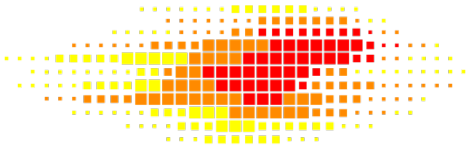


WSCAD 2023

Minicursos do WSCAD 2023

XXIV Simpósio em Sistemas Computacionais de Alto Desempenho
Porto Alegre – RS

17 a 20 de outubro de 2023

Porto Alegre
Sociedade Brasileira de Computação
2023

Minicursos do WSCAD 2023
XXIV Simpósio em Sistemas Computacionais de Alto Desempenho
Porto Alegre – RS
17 a 20 de outubro de 2023

Organizadores
Edward David Moreno Ordone
Kalinka Regina Lucas Jaquie Castelo Branco

ISBN: 978-85-7669-563-9

Porto Alegre
Sociedade Brasileira de Computação
2023

Dados Internacionais de Catalogação na Publicação (CIP)

S612 Simpósio em Sistemas Computacionais de Alto Desempenho (24. : 17 – 20 outubro 2023 : Porto Alegre)
Minicursos do WSCAD 2023 [recurso eletrônico] / organização: Edward David Moreno Ordone ; Kalinka Regina Lucas Jaquie Castelo Branco. Dados eletrônicos. – Porto Alegre: Sociedade Brasileira de Computação, 2023.
110 p. : il. : PDF ; 12 MB

Modo de acesso: World Wide Web.
Inclui bibliografia
ISBN 978-85-7669-563-9 (e-book)

1. Computação – Brasil – Evento. 2. Sistemas computacionais. 3. Processamento de dados. 4. Plataformas computacionais. 5. Armazenamento de dados. I. Ordone, Edward David Moreno. II. Branco, Kalinka Regina Lucas Jaquie Castelo. III. Sociedade Brasileira de Computação. IV. Título.

CDU 004(063)

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

Prefácio

Esta edição do Livro de Minicursos do WSCAD traz os seis minicursos apresentados durante o XXIV Simpósio em Sistemas Computacionais de Alto Desempenho, realizado entre os dias 17 e 20 de outubro de 2023 em Porto Alegre, RS. O primeiro capítulo versa sobre conceitos essenciais de processamento e análise de volumes massivos de dados, faz uso de algoritmos de aprendizado de máquina de forma prática. Já no segundo capítulo, é apresentada ao leitor a possibilidade de compreender o funcionamento do modelo de programação distribuída OpenMP por meio da introdução do OpenMP Cluster. No capítulo três, por sua vez, são explorados os desafios e soluções associados a aplicações complexas e que lidam com grande volume de dados, de modo que possam ser otimizadas melhorando o desempenho e o consumo de energia de aplicações paralelas. Soluções automatizadas que permitem a instalação de pacotes inteiramente em espaços de usuário de maneira reprodutível e compartilhável, reduzindo o custo e o erro por parte dos administradores de sistema, são apresentadas no capítulo quatro. No capítulo cinco tem como objetivo introduzir os conceitos de um sistema adaptativo/evolutivo, o Hardware Evolutivo (EHW), que apresenta um nível elevado de paralelismo. E por fim, no capítulo seis é discutido sobre sistema de armazenamento abordando as várias conexões, como SATA, SCSI, SAS, iSCSI e Fibre Channel, sendo destacadas as vantagens e desvantagens de cada uma dessas conexões.

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 WSCAD 2023, possam aproveitar e desfrutar dos conhecimentos aqui apresentados. Uma ótima leitura a todos.

Edward Moreno (UFS) e Kalinka Branco (ICMC-USP)

Coordenadores dos minicursos do WSCAD 2023

Capítulo

1

Processamento e Análise de *Big Data* e Aplicação de Algoritmos de *Machine Learning* através da utilização da Plataforma *HPCC Systems*

Mauro Donato Marques (LexisNexis Risk Solutions) e Alysson Rogerio Oliveira (LexisNexis Risk Solutions)

Abstract

Throughout the course, participants will have the opportunity to learn the essential concepts of massive data volume processing and analyzing (Big Data) and the process of developing a query service using the open-source platform High-Performance Computing Cluster (HPCC Systems) and also the use of Machine Learning algorithms, as well as having the possibility to apply the knowledge acquired in a training environment 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 fazendo uso da plataforma open-source composta por um Cluster Computacional de Alto Desempenho (HPCC Systems) e, também, a utilização de algoritmos de Aprendizado de Máquina, 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/WSCAD_2023

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 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.

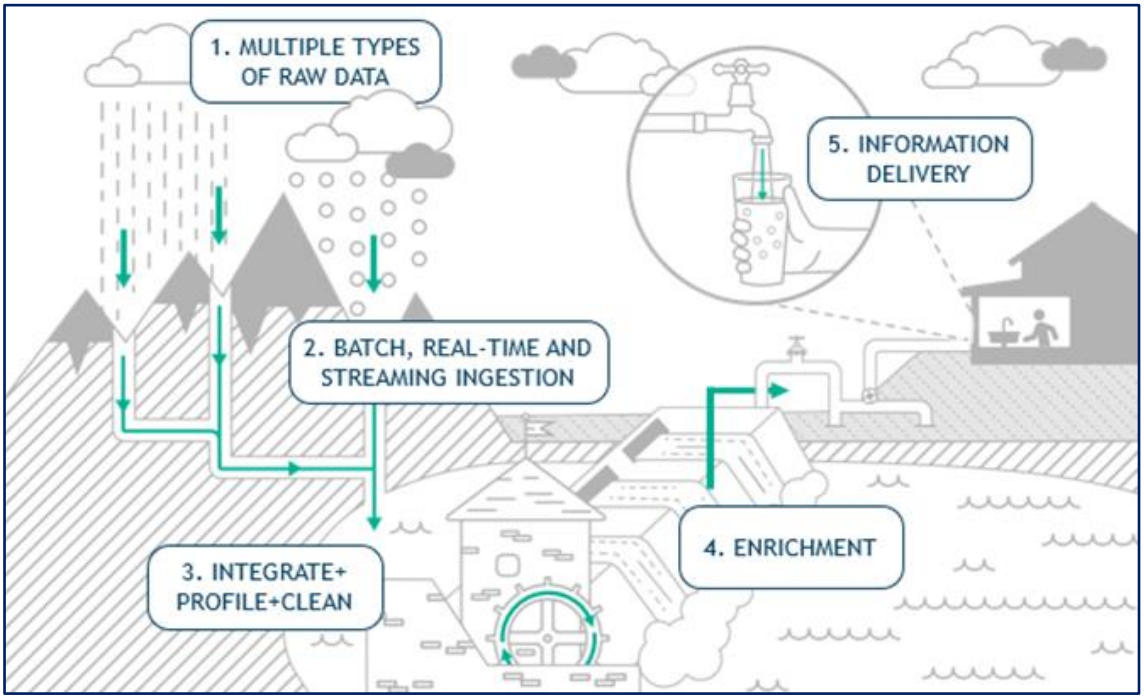


Figura 1.2: Um “Data Lake” deve ser capaz de ingerir, formatar e enriquecer dados com disponibilidade 24 horas por dia, 7 dias por semana.

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.

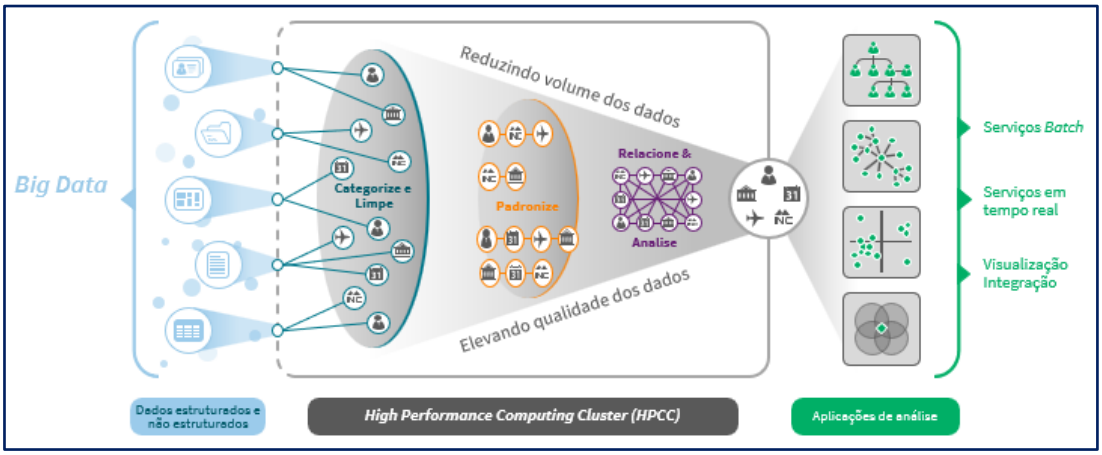


Figura 1.3: Fluxo de dados no HPCC Systems – “Funil” de dados.

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:

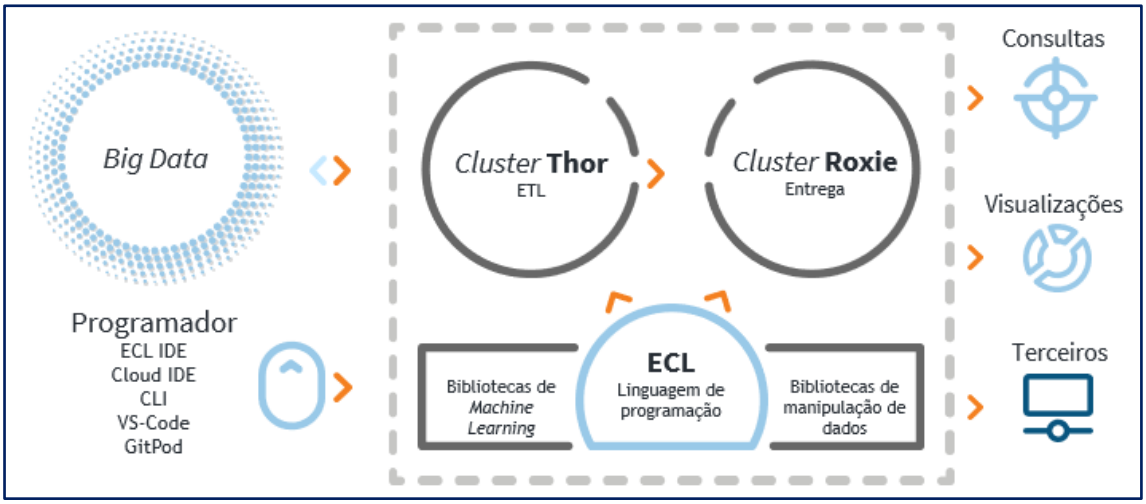


Figura 1.5: Clusters Thor e Roxie.

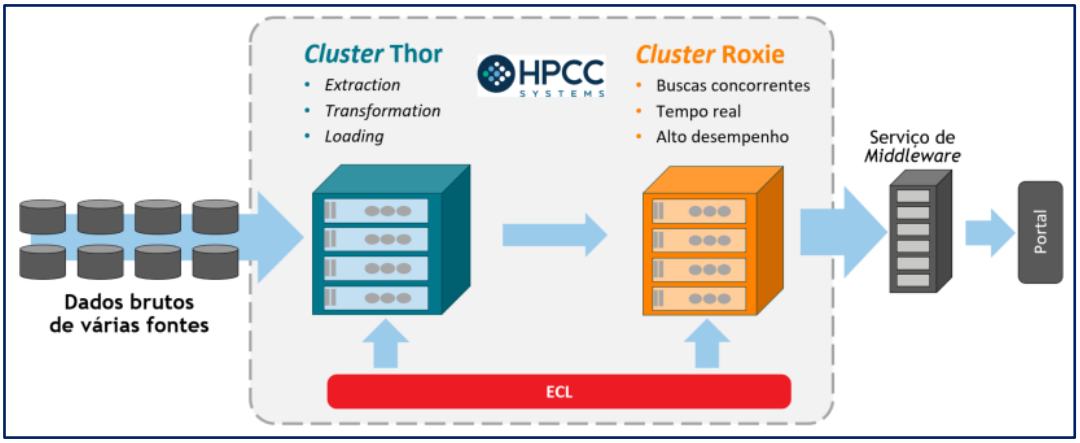


Figura 1.6: Power Trio: A) Thor, B) Roxie e C) ECL.

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.

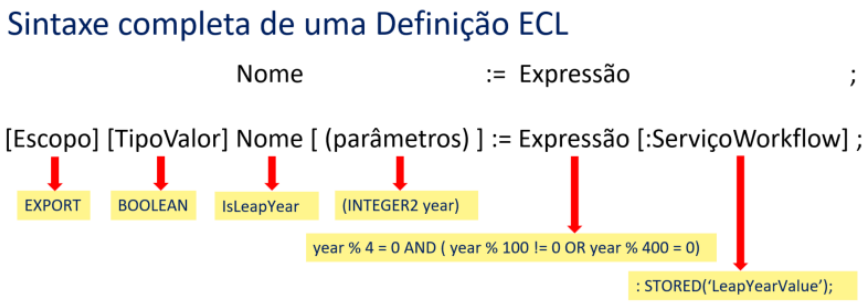
ECL é a única linguagem necessária para expressar algoritmos de dados em toda a plataforma HPCC Systems. No Thor, a linguagem ECL expressa "workflows" de dados que consistem em carregamento dados, transformação, vinculação, indexação etc. No Roxie, a linguagem ECL define consultas de dados. Isso significa 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
- Ações vs. Definições
  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.

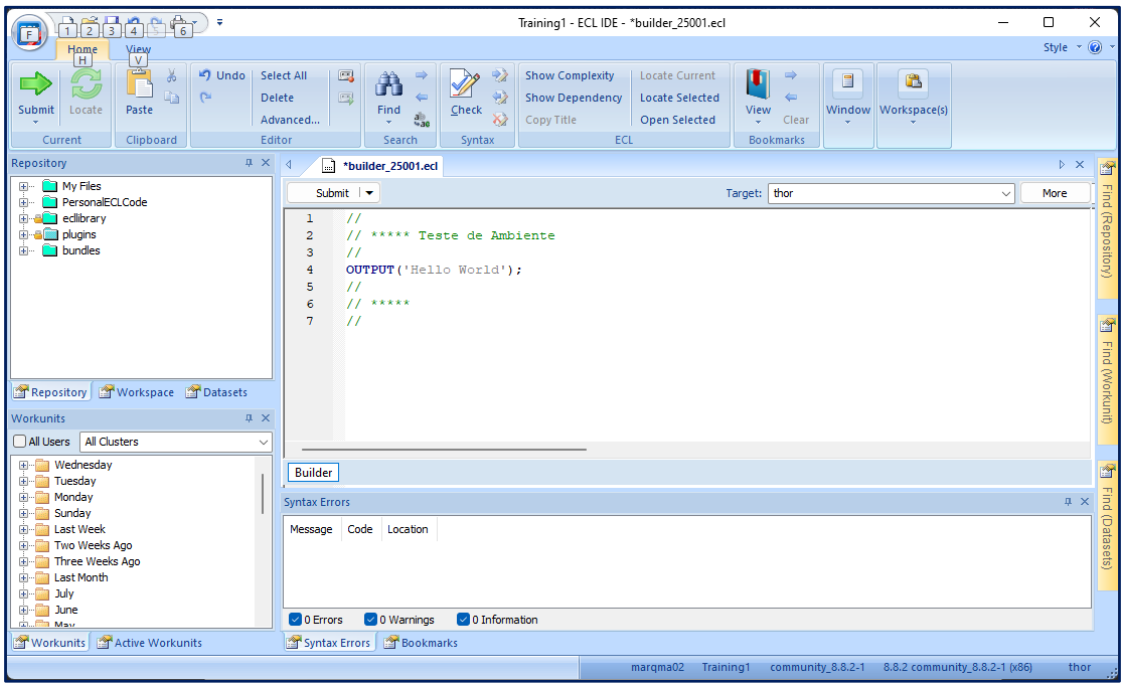


Figura 1.7: ECL Integrated Development Environment (IDE).

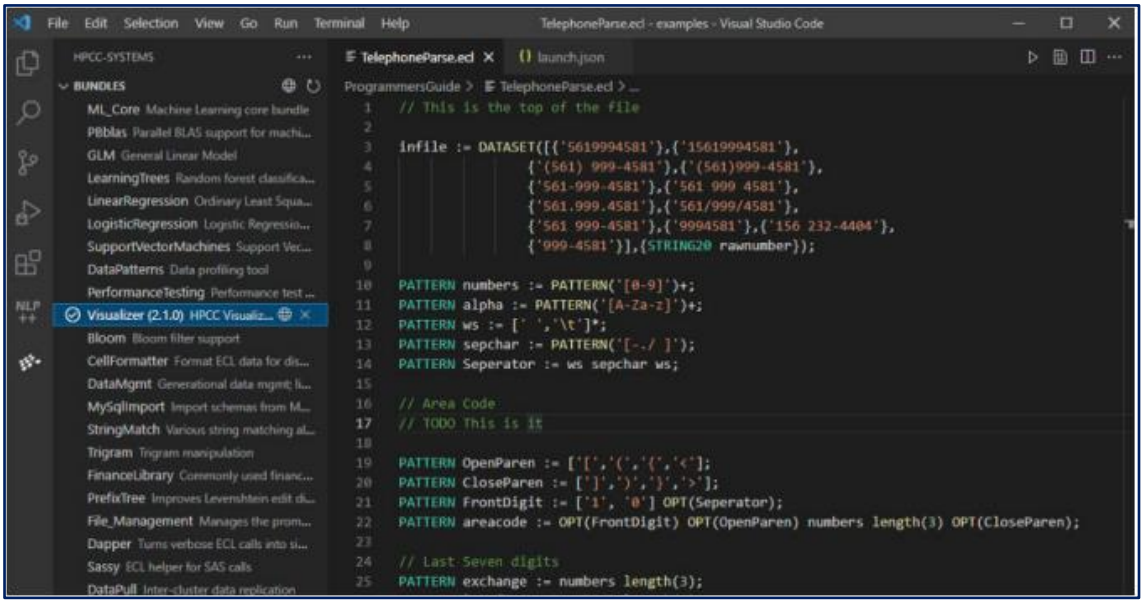


Figura 1.8: VS Code IDE.

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. Machine Learning bundles 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 qe não fornecerá bons resultados.

1.4. 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?

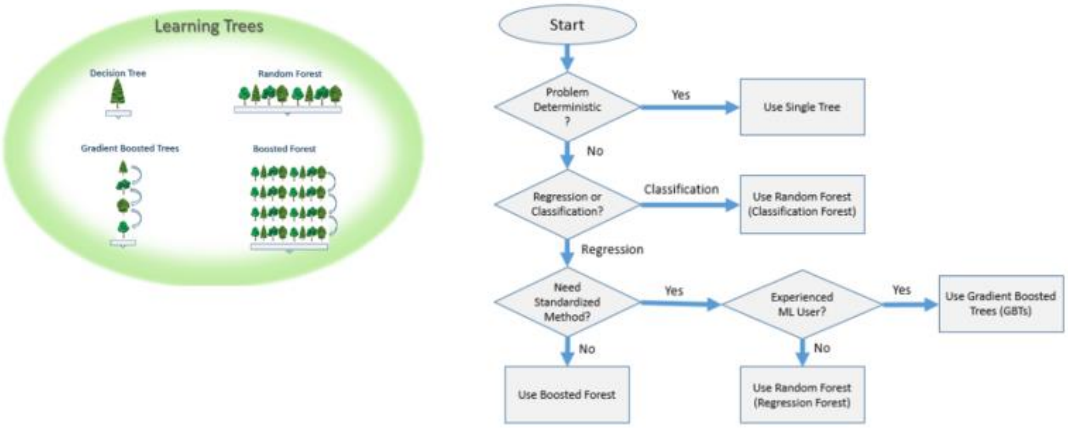


Figura 1.9: Fluxograma a ser usado para auxiliar nessa escolha.

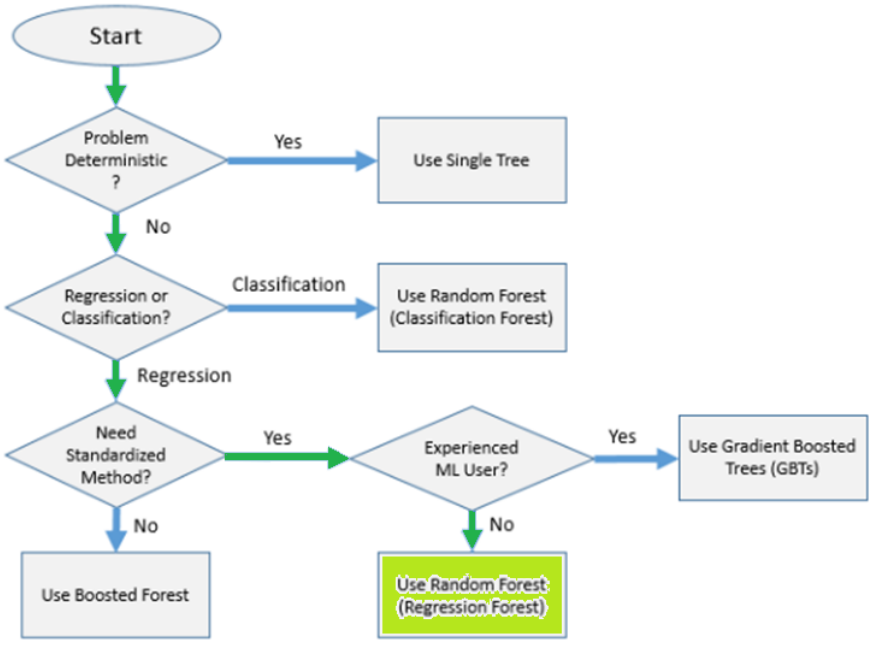


Figura 1.10: No caso em estudo "preço de imóveis" – Escolha: Random Forest - Regression Forest.

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:

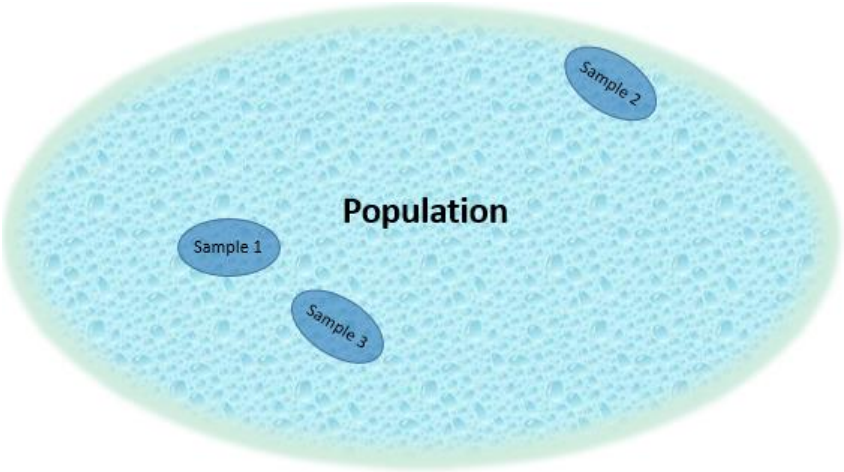


Figura 1.11: Generalização.

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.

1.4.1. O Fluxo de Aprendizado de Máquina Supervisionado

No caso em estudo 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.

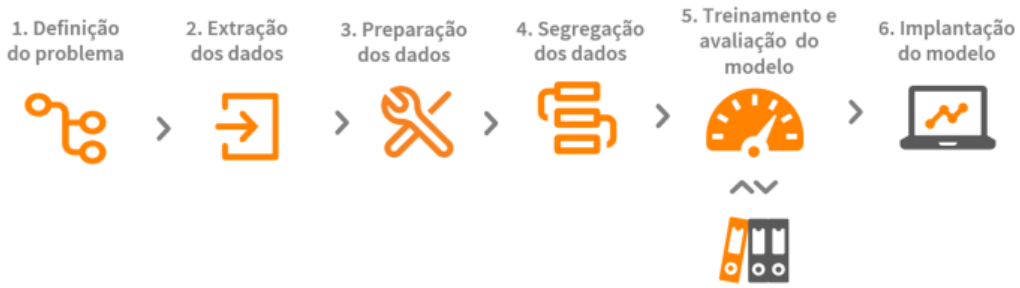


Figura 1.12: Fluxo de ML Supervisionado.

1.4.1.1. 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_numbe	house_nu	predir	street	streett	postdir	apt	city	state	zip	total_value	assessed_value	year_acquired	land_square_foot	living_square_fee	bedrooms	full_bath
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	61610	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			MYRTLE OAK	DR		000025	SAN BERNARDI	CA	92376	32750		2007	93654	0	0	0

Figura 1.13: Preço do imóvel – Valor real de venda (total_value).

1.4.1.2. 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

1.4.1.3. 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);
- 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 1.14: Limpeza, padronização e consolidação de registros.

1.4.1.4. 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
myTrainData := PROJECT(PrepDataSort[1..5000], $.modPrepData.ML_Prop)
                    :PERSIST('~CLASS::XYZ::ML::Train');
// Considerando os 2.000 registros seguintes como amostra de Teste
myTestData  := PROJECT(PrepDataSort[5001..7000], $.modPrepData.ML_Prop)
                    :PERSIST('~CLASS::XYZ::ML::Test');
```

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". O bundle configurável ML_Core facilita essa tarefa:

```
IMPORT ML_Core;
// Conversão Matricial dos campos numéricos
ML_Core.ToField(myTrainData, myTrainDataNF);
ML_Core.ToField(myTestData, myTestDataNF);
```

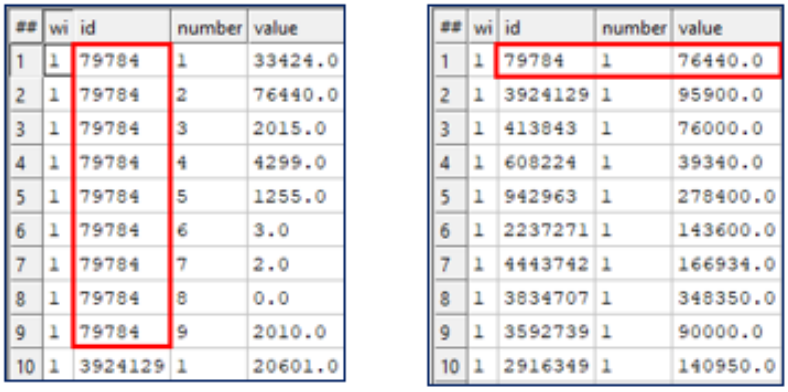
A etapa final antes de aplicar o modelo ML é separar os dados "independentes" dos dados "dependentes", definindo os dados do ML RECORD para colocar o campo de dados dependente no "final" do layout de registro. Um filtro simples nos dados de treinamento e de teste conclui esta etapa.

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 estará sendo 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 myIndTrainDataNF := myTrainDataNF(number < 10);
EXPORT myDepTrainDataNF := PROJECT(myTrainDataNF(number = 10),
TRANSFORM(RECORDOF(LEFT), SELF.number := 1, SELF := LEFT));
EXPORT myIndTestDataNF := myTestDataNF(number < 10);
EXPORT myDepTestDataNF := PROJECT(myTestDataNF(number = 10),
TRANSFORM(RECORDOF(LEFT), SELF.number := 1, SELF := LEFT));
```

- Códigos ECL utilizados:
  modSegConvData.ecl
  BWR_ViewData2.ecl



##	wi	id	number	value
1	1	79784	1	33424.0
2	1	79784	2	76440.0
3	1	79784	3	2015.0
4	1	79784	4	4299.0
5	1	79784	5	1255.0
6	1	79784	6	3.0
7	1	79784	7	2.0
8	1	79784	8	0.0
9	1	79784	9	2010.0
10	1	3924129	1	20601.0

##	wi	id	number	value
1	1	79784	1	76440.0
2	1	3924129	1	95900.0
3	1	413843	1	76000.0
4	1	608224	1	39340.0
5	1	942963	1	278400.0
6	1	2237271	1	143600.0
7	1	4443742	1	166934.0
8	1	3834707	1	348350.0
9	1	3592739	1	90000.0
10	1	2916349	1	140950.0

Figura 1.15: Nove (9) campos independentes e o último campo (10º) é o campo dependente.

1.4.1.5. 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=[]);
*/
myLearnerR := LT.RegressionForest();          // parâmetros default
```

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

```
myModelR := myLearnerR.GetModel($.modSegConvData.myIndTrainDataNF,
    $.modSegConvData.myDepTrainDataNF);
```

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:

```
predictedDeps := myLearnerR.Predict(myModelR,
$.modSegConvData.myIndTestDataNF);
```

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 [myDepTestDataNF], 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:

```
assessmentR := ML_Core.Analysis.Regression.Accuracy(predictedDeps,
$.modSegConvData.myDepTestDataNF);
```

A “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	1	1	0.7304899830671003	7982069594.129144	89342.4288573416

Figura 1.16: Métricas chaves.

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:

```
myLearnerR := 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 usando o código [CleanProperty].

