 using a case study architecture. The goal is to start with an existing gem5 RISC-V simulator and discuss how to expand it to simulate new instructions.

Matrix multiplication is a cornerstone operation in many artificial intelligence (AI) workloads, particularly Deep Learning. It forms the backbone of neural network computations, where it is used extensively in the forward and backward propagation of data through the network. This operation is vital for the training and inferencing phases of

neural networks, where the multiplication of weight matrices with input data matrices—or subsequent layer outputs—generates the activations that move through the network.

Following this trend, several popular processor architectures are introducing new Matrix Extensions to their ISAs. RISC-V is no exception, although the specification for this extension is still in its discussion phase.

Our case study will introduce new matrix-based instructions in an existing RISC-V gem5 model. Figure 4.1 illustrates the architecture of an Attached In-core Matrix Engine (AIME). The AIME resides inside the existing core, introducing new matrix-specialized registers. The core decodes and dispatches the new instructions to the matrix processing units (MPU). For example, we will introduce matrix-based versions of the load, store, and multiply-and-accumulate (mac) instructions.

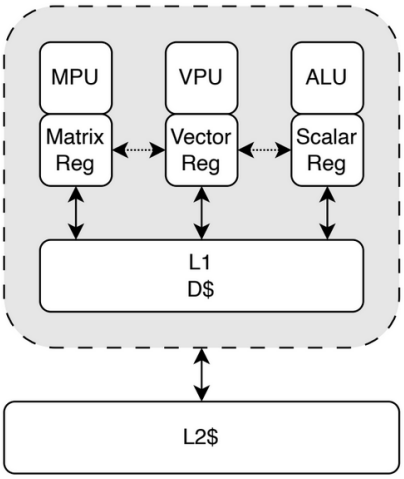


Figure 4.1: An Attached In-core Matrix Accelerator Architecture

4.1.3. Text Organization

Section 4.2 presents a high-level view of the gem5 simulation framework. Section 4.3 includes instructions on setting up gem5 and running basic RISC-V code on the simulator. Section 4.4. Section 4.4 introduces gem5 configuration scripts. Section 4.5 presents a case study wherein the RISC-V model in gem5 is extended to support new instructions for optimized matrix multiplication. Finally, Section 4.6 introduces strategies for performance analysis with gem5.

4.2. The gem5 Simulation Framework

The gem5 simulation framework[Binkert et al. 2011, Lowe-Power et al. 2020] is a widely-used, modular simulation framework designed for computer architecture research and development. It supports a broad range of instruction set architectures (ISAs), including x86, ARM, and RISC-V, and provides flexible simulation modes for both functional and cycle-accurate (timing) simulations. With gem5, users can model complex systems that span from single-core processors to full-system simulations with multicore architectures, memory hierarchies, and peripherals. It offers extensive configurability, making it ideal

for exploring design choices and conducting performance analysis under various scenarios.

gem5 originates in a fusion between two well-established simulators: M5 and GEMS. The M5 simulator[Binkert et al. 2006], developed at the University of Michigan, was known for its focus on full-system simulation, providing detailed models of CPUs and memory systems. GEMS[Martin et al. 2005], developed at the University of Wisconsin-Madison, offered advanced memory system modeling and support for parallel simulations. In 2011, these two simulators were combined to form gem5, bringing together both strengths. Since then, gem5 has evolved significantly, with ongoing contributions from a global community of researchers and developers. The gem5 simulator is released under a BSD (Berkeley Software Distribution) 3-clause license.

One of gem5's standout capabilities is its dual-mode simulation: full-system (FS mode) and syscall emulation (SE mode). In FS mode, gem5 simulates the complete hardware environment in detail, enabling the execution of unmodified operating systems. This comprehensive approach allows for intricate interactions between the application and the underlying system, capturing low-level operational details. Conversely, in SE mode, gem5 focuses solely on running the application as a user-level process, with the simulator intercepting and responding to system calls. This mode reduces the granularity of the simulation, thereby accelerating execution times. SE mode is particularly beneficial when the primary objective is to analyze the application's interaction with the memory hierarchy.

4.2.1. The gem5 System Architecture Model

In gem5, the design of the CPU model exemplifies a clear delineation between the microarchitecture and the instruction set architecture (ISA). This distinction is implemented through two primary classes: Static Instruction and Execution Context. The Static Instruction class and its derivatives articulate the behavior of individual instructions with methods that detail their interaction with various stages of the CPU pipeline. Conversely, the Execution Context class manages all alterations to the CPU's state, storing essential information such as register values, program counters, and memory mappings. It further facilitates access to memory ports and other architectural features offered by the CPU. This architectural separation enables the association of various CPU models with different ISAs (e.g., RISC-V, ARM, x86) without necessitating specific adaptations for each configuration. Figure 4.2, adapted from documentation [gem5 2022, Black et al. 2010] illustrates the components of the gem5 CPU Model, indicating which elements depend on the ISA and which do not. gem5 includes five processor models:

- **AtomicSimpleCPU:** a purely functional, in-order model that is suited for cases where a detailed model is not necessary, like warm-up periods, client systems that are driving a host, or just testing to make sure a program works. It uses atomic memory accesses;
- **TimingSimpleCPU:** similar to the AtomicSimpleCPU, but uses timing memory accesses and stalls on cache accesses, waiting for the memory system to respond before proceeding;

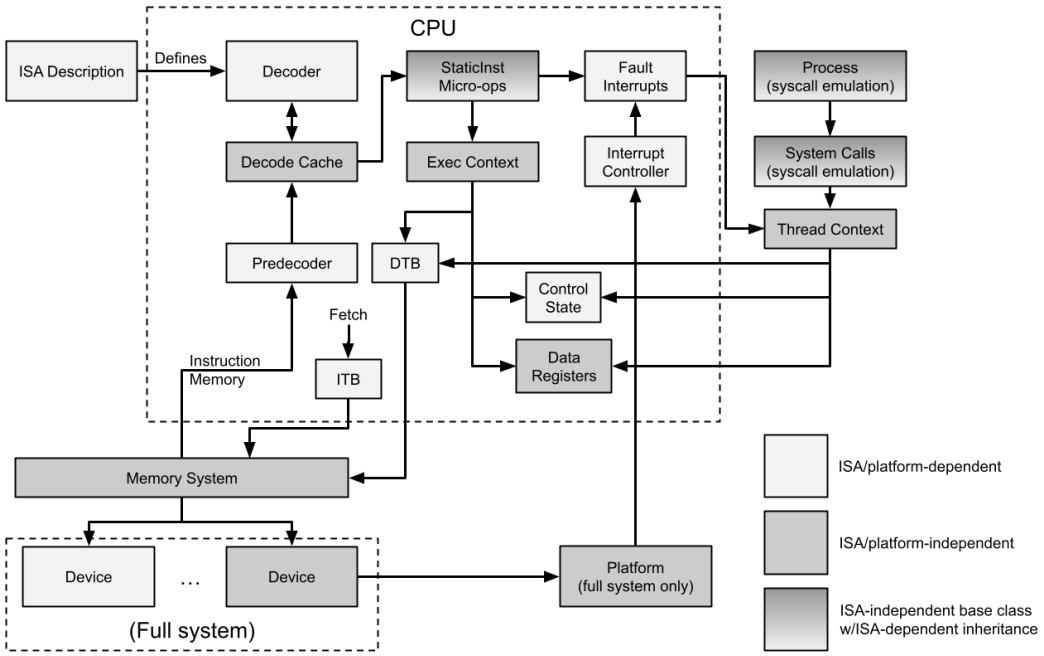


Figure 4.2: gem5's architecture diagram adapted from[gem5 2022, Black et al. 2010].

- **MinorCPU:** is an in-order processor model with a fixed pipeline but configurable data structures and execution behavior designed to model processors that enforce strict in-order execution.;

- **O3CPU:** is a detailed, timing-accurate out-of-order CPU model, loosely based on the Alpha 21264, which executes instructions at the execute stage of the pipeline, unlike most other simulators;

- **TraceCPU:** is a model that replays dependency- and timing-annotated traces from the O3 model, allowing for faster yet reasonably accurate memory-system performance exploration.

4.2.2. gem5 standard library

The **gem5 standard library (stdlib)** is a set of components that allows users to create systems quickly. These components act as reusable and modular blocks that can be connected to simplify the process of constructing custom systems, they range from memory and cpu models to complete cache hierarchies and systems. These components will be extended while developing your custom configurations using object-oriented semantics.

Let's say we want to implement a cache model for our system. We can extend the Cache class to express our L1 cache configuration, as shown in Listing 4.1, and then extend this L1 class to differentiate the data cache from the instruction cache.

```
class L1Cache(Cache):
    assoc = 2
    tag_latency = 2
    data_latency = 2
    response_latency = 2
    mshrs = 4
    tgts_per_mshr = 20

    def connectCPU(self, cpu):
        raise NotImplementedError

    def connectBus(self, bus):
        self.mem_side = bus.cpu_side_ports


class L1ICache(L1Cache):
    size = '16kB'

    def connectCPU(self, cpu):
        self.cpu_side = cpu.icache_port


class L1DCache(L1Cache):
    size = '64kB'

    def connectCPU(self, cpu):
        self.cpu_side = cpu.dcache_port
```

Listing 4.1: Example of new L1 Cache component

4.3. Getting started with gem5

The gem5 Getting Started Guide [Gem5 Project 2024] includes detailed instructions on setting up, building, and running basic examples. Here, we provide a summarized outline of this process, which includes setting up required packages, installing a RISC-V compiler toolchain, downloading and building gem5, and running a simple example using preconfigured systems. These steps were tested for an Ubuntu 22.04.5 LTS Linux distribution, both natively and in a Docker container. Reference Dockerfiles for the toolchain and gem5 are available at https://github.com/LSC-Unicamp/gem5-sscad-2024.

4.3.1. Required Packages

As illustrated in Listing 4.2 required packages for the GNU toolchain for RISC-V (line 1) and gem5 (line 2) may be installed in Ubuntu using `apt`.

```
1 sudo apt install -y autoconf automake autotools-dev curl python3 python3-pip libmpc-dev
      libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool
      patchutils bc zlib1g-dev libexpat-dev git
2 sudo apt install -y build-essential git m4 scons zlib1g zlib1g-dev libprotobuf-dev
      protobuf-compiler libprotoc-dev libgoogle-perftools-dev python-is-python3
```

Listing 4.2: APT command for installing requires packages for the RISC-V GNU Toolchain (line 1) and gem5 (line 2)

4.3.2. GNU Toolchain for RISC-V

To compile RISC-V code on platforms that don't natively support this architecture, we use the GNU RISC-V toolchain, a collection of compilers, assemblers, and related tools

tailored for the RISC-V instruction set architecture (ISA). This setup allows for cross-compilation, where code is compiled on one platform (e.g., an x86 machine) to run on a different target platform, in our case a RISC-V processor.

The build process for the GNU RISC-V toolchain is illustrated in Listing 4.3. The tools are installed under `/opt/riscv` and must be added to the system path. After building the toolchain, two common targets are available: ELF (Executable and Linkable Format) and Linux. ELF is typically used for bare-metal applications, running directly on hardware without an operating system, while the Linux target generates binaries designed to run on systems with glibc and a Linux OS.

```
export RISCV=/opt/riscv
export PATH=$PATH:$RISCV/bin

git clone --recursive https://github.com/riscv/riscv-gnu-toolchain -b 2024.04.12
cd riscv-gnu-toolchain
./configure --prefix=$RISCV --enable-multilib
make -j8
make linux -j8
cd ..
rm -rf riscv-gnu-toolchain
```

Listing 4.3: Build steps for the GNU RISC-V toolchain

Once the toolchain is installed, code can be compiled using gcc with the O3 optimization flag, as illustrated in Listing 4.4

```
riscv64-unknown-elf-gcc -O3 code.c -o code.riscv
```

Listing 4.4: Compiling with GCC

The instructions of the generated binary can be inspected using the `objdump` tool, which disassembles the binary back into assembly language, as shown in Listing 4.5:

```
riscv64-unknown-elf-objdump -S code.riscv | less
...
0000000000010104 <main>:
10104:       6549                    lui     a0,0x12
10106:       1141                    addi    sp,sp,-16
10108:       5c850513                addi    a0,a0,1480 # 125c8 <__errno+0x8>
1010c:       e406                    sd      ra,8(sp)
1010e:       418000ef                jal     10526 <puts>
10112:       60a2                    ld      ra,8(sp)
10114:       4501                    li      a0,0
10116:       0141                    addi    sp,sp,16
10118:       8082                    ret
...
```

Listing 4.5: Inspecting the Binary with objdump

For applications not requiring extensive OS support, gem5 can be used to run the resulting binary in SE mode, simulating the RISC-V architecture without an operating system.

4.3.3. Building and Running gem5

Gem5 is compiled using the scons build system, which is a Python-based build tool. To build Gem5 for the RISC-V architecture, the process starts by cloning the repository, followed by navigating to the Gem5 directory, and then compiling for RISC-V. The build

process can be accelerated by utilizing all available CPU cores through the use of the -j option with the number of processing units, as shown in Listing 4.6.

```
git clone https://github.com/gem5/gem5
cd gem5
scons build/RISCV/gem5.opt -j`nproc`
```

Listing 4.6: Building Gem5

4.3.4. Using gem5 default configuration scripts

A collection of configuration scripts is available within the `configs` directory covering different features of the framework, such as checkpoints, full system simulation, different CPU models, among others. We recommend exploring the examples there.

Taking `configs/learning_gem5/part1/simple-riscv.py` as an example, we have a basic system using the RISC-V ISA with the TimingSimple CPU model running a program that prints "Hello World!", the complete output of the simulation is exemplified by the output in Listing 4.7.

```
gem5 Simulator System.  https://www.gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 24.0.0.1
gem5 compiled Oct  5 2024 20:32:53
gem5 started Oct  5 2024 23:35:42
gem5 executing on desktop, pid 41833
command line: ./gem5/build/RISCV/gem5.opt
    ./gem5/configs/learning_gem5/part1/simple-riscv.py

Global frequency set at 1000000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and pdf.
src/mem/dram_interface.cc:690: warn: DRAM device capacity (8192 Mbytes) does not match
    the address range assigned (512 Mbytes)
src/arch/riscv/isa.cc:276: info: RVV enabled, VLEN = 256 bits, ELEN = 64 bits
src/base/statistics.hh:279: warn: One of the stats is a legacy stat. Legacy stat is a
    stat that does not belong to any statistics::Group. Legacy stat is deprecated.
system.remote_gdb: Listening for connections on port 7000
Beginning simulation!
src/sim/simulate.cc:199: info: Entering event queue @ 0.  Starting simulation...
Hello world!
Exiting @ tick 499797000 because exiting with last active thread context
```

Listing 4.7: Basic hello world running in gem5

4.4. gem5 configuration

Gem5's flexibility comes from the Python interface for describing systems, which allows multiple configurations to be executed without recompiling the entire framework. The basic components of such a script describe various aspects of the system, such as CPU type, memory hierarchy, cache configuration, and simulation parameters. The script specifies how to wire together different system components and how the simulation should be executed.

Expanding on the example presented by the `simple-riscv.py` script, we will add a simple L1 cache and read the binary to be executed from the command line argument. Listing 4.8 shows the modified script. From here, we could, for example, add an

```python
from m5.objects import *

# Define system components
system = System()
system.clk_domain = SrcClockDomain(clock='1GHz', voltage_domain=VoltageDomain())

# Set up the CPU
system.cpu = RiscvTimingSimpleCPU()

# Set up the memory system
system.mem_mode = 'timing'  # Use timing model for memory accesses
system.mem_ranges = [AddrRange('512MB')]
system.membus = SystemXBar()
system.system_port = system.membus.cpu_side_ports

# Configure L1 cache
system.cpu.icache = Cache(size='32kB', assoc=2)
system.cpu.dcache = Cache(size='32kB', assoc=2)
system.cpu.icache.cpu_side = system.cpu.icache_port
system.cpu.dcache.cpu_side = system.cpu.dcache_port
system.cpu.icache.mem_side = system.membus.cpu_side_ports
system.cpu.dcache.mem_side = system.membus.cpu_side_ports

# Set up memory controller
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.mem_side_ports

# Load binary
thispath = os.path.dirname(os.path.realpath(__file__))
binary = os.path.join(
    thispath,
    sys.argv[1],
)
system.workload = SEWorkload.init_compatible(binary)

# Run simulation
process = Process()
process.cmd = [binary] + sys.argv[2:]
system.cpu.workload = process
system.cpu.createThreads()

root = Root(full_system=False, system=system)
m5.instantiate()

print(f"Beginning simulation!")
exit_event = m5.simulate()
print(f"Exiting @ tick {m5.curTick()} because {exit_event.getCause()}")
```

Listing 4.8: A Simple RISC-V System Configuration

L2 cache just by instantiating another Cache object and connecting it between the L1 and the memory bus.

4.4.1. Full System simulation with gem5-resources

Beyond the `stdlib` components, there is gem5-resources, which is a repository and a framework designed to provide a collection of pre-built resources, such as disk images, binaries, benchmarks, and workloads, known and proven compatible with the gem5 architecture simulator.

One of the uses of this repository is to easily simulate full systems since it eliminates the need to prepare an image with the kernel and operating system for execution. For example, to run a test with Ubuntu 20.04, we set the workload to `workload = obtain\_resource("riscv-ubuntu-20.04-boot", resource\_version="3.0.0")` and the image with the system will be downloaded automatically.

The `riscvmatched-fs.py` file in `configs/example/gem5_library/` illustrates the complete configuration for this workload using a prebuilt system. The details of configuring a script for Full System simulation are beyond our scope, but there are many more details to be configured, such as I/O devices and device tree generation.

4.5. Expanding gem5

Extending the ISA (Instruction Set Architecture) is a fundamental process in processor development, especially when adapting a general-purpose architecture like RISC-V for specific workloads, such as matrix operations. This section focuses on adding new instructions to optimize matrix multiplication, a key operation in many high-performance and AI-related applications, as detailed in Section 4.1.2.

4.5.1. The ISA Extension

Instruction specification is critical to processor architecture development, as it defines how a processor communicates with software and executes tasks. An instruction's behavior determines its function and must be clearly defined to ensure the processor implementation correctly interprets and executes each one as intended.

RISC-V's Matrix Specification is still in the discussion stages, meaning it can change a lot. A matrix multiplication, for example, can be performed by considering each line and column as vectors or by working on small blocks of the full matrix. Both approaches have different design considerations that impact the overall performance and efficiency of the system.

One of the key challenges in adding new instructions to RISC-V is the strict limitation on instruction sizes and opcode availability. The base RISC-V ISA supports fixed 32-bit instructions, so we must carefully pack essential information such as operation type, data type, operands, and other metadata into a limited space. This constraint impacts the immediate design of the instruction and could limit future extensibility if not handled carefully. Efficiently encoding these instructions while preserving flexibility is crucial for balancing the immediate needs of matrix operations with the long-term goals of keeping the ISA lightweight and adaptable.

Mnemonic	Format	Behavior
mls	load/store	
mss	load/store	
mmacf	arithmetic	multiply and accumulate
maddf	arithmetic	add matrices
msubf	arithmetic	subtract matrices
mm_xmu	reg-reg mov	fill matrix with scalar value
mzero	reg-reg mov	move zero to the entire destination register

Table 4.1: Matrix Extension to the RISC-V ISA

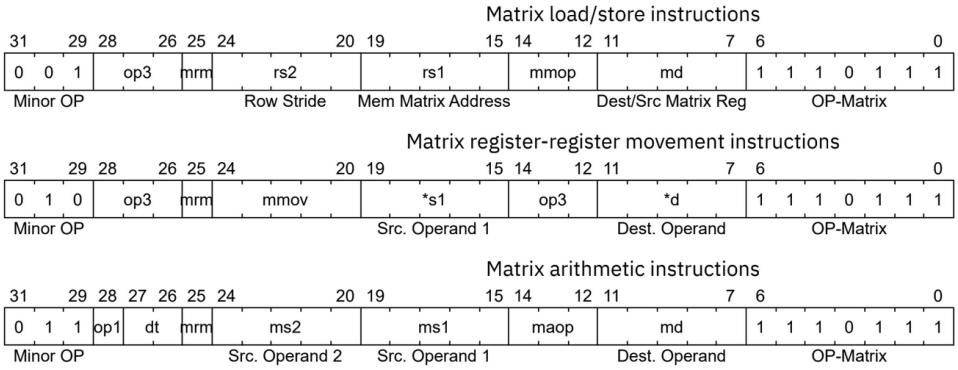


Figure 4.3: Instruction encodings for the Matrix Extension.

Since the final matrix specification for RISC-V has not yet been published, the already-established vector extension has inspired our design choices. The vector extension efficiently handles data parallelism, and its flexible encoding allows for operations on various data types and sizes. Drawing from these principles, we have drafted a preliminary set of instructions to handle matrix operations efficiently. These instructions leverage matrix-specific registers and provide the essential functionality for matrix-based software, including load/store operations and arithmetic functions such as addition, subtraction, and multiplication.

The following instructions highlight the key operations that form the basis of this extension. These instructions, detailed in Figure 4.3 and summarized in Table 4.1, represent a first best estimate of the necessary operations to support matrix computations. These include basic arithmetic instructions, data movement between matrix and scalar registers, and memory operations. For the sake of simplicity, in our example, the instructions are fixed to 32-bit elements and a fixed four-by-four panel. This fixed format was chosen to streamline early development and debugging. Later development phases can introduce more sophisticated encoding and support for different data types. Although the exact encoding may evolve as the specification matures, these examples illustrate the core functionality we aim to integrate into the RISC-V model.

4.5.2. How to Include a New Instruction

This section discusses how we add the new matrix registers to our gem5 model and then advances to including the new instructions in the decoder.

4.5.2.1. Adding new registers

In this example, we are adding new registers that hold panels of our matrix. These panels will be 4x4. We do not have a ready-to-use type for this shape, so we are borrowing the *MatStore* container from ARM's implementation to use as our register datatype. For simplicity, we took only the essential part of it. Following the same template as the other register types, we will create a file inside the *regs* folder to define the matrix registers. Note that we are using the MatStore class that was described before. We have 32 matrix registers of 256 bits named m0– m31.

```
namespace gem5 {
namespace RiscvISA {
constexpr unsigned MaxMLenInBits = 256;
constexpr unsigned MaxMLenInBytes = MaxMLenInBits >> 3;

// Import the MatStore type that will be our register type
using MatRegContainer = gem5::MatStore;
using mreg_t = MatRegContainer;

// Define the number of registers this architecture uses
const int NumMatrixStandardRegs = 32;
const int NumMatrixInternalRegs = 0;
const int NumMatrixRegs = NumMatrixStandardRegs + NumMatrixInternalRegs;

// Define the register names
const std::vector<std::string> MatrixRegNames = {
     "m0",   "m1",   "m2",   "m3",   "m4",   "m5",   "m6",   "m7",
     "m8",   "m9",  "m10",  "m11",  "m12",  "m13",  "m14",  "m16",
    "m17",  "m18",  "m19",  "m20",  "m21",  "m22",  "m23",  "m24",
    "m25",  "m26",  "m27",  "m28",  "m29",  "m30",  "m31",  "m32"
};

static inline TypedRegClassOps<RiscvISA::MatRegContainer> matRegClassOps;
inline constexpr RegClass matRegClass =
    RegClass(MatRegClass, MatRegClassName, NumMatrixRegs, debug::MatRegs).
        ops(matRegClassOps).
        regType<MatRegContainer>();
} // namespace RiscvISA
} // namespace gem5
```

Listing 4.9: Extract of the register definition.

The matrix register header file must be with `#include "arch/riscv/regs/matrix.hh"` added to the following files located under `gem5/src/arch/riscv/`: `utility.hh`, `pcstate.hh`, `isa/includes.isa` and `isa.cc`.

In the `isa.cc` file, we also need to comment on the following line to allow the ISA to include the new registers we defined in the matrix header:

```
RegClass matRegClass(MatRegClass, MatRegClassName, 0, debug::MatRegs);
```

We can now rebuild gem5. However, nothing will apparently change; we need to add instructions to start using these registers.