- Códigos ECL utilizados:

  BWR_TrainReg.ecl

1.4.1.6. 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:

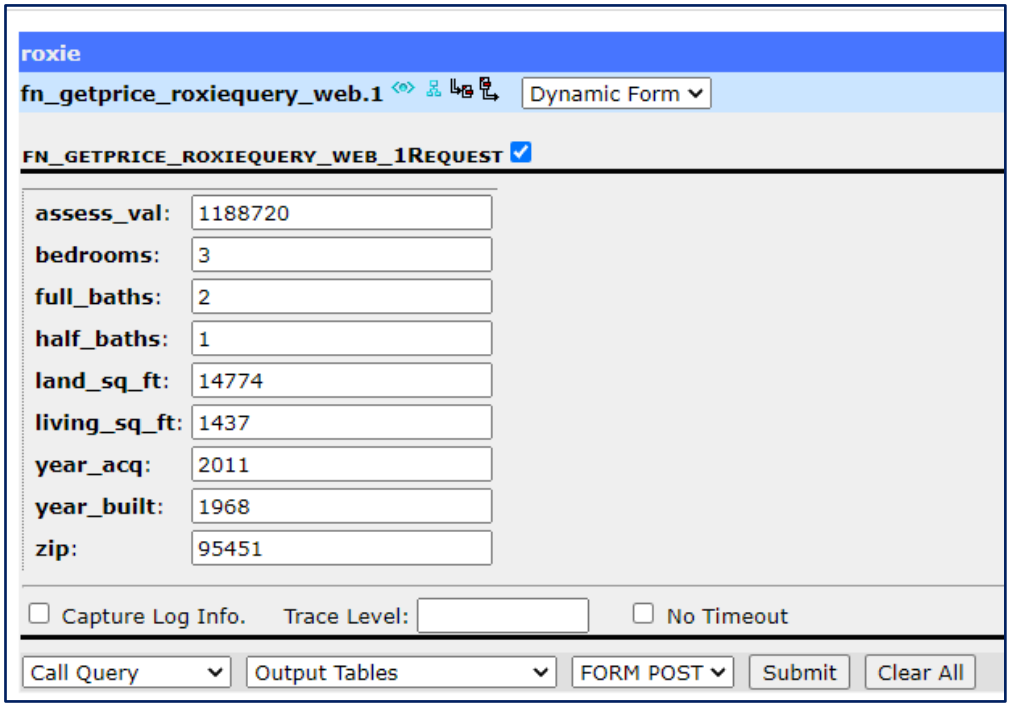


Figura 1.17: Carregamento de dados e disponibilização de serviço de consulta.

- Códigos ECL utilizados:

  fn_GetPriceReg.ecl
  BWR_TestQueriesReg.ecl

1.4.2. 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 e testar modelos de ML 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. Foi utilizado nesse estudo apenas os aspectos mais básicos dos bundles de ML.

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.

1.4.3. Apêndice

- RoadMap
  - Procedimento - Sequência de criação e execução (submit) 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_GetPriceReg.ecl (compilar em Roxie)
    BWR_ViewData3.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 := $.modProperty.File;
//
EXPORT modPrepData := MODULE
EXPORT ML_Prop := RECORD
UNSIGNED8 PropertyID;
UNSIGNED3 zip; // variável Categórica
UNSIGNED4 assessed_value;
UNSIGNED2 year_acquired;
UNSIGNED4 land_square_footage;
UNSIGNED4 living_square_feet;
UNSIGNED2 bedrooms;
UNSIGNED2 full_baths;
UNSIGNED2 half_baths;
UNSIGNED2 year_built;
UNSIGNED4 total_value; // variável Dependente (a ser determinada)
END;
//
  EXPORT ML_PropExt := RECORD(ML_Prop)
    UNSIGNED4 rnd;                // número randômico
  END;
//
EXPORT PrepData := PROJECT($.CleanProperty, TRANSFORM(ML_PropExt,
                                        SELF.rnd := RANDOM(),
                                        SELF.zip := (UNSIGNED3)LEFT.zip,
                                        SELF := LEFT))
                    :PERSIST('~CLASS::XYZ::ML::PrepData');
```

```
//
END;
//
```

[BWR_ViewData1.ecl]

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

[modSegConvData.ecl]

```
IMPORT $;
IMPORT ML_Core;
//
PrepData := $.modPrepData.PrepData;
//
// Torne os dados aleatórios, ordenando os registros pelo número randômico
PrepDataSort := SORT(PrepData, rnd);
//
//
// Segregação dos dados - Considerando os primeiros 5.000 registros como amostra de Treinamento
myTrainData := PROJECT(PrepDataSort[1..5000], $.modPrepData.ML_Prop)
                    :PERSIST('~CLASS::XYZ::ML::Train');            // layout sem o campo rnd
//
// Segregação dos dados - Considerando os 2.000 registros seguintes como amostra de Teste
myTestData  := PROJECT(PrepDataSort[5001..7000], $.modPrepData.ML_Prop)
                    :PERSIST('~CLASS::XYZ::ML::Test');             // layout sem o campo rnd
//
//
// Conversão Matricial dos campos numéricos
ML_Core.ToField(myTrainData, myTrainDataNF);
ML_Core.ToField(myTestData, myTestDataNF);
//
//
EXPORT modSegConvData := MODULE
EXPORT myIndTrainDataNF := myTrainDataNF(number < 10);    // excluindo o campo propertyid
//
EXPORT myDepTrainDataNF := PROJECT(myTrainDataNF(number = 10),
TRANSFORM(RECORDOF(LEFT),
                                            SELF.number := 1,
                                            SELF := LEFT));
//
EXPORT myIndTestDataNF := myTestDataNF(number < 10);      // excluindo o campo propertyid
//
EXPORT myDepTestDataNF := PROJECT(myTestDataNF(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($.modSegConvData.myIndTrainDataNF, NAMED('IndTrainData'));
OUTPUT($.modSegConvData.myDepTrainDataNF, NAMED('DepTrainData'));
OUTPUT($.modSegConvData.myIndTestDataNF,  NAMED('IndTestData'));
OUTPUT($.modSegConvData.myDepTestDataNF,  NAMED('DepTestData'));
//
```

[BWR_TrainReg.ecl]

```
IMPORT $;
IMPORT ML_Core;
IMPORT LearningTrees AS LT;
//
// Selecionando o algoritmo
//   Sintaxe: RegressionForest(UNSIGNED numTrees=100, UNSIGNED featuresPerNode=0,
//                             UNSIGNED maxDepth=100, SET OF UNSIGNED nominalFields=[])
// myLearnerR := LT.RegressionForest();          // parâmetros default
// myLearnerR := LT.RegressionForest(,,,[1]);    // zip não é realmente quantitativo, mas qualitativo
myLearnerR := LT.RegressionForest(10,,10,[1]);
//
//
// Obtendo o modelo treinado
myModelR :=
myLearnerR.GetModel($.modSegConvData.myIndTrainDataNF,$.modSegConvData.myDepTrainDataN
F);
OUTPUT(myModelR,,'~CLASS::XYZ::ML::myModelR', NAMED('Modelo_Treinado'),OVERWRITE);
//
//
// Testando o modelo
predictedDeps := myLearnerR.Predict(myModelR,$.modSegConvData.myIndTestDataNF);
OUTPUT(predictedDeps, NAMED('Valores_Previstos'));
//
//
// Avaliando o modelo
assessmentR :=
ML_Core.Analysis.Regression.Accuracy(predictedDeps,$.modSegConvData.myDepTestDataNF);
OUTPUT(assessmentR, NAMED('Avaliacao_do_Modelo'));
//
```

[fn_GetPriceReg.ecl]

```
IMPORT $;
IMPORT ML_Core;
IMPORT LearningTrees AS LT;
//
// Função de predição de preços de imóveis
EXPORT fn_GetPriceReg(Zip,
                      Assess_val,
                      Year_acq,
                      Land_sq_ft,
                      Living_sq_ft,
                      Bedrooms,
                      Full_baths,
                      Half_baths,
                      Year_built) := FUNCTION
//
// Transformação dos parâmetros de entrada no formato de ML data frame
  myInSet := [Zip, Assess_val, Year_acq, Land_sq_ft, Living_sq_ft, Bedrooms, Full_baths, Half_baths,
Year_built];
  myInDS := DATASET(myInSet, {REAL8 myInValue});
```

```
ML_Core.Types.NumericField PrepDataReg(RECORDOF(myInDS) Le, INTEGER cnt) :=
TRANSFORM
SELF.wi     := 1,
SELF.id     := 1,
SELF.number := cnt,
SELF.value  := Le.myInValue;
END;
myNewIndDataReg := PROJECT(myInDS, PrepDataReg(LEFT,COUNTER));
//
// Predição e retorno do valor do imóvel consultado
myModelR :=
DATASET('~CLASS::XYZ::ML::myModelR',ML_Core.Types.Layout_Model2,FLAT,PRELOAD);
myLearnerR := LT.RegressionForest(10,,10,[1]);
myPredictDeps := MyLearnerR.Predict(myModelR, myNewIndDataReg);
//
RETURN OUTPUT(myPredictDeps,{preco:=ROUND(value)});
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
//
$.fn_GetPriceReg(95451,118720,2011,14774,1437,3,2,1,1968);
//
/*
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
*/
//
```

1.5. 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.

Machine Learning Demystified. Disponível em: https://hpccsystems.com/resources/machine-learning-demystified/ Acesso em: 25 set. 2023.

HPCC Systems Machine Learning Library. Disponível em: https://hpccsystems.com/download/free-modules/hpcc-systems-machine-learning-library/ Acesso em: 25 set. 2023.

Using HPCC Systems Machine Learning. Disponível em: https://hpccsystems.com/resources/using-hpcc-systems-machine-learning/ Acesso em: 25 set. 2023.

ML_Core Documentation. Disponível em: https://cdn.hpccsystems.com/pdf/ml/ML_Core.pdf Acesso em: 25 set. 2023.

Source code: HPCC Systems ML_Core repository on GitHub. Disponível em: https://github.com/hpcc-systems/ML_Core Acesso em: 25 set. 2023.

Introduction to using PBblas on HPCC Systems. Disponível em: https://hpccsystems.com/resources/introduction-to-using-pbblas-on-hpcc-systems/ Acesso em: 25 set. 2023.

Documentation: PBblas Documentation. Disponível em: https://cdn.hpccsystems.com/pdf/ml/PBblas.pdf Acesso em: 25 set. 2023.

Source code: HPCC Systems PBblas repository on GitHub. Disponível em: https://github.com/hpcc-systems/PBblas Acesso em: 25 set. 2023.

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. 2023.

Documentation: LearningTrees Documentation. Disponível em: https://cdn.hpccsystems.com/pdf/ml/LearningTrees.pdf Acesso em: 25 set. 2023.

Source code: LearningTrees repository on GitHub. Disponível em: https://github.com/hpcc-systems/LearningTrees Acesso em: 25 set. 2023.

Chapter 2

Getting Up and Running with the OpenMP Cluster Programming Model

Emilio Francesquini (UFABC)
e.francesquini@ufabc.edu.br
http://lattes.cnpq.br/8949216028517727

Hervé Yviquel (UNICAMP)
hyviquel@unicamp.br
http://lattes.cnpq.br/3339703725728623

Marcio Pereira (UNICAMP)
mpereira@ic.unicamp.br
http://lattes.cnpq.br/2671525924181612

Sandro Rigo (UNICAMP)
srigo@unicamp.br
http://lattes.cnpq.br/8308517667746974

Guido Araújo (UNICAMP)
guido@unicamp.br
http://lattes.cnpq.br/8683914780987242

Abstract

In this text we present a short introduction to the new OpenMP Cluster (OMPC) distributed programming model. The OMPC runtime allows the programmer to annotate their code using OpenMP target offloading directives and run the application in a distributed environment seamlessly using a task-based programming model. OMPC is responsible for scheduling tasks to available nodes, transferring input/output data between nodes, and triggering remote execution all the while handling fault tolerance. The runtime leverages the LLVM infrastructure and is implemented using the well-known MPI library.

2.1. Introduction

The increasing utilization of supercomputers and data centers, particularly in the domain of scientific research, that employ heterogeneous architectures (with accelerators such as FPGAs and GPUs) has led to an escalation in the complexity of the program parallelization process [2, 4]. For instance, real-world high-performance applications frequently comprise a fusion of diverse technologies, such as MPI (Message Passing Interface), multi-threading (either pure, *e.g.* POSIX, or facilitated by libraries such as OpenMP), accelerator-specific programming languages (*e.g.*, CUDA, Verilog), and, occasionally, the inclusion of checkpointing libraries to provide a degree of Fault Tolerance (FT).

In response to this complexity, one of the most widely adopted approaches is the use of directives in the application code. These directives serve as guiding instructions both to the compiler and to the runtime environment, enabling them to fully leverage the available computational resources. Among these solutions, OpenMP [5] stands as a prominent example. Recent versions of the OpenMP standard allow programmers to annotate their code to enable task parallelism as well as to perform the computation on accelerators (in a process called *computation offloading*). Within OpenMP, the OpenMP Target Library library assumes a specialized role, streamlining the parallelization of applications to harness various types of accelerators, including GPUs and FPGAs. These accelerators are referred to as *devices*.

While computation offloading has been widely employed for executing computations on devices within the same computing node, the distribution of these tasks across several nodes of a cluster has commonly been a manual and intricate process. OpenMP Cluster project (OMPC) capitalizes on OpenMP's offloading capabilities and introduces the concept of a "remote device". This concept allows for the offloading of computations to remote nodes. OMPC effectively conceals the underlying communication, which is MPI-based, behind OpenMP task dependencies. Importantly, OMPC adheres to the OpenMP standard, affording application developers the opportunity to harness intra- and inter-node parallelism using a unified set of tools.

In this text[1] we present OMPC [2], an OpenMP task parallelism-based execution model for clusters. OMPC allows for the offloading of complex scientific tasks across HPC cluster nodes in a transparent and balanced way. OMPC has some interesting features such (a) an underlying MPI communication layer for inter-node communication; (b) an event handler that transparently offloads tasks and data to cluster nodes; (c) a cluster-wide HEFT-based task scheduler [7] that balances the workload distribution across

[1]This text results from the compilation of the authors' prior papers and documentation. At the beginning of each section, we provide references enabling readers to access more comprehensive information if they desire further details.

[2]OMPC is available at `https://ompcluster.gitlab.io/`

the cluster; (d) a transparent fault-tolerant mechanism. All these features are provided transparently to the programmer that only interfaces with the OpenMP programming model. OMPC has been successfully used for the development of applications ranging from benchmarks (*e.g.*, TaskBench) to real world applications (*e.g.*, seismic applications for oil/gas reservoir detection, high energy plasma simulations,...). More information and details about the inner workings of OMPC can be seen on the papers [8, 3].

2.2. OpenMP Cluster (OMPC)[3]

OMPC builds upon the OpenMP Target Library which was created to allow the offloading of computation to accelerators, also called devices. Devices can be, for example, GPUs or FPGAs in a single computing node. Using this library, OpenMP users can offload computation directly to the accelerators simply using the OpenMP Target directives. These target directives are very similar to an existing OpenMP feature called *task*, both of which we now explain in further details.

2.2.1. OpenMP's `task` and `target` directives

OpenMP 3.0 introduced the concept of tasks, which enable a higher level of parallelism by allowing code fragments annotated with the `task` directive to execute asynchronously. These annotated code segments are treated as individual tasks and are dispatched to the OpenMP runtime by the *control thread*. Dependencies between tasks are specified using the `depend` clause, which defines both the input variables that a task relies on and the output variables it modifies. This arrangement effectively creates a task graph in which dependencies between tasks are represented as arcs between the nodes (tasks). Once a task's dependencies are satisfied, it is added to a *ready queue* by the OpenMP runtime, from which a pool of *worker threads* can draw tasks for execution. The management of task dependencies, data handling, and thread creation and synchronization are among the core responsibilities of the OpenMP runtime.

Originally, the OpenMP `task` directive was designed with multicore architectures in mind. However, as acceleration devices like GPUs and FPGAs gained prominence, OpenMP evolved to support these devices through the "OpenMP accelerator model" (introduced in OpenMP 4.X). This expansion introduced the `target` directive, which shares similarities with the `task` directive but is intended for offloading computations to acceleration devices instead of CPU cores. It utilizes clauses like `depend` to define dependencies, and it employs the `map` clause to specify the data transfer direction between the host and the accelerator (e.g., `to`, `from`, and `tofrom`). Additionally, the `nowait` clause was introduced to indicate that the `target` directive is non-blocking, allowing it to execute asynchronously on the accelerator.

[3]This section was based on the paper [8], where you can find more details about OMPC inner workings.

```
1 #pragma omp target enter data map(to: A[:N]) nowait depend(out: *A)
2 #pragma omp target nowait depend(inout: *A)
3 foo(A)
4 #pragma omp target nowait depend(inout: *A)
5 bar(A)
6 #pragma omp target exit data map(release: A[:N]) nowait depend(out:
  ↪ *A)
```

Listing 1: OpenMP target tasks

In this scheme, `target` directives are treated as tasks, allowing us to represent the program execution as a task graph. OMPC expands the concept of devices to include nodes of a cluster, enabling the distribution of computations, such as the one involving `foo`, across any node in the cluster. This distribution occurs seamlessly while at the same time maintaining the code identical to the one used for a single node.

As an example, let's examine the code snippet Listing 1. Within this code, we encounter two tasks, namely, `foo` and `bar`, both marked with the `target` directive from OpenMP. This directive, in accordance with OpenMP's specifications [1], enables these tasks to be offloaded for execution on accelerators, such as GPUs.

In this context, what occurs is that the code and associated data for tasks `foo` and `bar` are dispatched to an accelerator for processing. However, it's essential to note that the host machine, which is responsible for executing the code found in Listing 1, also plays a role in managing the data movement required to satisfy the dependencies between these tasks. This means that the host orchestrates the necessary data transfers to ensure that `foo` and `bar` can execute efficiently on the accelerator while meeting their dependencies.

Let's delve into the semantics of this code to gain a better understanding. Starting with line 1, we encounter a critical operation: the vector `A` is transferred from the host memory to the accelerator memory. This process is referred to as *offloading*. Moving on to lines 2-3, we see that these lines play a crucial role in executing the `foo` task on the accelerator. The code of `foo` is dispatched to the accelerator, where it is executed. The result of this execution is stored directly on `A`. Subsequently, `A` is brought back to the host memory. To highlight the importance of this data movement, the `depend(inout: *A)` clause is employed, indicating that `A` is both read and written by the `foo` task. Lines 4-5 mirror a similar computation, this time involving the `bar` task. In line 4, the `target` directive assigns `bar` to an accelerator while specifying that it will read `A`, which was previously written by `foo` and resides in the host memory. Like before, the runtime orchestrates the process: it reads `A` from host memory, transfers it to the accelerator assigned to `bar`, executes the `bar` task, stores the result in accelerator memory, and

finally returns `A` to host memory. Line 6 marks the end of the execution. Notably, all the `target` directives in lines 2-4 include a `nowait` clause, signifying that both `foo` and `bar` are executed asynchronously. Consequently, it is the responsibility of OpenMP runtime to ensure that all `depend` clauses specified in lines 2-4 are satisfied, in accordance with the programmer's directives.

2.2.2. OMPC/OpenMP integration

The use of OpenMP directives makes for a streamlined approach to application parallelization on large computing clusters. OMPC makes use of OpenMP Target Library , which consists of two distinct layers: the "Plugin" layer and the "Agnostic" layer.

The "Plugin" layer comprises various plugin implementations, with each plugin specializing in offloading computations to specific accelerators. For instance, a CUDA plugin provides the code to allow offloading to GPUs. On the other hand, the "Agnostic" layer encompasses the generic aspects of OpenMP Target Library , including program and task execution management, as well as data handling.

Figure 2.1 illustrates the operational framework of OpenMP Target Library. Starting from the user program, device and host code are generated and encapsulated within a fat binary when compiling for a chosen device. This is done using the Clang compiler from LLVM, which contains the OpenMP Target Library implementation. This executable then utilizes the generated dynamic libraries to offload computations to the device specified during compilation. The components introduced by OMPC to OpenMP Target Library are highlighted in green in Figure 2.1.

The OMPC plugin encompasses a few critical elements: MPI, Event System, and Fault Tolerance components, which collectively form the core of OMPC. These components enable the system to distribute instructions and data between nodes and allow the offloading of computations across multiple nodes within a cluster. Notably, all communication is accomplished using MPI.

In the agnostic layer, OMPC introduces the scheduling of tasks through the High-Performance Earliest Finish Time (HEFT) scheduler [7]. Through an evaluation of the task graph, OMPC can optimally determine the node for executing each task. Furthermore, OMPC incorporates a Data Manager component, which plays a pivotal role in orchestrating data movements across nodes. This component bears the responsibility for optimizing inter-node communications, effectively preventing unnecessary data transfers and thereby improving the overall communication efficiency within the cluster.

2.2.3. The OMPC Programming Model

This section outlines the key distinctions between the proposed OpenMP Cluster (OMPC) model and the OpenMP Accelerator Model. OMPC was purposefully designed to seam-

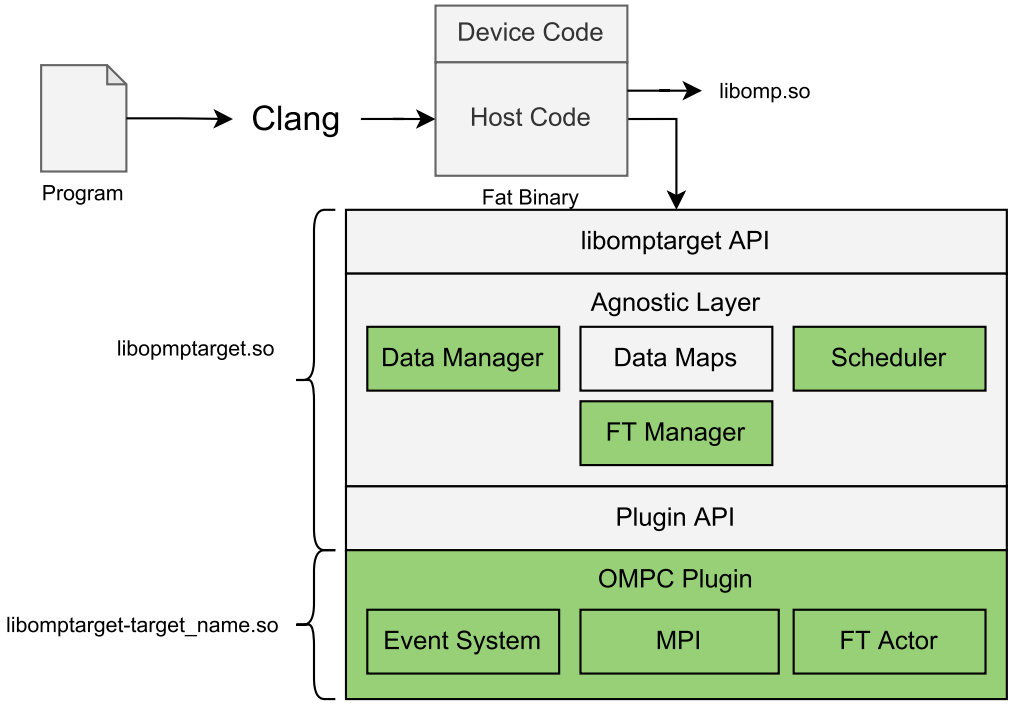


Figure 2.1. Model for OpenMP Target Library . In green, we display the OMPC additions to the library [8].

lessly extend OpenMP semantics to cluster environments. To achieve this, a straightforward abstraction was employed, expanding the concept of "cores" in OpenMP to "nodes" in OMPC. To illustrate this concept further, consider the following use cases:

1. In OpenMP, a `target` directive assigns a task to an accelerator on the same machine, while in OMPC, the task is assigned to a cluster node.

2. While in OpenMP, the `depend` clause handles data movement between host and accelerator memories (both on the same machine), in OMPC, the `depend` clause utilizes underlying MPI calls to make the data transfers between cluster nodes.

Overall, by recognizing that a core in OpenMP corresponds to a node in OMPC, it becomes clear that the foundational semantics of the OpenMP specification still apply in the OMPC context. For instance, the OpenMP code provided in Listing 1 remains unchanged when executed on a cluster using the OMPC runtime. In this scenario, tasks `foo` and `bar` are assigned to cluster nodes, and vector `A` is seamlessly transferred between nodes using OMPC's efficient implementation of the depend clause, which leverages MPI calls to efficiently move `A` from `foo` to `bar`.

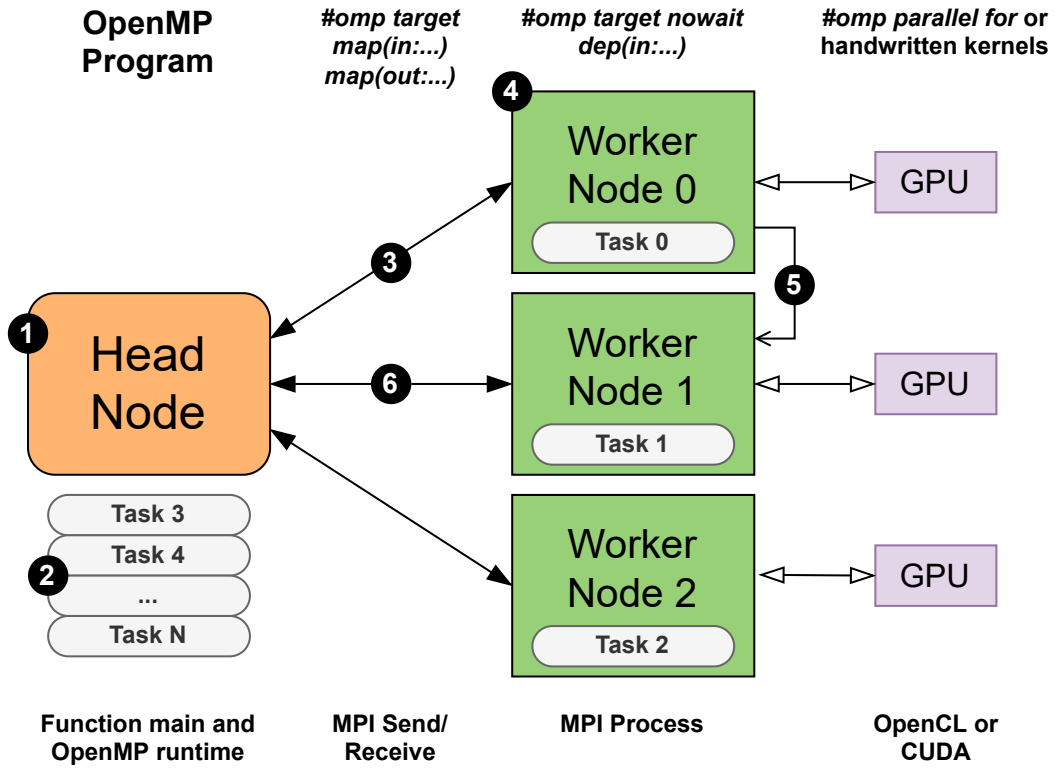


Figure 2.2. Execution Model of the Heterogeneous Cluster Device using OpenMP

Execution model As depicted in Figure 2.2, a standard cluster setup comprises a *head node* responsible for executing the OMPC runtime and a collection of *worker nodes*. This workflow encapsulates the operation of code execution within the cluster, balancing tasks across nodes, and managing data exchanges, all orchestrated by the OMPC runtime. The execution workflow for the annotated code within the cluster unfolds as follows:

1. The user initiates their program from the head node.

2. When the OpenMP kernel is encountered, the OpenMP runtime generates tasks automatically (without executing them) and places them into a dedicated pool, where they await execution. For clarity, let us assume that tasks `foo` and `bar` from the code in Listing 1 correspond to "Task 0" and "Task 1" in Figure 2.2.

3. The OMPC runtime distributes tasks, for example, `foo`, along with their input data (*e.g.*, vector `A`), to be executed on specific worker nodes (*e.g.*, `foo` on "Node 0").

This distribution leverages calls to the underlying MPI subsystem and adheres to a scheduling strategy, such as HEFT, to ensure a balanced computational workload.

4. Worker nodes process the received data.

5. The OMPC runtime forwards the results (*i.e.*, the output of `foo`) to tasks dependent on them (e.g., `bar` at *Node 1*). This transfer is accomplished using MPI calls. Subsequently, the task is removed from the dependency graph, and dependencies are updated. On the head node side, worker node tracking is managed through the dependency graph, following as described by the OpenMP specification.

6. To conclude the computation, OMPC retrieves vector `A` and places it back in the head node.

A crucial aspect to emphasize is the versatility of the code offloaded to the nodes. The code within `foo` or `bar` represents regular code that can potentially harness a second level of parallelism. For instance, if `foo` were to contain a loop annotated with a `parallel for` directive, it could still benefit from OpenMP parallelism within the node. Moreover, `foo` or `bar` could also be written in languages like OpenCL or CUDA, following the practices typically employed in distributed clusters. As previously mentioned, OMPC was intentionally designed to seamlessly integrate with the standard OpenMP runtime.

Furthermore, OMPC was developed with fault tolerance in mind. To facilitate this, each node in OMPC, including both the head node and worker nodes, features a heartbeat mechanism arranged in a ring topology. This arrangement enables nodes to monitor the status of their neighbors. Consequently, if a node experiences a failure, the system promptly detects it and initiates the process of restarting the affected tasks. The implementation of fault tolerance on OMPC is underway and will be released in a future version.

2.3. Using OMPC

In this section, we show how to setup the environment up and get up and running with OMPC[4].

2.3.1. Compilation and execution

To compile OpenMP code for OmpCluster, you need to specify an OpenMP target, denoted as `x86_64-pc-linux-gnu` to instruct the compiler to compile the OpenMP

[4]This section and the next are revised versions of the public documentation of OMPC, written by the authors and the OMPC Team (available at: `https://ompcluster.readthedocs.io`).

target code region for a particular device. For instance, you can compile the `mat-mul` example using the following command:

```
clang -fopenmp -fopenmp-targets=x86_64-pc-linux-gnu \
      mat-mul.cpp -o mat-mul
```

Listing 2: Compiling an application with OMPC.

Then, the newly generated program can be executed. However, unlike conventional OpenMP programs, OmpCluster programs require to be executed using MPI. To that end, tools such as `mpirun` and `mpiexec` should be employed. This is necessary so that the OMPC's distributed runtime system is able to use the configured (for MPI) infrastructure. The command line should be like:

```
mpirun -np 3 ./mat-mul
```

Listing 3: Executing an OMPC application with MPI.

In the example provided in Listing 3, the runtime will automatically generate three MPI processes: one *head process* and two *worker processes*. The head process is responsible for offloading OpenMP target regions, which are then executed on the worker processes, following the currently implemented scheduling strategy.

The runtime also supports offloading to remote MPI processes (located on different computers or containers). These remote configurations can be established using the `-host` or `-hostfile` available on the `mpirun` command (please note that specific flag names may vary among different MPI implementations). However, similar to any MPI program, it is essential for the user to ensure that the binary executable is copied to all relevant computers or containers before execution. This can be accomplished using commands such as `pdcp` or by employing an NFS (Network File System) directory for seamless access across the networked nodes.

Execution logs are available using the flag `LIBOMPTARGET_INFO`, set as an environment variable. The execution in this case will look like this:

```
LIBOMPTARGET_INFO=-1 mpirun -np 3 ./mat-mul
```

Listing 4: Executing an OMPC application with MPI.

2.3.2. Containerized images

You do not have to go through the process of downloading and compiling the most recent OMPC version from the official website at `https://ompcluster.gitlab.io/`. Instead, we strongly encourage you to opt for pre-compiled Docker images. These images come equipped with Clang/LLVM, along with all the essential OpenMP and MPI libraries required to run OMPC programs seamlessly. You can conveniently access these pre-compiled images at `https://hub.docker.com/r/ompcluster/`. This approach simplifies your setup process and ensures that you have all the necessary tools readily available.

All images are built based on Ubuntu 20.04. However, we offer various configurations with different CUDA versions, as well as the choice of MPICH or OpenMPI. You can select the appropriate Docker image tag that matches your preferred configuration, or simply use *latest* to use the default setup.

The container images adhere to the following naming convention:

```
ompcluster/<image_name>:<tag>
```

In our Docker Hub repository, you can find several available images. We list below a few of the most used:

hpcbase This serves as the base image for all other containers. It includes the MPI implementation, CUDA, Mellanox drivers, etc.

runtime This image contains pre-built Clang and the stable OMPC runtime based on stable releases.

runtime-dev This image also contains pre-built Clang and the OmpCluster runtime but is sourced directly from the Git repository. Please note that this version is considered unstable and should not be used in production.

Application-specific These images (`awave-dev`, `beso-dev`, `plasma-dev`, etc) are built on the runtime image and incorporate additional libraries and tools necessary for the development of specific applications.

Once you've selected the most suitable image for your needs, you can run the application within the container using the following method:

```
docker run -v /path/to/my_program/:/root/my_program \
           -it ompcluster/runtime:latest /bin/bash
cd /root/my_program/
```

Listing 5: Executing an OMPC application within a Docker container.

The flag `-v` is used to share a folder between the operating system of the host and the container. You can get more information on how to use Docker in the official *Get Started* guide (`https://docs.docker.com/get-started/`).

Running on Singularity is also supported as seen below:

```
singularity pull docker://ompcluster/runtime:latest
singularity shell ./runtime_latest.sif
cd /path/to/my_program/
```

Listing 6: Executing an OMPC application within a Singularity container.

For additional information, please consult the Singularity Documentation at `https://sylabs.io/guides/3.2/user-guide/`. It is worth noting that certain cluster environments may adopt the newer Singularity version known as Apptainer, which you can learn more about at `http://apptainer.org/docs/user/latest/`.

2.3.3. Slurm

You can seamlessly integrate the OmpCluster runtime with a cluster job manager, such as Slurm. Once you've compiled your code within a container, launching the job becomes as straightforward as running any MPI program. Here's an example of how to do it using Slurm:

```
srun -N 3 --mpi=pmi2 singularity exec ./runtime_latest.sif
 ↪  ./my_program
```

Listing 7: Executing an OMPC application within a Singularity container.

Please refer to Slurm Documentation (`https://slurm.schedmd.com/quickstart.html`) for more information.

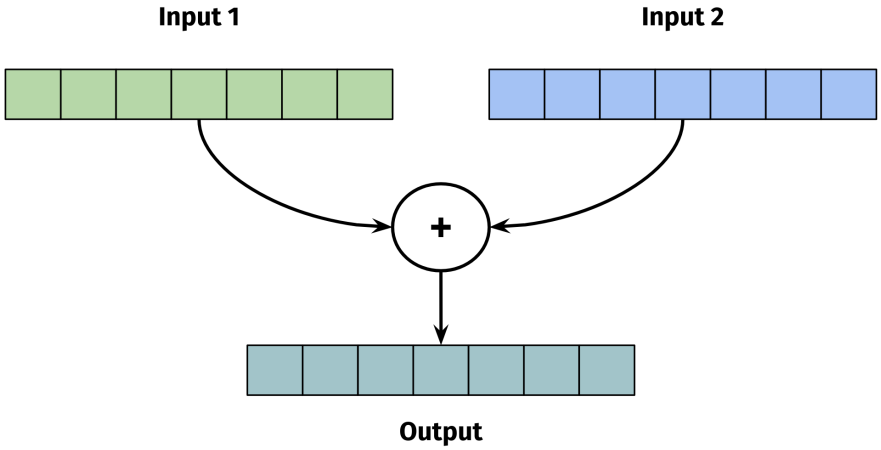


Figure 2.3. Vector sum.

2.4. Usage examples

In this section we present three applications using OMPC. The first is a trivial vector addition, the second is a vector reduction and the third is a matrix multiplication. We start with the sequential trivial implementation and improve on this implementation until we have an optimized OMPC version. For additional examples of code using OMPC see the examples page located at: `https://gitlab.com/ompcluster/ompc-examples`.

2.4.1. Example 1: Vector Addition

In this first example we want to sum two vectors, element wise. In other words given two vectors `A` and `B` of size `N`, we want to calculate a vector `C`, also of size `N` such that `C[i] = A[i] + B[i]`, for $0 \leq i \leq n$. Figure 2.3 illustrates this process.

A trivial implementation of vector addition in C++ can be seen in the code snippet below:

```
1 void vadd(int *A, int *B, int *C, size_t N) {
2   for (size_t i = 0; i < N; i++){
3     C[i] = A[i] + B[i];
4   }
5 }
```

Listing 8: Simple vector addition in C++.

Since we want to parallelize this code, we need some way of dividing the pro-

cessing in parts to be executed by each available computing node. To do that, we split the input vectors in chunks or blocks, and then, using OMPC, distribute these blocks. Listing 9 shows the code used to do that.

```
1  void vadd(int *A, int *B, int *C, size_t N) {
2    for (size_t i = 0; i < N; i++) {
3      C[i] = A[i] + B[i];
4    }
5  }
6
7  void blocked_vadd(int *in1, int *in2, int *out, int N, int BS) {
8    for(int i = 0; i < N / BS; i++) {
9      int *A  = &in1[BS * i], *B  = &in2[BS * i], *C  = &out[BS * i];
10     #pragma omp target nowait           \
11             map(to: A[:BS], B[:BS])     \
12             map(from: C[:BS])           \
13             depend(in: A[0], B[0])      \
14             depend(out: C[0])
15     vadd(A, B, C, BS);
16   }
17   #pragma omp taskwait
18 }
```

Listing 9: Blocked vector addition in C++.

The `vadd` function remains the same. However, we now also have the function `blocked_vadd`. This function breaks the vectors in blocks of size `BS`, given as input, and distributes these blocks throught each available computing node. `N / BS` tasks (and blocks) are created by the `for` on Line 8 (this function assumes `N` is a multiple of `BS`). Then on Line 9, local variables `A`, `B` and `C` are created. These are nothing more than new pointers/references to the original vectors (`in1`, `in2`, `out`), taking into account the task number and the block size. These blocks are then distributed to the working nodes according to the pragma on Lines 10-14. On Line 11 we map `BS` elements from `A` and `B` to the work node (using `to:`), and on Line 12 we copy the result back from the work nodes to the head node (using `from:`). Note the pragma `omp taskwait` on Line 17. This pragma is needed because since the tasks are executing asynchronously, we have to explicitly wait for them to finish before returning from the function.

It is also possible to use an accelerator on the remote node, and let OMPC distribute the work. For instance, one could use FPGAs to perform the same vector addition we just implemented. The support for FPGAs in OMPC is still in an early stage of development but, it already works and will be released as full implemantation in a future version of OMPC.

To do so, we use the OpenMPs `variant` pragma to annotate a function that contains an alternative implementation of the kernel. This first step is show below:

```
1 // FPGA prototype
2 void vadd_hw(int *in1, int *in2, int *out, unsigned int num);
3
4 // CPU prototype
5 #pragma omp declare variant( vadd_hw ) match( device={arch(alveo)} )
6 void vadd(int *in1, int *in2, int *out, unsigned int num);
```

Listing 10: OpenMP `variant` pragma.

The pragma on Line 5 states that if the device being used for offloading matches `alveo`[5], then the variant implementation (`vadd_hw`) should be used instead of the regular implementation of `vadd`. The code for execution on the FPGA is straightforward:

```
1 // HLS CODE
2 void vadd_hw(int *in1, int *in2, int *out, unsigned int num) {
3 #pragma HLS INTERFACE m_axi port=a bundle=gmem0
4 #pragma HLS INTERFACE m_axi port=b bundle=gmem1
5 #pragma HLS INTERFACE m_axi port=c bundle=gmem0
6 for (int i = 0; i < num; i++)
7   out[i] = in1[i] + in2[i];
8 }
```

Listing 11: Vector addition, FPGA version.

To make this work, one needs to write the kernel code in C, then add HLS directives and, finally, compile it into a xclbin file. To compile the OMPC code, we change the target to `alveo`), but the execution command line remains exactly the same:

```
1 $ clang++ -fopenmp -fopenmp-targets=alveo -fno-openmp-new-driver
   ↪  vadd.cpp -o vadd
2 $ mpirun -np $(N_PROCESSES) ./main
```

Listing 12: Executing an application using OpenMP `variant` pragma.

[5] `alveo` is the commercial name for a series of adatable accelerator cards (FPGAs) from Xilinx.

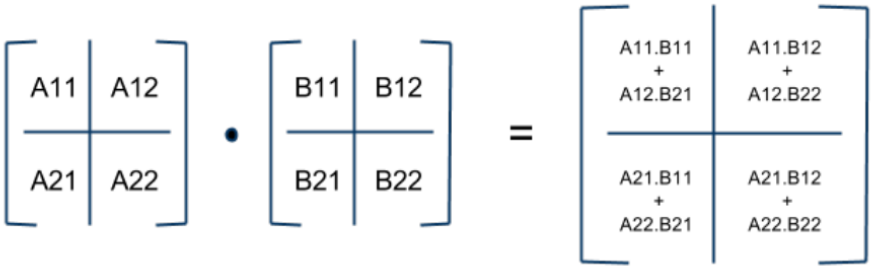


Figure 2.4. Block matrix multiplication

2.4.2. Example 3: Matrix Multiplication

To gain a clearer insight into this model, let's illustrate it with an example involving block-based matrix multiplication. In this scenario, we have two matrices, and our program's objective is to multiply them, resulting in a new matrix as the output.

Given two input matrices, `A` and `B`, each of size `N`, we would like to calculate `C` such that `C = A x B`. A first, naive, implementation in C++ would look like this:

```
1 void MatMul(int &A, int &B, int &C) {
2   for (int i = 0; i < N; ++i)
3     for (int j = 0; j < N ; ++j) {
4       C[i][j] = 0;
5       for (int k = 0; k < N ; ++k)
6         C[i][j] += A[i][k] * B[k][j];
7     }
8 }
```

Listing 13: Simple matrix multiplication in C++.

This first implementation, however, has a poor behavior regarding memory locality. Blocked (or block) matrix multiplication, is a technique used to multiply large matrices by dividing them into smaller blocks or submatrices. Instead of performing the entire matrix multiplication at once, in this technique one breaks the task into smaller, more manageable subproblems.

It's important to note that the partitioning of these factors isn't arbitrary; instead, it necessitates conformable partitions between two matrices, `A` and `B`, ensuring that all submatrix products that will be used are well-defined. Figure 2.4 illustrates this process.

In this case, the code would be:

```
void BlockMatMul(BlockMatrix &A, BlockMatrix &B, BlockMatrix &C) {
  // Go through all the blocks of the matrix.
  for (int i = 0; i < N / BS; ++i)
    for (int j = 0; j < N / BS; ++j) {
      float *BlockC = C.GetBlock(i, j);
      for (int k = 0; k < N / BS; ++k) {
        float *BlockA = A.GetBlock(i, k);
        float *BlockB = B.GetBlock(k,j);
        // Go through the block.
        for (int ii = 0; ii < BS; ii++)
          for (int jj = 0; jj < BS; jj++) {
            for (int kk = 0; kk < BS; ++kk)
              BlockC[ii + jj * BS] += BlockA[ii + kk * BS] *
               ↪  BlockB[kk + jj * BS];
          }
      }
    }
}
```

Listing 14: Blocked matrix multiplication in C++.

In our example, the `BlockMatrix` class serves as a utility wrapper designed to partition the entire matrix into blocks, thereby capitalizing on data locality. Each block is encapsulated within a separate array. To achieve parallelization, we can compute the multiplication of each pair of blocks on distinct nodes, employing the following code:

```
void BlockMatMul(BlockMatrix &A, BlockMatrix &B, BlockMatrix &C) {
  #pragma omp parallel
  #pragma omp single
  for (int i = 0; i < N / BS; ++i)
    for (int j = 0; j < N / BS; ++j) {
      float *BlockC = C.GetBlock(i, j);
      for (int k = 0; k < N / BS; ++k) {
        float *BlockA = A.GetBlock(i, k);
        float *BlockB = B.GetBlock(k,j);
        #pragma omp target depend(in: *BlockA, *BlockB) \
                           depend(inout: *BlockC) \
                           map(to: BlockA[:BS*BS], BlockB[:BS*BS]) \
                           map(tofrom: BlockC[:BS*BS]) nowait
        for (int ii = 0; ii < BS; ii++)
          for (int jj = 0; jj < BS; jj++) {
            for (int kk = 0; kk < BS; ++kk)
              BlockC[ii + jj * BS] += BlockA[ii + kk * BS] *
               ↪  BlockB[kk + jj * BS];
          }
      }
    }
}
```

Listing 15: Blocked parallel matrix multiplication in C++ using OMPC.

As we carry out the multiplication of each block in the node, we have to send the block of matrix `A`, the block of matrix `B` as input (`map(to: BlockA[:BS*BS], BlockB[:BS*BS])`), and the block of matrix `C` as output and input (`map(tofrom: BlockC[:BS*BS])`). The multiplication process depends on input blocks `A` and `B` (`depend(in: BlockA[0], BlockB[0])`) and block `C` as output (`depend(inout: BlockC[0])`).

It is also possible to further optimize the code by using a second level of parallelism within each node using the `parallel for` directive as shown below:

```
void BlockMatMul(BlockMatrix &A, BlockMatrix &B, BlockMatrix &C) {
  #pragma omp parallel
  #pragma omp single
  for (int i = 0; i < N / BS; ++i)
    for (int j = 0; j < N / BS; ++j) {
      float *BlockC = C.GetBlock(i, j);
      for (int k = 0; k < N / BS; ++k) {
        float *BlockA = A.GetBlock(i, k);
        float *BlockB = B.GetBlock(k,j);
        #pragma omp target depend(in: *BlockA, *BlockB) \
                           depend(inout: *BlockC) \
                           map(to: BlockA[:BS*BS], BlockB[:BS*BS]) \
                           map(tofrom: BlockC[:BS*BS]) nowait
        #pragma omp parallel for
        for(int ii = 0; ii < BS; ii++)
          for(int jj = 0; jj < BS; jj++) {
            for(int kk = 0; kk < BS; ++kk)
              BlockC[ii + jj * BS] += BlockA[ii + kk * BS] *
               ↪  BlockB[kk + jj * BS];
          }
      }
    }
}
```

Listing 16: Blocked parallel matrix multiplication in C++ using OMPC.

However, this implementation is inneficient. The issue arises from the need to transmit all three blocks between the head and worker processes for each target task, with no opportunity for the runtime to optimize the inter-node communication.

To fix that problem, the input blocks can be sent in advance using *target enter data tasks* (`target enter data map(...) depend(...) nowait`) and the resulting blocks retrieved back using the inverse *target exit data tasks* (`target exit data map(from: ...) depend(...) nowait`) at the end. In this scenario, the OMPC scheduler and data manager gain the capability to optimize both the allocation of target tasks across worker processes and the communication among them. The resulting code would look as follows:

```
1  void BlockMatMul(BlockMatrix &A, BlockMatrix &B, BlockMatrix &C) {
2  #pragma omp parallel
3  #pragma omp single
4  {
5    // Maps all matrices' blocks asynchronously (as tasks).
6    for (int i = 0; i < N / BS; ++i) {
7      for (int j = 0; j < N / BS; ++j) {
8        float *BlockA = A.GetBlock(i, j);
9        #pragma omp target enter data map(to: BlockA[:BS*BS]) \
10                                       depend(out: *BlockA) nowait
11       float *BlockB = B.GetBlock(i, j);
12       #pragma omp target enter data map(to: BlockB[:BS*BS]) \
13                                       depend(out: *BlockB) nowait
14       float *BlockC = C.GetBlock(i, j);
15       #pragma omp target enter data map(to: BlockC[:BS*BS]) \
16                                       depend(out: *BlockC) nowait
17     }
18   }
19
20   for (int i = 0; i < N / BS; ++i)
21     for (int j = 0; j < N / BS; ++j) {
22       float *BlockC = C.GetBlock(i, j);
23       for (int k = 0; k < N / BS; ++k) {
24         float *BlockA = A.GetBlock(i, k);
25         float *BlockB = B.GetBlock(k,j);
26         // Submits the multiplication for the ijk-block
27         // Data is mapped implicitly and automatically moved by the
             ↪ runtime
28         #pragma omp target depend(in: *BlockA, *BlockB) \
29                                 depend(inout: *BlockC)
30         #pragma omp parallel for
31         for(int ii = 0; ii < BS; ii++)
32           for(int jj = 0; jj < BS; jj++) {
33             for(int kk = 0; kk < BS; ++kk)
34               BlockC[ii + jj * BS] += BlockA[ii + kk * BS] *
                   ↪ BlockB[kk + jj * BS];
35           }
36       }
37     }
38
39     // Removes all matrices' blocks and acquires the final result
         ↪ asynchronously.
40     for (int i = 0; i < N / BS; ++i) {
41       for (int j = 0; j < N / BS; ++j) {
42         float *BlockA = A.GetBlock(i, j);
43         #pragma omp target exit data map(release: BlockA[:BS*BS]) \
44                                      depend(inout: *BlockA) nowait
45         float *BlockB = B.GetBlock(i, j);
46         #pragma omp target exit data map(release: BlockB[:BS*BS]) \
47                                      depend(inout: *BlockB) nowait
48         float *BlockC = C.GetBlock(i, j);
49         #pragma omp target exit data map(from: BlockC[:BS*BS]) \
50                                      depend(inout: *BlockC) nowait
51       }
52     }
53
54 }
55 }
```

Listing 17: Optimized blocked parallel matrix multiplication in C++ using OMPC.

It's crucial to emphasize that each target task must use the first position of the block as a dependency. As mentioned earlier, this is a mandatory requirement for the runtime to accurately monitor the data usage and effectively manage communication among worker processes.

2.5. Profiling

Sometimes we need a deeper understanding and information about the execution of our application to optimize it and remove performance bottlenecks. In this section we provide a cookbook on how to collect, process and analyze OMPC traces.

2.5.1. Collecting a trace

The OmpCluster runtime includes native support for gathering execution traces in the JSON format. The activation is done through an environment variable:

```
1 export OMPCLUSTER_PROFILE="/path/to/file_prefix"
```

Listing 18: Execution traces are enabled via an environment variable which contais the prefix for the trace file name.

OMPC can also generate a task graph in DOT format. As it is the case for the trace file, the task graph generation can also can be enabled via an environment variable:

```
1 export OMPCLUSTER_TASK_GRAPH_DUMP_PATH="/path/to/graph_file_prefix"
```

Listing 19: Task graph generation can be enabled via an environment variable which contais the prefix for the DOT file names.

Once you've enabled tracing and/or task graph dump, you can proceed with running the application as usual. Upon completion of the execution, the runtime will generate timeline files with the naming convention `<file_prefix>_<process_name>.json` and two graph files named `<graph_file_prefix>_graph_<graph_number>.dot`. Each MPI process will have its corresponding JSON file. Analyzing these traces individually can be a bit challenging; hence, we offer the *OMPCBench* tool (available at `https://gitlab.com/ompcluster/ompcbench`) to simplify the analysis process.

2.5.2. Merging timelines

After a successful application execution, multiple trace files may be generated. Merging these files into a single consolidated trace can be beneficial.

To achieve this, you can begin by cloning and installing the OMPCBench tool on your machine. Follow the instructions outlined in the README file available at `https://gitlab.com/ompcluster/ompcbench/-/blob/main/README.md` to set up OMPCBench within a virtual environment. Once the installation is complete, you can merge the timelines of all the processes into a single trace by executing the following command:

```
# Run the following command inside the virtualenv:
ompcbench merge --no-sync # (Optional) Synchronize timelines disabled. The clocks may differ between processes, so by default the timelines are synchronized.
                --developer # (Optional) Generate a timeline for runtime developers (with more information and no filters applied).
                --ompc-prefix /path/to/file_prefix # Specify the common prefix or directory of the timelines to merge.
                --ompt-prefix /path/to/file_prefix # (Optional) Specify the common prefix or directory of the OmpTracing timelines to merge.
                --output tracing.json # (Optional) Merged timeline name. If not passed, default name is tracing.json
```

Listing 20: Merging of traces using the ompcbench tool.

For further details and explanations regarding the options available with the ompcbench command, you can access the help documentation by running `ompcbench --help`, which will provide a comprehensive overview of the available options and their usage.

Upon execution, a file named `tracing.json` will be generated, allowing you to move on to the subsequent inspection stage. If the task graph file is located in the traces folder, the timeline will include task dependencies and identifiers.

2.5.3. Inspecting the trace file

The trace files are compatible with the Chrome Tracing tool of the Chrome Web Browser. To visualize your trace within a timeline, follow these steps:

1. Open the Chrome Browser and go to the URL `chrome://tracing`.

2. In the top-left corner, click “Load”.

3. Then, either open your merged timeline by selecting the appropriate file, or simply drag and drop the file into the browser window.

You should now have your application trace displayed, complete with runtime operations. An example timeline can be seen in Figure 2.5.

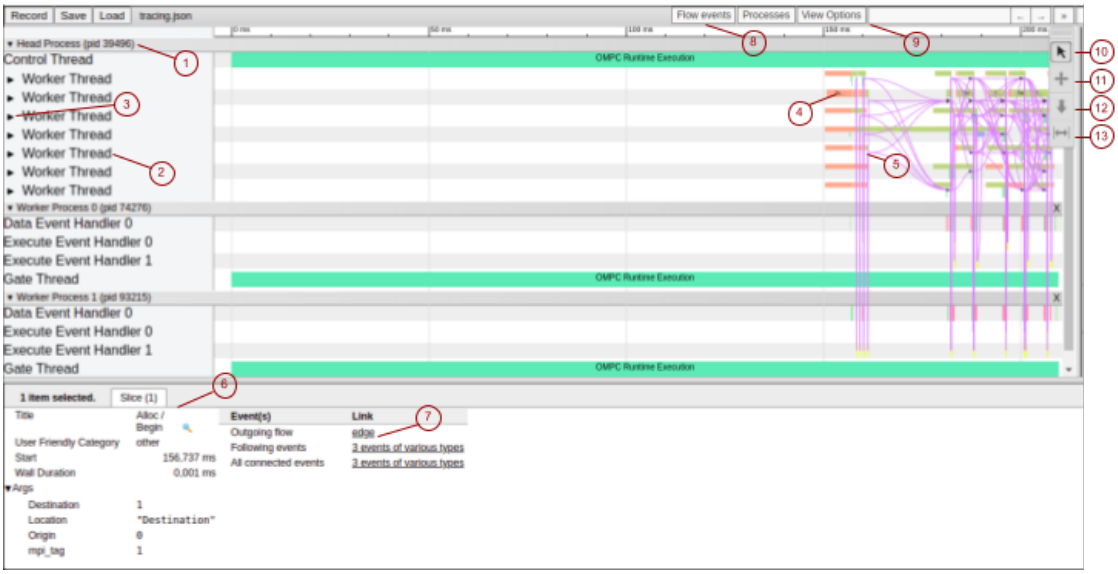


Figure 2.5. Visualization of a timeline using the `chrome://tracing` tool.

The numbers indicated in the timeline (Figure 2.5) are:

1. Process separation: all threads below belongs to the referenced process.

2. Thread separation: all events on the right belongs to the referenced thread.

3. Arrow to hide events: used to decrease the height of the timeline, as events from that thread are compressed vertically. It is useful when users need to analyze events that are vertically distant on the timeline. The arrow next to the process name has a similar function, but completely hides the threads and events of that process.

4. Timeline events: the label indicates what it represents on OMPC. All the colors are chosen by Chrome Tracing except for the events named “Task XX”, where events of the same color have the same source location and `XX` is the task id.

5. Arrows that indicate relations between different events.

6. Event information: when an event is selected, by clicking on it, this panel shows some event information. The first lines are information provided by the Chrome Tracing tool (as event start, duration, and arrows) and the args section is specific information about this event provided by OMPC.

7. Provides information about any arrow from or to this event. If click on, it will show the two events linked.

8. If clicked on, shows a more clear view of the timeline by hiding the events arrows.

9. Used to search for events by label or any of its arguments.

10. Chrome Tracing tool to select events. This feature must be enabled to exhibit event info by clicking on it.

11. Chrome Tracing tool to move across the timeline. It is useful when the timeline is zoomed in to a specific point.

12. Chrome Tracing tool to zoom the timeline. It is useful to analyze events more precisely and see events that have a short duration (like communication events). It is possible to zoom into a specific event by pressing `f` on the keyboard.

13. Chrome Tracing tool to measure the duration between two events on the timeline. It is useful when events are in different processes.

More information about OMPC Profiling can be seen on the paper by Pinho et al. [6] and on OMPC Documentation (`https://ompcluster.readthedocs.io/en/latest/profiling.html`).

2.6. Conclusion

Parallel and distributed computing are increasingly crucial in the era of Big Data, AI, and scientific computing. However, efficient parallel programming, especially in HPC environments, has historically been a challenging task reserved for specialists. This complexity often stems from the need to employ numerous tools and techniques simultaneously to achieve satisfactory results.

In this context, OMPC offers an alternative that simplifies the process of developing new HPC applications. OMPC is a distributed runtime based on tasks that leverages OpenMP's task programming model for parallelizing code. Unlike traditional OpenMP

tasks, OMPC distributes tasks across heterogeneous computers, allowing for the exploration of both shared-memory and distributed-memory parallelism while automatically handling all communications using MPI.

This text explores the core functionalities of OMPC and provides examples of its use. For additional information, please refer to the online documentation at `https://ompcluster.readthedocs.io/` or visit the project's website at `https://ompcluster.gitlab.io/`.

References

[1] OAR Board. Openmp application programming interface-version 5.2, 2021.

[2] Stephen P. Crago and John Paul Walters. Heterogeneous cloud computing: The way forward. *Computer*, 48(1):59–61, 2015.

[3] Pedro Henrique Di Francia Rosso and Emilio Francesquini. Ocftl: An mpi implementation-independent fault tolerance library for task-based applications. In Isidoro Gitler, Carlos Jaime Barrios Hernández, and Esteban Meneses, editors, *High Performance Computing*, pages 131–147, Cham, 2022. Springer International Publishing.

[4] Hans Werner Meuer, Erich Strohmaier, Jack Dongarra, and Horst D Simon. *The TOP500: History, Trends, and Future Directions in High Performance Computing*. Chapman & Hall/CRC, 1st edition, 2014.

[5] OpenMP. OpenMP Application Program Interface. Technical report, 2013.

[6] Vitoria Pinho, Hervé Yviquel, Marcio Machado Pereira, and Guido Araujo. Omptracing: Easy profiling of openmp programs. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 249–256, 2020.

[7] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[8] Hervé Yviquel, Marcio Pereira, Emílio Francesquini, Guilherme Valarini, Gustavo Leite, Pedro Rosso, Rodrigo Ceccato, Carla Cusihualpa, Vitoria Dias, Sandro Rigo, Alan Sousa, and Guido Araujo. The OpenMP cluster programming model. *51st International Conference on Parallel Processing Workshop Proceedings (ICPP Workshops 22)*, 2022.

Capítulo 3

Além do Básico: Otimizando Aplicações Paralelas em Arquiteturas Modernas

Arthur Francisco Lorenzon

Abstract

Applications characterized by their complexity and huge data sets from different areas have driven the demand for fast and efficient computational processing. In this context, exascale architectures emerge as a promising solution. However, to extract the maximum possible performance from these architectures, there is a need to optimize parallel applications. Therefore, this chapter is dedicated to exploring the challenges and solutions associated with such applications, highlighting the main optimizations used in both hardware and software that can be employed to improve the performance and energy consumption of parallel applications.

Resumo

Aplicações caracterizadas por sua complexidade e enormes conjuntos de dados de diferentes áreas têm impulsionado a demanda por processamento computacional rápido e eficiente. Nesse contexto, as arquiteturas exascale emergem como uma solução promissora. No entanto, para extrair o máximo possível de desempenho destas arquiteturas, há a necessidade de otimizar as aplicações paralelas. Assim, este capítulo se dedica a explorar os desafios e soluções associados à tais aplicações, destacando as principais otimizações utilizadas tanto em hardware e software que podem ser empregadas para melhorar o desempenho e consumo de energia de aplicações paralelas.

3.1. Introdução

Novas aplicações de diferentes áreas como inteligência artificial, medicina, biologia e geofísica têm impulsionado a demanda por processamento computacional rápido e eficiente. Estas aplicações, muitas vezes caracterizadas por sua complexidade e enormes conjuntos de dados, exigem uma capacidade computacional que vai além do convencional. Nesse contexto, as arquiteturas *exascale* emergem como uma solução promissora.

Capazes de realizar quintilhões de operações por segundo, essas arquiteturas representam o ápice do poder computacional atual. Elas não apenas atendem às demandas de processamento de aplicações avançadas, mas também abrem portas para inovações e descobertas que antes eram consideradas inatingíveis. Com o poder das arquiteturas *exascale*, os limites da pesquisa e da inovação em campos críticos estão sendo constantemente redefinidos, permitindo avanços em diversas áreas do conhecimento.

No entanto, simplesmente possuir o poder bruto das arquiteturas *exascale* não é suficiente. Para aproveitar ao máximo esse potencial, é essencial otimizar aplicações paralelas para essas arquiteturas. A otimização em termos de desempenho garante que as aplicações sejam executadas de maneira eficaz, reduzindo tempos de espera e maximizando a produtividade. Além disso, com a crescente preocupação global sobre o consumo de energia e seu impacto ambiental, a eficiência energética tornou-se uma prioridade. Otimizar aplicações paralelas para consumo eficiente de energia em arquiteturas *exascale* não apenas reduz os custos operacionais, mas também contribui para uma abordagem mais sustentável e responsável em termos de tecnologia. Portanto, enquanto avançamos para uma era dominada por supercomputadores *exascale*, a otimização de aplicações se torna uma peça chave para garantir que essas máquinas operem de maneira eficaz e consciente.

Apesar destes avanços em *hardware*, um desafio persistente está relacionado a escalabilidade das aplicações paralelas. Muitas delas não escalam linearmente com o aumento do número de recursos de *hardware* disponíveis. Em teoria, ao dobrar os recursos, esperaríamos dobrar o desempenho. No entanto, na prática, isto rasamente acontece devido à fatores associados ao *hardware* e *software* que afetam diretamente nesta escalabilidade. Componentes de *hardware*, como a interconexão entre núcleos de processamento, a latência e largura de banda da memória, e o acesso ao armazenamento, podem se tornar gargalos significativos à medida que mais recursos são adicionados. Por outro lado, no âmbito de *software*, a forma como o código é escrito, a eficiência dos algoritmos e a gestão de *threads* e processos podem limitar a capacidade de uma aplicação aproveitar totalmente os recursos disponíveis. Além disso, desafios como a comunicação entre *threads*, o balanceamento de carga e a sincronização podem impedir que aplicações paralelas alcancem seu potencial máximo de desempenho.

Assim, este capítulo se dedica a explorar os desafios e soluções associados às aplicações paralelas em arquiteturas modernas. Inicialmente, a Seção 1.2 destaca as principais características de arquiteturas multicore modernas. Então, serão abordados os principais gargalos de hardware e software que impactam diretamente a escalabilidade dessas aplicações, na Seção 1.3. Em seguida, na Seção 1.4, otimizações utilizadas tanto em hardware quanto em software serão abordadas, destacando estratégias e ferramentas que podem ser empregadas para aprimorar a execução de aplicações paralelas. Por fim, a Seção 1.5 conclui este capítulo.

3.2. Referencial Teórico

Para compreender os desafios e oportunidades associados à escalabilidade de aplicações paralelas, é essencial ter uma base sólida sobre a organização interna dos componentes de hardware. Assim, nesta seção, os principais componentes da CPU e GPU são explorados.

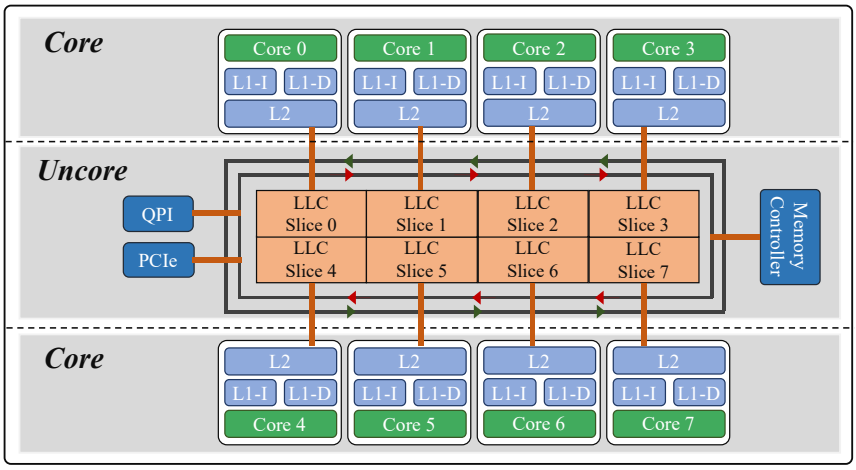


Figura 3.1: Exemplo do *package* de um processador com a parte do *Core* e *Uncore*

3.2.1. Arquitetura e Organização de CPUs

As arquiteturas multicore possuem múltiplas unidades de processamento (núcleos) e um sistema de memória que permite a comunicação entre os núcleos. Cada núcleo é um processador lógico independente com seus recursos, como unidades funcionais, execução de pipeline, registradores, entre outros. O sistema de memória consiste em memórias privadas mais próximas do processador e acessíveis apenas por um único processador; e memórias compartilhadas, que estão mais distantes do processador e podem ser acessadas por múltiplos processadores [Hennessy and Patterson 2003]. A Figura 1.1 mostra um exemplo de arquitetura multicore com oito núcleos (C0, C1, ..., C7) e suas memórias privadas (caches L1 e L2) e compartilhadas (cache L3 e memória principal).

Processadores multicore podem explorar o paralelismo no nível de *threads*. Neste caso, múltiplos processadores executam simultaneamente partes do mesmo programa, trocando dados em tempo de execução através de variáveis compartilhadas ou passagem de mensagens. Independentemente do processador ou modelo de comunicação, a troca de dados é feita através de instruções de carregamento/armazenamento em regiões de memória compartilhada. Essas regiões estão mais distantes do processador (por exemplo, cache L3 e memória principal), e possuem maior atraso e consumo de energia quando comparadas às memórias mais próximas dele (por exemplo, registro , caches L1 e L2) [Korthikanti and Agha 2010].

Dentre os desafios enfrentados no projeto de arquiteturas multicore, um dos mais importantes está relacionado ao acesso a dados em aplicações paralelas. Quando um dado privado é acessado, sua localização é migrada para o cache privado de um núcleo, pois nenhum outro processador utilizará a mesma variável. Por outro lado, um dado compartilhado é replicado em múltiplos caches, uma vez que outros processadores podem acessá-lo para se comunicar. Portanto, embora o compartilhamento de dados melhore a simultaneidade entre vários processadores, ele também introduz o problema de coerência do cache: quando um processador grava em quaisquer dados compartilhados, as informações armazenadas em outros caches tornam-se inválidas. Para resolver este problema, são utilizados protocolos de coerência de cache.

Os protocolos de coerência de cache são classificados em duas classes: baseados em diretório e espionagem [Patterson and Hennessy 2013]. No primeiro caso, um diretório centralizado mantém o estado de cada bloco em diferentes caches. Quando uma entrada for modificada, o diretório será responsável por atualizar ou invalidar os outros caches com aquela entrada. No protocolo de espionagem, em vez de manter o estado do bloco de compartilhamento em um único diretório, cada cache que possui uma cópia dos dados pode rastrear o status de compartilhamento do bloco. Assim, todos os processadores observam as operações de memória e tomam as medidas adequadas para atualizar ou invalidar o conteúdo do cache local, se necessário.

Ao desenvolver aplicações paralelas, o desenvolvedor de software não precisa conhecer todos os detalhes da coerência do cache. Porém, saber como a troca de dados é realizada em nível de hardware pode ajudar o programador a tomar melhores decisões durante o desenvolvimento de aplicações paralelas.

3.2.2. Arquitetura e Organização da GPU

Uma arquitetura GPU é normalmente composta de dois componentes principais: (*i*) Memória Global, que é análoga a RAM em um servidor CPU, sendo acessível por ambos GPU e CPU; e (*ii*) processadores SIMD *multithreaded*, onde a computação é realizada. Estes processadores são chamados de *streaming multiprocessors* (SMs) nas arquiteturas NVIDIA e cada SM tem sua própria unidade de controle, registradores, *pipelines* de execução, *caches*, entre outros componentes de *hardware*. Considerando que a nomenclatura que envolve a discussão de GPUs pode variar de acordo com a bibliografia e empresa (e.g., NVIDIA e AMD utilizam nomenclaturas diferentes para se referir a mesma informação), o resto deste texto considera a nomenclatura utilizada pela NVIDIA.

SMs são processadores de propósito geral porém com uma frequência de operação mais baixa. Um SM é composto basicamente dos seguintes componentes (no entanto, é importante destacar que, a cada nova versão da arquitetura da NVIDIA, diferenças significativas são realizadas nos componentes internos a um SM). Estas diferenças são discutidas abaixo e podem ser acompanhadas na Figura 1.2, que apresenta um SM da arquitetura Ampere da NVIDIA: **Núcleos de processamento**: podendo ser sub-divididos em *Cuda cores, Ray-Tracing cores* e *Tensor cores*. Cada GPU NVIDIA contém centenas ou dezenas de *CUDA cores*, onde um *CUDA core* não pode buscar ou decodificar instruções. Ele apenas faz requisições, computa sobre os dados e responde com o resultado. Assim, *CUDA cores* contém unidades de ponto flutuante de precisão simples, dupla, unidades funcionais especiais, unidades lógicas, unidades de *branch*, entre outros componentes. Por outro lado, *Tensor Cores* apresentam computação de multi-precisão para inferência eficiente em inteligência artificial e aprendizado de máquina. Por fim, *Ray-Tracing Cores* são unidades aceleradoras dedicadas para realizar operações de *ray-tracing* e são exclusivos para placas gráficas NVIDIA RTX. ***Warp schedulers***: unidades de escalonamento e despacho para conjuntos de threads; ***Caches***: *Cache L0 de instrução*, encontrada em arquiteturas mais atuais, que é privada a cada bloco de processamento; *L1 data cache*, que é privada a cada SM, com baixa latência de acesso e alta largura de banda, podendo ser reconfigurada; *Constant cache*, para comunicar as leituras de uma memória somente leitura; Adicionalmente, externo ao SM, podem ser encontradas as seguintes memórias: *L2 data cache*, que é compartilhada entre todos os SMs da GPU, usada para compartilhar

Figura 3.2: SM da arquitetura Ampere da NVIDIA - [Zone 2019]

dados, constantes e instruções; e *RAM*, consistindo da memória global.

O número de *warps* que podem ser executadas em paralelo é dependente da quantidade de *CUDA cores* disponíveis na arquitetura. Por exemplo, se existirem 8 *CUDA cores* em cada SM, então as 32 *threads* do *warp* levarão 4 ciclos para concluir a execução (8 *threads* ativas por ciclo). Como esta é uma definição utilizada pelo *hardware*, o número de *warps* não pode ser mudado pelo programador. Assim, para fornecer maior flexibilidade ao programador, a sequência de operações SIMD são agrupadas em blocos de *threads* (e no nível de hardware elas são dinamicamente distribuídas em *warps*). Portanto, os programadores podem definir o número de *CUDA threads* por bloco assim como o número de blocos que são criados na chamada da função que irá executar na GPU.

Um bloco de *threads* consiste de uma parte do laço vetorizado que será executado em SMs, composto de um ou mais *warps* onde a comunicação acontece via memória local. Um conjunto de blocos de threads é denominado *Grid* (*loop* vetorizado). Assim, para oferecer mais oportunidades para o *hardware* da GPU gerenciar a execução e explorar o paralelismo disponível, um *grid* é decomposto em múltiplos blocos de *threads*. Um bloco de *thread* é então atribuído pelo escalonador de *warps* ao SM, que executa este código. O programador informa ao escalonador do *warp*, que é implementado em hardware, quantos blocos de *threads* devem ser executados.

Uma vez que SMs são processadores completos com PCs separados, uma GPU pode ter de uma a várias dezenas de SMs. Por exemplo, o sistema Pascal P100 tem

56, enquanto que chips menores podem ter apenas um ou dois. Portanto, para fornecer escalabilidade transparente entre modelos de GPUs com um número diferente de SMs, o escalonador de blocos de *threads* é responsável por atribuir os blocos de *threads* aos SM. Uma vez que o bloco de *threads* foi escalonado para um SM, o escalonador do *warp* sabe quais *warps* estão prontos para executar e os envia para uma unidade de despacho. Assim, a GPU tem dois níveis de escalonadores de *hardware*: (*i*) o *escalonador de blocos de threads* que atribui blocos de threads a SMs e (*ii*) o *escalonador de warps* interno ao SM, que escalona os *warps* quando estiverem prontos para execução. Como por definição os *warps* são independentes um dos outros, o escalonador de *warps* pode escolher qualquer um que esteja pronto para execução.

3.2.3. Computação Paralela

A computação paralela refere-se ao processo de executar múltiplas operações ou tarefas simultaneamente, aproveitando a capacidade de sistemas com múltiplos núcleos de processamento ou unidades de processamento. Em vez de processar uma única tarefa de cada vez, como na computação sequencial, a computação paralela executa de maneira concomitante tarefas menores do problema inicial. A programação paralela, por outro lado, é a arte e ciência de criar software que pode explorar o potencial da computação paralela. Ela envolve o uso de técnicas, ferramentas e linguagens de programação específicas para desenvolver aplicações que podem dividir tarefas em partes menores e gerenciar sua execução simultânea.

O desenvolvimento de aplicações capazes de explorar o potencial paralelismo das arquiteturas de multiprocessadores é uma tarefa complexa que envolve uma compreensão profunda da organização dos sistemas, incluindo aspectos como tamanho, estrutura e hierarquia da memória. Embora os sistemas operacionais forneçam transparência em relação à alocação e agendamento de diferentes processos nos vários núcleos, a verdadeira exploração do TLP, que se refere à divisão da aplicação em threads ou processos, recai sobre o programador. Neste cenário, as Interfaces de Programação Paralela (IPPs) surgem como ferramentas essenciais, tornando a extração do paralelismo mais fácil, rápida e menos propensa a erros.

Dentre as IPPs disponíveis, e linguagens de programação disponíveis para exploração do paralelismo, temos uma variedade que atende a diferentes necessidades e arquiteturas. OpenMP é uma opção amplamente adotada para memória compartilhada em C/C++ e FORTRAN, enquanto PThreads oferece ajustes mais refinados na granularidade da carga de trabalho. Cilk Plus estende a linguagem C/C++ para indicar paralelismo, e TBB proporciona um modelo de tarefas para paralelismo. MPI é uma biblioteca padrão de passagem de mensagens para várias linguagens. Além dessas, temos CUDA, desenvolvida pela NVIDIA, que é específica para GPUs da mesma marca. OpenACC e OpenCL são frameworks que suportam programação paralela em diversas plataformas, incluindo GPUs e CPUs. SYCL é uma abstração de alto nível para OpenCL, enquanto AMDRocM é uma plataforma aberta da AMD que permite a programação de GPUs Radeon. Cada uma dessas linguagens e interfaces tem suas peculiaridades e vantagens, desde a exploração de paralelismo em nível de tarefa até a otimização de comunicações em ambientes de memória compartilhada. A escolha correta e o entendimento profundo de suas nuances são cruciais para maximizar o desempenho e a eficiência das aplicações paralelas.

3.3. Escalabilidade de Aplicações Paralelas

Dentro do contexto de escalabilidade de aplicações paralelas, o conceito de *speedup* torna-se fundamental. Ele é uma métrica usada para quantificar o desempenho de um sistema ou algoritmo quando ele é paralelizado em comparação com sua versão sequencial. Em termos simples, o *speedup* indica quantas vezes um programa paralelo é mais rápido do que sua contraparte sequencial. Matematicamente, dado o tempo de execução do programa quando executado de maneira sequencial T_1, e o tempo de execução do programa quando executado em paralelo T_p, o *speedup* (S) da execução paralela é definido como:

$$S = \frac{T_1}{T_p} \tag{1}$$

Em um cenário ideal, uma aplicação paralela que dobra a quantidade de núcleos de processamento deveria, teoricamente, dobrar seu desempenho, alcançando o que é conhecido como escalabilidade linear. No entanto, na prática, alcançar essa escalabilidade perfeita é raro devido a uma série de desafios inerentes às arquiteturas multicore modernas, os quais são discutidos a seguir.

3.3.1. Lei de *Amdahl*

Esta lei teoriza que a melhoria obtida com a otimização de uma parte de um sistema é limitada pela fração do tempo que essa parte é usada. Em outras palavras, se apenas uma pequena porção de uma aplicação pode ser paralelizada, o desempenho geral será fortemente influenciado pela porção sequencial, independentemente de quantos núcleos estejam disponíveis. A Lei de *Amdahl*, por sua vez, define o *speedup* máximo que uma aplicação pode alcançar quando apenas uma parte dela pode ser paralelizada. Esta lei destaca uma limitação fundamental da programação paralela: mesmo com um número infinito de processadores, o *speedup* será limitado pela porção sequencial do programa. Dado a proporção do programa que pode ser paralelizada P, o número de processadores N, o *speedup* máximo S é calculado pela equação abaixo:

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{2}$$

3.3.2. Sincronização

A necessidade de sincronizar threads para garantir a consistência dos dados e evitar condições de corrida pode introduzir latências significativas. Mecanismos de sincronização, como semáforos e mutexes, podem causar bloqueios e esperas, limitando a escalabilidade. A Figura 1.3 destaca um exemplo de como a sincronização afeta o desempenho de aplicações paralelas. Ela apresenta um pseudocódigo da função *histograma*, que possui uma região paralela definida pela diretiva *pragma omp parallel for*. Nota-se que, interno à essa região, uma sessão crítica realiza a atualização da variável h. Deste modo, apenas uma única *thread* pode acessar esta sessão a cada momento da execução. Ao lado do pseudocódigo, o esquemático demonstra o perfil de execução desta função com 1, 2 e 4 *threads*. Conforme observado, há ganhos de desempenho até a execução com 2 *threads*. No entanto, para a execução com 4 *threads*, o custo de serialização imposta pela sessão

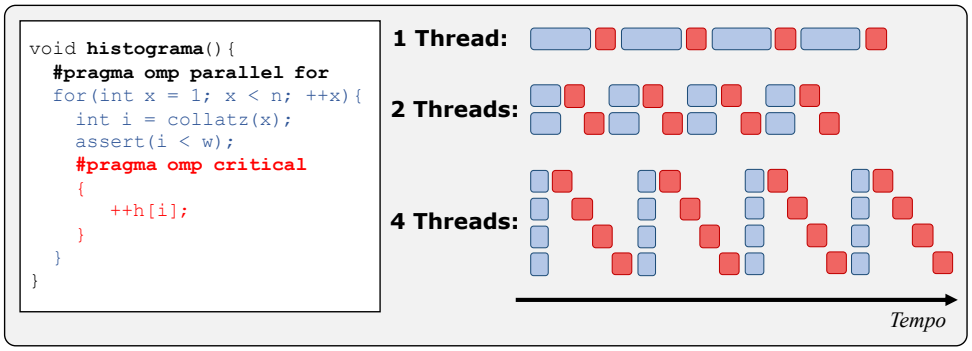


Figura 3.3: Escalabilidade de uma região crítica

crítica sobrepõe os ganhos obtidos pela paralelização do laço *for*. É importante destacar que em arquiteturas GPUs, a sincronização é ainda mais crítica devido ao grande número de *threads* em execução.

3.3.3. Concorrência por Recursos Compartilhados

Em arquiteturas multicore, vários núcleos compartilham certos recursos, como memória cache, barramentos e controladores de memória. Quando múltiplos núcleos tentam acessar esses recursos simultaneamente, pode ocorrer contenção, levando a atrasos e reduzindo o desempenho geral. Adicionalmente, à medida que o número de threads aumenta, a necessidade de comunicação entre elas também cresce. Esta comunicação pode se tornar um gargalo, especialmente se os dados precisarem ser transferidos entre núcleos ou até mesmo entre diferentes processadores.

Quando aplicações lidam com grandes quantidades de dados privados na memória principal e precisam acessar esses dados com frequência, elas enfrentam problemas de escalabilidade devido à sobrecarga do barramento off-chip [Suleman et al. 2008]. Nesse contexto, o aumento no número de threads leva a uma demanda crescente pelo barramento. No entanto, a capacidade de transmissão desse barramento é restrita pelos pinos de I/O [Ham et al. 2013] e não cresce proporcionalmente ao número de threads ativas. Isso significa que adicionar mais threads pode não melhorar o desempenho, mas sim aumentar o consumo de energia quando o barramento atinge sua capacidade máxima.

Da mesma forma, quando as threads precisam acessar dados compartilhados, a frequência com que acessam endereços de memória compartilhada pode impactar tanto o desempenho quanto o consumo de energia à medida que mais threads são adicionadas. A comunicação entre threads, que ocorre ao acessar dados em locais mais distantes do núcleo, como último nível da *cache* e memória principal, pode ser mais lenta e consumir mais energia do que acessar níveis privados da *cache*. Esse tipo de comunicação pode criar pontos de gargalo, afetando a eficiência e o consumo de energia de aplicações paralelas [Subramanian et al. 2013].

3.3.4. Balanceamento de Carga

Nem todas as tarefas em uma aplicação paralela podem ter a mesma quantidade de trabalho. Assim, em arquiteturas NUMA, onde as *threads* frequentemente possuem tempos

de acesso à memória que variam, uma distribuição desigual de carga de trabalho entre as *threads* pode impactar diretamente o desempenho das aplicações. Assim, a política de alocação de *threads* e dados é importante neste contexto, pois uma alocação não ideal pode intensificar o desbalanceamento de carga. Quando isso ocorre, observa-se que algumas *threads* finalizam sua computação rapidamente, enquanto outras ainda estão em processamento, levando a um subaproveitamento dos recursos e, consequentemente, a um desempenho comprometido.

Neste sentido, o desbalanceamento de carga é uma das principais barreiras para otimizar aplicações paralelas em arquiteturas NUMA. Em cenários ideais, a carga de trabalho seria distribuída de forma equitativa entre as *threads*, permitindo que todas elas trabalhassem de forma contínua e eficiente. No entanto, com o desequilíbrio, surgem gargalos que impedem a utilização eficaz dos recursos disponíveis. À medida que o número de *threads* aumenta, esse desequilíbrio se torna ainda mais evidente, restringindo a capacidade da aplicação de escalar de forma linear em termos de desempenho e eficiência energética. Portanto, é essencial abordar e mitigar esses desequilíbrios para garantir a escalabilidade e otimização de aplicações paralelas.

3.3.5. *Overhead* de Paralelização

Em interfaces de programação paralela que utilizam o modelo *fork-join*, criar as *threads*, dividir uma tarefa em várias sub-tarefas menores e, posteriormente, combinar os resultados pode introduzir um custo adicional à execução da aplicação. Isso é particularmente verdadeiro para tarefas que são intrinsecamente pequenas, onde o tempo de execução da tarefa em si pode ser comparável ou até menor que o sobrecusto de paralelização. Além disso, existem tarefas que não se fragmentam facilmente em sub-tarefas independentes ou que têm dependências intrincadas entre elas. Nestes casos, tentar paralelizá-las pode resultar em complicações adicionais, tornando o processo menos eficiente do que uma execução sequencial.

3.3.6. Divergência de Threads em GPUs

As GPUs são projetadas para maximizar o processamento paralelo, permitindo que uma grande quantidade de operações seja executada simultaneamente. Para alcançar essa eficiência, as *threads* são agrupadas em blocos maiores, conhecidos como *warps* nas arquiteturas NVIDIA e *wavefronts* nas arquiteturas AMD. Quando todas as *threads* dentro desses blocos executam a mesma instrução, elas operam em perfeita harmonia, sendo processadas em paralelo e aproveitando ao máximo os recursos de hardware da GPU.

No entanto, desafios de escalabilidade surgem quando as *threads* dentro de um *warp* ou *wavefront* começam a divergir em suas instruções, como resultado de comandos condicionais. Nesses casos, a GPU pode ser forçada a adotar uma abordagem mais serializada, processando um grupo de threads antes de passar para o próximo. Isso compromete a eficiência inerente das GPUs, pois não estão utilizando plenamente seu potencial de processamento paralelo. Portanto, ao desenvolver códigos paralelos para GPUs, é crucial minimizar essa divergência para garantir um desempenho otimizado.

Exemplos de aplicações que podem apresentar divergência entre as *threads* compreendem simulações de partículas e processamento gráfico. Em simulações de partículas,

como a dinâmica de fluidos, cada partícula pode ter um conjunto diferente de vizinhos e, portanto, um conjunto diferente de forças atuando sobre ela. Se as partículas dentro de um *warp* têm vizinhos significativamente diferentes, as *threads* podem divergir ao calcular as forças. Já em processamento gráfico, suponha que você esteja aplicando um filtro a uma imagem, onde *pixels* acima de um certo valor de brilho recebem um tratamento, enquanto os abaixo desse valor recebem outro. Se os *pixels* dentro de um *warp* variam em brilho em relação ao limiar, haverá divergência nas *threads*, pois algumas *threads* aplicarão o tratamento e outras não.

3.4. Otimizando a execução de aplicações paralelas