4.5.2.2. Expanding the decoder

Following the encodings we defined in Figure 4.3, we will expand the ISA description. In the *decoder.isa* file, we will first find a suitable place for our new extension. From the RISC-V ISA specification [Waterman et al. 2014], we found that the reserved OPCODE 1110111 is not in use yet, thus we will be using it for our matrix extension. The decoder structure is similar to multiple nested switch clauses, each defining a bitfield that will be used to index the instruction, so we will first decide in which QUADRANT our instruction is, 0b11 or 0x3 for 32b instructions. Then, we are addressing the 5-bit opcode, 0b11101 or 0x1D. With this, we can finally add our new instructions as illustrated by the structure in Listing 4.10.

```
decode QUADRANT default Unknown::unknown() {
    ...
    0x3: decode OPCODE5 {
        ...
        0x1D: // Our new instructions here
        ...
    }
    ...
}
```

Listing 4.10: Extract of the ISA definition.

Our operations will be decoded by the *Minor OP* field, bits 31 to 29, which we will be adding to the bitfield list file, as any other entry does not define it. We will also add the bitfield definitions for the source and destination matrix registers used by the instructions, as illustrated by Listing 4.11. We will not add the other fields described in the encoding because they map details we will not address in this chapter. The caption of the following Listings will show the path in our repository where the reader can find the file being modified.

```
// Matrix instructions
def bitfield MMINOP     <31:29>;

def bitfield MRD        mrd;
def bitfield MRS1       mrs1;
def bitfield MRS2       mrs2;
def bitfield MRS3       mrs3;
```

Listing 4.11: gem5/src/arch/isa_parser/isa/bitfields.isa

The registers don't define the bit ranges. They reference the definitions made in the *types.hh* file as shown in 4.12.

```
// matrix
Bitfield<11,  7>    mrd;
Bitfield<19, 15>    mrs1;
Bitfield<24, 20>    mrs2;
Bitfield<11,  7>    mrs3;
```

Listing 4.12: gem5/src/arch/riscv/types.hh

For the registers to be usable in our instructions, we must add them to the *operands.isa* file as shown in Listing 4.13. This tells gem5 that there is some information encoded in the instruction that the instruction behavior and the expected type will use.

```
'Mrd':  MatRegOp('mc', 'MRD',  'IsMatrix', 1),
'Mrs1': MatRegOp('mc', 'MRS1', 'IsMatrix', 2),
'Mrs2': MatRegOp('mc', 'MRS2', 'IsMatrix', 3),
'Mrs3': MatRegOp('mc', 'MRS3', 'IsMatrix', 4),
```

Listing 4.13: gem5/src/arch/riscv/isa/operands.isa

Now, we can specify our instructions in the decoder file as shown in Listing 4.14. For memory and arithmetic, we also use the *FUNCT3* field to differentiate between load and store operations or multiply and accumulate, addition, or subtraction operations.

```
...
0x1d: decode MMINOP {
    0x1: decode FUNCT3 {
        0x4: MatrixLoadOp::mls({{
            for (int k = 0; k < 4; k++) {
                Mrd_uw[i + k] = Mem_mc.as<uint32_t>()[k];
            }
        }});
        0x5: MatrixStoreOp::mss({{
            for (int k = 0; k < 4; k++) {
                Mem_mc.as<uint32_t>()[k] = Mrd_uw[i + k];
            }
        }});
    }
    0x2: ScalarMatrixMoveOp::mmv_xmu({{
        Mrd_uw[i] = Rs1_uw;
    }});
    0x3: decode FUNCT3 {
        0x0: MatrixArithLineOp::mmacf({{
            uint32_t row = i / 4;
            uint32_t col = i % 4;
            uint32_t row_stride = 4;

            float acc = Mrs3_mc->as<float>()[i];

            for (int j = 0; j < 4; j++) {
                acc += Mrs1_mc->as<float>()[row * row_stride + j] *
                    Mrs2_mc->as<float>()[col * row_stride + j];
            }

            Mrd_sf[i] = acc;
        }});
        0x1: MatrixArithOp::maddf({{
            Mrd_sf[i] = Mrs1_sf[i] + Mrs2_sf[i];
        }});
        0x2: MatrixArithOp::msubf({{
            Mrd_sf[i] = Mrs1_sf[i] - Mrs2_sf[i];
        }});
    }
}
...
```

Listing 4.14: Extract of the riscv/isa/decoder.isa definition.

Each instruction derives from one template that we will be declaring and a name. There is also the specification of the instruction behavior that will be composed inside

the template. A script named isa_parser aggregates and expands all the *.*isa* files into the decoder *.*cpp* files.

Note that each register reference is followed by a suffix defined in the `src/arch/riscv/isa/operands.py` file. These are used to indicate which data type should be used to interpret the contents of the registers. Since we added a new register type, we must add the code in Listing 4.15 to the operand_types list in *operands.py*.

```
def operand_types {{
    ...
    'vc'    : 'RiscvISA::VecRegContainer',
    'mc'    : 'RiscvISA::MatRegContainer'
}};
```

Listing 4.15: Extract of the riscv/isa/operands.py file

We still need to define the instruction formats, i.e., the templates that isa_parser uses to expand the instruction definitions. Taking the *mls* as an example, we have the *MatrixLoadOp* format, which is defined in the `formats/matrix.isa` file that we are going to create.

In the format definition exemplified in Listing 4.16, we specify this operation's default parameters and base template, which will generate a tuple with all the necessary parts to compose one instruction. The base template used for the instruction defines which templates are used for each one of these parts, which are listed below:

- **Header output:** used to define the header file declaring the instruction class. Here, we declare the number of source and destination registers and the functions of the instruction in the pipeline. Most will have only an execute method, but memory instructions may have initiateAcc and completeAcc methods;
- **Decoder output:** defines a constructor for the instruction. This is where the necessary registers are specified;
- **Decode block:** defines the decoder entry, which is how a new object of the class should be created;
- **Exec output:** specifies the implementation of the instruction, manipulating the registers and ExecContext.

The base template shows one important characteristic of gem5: the division of instructions into macro-ops and micro-ops. The macro ops are used by the frontend portion of the simulation, aggregating the necessary information for the instruction and advancing in the pipeline as necessary. The micro-ops are used on the backend portion, allowing complex operations to be broken up into fine-grained operations, allowing for more simulation detail.

Listing 4.17 illustrates the templates used to compose the format. They reflect the parts mentioned before and illustrate how we define the number of registers the instruction will use in the Declare portion, the behavior of the instruction in the Execute portion, and

```
def MMemBase(name, Name, ea_code, code, base_class, mem_flags, inst_flags,
             declare_template_base=MatrixMacroDeclare, decode_template=MatrixDecode,
             exec_template_base='', macro_construtor=MatrixMacroConstructor,
             op_class='', is_macroop=True):
    iop = InstObjParams(name, Name, base_class, { 'code': code, 'ea_code': ea_code,
        'op_class': op_class },
          inst_flags)
    header_output   = declare_template_base.subst(iop)
    decoder_output  = macro_construtor.subst(iop)
    decode_block    = decode_template.subst(iop)
    exec_output     = ''

    if not is_macroop:
        return (header_output, decoder_output, decode_block, exec_output)

    micro_class_name = exec_template_base + 'MicroInst'
    microiop = InstObjParams(name + '_micro',
        Name + 'Micro', exec_template_base + 'MicroInst', {'ea_code': ea_code, 'code':
            code},
        inst_flags)

    microDeclTemplate = eval(exec_template_base + 'MicroDeclare')
    microConsTemplate = eval(exec_template_base + 'MicroConstructor')
    microExecTemplate = eval(exec_template_base + 'MicroExecute')
    microInitTemplate = eval(exec_template_base + 'MicroInitiateAcc')
    microCompTemplate = eval(exec_template_base + 'MicroCompleteAcc')
    header_output = microDeclTemplate.subst(microiop) + header_output
    decoder_output = microConsTemplate.subst(microiop) + decoder_output
    micro_exec_output = (microExecTemplate.subst(microiop) +
        microInitTemplate.subst(microiop) +
        microCompTemplate.subst(microiop))
    exec_output += micro_exec_output

    return (header_output, decoder_output, decode_block, exec_output)

def format MatrixLoadOp(
    code, ea_code={{ EA = Rs1 + ((uint64_t) microIdx / 4) * Rs2 + (microIdx % 4) * 4;
        }},
    mem_flags=[], *flags ) {{
    (header_output, decoder_output, decode_block, exec_output) = \
        MMemBase(name, Name, ea_code, code, 'MatrixLoadMacroInst',
                 mem_flags, flags, op_class='MemReadOp',
                 macro_construtor=MatrixMacroConstructor,
                 exec_template_base='MatrixLoad')
}};
```

Listing 4.16: Extract of the matrix format specification

```
def template MatrixLoadMicroDeclare {{
    class %(class_name)s : public %(base_class)s
    {
      private:
        RegId srcRegIdxArr[2];
        RegId destRegIdxArr[1];
      public:
        %(class_name)s(ExtMachInst _machInst);
        %(class_name)s(ExtMachInst _machInst, uint32_t _microIdx);

        Fault execute(ExecContext *, trace::InstRecord *) const override;
        Fault initiateAcc(ExecContext *, trace::InstRecord *) const override;
        Fault completeAcc(PacketPtr, ExecContext *,
                          trace::InstRecord *) const override;

        using %(base_class)s::generateDisassembly;
    }; }};

def template MatrixLoadMicroExecute {{
    Fault
    %(class_name)s::execute(ExecContext *xc,
        trace::InstRecord *traceData) const
    {
        Addr EA;
        %(op_decl)s;
        %(op_rd)s;
        uint32_t i = microIdx;
        for (int row; row < 4; row++) {
            EA = Rs1 + ((uint64_t) i / 4) * Rs2 + (i % 4) * 4;
            const size_t mem_size = 16;
            std::vector<bool> byte_enable(mem_size, true);
            Fault fault = xc->readMem(EA, (uint8_t*) &Mem, mem_size,
                                      memAccessFlags, byte_enable);
            if (fault != NoFault)
                return fault;
            %(code)s;
            i += 4;
        }
        %(op_wb)s;
        return NoFault;
    } }};

def template MatrixLoadMicroConstructor {{
%(class_name)s::%(class_name)s(ExtMachInst _machInst, uint32_t _microIdx)
  : %(base_class)s(
        "%(mnemonic)s", _machInst, %(op_class)s, _microIdx)
{
    %(set_reg_idx_arr)s;
    _numSrcRegs = 0;
    _numDestRegs = 0;
    setDestRegIdx(_numDestRegs++, matRegClass[_machInst.mrd]);
    _numTypedDestRegs[MatRegClass]++;
    setSrcRegIdx(_numSrcRegs++, intRegClass[_machInst.rs1]);
    setSrcRegIdx(_numSrcRegs++, intRegClass[_machInst.rs2]);
} }};
```

Listing 4.17: Extract of the matrix templates specifications

the register values that need to be fetched in the Constructor portion. Note the presence of `%(...)s` structures, these define smaller template pieces that will be put expanded during the build process. The `%(code)s` template is an important one, as it is the one that brings the behavior we defined in the `decoder.isa` into the instruction definition.

These different parts can be shaped to specific instructions or be flexible and reused between multiple instructions. The *isa_parser* will generate the complete C++ classes representing our instructions.

Note that the base template has a *base_class* parameter, this indicates which base class the instruction follows, these being defined in the `src/arch/riscv/insts/matrix.hh` file. These classes group the instructions by common behavior and allow us to define parameters that are shared between them, as is the case of the flags that define if an instruction is a matrix operation, a load or store, etc. These flags are used by gem5 to identify the functional units that will process the instructions for more detailed CPU models. The base classes for the matrix load instructions are illustrated in Listing 4.18.

```
class MatrixLoadMacroInst : public MatrixMemMacroInst {
  protected:
    MatrixLoadMacroInst(const char* mnem, ExtMachInst _machInst, OpClass __opClass)
        : MatrixMemMacroInst(mnem, _machInst, __opClass)
    { this->flags[IsLoad] = true; }
};

class MatrixLoadMicroInst : public MatrixMicroInst {
  protected:
    Request::Flags memAccessFlags;
    MatrixLoadMicroInst(const char* mnem, ExtMachInst _machInst, OpClass __opClass,
        uint32_t _microIdx)
        : MatrixMicroInst(mnem, _machInst, __opClass, _microIdx), memAccessFlags(0)
    { this->flags[IsLoad] = true; }
};
```

Listing 4.18: Extract of the matrix instruction base classes

Some files need other specific changes for the instructions to work correctly, so we recommend looking at the full details in the GitHub repository to get a reference of all the necessary adjustments. These are mostly one-off changes that you don't need to perform again if you're going to add more instructions.

The same logic applies to the definitions of arithmetic instructions, which differ only in that they do not implement the *initiateAcc* and *completeAcc* methods.

4.5.3. The Matrix Multiplication Kernel

Testing is one of the most important aspects of designing new instructions for an architecture. Usually, the software development kit is not done or fully updated to include the ISA extensions. So, the designer does not have a compiler, assembler, or other tools to validate the simulation model. Let's discuss strategies to overcome this obstacle.

Suppose you want to test your matrix ISA extension using the most basic implementation of matrix multiplication, also called naive or iterative multiplication. It consists of three nested loops. The first two go through every one of the elements of the destination, while the third accumulates the sum of every multiplication of the element of a line

by the element of a column.

Due to the complexity and time required to modify a compiler, we wrote a kernel to use our new instructions with inline assembly instead of compiler-generated instructions. Inline assembly can be quickly adapted or modified, making it easier to maintain across different header versions, which is necessary for faster iteration and testing. To simplify our programming, a header file defines Macros to generate the correct encoding for matrix instructions in a similar form to how it would be done with compiler support.

The Macros define a function-like interface that receives as parameters the necessary operands to compose the entire instruction, like the source and destination registers. In addition, the instruction names encode some of the information in their own names, as is the case of the move instructions, where the *XMU* part of it indicates that the source register is scalar (X), the destination is a matrix (M), and the data should be interpreted as unsigned (U). The definition gets every operand as a string of bits. It merges it with the fixed parts of the instruction, generating a 32-bit word incorporated inside the code as text but interpreted correctly by the simulators. There is also a pseudo-instruction for MZERO, which fills a matrix register with zeroes. The macro definitions can be seen in Listing 4.19. Notice that `mX` denotes the `Xth` matrix register. Other register names follow the usual naming convention for RISC-V.

```
#define MLS(md, rs1, rs2)    ".word 0b0010001" rs2  rs1 "100" md "1110111"
#define MSS(md, rs1, rs2)    ".word 0b0010001" rs2  rs1 "101" md "1110111"

#define MMV_XMU(md, rs1)     ".word 0b010000100001" rs1 "000" md "1110111"
#define MZERO(md)            MMV_XMU(md, X0)

#define MMACF(md, ms1, ms2) ".word 0b0110101" ms2  ms1 "000" md "1110111"
#define MADDF(md, ms1, ms2) ".word 0b0110101" ms2  ms1 "001" md "1110111"
#define MSUBF(md, ms1, ms2) ".word 0b0110101" ms2  ms1 "010" md "1110111"
```

Listing 4.19: Extract of the definitions of the RISC-V Matrix Header.

Using these definitions, we can adapt the matrix multiplication kernel presented in Listing 4.20 to leverage the matrix instructions and perform the multiplication by panels instead of individual elements, enabling higher throughput. The result is shown in Listing 4.21.

As our registers store 4x4 matrices, note that the kernel using matrix multiplication advances four elements per iteration instead of one in the naive multiplication example.

```
void naive_multiply(int M, int N, int K, int ldc, int ldb, int lda, float *C, float *A,
    float *B) {
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            float sum = 0.0f;
            for (int k = 0; k < K; k++) {
                sum = (double) A[i * lda + k] * (double) B[k * ldb + j] + (double) sum;
            }
            C[i * ldc + j] = sum;
        }
    }
}
```

Listing 4.20: Basic multiplication kernel.

```
void matrix_multiply(int M, int N, int K, int ldc, int ldb, int lda, float *C, float
    *A, float *B){
  float *B_t = (float *) malloc(N*K*sizeof(float));
  float *C_ptr = C, *A_ptr = A, *B_ptr = B_t;
  transpose(B, B_t, N, K);
  ldb = K;
  int elem_per_row = 4;                  // Number of elements per row

  // Matrix Multiplication
  for (int i = 0; i < M; i += elem_per_row) {
    for (int j = 0; j < N; j += elem_per_row) {
      C_ptr = C + j + i*ldc;

      asm(MZERO(M2));                    // Reset the accumulator

      for (int k = 0; k < K; k += elem_per_row) {
        A_ptr = A + k + i*lda;
        B_ptr = B_t + k + j*ldb;

        asm(
          "mv t0, %0\n\t"
          "mv t1, %1\n\t"
          MLS(M0, T1, T0)                // Load A
          "mv t0, %2\n\t"
          "mv t1, %3\n\t"
          MLS(M1, T1, T0)                // Load B
          MMACF(M2, M0, M1)              // Multiply-Accumulate
          :: "r" (lda * 4), "r" (A_ptr), "r" (ldb * 4), "r" (B_ptr)
          : "t0", "t1", "memory"
        );
      }

      asm(
        "mv t0, %0\n\t"
        "mv t1, %1\n\t"
        MSS(M2, T1, T0)                  // Store C
        :: "r" (ldc * 4), "r" (C_ptr)
        : "t0", "t1", "memory"
      );
    }
  }
}
```

Listing 4.21: Basic multiplication kernel using matrix instructions.

4.6. Performance Analysis

Performance analysis is a central task for researchers and developers to gain insights into the performance characteristics of architectural models. gem5 provides several features and options supporting detailed performance analysis and high-level performance evaluation. We start with an overview of the primary options and features available in gem5 for performance analysis. Then, we present more detailed examples of some simulation statistics and the Python-based configuration and controlling of simulations.

4.6.1. Simulation Modes

As mentioned above, two different simulation modes affect the performance analysis options available to the user.

- **Full-System Simulation (FS Mode):** gem5 simulates the entire system, including the operating system, peripheral devices, and hardware interactions in this mode.

This mode is crucial for analyzing how architectural decisions impact the overall system performance, including I/O operations, memory management, and interaction with accelerators.

- **System-Call Emulation (SE Mode):** SE mode focuses on user-level processes, making it faster for testing application performance without simulating the entire OS or hardware devices. While less detailed, this mode allows performance analysis on individual components like CPU and cache.

4.6.2. CPU Models

gem5 supports multiple types of CPU models that enable different types of performance analysis:

- **In-order models:** Simulate simple processors where instructions are executed in the order they are issued. These are useful for evaluating simpler designs or embedded systems.
- **Out-of-order models:** These are more complex and allow for performance analysis of advanced, modern CPUs where instructions are executed out of order. This helps identify performance bottlenecks like pipeline stalls, branch mispredictions, and cache misses.

Please refer to Section 4.2.1 for a list of the processor models.

4.6.3. Instrumentation and Statistics Collection

gem5 has built-in instrumentation for collecting detailed statistics about various aspects of the simulated system. Some key features include:

- **Cycle counts:** For tracking the number of cycles required for executing instructions or completing specific tasks.
- **Cache:** gem5 provides cache hit/miss rates, latency statistics, and other detailed information about cache behavior. This is vital for analyzing how different cache configurations affect performance.
- **Pipeline:** Out-of-order CPU models provide detailed insights into how instructions are scheduled, executed, and retired. This allows you to analyze pipeline efficiency and the impact of hazards, stalls, and dependencies.
- **Branch prediction:** gem5 can capture data on branch mispredictions, which is critical for analyzing speculative execution and its effect on performance.

4.6.4. Memory System Analysis

The memory hierarchy is one of the most important aspects of the performance of any real-world application. It is important to be able to simulate and compare different configurations.

- **Cache Models:** gem5 supports configurable cache hierarchies (L1, L2, L3 caches), with options to adjust cache sizes, associativity, and policies (e.g., replacement and write-back/write-through policies). This enables a detailed analysis of how the memory subsystem impacts overall performance.
- **DRAM and Interconnect Models:** For more comprehensive memory performance analysis, gem5 includes detailed models for DRAM memory and interconnects. Researchers can study how memory latency, bandwidth, and contention affect the system's behavior under various workloads.

4.6.5. Interconnect

gem5 allows users to simulate detailed interconnect architectures such as crossbars and network-on-chip (NoC) systems, which are crucial for performance analysis in multicore or many-core systems. These models allow latency, bandwidth usage, and contention analysis, giving insight into how inter-core communication affects overall performance. This topic is out of the scope of our chapter, but the following link has more information on the subject: `https://www.gem5.org/documentation/general_docs/ruby/interconnection-network/`.

4.6.6. Power and Energy Modeling

gem5 integrates an energy consumption estimate into its memory components based on a DRAMPower model. Thus, when analyzing the statistics generated by a simulation, values in Joules correspond to the estimated energy consumption of the memory modules.

McPAT (a power, area, and timing analysis tool) can be combined with gem5 to estimate power consumption, enabling a comprehensive analysis that includes both performance and energy efficiency—critical for modern processor designs that aim to optimize performance per watt.

As with DRAMPower, work has already been done linking gem5 outputs with McPAT inputs. Some scripts can be found on GitHub to do this, allowing gem5 parameters to be ported effortlessly to generate estimates of energy consumption and design area.

Similarly, gem5 integrates with DRAMPower to model and estimate the energy consumption of memory systems, aiming at collecting insights into how different memory configurations and access patterns affect overall energy usage. These integrations allow researchers to perform detailed power and energy evaluations across the entire system, from processors to memory hierarchies.

4.6.7. Performance Profiling and Debugging Tools

gem5 provides powerful tools for performance analysis, including detailed tracing and performance counters. Detailed tracing, as in Listing 4.22, allows users to track simulation events at a fine-grained level, capturing specific instruction executions, memory accesses, and interconnect traffic. This detailed information is invaluable for identifying bottlenecks and performing in-depth performance analysis. Additionally, gem5 includes performance counters that monitor key metrics in real-time, such as instructions per cycle

(IPC), memory accesses, and cache hit rates. These counters enable users to dynamically observe performance trends and system behavior during the simulation, offering insights that help optimize architectural designs. Listing 4.23 shows a sumarized and abridged example of statistics collected from our matrix multiplication implementation.

```
...
3996666: system.cpu: T0 : 0x104c0 @mm+182 : c_slli a2, 2
3996999: system.cpu: T0 : 0x104c2 @mm+184 : c_add a0, s4
3997332: system.cpu: T0 : 0x104c4 @mm+186 : c_add s8, s7
3997665: system.cpu: T0 : 0x104c6 @mm+188 : c_add s9, s3
3997998: system.cpu: T0 : 0x104c8 @mm+190 : c_add a2, s6
3998331: system.cpu: T0 : 0x104ca @mm+192 : c_li a6, 0
3998664: system.cpu: T0 : 0x104cc @mm+194 : mmv_xmu m2, zero
```

Listing 4.22: Abridged extract from matrix multiplication execution trace

```
---------- Begin Simulation Statistics ----------
simSeconds                   0.000385 # Number of seconds simulated
simTicks                    385437177 # Number of ticks simulated
hostSeconds                      0.62 # Real time elapsed on the host
hostTickRate                619899155 # Number of ticks simulated per host second
hostMemory                     650268 # Number of bytes of host memory used
simInsts                       967487 # Number of instructions simulated
simOps                         967487 # Number of ops (including micro ops) simulated
hostInstRate                  1555710 # Simulator instruction rate (inst/s)
hostOpRate                    1555693 # Simulator op (including micro ops) rate
system.cpu.numCycles          1157470 # Number of cpu cycles simulated
system.cpu.cpi               1.196330 # CPI: cycles per instruction (core level)
system.cpu.ipc               0.835889 # IPC: instructions per cycle (core level)
...
dram.bytesRead::cpu.inst      4629880 # bytes read from this memory
dram.bytesRead::cpu.data      1098880 # bytes read from this memory
dram.bytesRead::total         5728760 # bytes read from this memory
dram.bytesInstRead::cpu.inst   462988 # instructions bytes read from this memory
dram.bytesInstRead::total     4629880 # instructions bytes read from this memory
dram.bytesWritten::cpu.data    880270 # bytes written to this memory
dram.bytesWritten::total       880270 # bytes written to this memory
dram.numReads::cpu.inst       1157470 # read requests responded to by this memory
dram.numReads::cpu.data        173591 # read requests responded to by this memory
dram.numReads::total          1331061 # read requests responded to by this memory
dram.numWrites::cpu.data       125132 # write requests responded to by this memory
dram.numWrites::total          125132 # write requests responded to by this memory
```

Listing 4.23: Abridged extract from matrix instruction multiplication stats.txt

4.6.8. Python-based Configuration

In gem5, the Python-based scripting framework provides flexibility and ease of use for configuring and automating simulations. gem5's simulation environment is driven by Python scripts, allowing users to dynamically configure all aspects of the simulation. These scripts can define CPU models, memory hierarchies, cache configurations, interconnects, and peripherals.