Otimizar aplicações paralelas geralmente envolve o ajuste de diferentes parâmetros configuráveis em *hardware* e *software*. A Figura 1.4 ilustra o processo de exploração pelos parâmetros configuráveis (aqui chamado de DSE - *design space exploration*). Primeiramente, os dados de entrada que devem ser avaliados são fornecidos ao DSE. Esses valores são calculados através de qualquer modelo de classificação, como rede neural, regressão linear, modelo matemático, entre outros modelos. Por fim, a saída da fase de exploração dos diferentes parâmetros configuráveis são os resultados contendo as configurações que entregam os melhores resultados de acordo com uma métrica alvo (e.g., desempenho ou energia). Esta exploração pelos vários parâmetros pode ocorrer em diferentes momentos da execução da aplicação [Lorenzon and Beck Filho 2019]: *offline* (antes da aplicação iniciar a execução) ou *online* (durante a execução) e com ou sem adaptação da aplicação paralela em tempo de execução. Estas classes são discutidas a seguir.

Informações *offline* sem decisão e adaptação em tempo de execução. Nesta abordagem a exploração dos parâmetros é realizada totalmente antes da execução da aplicação. Compreende modelos de predição que utilizam uma variedade de modelos estatísticos para analisar valores atuais e históricos da arquitetura e aplicações alvo para fazer previsões sobre o futuro. Porém, os dados obtidos pela predição são utilizados apenas para decidir a melhor configuração para executar uma dada aplicação. Portanto, não há tomada de decisão e adaptação da aplicação paralela em tempo de execução. Geralmente, o processo de modelagem preditiva pode ser dividido em quatro etapas, conforme ilustrado na Figura 1.5a. Na primeira etapa, são coletados dados da arquitetura e aplicações para gerar um modelo. Em seguida, um modelo estatístico é formulado aplicando algum método (e.g., regressão linear ou rede neural) sobre os dados coletados. Depois, são feitas previsões para os novos dados de entrada. Por fim, dados adicionais são usados para validar o modelo.

Informações *offline* com decisão e adaptação em tempo de execução. Esta estratégia se distingue da anterior pois utiliza informações provenientes do modelo preditivo para tomar decisões e adaptar a aplicação paralela enquanto ela está sendo executada. A Figura 1.5b representa essa abordagem, que pode se desenvolver da seguinte maneira:

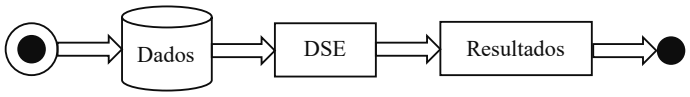
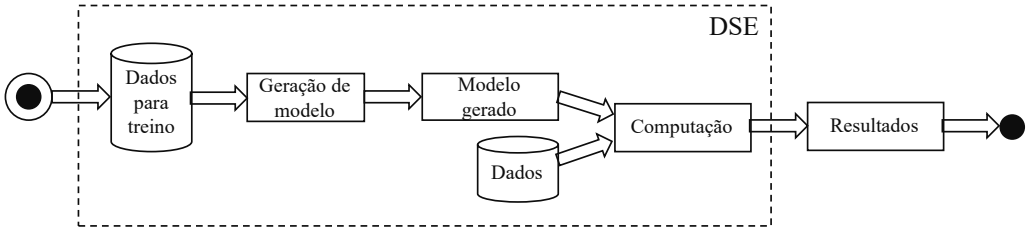
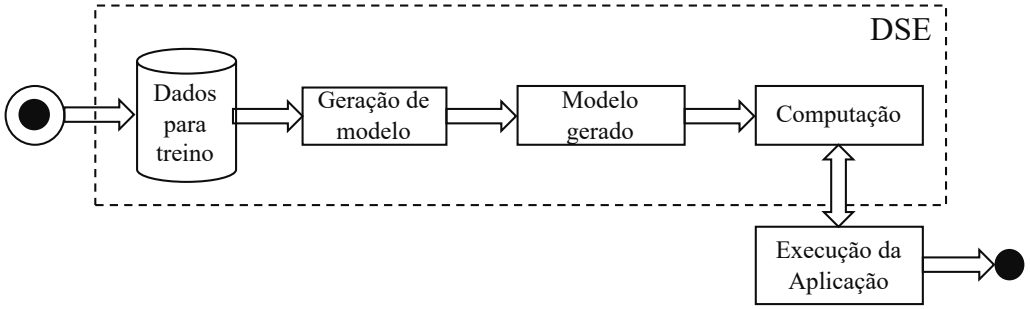


Figura 3.4: Processo de exploração dos parâmetros de configuração



(a) Informação *offline* sem decisão e ajuste em tempo de execução



(b) Informação *offline* com decisão e ajuste em tempo de execução

Figura 3.5: Processo de exploração dos parâmetros de configuração em diferentes momentos da execução

(*i*) criação do modelo com base em informações estáticas fornecidas pela arquitetura e aplicações; (*ii*) então, durante a execução da aplicação, são coletados dados que refletem o seu comportamento; (*iii*) estes dados são inseridos no modelo estabelecido no passo (*i*); e (*iv*) o resultado desse processamento orienta as adaptações no ambiente de execução. Um desafio desta abordagem é a necessidade de recalibrar o modelo estatístico sempre que ocorrem mudanças no ambiente de execução, como alteração nas características da aplicação, na microarquitetura ou nas métricas avaliadas.

Informações *online* sem decisão e adaptação em tempo de execução. Nesta categoria, as estratégias levam em consideração o comportamento atual da microarquitetura do processador durante a compilação da aplicação. O modelo analisa as características da microarquietura e da aplicação para orientar o compilador na produção de um código otimizado para esse ambiente específico para uma futura execução da aplicação. No entanto, não ocorre nenhuma adaptação da aplicação paralela durante sua execução.

Informações *online* com decisão e adaptação em tempo de execução. Em contraste com a abordagem anterior, nesta categoria, os modelos utilizam informações coletadas em tempo real para tomar decisões e ajustar a execução da aplicação. São levadas em conta características específicas da aplicação que só se tornam conhecidas durante a execução, como o tamanho da entrada. A capacidade de se adaptar com base em informações dinâmicas é crucial para aplicações cujo comportamento é variável – onde a carga de trabalho flutua constantemente – e para aquelas com múltiplas regiões paralelas, nas quais cada região apresenta um comportamento distinto.

Neste sentido, as categorias descritas acima podem ser empregadas para selecio-

nar os parâmetros de *hardware* e *software* que maximizam o desempenho e a eficiência energética de aplicações paralela. Ao longo desta seção, são discutidos os principais parâmetros mais relevantes, além de estratégias do estado da arte que oferecem soluções para otimizá-los.

3.4.1. Ajuste do Número de Threads

Conforme discutido na Seção 1.3, muitas aplicações paralelas não escalam com o número de *threads* devido a diferentes fatores de *hardware* e *software*. Nestes cenários, estratégias que realizam o ajuste do número de *threads* podem reduzir o consumo de energia ao limitar o uso dos recursos quando gargalos no *off-chip* são detectados e melhorar o desempenho quando limitar o número de *threads* leva a uma menor contenção dos recursos compartilhados. Estas estratégias podem ser divididas de acordo as categorias descritas anteriormente nesta seção.

Abordagens que não realizam ajuste do número de threads em tempo de execução são discutidos a seguir. Considerando ambientes de alto desempenho, [Witkowski et al. 2013] propõem um modelo de regressão linear para estimar o consumo de energia de aplicações paralelas. [Benedict et al. 2015] utilizam uma abordagem de *Random Forest Modeling* (RFM) para predizer o consumo de energia de aplicações OpenMP. *DwarfCode* é um modelo de predição para aplicações híbridas implementadas com MPI+OpenMP e MPI+-OpenACC [Zhang et al. 2016]. [Jayakumar et al. 2015] fornecem um *framework* de predição que combina assinaturas da execução para predizer o desempenho de aplicações científicas.

Considerando que as aplicações podem ter diferentes comportamentos durante a execução, os seguintes trabalhos realizam o ajuste dinâmico do número de *threads*. Varuna é um sistema que continuamente monitora mudanças no sistema, modela o comportamento de execução e determina o grau de paralelismo ideal [Sridharan et al. 2014]. LAANT é uma biblioteca que ajusta o número de *threads* de aplicações paralelas implementadas com OpenMP [Lorenzon et al. 2017]. Nornir é um sistema que monitora a execução da aplicação e ajusta diversos parâmetros de configuração, como por exemplo, número de threads e frequência de operação do processador com o objetivo de otimizar o desempenho e consumo de energia [Sensi et al. 2016]. Aurora implementa um algoritmo baseado no *hill-climbing* para encontrar o melhor grau de paralelismo para cada região paralela de aplicações OpenMP [Lorenzon et al. 2019]. Hoder implementa um algoritmo de otimização baseado na série de *Fibonnacci* para selecionar o melhor número de *threads* e frequência de operação para cada região paralela de aplicações OpenMP [Schwarzrock et al. 2021].

3.4.2. Mapeamento de Threads e Dados

O mapeamento adequado de threads e dados é fundamental para a escalabilidade de aplicações paralelas, especialmente em arquiteturas NUMA (Non-Uniform Memory Access). Conforme ilustrado na Figura 1.6, nestes sistemas, a memória é dividida em vários nós, e o tempo de acesso à memória pode variar dependendo de onde uma *thread* está executando em relação a um nó de memória específico. Por exemplo, se uma *thread* está alocada no *core 0* (C0) do nodo NUMA 0 e precisa acessar um dado que está na memória local do nodo NUMA 1, ela precisará realizar um acesso remoto à memória, o que tem

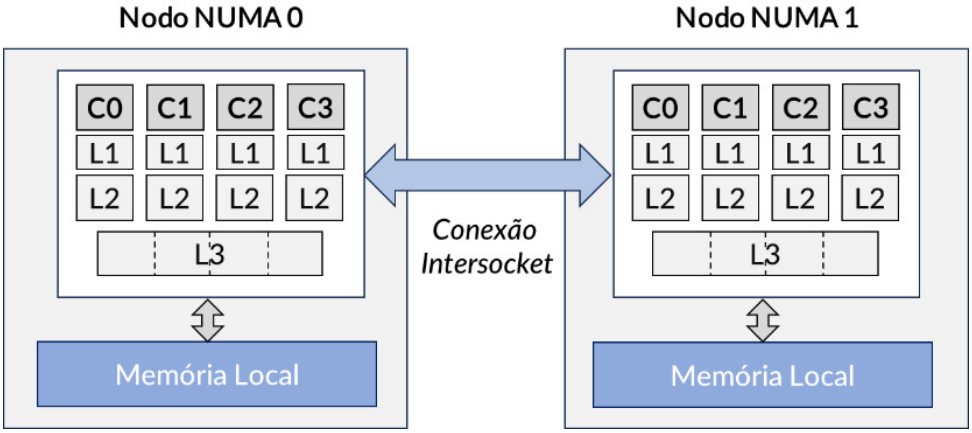


Figura 3.6: Representação de uma máquina com dois nodos NUMA

um custo maior em termos de desempenho e consumo de energia quando comparado ao acesso à memória local.

Portanto, a localização física onde as threads são alocadas em relação aos núcleos de processamento e aos recursos de memória pode ter um impacto significativo no desempenho geral da aplicação. Um mapeamento inadequado pode levar a um aumento na latência de comunicação entre threads e a acessos de memória mais lentos, resultando em esperas desnecessárias e reduzindo a eficiência. Além disso, pode causar desequilíbrios de carga, onde alguns núcleos são sobrecarregados enquanto outros ficam ociosos. Ao considerar a afinidade de threads, garantindo que threads que frequentemente interagem ou compartilham dados sejam mapeadas para núcleos próximos ou para o mesmo núcleo, e levando em conta a proximidade dos nós de memória em sistemas NUMA, pode-se minimizar os custos de comunicação e melhorar o uso do cache. Assim, um mapeamento estratégico de threads é ímportante para maximizar a escalabilidade e o desempenho de aplicações paralelas em arquiteturas multicore, multiprocessador e NUMA.

De modo similar, o mapeamento adequado de dados é importante para otimizar o desempenho das aplicações. Dada a natureza dessas arquiteturas, a localização dos dados pode influenciar significativamente a latência e a largura de banda de acesso à memória. Se os dados acessados frequentemente por uma thread estiverem localizados em um nó de memória distante, isso pode resultar em acessos de memória mais lentos e, consequentemente, em um desempenho subótimo. Ao garantir que os dados sejam alocados próximos aos núcleos que os acessam mais frequentemente, pode-se minimizar a latência de acesso à memória e maximizar a utilização da largura de banda disponível. Portanto, uma estratégia de mapeamento de dados bem planejada é essencial para aproveitar ao máximo as capacidades das arquiteturas NUMA, garantindo uma execução eficiente e escalável das aplicações.

Neste cenário, [Diener et al. 2016b] introduzem uma taxonomia para classificar diferentes mecanismos de mapeamento e fornecer uma compreensiva discussão de soluções existentes. kMAF [Diener et al. 2016a] é um mecanismo que utiliza *page faults* para realizar mapeamento de *threads* e dados e é integrado no *kernel* do sistema operacional. [Serpa et al. 2018] analizam o desempenho de estratégias de mapeamento de *threads* e

dados em arquiteturas multicore e no acelerador Xeon Phi Knights Landing, demonstrando a necessidade de empregar mapeamento para otimizar o desempenho de aplicações paralelas. Graphith [Rocha et al. 2021] é um *framework* que automaticamente adapta o mapeamento de *threads* e dados para melhorar o desempenho de aplicações que processam sobre grafos.

3.4.3. DVFS

O DVFS (*dynamic voltage and frequency scaling* permite que softwares ajustem a frequência e tensão de processadores em tempo real, otimizando o consumo de energia para a tarefa atual sem a necessidade de reinicialização [Le Sueur and Heiser 2010]. Nas arquiteturas modernas, a frequência da CPU e o gerenciamento de energia dos componentes de hardware podem ser controlados pela configuração avançada e interface de energia (ACPI). Ela utiliza *C-states* para economizar energia, desligando subsistemas e núcleos inativos, e *P-states* para ajustar a frequência e tensão do núcleo ativo, equilibrando desempenho e economia. Como destacado na Figura 1.7, os *C-states* básicos definidos pela ACPI são os seguintes: *CO*, no qual a CPU/Core está ativa e execução de instruções; e de *C1* até *Cn*, onde a CPU/Núcleo é configurado para reduzir o consumo de energia cortando o sinal de *clock* e a energia dos núcleos não utilizados e desligando componentes de hardware. Nestes casos, quanto mais unidades desligadas (quanto maior o valor de *Cn*), menos energia é gasta. Por outro lado, *P-states* permite alterar os pontos de operação de frequência e tensão da CPU/núcleo para melhorar o desempenho ou reduzir o consumo de energia enquanto ele está ativo (*CO*).

Em processadores que implementam tecnologia de *boosting* e usam o padrão ACPI para gerenciamento de energia, o *P0-state* representa a frequência operacional mais alta alcançada quando o boosting é ativado. Quando desligado, é definido o *P1-state*, que representa a frequência base máxima de operação sem a interferência de qualquer técnica de *boosting*. Por outro lado, quanto maior o valor de *Pn*, menor será a frequência de operação e o nível de tensão [Wamhoff et al. 2014]. ACPI implementa métodos que permitem ao sistema operacional (SO) alterar o nível *P-state* (por exemplo, através do uso de *governors* DVFS). Nesses casos, o sistema operacional seleciona uma frequência operacional enquanto a tensão é definida automaticamente pelo processador. Os reguladores mais comuns são *desempenho* e *powersave*, nos quais a frequência da CPU é definida estaticamente para a frequência mais alta e mais baixa, respectivamente; e *ondemand*, onde a frequência da CPU é definida dependendo da carga atual do sistema [Le Sueur and Heiser 2010].

Assim, ao ajustar dinamicamente a tensão e a frequência de operação dos núcleos de processamento, é possível adaptar-se às demandas de carga de trabalho em tempo real, permitindo que a aplicação maximize o desempenho quando necessário e reduza o consumo de energia durante períodos de menor demanda. Deste modo, diferentes trabalhos têm focado em aplicar DVFS para otimizar o desempenho e consumo de energia de aplicações paralelas. Pack & Cap [Cochran et al. 2011] é uma técnica para controlar a frequência de operação com o objetivo de maximizar o desempenho considerando um limite de potência. [Wu et al. 2014] emprega DVFS com algoritmos de escalonamento para melhorar a eficiência energética em ambientes de computação na nuvem. De modo similar, DEWTS (*DVFS-enabled energy-efficient workflow task schedulng*) [Tang et al. 2016]

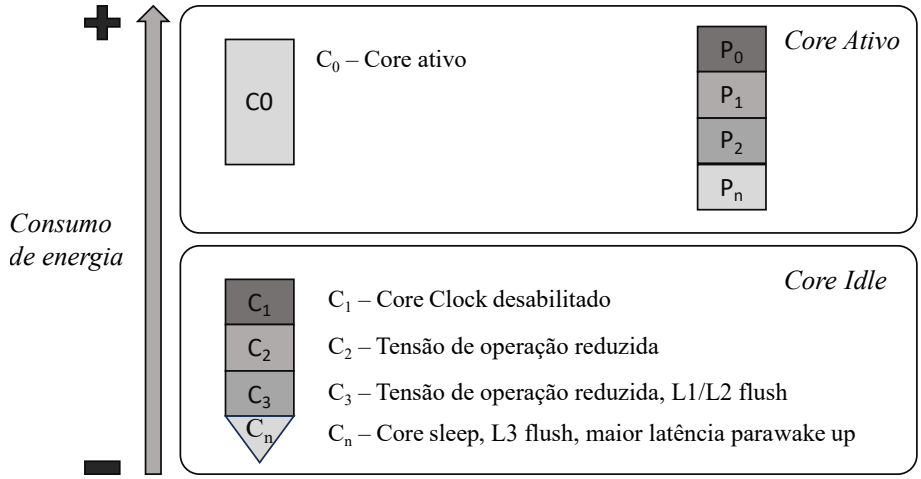


Figura 3.7: Representação dos *P-states* e *C-states*.

explora o escalonamento de tarefas com DVFS para reduzir o consumo de energia de aplicações paralelas com mínimo impacto no desempenho. CoScale [Deng et al. 2012] explora um conjunto de frequências de CPU e memória de modo que elas minimizem o consumo de energia de todo o sistema. [dos Santos Marques et al. 2017] avalia diferentes frequências de operação de CPU e sistema de memória para otimizar a eficiência energética de aplicações paralelas em ambientes embarcados. Poseidon [Marques et al. 2021] é uma abordagem que emprega frequências de turbo em conjunto com DVFS e ajuste dinâmico do número de *threads* para otimizar a eficiência energética de aplicações paralelas.

3.4.4. *Uncore Frequency Scaling*

Uncore Frequency Scaling (UFS) é uma técnica que permite ajustar a frequência de operação do uncore, que é a parte do processador que não inclui os núcleos de processamento, conforme ilustrado na Figura 1.1. Assim, enquanto a frequência dos núcleos de processamento pode ser ajustada para atender às demandas de computação, os componentes do *uncore* - que incluem caches, controladores de memória e interconexões - também têm suas frequências ajustáveis. Ao escalar dinamicamente a frequência desses componentes *uncore*, é possível equilibrar a taxa de transferência de dados e a comunicação entre núcleos com o consumo de energia. Em cenários onde a comunicação ou o acesso à memória são intensivos, aumentar a frequência ”uncore”pode melhorar significativamente o desempenho. Por outro lado, em situações com menos demanda nesses componentes, reduzir a frequência pode resultar em economias de energia consideráveis sem comprometer significativamente o desempenho. Assim, o UFS oferece uma dimensão adicional de otimização, permitindo que sistemas *multicore* alcancem um equilíbrio mais refinado entre desempenho e eficiência energética.

Com a disponibilidade de *drivers* nas últimas versões do *kernel* do *Linux*, diferentes esforços têm sido desenvolvidos para otimizar o consumo de energia de aplicações paralelas via ajuste da frequência do *uncore*. [Won et al. 2014] empregam inteligência artificial para predizer o comportamento de frequência do *uncore* ao executar uma aplicação e ajustar os níveis de acordo. *Uncore Power Scavenger* é um sistema que dinamicamente

detecta fases de execução de uma aplicação paralela e automaticamente define a melhor frequência de *uncore* para cada fase com o objetivo de reduzir o consumo de energia sem um impacto significativo no desempenho [Gholkar et al. 2019]. *Dynamic Uncore Frequency* (DUF) é um processo *daemon* que dinamicamente adapta a frequência de *uncore* para reduzir a potência dissipada considerando um limite definido de degradação de desempenho. [Wang et al. 2022] propõem uma abordagem de aprendizagem por reforço que dinamicamente configura a frequência do *uncore* para otimizar o consumo de energia de sistemas multicore

3.4.5. Sobrepondo Comunicação com Computação

Otimizar o desempenho de aplicações paralelas em ambientes heterogêneos, como aqueles que combinam CPUs, GPUs e outros aceleradores, tem o desafio de equilibrar a computação e a comunicação entre diferentes dispositivos. Assim, a sobreposição de comunicação com computação é essencial para otimizar o desempenho nesses ambientes. Isso ocorre porque, enquanto um dispositivo está ocupado processando dados, outros dispositivos podem estar ociosos esperando por dados ou instruções. Deste modo, ao sobrepor comunicação com computação, é possível minimizar esses tempos de espera, permitindo que os dispositivos troquem informações enquanto ainda estão processando tarefas, reduzindo o tempo total de execução e melhorando o desempenho geral da aplicação.

Assim, diferentes interfaces de programação paralela têm oferecido mecanismos para mitigar este desafio. CUDA, uma plataforma de computação paralela e modelo de programação da NVIDIA, introduziu conceitos como *stream* e *prefetch assíncrono* para otimizar a troca de dados entre CPU e GPU. *Streams* permitem que operações de cópia de memória e execução de *kernels* sejam realizadas em paralelo, possibilitando a sobreposição de computação e comunicação. Ao dividir tarefas em diferentes *streams*, é possível enviar dados para a GPU enquanto outros dados já estão sendo processados, maximizando a utilização da GPU. Por outro lado, o *prefetch assíncrono* é uma técnica que antecipa a transferência de dados necessários para a GPU antes de serem realmente necessários para a computação. Ao prever e transferir dados de forma assíncrona, reduz-se a latência associada às operações de cópia de memória, garantindo que a GPU tenha acesso imediato aos dados quando necessário. Juntas, essas técnicas proporcionam uma otimização significativa na troca de dados entre CPU e GPU, minimizando gargalos e melhorando o desempenho geral das aplicações.

De modo similar, em arquiteturas de memória distribuída, onde múltiplos processadores ou nós de computação operam de forma independente e têm sua própria memória local, a eficiência na comunicação entre esses nós é crucial para o desempenho geral da aplicação. O MPI (*Message Passing Interface*), uma das principais bibliotecas utilizadas para programação em tais arquiteturas, oferece mecanismos de comunicação síncrona e assíncrona (*MPI_Isend* e *MPI_Irecv*). As comunicações assíncronas e não bloqueantes no MPI possibilitam iniciar uma operação de comunicação e, em seguida, continuar com outras tarefas de computação sem ter que esperar que a comunicação seja concluída. Isso permite a sobreposição de comunicação com computação.

Diferentes trabalhos têm focado na implementação de estratégias para otimizar o desempenho de aplicações paralelas através da sobreposição de comunicação com computação.

Considerando as capacidades fornecidas em CUDA, [Potluri et al. 2012] propõem eficientes designs para comunicação MPI intra-node em ambientes multi-GPU para otimizar o desempenho de comunicação MPI *one-sided*. Groute é um construtor para programação assíncrona em ambientes multi-GPUs que habilita o desenvolvimento de aplicações irregulares [Ben-Nun et al. 2017]. LibMP é uma biblioteca de troca de mensagens que demonstra o uso de GPUDirect Async em aplicações através do emprego de *streams* assíncronos para otimizar o desempenho de aplicações numéricas [Agostini et al. 2018].

3.5. Conclusão

Ao longo deste capítulo, foi explorado a necessidade de otimizar aplicações paralelas para arquiteturas *multicore* modernas. As estratégias de otimização discutidas não apenas realçam a importância de maximizar o desempenho, mas também de garantir a eficiência energética e a sustentabilidade. À medida que avançamos na era *exascale*, fica evidente que a otimização não é apenas um complemento, mas uma necessidade crítica. Através das abordagens apresentadas, pode-se não apenas atender às demandas atuais, mas também pavimentar o caminho para futuras inovações, garantindo que as aplicações paralelas sejam energeticamente eficientes.

Referências

[Agostini et al. 2018] Agostini, E., Rossetti, D., and Potluri, S. (2018). Gpudirect async: Exploring gpu synchronous communication techniques for infiniband clusters. *JPDC*, 114:28–45.

[Ben-Nun et al. 2017] Ben-Nun, T., Sutton, M., Pai, S., and Pingali, K. (2017). Groute: An asynchronous multi-gpu programming model for irregular computations. *SIGPLAN Not.*, 52(8):235–248.

[Benedict et al. 2015] Benedict, S., Rejitha, R. S., Gschwandtner, P., Prodan, R., and Fahringer, T. (2015). Energy prediction of openmp applications using random forest modeling approach. In *IEEE IPDPSW*, pages 1251–1260.

[Cochran et al. 2011] Cochran, R., Hankendi, C., Coskun, A. K., and Reda, S. (2011). Pack amp; cap: Adaptive dvfs and thread packing under power caps. In *IEEE/ACM MICRO*, pages 175–185, Los Alamitos, CA, USA. IEEE Computer Society.

[Deng et al. 2012] Deng, Q., Meisner, D., Bhattacharjee, A., Wenisch, T. F., and Bianchini, R. (2012). Coscale: Coordinating cpu and memory system dvfs in server systems. In *IEEE/ACM MICRO*, pages 143–154.

[Diener et al. 2016a] Diener, M., Cruz, E. H. M., Alves, M. A. Z., Navaux, P. O. A., Busse, A., and Heiss, H.-U. (2016a). Kernel-based thread and data mapping for improved memory affinity. *IEEE TPDS*, 27(9):2653–2666.

[Diener et al. 2016b] Diener, M., Cruz, E. H. M., Alves, M. A. Z., Navaux, P. O. A., and Koren, I. (2016b). Affinity-based thread and data mapping in shared memory systems. *ACM Comput. Surv.*, 49(4).

[dos Santos Marques et al. 2017] dos Santos Marques, W., de Souza, P. S. S., Lorenzon, A. F., Schneider Beck, A. C., Beck Rutzig, M., and Diniz Rossi, F. (2017). Improving edp in multi-core embedded systems through multidimensional frequency scaling. In *IEEE ISCAS*, pages 1–4.

[Gholkar et al. 2019] Gholkar, N., Mueller, F., and Rountree, B. (2019). Uncore Power Scavenger: A Runtime for Uncore Power Conservation on HPC Systems. In *SC '19*, NY, USA. ACM.

[Ham et al. 2013] Ham, T. J., Chelepalli, B. K., Xue, N., and Lee, B. C. (2013). Disintegrated control for energy-efficient and heterogeneous memory systems. In *IEEE HPCA*, pages 424–435.

[Hennessy and Patterson 2003] Hennessy, J. L. and Patterson, D. A. (2003). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition.

[Jayakumar et al. 2015] Jayakumar, A., Murali, P., and Vadhiyar, S. (2015). Matching application signatures for performance predictions using a single execution. In *IEEE IPDPS*, pages 1161–1170.

[Korthikanti and Agha 2010] Korthikanti, V. A. and Agha, G. (2010). Towards optimizing energy costs of algorithms for shared memory architectures. In *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*, pages 157–165.

[Le Sueur and Heiser 2010] Le Sueur, E. and Heiser, G. (2010). Dynamic voltage and frequency scaling: The laws of diminishing returns. In *HotPower'10*, pages 1–8, Berkeley, CA, USA. USENIX Association.

[Lorenzon and Beck Filho 2019] Lorenzon, A. F. and Beck Filho, A. C. S. (2019). *Parallel computing hits the power wall: principles, challenges, and a survey of solutions*. Springer Nature.

[Lorenzon et al. 2019] Lorenzon, A. F., de Oliveira, C. C., Souza, J. D., and Beck, A. C. S. (2019). Aurora: Seamless optimization of openmp applications. *IEEE TPDS*, 30(5):1007–1021.

[Lorenzon et al. 2017] Lorenzon, A. F., Souza, J. D., and Beck, A. C. S. (2017). Laant: A library to automatically optimize edp for openmp applications. In *DATE*, pages 1229–1232.

[Marques et al. 2021] Marques, S. M., Medeiros, T. S., Rossi, F. D., Luizelli, M. C., Beck, A. C. S., and Lorenzon, A. F. (2021). Synergically rebalancing parallel execution via dct and turbo boosting. In *ACM/IEEE DAC*, pages 277–282.

[Patterson and Hennessy 2013] Patterson, D. A. and Hennessy, J. L. (2013). *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition.

[Potluri et al. 2012] Potluri, S., Wang, H., Bureddy, D., Singh, A., Rosales, C., and Panda, D. K. (2012). Optimizing mpi communication on multi-gpu systems using cuda inter-process communication. In *IEEE IPDPSW*, pages 1848–1857.

[Rocha et al. 2021] Rocha, H. M. G. d. A., Schwarzrock, J., Lorenzon, A. F., and Beck, A. C. S. (2021). Boosting graph analytics by tuning threads and data affinity on numa systems. In *Euromicro PDP)*, pages 161–168.

[Schwarzrock et al. 2021] Schwarzrock, J., de Oliveira, C. C., Ritt, M., Lorenzon, A. F., and Beck, A. C. S. (2021). A runtime and non-intrusive approach to optimize edp by tuning threads and cpu frequency for openmp applications. *IEEE TPDS*, 32(7):1713–1724.

[Sensi et al. 2016] Sensi, D. D., Torquati, M., and Danelutto, M. (2016). A reconfiguration algorithm for power-aware parallel applications. *TACO*, 13(4):43:1–43:25.

[Serpa et al. 2018] Serpa, M. S., Krause, A. M., Cruz, E. H., Navaux, P. O. A., Pasin, M., and Felber, P. (2018). Optimizing machine learning algorithms on multi-core and many-core architectures using thread and data mapping. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 329–333.

[Sridharan et al. 2014] Sridharan, S., Gupta, G., and Sohi, G. S. (2014). Adaptive, efficient, parallel execution of parallel programs. *ACM SIGPLAN Notices*, 49(6):169–180.

[Subramanian et al. 2013] Subramanian, L., Seshadri, V., Kim, Y., Jaiyen, B., and Mutlu, O. (2013). MISE: Providing performance predictability and improving fairness in shared main memory systems. In *IEEE HPCA*, pages 639–650.

[Suleman et al. 2008] Suleman, M. A., Qureshi, M. K., and Patt, Y. N. (2008). Feedback-driven Threading: Power-efficient and High-performance Execution of Multi-threaded Workloads on CMPs. *SIGARCH Computer Architecture News*, 36(1):277–286.

[Tang et al. 2016] Tang, Z., Qi, L., Cheng, Z., Li, K., Khan, S. U., and Li, K. (2016). An energy-efficient task scheduling algorithm in dvfs-enabled cloud environment. *Journal of Grid Computing*, 14:55–74.

[Wamhoff et al. 2014] Wamhoff, J.-T., Diestelhorst, S., Fetzer, C., Marlier, P., Felber, P., and Dice, D. (2014). The {TURBO} diaries: Application-controlled frequency scaling explained. In *USENIX ATC 14*, pages 193–204.

[Wang et al. 2022] Wang, Y., Zhang, W., Hao, M., and Wang, Z. (2022). Online Power Management for Multi-Cores: A Reinforcement Learning Based Approach. *IEEE TPDS*, 33(4):751–764.

[Witkowski et al. 2013] Witkowski, M., Oleksiak, A., Piontek, T., and Weglarz, J. (2013). Practical power consumption estimation for real life hpc applications. *Future Gener. Comput. Syst.*, 29(1):208–217.

[Won et al. 2014] Won, J.-Y., Chen, X., Gratz, P., Hu, J., and Soteriou, V. (2014). Up by their bootstraps: Online learning in Artificial Neural Networks for CMP uncore power management. In *IEEE HPCA*, pages 308–319.

[Wu et al. 2014] Wu, C.-M., Chang, R.-S., and Chan, H.-Y. (2014). A green energy-efficient scheduling algorithm using the dvfs technique for cloud datacenters. *Future Generation Computer Systems*, 37:141–147.

[Zhang et al. 2016] Zhang, W., Cheng, A. M. K., and Subhlok, J. (2016). Dwarfcode: A performance prediction tool for parallel applications. *IEEE Transactions on Computers*, 65(2):495–507.

[Zone 2019] Zone, N. D. (2019). Cuda toolkit documentation. *NVIDIA,[Online]. Available: http://docs. nvidia. com/cuda/cuda-c-best-practices-guide/# occupancy.[Acedido em Fevereiro 2015]*.

Chapter 4

Are you root? Reproducible Experiments in User Space

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

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

Abstract

High-performance computing platforms are usually shared by users with diverse demands. Often legacy applications or those with very specific requirements and dependencies overload system administrators or simply cannot run. A solution is to build, compile, and install the application and its dependencies fully in user space. However, this task is timely expensive and error-prone, which motivates the use of automated solutions. In this chapter, we present two solutions to install packages fully in user space in a shareable and reproducible way.

Resumo

Plataformas computacionais para processamento de alto desempenho são usualmente compartilhadas por usuários com demandas variadas. Com frequência aplicações legadas ou aquelas que possuem requisitos e dependências muito específicos sobrecarregam os administradores do sistema ou simplesmente tem sua execução inviabilizada. Uma solução passa a ser configurar, compilar e instalar a aplicação e as respectivas dependências inteiramente em espaço de usuário. Entretanto, tal tarefa é custosa e propensa a erros, motivando a adoção de soluções automatizadas. Neste capítulo, apresentamos duas soluções que permitem a instalação de pacotes inteiramente em espaço de usuário de maneira reprodutível e compartilhável.

4.1. Introduction

One fundamental step when using computational platforms is the correct installation of software. When considering large, shareable resources, users usually do not have administrative permissions and must install their software stack in regular user environments. This installation may also require many dependencies that suffer from the same principle.

Although one can manually download the latest source and check the newest manual for each one of the dependencies, this would present many problems. These problems include spending significant time in this process, a chance that such steps are not portable for other platforms, and discovering that specific versions of dependencies may not work for that software stack. These problems motivate modern approaches that can handle complex dependencies stacks while maintaining the installation reproducibility of that build in a non-privileged environment.

Two initiatives, Spack and Guix, aim to install software and its dependencies in user space while being portable and reproducible. This course focuses on explaining both tools and how to create installation recipes for desired software.

The remainder of this chapter is structured as follows. Section 4.2 presents the Spack package manager. Section 4.3 presents the usage of Guix as a package manager. Section 4.4 concludes the chapter with a discussion and future perspectives.

4.2. Part I – Spack

Spack (GAMBLIN et al., 2015) is a fully user-space package manager that targets High-Performance Computing platforms. Package definitions and Spack itself are written in Python, each package being a class that inherits from a base class representing a standard build tool. To request the installation of a package, the user must prepare a package specification that can be as brief as `<package-name>` or as detailed as `<architecture compiler compiler-version package-name package-version list-of-package-options>`. The same structure applies recursively to package dependencies, resulting in a Directed Acyclic Graph (DAG). Figure 4.1 shows the DAG with the software stack of `openblas` in two stacks: one depending on `perl@5.37.9`, and the other on `perl@5.38.0`. Each node represents a unique package specification identified with a hash. At the install phase, Spack will download the tarball and build each package in the DAG, setting the correct paths at `configure` step (or equivalent). Even if identical dependencies can be recycled, as one can see in the figure, many installations of the same package can coexist.

4.2.1. Installation

Spack can be installed and deployed entirely in user space as long as the target system meets some minimal requirements. For a modern Linux distribution, the expected tools are: a Python 3 interpreter; C/C++/Fortran compilers; standard build tools such as `configure`, `make`, and `cmake`; `git`, compression/decompression tools such as `tar` and `gzip`; `curl`.

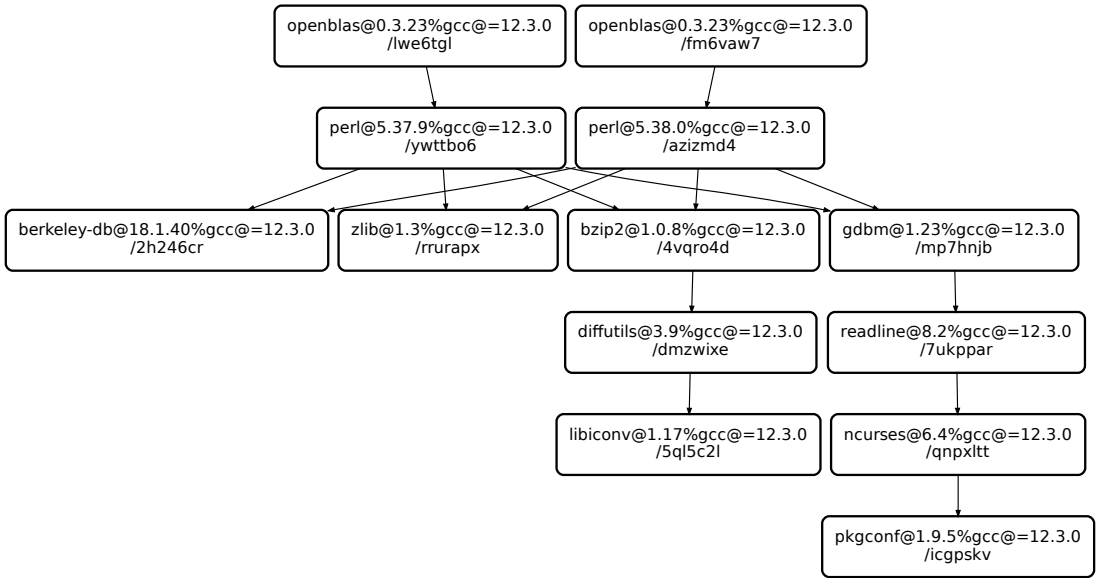


Figure 4.1. Dependency graph of two OpenBLAS installs at Spack

Once these dependencies are available, one can install Spack with:

SH

```
git clone --branch v0.20.1 https://github.com/spack/spack.git
source spack/share/spack/setup-env.sh
```

Spack will try to detect installed compilers automatically. It is possible to check which ones were detected with:

SH

```
spack compilers
```

STDOUT

```
Available compilers
-- clang ubuntu23.04-x86_64
   ------------------------------------
clang@15.0.7

-- gcc ubuntu23.04-x86_64
   ------------------------------------
gcc@12.3.0
gcc@13.2.0
gcc@4.8.5
```

After upgrades or when installing new compilers on the system, one should run `spack compiler find` before being able to use it. Inclusion of compilers installed in non-standard paths can be done with:

SH

```
spack compiler add /local/my-compiler
```

4.2.2. Installing packages

Before installing a new package, one may wish to check the full list of available packages with `spack list`. Filtering this output with a pattern is also possible:

SH

```
spack list "z*"
```

STDOUT

```
z-checker  z3  zabbix  zfp  zfs  zig  zip  zipkin  zlib  zlib-ng
    zoltan  zookeeper  zookeeper-benchmark  zopfli  zpares  zsh
    zstd  zstr  zziplib
19 packages
```

The basic command to install a new package is `spack install <package name>`. This command will download, configure, build, and install `zlib` in the current Spack tree. Any other `zlib` installation living in the root system or in another spack instance will not be touched. Let's see an example:

SH

```
spack install zlib
```

The previous `install` command used only the default parameters defined in the `zlib` package. A summary of all the package information, including its description, versions, configure and build options (i.e., the variants in the Spack jargon), and dependencies is obtained with:

SH

```
spack info zlib
```

STDOUT

```
MakefilePackage:   zlib

Description:
    A free, general-purpose, legally unencumbered lossless
       data-compression
    library.

Homepage: https://zlib.net

Preferred version:
    1.3       http://zlib.net/fossils/zlib-1.3.tar.gz

Safe versions:
    1.3       http://zlib.net/fossils/zlib-1.3.tar.gz
    1.2.13    http://zlib.net/fossils/zlib-1.2.13.tar.gz

Deprecated versions:
    1.2.12    http://zlib.net/fossils/zlib-1.2.12.tar.gz
    1.2.11    http://zlib.net/fossils/zlib-1.2.11.tar.gz
    1.2.8     http://zlib.net/fossils/zlib-1.2.8.tar.gz
    1.2.3     http://zlib.net/fossils/zlib-1.2.3.tar.gz

Variants:
    Name [Default]             When      Allowed values
       Description
    =======================    ======    ================
       ==================================================

    build_system [makefile]    --        makefile, generic
       Build systems supported by the package
    optimize [on]              --        on, off
       Enable -O2 for a more optimized lib
    pic [on]                   --        on, off
       Produce position-independent code (for shared libs)
    shared [on]                --        on, off
       Enables the build of shared libraries.

Build Dependencies:
    None

Link Dependencies:
    None

Run Dependencies:
    None
```

One of the main features of Spack is to allow the coexistence of multiple installations of

the same package. This way, one can install a second instance of `zlib` enabling (with `+`) the `pic` variant and disabling (with `~`) the `shared` and `optimize` variants:

SH

```
spack install zlib+pic~shared~optimize
```

As well as variants, installations using different version numbers, specified with `@<version-number>` and/or compiled with other compilers (using `%<compiler>`) can also coexist:

SH

```
spack install zlib@1.2.13+pic~shared~optimize%clang
```

These three installations live in the same Spack tree. Spack uses rpath (*run-time search path*) to keep each executable isolated with its dependencies. One can check a list of installed packages with `spack find`, which can include a package name to filter the search, for example:

SH

```
spack find zlib
```

STDOUT

```
-- linux-ubuntu23.04-ivybridge / clang@15.0.7
   ------------------
zlib@1.2.13

-- linux-ubuntu23.04-ivybridge / gcc@12.3.0
   --------------------
zlib@1.3  zlib@1.3
3 installed packages
```

When there are several installations of the same package, it may become difficult to differentiate them. The output of `spack find` can be extended with the options `-v` to show the full spec and `-L` to list the hash that acts as an installation's unique identifier:

SH

```
spack find -L -v zlib
```

STDOUT

```
-- linux-ubuntu23.04-ivybridge / clang@15.0.7
   ------------------
qkdk6s2blk5dqqyha3pper6uelymtwpr zlib@1.2.13~optimize+pic~shared
    build_system=makefile

-- linux-ubuntu23.04-ivybridge / gcc@12.3.0
   --------------------
xkwdb6nqvajfcwfjuc75vrc7stzc7zck zlib@1.3~optimize+pic~shared
   build_system=makefile
rrurapxw3hus5ia5vrszozxykmcoffw2 zlib@1.3+optimize+pic+shared
   build_system=makefile
3 installed packages
```

To distinguish very similar installations, one can provide hashes to `spack diff`:

SH

```
spack diff /qkdk6s2b /xkwdb6nq
```

STDOUT

```
--- zlib@1.2.13/qkdk6s2blk5dqqyha3pper6uelymtwpr
+++ zlib@1.3/xkwdb6nqvajfcwfjuc75vrc7stzc7zck
@@ hash @@
-  zlib qkdk6s2blk5dqqyha3pper6uelymtwpr
+  zlib xkwdb6nqvajfcwfjuc75vrc7stzc7zck
@@ node_compiler @@
-  zlib clang
+  zlib gcc
@@ node_compiler_version @@
-  zlib clang 15.0.7
+  zlib gcc 12.3.0
@@ package_hash @@
-  zlib 2w7eqxylpvimalu26prt37tmbdsbqbvjytjjlvpyph6yuqwl63ga====
+  zlib u7vqvwmacj5j7zngg2evhytlxmzhad35mxlxna6tmr4bjyeisgsa====
@@ version @@
-  zlib 1.2.13
+  zlib 1.3
```

Installing a package may require a long dependency chain. With `spack spec`, one can check which new packages will be installed and which ones are already installed and can be reused (i.e., those prefixed with `[+]`).

SH

```
spack spec cbc
```

STDOUT

```
Input spec
--------------------------------
 -   cbc

Concretized
--------------------------------
 -   cbc@2.10.9%gcc@12.3.0 build_system=autotools arch=
    linux-ubuntu23.04-ivybridge
 -       ^cgl@0.60.7%gcc@12.3.0 build_system=autotools arch=
    linux-ubuntu23.04-ivybridge
 -           ^clp@1.17.7%gcc@12.3.0 build_system=autotools arch=
    linux-ubuntu23.04-ivybridge
[+]      ^coinutils@2.11.9%gcc@12.3.0 build_system=autotools
   arch=linux-ubuntu23.04-ivybridge
[+]      ^osi@0.108.8%gcc@12.3.0 build_system=autotools arch=
   linux-ubuntu23.04-ivybridge
[+]          ^pkgconf@1.9.5%gcc@12.3.0 build_system=autotools
   arch=linux-ubuntu23.04-ivybridge
```

Several High-Performance Computing platforms restrict Internet access from compute nodes, making users download all external data on the front-end and then automatically exporting `/home` or `/scratch` directories to compute nodes via NFS (*Network File System*). At the same time, many of these platforms also limit user process duration on the front-end. These two policies, however, can disturb the use of package managers as Spack, since even the building of a single package can demand the download and long compilation of several dependencies. To handle this scenario, one can rely on `spack fetch` to download not only the target package but also its dependencies. For example, to fetch the default `llvm` release and all its missing dependencies (i.e., those that are not yet installed), we run:

SH

```
spack fetch --dependencies --missing llvm
```

After fetching the package and its dependencies on the front-end, the user can log into the compute nodes and proceed the installation with `spack install` using the cached files available in NFS home directory.

When removing a package, it is possible to remove only the package or include all its dependents, i.e., any package that depends on the one being removed. For example, to remove `zlib` and its dependents:

SH

```
spack uninstall --dependents zlib
```

4.2.3. External packages

In terms of reproducibility, ideally, the entire software stack should be managed by Spack. However, many production HPC platforms provide customized or third-party MPI, accelerator, compiler, or linear algebra libraries. To take advantage of these optimizations, one can rely on `spack external` commands that will incorporate these package and their respective paths into the Spack structure. For example, to add a system installed `openmpi`:

SH

```
spack external find openmpi
```

STDOUT

```
==> The following specs have been detected on this system and
   added to /home/user/.spack/packages.yaml
-- no arch / gcc@11.4.0
   ----------------------------------------
openmpi@4.1.2
```

After this, one can use the discovered package (i.e., `openmpi@4.1.2`) as a usual dependency to build other packages:

SH

```
spack install hwloc^openmpi@4.1.2
```

4.2.4. Using a package

After installing a package, Spack provides two ways to use it. The simpler one is `spack load` which will load and update the `PATH` and `MANPATH` environment variables with the install paths of the package and its dependencies. For example, one can load the `unrar` package and then use the homonymous command.

SH

```
spack load unrar
```

Another option is to use `spack view` to create a new directory populated with links to mimic the traditional UNIX tree structure, i.e., `bin/`, `lib/`, `include/`. Users can choose between symbolic or hard links. This is more powerful than `spack load` enabling not only the execution but also development using the installed packages as third-party libraries. For example, to create such structure using soft links for `openblas` and its dependencies, and then compile and execute an application:

SH

```
spack view soft blas-dir openblas
gcc blas-example.c -Lblas-dir/lib -Iblas-dir/include -lopenblas
   -o ex-openblas
LD_LIBRARY_PATH=blas-dir/lib ./ex-openblas
```

4.2.5. Sharing configs

Users can share their stack of installed packages by creating a `spack.yaml` file. This file can be later used to rebuild and deploy the setup in another moment and machine in a reproducible way. Each `spack.yaml` is associated with an environment, so we start with `spack env create` and `activate`. Once the environment is ready, we can add the desired packages. At this point, these packages are not installed, they just are included in the `spack.yaml` file. To effectively install them, we need to explicitly run `spack install`.

SH

```
spack env create --dir spack-env-wscad-2023
spack env activate spack-env-wscad-2023
spack add zlib@1.2.3
spack add nano@4.7
spack add vim features=tiny ^ncurses@6.1
spack install
spack env deactivate
```

The `spack.yaml` file contains abstract information about the packages explicitly added to the environment. This information is flexible since it does not contain details such as implicit dependencies, compiler, or the target platform. The concrete information, as returned by `spack spec`, is stored in another file named `spack.lock`. Such file allows a faithful reproduction but works only on a very similar platform, i.e., same OS version (e.g., `ubuntu22.04`) and same CPU family (`intel.ivybridge`).

YAML

```
# This is a Spack Environment file.
#
# It describes a set of packages to be installed, along with
# configuration settings.
spack:
  # add package specs to the `specs` list
  specs: [zlib@1.2.3, nano@4.7, vim features=tiny ^ncurses@6.1]
  view: true
```

Both files can be shared and later imported in another machine with `spack env create`. For example, to import on a very similar machine:

SH

```
# machine with same arch, OS, compiler
spack env create foo spack.lock
```

Otherwise:

SH

```
# on every machine
spack env create foo spack.yaml
```

4.2.6. Preparing a new package

Spack packages are written in Python. Creating a new package boils down to creating a `package.py` file containing basic package information, such as description, download URL, versions, maintainers, dependencies, and build system. The command `create` provides templates for common general-purpose build systems such as `autotools`, `meson`, and `cmake`. There is also some support for language-specific packages, e.g., `R`, `Python`, and `Ruby`. As an example, a draft package for the `pajeng` tool[1], which is built using CMake, can be obtained with:

SH

```
spack create --name pajeng --template cmake "https://github.com/
   schnorr/pajeng/archive/1.3.6.tar.gz"
```

This command will prepare a directory for the new package `pajeng` in the default spack tree. This directory contains at least the `package.py` file but can also store patch and test files.

In the code listing below, source code lines 4 to 8 describe basic information as description and homepage of `pajeng`; this information will be displayed by `spack info`. Line 10 lists the github username of package maintainers while lines 12 to 16 define two versions to install `pajeng`: release 1.3.6 and from the `git` repository. The parameter `preferred` sets the first one as preferential, i.e., the one that will be installed if no version is provided in `spack install`. The last three lines (18 to 20) declare the required dependencies to build `pajeng`. Such dependencies should also be Spack packages and, if needed, will be automatically installed by Spack.

[1] <https://github.com/schnorr/pajeng>

PYTHON

```python
from spack import *

class Pajeng(CMakePackage):
  """PajeNG is a re-implementation of the well-known Paje
     visualization tool for the analysis of execution traces."""

  homepage = "https://github.com/schnorr/pajeng"
  git = "https://github.com/schnorr/pajeng.git"
  url = "https://github.com/schnorr/pajeng/archive/1.3.6.tar.gz"

  maintainers = ['viniciusvgp', 'schnorr']

  version('1.3.6',
          sha256 = '1a2722bfaeb0c6437fb9e8efc2592edbf14ba01172f9
              7e01c7839ffea8b9d0b3',
          preferred = True)
  version('develop',
          git = 'https://github.com/schnorr/pajeng.git')

  depends_on('boost')
  depends_on('flex')
  depends_on('bison')
```

From this point, it is now possible to run `spack install pajeng`. Editions in packages' receipts, both user-created or existing ones, are possible through `spack edit`. For example, to look up for other releases from the base URL and update the package file, one can use `checksum`:

SH

```
spack checksum --add-to-package pajeng
```

The result is a list of several lines containing all available releases with the respective `sha256` sum. Such lines can now be appended to our initial package definition.

PYTHON

```
version('1.3.6',
  sha256 = '1a2722bfaeb0c6437fb9e8efc2592edbf14ba01172f97e01c7839ffea8b9d0b3')
version('1.3.5',
  sha256 = 'ea8ca02484de4091dcf57289724876ec17dd98e3a032dc609b7ea020ca2629eb')
version('1.3.4',
  sha256 = '284e9a590a2861251e808542663bf1b77bc2c99650a1fbf945cd5bab65402f9e')
version('1.3.3',
  sha256 = '42cf44003d238fd5c4ab512bdeb445fc12f7e3bd3f0526b389f080c84b83b19f')
version('1.3.2',
  sha256 = '97154415a22f9b7f83516e988ea664b3990377d69fca859275ca48d7bfad0932')
version('1.3.1',
  sha256 = '4bc3764aaa7e79da9a81f40c0593b646007b689e4ac20886d06f271ce0fa0a60')
version('1.3',
  sha256 = '781b8be935e10b65470207f4f179bb1196aa6740547f9f1af0cb1c0193f11c6f')
version('1.1',
  sha256 = '986d03e6deed20a3b9d0e076b1be9053c1bc86c8b41ca36cce3ba3b22dc6abca')
version('1.0',
  sha256 = '4d98d1a78669290d0a2e6bfe07a1eb4ab96bd05e5ef78da96d2c3cf03b023aa0')
```

Software evolves over time; new versions include new build options or dependencies and deprecate old ones, resulting in specific dependencies among different versions. For example, older versions of `pajeng` require `qt` versions `4.x` while the version under development needs `fmt` library. To handle these cases, we can add new dependencies that are restricted to some versions:

PYTHON

```
depends_on('qt@:4.999+opengl', when='@:1.3.2')
depends_on('freeglut', when='@:1.3.2')
depends_on('fmt', when='@develop')
```

Note that we use the Python colon (`:`) syntax to deal with ranges of versions, e.g., `depends_on('qt@:4.999+opengl', when='@:1.3.2')` means `pajeng` up to version `1.3.2` depends on `qt` library up to version `4.99` with `opengl` variant enabled.

Packages usually have some build options allowing users to enable or disable features, e.g., passing `cmake` or `configure` parameters. To illustrate this capability, we add

package variants to enable static linking, documentation, and to disable the building of `libpaje` and auxiliary tools:

PYTHON

```
variant('static',
        default = False,
        description = "Build as static library")
variant('doc',
        default = False,
        description = "The Paje Trace File documentation")
variant('lib',
        default = True,
        description = "Build libpaje")
variant('tools',
        default = True,
        description = "Build auxiliary tools")

def cmake_args(self):
  args = [
    self.define_from_variant('STATIC_LINKING', 'static'),
    self.define_from_variant('PAJE_DOC', 'doc'),
    self.define_from_variant('PAJE_LIBRARY', 'lib'),
    self.define_from_variant('PAJE_TOOLS', 'tools')
  ]
  return args
```

Note the use of the `default` parameter to set or unset variants by default. To become effective, the new variants must be transposed to the build options expected by the package building system during compilation and install. Since `pajeng` is built with `cmake`, we have to override the `cmake_args` method to write the proper variables (e.g., `PAJE_DOC`).

The addition of variants and version-specific dependencies introduces a new issue, e.g., incompatible options. The only solution in such cases is to prevent the installation and tell the user the reason. For example, previously, we defined two variants for pajeng: `lib` and `tools`. While requiring the installation of `pajeng+lib~tools` is completely valid, the contrary, i.e., `pajeng~lib+tools`, is inconsistent since building the auxiliary tools (`+tools`) depends on building the library (that was disabled with `~lib`). Besides handling contradictory package options, the same feature is useful for dealing with known bugs or declaring incompatibilities with a given compiler or dependency.

PYTHON

```
conflicts('+tools',
          when = '~lib',
          msg = "Enable libpaje to compile tools.")
```

4.2.7. Publishing a new package

There are two ways to make a new Spack package public. The first one, and the more comprehensive, is submit it to the official repository. At the time this chapter was written, this repository stores more than 7000 packages. Submissions of new packages can be done with pull requests. Once the package passes by the automated tests, it will be available for installation by any user. A complete version of the `pajeng` package, following the steps listed in this chapter, was submitted to the Spack official repository and be found at https://github.com/spack/spack/blob/develop/var/spack/repos/builtin/packages/pajeng/.

The second approach to publishing a new package is through the creation of additional public or private repositories. Such repositories can be created with `spack repo create` and later shared with other users who can add them to their local instances by running `spack repo add`. Additional repositories are useful for software whose access is restricted or for Spack packages in development that are not yet mature enough to be committed to the official repository. Some institutions maintain external and public Spack repositories to publicize and facilitate the installation of *software* developed by them. Another use case for additional repositories is when it is necessary to override a package from the official repository for some reason.

4.3. Part II – Guix

Guix (COURTÈS; WURMUS, 2015; GNU, 2023a; GNU, 2023b) provides user-space commands to automate package builds with a strong focus on reproducibility. The building process executes in a controlled environment to isolate details about host and user configuration. Guix follows Nix (DOLSTRA; JONGE; VISSER, 2004) strategy and makes the build be executed by a daemon on a Linux kernel container. This way, all package builds execute with a dedicated Guix user and with a restricted set of environment variables containing only explicitly declared dependencies. Built packages reside on a system-wide directory (e.g., `/gnu/store/`) and can be shared among different users. Each install is identified by a hash computed using package source code, compiler, libraries, scripts, and dependencies. When a user requests a package install that matches an existing hash, the build is skipped, and Guix only links the files on the user directory, avoiding duplicated builds. Figure 4.2 shows the DAG for Openblas at Guix.

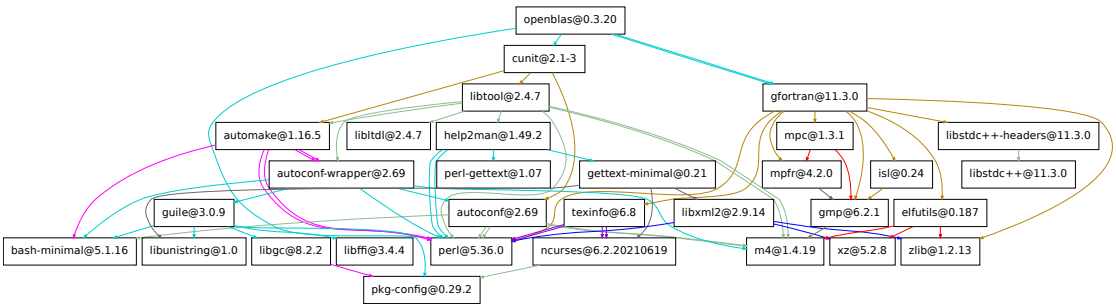


Figure 4.2. Dependency graph of OpenBLAS at Guix

4.3.1. Installation

Unlike Spack, Guix requires administrator rights to be installed. On a Debian-like Linux, one can run[2]:

SH

```
sudo apt install guix
```

From this command, all package install and administration directives can run entirely in user space. Alternatively, in systems where the `guix` daemon itself is not installed and the install cannot be requested, `guix pack` can be used with some limitations (see Section 4.3.4). Before executing install commands, Guix recommends that every user runs some bootstrap configuration for locales:

SH

```
guix install glibc-locales
export GUIX_LOCPATH="$HOME/.guix-profile/lib/locale"
GUIX_PROFILE="$HOME/.guix-profile" source "GUIX_PROFILE/etc/
   profile"
```

4.3.2. Installing packages

The first step when installing a new package is to check the list of all available packages with `guix package --list-available`. At the time this chapter was written, it reported more than 25000 packages. Since the database of packages is huge, a better option is to search using a pattern. For example, to get a filtered list of all packages starting with "ascii" in their synopsis or description fields, one can run:

SH

```
guix package --search="^ascii*"
```

Or to restrict the search to the package name:

SH

```
guix package --list-available="^ascii"
```

STDOUT

```
ascii           3.18    out     gnu/packages/shellutils.scm:67:2
ascii2binary    2.14    out     gnu/packages/textutils.scm:415:2
asciidoc        9.1.0   out     gnu/packages/documentation.scm
   :110:2
asciinema       2.3.0   out     gnu/packages/terminals.scm:226:2
```

[2]Commands presented here consider Guix version 1.3.0

Once the exact package name is known, the installation can proceed with:

SH

```
guix install asciidoc
```

Option `package --list-installed` lists all the installed packages with their respective version and path. For example:

SH