Instead of manually adjusting numerous parameters or creating multiple static configuration files for different simulation scenarios, Python scripting allows for programmatic control. Users can easily modify simulation parameters by changing values in their Python scripts, making it easier to explore various architectural designs. For example, you can dynamically change cache sizes, adjust memory latency, or configure the number of CPU cores using Python constructs. This allows for rapid experimentation and

design space exploration, as a single script can be reused with different configurations by adjusting parameters programmatically.

One significant advantage of Python scripting is the ability to automate repetitive tasks, such as running a series of simulations with different configurations (often called parameter sweeping). Instead of manually configuring each scenario, you can write a Python script to loop over various parameter values (e.g., different cache sizes or clock frequencies) and run simulations automatically. For example, as depicted in Listing 4.24, you could write a loop in Python to run multiple simulations with different L1 cache sizes and gather performance data, greatly speeding up the process of design space exploration and optimization.

```
cache_sizes = ['32kB', '64kB', '128kB']
for size in cache_sizes:
    system.cpu.icache = L1ICache(size=size)
    system.cpu.dcache = L1DCache(size=size)
    m5.instantiate()
    print(f"Running simulation with L1 cache size: {size}")
    exit_event = m5.simulate()
    print(f"Simulation with {size} cache size ended at tick {m5.curTick()}")
```

Listing 4.24: Parameter Sweeping

Python-based scripts, like Listing 4.25, allow easy integration with external tools to process the simulation results. After running the simulation, you can use Python to analyze the generated statistics, format the output, and even automate performance data visualization using libraries like Matplotlib or Pandas. This post-processing capability helps to automate the performance evaluation and analysis pipeline, allowing users to generate plots, tables, or reports directly from the simulation results.

```
import pandas as pd
import matplotlib.pyplot as plt

# Post-process the simulation data and generate a report
results = pd.read_csv('stats.txt', sep=' ')
plt.plot(results['sim_ticks'], results['IPC'])
plt.title('IPC Over Time')
plt.xlabel('Simulation Ticks')
plt.ylabel('Instructions Per Cycle (IPC)')
plt.show()
```

Listing 4.25: Post-processing Simulation Data

4.7. Conclusion

This tutorial has explored the fundamentals of architectural simulation using gem5, emphasizing its flexibility in modeling and evaluating complex computer architectures. Using practical examples, we demonstrated how to extend a RISC-V architecture with new instructions for matrix multiplication, a key operation in AI workloads.. By following this guide, readers should now be equipped to experiment with architectural models and make informed design decisions.

References

[Binkert et al. 2011] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. (2011). The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7.

[Binkert et al. 2006] Binkert, N., Dreslinski, R., Hsu, L., Lim, K., Saidi, A., and Reinhardt, S. (2006). The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60.

[Black et al. 2010] Black, G., Binkert, N., Reinhardt, S. K., and Saidi, A. (2010). *Modular ISA-Independent Full-System Simulation*, pages 65–83. Springer US, Boston, MA.

[gem5 2022] gem5 (2022). gem5: Execution Basics. `https://www.gem5.org/documentation/general_docs/cpu_models/execution_basics`. [Accessed 23-09-2024].

[Gem5 Project 2024] Gem5 Project (2024). Getting started with gem5. `https://www.gem5.org/getting_started/`. Accessed: 2024-10-02.

[Lowe-Power et al. 2020] Lowe-Power, J., Ahmad, A. M., Akram, A., Alian, M., Amslinger, R., Andreozzi, M., Armejach, A., Asmussen, N., Bharadwaj, S., Black, G., Bloom, G., Bruce, B. R., Carvalho, D. R., Castrillón, J., Chen, L., Derumigny, N., Diestelhorst, S., Elsasser, W., Fariborz, M., Farahani, A. F., Fotouhi, P., Gambord, R., Gandhi, J., Gope, D., Grass, T., Hanindhito, B., Hansson, A., Haria, S., Harris, A., Hayes, T., Herrera, A., Horsnell, M., Jafri, S. A. R., Jagtap, R., Jang, H., Jeyapaul, R., Jones, T. M., Jung, M., Kannoth, S., Khaleghzadeh, H., Kodama, Y., Krishna, T., Marinelli, T., Menard, C., Mondelli, A., Mück, T., Naji, O., Nathella, K., Nguyen, H., Nikoleris, N., Olson, L. E., Orr, M. S., Pham, B., Prieto, P., Reddy, T., Roelke, A., Samani, M., Sandberg, A., Setoain, J., Shingarov, B., Sinclair, M. D., Ta, T., Thakur, R., Travaglini, G., Upton, M., Vaish, N., Vougioukas, I., Wang, Z., Wehn, N., Weis, C., Wood, D. A., Yoon, H., and Zulian, É. F. (2020). The gem5 simulator: Version 20.0+. *CoRR*, abs/2007.03152.

[Martin et al. 2005] Martin, M. M. K., Sorin, D. J., Beckmann, B. M., Marty, M. R., Xu, M., Alameldeen, A. R., Moore, K. E., Hill, M. D., and Wood, D. A. (2005). Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99.

[Waterman et al. 2014] Waterman, A., Lee, Y., Patterson, D. A., and Asanovic, K. (2014). The risc-v instruction set manual, volume i: User-level isa, version 2.0. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54*, page 4.

Capítulo 5

Introdução à Computação Quântica com IBM/Qiskit

Calebe P. Bianchini, Giancarlo P. Gamberi, Ryan M. A. Santos

Abstract

This chapter demonstrate how to develop algorithms for the architecture of a Quantum Computer using the IBM/Qiskit development kit. Consequently, we aim to solve classical computing problems using this new architecture. We understand that writing algorithms and programs in this new paradigm presents a challenge, and therefore, key topics in Quantum Computing will be defined and presented, such as its architecture, entanglement, logic gates, the circuits used, and how they behave differently from those in a traditional architecture. Through demonstrations, we will show how this emerging technology offers an interesting degree of parallelism and significant computational acceleration compared to classical architectures, helping the reader better prepare to program on a quantum computer.

Resumo

O objetivo deste capítulo é mostrar como desenvolver algoritmos para a arquitetura de um Computador Quântico usando o kit de desenvolvimento IBM/Qiskit. Consequentemente, pretendemos resolver problemas clássicos da computação tradicional nessa nova arquitetura. Sabemos que escrever algoritmos e programas nesse novo paradigma é um desafio e, por isso, serão definidos e apresentados assuntos importantes da Computação Quântica como sua arquitetura, o emaranhamento, as portas lógicas, os circuitos utilizados e como eles se comportam de maneira diferente em relação a uma arquitetura tradicional. Por meio de demonstrações, será apresentado como essa tecnologia emergente fornece um grau interessante de paralelismo e aceleração computacional significativos em relação às arquiteturas clássicas, permitindo que o leitor se prepare melhor para programar em um computador quântico.

5.1. Introdução

As primeiras propostas de Computadores Quânticos, operantes de acordo com a mecânica quântica, tiveram motivações guiadas pelo estudo dos sistemas físicos e simulações quânticas [Feynman, 1982]. Rapidamente, então, houve diversos estudos na literatura da Ciência da Computação que buscam entender e projetar Computadores Quânticos, o que se mostrou efetivo ao longo do tempo pela popularidade alcançada por estes sistemas: [Deutsch, 1985] provou que estes seriam Turing-completo, isto é, dotados da capacidade de computação universal; [Shor, 1994] desenvolveu um algoritmo puramente quântico para fatoração de números compostos; [Grover, 1996] apresentou uma busca eficiente em uma base de dados desordenada de complexidade $O(\sqrt{N})$; para citar alguns dos grandes avanços na área.

Todos estes exemplos utilizam das vantagens da informação quântica, que é a informação em estado de processamento usando sistemas da mecânica quântica [Nielsen and Chuang, 2010]. Obviamente, elas se diferenciam fundamentalmente da computação clássica tradicional. A unidade da informação assume as leis da mecânica quântica, como emaranhamento e superposição quântica [Shor, 1994, Preskill, 2012]. Sistemas quânticos então fazem uso de técnicas estocásticas que encontram atalhos em meio a todos os caminhos que um algoritmo pode seguir. Por conta disso, sistemas construídos a partir dos *qubits* já são considerados mais poderosos para a solução de certas tarefas do que computadores clássicos, como busca, simulações físicas complexas, modelagem de moléculas e materiais, dentre diversos outros problemas da ciência [Silva, 2018a, Preskill, 2012].

Outro fator que corrobora para a popularização da computação quântica são os desenvolvimentos no campo da engenharia de computadores quânticos, os quais estão cada vez mais acessíveis. Ferramentas como o SDK *Qiskit* da IBM, *pyQuil* da Google, Q# da Microsoft e *myQLM* da Eviden permitem que sejam feitos estudos sobre os comportamentos de um circuito e computador quântico sem um investimento na tecnologia. O que se comprova pela quantidade de trabalhos realizados com a tecnologia, com 33.230 no tópico de '*quantum computing*' no *Web of Science* [Valdez and Melin, 2022], e a ideia de que este paradigma superará Moore [Arute et al., 2019].

Os algoritmos quânticos se caracterizam por operações unitárias sobre os *qubits*, de forma a manipular os estados de superposição [Nielsen and Chuang, 2010]. Estas operações são empregadas nos circuitos por portas lógicas que agem sobre os *qubits* desejados, similarmente às portas lógicas da computação clássica [Silva, 2018a]. Contudo, a saída de um algoritmo normalmente se difere de acordo com a entrada e, por isso, há a implementação de oráculos, também referidos como "*caixas-preta*". Eles adaptam o circuito à entrada recebida, de forma a adquirir a saída esperada por um algoritmo [Silva, 2018a].

A proposta deste deste minicurso é apresentar os princípios mais básicos desse novo modelo de computação, descrevendo as principais características de um Computador Quântico, bem como descrevendo como construir algoritmos quânticos usando o *kit* de desenvolvimento IBM/Qiskit [Javadi-Abhari et al., 2024, Gamberi and Bianchini, 2023]. Ao longo do minicurso, resolveremos problemas clássicos da computação tradicional nesse nova arquitetura.

5.2. Introdução a Mecânica Quântica

A Mecânica Quântica é uma teoria fundamental da física que descreve o comportamento da matéria e da energia em níveis atômicos e subatômicos [Yanofsky and Mannucci, 2008]. Diferente da física clássica, que trata do mundo macroscópico, a mecânica quântica introduz o conceito de *quantização*. Isso significa que propriedades como energia, momento e momento angular existem em pacotes discretos chamados *quanta*. Por exemplo, um átomo só absorve ou emite energia em quantidades específicas, quantizadas, resultando nos níveis de energia discretos observados em espectros atômicos. Essa quantização contrasta com a física clássica, onde essas propriedades variam continuamente [Jorio and Frossard, 2024].

Na mecânica clássica, o estado de um sistema, como a posição e o momento de uma partícula, pode ser determinado com precisão. No entanto, a mecânica quântica descreve o estado de um sistema usando a *função de onda*, denotada por ψ. Essa função matemática codifica as probabilidades de vários resultados possíveis após uma medição, refletindo a incerteza inerente aos sistemas quânticos. Essa abordagem probabilística é um afastamento da visão determinista da física clássica, onde a evolução futura de um sistema poderia ser prevista com certeza. No final, esta função de onda não especifica a localização exata da partícula, mas fornece uma distribuição de probabilidade para encontrar a partícula em várias posições (quando medida). O quadrado da magnitude da função de onda, $|\psi(\mathbf{r},t)|^2$, descreve a probabilidade de encontrar o sistema na posição $\mathbf{r}$ e no tempo t.

A evolução de um sistema quântico ao longo do tempo é regida pela *equação de Schrödinger*. Essa equação, um dos pilares da mecânica quântica, descreve como a função de onda muda sob diferentes potenciais. Ao resolver a equação de Schrödinger, é possível determinar os níveis de energia permitidos e as funções de onda de um sistema quântico. Em sistemas com condições estáveis, costuma-se usar a equação de Schrödinger independente do tempo para identificar os estados estacionários do sistema.

A mecânica quântica também é conhecida devido a três conceitos que se destacam [Shafique et al., 2024]:

- Superposição: um sistema quântico pode existir em uma superposição de múltiplos estados simultaneamente, ao contrário dos sistemas clássicos, que só podem estar em um estado de cada vez.
- Emaranhamento: ele descreve uma forte correlação entre dois ou mais sistemas quânticos, mesmo quando separados por grandes distâncias.
- Decoerência: é um processo pelo qual um sistema quântico perde sua superposição e outras propriedades quânticas devido a interações com seu ambiente.

5.2.1. Superposição Quântica

A *superposição quântica* [Yanofsky and Mannucci, 2008, Microsoft Quantum, ndb] é outro princípio fundamental da mecânica quântica, permitindo que sistemas quânticos existam em múltiplos estados ao mesmo tempo. Enquanto um sistema clássico, como uma

moeda, está em um estado definido (cara ou coroa), um *qubit* pode existir em uma combinação de ambos os estados simultaneamente [Jorio and Frossard, 2024].

O estado geral de um *qubit* é expresso na Equação 1.

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{1}$$

Nesta equação, α e β são números complexos chamados *amplitudes de probabilidade*, e os quadrados dos módulos dessas amplitudes representam as probabilidades de encontrar o *qubit* em cada estado após a medição. A soma dessas probabilidades deve ser igual a 1, ou seja, $|\alpha|^2 + |\beta|^2 = 1$.

O conceito de superposição é crucial na computação quântica. Computadores clássicos processam uma computação de cada vez, pois *bits* clássicos só podem estar em um dos dois estados possíveis, 0 ou 1. No entanto, computadores quânticos, aproveitando a superposição, podem explorar várias possibilidades simultaneamente ao processar todos os possíveis estados dos *qubits* de uma vez, permitindo uma forma de processamento paralelo conhecida como paralelismo quântico.

O equilíbrio delicado da superposição, no entanto, é interrompido quando há uma medição ou observação. Semelhante ao exemplo de uma moeda que eventualmente cai de um lado ou do outro, o ato de observação força o *qubit* a "colapsar" em um de seus estados base (0 ou 1), de acordo com uma probabilidade determinada pela sua superposição. Esse colapso da superposição é uma diferença fundamental entre sistemas quânticos e clássicos.

A superposição não é apenas um conceito teórico; é uma característica fundamental dos sistemas quânticos. Sua capacidade de permitir o paralelismo quântico forma a base do potencial da computação quântica e de outras tecnologias quânticas. Ela sustenta muitos fenômenos e aplicações quânticas, incluindo algoritmos quânticos, simulações quânticas de comportamento molecular e de materiais, criptografia quântica para comunicações seguras e sensoriamento quântico para medições ultra-sensíveis.

Além disso, a superposição não se limita a *qubits* individuais, mas também pode se estender a sistemas com múltiplos *qubits*. Ao emaranhar vários *qubits* e manipulá-los em superposição, os computadores quânticos podem realizar operações em vários estados possíveis simultaneamente. Essa capacidade é fundamental para o potencial poder computacional dos computadores quânticos.

5.2.2. Emaranhamento Quântico

O *emaranhamento quântico* [Yanofsky and Mannucci, 2008, Microsoft Quantum, nda] é um dos fenômenos mais intrigantes da mecânica quântica, onde duas ou mais partículas tornam-se correlacionadas de tal maneira que seus estados estão entrelaçados, mesmo quando separadas por grandes distâncias. Uma vez emaranhadas, a medição do estado de uma partícula determina instantaneamente o estado da outra, independentemente da distância entre elas. Isso desafia a física clássica e representa uma conexão não local entre as partículas.

Para entender o emaranhamento, é crucial lembrar como esses sistemas são des-

critos. Diferente dos sistemas clássicos, que possuem propriedades definidas, os sistemas quânticos são descritos por uma função de onda. Essa função de onda não determina um único resultado para uma medição, mas fornece uma distribuição de probabilidade sobre todos os resultados possíveis.

Quando dois sistemas quânticos tornam-se entrelaçados, suas funções de onda individuais não podem mais ser escritas separadamente. Em vez disso, eles são descritos por uma única função de onda combinada que abrange ambos os sistemas. É essa função de onda compartilhada que resulta na forte correlação entre os sistemas entrelaçados.

Considere, por exemplo, dois *qubits* inicialmente preparados em um estado emaranhado representado pelo *estado de Bell*, conforme mostra a Equação 2.

$$\left|\Phi^{+}\right\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \tag{2}$$

Essa equação ilustra que os dois *qubits* existem em uma superposição em que ambos estão simultaneamente no estado $|0\rangle$ e ambos estão no estado $|1\rangle$. Se uma medição em um *qubit* revelar que ele está no estado $|0\rangle$, o estado do outro *qubit* colapsa instantaneamente para $|0\rangle$ também, apesar da separação física entre eles. Da mesma forma, se o primeiro *qubit* for medido no estado $|1\rangle$, o estado do segundo *qubit* colapsa instantaneamente para $|1\rangle$. Isso destaca a natureza não local do emaranhamento, onde a correlação entre sistemas emaranhados transcende as distâncias espaciais.

5.2.3. Ruído Quântico

O *ruído quântico* [Yanofsky and Mannucci, 2008, Shafique et al., 2024] representa a incerteza e as flutuações inerentes presentes nos sistemas quânticos, devido à natureza probabilística fundamental da mecânica quântica. Diferentemente do ruído clássico, frequentemente atribuído a distúrbios externos, o ruído quântico é intrínseco e persiste mesmo ambiente extremamente controlados. Ele apresenta desafios significativos para a computação quântica, especialmente ao afetar os resultados dos circuitos quânticos. Uma das principais formas pelas quais o ruído quântico se manifesta é por meio do ruído da própria medição.

No domínio quântico, o ato de medir um sistema quântico inevitavelmente perturba o sistema. Quando uma medição é realizada em um *qubit* em superposição, a função de onda colapsa, forçando o *qubit* a assumir um estado definido, seja 0 ou 1. Esse colapso introduz incerteza, pois o resultado específico da medição é probabilístico e determinado pelas amplitudes de probabilidade associadas a cada estado. O próprio processo de obtenção de informações sobre um sistema quântico por meio da medição introduz ruído.

A decoerência, outra consequência significativa do ruído quântico, surge da interação inevitável entre um sistema quântico e seu ambiente. Essa interação leva ao emaranhamento entre os *qubits* do sistema e o ambiente externo, resultando na perda de propriedades quânticas como superposição e emaranhamento. A decoerência perturba o equilíbrio delicado necessário para a computação quântica, causando erros e afetando a precisão dos algoritmos quânticos.

Para ilustrar, imagine um registrador quântico composto por elétrons, com seus

estados de *spin* representando *qubits*. Quando isolados, os elétrons podem ser cuidadosamente inicializados em um estado puro, como o *spin* para cima, representando um valor específico de *qubit*. No entanto, quando esse registrador interage com o ambiente, os elétrons se entrelaçam com inúmeras outras partículas, cada uma em um estado desconhecido. Esse entrelaçamento destrói a pureza do estado do registrador, transformando-o em um estado misto, onde as relações de fase definidas entre os *qubits*, cruciais para a interferência quântica, são perdidas. Só é possível recuperar essa informação de fase perdida rastreando e medindo o estado de cada partícula ambiental que interagiu com o registrador – uma tarefa praticamente impossível.

A decoerência representa um obstáculo fundamental para a construção de computadores quânticos práticos. O desafio reside em encontrar o equilíbrio delicado entre isolar o sistema quântico para minimizar a decoerência, e ainda assim ser capaz de interagir com ele para realizar operações e medições. Superar a decoerência requer abordagens inovadoras, incluindo o desenvolvimento de operações de portas quânticas mais rápidas para superar a decoerência e a implementação de técnicas de computação quântica tolerante a falhas, como códigos de correção de erros quânticos, para mitigar o impacto dos erros.

5.3. Computação Quântica

A computação quântica representa uma mudança fundamental na forma como a informação é processada e os cálculos são realizados [Yanofsky and Mannucci, 2008]. Diferentemente da computação clássica, que utiliza *bits* como a unidade fundamental de informação, a computação quântica aproveita os princípios da mecânica quântica para processar informações de uma maneira essencialmente diferente [Nayak et al., 2024]. No centro desse processamento está o *qubit*, o análogo quântico do *bit*. Enquanto um *bit* clássico pode existir em apenas um de dois estados, 0 ou 1, um *qubit* pode existir em uma superposição, representando simultaneamente tanto 0 quanto 1 com certas probabilidades. Essa capacidade de ocupar múltiplos estados simultaneamente oferece aos computadores quânticos o potencial de explorar vastos espaços computacionais, permitindo que eles resolvam problemas que são intratáveis até mesmo para os mais poderosos computadores clássicos.

O poder da computação quântica une a superposição e o emaranhamento. Assim, a capacidade probabilística de um sistema quântico e o entrelaçamento entre suas partes (ou subsistemas) desbloqueia o potencial para o processamento paralelo em uma escala sem precedentes. À medida que múltiplos *qubits* são manipulados dentro de um computador quântico, eles exploram efetivamente inúmeras possibilidades computacionais simultaneamente, acelerando significativamente certos tipos de cálculos. Esse paralelismo forma a base para os algoritmos quânticos, permitindo que eles superem os algoritmos clássicos em problemas específicos.

Algoritmos quânticos, projetados especificamente para aproveitar a superposição e o emaranhamento, oferecem o potencial de acelerações exponenciais em certas tarefas computacionais. O algoritmo de Shor, por exemplo, pode fatorar grandes números exponencialmente mais rápido do que qualquer algoritmo clássico conhecido, representando um desafio significativo para a criptografia moderna, que se baseia na dificuldade de fa-

torar grandes números. Da mesma forma, o algoritmo de Grover oferece uma aceleração quadrática para a busca em bases de dados não ordenadas, demonstrando o potencial da computação quântica para revolucionar áreas como análise de dados e otimização.

A implementação de computadores quânticos apresenta desafios significativos, principalmente devido à fragilidade dos estados quânticos e à sua suscetibilidade ao ruído. Como apresentado anteriormente, o ruído quântico, particularmente a decoerência resultante das interações entre o sistema quântico e seu ambiente, perturba a delicada superposição e o emaranhamento, levando a erros computacionais. A busca pela computação quântica tolerante a falhas, que visa mitigar o impacto do ruído por meio de técnicas como a correção de erros quânticos, continua sendo um foco central no campo. Superar esses obstáculos é essencial para escalar os computadores quânticos de forma eficaz para lidar com problemas do mundo real.

Apesar desses desafios, o campo da computação quântica continua avançando rapidamente. O surgimento dos computadores quânticos ruidosos de escala intermediária (*Noisy Intermediate-Scale Quantum*, ou NISQ), embora ainda limitados em tamanho e sujeitos a erros, abriu novas possibilidades para explorar o potencial da computação quântica. Esses dispositivos quânticos de estágio inicial demonstraram a capacidade de realizar cálculos que desafiam as capacidades dos computadores clássicos. À medida que a pesquisa avança, abordar as limitações dos dispositivos NISQ por meio de técnicas de correção de erros e tempos de coerência de *qubit* aprimorados abre caminho para computadores quânticos mais poderosos e confiáveis. O impacto potencial da computação quântica abrange diversos campos, desde criptografia e ciência dos materiais até descoberta de medicamentos e inteligência artificial. Embora a jornada para realizar todo o potencial da computação quântica esteja em andamento, o poder transformador que ela detém promete remodelar nosso cenário tecnológico nos próximos anos.

5.3.1. Qubit vs. Bit

O *qubit* [Yanofsky and Mannucci, 2008, Shafique et al., 2024] é o bloco fundamental da computação quântica, análogo ao bit clássico, mas com a capacidade de existir em uma superposição de dois estados simultaneamente. Essa superposição é uma das propriedades mais intrigantes do *qubit* e permite que ele não esteja limitado apenas aos estados binários fixos, como acontece na computação clássica.

Uma forma importante de representar o estado de um *qubit* é através da notação *bra-ket*. Essa notação é amplamente utilizada para descrever estados quânticos e suas interações.

Um *ket*, denotado como $|\psi\rangle$, representa um vetor coluna que descreve um estado quântico. Esse estado pode corresponder a um estado específico de um *qubit*, a uma superposição de estados ou até mesmo a um estado emaranhado envolvendo múltiplos *qubits*. A notação $|\psi\rangle$ encapsula todas as informações necessárias sobre o estado do sistema.

O *bra*, por sua vez, é denotado como $\langle\psi|$ e representa a transposta conjugada do *ket* $|\psi\rangle$. Matematicamente, isso significa tomar a transposta do vetor (convertendo-o de coluna para linha) e, em seguida, aplicar o conjugado complexo a cada elemento.

Por exemplo, se o *ket* $|\psi\rangle$ for representado como $[3, 1-2i]^T$, o *bra* correspondente seria $\langle\psi| = [3, 1+2i]$. Dessa forma, o *bra* pode ser interpretado como um vetor linha.

Quando um *bra* e um *ket* são combinados, formando uma expressão do tipo $\langle\psi|\phi\rangle$, eles representam o produto interno entre os dois estados quânticos $|\psi\rangle$ e $|\phi\rangle$. Em espaços vetoriais de dimensão finita, o produto interno é calculado como o produto escalar entre os vetores *bra* e *ket*, resultando em um valor escalar. Esse valor, tipicamente um número complexo, é conhecido como a amplitude de transição entre os dois estados $|\psi\rangle$ e $|\phi\rangle$. A combinação "*bra-ket*" muitas vezes é referida de forma abreviada como "*braket*".

Além de servir para descrever estados, a notação *bra-ket* é amplamente utilizada em algoritmos quânticos, que consistem em operações sequenciais sobre *qubits*. Essas operações, chamadas de portas lógicas quânticas, modificam o estado dos *qubits* de maneira controlada e são a base de qualquer processamento em um computador quântico.

Os estados $|0\rangle$ e $|1\rangle$ são definidos como os estados de base computacional para um único *qubit*. Eles são análogos aos estados 0 e 1 de um bit clássico. O estado $|0\rangle$, frequentemente chamado de "*ket 0*", indica um *qubit* no estado "desligado" ou "*spin-down*". Ele é representado pela matriz coluna, conforme descreve a Equação 3. O valor no topo da matriz representa a amplitude de probabilidade de o *qubit* estar no estado *desligado*, enquanto o valor na parte inferior representa a amplitude de probabilidade de o *qubit* estar no estado "ligado" ou "*spin-up*".

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{3}$$

Já o estado $|1\rangle$, conhecido como "*ket 1*", representa um *qubit* no estado "ligado" ou "*spin-up*". Sua representação matricial pode ser vista na Equação 4. De forma similar à representação de $|0\rangle$, a entrada superior corresponde à amplitude de probabilidade do estado *desligado*, enquanto a entrada inferior corresponde à amplitude de probabilidade do estado *ligado*.

$$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{4}$$

Esses estados de base são fundamentais para representar e manipular *qubits* em cálculos quânticos. Uma das formas de utilização é na construção de *qubits* arbitrários. Conforme descrito na Equação 1, qualquer *qubit* único pode ser expresso como uma combinação linear de $|0\rangle$ e $|1\rangle$.

Por exemplo, o estado do *qubit* representado na Equação 5 pode ser expresso como uma combinação linear de $|0\rangle$ e $|1\rangle$, conforme mostra a Equação 6.

$$V = \begin{bmatrix} 3+2i \\ 4-2i \end{bmatrix} \tag{5}$$

$$V = (3+2i)\,|0\rangle + (4-2i)\,|1\rangle \tag{6}$$

Outra aplicação importante é na formação de estados de múltiplos *qubits*, que utilizam os estados de base para construir estados de sistemas quânticos utilizando o produto tensorial. Por exemplo, um sistema de dois *qubits* pode existir em quatro estados possíveis, derivados do produto tensorial dos estados de *qubits* individuais, conforme apresenta a Equação 7.

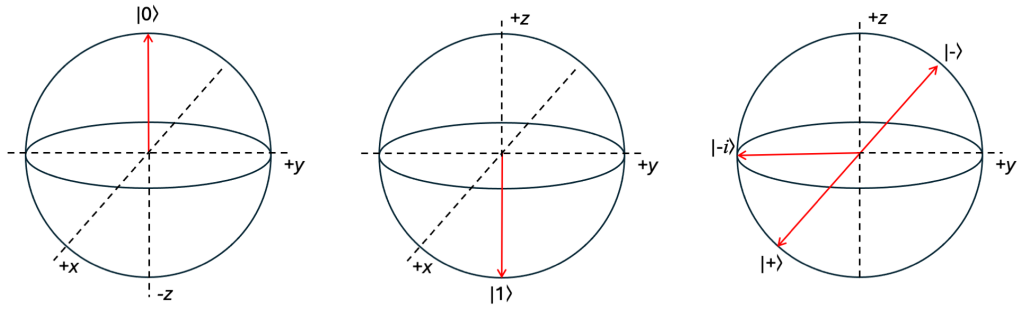
$$|00\rangle = |0\rangle \otimes |0\rangle, \quad |01\rangle = |0\rangle \otimes |1\rangle, \quad |10\rangle = |1\rangle \otimes |0\rangle, \quad |11\rangle = |1\rangle \otimes |1\rangle \tag{7}$$

Esses estados de base formam o alicerce para representar estados mais complexos e emaranhados em sistemas de múltiplos qubits.

Por fim, os estados $|0\rangle$ e $|1\rangle$ são cruciais para as operações de portas quânticas. As portas quânticas, representadas como matrizes, atuam sobre esses estados de *qubit* para realizar transformações, da mesma forma que as portas lógicas operam sobre *bits* clássicos.

Outra forma importante de compreender os estados de um qubit é através da *Esfera de Bloch*, uma representação geométrica que permite visualizar o estado quântico de um único *qubit* [Yanofsky and Mannucci, 2008, ShareTechNote, nd]. Todos os estados puros de um *qubit* podem ser representados como pontos na superfície dessa esfera.

Na Esfera de Bloch, os estados base $|0\rangle$ e $|1\rangle$ são visualizados nos polos norte e sul da esfera, respectivamente, conforme pode ser observado na Figura 5.1a e na Figura 5.1b. Todos os outros pontos na superfície da esfera correspondem a superposições desses dois estados.



(a) *Qubit* básico $|0\rangle$ (b) *Qubit* básico $|1\rangle$ (c) *Qubits* diversos

Figura 5.1: Estados de base computacional quânticos

Por exemplo, o estado $|+\rangle$, que é uma superposição equitativa de $|0\rangle$ e $|1\rangle$, pode ser expresso conforme mostra a Equação 8.

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \tag{8}$$

Este estado é representado como um ponto no equador da esfera, indicando uma probabilidade de 50% de o *qubit* colapsar para o estado $|0\rangle$ e 50% para o estado $|1\rangle$.

A Esfera de Bloch também é uma ferramenta valiosa para visualizar operações sobre *qubits*, como rotações e mudanças de fase. Por exemplo, uma rotação do estado quântico ao redor do eixo z da esfera altera a fase do *qubit*, sem mudar a probabilidade de medir o *qubit* nos estados $|0\rangle$ ou $|1\rangle$. Estados ortogonais, como $|0\rangle$ e $|1\rangle$, são representados por pontos opostos na esfera.

Por exemplo, o estado $|+\rangle$, que, como mencionado, está localizado no equador da Esfera de Bloch ao longo do eixo x positivo, tem como estado ortogonal outro estado relevante, que é o estado $|-\rangle$, conforme expresso na Equação 9.

$$|-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} \tag{9}$$

Este outro estado também se encontra no equador, mas ao longo do eixo x negativo. Estes dois estados, $|+\rangle$ e $|-\rangle$ podem ser vistos na Figura 5.1c.

Considerando o estado estado $|+\rangle$, e se aplicarmos uma porta quântica que rotacione o *qubit* em 90 graus em torno do eixo y, ele seria transformado no estado $|1\rangle$, ilustrando a mudança geométrica na superfície da esfera. Imagine a mudança de estado gerado por esta porta a partir da Figura 5.1c.

Por fim, os estados que estão localizados no equador ao longo do eixo z também podem ser deduzidos a partir da Figura 5.1c, como o estado $|-i\rangle$.

Embora a Esfera de Bloch seja extremamente útil para visualizar os estados de um *qubit* único, ela apresenta limitações para sistemas quânticos mais complexos, como os estados emaranhados, que envolvem interações entre múltiplos *qubits*. A Esfera de Bloch é eficaz para representar estados de *qubits* individuais, mas a complexidade de sistemas envolvendo múltiplos *qubits* e suas interdependências requer representações mais sofisticadas.

5.3.2. Operações Quânticas e Portas Lógicas

As operações quânticas estão no centro da computação quântica, servindo como transformações matemáticas que atuam sobre estados quânticos [Yanofsky and Mannucci, 2008, Shafique et al., 2024]. Essas operações são inerentemente reversíveis devido à natureza da mecânica quântica e são representadas por matrizes unitárias. A reversibilidade garante que, dado o resultado de uma operação, a entrada possa sempre ser determinada, destacando a natureza determinística da evolução quântica fora dos processos de medição.

As portas quânticas, os blocos de construção fundamentais dos circuitos quânticos, desempenham um papel crucial na manipulação de *qubits* e na transformação da informação quântica. Assim como as portas lógicas clássicas na computação tradicional, as portas quânticas também operam por meio de matrizes unitárias. Vários tipos de portas quânticas servem a funções distintas.

As portas quânticas são instâncias específicas de operações quânticas e são as ferramentas práticas usadas para implementar as transformações teóricas descritas por essas operações. Um circuito quântico é, essencialmente, uma sequência de portas quânticas

aplicadas a *qubits*, representando uma série controlada de operações quânticas que transformam estados quânticos de maneira precisa e estruturada. Essa manipulação de *qubits* forma a base dos algoritmos quânticos e possibilita a computação quântica.

Um dos conceitos mais significativos na computação quântica é a ideia de universalidade. Assim como conjuntos de portas lógicas clássicas podem simular qualquer computação clássica, existem conjuntos universais de portas quânticas que podem aproximar qualquer porta quântica arbitrária com alta precisão. Apesar dessa universalidade, a computação quântica enfrenta algumas limitações inerentes. Todas as operações devem ser reversíveis, e o Teorema da Não-Clonagem proíbe a cópia exata de um estado quântico arbitrário [Jorio and Frossard, 2024].

Um porta quântica importante é a porta de Identidade (I). Ela é um exemplo de porta quântica de um único *qubit*. Sua função é preservar o estado do *qubit* sobre o qual atua, sem alteração. Embora aparentemente trivial, a porta de Identidade possui uma importância significativa em estruturas teóricas e implementações práticas de algoritmos quânticos.

Esta porta é representada por uma matriz identidade 2×2 caracterizada por 1's ao longo da diagonal principal e 0's nos outros lugares, garantindo a manutenção do estado do *qubit* de entrada, conforme mostra a Equação 10.

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{10}$$

Considere, por exemplo, um *qubit* em um estado de superposição arbitrária $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, onde α e β são amplitudes de probabilidade. A aplicação da porta Identidade mantém o estado deste *qubit*, deixando o estado de superposição inalterado, conforme pode ser visto na Equação 11.

$$I|\psi\rangle = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \alpha|0\rangle + \beta|1\rangle = |\psi\rangle \tag{11}$$

Embora a porta de Identidade não induza mudanças observáveis no estado de um *qubit*, ela atua como o elemento identidade para a multiplicação de portas, uma propriedade que é crucial para descrever e analisar circuitos quânticos complexos. Embora não manipule diretamente os estados dos *qubits*, a porta de Identidade facilita a construção de circuitos ao fornecer um espaço reservado para potenciais operações futuras ou manter alinhamentos específicos de portas. Além disso, manter a coerência do qubit é um desafio significativo na computação quântica prática. A porta de Identidade contribui também para esquemas de correção de erros ao preservar o estado do qubit, ajudando a mitigar o impacto do ruído e da decoerência.

Outras portas quânticas que operam sobre um único *qubit* são as portas Pauli, representadas pelas matrizes X, Y e Z. Semelhantes a porta NOT clássica, elas realizam rotações específicas nos *qubits*, e que podem ser visualizados na esfera de Bloch.

A porta Pauli-X, muitas vezes referido como porta de inversão de *bit*, inverte o estado de um *qubit*, transformando $|0\rangle$ em $|1\rangle$ e vice-versa. Sua representação matricial

pode ser vista na Equação 12.

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{12}$$

A ação sobre os estados os estados base $|0\rangle$ e $|1\rangle$ pode ser vista nas Equação 13 e na Equação 14, respectivamente.

$$X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle \tag{13}$$

$$X|1\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle \tag{14}$$

Considerando a esfera de Bloch, a porta Pauli-X realiza uma rotação de 180 graus do *qubit* em torno do eixo X.

Uma outra porta, a Pauli-Y, aplica uma operação combinada de inversão e de inversão de fase a um *qubit*. Ela realiza uma rotação de 180 graus em torno do eixo Y do *qubit* na esfera de Bloch. Sua representação matricial pode ser vista na Equação 15, onde também se encontra sua aplicação sobre os estados base $|0\rangle$ e $|1\rangle$, respectivamente.

$$\begin{aligned} Y &= \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \\ Y|0\rangle &= i|1\rangle \\ Y|1\rangle &= -i|0\rangle \end{aligned} \tag{15}$$

A última porta Pauli, chamada Pauli-Z, é conhecido como o portão de inversão de fase. Ela altera a fase de um *qubit* sem afetar sua amplitude de probabilidade. Na esfera de Bloch, a porta Pauli-Z rotaciona o estado do *qubit* em 180 graus em torno do eixo Z. Sua representação matricial pode ser vista na Equação 16, onde também se encontra sua aplicação sobre os estados base $|0\rangle$ e $|1\rangle$, respectivamente.

$$\begin{aligned} Z &= \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \\ Z|0\rangle &= |0\rangle \\ Z|1\rangle &= -|1\rangle \end{aligned} \tag{16}$$

Dentre diversas características importantes, as portas Pauli desempenham um papel crucial nas técnicas de correção de erros quânticos. Ao entender como esses portões afetam os estados dos *qubits*, podemos projetar esquemas para detectar e corrigir erros que surgem devido ao ruído e à decoerência, possibilitando o desenvolvimento de circuitos mais robustos.

Uma outra porta que atua em um único *qubit* é chamada de Hadamard. Ela incorporando a essência da mecânica quântica através de sua capacidade de gerar superposições.

A porta Hadamard (H) transforma o estado de um *qubit* através de uma matriz unitária 2×2, conforme Equação 17.

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{17}$$

Aplicar uma porta Hadamard a qualquer estado base $|0\rangle$ ou $|1\rangle$ produz uma superposição, mostrando efetivamente o fenômeno quântico onde o *qubit* existe em uma combinação linear de ambos os estados simultaneamente. A Equação 18 mostra a ação desta porta sobre os estados $|0\rangle$ e $|1\rangle$, respectivamente.

$$\begin{aligned} H|0\rangle &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ H|1\rangle &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \end{aligned} \tag{18}$$

A interpretação geométrica da ação da porta Hadamard pode ser visualizada usando a esfera de Bloch, que representa os estados de um único *qubit*. Aplicar o portão de Hadamard a um *qubit* significa rotacionar seu estado 180° ao redor de um eixo que está a 45° entre os eixos X e Z. Este eixo, muitas vezes denominado de "eixo de Hadamard", destaca a transformação única do portão, distinta das rotações induzidas pelos portões Pauli.

Além disso, a capacidade da porta Hadamard de gerar superposições o torna indispensável em vários algoritmos quânticos [Yanofsky and Mannucci, 2008] [Jorio and Frossard, 2024, Shafique et al., 2024]:

- aplicar o portão de Hadamard a um qubit inicializado no estado $|0\rangle$ produz uma superposição igual, efetivamente criando um "lançamento de moeda"quântico. Medir este qubit resulta em $|0\rangle$ ou $|1\rangle$ com igual probabilidade, fornecendo uma fonte de verdadeira aleatoriedade inalcançável em sistemas clássicos.
- muitos algoritmos quânticos dependem fortemente da porta Hadamard. Ao criar superposições de estados de entrada, esses algoritmos podem explorar múltiplos caminhos computacionais simultaneamente, levando a acelerações exponenciais em relação aos seus equivalentes clássicos.
- a porta Hadamard desempenha um papel crucial no protocolo para teletransporte quântico. Sua capacidade de transformar entre a base computacional e a base de Bell (um conjunto de estados maximizadamente emaranhados) possibilita a transferência de informações quânticas entre partes distantes.

A porta Hadamard, apesar de sua forma matemática aparentemente simples, incorpora a *estranheza* e o *poder* da mecânica quântica. Sua capacidade de gerar superposições e realizar rotações específicas na esfera de Bloch o destaca na computação quântica. Ao entender suas propriedades e aplicações, desbloqueamos um conjunto de novas possibilidades computacionais a serem aplicadas em diversos cenários, que vão desde criptografia e ciência dos materiais até descoberta de medicamentos e inteligência artificial.

As portas Identidade (I), Pauli (X, Y, Z) e Hadamard (H) discutidas formam os blocos de construção para criar circuitos quânticos mais complexos que manipulam os estados de múltiplos *qubits*. Esses circuitos, visualizados como sequências de portas aplicadas aos *qubits*, representam algoritmos quânticos. Esses circuitos quânticos utilizam essas portas quânticas para explorar os princípios de superposição, emaranhamento e ruído para resolver problemas computacionais. Por exemplo, múltiplas portas de Hadamard aplicadas a vários *qubits* geram um estado de superposição que abrange todas as possíveis combinações dos estados individuais dos *qubits*. A aplicação subsequente de outras portas quânticas, como as portas de Pauli, manipula as amplitudes e fases desses estados de superposição, codificando informações e realizando computações de maneiras inimagináveis no mundo clássico. Os circuitos quânticos básicos das portas I, Pauli-X, Pauli-Y, Pauli-Z e H são apresentados na Tabela 5.1.

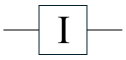
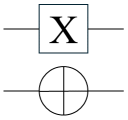
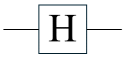
Porta Quântica	Circuito Quântico
Identidade (I)	
Pauli-X (X)	
Pauli-Y (Y)	
Pauli-Z (Z)	
Hadamard (H)	

Tabela 5.1: Relação entre as portas e os circuitos quânticos.

5.4. Modelo de Programação do IBM/Qiskit

O *Qiskit* é uma biblioteca *python* de propriedade da IBM publicada em 2017 e tendo o lançamento de sua versão estável em 2024 [Silva, 2018b, Javadi-Abhari et al., 2024]. Dentre suas diversas características, é possível utilizar essa biblioteca para:

- o desenvolvimento e experimentação de circuitos quânticos, com acesso a recursos como criação de circuitos com diferentes quantidades de *qubits*.

- integração com provedores para execução dos circuitos em computadores quânticos como os da IBM, Azure, Amazon, entre outros provedores.
- utilização de transpiladores, responsáveis por adaptar e traduzir o circuito para execução em computadores quânticos reais ou simulados.
- simular a execução dos circuitos em sistemas clássicos.
- visualizar os circuitos e os *qubits* em forma de esferas de Bloch, e dos resultados observados.
- criação de portas quânticas personalizadas.

Neste capítulo, usamos a versão 1.2.2 do *Qiskit*. A instalação pode ser realizada usando o comando `pip install qiskit` no seu ambiente. Recomendamos também a instalação do módulo de visualização, através do comando `pip install qiskit[visualization]`, além das bibliotecas: *matplotlib*, como complemento na visualização dos resultados; *numpy*, para auxiliar com a manipulação dos dados; e a biblioteca *math*, para uso de algumas funções matemáticas importantes para o desenvolvimento de um algoritmo quântico.

5.4.1. Circuitos no Qiskit

Para a criação de um circuito, a preparação de um ambiente de execução e a análise dos resultados é necessário usar um conjunto de classes e funções disponíveis no Qiskit.

Conforme apresentado no Código 5.1, a representação de um circuito é feita por meio da classe *QuantumCircuit*. É necessário também usar a função *transpile* que adapta as portas e os circuitos para a execução.

Já a classe *BasicProvider* é o provedor escolhido para os experimentos, neste caso simulando o circuito em um ambiente clássico de computação quântica. A classe *Statevector* é usada para retornar o vetor de estados do circuito; no nosso caso foi usado também para visualização das esferas de Bloch.

Por fim, foram usadas as funções *plot_histogram* e *plot_bloch_multivector* para a renderização de histograma e esferas de Bloch, respectivamente.

```
from qiskit import QuantumCircuit, transpile
from qiskit.providers.basic_provider import BasicProvider
from qiskit.quantum_info import Statevector
from qiskit.visualization import plot_histogram,plot_bloch_multivector
```

Código 5.1: Importações necessárias para o desenvolvimento do circuito teste proposto

Ao instanciar um objeto a partir da classe *QuantumCircuit*, é possível definir no construtor quantos *qubits* serão utilizados em um circuito. No Código 5.2, o circuito *qc* foi criado com 3 *qubits*.

```
qc = QuantumCircuit(3)
qc.h(0)
qc.h(1)
qc.h(2)
```

Código 5.2: Criação e inicialização de um circuito

Deste ponto em diante, são definidas as portas quânticas de forma declarativas diretamente no circuito. Para cada porta, o Qiskit possui diferentes comandos e parâmetros para configurará-las. Por exemplo, a porta Hadamard é alocada através do método *h* no objeto do circuito e pede apenas um parâmetro: quais os *qubits* são entradas para esta porta. No Código 5.2, os três *qubits* do circuito passarão pela porta Hadamard conforme declarado nas linhas 3 a 5.

Caso medíssemos este circuito neste exato momento, observaríamos que todos os estados possuem a mesma probabilidade de serem observados. Isto se deve em função das portas Hadamard realizar uma superposição simples, conforme apresentado na Seção 5.3.2. Isso também pode ser observado nas esferas de Bloch apresentadas na Figura 5.2.

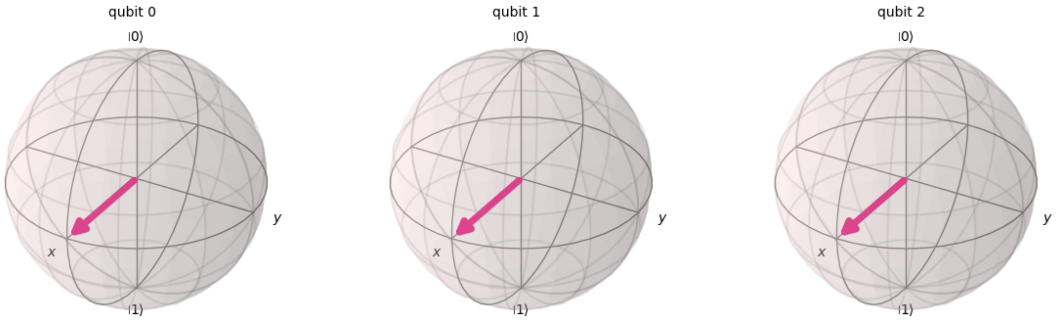


Figura 5.2: Circuito com três *qubits* em superposição inicial renderizado com o Qiskit

Para entender melhor como adicionar novas portas neste circuito e, consequentemente entender melhor as rotações que podem ser feitas nos *qubits*, vamos aumentar nosso sistema quântico, como pode ser visto no Código 5.3. Ao primeiro *qubit*, adicionamos uma rotação Z de 135°. Tal rotação é realizada pelo método *ry*, passando como parâmetro os graus em radianos, $3\frac{\pi}{4}$, seguido do índice do *qubit*, 0.

```
qc.rz(3*pi/4,0)
qc.ry(pi/4,1)
qc.ry(-pi/4,2)
qc.rx(pi/4,2)
qc.measure_all()
qc.draw('mpl')
```

Código 5.3: Aplicação das rotações nos 3 *qubits* seguida da medição do circuito

Ao segundo *qubit*, adicionamos uma rotação Y de 45°, representada pelo método *ry*, passando como parâmetro π radianos seguido d índice do *qubit*, 1. Ao terceiro qubit, adicionamos uma rotação Y de $-45°$ (ou 315°) seguida de uma rotação X de 45°, executada pelos métodos *ry* e *rx*, com os parâmetros $-\frac{\pi}{4}$ e $\frac{\pi}{4}$, respectivamente, com o índice do *qubit*, 2.

Após a execução dessas movimentações no circuito, é possível medi-lo. Para isso, utilizamos o método *measure* indicado quais *qubits* que desejamos observar. No caso do Código 5.3, foram medidos todos os *qubits* com o método *measure_all*. Ao final, pode ser interessante solicitar para o Qiskit renderizar o circuito construído usando o método *draw('mpl')*. O resultado do desenho deste circuito pode ser visto na Figura 5.3. Vale mencionar que essa visualização pode ser feita em qualquer lugar do seu código, durante a montagem do circuito.