```
guix package --list-installed
```

STDOUT

```
glibc-locales   2.35    out     /gnu/store/03
   v1svhv6wj9pd6awpdi5zn4wd31b23f-glibc-locales-2.35
asciidoc        9.1.0   out     /gnu/store/91
   h31gnblv2jgqx7majqxa4wvjyr0ax5-asciidoc-9.1.0
```

Software evolves with time, and maybe you want to check if there are new releases for installed packages:

SH

```
guix pull
guix upgrade
```

You can remove `asciidoc` package with:

SH

```
guix remove asciidoc
```

4.3.3. Using a package

Once a package was installed, you might want to use it to develop another software. Guix provides `guix shell` to set-up a proper environment with dependencies and environment variables. For example, to start a shell to code something using `openblas` as a library:

SH

```
guix shell --development openblas gcc
```

Another solution is creating a new directory populated with symbolic links to mimic the traditional UNIX tree structure with `bin/`, `lib/`, and `include/`. For example, to compile an `openblas` code:

SH

```
guix install openblas --profile=$HOME/blas-dir
gcc blas-example.c -Lblas-dir/lib -Iblas-dir/include -lopenblas
   -o ex-openblas
LD_LIBRARY_PATH=blas-dir/lib ./ex-openblas
```

4.3.4. Sharing configs

There are two approaches to exporting a Guix setup to another machine. If both machines have Guix, the first way is to use the `guix archive` command that packs a list of packages into a single file. For example:

SH

```
# on machine A
guix archive --export openblas gcc > foo.nar
# on machine B
guix archive --import foo.nar
```

The second strategy allows to export software to a machine without Guix. The command `guix pack` combines a list of packages in a standalone binary tarball that can be extracted on the other machine with standard user-space commands.

SH

```
# on machine A
guix pack openblas gcc
# on machine B
tar xf pack.tar.gz
```

4.3.5. Preparing a new package

Guix packages, or "package definitions" in `guix` jargon, are written in GNU Guile, an implementation of the Scheme language, which itself is a dialect of Lisp. Once again, we use the `pajeng` tool to illustrate a package creation:

SCHEME

```scheme
(use-modules
 (guix packages)
 (guix download)
 (guix build-system cmake)
 (guix licenses)
 (gnu packages boost)
 (gnu packages bison)
 (gnu packages flex)
)
(package
 (name "pajeng")
 (version "1.3.6")
 (source
   (origin
      (method url-fetch)
      (uri
        (string-append
          "https://github.com/schnorr/pajeng/archive/refs/tags/"
          version
          ".tar.gz"))
      (sha256
         (base32
          "1cyhp6lgx7w3qw0pxybj26h4pwfv5rcw5vz8p5zl7imhmszj49qs"
          ))))
 (build-system cmake-build-system)
 (arguments
  `(#:tests? #f
    #:validate-runpath? #f)
 )
 (inputs (list
           boost
           bison
           flex
         ))
 (synopsis
    "PajeNG: library and associated tools for Paje trace files")
 (description
    "PajeNG is a re-implementation of the well-known Paje
    visualization tool for the analysis of execution traces.
    PajeNG comprises the libpaje library, and an auxiliary tool
    called pj_dump to transform Paje trace files to
    Comma-Separated Value (CSV). The space-time visualization
    tool called pajeng had been deprecated (removed from the
    sources) since modern tools do a better job (see
    pj_gantt).")
 (home-page "https://github.com/schnorr/pajeng")
 (license gpl3))
```

The first 9 lines load required modules, i.e., the Scheme equivalents of libraries or packages. Lines 2 to 5 refer to Guix internal modules, while lines 6-9 load `pajeng` dependencies. The package definition itself starts on line 10 with package name, version, and tarball download instructions. Lines 23 to 27 specify the `cmake` building system. We disable tests and validate-runpath phases due to incompatibilities between `pajeng` building configurations and Guix expected ones. Lines 28 to 32 declare package dependencies. Finally, lines 33 to 40 contain package synopsis and description texts, which are presented to the user in commands as `guix show` and `guix search`. The last two lines define the package home-page and license.

From this point, one can try to build and install the created package with:

SH

```
guix build --file=/path-to-created-file/pajeng.scm
guix package
   --install-from-file=/path-to-created-file/pajeng.scm
```

In Guix, working with different version numbers or compilation options requires creating a new package. Fortunately, it is possible to rely on inheritance to extend from an existing package. For example, to create a new package for `pajeng` building from its git repository (at a given commit):

SCHEME

```
(let ((commit "ca24c95cc5b4e53455058e180f01d5a5febccac6")
      (revision "1"))
(package (inherit pajeng)
  (name "pajeng")
  (version (git-version "1.3.6" revision commit))
   (source
     (origin
       (method git-fetch)
       (uri (git-reference
         (url (string-append
                "https://github.com/schnorr/"
                name))
                   (commit commit)))
       (sha256
         (base32
          "1c1ggfgdl5xxq8jkvf774440j5lyn37n8qll354d4lbqxm81v9av")
        )
      )
    )
   (inputs (modify-inputs (package-inputs pajeng)
                     (prepend fmt)))
  )
)
```

Inheritance is also helpful in building with different compilation options, which again requires creating a new package:

SCHEME

```
(package (inherit pajeng)
   (name "pajeng-doc")
   (build-system cmake-build-system)
   (arguments
     '(#:configure-flags '("-DPAJE_DOC=ON")))
   (inputs (modify-inputs (package-inputs pajeng)
                          (prepend texlive-scheme-basic
                             ghostscript poppler asciidoc
                             texlive-titlesec texlive-setspace
                             texlive-listings
                             texlive-gsftopk)))
)
```

4.3.6. Publishing a new package

Guix provides three ways to make a new package public. The first and simpler one is just make available the `scm` file of a package. After obtaining it, one can just run `guix package`, passing the file to the option `--install-from-file`. The `scm` files created here for `pajeng` can be downloaded from: <https://exp-hpc.gitlab.io/wscad-spack-guix-2023>.

The second option is to create a Guix channel, which is just a Git repository[3] containing one or more `scm` files providing Guix package definitions. The `scm` files stored in the repository should have some slight adjustments from the above examples due to some Guix idiosyncrasies. The final step is to add the new channel to the Guix user local configuration by including the following lines to `~/.config/guix/channels.scm`:

SCHEME