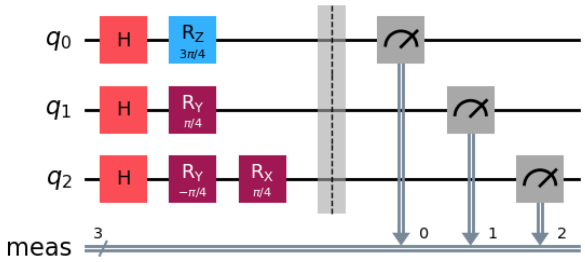


Figura 5.3: Desenho automático do circuito com inicialização de sobreposição, rotações, e medição dos *qubits*

Após a descrição do circuito, podemos executá-lo. Para isso, é necessário escolher um provedor disponível. No caso de uma simulação simples pode ser utilizado o provedor *BasicProvider* e seu *backend*, conforme mostra o Código 5.4. Esse *backend* simula o circuito, conseguindo obter resultados muito próximos de um cenário real.

Em seguida, é necessário *transpilar* o circuito para poder executá-lo. *Transpilar* um circuito é um processo onde as portas são adaptadas para execução no provedor selecionado. Cada provedor pode ter particularidades especificas que remetem a forma como o *qubit* é manipulado. Ao usar o Qskit, o simulador é baseado no funcionamento dos computadores quânticos da IBM, e recebem a tratativa para isso. O processo de transpilação é feito pela função *transpile*, indicando o circuito e o *backend* como parâmetros. O retorno é um circuito transpilado, o qual será utilizado para a execução.

```
provider = BasicProvider()
backend = provider.get_backend("basic_simulator")
new_circuit = transpile(qc, backend)
job = backend.run(new_circuit, shots=1024)
result = job.result()
```

Código 5.4: Exemplo de como simular um circuito quântico e obter seu resultado

A execução do (novo) circuito é feita no método *run* do *backend* escolhido, indicando o circuito transpilado. Nesta etapa, podemos controlar a quantidade de *shots* a serem executados, isto é, quantas medições do mesmo circuito serão executadas. Para que possamos calcular a probabilidade que cada estado obteve nas medições, o número que utilizaremos será um padrão de 1024. Caso estivéssemos utilizando um provedor de um computador físico (real), precisaríamos aguardar a execução do nosso circuito (que pode estar em uma fila) para podermos obter os resultados. Como não é o caso, podemos recuperar os resultados logo em seguida através do método *result* do *job* criado anteriormente.

Os resultados dessa execução podem ser visto usando a função `plot_histogram`, que recebe como parâmetro as contagens da execução dos resultados medidos (obtidas através do método `get_counts`, conforme pode ser visto no Código 5.5. A Figura 5.4 é um exemplo de execução do circuito apresenta na Figura 5.3. Como computadores quânticos possuem tanto o fator aleatório, quanto o fator ruido (que, neste caso, também é simulado), a cada execução, obtemos resultados diferentes, ainda que semelhantes, devido a probabilidade do nosso circuito e ao número de medições.

```
counts = result.get_counts(qc)
plot_histogram(counts)
```

Código 5.5: Exemplo para visualização do histograma dos resultados

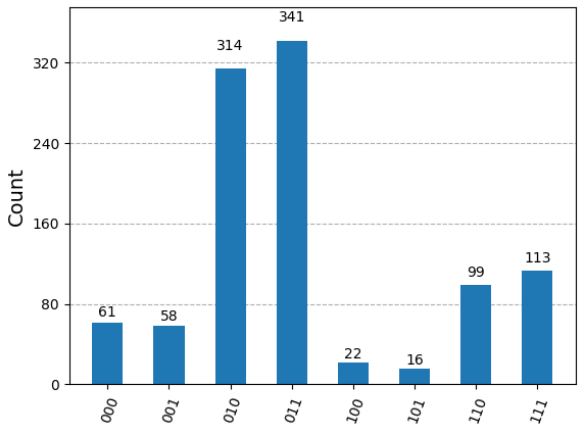


Figura 5.4: Histograma de uma execução do circuito da Figura 5.3

Podemos observar nesta Figura 5.4 alguns resultados das rotações previamente aplicadas: o terceiro *qubit* foi mais observado no estado $|0\rangle$ (ele é o primeiro digito dos números do eixo x do histograma); o segundo *qubit* foi mais observado no estado $|1\rangle$; enquanto o primeiro *qubit* teve observações semelhantes para ambos os estados. Tais resultados eram esperados, uma vez que o segundo e o terceiro *qubits* foram rotacionados de tal forma a possuírem uma maior probabilidade nos respectivos estados observados. Se observamos a Figura 5.5, podemos notar que o terceiro *qubit* ficou em uma posição próxima do estado $|0\rangle$, da mesma forma que o segundo se aproximou do estado $|1\rangle$, enquanto o primeiro *qubit*, apesar de ser rotacionado, não favoreceu sua medição em qualquer estado específico – portanto manteve-se com 50% para ambos os estados.

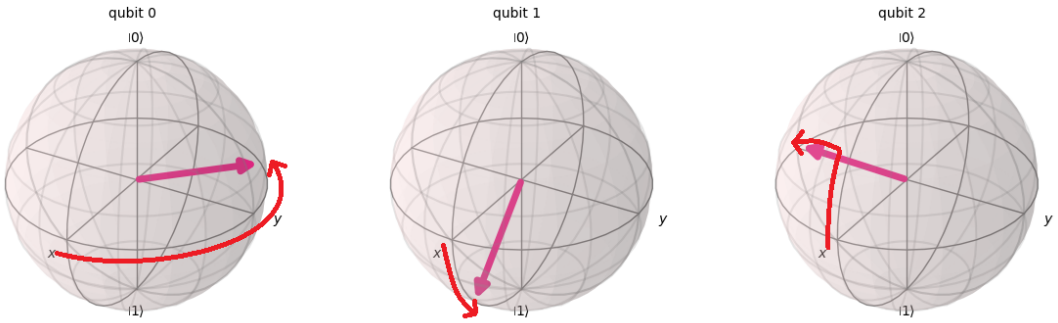


Figura 5.5: Esferas de Bloch para o circuito da Figura 5.3 após aplicarmos a rotações, destacando seu movimento em vermelho

Por fim, a partir dos trechos de códigos apresentados, temos os conceitos básicos de como construir um circuito e executá-lo por meio da biblioteca IBM/Qiskit.

5.5. Aplicações usando IBM/Qiskit

Neste seção apresentamos algumas aplicações amplamente conhecidas que utilizam computação quântica. As soluções são apresentados usando a biblioteca Qiskit, bem como seu circuito e o resultado de suas execuções.

5.5.1. Algoritmo de Grover

Imagine uma lista Telefônica contendo N nomes, arranjados em ordem aleatória. Para se encontrar o telefone de alguém com probabilidade de $1/2$, qualquer algoritmo clássico (seja ele determinístico ou probabilístico) vai necessitar observar no mínimo $N/2$ nomes [Grover, 1996, Shafique et al., 2024].

Grover propôs em 1996 um algoritmo quântico que resolve o problema da busca desordenada com complexidade $O(\sqrt{N})$, onde N é a quantidade de *qubits* sendo utilizados, superando portanto a complexidade de qualquer algoritmo clássico que resolva o problema de busca. Mesmo em cenários ordenados, a complexidade mínima necessária é de $O(n \log n)$.

O algoritmo de Grover originalmente consiste em três passos:

1. Inicializar o circuito em uma superposição
2. Repetir as seguintes etapas $\sqrt{N}$ vezes:
   - Rotacionar a fase buscada em π radianos
   - Aplicar a matriz de difusão D no circuito, definida por $D = HRH$, onde H é a porta $Hadamard$ e R é uma matriz diagonal cujo primeiro elemento é 1, e o restante é -1
3. Observar o circuito

O primeiro passo é trivial para a operação de circuitos quânticos, e não necessariamente é obrigatório para o funcionamento do algoritmo de Grover. O que é necessário

no entanto, é que o sistema esteja em superposição quando aplicarmos o passo seguinte. Por conta de tal comportamento, este primeiro passo pode ser substituído por um oráculo que opere como uma forma de *algoritmo inicial*, que poderia processar diferentes dados, e que usasse o algoritmo de Grover para buscar o resultado desejado.

O segundo passo podemos chamar de principal componente do algoritmo, pois é nele que a *busca* é aplicada e operada. Explicando de forma bem simplificada, este passo essencialmente *"marca"* o estado a ser buscado e então e através de um operador matriz, *"acentua"* tal estado, tornando-o mais provável de ser observado. De forma mais intrínseca, essa *marcação* é realizada através de uma rotação Y de π radianos no estado desejado, seguida por uma matriz de difusão D, que se trata de uma porta definida por Grover como a sequência das portas HRH. A porta H é simplesmente a aplicação da porta Hadamard, enquanto R é um operador matriz, mais especificamente uma matriz diagonal, com a diagonal definida pelo primeiro elemento como -1, e os demais 1. O segundo passo precisa ser repetido $\sqrt{N}$ vezes, pois, quanto maior o número de *qubits*, menos efetiva a acentuação da fase, cenário contornado portanto pela múltipla execução da matriz de difusão.

O terceiro e ultimo passo também é trivial, e se trata da medição do circuito. Vale mencionar, no entanto, que não necessariamente o circuito precisaria finalizar com o algoritmo de Grover, uma vez que o estado encontrado poderia passar por mais uma camada de processamento. O resultado da acentuação da fase pode ser observado por uma chance de *pelo menos* 50% de medição de tal estado. Outro detalhe que podemos mencionar é que podemos buscar mais de uma fase ao mesmo tempo, porém a probabilidade de medição seria dividida entre tais estados. Portanto, quanto mais elementos buscado, menos efetiva é a acentuação de fase, sendo um ponto de atenção ao trabalharmos com o algoritmo de Grover.

Logicamente, podemos reproduzir o algoritmo no Qiskit sem grandes adaptações. Além das importações discutidas na Seção anterior, no Código 5.1, é necessário algumas outras bibliotecas, conforme visto no Código 5.6. A classe *DiagonalGate* é utilizado para aplicação de matrizes diagonais sem a necessidade de criar toda a matriz – basta usar um vetor que represente tal diagonal. Já as demais bilbiotecas são conhecidas: *numpy*, para criação e manipulação de vetores; e *sqrt* e *floor*, utilizada para calcular raízes quadradas e para arredondar para baixo, respectivamente.

```
from qiskit.circuit.library import DiagonalGate
import numpy as np
from math import sqrt,floor
```

Código 5.6: Importações auxiliares para o algoritmo de Grover

O algoritmo completo de Grover pode ser observado no Código 5.7. Na linha 1 estamos declarando a quantidade de *qubits N* que, neste caso, é 3. Na segunda linha há a variável $faseBuscada$ que define o estado que será buscado utilizando o algoritmo de Grover que, neste caso o estado 011 (vale mencionar que podemos utilizar diversas maneiras para identificar tal fase). Para demontrar a explicação anterior, por hora, vamsos para a linha 23 que é onde o circuito *Grover* é criado com *N qubits*. Todos estes *qubits* são

iniciador com a porta Hadamard na linha seguinte. Dessa forma, concluímos o primeiro passo do algoritmo.

```
1  N=3
2  faseBuscada='011'
3
4  #preparação da matriz de rotaço
5  diagonalMatrizRotacao = -np.ones(2**N,dtype=int)
6  diagonalMatrizRotacao[0] = 1
7  matrizRotacao = DiagonalGate(diagonalMatrizRotacao)
8  matrizRotacao.name = 'R'
9
10 #preparação da matriz de difusão
11 matrizDifusao = QuantumCircuit(N, name='D')
12 matrizDifusao.h(range(N))
13 matrizDifusao.append(matrizRotacao,range(N))
14 matrizDifusao.h(range(N))
15
16 #preparação do oráculo
17 rotacaoFase = np.ones(2**N,dtype=int)
18 rotacaoFase[int(faseBuscada,2)] = -1
19 oraculo = DiagonalGate(rotacaoFase)
20 oraculo.name='oraculo'
21
22 #Passo 1
23 Grover = QuantumCircuit(N)
24 Grover.h(range(N))
25
26 #passo 2
27 for i in range(floor(sqrt(N))):
28     #etapa I
29     Grover.append(oraculo,range(N))
30     #etapa II
31     Grover.append(matrizDifusao,range(N))
32
33 #passo 3
34 Grover.measure_all()
35
36 Grover.draw('mpl')
```

Código 5.7: Algoritmo de Grover usando o Qiskit

Voltando para as linhas 4 a 8 do Código 5.7, criamos um pequeno circuito adicional para ser utilizado como uma porta matriz de rotação R utilizada pela matriz de difusão. Na linha 5 temos um vetor $diagonalMatrizRotacao$ composto pelos números inteiros -1, de tamanho 2^N, criado automaticamente pela função do *numpy*: *ones*. Em seguida, alocamos o valor 1, concluindo assim a diagonal de rotação. Na linha 7 utilizamos o comando *DiagonalGate* que, dada a diagonal de rotação, retorna um circuito com a aplicação desta matriz digonal. Este circuito posteriormente pode ser aplicado a outro circuito no formato de porta, explicitando nossa intenção. A linha 8 altera o rótulo desse circulo, por meio da propriedade *name*, para R.

As linhas 10 a 14 do Código 5.7 preparam um circuito que representa a matriz de difusão D. Na linha 11 criamos este circuito $matrizDifusao$, de tamanho N e com o rótulo como D. Nas linhas 12 e 14 aplicamos as portas Hadamard em todos os *qubits*, conforme previsto na definição da matriz de difusão. Na linha 13, entre as duas Hadamard, aplicamos a matriz de rotação, através do comando *append*, passando o circuito a ser utilizado como porta.

Nas linhas 16 a 20, é preparado um oráculo básico que rotaciona em π radianos o estado procurado. Tal rotação pode ser obtida através da aplicação de uma matriz diagonal, composta por elementos 1, e com as fases a serem marcadas com valor -1. Tal diagonal, nomeada de *rotacaoFase* é criada na linha 17. Na linha 18 aplicamos o valor negativo ao respectivo estado buscado. Na linha 19 criamos o circuito *oraculo* que servirá como porta utilizando novamente a função *DiagonalGate*, e por fim renomeamos o circuito para *oraculo* na linha 20.

Agora que finalizamos de configurar os circuitos e as portas, passamos para o segundo passo do algoritmo. Na linha 27 do Código 5.7 iniciamos um *loop* para repetir $\sqrt{N}$ vezes o processo do passo 2. Dentro desta repetição aplicamos a rotação do elemento buscado e a matriz de difusão, ambos aplicados pelo comando *append* usando, como parâmetros, o circuito (ou porta) *oraculo* e $matrizDifusao$, em todos os *qubits*, respectivamente.

Por fim, no terceiro e último passo do algoritmo, realizaremos a medida do circuito para observar os resultados, conforme descrito na linha 34. Na linha 36 há o desenho do circuito, conforme podemos observar na Figura 5.6.

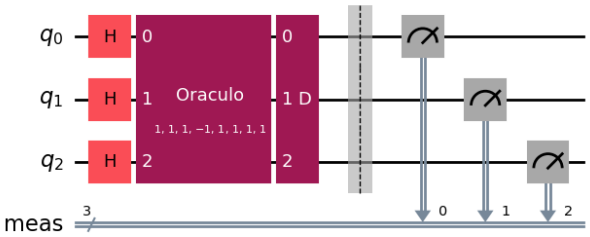


Figura 5.6: Exemplo de um circuito do algoritmo de Grover com um oráculo

Para executar esse código utilizaremos o mesmo princípio apresentado no Código 5.4. Além de usar o novo circuito *Grover*, mudamos a quantidade de *shots* para 10000, mas mantendo a simulação. O histograma resultado da execução pode ser observado na Figura 5.7. Notamos facilmente a medição do estado 011, que se destaca pelas 7800 medições, ou 78% delas, demonstrando a efetividade do algoritmo de Grover para a procura de um elemento. Na Figura 5.8 podemos ver a execução do algoritmo com 4 *qubits*, buscando os estados 0011 e 1010, evidenciando a capacidade de execução do algoritmo para busca de mais de um elemento.

Note como o algoritmo de Grover foi capaz de operar com todos os estados ao mesmo tempo, em sobreposição, utilizando rotações para diferenciá-los, ainda que inicialmente estivessem todos no mesmo estado. O algoritmo de Grover é por muitas vezes

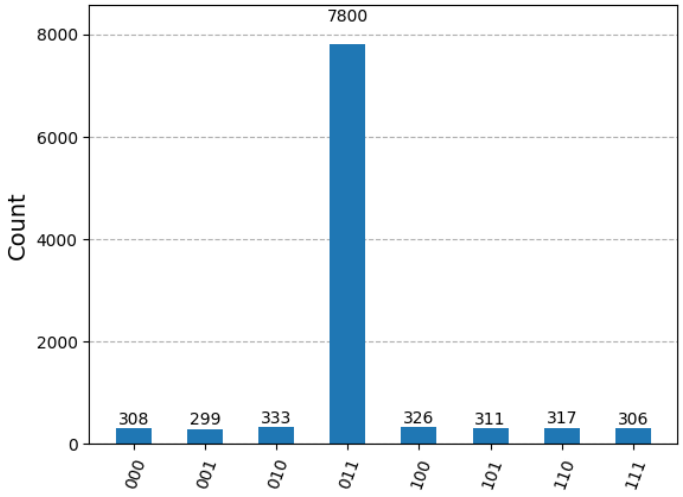


Figura 5.7: Contagem das medições da execução do algoritmo de Grover, com 3 qubits, buscando o estado 011

utilizado em conjunto com outros algoritmos. É também utilizado para fins didáticos pela sua simplicidade e eficácia de execução, demonstrando o potencial da computação quântica para diferentes aplicações.

5.5.2. Resolvendo um Autômato Celular

Autômatos celulares são um modelo de computação úteis na representação de sistemas dinâmicos e complexos por apresentarem um comportamento local (em cada célula) simples, cujo sistema completo detém um comportamento complexo e caótico com o passar do tempo. Cada célula é uma unidade de computação que age de acordo com regras de transição em passos discretos de tempo, de modo que podemos representar tais sistemas de forma genérica por uma função, conforme mostra a Equação 19.

$$\sigma_i^{T+1} = \phi(\{\sigma_j^{(T)}\}, j \in N_i) = \phi(\sigma_{i-1}^{(T)}, \sigma_i^{(T)}, \sigma_{i+1}^{(T)}), \tag{19}$$

Nesta equação, σ_i^{T+1} é uma célula i na próxima geração, e ϕ é a regra de transição do autômato, sendo N_i o conjunto da vizinhança de σ_i [McIntosh, 2009]. Em outras palavras, o estado de uma célula na próxima geração do sistema é definida por regras que consideram os vizinhos mais próximos daquela célula, assim como ela mesma, no momento atual do tempo.

A Equação 19 considera autômatos celulares de apenas uma dimensão, isto é, o sistema pode ser representado por um vetor de células, lado à lado. Portanto esta regra também considera que cada célula possua apenas 2 vizinhos, configurando uma vizinhança de 3 células ao todo; logo, este é o autômato celular mais simples possível, também chamado de elementar.

Por fim, a notação utilizada para representar as regras em um autômato celular elementar de uma dimensão surge a partir do seu valor binário. Há 8 possibilidades de

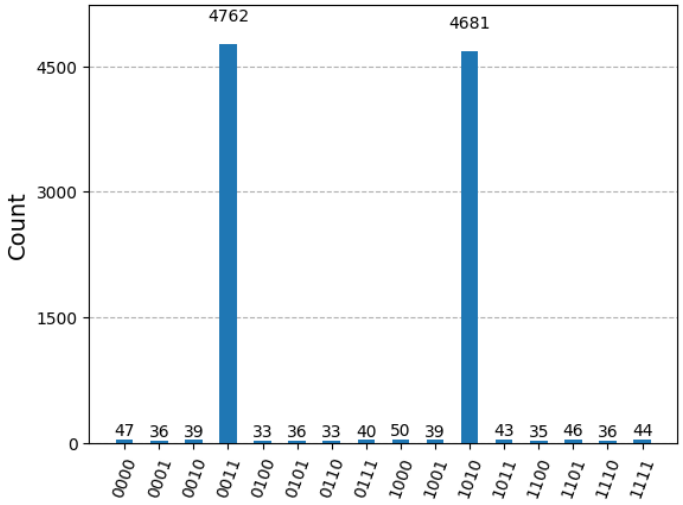


Figura 5.8: Contagem das medições da execução do algoritmo de Grover, com 4 qubits, buscando os estados 0011 e 1010

configurações dentro de uma vizinhança de tamanho 3 e, ao se pensar em cada uma dessas possibilidades como um *bit* em uma *string* de *bits*, podemos adquirir um valor inteiro que informa com qual regra se está trabalhando e quais as configurações que fazem uma célula viver ou morrer [Wolfram, 2002]. A Equação 20 descreve de forma genéria a transcrição das configurações e seu arranjo na *string* de *bits*, onde são apresentadas todas as configurações possíveis em uma vizinhança de um autômato celular elementar em relação ao próximo estado da célula ao centro, sendo β o estado da célula i na próxima geração.

$$(\sigma_{i-1}^{(T)}, \sigma_{i}^{(T)}, \sigma_{i+1}^{(T)}) : \frac{111}{\beta_1}, \frac{110}{\beta_2}, \frac{101}{\beta_3}, \frac{100}{\beta_4}, \frac{011}{\beta_5}, \frac{010}{\beta_6}, \frac{001}{\beta_7}, \frac{000}{\beta_8}. \tag{20}$$

A *Regra 90* é utilizada neste capítulo como demonstração em função da didática e simplicidade, o que torna a derivação dela direta, tanto da fórmula *booleana* quanto do circuito. Na Equação 21 há a apresentação por extenso da *Regra 90*, sendo os valores dos numeradores as configurações possíveis para uma determinada vizinhança e, denominadores, o valor da célula central no próximo momento no tempo. Em seguida, na Equação 22, é apresentada sua forma reduzida.

$$R_{90} : \frac{111}{0}, \frac{110}{1}, \frac{101}{0}, \frac{100}{1}, \frac{011}{1}, \frac{010}{0}, \frac{001}{1}, \frac{000}{0}. \tag{21}$$

$$f_{90}(l,m,r) = XOR[l,r] \tag{22}$$

, onde l denota a célula da esquerda, m a do meio e r a da direita.

A Equação 22 pode ser obtida ao observar o comportamento da regra de acordo com a interação que ocorre na vizinhança. A célula central passa a assumir o valor 1 se e somente se as células à esquerda e à direita forem diferentes, isto é, elas não podem ser tanto 1 quanto 0 ao mesmo tempo. Isto pode ser transcrito para uma fórmula *booleana* que descreve este comportamento, simplesmente definida pela porta XOR. Da mesma

forma, outras regras definidas para este tipo de autômato também podem ser transcritas para suas versões *booleanas* [Wolfram, 2002].

Para o algoritmo quântico, propomos uma espécie de *framework* de trabalho para criação de oráculos [Zhao, 2022]. Primeiro, encontre a fórmula *booleana* que descreva o comportamento desejado. Em seguida, construa o circuito que a descreve. Vale observar, por exemplo que, neste caso, a fórmula é facilmente encontrada ao observar a vizinhança e como a célula central se altera de acordo com seus companheiros. Porém para situações mais complexas, é recomendável ter calma e construir uma tabela verdade do sistema.

A partir deste ponto, é possível transpor a regra de transição para um circuito quântico que realiza esta operação utilizando uma das portas quânticas que melhor se encaixa ao automato desejado. Uma outra opção é encontrar a porta unitária que descreve as transformações necessárias aos *qubits* e trabalhar a partir dele para derivar quais outras portas podem fazer tais transformações. Neste capítulo, usamos a primeira abordagem, tanto pela sua simplicidade quanto pela praticidade.

A operação clássica XOR possui como análogo na computação quântica a porta $CNOT$ (NOT controlado ou X-controlado) em um sistema de dois *qubits*, O comportamento da porta $CNOT$ pode ser observado na Tabela 5.2. Esta porta age de acordo com o estado de um *qubit* de controle e aplica uma rotação *pi* (180°) no eixo X do *qubit* alvo, ou seja, altera seu estado de 0 para 1 e vice-versa. Esta rotação tem um efeito semelhante ao de uma porta NOT em álgebra booleana. Assim, se o *qubit* de controle está no estado 1, o *qubit* alvo terá seu estado revertido.