```
(cons
  (channel
   (name 'wscad-mc-2023)
   (url
      "https://gitlab.com/exp-hpc/guix-channel-2023-wscad.git"))
  %default-channels)
```

The last way is to push the contributions to the official Guix repository directly. One can checkout `https://git.savannah.gnu.org/git/guix.git` and commit the `scm` file of the new package. This is the recommended method for mature packages that fully comply with the coding style and contributing rules of Guix.

[3] with `master` as the default branch

4.4. Conclusion

This short course covered the fundamental basic concepts for managing software packages at the user level. Facilitating experimental reproducibility, a paramount topic in general research activities, we present the Spack and the Guix tools. We hope that the general goal provided in this short course will cause a positive involvement of new students working in High-Performance Computing and the importance of keeping track of their software versions along their experiments. We expect that these tools provide a more rigorous data collection and observation methods for new researchers. As examples of Spack and Guix usage, we aggregated the code snippets presented in this text in a companion material which is available at: <https://exp-hpc.gitlab.io/wscad-spack-guix-2023/>.

Acknowledgments

We would like to thank Jessica Imlau Dagostini which was our coauthor in a first version of this chapter, prepared in Portuguese, for the *Escola Regional de Alto Desempenho* 2021 (DAGOSTINI et al., 2021) covering the Spack instructions. We also would like to thank the developers and the community of Spack and Guix. This work was partly financed by the Fundação de Amparo à Pesquisa do Rio Grande do Sul (FAPERGS – ARD/ARC 14/2022) to the first author.

References

COURTÈS, L.; WURMUS, R. Reproducible and User-Controlled Software Environments in HPC with Guix. In: *Euro-Par 2015: Parallel Processing Workshops*. Cham: Springer International Publishing, 2015. p. 579–591. ISBN 978-3-319-27308-2.

DAGOSTINI, J. I.; PINTO, V. G.; NESI, L. L.; SCHNORR, L. M. Are you root? Experimentos Reprodutíveis em Espaço de Usuário. In: CHARÃO, A.; SERPA, M. (Ed.). *Minicursos da XXI Escola Regional de Alto Desempenho da Região Sul*. Porto Alegre: Sociedade Brasileira de Computação - SBC, 2021. cap. 3, p. 70–87. ISBN 9786587003504.

DOLSTRA, E.; JONGE, M. de; VISSER, E. Nix: A safe and policy-free system for software deployment. In: *Proceedings of the 18th USENIX Conference on System Administration*. USA: USENIX Association, 2004. (LISA '04), p. 79–92.

GAMBLIN, T. et al. The Spack package manager: bringing order to HPC software chaos. In: *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. [S.l.: s.n.], 2015. p. 1–12.

GNU. *GNU Guix Cookbook*. 2023. Disponível em: <https://guix.gnu.org/cookbook/en/guix-cookbook.html>.

GNU. *GNU Guix Reference Manual*. 2023. Disponível em: <https://guix.gnu.org/manual/en/html_node/index.html>.

Chapter

5

Abordagem Gamificada para Introdução em Arquiteturas de Hardware Evolutivo

Bernardo G. P. Cunha, Carlos Augusto P. da S. Martins

Abstract

In recent years, there has been a search for adaptive systems in a diversity of fields, such as in industries, control systems, aerospace, automobiles, among others. Many approaches were made to reach feasible, adaptive and scalable solutions, and one of them is Evolvable Hardware (EHW). It has the advantages hardware brings along, including high parallelism level, low response time - evolvable systems implemented in hardware can have evolution time of microseconds. The objective of this course is to introduce Evolvable Hardware to computer scientists, computer architects and engineers in a "hands on" approach, first playing a conceptual game, then hacking a developed system and then competing to discover who developed the architecture with best/fast response. After the event, the participants may have a glance at how they can apply hardware computation in their researches or even EHW in their projects.

Resumo

Recentemente há uma procura por sistemas adaptativos ou evolutivos em diversos campos da indústria, em sistemas de controle, aeroespacial, automotiva, entre outras. Muitas abordagens foram desenvolvidas para se chegar a soluções adaptativas, escaláveis e práticas, sendo uma delas o Hardware Evolutivo (EHW). Além de apresentar um elevado nível de paralelismo, sistemas evolutivos implementados em hardware podem apresentar tempo de evolução na ordem de microssegundos. O objetivo deste minicurso é introduzir EHW a cientistas da computação, arquitetos de computação e engenheiros, de maneira "hands on", primeiro introduzindo os conceitos de computação evolucionária como um game, depois hackeando um sistema de EHW já desenvolvido, e por último competindo entre si para descobrir qual arquitetura obteve melhores resultados. Após o minicurso, os participantes terão obtido uma visão geral sobre o tema, e poderão aplicar computação em hardware em suas pesquisas ou até mesmo EHW em seus projetos.

1.1. Descrição do Minicurso - Apresentação

Este minicurso consiste em 3 partes, de aproximadamente 50 minutos. Inicialmente, haverá exposição dos conceitos básicos para compreensão do tema, utilizando abordagem expositiva. Na segunda parte, os participantes colocarão estas novas informações em prática, "hackeando" o sistema evolutivo, experimentando variações nos parâmetros de entrada e de configuração, de uma forma gamificada, em equipes e por turno. Por último, será escolhida a melhor abordagem a ser implementada em um próximo encontro (virtual), e demonstrando como o sistema selecionado pode ser implementado utilizando um dispositivo de computação reconfigurável (FPGA). Aqueles que desejarem pesquisar mais sobre o assunto, pode acessar [**Cunha**, **2020**] para mais conteúdo.

Figure 1.1. Contexto envolvendo EHW. Autores, 2023.

1.2. Estrutura do texto

A estrutura do texto se dispõe em cinco etapas (subcapítulos): uma introdutória, contendo um breve resumo do estado da arte e origem de Evolvable Hardware; em seguida a apresentação dos conceitos necessários para uma melhor compreensão da área; um subcapítulo sobre implementação e obtenção de resultados a partir das ferramentas utilizadas; uma quarta, com os resultados, análises e possíveis modificações, de modo a gerar um sistema que chega à um resultado com melhor valor, de acordo com as métricas escolhidas; e por fim um subcapítulo com conclusões, considerações finais e próximos passos que os interessados podem utilizar para aprofundar em suas pesquisas em EHW.

1.3. Hardware Evolutivo

Evolvable Hardware (EHW) ou Hardware Evolutivo é uma abordagem baseada em três áreas do conhecimento, Engenharia Eletrônica, Ciências da Computação e Biologia. Em computação e engenharia, EHW pode ser utilizado para uma gama de aplicações, como em comunicação de dados [**Grochol**, **2013**], áreas como processamento de imagens digi-

tais [**Vasicek**, **2013**] [**Almeida**, **2018**] [**Salvador**, **2013**], para a criação de circuitos adaptativos [], detecção de falhas [**Wang**, **2014**], computação aproximada [**Vasicek**, **2015**]

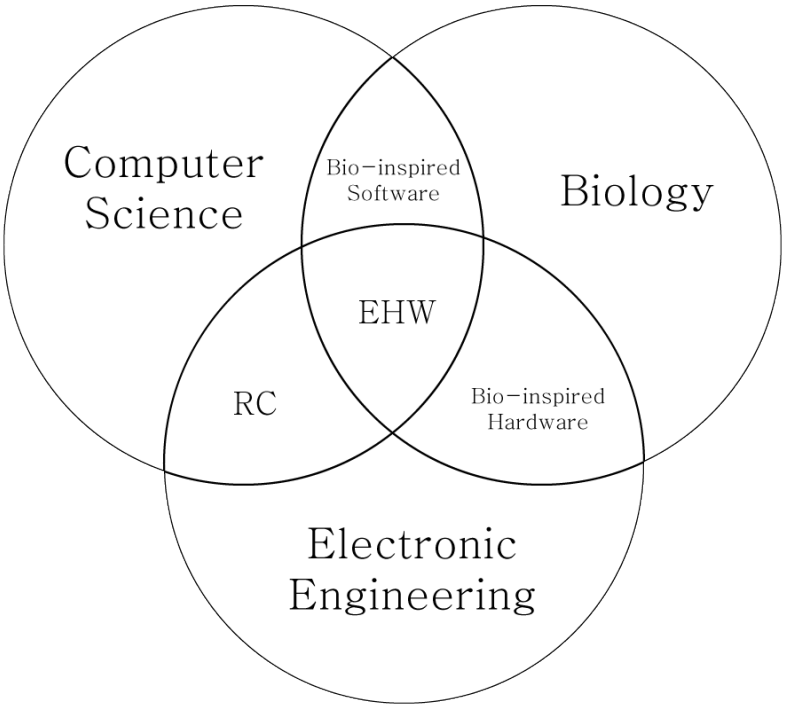


Figure 1.2. Diiagrama explicativo sobre as áreas que compõe Hardware Evolutivo.[Bentley, 2002]

1.4. Computação Reconfigurável

A Computação Reconfigurável (CR) é uma área idealizada na década de 1960, mas só foi fisicamente implementada na década de 1990. Um dos dispositivos digitais mais comuns na CR é o FPGA (Field Programmable Gate Array) [**Salvador**, **2016**]. Nas últimas décadas, esse campo tem crescido devido à demanda por sistemas computacionais de alto desempenho e flexibilidade, entre outros requisitos. Os FPGAs são tradicionalmente configurados por meio de linguagens alfanuméricas de descrição de hardware ou diagramas de circuito. As linguagens mais utilizadas são o Verilog e o VHDL (Very High-Speed Integrated Circuit).

Uma característica inerente à Hardware Evolutivo (EHW) é a reconfigurabilidade, uma vez que a evolução exige diversas alterações na topologia do hardware. O mecanismo responsável por reconfigurar o hardware torna-se um ponto crítico no EHW. Se o processo de modificação da topologia do hardware levar muito tempo, consequentemente, o tempo necessário para evoluir uma solução será prolongado.

No contexto de FPGAs e EHW, existem duas técnicas principais de reconfigu-

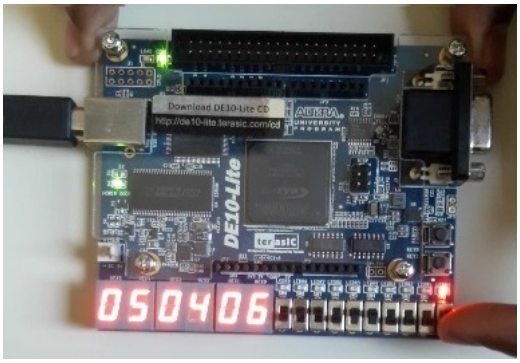


Figure 1.3. FPGA utilizado para testes do EHW. FONTE: Intel, 2019.

ração dinâmica implementadas: Circuito de Reconfiguração Virtual (VRC) e Reconfiguração Dinâmica e Parcial (DPR). Na literatura, é dito que existem compensações no uso de ambas as técnicas. O VRC basicamente utiliza o paralelismo espacial controlado por multiplexadores para criar vários cenários desejados, e os bits de seleção dos multiplexadores reconfiguram o hardware. O DPR consiste em reconfigurar apenas parte do FPGA enquanto o restante está em operação, de forma dinâmica, por meio da alteração do bitstream do dispositivo. O DPR é um recurso que apenas alguns dispositivos FPGA possuem, geralmente a um alto custo financeiro. [**Yao**, **2018**]

1.4.1. VRC

O cerne da solução pretendida foi batizado de *Evolvable Nucleic Architecture* (ENA), com base no conceito de Circuito de Reconfiguração Virtual (VRC) [**Srivastava**, **2014**]. Este módulo é chamado de "ENA" porque é a base, o alicerce (o núcleo) da Arquitetura Evolutiva. Foi projetado com três camadas, ou seja, três circuitos interconectados, compostos por portas AND, OR e NOT, dispostas em uma estrutura semelhante ao CGP (*Carthesian Genetic Programming*). As entradas estão conectadas à primeira camada, que está ligada à seguinte e, em seguida, à última camada. O genótipo é um vetor de bits que configura a topologia do circuito e representa um ponto no espaço de busca da solução. Cada alelo no genótipo configura um multiplexador que seleciona um operando ou operação em uma camada.

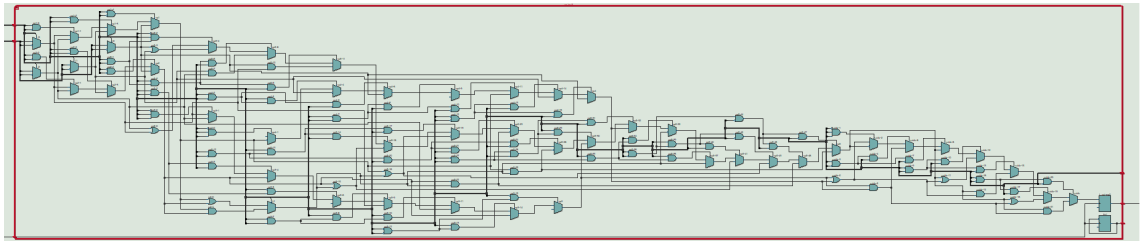


Figure 1.4. Modelo de exemplo de ENA elaborado para genótipos com 22 bits. Cunha, 2020.

Como estudo de caso, neste capítulo é apresentado um genótipo de nove bits. Quando o genótipo sofre uma variação, mesmo em um período muito curto (na ordem dos nanossegundos), um novo circuito é montado. Deve ser destacado que dois genótipos

diferentes podem resultar em um circuito com a mesma funcionalidade (cada linha de sua tabela verdade corresponde à mesma tabela de referência), mas a topologia do circuito pode diferir no número e na disposição das portas lógicas. No entanto, genótipos iguais têm a mesma topologia de circuito, mas podem diferir no valor da avaliação da aptidão, porque a tabela de referência (indicadora da solução desejada) pode ser diferente em ambas as situações.

O modelo utilizado implementa em hardware a ENA baseada em camadas para circuitos de duas entradas e uma saída. Esta foi projetada para ser modular, escalável e explorar o espaço de busca o máximo possível. Embora seja de fato um circuito combinacional, para estar sincronizado com toda a arquitetura, a instância VHDL teve que incluir uma entrada de *clock* para sincronizar a organização do circuito com as outras etapas do AE.

O módulo ENA for projetado especificamente para o objetivo do problema. Dependendo da complexidade do problema ou de sua área de origem, a estrutura da ENA pode variar. No entanto, independentemente da natureza do problema combinacional, é impossível gerar uma solução que não possa ser implementada, uma vez que os blocos de construção básicos são apenas portas lógicas. No entanto, existem soluções que devem ser evitadas: aquelas que fazem uso excessivo de portas lógicas, o que é feito pelo Motor Evolutivo.

Este minicurso escolheu a abordagem através de VRC, uma vez que é mais universal do que o DPR e porque a reconfiguração é mais rápida (para o requisito de desempenho).

1.5. Computação Evolucionária

Computação Evolucionária (CE) é uma área da Ciência da Computação que utiliza conceitos e abstrações inspirados na natureza para desenvolver estratégias otimizadas na exploração de um espaço de busca em um domínio específico de problemas. Esse espaço de busca consiste em um conjunto de soluções possíveis para um problema dado, sendo que cada solução é representada por um indivíduo, que é expresso como um cromossomo. Esse indivíduo possui um fenótipo, que funciona como um código para a interação do cromossomo com o ambiente, ou seja, ele se adapta de acordo com o meio em que está inserido. Neste trabalho, considera-se que cromossomo e genótipo são termos intercambiáveis. O indivíduo é composto por uma combinação do genótipo e do fenótipo.

O genótipo pode ser dividido em genes e alelos, sendo que os alelos são as menores divisões do genótipo, representando um único bit. Um gene é um grupo de alelos que desempenham uma função específica no genótipo. Na estrutura de VRC abordada pelo minicurso, existem 9 alelos e 3 genes.

A avaliação do indivíduo é realizada de acordo com uma função de aptidão (*fitness*), que quantifica o quão adequado o indivíduo é para resolver o problema em questão. A função de aptidão pode ter um único critério, mas geralmente é aplicada em cenários com dois ou mais critérios (otimização multicritério). No entanto, este minicurso não aborda algoritmos de otimização multicritério com mais de 2 objetivos.

Um conjunto de genótipos analisados em um determinado período de tempo é

chamado de população. Cada ciclo de operações do algoritmo é denominado geração. À medida que o tempo avança, a tendência do algoritmo é encontrar indivíduos mais aptos, cujo valor de aptidão é maior do que os das gerações anteriores, até encontrar a solução ótima global (ou ótimas).

Algorithm 1: Algoritmo Evolutivo Genérico

Entrada: Tamanho da população N, Número de gerações G, Taxa de mutação p_m, Taxa de cruzamento p_c
Saida : Solução ótima

1 Inicialize uma população aleatória com tamanho N;
2 Avalie a aptidão de cada indivíduo na população;
3 $geracao \leftarrow 1$ Ate G Selecione pais para cruzamento;
4 Crie uma nova população por meio do cruzamento;
5 Aplique mutações na nova população com probabilidade p_m;
6 Avalie a aptidão de cada indivíduo na nova população;
7 Selecione os melhores indivíduos para a próxima geração;
8 Selecione o melhor indivíduo da população final como a solução ótima;

Como o algoritmo foi implementado em hardware, algumas alterações foram realizadas, principalmente buscando uma alta performance em desempenho do hardware.

1.5.1. Exploração de espaço de projeto

A Exploração do Espaço de Projeto (EEP), de acordo com Kang, Jackson e Shulte (2011), é definida como "a atividade de descobrir e avaliar alternativas de design durante o desenvolvimento de sistemas". É usada para prototipagem rápida, otimização ou integração de sistemas com outros. Um dos principais objetivos deste minicurso é entregar uma arquitetura de hardware evolutivo de alto desempenho e fornecer ferramentas para abordar circuitos combinacionais mais complexos. A EEP é um meio potencial para alcançar as características desejadas. [**Kang**, **2011**]

O objetivo deste minicurso em relação à EEP é obter uma compreensão mais profunda da "caixa-preta", descobrindo quais parâmetros e níveis são mais adequados para um melhor desempenho esperado. O uso da EEP pode ser eficaz quando o sistema desejado possui diversas configurações e possibilidades. Este é o caso do projeto do VRC, que é uma estrutura combinacional dependente de um conjunto de parâmetros em três níveis diferentes.

O principal problema ao especificar uma Arquitetura de Hardware Evolutivo é que o design como um todo possui um grande número de graus de liberdade (Torresen, 2004), representado por um espaço multidimensional. Mesmo quando reduzimos o escopo ou limitamos os parâmetros (como a tecnologia alvo ou qual AE será utilizada), ainda existem vastas possibilidades a serem consideradas no design.

1.5.2. Motor Evolucionário

O Motor Evolutivo (ME), aqui definido como uma arquitetura que implementa as Operações Evolucionárias e a Avaliação da Aptidão em hardware, é, de fato, um módulo

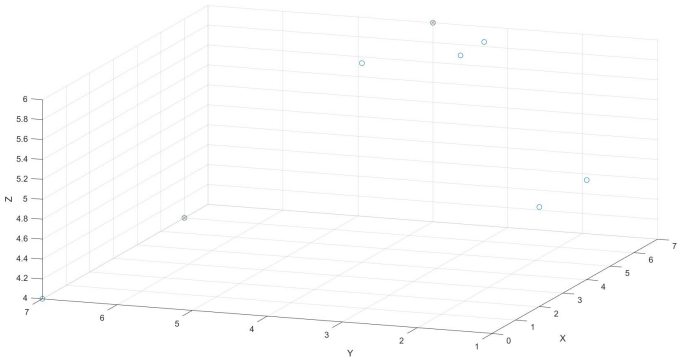


Figure 1.5. Visualização 3D dos genótipos ótimos (círculos azuis preenchidos de vermelho) e sub-ótimos (apenas círculos com bordas azuis) para o problema da porta XOR. Cunha, 2020.

computacional que otimiza o circuito desejado e encontra uma solução que corresponde totalmente à solução do problema, ao mesmo tempo que minimiza o número de portas lógicas necessárias para essa solução. A principal razão para o desenvolvimento de um Motor Evolutivo em vez de um Algoritmo Evolutivo tradicional (como Algoritmos Genéticos ou Programação Genética Cartesiana)[**Babu**, **2016**] é que o Motor Evolutivo é uma implementação mais adequada de Algoritmos Evolutivos para Hardware Evolutivo (EHW). Cancare et al. [**Cancare**, **2011**] menciona que abordagens canônicas genéricas devem ser evitadas nas implementações de hardware de AE.

O ME projetado tem como objetivo oferecer alto desempenho à solução de EHW, pois cada etapa e operação foram planejadas para serem executadas em hardware. Como não há software envolvido na evolução, não há atraso de comunicação entre um processador e o hardware em evolução. A EHW proposta é composta por duas partes principais: o Motor Evolutivo e o Hardware Operacional, que recebe a configuração da solução evoluída criada pelo ME. Na maioria dos trabalhos relacionados a EHW, algumas etapas do ME (ou todas) são implementadas em software, enquanto o Hardware Operacional é implementado em hardware. No entanto, essa abordagem costuma ser o gargalo da solução de EHW.

As operações envolvendo a Arquitetura Evolutiva apresentada neste capítulo incluem a Geração do Genótipo, Avaliação da Aptidão, Mutação, Cruzamento e Seleção. No entanto, cada uma dessas operações envolve diversos aspectos relacionados ao desenvolvimento de hardware, podendo estar relacionada a um ou mais parâmetros ou níveis. A Geração do Genótipo será feita

O AE implementado difere dos algoritmos comuns na literatura, e uma das razões para isso é que todo o processo ocorre simultaneamente, em paralelo. Por exemplo, enquanto o Gerador de Genótipos está fornecendo novos genótipos para a população inicial, eles estão sendo avaliados e outras operações estão sendo aplicadas a eles.

1.5.2.1. Geração de Genótipos

Como os genótipos são números que mapeiam o problema no espaço de solução, o algoritmo evolucionário precisa de um conjunto de indivíduos para criar uma população, e assim selecionar os indivíduos mais adaptáveis à situação-problema para uma próxima geração.

Para isso, é necessário gerar esses números, que aqui são representados por um conjunto de bits. Para tal, a arquitetura do Hardware Evolutivo conta com circuitos geradores de números pseudo-aleatórios (PRNGs), de 9 bits, chamados de LFSRs (Linear-Feedback Shift Registers).

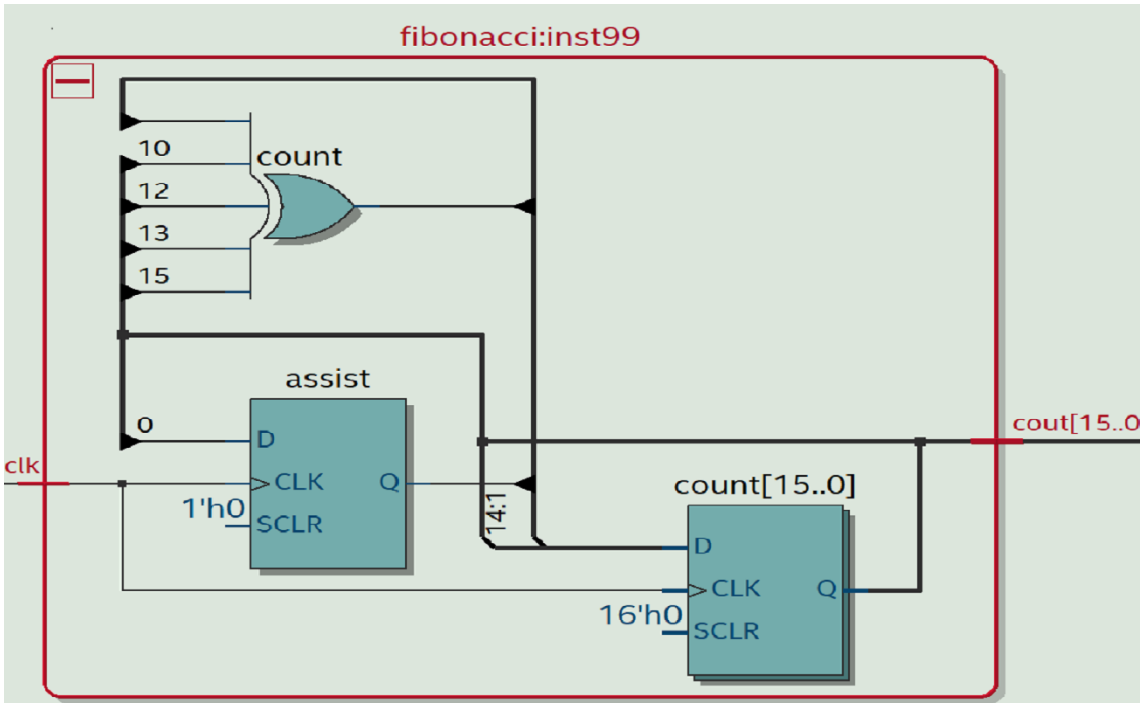


Figure 1.6. Imagem da extração do RTL de um LFSR Fibonacci presente no módulo gerador de PRNGs. Fibonacci é um tipo de circuito com uma retroalimentação (*feedback*) específica. Cunha, 2020

1.5.2.2. Função de Avaliação

A Avaliação de Aptidão (AA) é conhecida por ser a operação mais demorada na implementação de hardware de um AE [**Yao**, **1999**]. No entanto, neste caso, leva apenas um ciclo para avaliar uma solução candidata. Vamos considerar sistemas de dois entradas e uma saída, que podem ser descritos com quatro linhas de uma tabela verdade. Nesse caso, são gerados quatro pares de sinais digitais (chamados de Entradas Virtuais) que estimulam quatro instâncias do módulo ENA, representando as quatro possibilidades de combinações de entradas, simultaneamente.

A saída de cada instância do ANE é verificada com o vetor de referência correspondente pelo módulo de AA. A comparação é feita com um arranjo de portas XNOR

(quatro no total, uma para cada uma das quatro possibilidades), detectando semelhanças.

O Fenótipo é o resultado obtido quando o genótipo configura o circuito de base do EHW e sofre excitação de entrada. É um vetor do mesmo tamanho que o de referência e nos diz qual saída é encontrada pelo circuito para cada combinação de entradas.

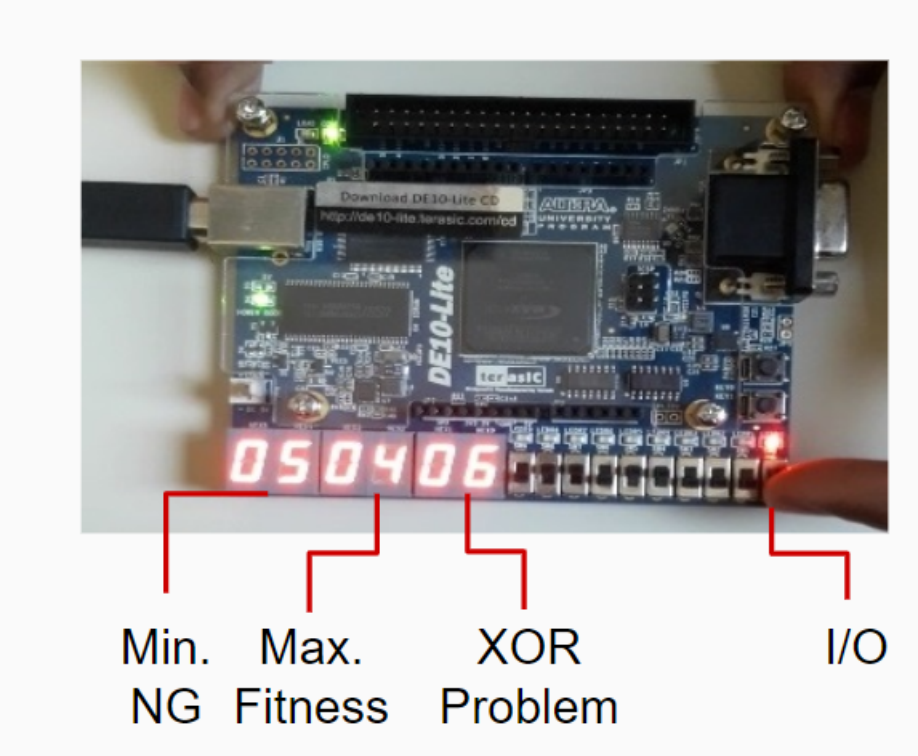


Figure 1.7. FPGA exportando nos displays e LED as saídas do EHW. Autores, 2023.

1.5.2.3. Mutação

A mutação é uma operação evolutiva que aumenta ligeiramente o poder de busca em um AE [**daSilva**, **2018**]. Consideramos esta operação aqui porque ela busca, nas proximidades de um genótipo, por outro que seja mais adequado do que o indivíduo atual. Genericamente, a operação de mutação é usada para evitar pontos de mínimo local. Como o contexto deste minicurso trata de circuitos digitais, a maior parte de seu projeto é representada por notações e cálculos binários. Para uma melhor compreensão de "proximidade" ou "closeness", usamos o seguinte exemplo: dois números podem ser completamente próximos em uma representação inteira e, em binário, ser extremamente distantes (como exemplo, 511 e 512, 0111111111 e 1000000000, respectivamente). É extremamente difícil gerar uma solução candidata 512 a partir de uma operação de mutação sobre 511, ou vice-versa.

A mutação consiste em "flippar" um alelo (bit) do genótipo, ou seja, alterar seu estado de ’1’ para ’0’ ou vice-versa. A ideia de proximidade tem a ver com o conjunto de genótipos que possuem semelhança em seus alelos, por exemplo, 8 em 9 bits são iguais.

1.5.2.4. Cruzamento

Cruzamento é outra operação evolutiva, e esta considera geralmente dois indivíduos que "trocam" entre si um conjunto de bits, gerando indivíduos novos compostos por bits de gerações anteriores.

Nas experiências realizadas em hardware, o cruzamento é aplicado a dois indivíduos de origens diferentes, com uma posição de corte aleatória, e exporta o melhor resultado, e a operação de cruzamento os combina em busca de candidatos a soluções ainda melhores.

1.5.2.5. Seleção e Elitismo

Essas duas operações estão basicamente conectadas nesse projeto. A Seleção é vista neste trabalho como um recurso onde uma solução candidata é escolhida em um grupo de várias outras, de acordo com uma função de avaliação de aptidão. E o Elitismo é um mecanismo que garante que as próximas gerações não terão um genótipo com uma avaliação pior. Elas são implementadas em uma conexão em cascata: primeiro, a Seleção escolhe o melhor resultado dos módulos anteriores, e então o Elitismo só registra a solução candidata se for melhor que o melhor indivíduo atual, de acordo com o código abaixo. Note que a linha 11 implica que a minimização dos portões lógicos só começa quando o requisito funcional é atendido, ou seja, um valor de aptidão igual a quatro ("100" em notação binária).

1.6. Hardware Evolutivo como um *Game*

Fazendo uma analogia com um recurso digital que é de interesse de várias faixas etárias, os jogos digitais, podemos analisar alguns parâmetros de um algoritmo evolucionário como um elemento de um jogo: a função de avaliação ou de *fitness* é como se fosse um *score* ou pontuação que cada jogador (ou genótipo possui). Quando avaliamos um "jogador", comparamos com o histórico de "jogadores" e, se sua avaliação ou score for a mais alta, selecionamos este jogador e dizemos que seu *score* foi a maior pontuação (*high score* ou melhor global).

Os outros genótipos gerados em uma geração são como NPCs (Non Playable Character) que competem com o "personagem principal", genótipo. Se algum dos NPCs tem *score* maior que o jogador, para ele em si, é *game over*. Na próxima geração, teremos o indivíduo que possui o *high score* das últimas gerações, e podemos alterar apenas algumas *features* desse jogador para serem os NPCs dessa geração. Podemos combinar *features* entre os melhores jogadores da rodada passada (cruzamento), modificar um pouquinho uma *feature* do melhor global, podendo resultar em um indivíduo ainda mais forte, ou não. Tudo depende de como a divisão dos bits for feita, sob quais bits dos genótipos originais.

Pensando nessa analogia, a abordagem realizada no minicurso é gamificada porque houve a divisão em equipes, que utilizaram o VRC como uma "caixa preta", e cada equipe tentava descobrir um genótipo que possui nota de avaliação melhor, cada rodada melhor que a da rodada anterior, e melhor que a outra equipe. Os números/genótipos eram argumento de uma função em MATLAB que devolvia a equação booleana, as saídas do

circuito configurado pelo genótipo, o fitness e o número de portas lógicas.

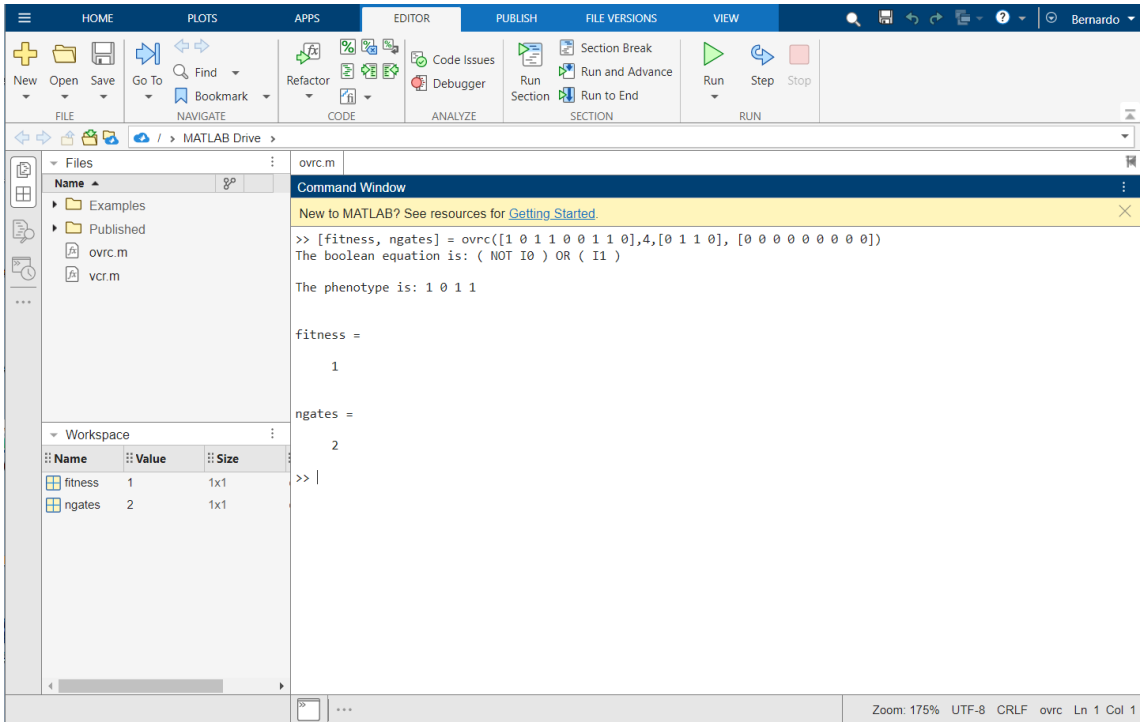


Figure 1.8. Tela de interface do MATLAB online com o console rodando a função OVRC, que devolve a avaliação do genótipo) para o problema da porta XOR. Autores, 2023.

O link para o código para verificação do genótipo está em [**github**, **2023**]

References

[1] P. Bentley, J and T. Gordon, G.W, "On Evolvable Hardware," in Soft Computing in Industrial Electronics, S. Ovaska and L. Sztandera, Eds. Heidelberg,Germany: PhysicaVerlag, 2002.

[2] F. Cancare, S. Bhandari, D. B. Bartolini, M. Carminati and M. D. Santambrogio, ”A bird’s eye view of FPGA-based Evolvable Hardware,” 2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), San Diego, CA, 2011, pp. 169-175. doi: 10.1109/AHS.2011.5963932

[3] R.Salvador. ”Evolvable Hardware in FPGAs: Embedded tutorial”. 11th International Conference on Design Technology of Integration on Nanoscale Era (DTIS), 2016.

[4] Z.Vasicek, M.Bidlo, L.Sekanina, ”Evolution of efficient real-time non-linear image filters for FPGAs”. Soft Comput (2013) 17: 2163.https://doi.org/10.1007/s00500-013-1040-8

[5] R. Salvador, A. Otero, J. Mora, E. De La Torre, T. Riesgo, L. Sekanina, "Self-Reconfigurable Evolvable Hardware System for Adaptive Image Processing", IEEE Transactions on Computers, vol. 62, no. 8, pp. 1481-1493, Aug. 2013

[6] Yao, X., Higuchi, T. (1999). Promises and challenges of evolvable hardware. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 29(1), 87-97.

[7] Huang, X., Wu, N., Zhang, X., Liu, Y. (2015). An evolutionary algorithm based on novel hybrid repair strategy for combinational logic circuits. IEICE ELECTRONICS EXPRESS, 12(22). https://doi.org/10.1587/elex.12.20150765

[8] Dobai, R., Sekanina, L. (2015). Low-Level Flexible Architecture with Hybrid Reconfiguration for Evolvable Hardware. ACM TRANSACTIONS ON RECONFIGURABLE TECHNOLOGY AND SYSTEMS, 8(3). https://doi.org/10.1145/2700414

[9] M.A. Almeida; E.C. Pedrino. "Hybrid Evolvable Hardware for automatic generation of image filters". INTEGRATED COMPUTER-AIDED ENGINEERING, 2018, V.25, no.3, pp. 289-303

[10] R. Yao, P. Zhu, J.J. Du, M.Q. Wang, Z.H. Zhou. "A General Low-Cost Fast Hybrid Reconfiguration Architecture for FPGA-Based Self-Adaptive System". IEICE TRANSACTIONS ON INFORMATION AND SYSTEMS, 2018

[11] Vasicek, Z., Sekanina, L. (2015). Evolutionary Approach to Approximate Digital Circuits Design. IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, 19(3), 432–444. https://doi.org/10.1109/TEVC.2014.2336175

[12] Sekanina, L., Vasicek, Z. (2013). Approximate Circuit Design by Means of Evolvable Hardware. In PROCEEDINGS OF THE 2013 IEEE INTERNATIONAL CONFERENCE ON EVOLVABLE SYSTEMS (ICES) (pp. 21–28). 345 E 47TH ST, NEW YORK, NY 10017 USA: IEEE.

[13] Babu, K. S., Balaji, N. (2016). Approximation of Digital Circuits Using Cartesian Genetic Programming. In PROCEEDINGS OF THE 2016 INTERNATIONAL CONFERENCE ON COMMUNICATION AND ELECTRONICS SYSTEMS (ICCES) (pp. 381–386). 345 E 47TH ST, NEW YORK, NY 10017 USA: IEEE.

[14] Kazarlis, S., Kalomiros, J., Kalaitzis, V., Balouktsis, A., Bogas, D. (2015). Intrinsic Evolution of Digital Circuits Based on a Reconfigurable Hyper-Structure. In E. Haase, J and Kakarountas, A and Grana, M and Fraile-Ardanuy, J and Debono, CJ and Quintian, H and Corchado (Ed.), IEEE EUROCON 2015 - INTERNATIONAL CONFERENCE ON COMPUTER AS A TOOL (EUROCON) (pp. 340–345). 345 E 47TH ST, NEW YORK, NY 10017 USA: IEEE.

[15] da Silva, J. E. H., Manfrini, F. A. L., Bernardino, H. S., Barbosa, H. J. C. (2018). Biased Mutation and Tournament Selection Approaches for Designing Combinational Logic Circuits via Cartesian Genetic Programming. In Anais do XV Encontro Nacional de Inteligencia Artificial e Computacional (pp. 835–846). Porto Alegre, RS, Brasil: SBC. https://doi.org/10.5753/eniac.2018.4471

[16] Stepney, S., Adamatzky, A. (2018). Inspired by Nature: Essays Presented to Julian F. Miller on the Occasion of his 60th Birthday (Vol. 28). https://doi.org/10.1007/978-3-319-67997-6

[17] Grochol, D., Sekanina, L., Zadnik, M., Korenek, J., Kosar, V. (2016). Evolutionary circuit design for fast FPGA-based classification of network application protocols. APPLIED SOFT COMPUTING, 38, 933–941. https://doi.org/10.1016/j.asoc.2015.09.046

[18] Wang, J., Liu, J. (2017). Fault-Tolerant Strategy for Real-Time System Based on Evolvable Hardware. JOURNAL OF CIRCUITS SYSTEMS AND COMPUTERS, 26(7). https://doi.org/10.1142/S0218126617501110

[19] Cunha, B.G.P., Ferreira, F.M.F., MARTINS, C. A. P. S. "A complete evolvable hardware architecture based on virtual reconfiguration: evolving combinational circuits". 2020. Dissertação (Mestrado em Engenharia Elétrica) - Pontifícia Universidade Católica de Minas Gerais.

[20] Srivastava, A. K., Gupta, A., Chaturvedi, S., Rastogi, V. (2014). Design and Simulation of Virtual Reconfigurable Circuit for a Fault Tolerant System. In 2014 RECENT ADVANCES AND INNOVATIONS IN ENGINEERING (ICRAIE). 345 E 47TH ST, NEW YORK, NY 10017 USA: IEEE.

[21] Mahdavi, S. J. S., Mohammadi, K. (2010). Reliability enhancement of digital combinational circuits based on evolutionary approach. MICROELECTRONICS RELIABILITY, 50(3), 415–423. https://doi.org/10.1016/j.microrel.2009.11.016

[22] Swarnalatha, A., Shanthi, A. P. (2014). Complete hardware evolution based SoPC for evolvable hardware. APPLIED SOFT COMPUTING, 18, 314–322. https://doi.org/10.1016/j.asoc.2013.12.014

[23] Liang, H., Luo, W., Li, Z., Wang, X. (2010). Designing Combinational Circuits with an Evolutionary Algorithm Based on the Repair Technique. In Tempesti, G and Tyrrell, AM and Miller, JF (Ed.), EVOLVABLE SYSTEMS: FROM BIOLOGY TO HARDWARE (Vol. 6274, pp. 193–201). HEIDELBERGER PLATZ 3, D-14197 BERLIN, GERMANY: SPRINGER-VERLAG BERLIN.

[24] KANG, E.; JACKSON, E.; SCHULTE, W. An Approach for Effective Design Space Exploration. In: Calinescu, R and Jackson, E. (Ed.). FOUNDATIONS OF COMPUTER SOFTWARE: MODELING, DEVELOPMENT, AND VERIFICATION OF ADAPTIVE SYSTEMS. HEIDELBERGER PLATZ 3, D-14197 BERLIN, GERMANY: SPRINGER- VERLAG BERLIN, 2011. (Lecture Notes in Computer Science, v. 6662), p. 33–54. ISBN 978-3-642-21292-5. ISSN 0302-9743

[25] TORRESEN, J. An evolvable hardware tutorial. In: Becker, J and Platzner, M and Vernalde, S. (Ed.). FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS, PROCEEDINGS. HEIDELBERGER PLATZ 3, D-14197 BERLIN, GERMANY: SPRINGER-VERLAG BERLIN, 2004. (Lecture Notes in Computer Science, v. 3203), p. 821–830. ISBN 3-540-22989-2. ISSN 0302-9743

[26] https://github.com/bguerrapc/EHW/blob/master/MATLAB/FUNCTIONS/ovrc.m

Capítulo

6

Armazenamento - Melhores praticas - Construindo sistemas de armazenamento com a performance adequada

Guilherme Friol

Abstract

O minicurso de armazenamento abordará as principais formas de conexões existentes, tais como SATA, SCSI, SAS, iSCSI, Fibre Channel, entre outras. Nele, os participantes poderão compreender as vantagens e desvantagens de cada uma dessas opções, bem como entender a importância da escolha correta na hora de definir a solução de armazenamento adequada para cada situação. Serão discutidos os diversos tipos de dispositivos de armazenamento existentes no mercado, tais como unidades de disco rígido (HDD), unidades de estado sólido (SSD), unidades de fita (tape drive), cartões de memória, entre outros. Os participantes poderão entender as características de cada um desses dispositivos, bem como saber qual a melhor opção.

Resumo

O minicurso de armazenamento aborda várias conexões, como SATA, SCSI, SAS, iSCSI e Fibre Channel, destacando as vantagens e desvantagens de cada uma. Ele enfatiza a importância de escolher a solução certa para diferentes cenários e explora dispositivos de armazenamento, como HDDs, SSDs, unidades de fita e cartões de memória. Isso ajuda os participantes a entender as características de cada dispositivo, facilitando a escolha da opção ideal para suas necessidades.

6.1. Principais tipos de armazenamentos

Os principais tipos de dispositivos de armazenamentos no mercado são: SSD (Disco de estado sólido), HDD (Disco mecânico), Pen-drive, M.2 (NVMe ou SATA), Ópico (CD, DVD, Bluray), LTO (Fita de armazenamento).

Em relação aos discos SATA, podem ser encontrados nos formatos 2.5” (SSDs e HDDs) ou 3.5” (HDDs).

Tamanho máximo de cada tipo de armazenamento:

HDD 2.5” - SATA – 2TB

HDD 3.5” - SATA – 20TB

SSD 2.5” - SATA – 15.36TB

HDD 2.5” - SAS – 2.4TB

HDD 3.5” - SAS – 24TB

SSD 2.5” - SAS – 30.72TB

M.2 - mSATA – 1TB

M.2 - NVME – 4TB

ÓPTICO - Blu-ray XL – 100GB

FITA DAT – LTO 9 – 18TB / 45TB

6.2. SATA versus SAS

SATA e SAS são padrões usados para conectar dispositivos de armazenamento a computadores eservidores.

O SATA é mais comum em computadores pessoais, enquanto o SAS é mais utilizado em servidores e equipamentos empresariais por ser mais rápido e confiável.

Discos rígidos SAS têm maior velocidade de transferência de dados e capacidade de processamento do que discos rígidos SATA, e são mais duráveis com menor taxa de falhas.

6.3. Conector SATA / SAS

A Conexão SATA é capaz de ligar 1 dispositivo por cabo e possui apenas 1 canal de comunicação por porta.

A conexão SAS é capaz de ligar (em teoria) até 65,535 dispositivos através de um unico canal.

A Conexão SAS possui 2 canais de comunicação simultâneos por device.

1.4. CONEXÃO M.2

Os módulos M.2 vêm em tamanhos diferentes e também podem ser utilizados para devices diferentes:

HD, SSD, Wi-Fi, WWAN, Bluetooth, GPS e NFC.

6.5. Tamanho

Os discos M.2, sejam no protocolo SATA ou NVMe possuem uma grande variação de tamanhos.

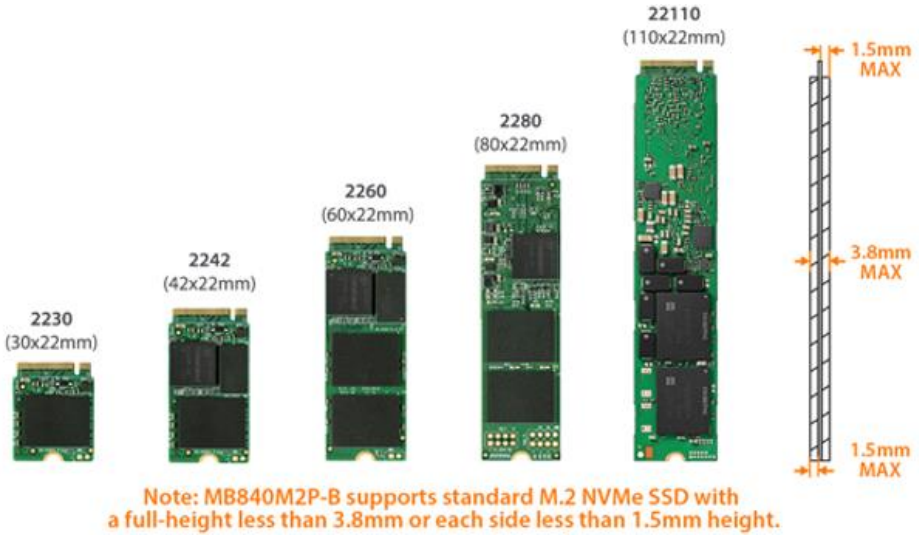


Figura 1. Tamanho dos discos M.2

6.6. Tipos de conectores

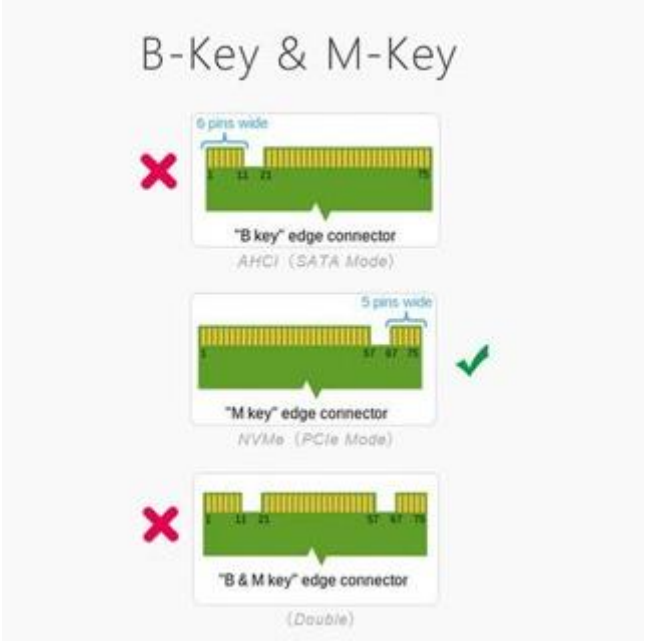


Figura 2. Conectores B e M

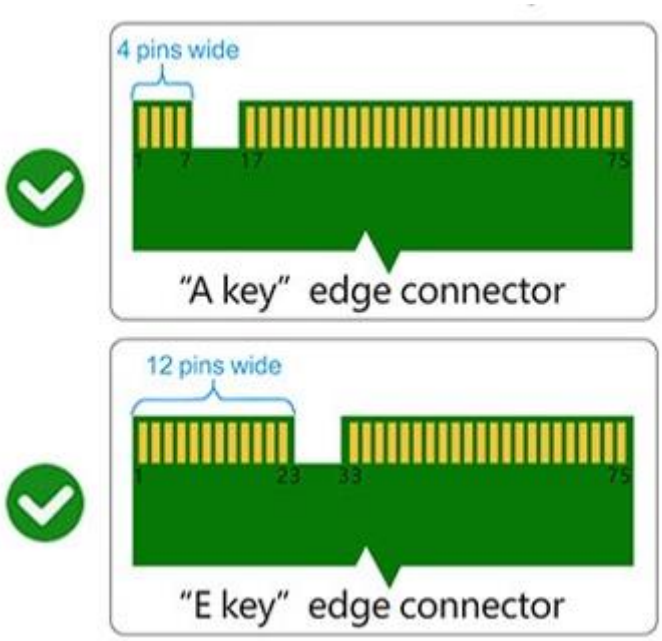


Figura 3. Conectores A e E

6.7. Performance

De maneira geral, é possível verificar que os SSDs e NVMEs apresentam velocidades muito superiores em relação aos HDs SATA e SAS, enquanto que o custo por GB é muito mais elevado.

O HD SAS apresenta uma velocidade superior ao HD SATA e um custo intermediário, e todos os tipos de armazenamento apresentam capacidade RAID.

Tipo de Armazenamento	Velocidade de Leitura (MB/s)	Velocidade de Gravação (MB/s)	Capacidade RAID	Custo por GB (média de mercado)
HD SATA	100-200	50-120	Sim	$0.03
HD SAS	200-400	200-350	Sim	$0.05
SSD	500-550	400-500	Sim	$0.20
NVME	3500-7000	2500-5500	Não	$0.35

Figura 4. Performance em cada tipo de armazenamento

Referências

[1] Kingston, “2 tipos de SSDs M.2: SATA e NVMe.” https://www.kingston.com/br/blog/pc-performance/two-types-m2-vs-ssd, 2023. Online, acessado em Mai 2023.

[2] Kingston, “Perguntas mais frequentes sobre unidades de estado sólido SATA e M.2.” https://www.kingston.com/br/ssd/ssd-faq, 2023. Online, acessado em Mai 2023.

[3] Avast, “SSD ou HDD: de qual você precisa?” https://www.avast.com/pt-br/c-ssd-vs-hdd, 2022. Online, acessado em Mai 2023.