ψ_1	ψ_2
00	00
01	11
10	10
11	01

Tabela 5.2: Configuração de um sistema com uma porta CNOT onde o *bit* de controle é o mais à direita, e o *bit* alvo o mais à esquerda

Desta lógica, podemos utilizar o segundo *qubit* (alvo) como indicativo de ambos estarem no mesmo estado ou não (com valores 0 e 1, respectivamente). Portanto, o *qubit* alvo determina se uma célula estará viva ou morta na próxima geração com a necessidade de codificar somente os valores de seus vizinhos para o circuito. O resultado desta operação em um circuito de 2 qubits pode ser observado na Figura 5.9.

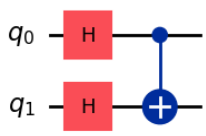


Figura 5.9: Configuração de um XOR em um circuito quântico

Com esta abordagem, considera-se que cada célula é um sistema quântico local com circuito próprio, ou seja, um autômato celular quântico é um vetor n-dimensional de sistemas quântico n-dimensionais [Arrighi, 2019].

O circuito ilustrado na Figura 5.9 forma a base para computar um avanço temporal em uma célula. Embora ele já realize a verificação da igualdade entre valores, é crucial indicar ao circuito se algum dos *qubits* tende a possuir o valor inicial 1. Esta tarefa é atribuída ao oráculo do autômato e pode ser realizada através de rotações nos *qubits*, utilizando, por exemplo, as portas T, que introduz uma fase no *qubit*, aumentando sua amplitude quando este assume o valor 1, e Z, que inverte a fase quando o *qubit* está no estado 1. Essas duas portas estão descritas na Equação 23. Esses operadores unitários agem apenas quando o *qubit* está no estado 1, permitindo-nos "*sinalizar*" ao circuito a configuração da vizinhança. Após tais modificações, aplicamos uma porta Hadamard adicional em ambas as células, consolidando as alterações nas amplitudes resultantes das rotações.

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix} \tag{23}$$

Antes de iniciarmos o processo de superposição, precisamos rotacionar o segundo *qubit* para o estado $|1\rangle$, pois, por padrão, o Qiskit inicializa todos os *qubits* do circuito no estado $|0\rangle$. Tal configuração indica um vizinho morto, enquanto a ausência dela indica um vizinho vivo. O resultado final do circuito que representa a *Regra 90* pode ser observado na Figura 5.10. O código completo desse circuito pode ser visto no Código 5.8.

Neste código definimos a operação local de cada célula do sistema ao passarmos como parâmetro os estados das células à direita e à esquerda em relação à central. Nele, computamos a igualdade dos valores dos vizinhos e medimos o resultado ao fim do circuito, o qual pode ser usado futuramente para alterar o valor da célula central. Com isso, temos o programa que traduz esta regra de transição para um sistema quântico.

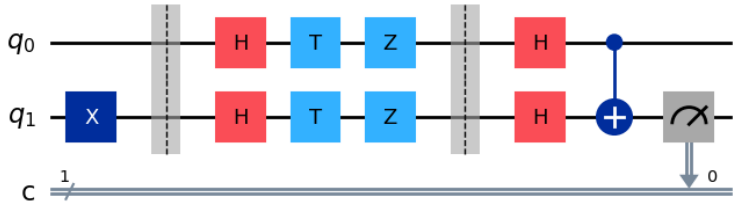


Figura 5.10: Circuito que considera uma configuração específica de uma vizinhança, neste caso $f(1,m,0)$

Com este circuito, podemos simular e observar se ele se comporta da maneira desejada. Conforme observado, de maneira geral, as simulações são feitas em alguns passos:

- Instancia-se o simulador desejado

- Otimiza o circuito para o tipo de hardware no qual ele será executado
- Executa o circuito para adquirir os resultados

```python
from qiskit import QuantumCircuit
from qiskit.circuit.library import HGate, CXGate, TGate, ZGate

def step(left: int, right: int) -> QuantumCircuit():
  # Instancia um circuito
  c = QuantumCircuit(2, 1, name="step")

  # Indica onde deve ser aplicada porta NOT
  if left == 0:
    c.x(0)
  if right == 0:
    c.x(1)
  c.barrier()

  # Preparação inicial - superposição e rotações
  c.append(HGate(), [0])
  c.append(HGate(), [1])
  c.append(TGate(), [0])
  c.append(TGate(), [1])
  c.append(ZGate(), [0])
  c.append(ZGate(), [1])
  c.barrier()

  # Alterar a superposição
  c.append(HGate(), [0])
  c.append(HGate(), [1])

  # Realizar a verificação se os estados são semelhantes
  c.append(CXGate(), [0,1])

  # Adquirir o resultado
  #   - Se 1, as células possuem estados diferentes
  #   - Se 0, as células possuem estados iguais
  c.measure(1,0)
  return c
```

Código 5.8: Código que traduz a regra de transição 90 para um circuito quântico

Essa descrição de simulação pode ser vista no Código 5.9. Apesar dessa simulação ter seus passos bem definidos, este código foi escrito para a versão atual do Qiskit, podendo sofre alterações, conforme seu fabricante achar importante.

Com os resultados da simulação, podemos analisar como o circuito se comportou e verificar se ele realmente implementa a regra de transição para cada célula.

Assim, ao executar o circuito definido com um simulador ideal, isto é, sem ruído, nota-se que conseguimos realizar a transição de uma célula de acordo com os estados de seus vizinhos, conforme mostra a Figura 5.11.

```python
from qiskit_aer import AerSimulator
from qiskit.transpiler.preset_passmanagers import
    ↪ generate_preset_pass_manager

def simulate_ideal(circuit, shots=1024):
    # Instancia o simulador
    sim_ideal = AerSimulator()

    # Otimiza o circuito para o tipo de hardware
    pm = generate_preset_pass_manager(backend=sim_ideal,
    ↪ optimization_level=3)
    isa_qc = pm.run(circuit)

    # Roda a simulação
    result = sim_ideal.run(circuit)

    return result
```

Código 5.9: Função para simulação ideal de um circuito quântico

A partir do retorno do circuito, podemos alterar o estado da célula em questão. Ao aplicar este circuito para cada célula de um conjunto de células é possível resolver um autômato celular onde cada célula constitui um sistema quântico. Neste caso, se seu retorno for uma medição 0, significa que aquela célula estará morta na próxima geração e, se uma medição for 1, ela estará viva. A localidade e simplicidade características de autômatos celulares é mantida e a complexidade na evolução do sistema também.

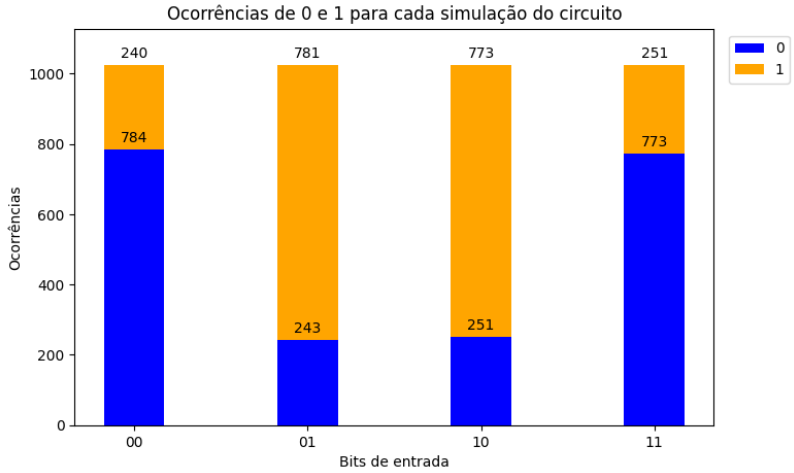


Figura 5.11: Contagem das medições de 1s e 0s de acordo com a configuração dos vizinhos de uma determinada célula

Os passos apresentados nessa seção podem ser usados para criação de oráculos com diversos propósitos em circuitos quânticos, em especial na tradução das regras de transição para autômatos celulares. Resumidamente, primeiro, adquire-se a função *booleana* que representa o comportamento desejado e esperado do sistema, a qual servirá

de orientação para a implementação do oráculo. Após a análise desta fórmula, busca-se encontrar um circuito que replique o seu comportamento. Para tal, podemos utilizar de tabelas verdades, árvores de decisão ou outros análogos na computação clássica como o feito nesta seção. E, por fim, constrói-se o circuito que deve ser modificado para diferentes entradas.

5.6. Conclusão

A computação quântica, um paradigma revolucionário que aproveita os princípios da mecânica quântica, tem imenso potencial para remodelar o cenário da computação e revolucionar diversos campos da ciência e da sociedade. Diferente da computação clássica, que depende de bits representando 0 ou 1, a computação quântica utiliza *qubits*. Eles podem existir em superposição, representando simultaneamente 0 e 1, o que permite que os computadores quânticos realizem cálculos exponencialmente mais rápidos para conjuntos específicos de problemas. Esse paralelismo inerente constitui a base do poder da computação quântica. O entrelaçamento quântico, outra característica única, estabelece uma forte correlação entre os *qubits*, ampliando ainda mais as capacidades computacionais.

Os fundamentos teóricos da computação quântica foram inicialmente estabelecidos neste capítulo, com os circuitos quânticos fornecendo modelos robustos para entender e projetar algoritmos quânticos. Outros aspectos desta fundamentação teórico ainda precisam ser exploradas, como, por exemplo, a máquina de Turing quântica [Yanofsky and Mannucci, 2008].

Além dos circuitos, pode-se perceber também a aplicação prática de algoritmos quânticos para resolver problemas clássicos da computação. Por exempo, o conceito de autômatos celulares quânticos apresenta um modelo promissor para a computação quântica. Os autômatos celulares quânticos operam em uma grade de *qubits*, evoluindo através de regras locais para realizar cálculos, oferecendo uma possível via para o paralelismo e a escalabilidade. Outro importante algoritmo quântico, o algoritmo de Grover, fornece uma aceleração quadrática para problemas de busca não estruturada. O algoritmo de Grover demonstra a ampla aplicabilidade da computação quântica para uma variedade maior de tarefas, incluindo busca em banco de dados e otimização.

Apesar de sua promessa, a computação quântica enfrenta desafios significativos. A decoerência, que é a perda das propriedades quânticas devido a interações com o ambiente, representa um grande obstáculo na construção de computadores quânticos confiáveis. O ruído quântico agrava ainda mais esse problema, exigindo técnicas robustas de correção de erros para preservar a integridade dos cálculos quânticos. Abordar esses desafios é crucial para escalar os computadores quânticos e seus respectivos algoritmos de forma eficaz para lidar com problemas do mundo real.

A busca pela computação quântica continua a cativar pesquisadores e líderes da indústria, talvez, motivado pelos seus grandes desafios [Parmar, 2024]. O campo testemunhou marcos notáveis, incluindo a demonstração da supremacia quântica pelo Google em 2019, ao realizar uma tarefa computacional impossível para computadores clássicos em um período de tempo aceitável. Embora a verdadeira supremacia quântica ainda seja um tema de pesquisa contínua, esses feitos destacam o progresso tangível em direção à realização do pleno potencial da computação quântica [Baczyk, 2024].

Por fim, a computação quântica representa uma mudança de paradigma na computação, prometendo revolucionar a pesquisa científica, a inovação tecnológica e diversos aspectos de nossas vidas. Embora desafios significativos estejam à frente, a busca incessante por essa tecnologia transformadora continua a inspirar, prometendo um futuro onde o enigmático mundo da mecânica quântica nos capacita a resolver problemas antes considerados impossíveis.

Todos os códigos utilizados neste capítulo estão disponíveis em um repositório no GitHub em `https://github.com/hpc-fci-mackenzie/SSCAD24-QuantumComputing`.

Agradecimentos

Os autores agradecem ao MackCloud[1], Laboratório Multidisciplinar de Computação Científica e Nuvem e ao projeto SPRACE - Processo nº 2018/25225-9, Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP).

Referências

[Arrighi, 2019] Arrighi, P. (2019). An overview of quantum cellular automata.

[Arute et al., 2019] Arute, F., Arya, K., Babbush, R., Bacon, D., Bardin, J. C., Barends, R., Biswas, R., Boixo, S., Brandao, F. G. S. L., Buell, D. A., Burkett, B., Chen, Y., Chen, Z., Chiaro, B., Collins, R., Courtney, W., Dunsworth, A., Farhi, E., Foxen, B., Fowler, A., Gidney, C., Giustina, M., Graff, R., Guerin, K., Habegger, S., Harrigan, M. P., Hartmann, M. J., Ho, A., Hoffmann, M., Huang, T., Humble, T. S., Isakov, S. V., Jeffrey, E., Jiang, Z., Kafri, D., Kechedzhi, K., Kelly, J., Klimov, P. V., Knysh, S., Korotkov, A., Kostritsa, F., Landhuis, D., Lindmark, M., Lucero, E., Lyakh, D., Mandrà, S., McClean, J. R., McEwen, M., Megrant, A., Mi, X., Michielsen, K., Mohseni, M., Mutus, J., Naaman, O., Neeley, M., Neill, C., Niu, M. Y., Ostby, E., Petukhov, A., Platt, J. C., Quintana, C., Rieffel, E. G., Roushan, P., Rubin, N. C., Sank, D., Satzinger, K. J., Smelyanskiy, V., Sung, K. J., Trevithick, M. D., Vainsencher, A., Villalonga, B., White, T., Yao, Z. J., Yeh, P., Zalcman, A., Neven, H., and Martinis, J. M. (2019). Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510.

[Baczyk, 2024] Baczyk, M. (2024). Shall you buy a quantum computer today? - analysis of qc on-premise deployments. *Quantum Computing Report*. Accessed: 2024-08-08.

[Deutsch, 1985] Deutsch, D. (1985). Quantum theory, the Church–Turing principle and the universal quantum computer. *Proc. R. Soc. Lond.*, 400(1818):97–117.

[Feynman, 1982] Feynman, R. P. (1982). Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6-7):467–488.

[Gamberi and Bianchini, 2023] Gamberi, G. P. and Bianchini, C. P. (2023). Study of quantum algorithms and their implementations. In *2023 International Conference on Electrical, Communication and Computer Engineering (ICECCE)*, pages 1–6.

[1] `https://mackcloud.mackenzie.br`

[Grover, 1996] Grover, L. K. (1996). A fast quantum mechanical algorithm for database search.

[Javadi-Abhari et al., 2024] Javadi-Abhari, A., Treinish, M., Krsulich, K., Wood, C. J., Lishman, J., Gacon, J., Martiel, S., Nation, P. D., Bishop, L. S., Cross, A. W., Johnson, B. R., and Gambetta, J. M. (2024). Quantum computing with qiskit.

[Jorio and Frossard, 2024] Jorio, A. and Frossard, J. V. (2024). *Material de Estudos para Mecânica Quântica*. Programa de Pós-Graduação em Física, UFMG, 2nd edition.

[McIntosh, 2009] McIntosh, H. V. (2009). *One Dimensional Cellular Automata*. Luniver Press.

[Microsoft Quantum, nda] Microsoft Quantum (n.d.a). Quantum computing concepts: Entanglement. `https://quantum.microsoft.com/en-us/insights/education/concepts/entanglement`. Accessed: 2024-09-05.

[Microsoft Quantum, ndb] Microsoft Quantum (n.d.b). Quantum computing concepts: Superposition. `https://quantum.microsoft.com/en-us/insights/education/concepts/superposition`. Accessed: 2024-09-05.

[Nayak et al., 2024] Nayak, P., Rathod, S., Surabhi, and Sukanya (2024). Quantum computing: Circuits, algorithms and application. *International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)*, 4(1).

[Nielsen and Chuang, 2010] Nielsen, M. A. and Chuang, I. L. (2010). *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press.

[Parmar, 2024] Parmar, D. (2024). Patent landscape for quantum computing: A survey of patenting activities on different physical realization methods. *IPWatchdog*. Accessed: 2024-08-08.

[Preskill, 2012] Preskill, J. (2012). Quantum computing and the entanglement frontier.

[Shafique et al., 2024] Shafique, M. A., Munir, A., and Latif, I. (2024). Quantum computing: Circuits, algorithms, and applications. *IEEE Access*, 12:22296–22314.

[ShareTechNote, nd] ShareTechNote (n.d.). Quantum computing - bloch sphere. `https://www.sharetechnote.com/html/QC/QuantumComputing_BlochSphere.html`. Accessed: 2024-09-05.

[Shor, 1994] Shor, P. (1994). Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134.

[Silva, 2018a] Silva, V. (2018a). *Practical Quantum Computing for Developers: Programming Quantum Rigs in the Cloud using Python, Quantum Assembly Language and IBM QExperience*. Apress.

[Silva, 2018b] Silva, V. (2018b). *Practical Quantum Computing for Developers: Programming Quantum Rigs in the Cloud Using Python, Quantum Assembly Language and IBM QExperience*. Apress L.P., New York.

[Valdez and Melin, 2022] Valdez, F. and Melin, P. (2022). A review on quantum computing and deep learning algorithms and their applications. *Soft Computing*.

[Wolfram, 2002] Wolfram, S. (2002). *A New Kind of Science*. Wolfram Media.

[Yanofsky and Mannucci, 2008] Yanofsky, N. S. and Mannucci, M. A. (2008). *Quantum computing for computer scientists*. Cambridge University Press.

[Zhao, 2022] Zhao, J. (2022). Possible implementations of oracles in quantum algorithms. *J. Phys. Conf. Ser.*, 2386(1):012010.

Capítulo 6

Técnicas para análise e otimização de programas

Aleardo Manacero (UNESP)

Abstract

Parallel programs are solutions to enable results production for high computing costs problems. Even though the simple parallelization of such programs produce better execution times, it does not assure achieving the best possible efficiency. In order to achieve such efficiency one has to apply optimization techniques all over the original program. However, when you have a very large code to optimize, it is virtually impossible to perform optimizations uniformly over the whole code. This demands the application of a preliminary step, which is performance analysis of the program. This will identify parts of the code that are more relevant for optimizations than others. In this chapter we will study some of the definitions about performance analysis, the major techniques used and some of the available tools to perform it. english

Resumo

Programas paralelos são soluções para viabilizar a produção de resultados em problemas de alto custo computacional. Embora a simples paralelização desses programas já melhore os tempos necessários para execução, ela não garante a efetiva obtenção da maior eficiência possível. Para obter tal eficiência é preciso aplicar técnicas de otimização sobre todo o programa original. Entretanto, quando se tem um código bastante grande para otimizar é virtualmente impossível aplicar otimizações de modo uniforme sobre todo o código. Isto demanda a aplicação de uma etapa preliminar, que é a análise de desempenho do programa. Esta análise identificará partes do código que são mais relevantes para otimização do que outras. Neste capítulo serão estudadas as definições sobre análise de desempenho, as principais técnicas utilizadas e algumas das ferramentas disponíveis para realizar a análise.

6.1. Objetivos de análise de desempenho

O processo de análise de desempenho é necessário para que se tenha informações adequadas para otimizar um programa. A otimização, que basicamente envolve a aplicação de determinadas técnicas de redução de esforço computacional sobre um programa, é uma ação necessária em qualquer sistema (basta imaginar aplicativos lentos em seu smartphone). Para sistemas de alto desempenho a otimização de uma dada aplicação é ainda mais necessária, pois nesse caso ineficiências implicam em desperdício de ciclos de máquina, ou seja, desperdício de dinheiro [1, 2].

Entretanto, otimizar um dado programa pode se tornar uma tarefa bastante complicada caso o código seja muito longo. Para esses casos a análise de desempenho se torna vital, pois é por meio dessa análise que teremos informações sobre que partes do programa consomem mais tempo de CPU, por exemplo.

A análise de desempenho é, na prática, uma atividade cíclica, em que um determinado programa é medido, analisado e otimizado. Esse ciclo pode ser visto na Figura 6.1, sendo que basicamente o processo consiste em medir a execução do programa, fazer seu perfilamento[1], aplicar possíveis otimizações e verificar se os resultados obtidos continuam formalmente válidos, antes de um novo ciclo.

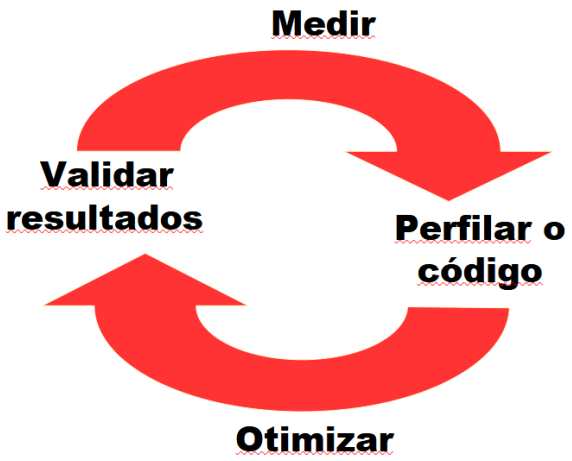


Figura 6.1. Ciclo de atividades no processo de otimização e análise de desempenho.

Mesmo considerando o ciclo indicado, que parece ser simples, ainda existem questões importantes a serem definidas. A principal delas é definir o que se quer de fato, isto é, qual o motivo de otimizar um programa. Mais acima indicamos que, por exemplo, podemos querer reduzir o tempo de CPU consumido. Entretanto, existem objetivos alternativos, como a redução no espaço ocupado em memória, ou ainda a redução no tempo gasto com a troca de dados pela rede. Logo, decidir o que medir e otimizar passa a ser função de quem faz a análise, embora exista uma forte tendência em considerar que o principal parâmetro é tempo [3].

[1]Perfilamento, como veremos mais adiante, é o ato de identificar quais partes do programa estão sendo executadas e em qual proporção isso ocorre

6.1.1. Formas de "medir" desempenho

Outro aspecto importante na definição de medição e otimização de desempenho envolve a definição de quais métodos serão aplicados na obtenção dos perfis de execução. Raj Jain [4] nos ensina que esses métodos podem ser classificados em três grupos: analíticos, baseados em simulação e medição efetiva (que chamaremos de *benchmarking* a partir daqui).

Desses métodos, apenas *benchmarking* é efetivamente uma técnica de medição, enquanto os métodos analíticos e de simulação, embora importantes, são técnicas para predição de desempenho. Na seção 6.4 detalharemos mais os conceitos envolvidos com os três métodos para produzir dados para a análise de desempenho.

6.1.2. Desempenho em sistemas paralelos

Para sistemas paralelos a definição do que devemos otimizar e como obteremos dados para essa otimização ficam ainda mais complicados. Nesses sistemas devemos considerar aspectos não tratados em sistemas sequenciais, como a competição por recursos, atrasos por comunicação, necessidade de sincronismo entre processos paralelos, etc.

Além disso, o fato de paralelizar a execução implica em otimizar o nível de paralelismo, isto é, quantos processos devemos executar em paralelo. A obtenção dessa medida implica em definir novas métricas, além de tempo de execução por exemplo. Na prática essas métricas envolvem o quanto uma execução será acelerada (*speedup*) e o ponto em que essa aceleração é de fato vantajosa (escalabilidade). Analisamos melhor esses aspectos na próxima seção.

6.2. Métricas para análise de desempenho: *speedup*, escalabilidade, eficiência

O aproveitamento de paralelismo introduz dificuldades adicionas, como a existência de atrasos criados pela necessidade de sincronismo, ou a impossibilidade de paralelizar determinadas atividades da aplicação. Esses fatores de degradação de desempenho acabam por definir, que dentro de ambientes paralelos, as métricas relevantes são *speedup* e escalabilidade.

6.2.1. Escalabilidade

A escalabilidade é, na prática, uma medida de até que ponto o paralelismo é vantajoso. Isso significa determinar o quanto um problema ou sistema pode crescer sem prejuízo significativo de desempenho. Sendo assim, a escalabilidade não é uma métrica que indique um valor máximo para algum parâmetro, mas sim de um valor a partir do qual fazer um sistema crescer passa a ser desvantajoso.

É importante destacar que escalabilidade não é uma medida única, ou seja, existem diferentes parâmetros que podem afetar a escalabilidade de um sistema. Entre esses parâmetros podemos destacar:

1. Custo de comunicação, em que a paralelização (aumento no número de processadores paralelos) fica limitada pelo que custarão roteadores, *switches*, cabos e placas de rede;

2. Necessidade de comunicação, em que o crescimento no número de processos paralelos pode ser limitado pelo volume adicional de comunicação;

3. Granulosidade do programa, em que o tamanho dos grãos (trechos de código executados em uma máquina) determina a necessidade de comunicação e uma eventual ociosidade na espera por novos dados;

4. Tamanho do problema, quando se identifica o quanto um problema pode crescer a partir dos recursos de hardware (discos, memória, etc.) disponíveis;

5. Custo de programação, uma vez que o nível e forma de paralelismo resulta em formas mais ou menos fáceis de se programar, podendo aumentar o custo no desenvolvimento de uma aplicação.

Outros parâmetros que impactam na escalabilidade incluem demanda por memória, demanda por entrada/saída, tamanho físico do sistema, etc.

Slowdown Deve ter ficado claro que a escalabilidade é uma medida de limitação ao paralelismo e desempenho. Um sistema pouco escalável implica em se ter um sistema que não pode crescer ou aumentar seu desempenho. A falta de escalabilidade de um sistema paralelo resulta, em geral, no que se chama de *slowdown* ou *parallel slowdown*. Um sistema entra em *slowdown* quando, a partir de um certo grau de paralelismo, o sistema passa a executar mais lentamente, perdendo desempenho apesar do crescimento no número de processos paralelos. Vejam que isso é ainda pior do que determinar o ponto no qual o crescimento deixa de ser vantajoso.

6.2.2. *Speedup*

Essa métrica indica o grau de aceleração obtido com a paralelização de um programa. Essa aceleração pode ser entendida como a razão entre o tempo necessário para executar o programa de forma sequencial e o tempo consumido em sua execução paralela, como visto na equação 1.

$$S = \frac{T_{sequencial}}{T_{paralelo}} \tag{1}$$

Embora essa definição pareça simples, a prática nos mostrou que existem formas diferentes de definir ou medir essa aceleração. Temos então, inicialmente, duas formas de *speedup*: teórico e medido. O *speedup* teórico é dado por relações matemáticas cujos parâmetros envolvem qual porcentagem de um código pode ser paralelizado e o número de processadores paralelos que serão usados. Já o *speedup* medido é obtido por meio de uma comparação direta entre os tempos medidos para execuções sequenciais e paralelas.

6.2.2.1. *Speedup* teórico

Speedups teóricos servem, basicamente, para predizer qual seria o grau ótimo de paralelismo para uma dada aplicação. Na prática não são uma medida real, mas sim uma

indicação sobre a vantagem ou não em se paralelizar. A primeira definição de *speedup* veio de um trabalho de Gene Amdahl, que buscava desestimular o uso de máquinas SIMD ao final da década de 60, mais precisamente o Illiac IV [5]. Os resultados apresentados nesse trabalho foram posteriormente formalizados e implicaram no que se conhece como Lei de Amdahl, dada pela equação 2.

$$S = \frac{n}{1 + (n-1) \times \alpha} \tag{2}$$

Em que n é o número de máquinas paralelas e α representa a probabilidade de processamento sequencial no programa. O problema dessa expressão é que sua formulação resulta num forte desestímulo ao paralelismo, uma vez que qualquer aplicação sempre terá alguma parte que deverá ser executada de forma estritamente sequencial. A figura 6.2 mostra essa característica de forma bastante clara. Observa-se, por exemplo, que com 128 máquinas temos um *speedup* de apenas 1,98 se $\alpha = 0,5$, ou de 56,4 com um $\alpha = 0,01$ (ou apenas 1% de processamento sequencial).

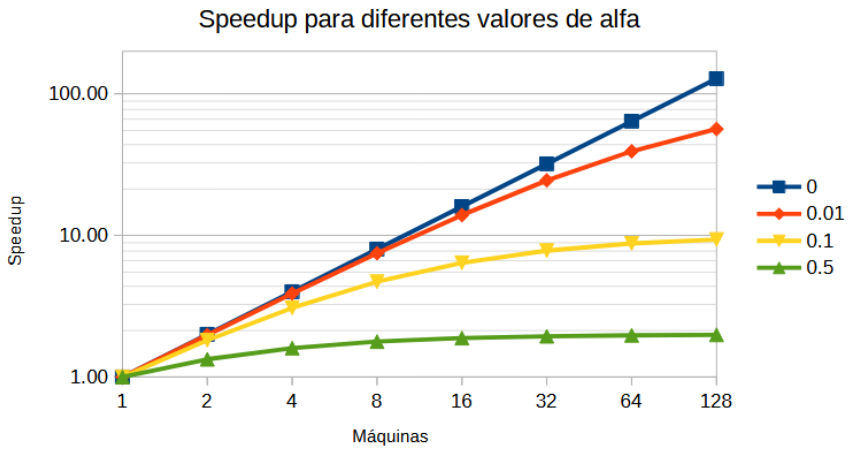


Figura 6.2. Curvas de *speedup*, segundo a Lei de Amdahl, para diferentes valores de α.

O problema com a Lei de Amdahl é que ela desconsidera um fator bastante relevante quando se usa um equipamento mais potente para resolver um problema, que é a ganância. Na prática, se recebemos uma máquina mais potente não a usaremos para resolver os mesmos problemas de antes. A tendência é que se passe a resolver problemas maiores, tanto aumentando a precisão dos resultados como trabalhando com maior quantidade de dados. Para Amdahl isso não foi considerado, levando a que seu equacionamento para *speedup* ficasse conhecido como sendo de tamanho fixo.

Gustafson, como muitos outros, percebeu que os valores de *speedup* dados pela Lei de Amdahl não correspondiam aos valores efetivamente medidos. Com isso acabou propondo o que se conhece como *speedup* de tempo fixo [6]. A ideia basica de Gustafson é considerar que ao paralelizar um programa podemos consumir, em paralelo, o mesmo tempo consumido em sua versão sequencial. Com isso a sua porção sequencial é significantemente reduzida, como vemos na Figura 6.3.

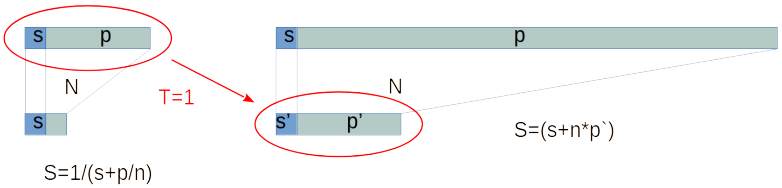


Figura 6.3. Concepção de speedup de tempo fixo.

A partir da ideia de manter o tempo de execução inicial, a razão entre os tempos de execução sequencial e paralelo são fortemente alterados. Isso leva à uma nova formulação, dada pela equação 3.

$$S = n - \alpha \times (n-1) \tag{3}$$

Os resultados obtidos com cada lei podem ser comparados ao examinar a figura 6.4. Fica evidente que os valores de *speedup* são bastante superiores quando determinados por Gustafson, sendo que para 128 máquinas temos S=64,5 para $\alpha = 0,5$ e S=126,7 para $\alpha = 0,01$.

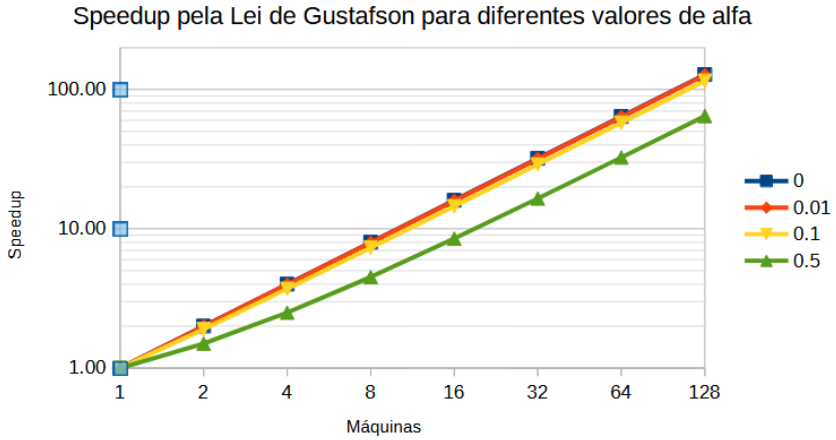


Figura 6.4. Curvas de *speedup*, segundo a Lei de Gustafson, para diferentes valores de α.

6.2.2.2. *Speedup* medido

Apesar de fornecer resultados mais realistas, a Lei de Gustafson ainda não é o procedimento mais adequado para se conhecer a aceleração propiciada pela paralelização de um programa. A forma mais realista de se obter o *speedup* efetivo de um programa é medindo os tempos de execução. Mas, obviamente, essa medição não é feita de forma única.

A medição do *speedup*, como definido pela equação 1, implica em medir dois tempos. Um deles só pode ser obtido de uma forma, que é com a execução paralela

do programa. O problema reside na determinação do tempo sequencial, que pode ser determinado de duas formas distintas. Numa delas esse tempo seria o da execução da versão estritamente sequencial do programa, se possível em sua forma mais eficiente. Na outra forma o tempo sequencial seria medido pela execução do programa paralelo usando apenas uma máquina.

Esses dois tempos sequenciais são primordialmente distintos, sendo o uso da versão sequencial a forma mais conservadora, uma vez que tipicamente será mais rápido do que a versão paralela executando em uma única máquina. Essa diferença é natural pois o programa paralelo sempre conterá instruções adicionais para efetivar o paralelismo. Essas instruções serão executadas, mesmo que a execução seja feita com apenas um processo paralelo, causando um aumento natural (*overhead*) no tempo de execução.

Este autor, assim como Lin e Snyder [7], entre outros, considera que a abordagem usando o tempo do programa paralelo executado em uma máquina deve ser evitado. A razão para isso é simples, pois se queremos saber quanto podemos acelerar um programa, temos que compará-lo com o que alguém executaria se não tivesse máquina paralela, ou seja, o melhor programa sequencial possível.

6.2.3. Eficiência

No início deste capítulo indicou-se que todo o processo de análise de desempenho é uma ferramenta para direcionar o processo de otimização. Neste sentido, uma métrica de desempenho importante é a eficiência. A medida de eficiência apresenta um certo paralelismo com a medida de escalabilidade, no sentido de que um sistema é escalável enquanto tiver uma alta eficiência.

Embora eficiência possa ser definida a partir de vários parâmetros, como estamos falando de computação de alto desempenho, e consequentemente de sistemas paralelos, adotaremos aqui uma definição de trate destes aspectos. Um sistema paralelo eficiente pode ser entendido como aquele em que a aceleração obtida com a sua paralelização é bastante próxima do grau de paralelismo utilizado. Assim, a eficiência de um sistema paralelo pode ser determinada pela equação 4, em que S é o valor de *speedup* e n o número de máquinas paralelas.

$$E(n) = \frac{S(n)}{n} \tag{4}$$

6.3. Medição: monitorar x instrumentar

A atividade de medir desempenho não se limita a determinar o *speedup* ou a eficiência do programa paralelo. Na verdade existem diversos outros aspectos que podem interessar a alguém, dependendo de seus objetivos. Com isso, objetivos diferentes levarão a métricas diferentes, sendo que a sua determinação é fundamental para a execução da análise de desempenho. Essa determinação resulta de respondermos a três perguntas, que são “para quê medir?”, “o quê medir?” e “como medir?”, que devem ser respondidas nessa ordem.

A primeira pergunta, ao ser respondida, restringe (ou direciona) as respostas às perguntas seguintes. Isso ocorre pois ao se definir para quê mediremos acabamos por eliminar certas respostas ao que deve ser medido e, consequentemente, como isso será

medido. Exemplos de respostas à esta pergunta incluem:

- Determinar o *speedup*,
- determinar o nível de ocupação de memória,
- determinar o tempo consumido em determinadas funções do programa,
- determinar o volume de tráfego na rede, etc.

Uma vez determinada a razão para quê realizaremos medidas no sistema é preciso identificar o que será efetivamente medido. Como já indicado, essa resposta depende, em parte, da resposta anterior. A lista a seguir representa possíveis respostas ao que deve ser medido, mas deve ficar claro que podem ou não ser uma opção, a depender do motivo da medição.

- Medir o tempo de entrega do programa,
- medir o tempo gasto em uma dada função,
- medir a quantidade de bytes transferidos por mensagem,
- medir o número de mensagens enviadas, etc.

Por fim, temos que determinar como mediremos os parâmetros desejados. Técnicas de medição podem ser divididas em duas grandes categorias, que são monitoração e instrumentação.

6.3.1. Monitoração

Técnicas de monitoração se baseiam no uso de mecanismos físicos, ou próximos disso, para medir aquilo que se deseja medir. A monitoração pode ser realizada por hardware especializado, como analisadores lógicos, contadores digitais, osciloscópios, etc. Também pode ser realizada por software, por meio da programação de determinadas interrupções do *kernel* do sistema operacional para a obtenção de medidas.

A principal vantagem da monitoração é fornecer medidas mais precisas, principalmente quando realizada com hardware especializado. Entretanto, tem como desvantagem a necessidade de equipamentos específicos e conhecimento detalhado do funcionamento da máquina ou do sistema operacional.

Na prática, técnicas de monitoração são aplicadas apenas pelos próprios fabricantes de hardware, durante o seu desenvolvimento. Para situações mais mundanas a opção recai em alguma das formas de instrumentação, que além de ser mais barata, exige menor conhecimento sobre o sistema.

```
1  #include <time.h>
2  #include "stdio.h"
3
4  double etimes()
5  {struct timespec tp;
6        clock_gettime(CLOCK_REALTIME, &tp);
7        return(tp.tv_sec + tp.tv_nsec/1000000000.0);
8  }
9
10 main()
11 {double Duration;
12 ......
13    Duration = etimes();
14    do_whatever_has_to_be_measured();
15    Duration = etimes() - Duration;
16    printf ("Function took %f seconds\n", Duration);
17 }
```

Figura 6.5. Instrumentação por modificação do código fonte.

6.3.2. Instrumentação

Técnicas de instrumentação fazem a modificação da aplicação que se quer analisar, de forma a que sua execução produza os resultados de interesse. A instrumentação pode ocorrer em três momentos da produção de um programa. Isso implica em modificar o programa, instrumentando-o para a geração das medidas, diretamente no código fonte, ou em seu objeto, ou ainda diretamente no executável.

O processo de instrumentação produz resultados que podem ser vistos como a descrição do perfil de uma execução. Essa descrição é o chamado perfilamento do programa (*profiling*), formando o conjunto de dados para sua análise de desempenho. Embora o perfilamento seja a forma mais comum de produção de resultados, é importante mencionar que um outro mecanismo usado é a geração de traços de eventos (*event tracing*). A diferença entre os dois é que o perfilamento produz resultados pela amostragem do que está sendo executado, enquanto os traços de eventos consistem em registrar o instante em que determinados eventos ocorrem.

6.3.2.1. Modificação do código fonte

A instrumentação de um programa por meio da modificação do código fonte é a que permite detalhar pontos mais específicos de interesse. Seu problema é que ela é aplicável apenas se o código fonte estiver acessível, o que nem sempre é uma realidade. A modificação do código fonte depende de funções de medição de tempo na linguagem utilizada. O processo de medição é bastante simples, consistindo em inserir chamadas para as funções de tempo antes e depois da parte do código que queremos medir.

O exemplo da Figura 6.5 usa a função `clock_gettime` para obter o tempo medido pelo sistema operacional para construir a função `etimes`. No caso se quer medir

o tempo consumido na chamada da função `do_whatever_has_to_be_measured` (linha 15), que é então cercada por duas chamadas para a função `etimes`, sendo que a diferença de valores retornados pela primeira e segunda chamadas o tempo consumido pela função medida.

Esse processo é intrusivo e acaba por criar uma carga adicional de execução, pelas chamadas de funções adicionais. No exemplo dado podemos diminuir um pouco a intrusão colocando chamadas para `clock_gettime` diretamente antes e depois da função medida. Mas, mesmo assim, ainda temos alguma imprecisão na medição.

Um outro problema, que ocorre com qualquer técnica de instrumentação, é a imprecisão decorrente da forma como o sistema atualiza seu relógio. Apesar de funções como a apresentada acima permitirem a medição de nanossegundos, essa não é a precisão das mesmas. O que ocorre é que apesar do relógio de um computador oscilar a mais de 3 GHz, o que dá um ciclo de relógio a cada 0,33 ns, a variável que armazena tempo não é atualizada a cada ciclo e sim a cada intervalo de alguns microssegundos. Assim, a leitura do tempo atual sempre incorre em um certo erro.

6.3.2.2. Modificação do código objeto ou do executável

Aqui as funções para medição de tempo são inseridas no momento da compilação. Com isso temos que usar ferramentas projetadas para essa função, sendo que boa parte delas também depende do acesso ao código fonte (para fazer a compilação, é claro). Talvez a ferramenta mais conhecida para este tipo de instrumentação é `gprof`, que faz parte do pacote *gnu*. Diferentes versões do gprof foram desenvolvidas para sistemas específicos, como Solaris, HPUX, etc., mas estão descontinuadas hoje em dia.

O procedimento para a instrumentação de código com a modificação de código objeto consiste em compilar o código utilizando o um parâmetro que faça a instrumentação. Após a compilação o programa deve ser executado, gerando seus resultados de perfilamento para análise. Por exemplo, o código apresentado na figura 6.6, pode ser compilado e medido da seguinte forma:

```
$[1] gcc -pg loops.c -o loops
$[2] ./loops
$[3] gprof > loops.prof
```

O parâmetro *pg* no comando de compilação é que insere chamadas para funções de perfilamento. Essas funções, ao serem executadas, produzem medidas que serão listadas pela chamada de *gprof*, cuja saída está direcionada para o arquivo *loops.prof*. Parte desse arquivo é apresentada na figura 6.7.

Nessa figura é possível ver que o *gprof* produz sobre quanto tempo é consumido em cada função, tanto no total como em cada chamada. Deve ser observado que apesar da contagem de chamadas estar correta (ele de fato conta quantas foram), os tempos consumidos não são estritamente proporcionais ao que se esperaria pelo que é executado. Essa diferença é causada pela amostragem, que como já indicado, depende da frequência de atualização do valor do relógio. Apesar desse erro, é importante lembrar que o objetivo

```
main() {
  int l;
  for (l=0; l < 10000; l++) {
     if (l%2 == 0) foo();
     bar();
     baz();
  }
}

foo(){int j;
  for (j=0; j<200000; j++);
}

bar(){int i;
  for (i=0; i<200000; i++);
}

baz(){int k;
  for (k=0; k<300000; k++);
}
```

Figura 6.6. Código do programa *loops.c*, usado como exemplo de modificação do código objeto.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self                         self    total
 time    seconds  seconds    calls  us/call  us/call  name
 53.89      0.08     0.08    10000     8.08     8.08  baz
 33.68      0.13     0.05    10000     5.05     5.05  bar
 13.47      0.15     0.02     5000     4.04     4.04  foo

 index  % time    self  children    called      name
                                                <spontaneous>
 [1]    100.0    0.00    0.15                      main [1]
                 0.08    0.00   10000/10000       baz [2]
                 0.05    0.00   10000/10000       bar [3]
                 0.02    0.00    5000/5000        foo [4]
 -----------------------------------------------
                 0.08    0.00   10000/10000         main [1]
 [2]     53.3    0.08    0.00   10000             baz [2]
 -----------------------------------------------
                 0.05    0.00   10000/10000         main [1]
 [3]     33.3    0.05    0.00   10000             bar [3]
 -----------------------------------------------
                 0.02    0.00   5000/5000           main [1]
 [4]     13.3    0.02    0.00   5000              foo [4]
```

Figura 6.7. Trechos de informações produzidas pelo *gprof*.

desse tipo de medida é a indicação das proporções de uso para a otimização, e é fácil perceber que a função *baz()* é a que merece maior atenção, por ter um laço mais longo.

6.3.2.3. Modificação dinâmica do executável

A modificação do código executável pode ser feita de modo *offline*, como ocorre com o uso de perfiladores como o *gprof*. Existem, entretanto, mecanismos de modificação de código que operam de modo *online*. A ferramenta mais utilizada que faz esse tipo de abordagem é o **Dyninst**, que é uma API de instrumentação dinâmica desenvolvida em um projeto entre a Universidade de Wisconsin-Madison e Universidade de Maryland [8, 9].

O que o *Dyninst* faz é inserir a instrumentação para medição enquanto o programa está sendo executado. A vantagem com essa abordagem é que a instrumentação pode ser direcionada aos trechos mais custosos do programa, reduzindo a sobrecarga com códigos intrusivos. O principal problema dela é que o código a ser medido tem que executar por alguns minutos, de forma a permitir que a ferramenta execute a instrumentação dinâmica em tempo de realizar as medições.

6.3.3. Observações sobre monitoração e modificação de código

Para finalizar esta seção faremos uma breve comparação entre os métodos de medição baseados em monitoração e modificação de código. Já antecipamos que os processos de monitoração são mais precisos, uma vez que suas medições ocorrem de forma exata, com instrumentos físicos e sem a necessidade de codificação adicional. Já os mecanismos baseados em modificação de código, ao acrescerem comandos de instrumentação ao código original, acabam gerando erros por métodos intrusivos, além dos problemas inerentes aos erros de amostragem do relógio.

Apesar de mais precisos, métodos de monitoração têm uso bastante restrito por demandarem um bom conhecimento sobre instrumentos e sobre o funcionamento do sistema operacional e o hardware. Já os métodos baseados em modificação do código não necessitam de conhecimento muito elaborado, permitindo um uso mais amplo por parte de desenvolvedores e analistas.

Como conclusão, vemos que apesar de certa imprecisão, os métodos baseados em modificação de código são os mais utilizados.

6.4. *Benchmarks*

Os métodos apresentados até aqui se enquadram na categoria de métodos baseados em medição, ou *benchmarking*, que serão melhor examinados nesta seção. Antes, porém, descreveremos de modo mais abreviado os métodos baseados em predição, que são os métodos analíticos e baseados em simulação.

6.4.1. Predição de desempenho

Apesar de aparecer estranho, existem situações em que a análise de desempenho pode ser feita com métodos preditivos, em que não ocorrem medições efetivas sobre o objeto da análise. Essas situações envolvem, por exemplo, a análise do possível desempenho de um

sistema que ainda não exista fisicamente, mas cujo modelo possa ser elaborado de forma razoável.

Os métodos preditivos podem ser baseados em modelos analíticos, como cadeias de Markov, redes de Petri e redes de filas [10]. A análise de desempenho usando essas técnicas se baseia na criação de de modelos matemáticos, que ao serem resolvidos fornecem previsões sobre o comportamento de um sistema computacional. A precisão desses modelos depende, fundamentalmente, da qualidade dos modelos construídos, ou seja, se o modelo consegue representar o ambiente software-hardware mais precisamente, então os resultados da predição serão mais precisos.

Deve ficar claro, entretanto, que a resolução dos modelos analíticos só é feita por métodos não computacionais para modelos relativamente pequenos. Modelos mais complexos, como sistemas paralelos e aplicações neles executadas, demandam a solução por meio de simulações computacionais.

Simulação de eventos discretos é, de forma isolada, outra classe de métodos preditivos. A diferença aqui é que os modelos não são resultantes de formulações algébricas, como cadeias de Markov, e sim de descrições qualitativas do sistema. Técnicas baseadas em simulação fazem a modelagem usando características observáveis do sistema, como a descrição de seu funcionamento, das interações entre partes, etc. Esses modelos são construídos usando técnicas conhecidas, como redes de Petri, Monte Carlo e redes de filas [11].

Mais uma vez, a precisão de técnicas baseadas em simulação depende da qualidade do modelo criado para o sistema. Uma vantagem da simulação, em relação aos modelos analíticos, é que a qualidade do modelo é mais facilmente ajustada, permitindo possivelmente resultados mais precisos.

6.4.1.1. Comparando métodos preditivos e *benchmarking*

A análise de desempenho feita por diferentes métodos tem, naturalmente, diferentes impactos sobre o processo. A comparação entre os métodos pode ser feita considerando alguns aspectos como em qual estágio de desenvolvimento do sistema se pode aplicar uma dada técnica. A tabela 6.1 apresenta tal comparação, sendo que dela podemos destacar os seguintes pontos:

Tabela 6.1. Quadro comparativo entre métodos preditivos e *benchmarking*.

Critério	Analíticos	Simulação	*Benchmarking*
Estágio de desenvolvimento	Qualquer	Qualquer	Pós-Protótipo
Tempo na obtenção de resultados	Baixo	Médio	Variável
Ferramentas para modelagem	Humanos	Linguagens	Instrumentação
Exatidão	Baixa	Moderada	Variável
Avaliação de alternativas	Fácil	Moderada	Difícil
Custo	Baixo	Médio	Alto
Credibilidade do método	Baixo	Médio	Alto

- Exatidão, em que mostra que métodos preditivos de fato não apresentam grande precisão, mas também mostra que a precisão de *benchmarking* pode variar, dependendo de como é feita a instrumentação e, principalmente, como é definida a carga de trabalho durante as medições.
- Avaliação de alternativas, ou seja, como avaliar mudanças no software ou hardware podem ser usadas na busca por melhoria de desempenho, o que é fácil de fazer em modelos abstratos (tanto analíticos como de simulação), mas é difícil de fazer com o sistema real, pois demanda modificações no hardware ou a implementação de novos algoritmos no software.
- Credibilidade do método, que é um indicativo de como um leigo enxerga os resultados apresentados, que aponta para maior credibilidade para medições reais sobre o ambiente real, e menor para medições geradas por modelos abstratos.

6.4.2. Introdução ao *benchmarking*

O processo de *benchmarking* envolve a medição física de parâmetros de um sistema. É, portanto, um mecanismo que atua diretamente sobre o objeto de nosso interesse, tipicamente utilizando técnicas de modificação de código. Apesar de sua efetividade e aparente simplicidade, existem problemas importantes que devem ser resolvidos antes de sua aplicação, que são a definição da carga de trabalho (o que será medido) e como a sua execução pode ser comparada com outras situações. Esses aspectos nos levam à existência de dois tipos básicos de *benchmarking*, que são:

1. ***Benchmarks* produzidos por organizações**, tipicamente utilizados como padrões para comparação de sistemas de hardware. Nesses casos se têm uma organização, como a SPEC [12] ou NASA [13], que define um conjunto de aplicações que devem ser executadas e medidas, de forma a produzir uma medida global de desempenho. É interessante notar que esse tipo de medição produz resultados que podem ser comparados entre si, desde que a carga de trabalho seja equivalente.

   Tipicamente as diferentes organizações indicam várias aplicações complexas e que demandem características distintas do sistema, como operações de ponto flutuante (dinânica de fluídos, previsão metereológica, modelagem oceânica, etc.), operações com inteiros (compilação do gcc, simulação discreta, compressão de dados, etc.), uso de memória, etc.

   Na mesma categoria podemos encontrar outros *benchmarks*, como o Rodinia[2], que tem como atratividade ser desenvolvido dentro de uma universidade, o Linpack/Lapack[3], que é o *benchmark* utilizado para classificar os sistemas da lista **TOP500**, e o Passmark[4] (entre outros), que apresentam resultados de desempenho de CPUs, por exemplo.

2. ***Benchmarks* produzidos com objetivos locais**, usados para mensurar uma aplicação especifica, com validade local. São especificamente desenvolvidos para uma

[2]Disponível em www.cs.virginia.edu/rodinia
[3]Disponível em www.netlib.org/lapack
[4]Disponível em www.cpubenchmark.net

dada aplicação, em que os programas/casos de teste são escolhidos localmente, de acordo com os objetivos da entidade interessada na análise. Por terem essa característica a validade dos resultados também é local, não podendo ser generalizada nem mesmo para ambientes similares.

Na criação desses *benchmarks* é importante observar que os resultados, quando envolverem vários programas de teste, devem ser normalizados de acordo com sua relevância.

6.4.2.1. Normalização de *benchmarks* locais

A normalização dos resultados de um *benchmark* local deve considerar que a comparação não pode usar apenas tempos absolutos de execução. Ela deve considerar também o grau de importância dos programas que estão sendo medidos. Para entender como a normalização funciona podemos trabalhar um exemplo simples, em que queremos comprar um sistema, entre três possíveis (X, Y e Z), para executar três programas (A, B e C). As medidas obtidas com esses programas e sistemas é vista na tabela 6.2.

Tabela 6.2. Tempos medidos (em u.t.) durante os *benchmarks* realizados.

	Sistema X	Sistema Y	Sistema Z
Programa A	322	369	310
Programa B	694	801	714
Programa C	440	484	441

Essas medidas, se desconsiderados quaisquer outros fatores, indicaria que o sistema X levaria 1456 u.t. para executar uma instância de cada um dos três programas, enquanto o sistema Y levaria 1654 u.t. e o sistema Z levaria 1465 u.t. Com tais resultados um analista diria que o sistema X é que seria o mais rápido entre os três. Essa análise, entretanto, pode ser falha caso os três programas tenham diferentes graus de uso ao longo de um dia. Se, por exemplo, o programa A ocupar 65% do tempo de uso do sistema, o programa B ocupar 25% e o programa C ocupar 10%, então os resultados da tabela 6.2 podem ser normalizados considerando essas novas informações.

O processo de normalização pode ser feito transformando os tempos apresentados em valores relativos ao menor tempo de execução para cada programa. Por exemplo, para o programa A consideramos 310 u.t. como sendo 1,00 (no sistema Z). Assim, os tempos de execução nos sistemas X e Y seriam respectivamente 1,04 e 1,19. Após aplicarmos a normalização aos três programas, podemos colocar os pesos relativos ao uso de cada programa, multiplicando seu uso pelo tempo normalizado, resultando nos valores apresentados na tabela 6.3, em que se observa que o sistema Z é que produziria os melhores resultados para o padrão de uso da empresa.

6.5. Exemplos de ferramentas

Apesar de vários obstáculos para a medição do desempenho de um sistema, como a definição correta da carga de trabalho, existem várias ferramentas produzidas com essa

Tabela 6.3. Tempos normalizados e ponderados para os *benchmarks* realizados.

	Sistema X	Sistema Y	Sistema Z
Programa A	1,04*0,65	1,19*0,65	1,00*0,65
Programa B	1,00*0,25	1,15*0,25	1,03*0,25
Programa C	1,00*0,10	1,10*0,10	1,00*0,10
Tempo normalizado	1,03	1,17	1,01

finalidade, algumas pagas outras de livre acesso. Descreveremos aqui algumas dessas ferramentas.

6.5.1. Paradyn/Dyninst [9]

É uma ferramenta de acesso livre, cujo desenvolvimento se iniciou há mais de 20 anos. Seu princípio de medição é fazer a instrumentação dinâmica do código. Esse processo consiste em criar uma cópia da função que será medida, com as chamadas para instrumentação e alterar a chamada para essa função, o que se denomina trampolim, para que fizesse a chamada da função modificada e não da original. Após a medida o trampolim é retirado e a função cópia é removida. A figura 6.8 mostra o grafo de chamadas de funções avaliadas em um dado programa, no qual as funções com nomes em preto ainda não foram instrumentadas.

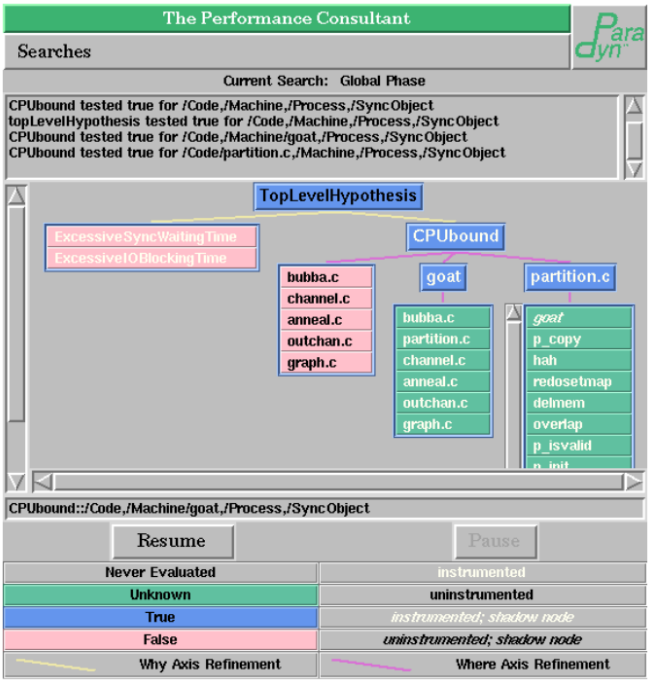


Figura 6.8. Interface do grafo de chamadas do Paradyn.

Um aspecto interessante nessa ferramenta é que sua base, o Dyninst, também é usada em várias outras ferramentas de análise de desempenho, como HPCToolkit, TAU e Omnitrace, por exemplo.

6.5.2. Vampir [14]

Diferente do Paradyn, esta é uma ferramenta com uma versão funcional em caráter demonstrativo, porém seu uso continuo depende de licenças pagas (algo entre 720 e 30.000 euros anuais). Na figura 6.9 temos a interface que mostra a linha de tempo de execução de um conjunto de processos, mostrando os trechos do código executados a cada momento.

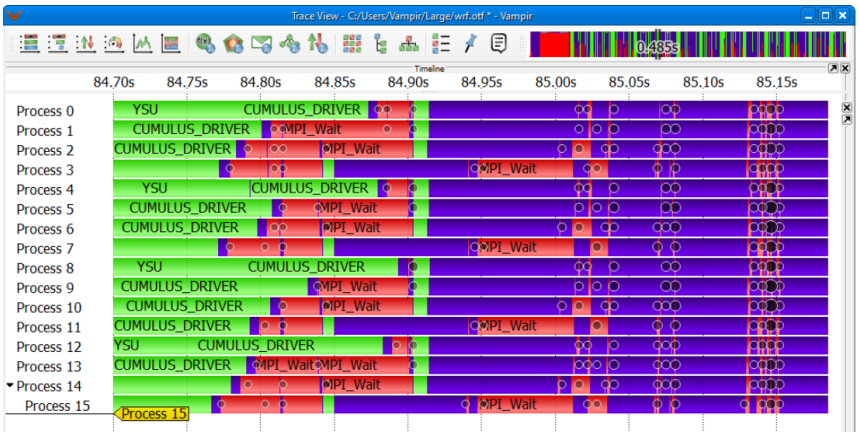


Figura 6.9. Interface de linha do tempo do Vampir.

6.5.3. vTune/Advisor [15]

Estas ferramentas fazem parte de uma suíte para análise de sistemas (oneAPI) oferecida pela Intel. Elas permitem análises em sistemas com CPU, GPU e FPGA, sendo portanto bastante versáteis. Advisor é uma ferramenta que indica otimizações em um programa, incluindo quais trechos do código deveriam ser executados por um acelerador (GPU, por exemplo), enquanto vTune faz o perfilamento da execução.

Uma métrica interessante oferecida pelo Advisor é o que se chama *roofline analysis*, que pode ser usada para identificar que volume de processamento pode ser levado para a GPU, ou ser vetorizado, como visto na figura 6.10.

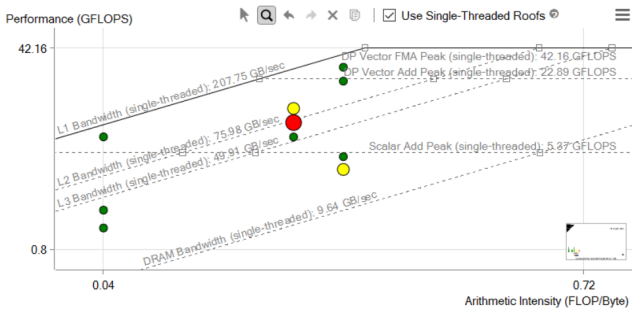


Figura 6.10. Interface do Advisor com informações sobre rooflines.

Em github.com/pvelesko/nbody-demo.git são encontrados vários exemplos de aplicação dessas ferramentas, que são disponibilizados por Paulius Velesko. A partir deles gerou-se, por exemplo, os dados mostrados na figura 6.11, que mostra a sequência de

execução de threads no programa. Na versão com interface gráfica se deve configurar o projeto, com indicação de executável e parâmetros de execução.

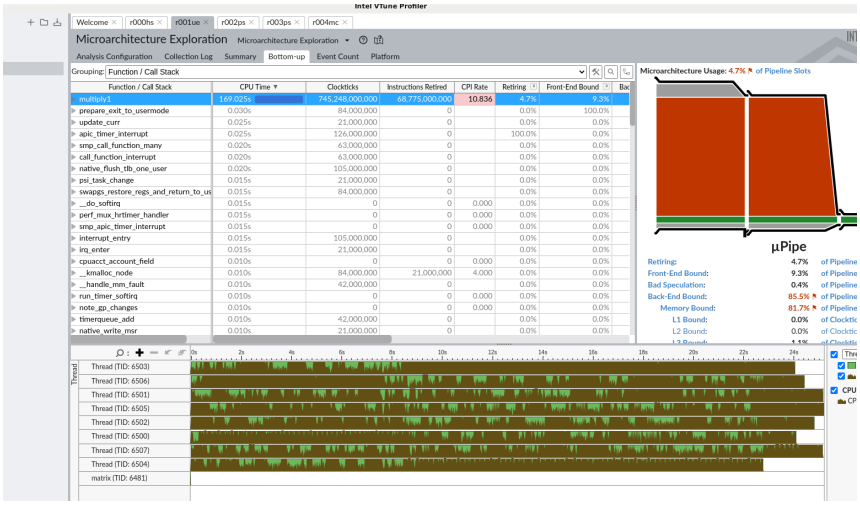


Figura 6.11. Interface do vTune com informações sobre threads executados.

6.5.4. TAU [16]

Desenvolvido na Universidade do Oregon, em parceria com Los Alamos National Laboratory (LANL) e Jülich Research Centre. É uma ferramenta bastante flexível, que pode ser configurada para diferentes máquinas e ambientes, gerando traços/perfilamentos da aplicação, que podem ser inclusive manipulados por outras ferramentas, como Vampir.

O programa a ser analisado deve ser compilado usando parâmetros específicos que indiquem o que deve ser medido. A execução do programa gera arquivos de traços para cada núcleo utilizado, que podem ser visualizados com duas ferramentas, que são o PerfExplorer e ParaProf.

Na figura 6.12, por exemplo, temos uma janela do Paraprof mostrando onde cada processo foi executado em um programa em MPI.

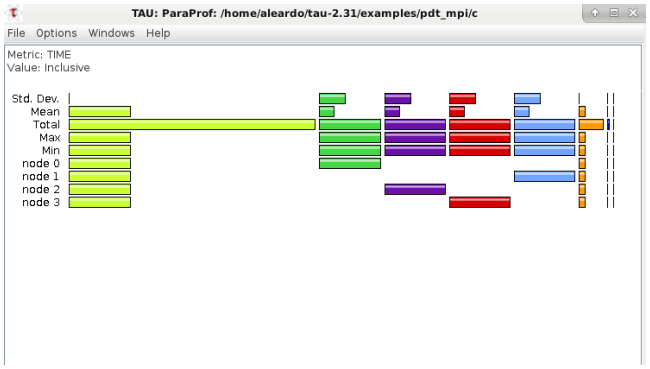


Figura 6.12. Interface do Tau/Paraprof, com a distribuição de processos por nó de processamento.

Com as ferramentas de visualização é possível obter um conjunto bastante amplo de métricas, incluindo a clusterização de resultados no caso de elevado número de processos paralelos, como pode ser visto na figura 6.13.

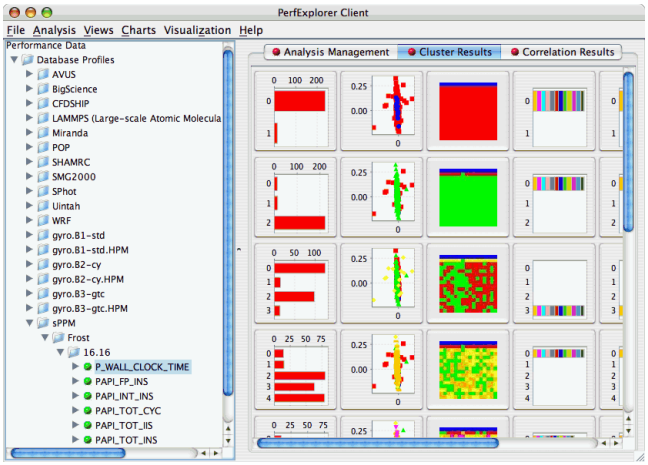


Figura 6.13. Interface do Tau/PerfExplorer, com uma análise clusterizada dos resultados.

6.6. Técnicas para otimização de desempenho

Otimização é a identificação de pontos em um programa que podem ser melhorados e, principalmente, a realização de tais melhorias. Esse processo pode ser feito de forma automatizada, pelo compilador, ou pelo próprio programador. Quando feita pelo compilador se utiliza as técnicas clássicas de otimização em compilação, como alinhamento de funções ou desenrolamento de laços.

6.6.1. Aspectos que impactam o desempenho

Um programa pode ser otimizado em três aspectos, que são o estilo de programação, a arquitetura geral do sistema ou o código do programa. Quando falamos em arquitetura geral do sistema, falamos em topologia, elementos de processamento ou conexão e modelos de particionamento do software. Destes, apenas a forma de particionamento é que pode ser modificada de modo simples, embora implique em modificar como dados são separados entre os processadores, de modo a garantir a melhor granulosidade dos processos paralelos.

Otimizar baseado em estilo de programação significa escolher uma linguagem mais eficiente para programar (para muitas aplicações numericamente intensivas isso ainda recai no Fortran), ou estruturas de dados mais eficientes. Sobre estas um aspecto importante é que se deve evitar o uso desnecessário de ponteiros. Por exemplo, apesar de ser possível armazenar um grafo usando ponteiros, evitando desperdício de memória com uma matriz de incidência, por exemplo, isso tem um custo computacional elevado[5].

[5]Um grafo de cerca de 70000 vértices, é lido e montado numa estrutura dinâmica em um tempo cerca de 500 vezes mais lento do que numa estrutura estática.

6.6.2. Otimizando o código

A otimização diretamente no código pode ocorrer em dois níveis: funções e laços. Para a otimização de funções basicamente se usa o conceito de *function inlining*, isto é, em vez de se fazer uma chamada convencional da função se faz a substituição direta no código. Isso reduz a legibilidade do código, assim como aumenta o tamanho do programa na memória. Quanto ao aspecto de legibilidade, podemos fazer uso de macros da linguagem, em que uma macro é definida e a chamada do que seria a função é substituída pela chamada da macro, como vemos no código da figura 6.14.

```
1  #define average(x,y) ((x+y)/2)
2
3  main()
4  {float a, q=100, p=50;
5      a = average (p,q);
6      printf("%f\n",a);
7  }
8  //  average (p, q)  vai ser trocada por (p+q)/2
```

Figura 6.14. Alinhamento de função com o uso de macros da linguagem.

Laços de repetição, são, por sua vez, uma fonte bastante ampla para otimizações. Em laços podemos der seu desenrolamento, desdobramento, remoção de testes desnecessários, fusão ou mesmo fissão do laço.

Desenrolamento de laço Quando se tem conhecimento do tamanho do laço (ou pelo menos um mínimo de iterações) é possível trocar o laço pela repetição explícita de seu corpo. Por exemplo:

```
1  for (i=0; i<3; i++)        pode ser        a[0] = b[0] + c[0];
2    a[i] = b[i] + c[i];    trocado por      a[1] = b[1] + c[1];
3                                            a[2] = b[2] + c[2];
```

Remoção de testes desnecessários Muitas vezes incluímos testes em laços que podem ser removidos por vários motivos. Um deles pode envolver um teste para o qual, a menos por problemas bastante intensos de precisão, podemos considerar que o resultado será sempre o mesmo, eliminando o teste. Outra situação, mais comum, é ter um teste para o qual o resultado será sempre igual pois envolve alguma variável não modificada no laço, como vemos na figura 6.15.

A mesma técnica pode ser aplicada se o teste interno, em laços aninhados, envolver a comparação entre as variáveis de controle dos laços. Nesse caso o laço interno pode ser desdobrado em dois laços, com iterações definidas pelo resultado previsível da comparação.

```
1  for (i=0; i<n; i++)       pode ser     if (k != 0)
2    if (k != 0)           trocado por       for (i=0; i<n; i++)
3        a[i] = b[i] + k;                       a[i] = b[i] + k;
4      else                                   else
5        a[i] = c[i] + k;                       for (i=0; i<n; i++)
6                                                 a[i] = c[i] + k;
```

Figura 6.15. Teste desnecessário com desdobramento de laço.

inversão de laço em laços aninhados A otimização nesse caso depende da forma como os laços, que envolvam matrizes, são percorridos. Sabemos que matrizes são armazenadas de modo vetorizado, sendo que em C (entre outras) isso ocorre linha por linha, enquanto em Fortran ocorre coluna por coluna. Assim, percorrer a matriz é feito mais eficientemente quando ocorre na mesma ordem de armazenamento.

Desse modo, laços aninhados podem ser percorridos mais eficientemente se o laço interno percorrer a matriz na ordem ótima (colunas da mesma linha) e o externo na outra ordem (linhas), exigindo menos mudanças de *cache*. O problema é que nem sempre isso é possível, como é o caso da multiplicação de matrizes, em que uma matriz é percorrida linha a linha e a outra coluna a coluna (embora existam alternativas para contornar esse problema).

Fusão e fissão de laços Fusão de laço ocorre quando temos laços consecutivos que executarão exatamente o mesmo número de iterações. Desse modo, para evitar a duplicidade de controle, os corpos dos dois laços podem ser aglomerados em um único laço, caso sejam independentes.

Já a fissão de um laço em dois novos laços pode ocorrer para reduzir a necessidade de trocas de linhas de *cache*. Assim as iterações em cada laço são modificadas para, em C por exemplo, envolver uma menor quantidade de colunas por laço.

6.6.3. Redução de força

Determinados operadores demandam mais esforço computacional que outros, principalmente quando tratamos de operandos de ponto flutuante. Em algumas situações é possível reduzir esse esforço trocando uma operação por outra que demande menos força. Por exemplo, podemos trocar o operador de potenciação por sua execução explícita, como em:

Multiplicação	$(y*y*y)$ é mais rápida que potenciação $(pow(y,3))$
Soma	$(k+k+k)$ é mais rápida que multiplicação $(3*k)$

Referências

[1] STERLING, T.; ANDERSON, M.; BRODOWICZ, M. *High Performance Computing: Modern Systems and Practices*. EUA: Morgan Kaufmann, 2017. 718 p.

[2] MANACERO, A. *Predição do desempenho de programas paralelos por simulação do grafo de execução*. Tese (Doutorado) — University of Campinas, Brazil, 1997. Disponível em: <https://doi.org/10.47749/T/UNICAMP.1997.118682>.

[3] SAHNI, S.; THANVANTRI, V. Performance metrics: keeping the focus on runtime. *IEEE Parallel & Distributed Technology: Systems & Applications*, v. 4, n. 1, p. 43–56, 1996.

[4] JAIN, R. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling*. [S.l.]: Wiley, 1991. 685 p. (Wiley professional computing). ISBN 978-0-471-50336-1.

[5] AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. New York, NY, USA: Association for Computing Machinery, 1967. (AFIPS '67 (Spring)), p. 483–485. ISBN 9781450378956. Disponível em: <https://doi.org/10.1145/1465482.1465560>.

[6] GUSTAFSON, J. L. Reevaluating amdahl's law. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 31, n. 5, p. 532–533, may 1988. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/42411.42415>.

[7] LIN, Y. C.; SNYDER, L. *Principles of Parallel Programming*. Boston, Mass: Pearson/Addison Wesley, 2008. ISBN 978-0321487902. Disponível em: <http://www.amazon.com/Principles-Parallel-Programming-Calvin-Lin/dp/0321487907>.

[8] MILLER, B.; HOLLINGSWORTH, J. *Paradyn Tools Project*. 2024. Accessed: 2024-09-13. Disponível em: <https://www.paradyn.org/>.

[9] HOLLINGSWORTH, J.; MILLER, B.; CARGILLE, J. Dynamic program instrumentation for scalable performance tools. In: *Proceedings of IEEE Scalable High Performance Computing Conference*. [S.l.: s.n.], 1994. p. 841–850.

[10] TAY, Y. C. *Analytical Performance Modeling for Computer Systems*. Springer International Publishing, 2014. ISSN 1932-1686. ISBN 9783031018008. Disponível em: <http://dx.doi.org/10.1007/978-3-031-01800-8>.

[11] CHEN, K. *Performance Evaluation by Simulation and Analysis with Applications to Computer Networks*. [S.l.]: John Wiley & Sons, Ltd, 2015. 286 p. ISBN 9781119006190.

[12] SPEC. *Standard Performance Evaluation Corporation*. 2024. Accessed: 2024-09-13. Disponível em: <https://www.spec.org>.

[13] NASA Advanced Suporcomputing Division. *NAS Parallel Benchmarks*. 2024. Accessed: 2024-09-13. Disponível em: <https://www.nas.nasa.gov/software/npb.html>.

[14] GWT-TUD GmbH. *Vampir 10.5*. 2024. Accessed: 2024-09-13. Disponível em: <https://vampir.eu>.

[15] REINDERS, J. *Vtune performance analyzer essentials*. [S.l.]: Intel Press, 2007.

[16] PERFORMANCE RESEARCH LAB. *TAU - Tuning and Analysis Utilities*. 2006. `https://www.cs.uoregon.edu/research/tau/home.php`. Accesseda em Julho de 2024.