Capítulo 3

Pensamento Computacional Paralelo: Desafios do Presente e do Futuro

Arthur F. Lorenzon e Lucas Mello Schnorr

Programa de Pós-Graduação em Computação (PPGC) - UFRGS

Abstract

Parallel processing becomes a fundamental piece in scientific and technological development. The advancement of artificial intelligence and machine learning relies on complex algorithms and models that require significant computational power for training and inference. In the scientific field, many computational simulations and modeling of natural phenomena demand immense processing power to perform complex calculations within a reasonable timeframe. Regarding emerging technologies like virtual reality and autonomous vehicles, such technologies require real-time processing power to provide immersive experiences and make swift decisions based on constantly changing inputs and data. Last but not least, the field of medicine and genomic researchbenefits from parallel computing to expedite DNA sequencing and identify genetic patterns relevant to diseases and treatments, leading to significant advances in personalized medicine.As the demand for computational power increases, there's a noticeable rise in the number of processing cores within a single chip, an increase in the scale of interconnected computer clusters through low-latency networks, and a higher quantity of general-purpose vector accelerators (GPGPUs). To harness the full potential of this performance, we need to develop applications capable of efficiently exploiting such architectures. The primary approach involves breaking down complex tasks into several smaller parts processed in parallel, potentially resulting in increased performance and the ability to handle larger workloads. Thus, this mini-course aims to introduce the concepts of parallel computational thinking and parallel programming, aiming to instill in participants the important reflexes necessary to break down the solution to a large problem into a set of smaller problems that can be calculated concurrently, effectively identifying the parallelism of the solution. We will also address technological trends for the future of parallel processing. Ultimately, it's

Vídeo com a apresentação do capítulo: `https://youtu.be/_eCci_5upLU`

expected that these elements will lead to the development of suitable parallel solutions for current societal problems.

Resumo

O processamento paralelo se torna uma peça fundamental no desenvolvimento científico e tecnológico. O avanço da inteligência artificial e aprendizagem de máquina dependem de algoritmos e modelos complexos que requerem poder computacional significativo para treinamento e inferência. Na área científica, muitas simulações computacionais e modelagem de fenômenos naturais exigem enormes quantidades de poder de processamento para executar cálculos complexos em um tempo razoável. Com relação às tecnologias emergentes, como realidade virtual e veículos autônomos, tais tecnologias requerem poder de processamento em tempo real para fornecer experiências imersivas e tomar decisões rápidas com base em entradas e dados em constante mudança. Por fim, mas não menos importante, a área da medicina e pesquisa genômica se beneficia da computação paralela para acelerar o sequenciamento de DNA e identificar padrões genéticos relevantes para doenças e tratamentos, levando a significativos avanços na medicina personalizada. Ao mesmo tempo que a demanda por poder computacional aumenta, observa-se um aumento da quantidade de núcleos de processamento em um único chip, um aumento na escala de clusters de computadores interconectados por redes de baixa latência e uma maior quantidade de aceleradores vetoriais de propósito geral (GPGPUs). Para extrair todo este potencial de desempenho, precisamos desenvolver aplicações capazes de explorar tais arquiteturas de maneira eficiente. A principal abordagem é a divisão de tarefas complexas em várias partes menores que são processadas em paralelo, resultando potencialmente em um aumento no desempenho e na capacidade de lidar com cargas de trabalho maiores. Assim, este minicurso tem por objetivo apresentar os conceitos de ***pensamento computacional paralelo*** *e* ***programação paralela****, com o objetivo de instigar nos participantes os reflexos importantes necessários para quebrar a solução de um problema grande em um conjunto de problemas menores que podem ser calculados de maneira concorrente, efetivamente identificando o paralelismo da solução. Abordaremos também as tendências tecnológicas para o futuro do processamento paralelo. Enfim, espera-se que estes elementos levem ao desenvolvimento de soluções paralelas adequadas para problemas atuais da sociedade.*

3.1. Introdução

A computação tem sido objeto de um interesse cada vez maior por nações ao redor do mundo para inclusão de seu currículo como elemento base de escolas. O intuito principal é que a computação ocupe espaços como já fazem a matemática, a física, a geografia, a filosofia e a língua oficial do país. O *pensamento computacional*, ou seja, o método onde quebra-se problemas maiores em subproblemas, é sem dúvida fundamental para os dias de hoje e sobretudo para o futuro, para a próxima geração da sociedade, pois permite resolver problemas de maneira mais eficiente com a programação de computadores. Enquanto saúda-se e encoraja-se esses esforços, ressaltamos aqui a importância coadjuvante mas cada vez mais essencial do **pensamento computacional paralelo**. Além da tradicional quebra em subproblemas, a forma de pensar "em paralelo" exige imaginar atividades e

tarefas que possam ser executadas de maneira concorrente. É inegável a importância desse tipo de método, pois o que se observa é uma tendência inexorável para plataformas computacionais cada vez mais paralelas, compostas de uma grande quantidade de núcleos de processamento. O pensamento computacional paralelo é portanto fundamental para imaginar soluções que possam usufruir das plataformas computacionais modernas com a criação de aplicações paralelas.

O projeto de aplicações paralelas é uma etapa primordial na resolução de problemas complexos da sociedade atual, tais como previsão climática, mitigação de cenários de catástrofe, a procura por fontes renováveis de energia e a simulação de inundações para identificar áreas alagadiças. Em geral essas aplicações exigem um enorme poder computacional pois envolvem equações matemáticas complexas que modelam comportamentos físicos. As simulações decorrentes podem inclusive substituir onerosos experimentos reais, permitindo a criação de soluções inovadoras por um número maior de pesquisadores e empreendedores.

Projetar de maneira adequada uma aplicação paralela envolve uma interdisciplinariedade muito grande, pois se de um lado exige conhecimentos profundos da solução candidata de um problema, de outro exige conhecimentos em poder adaptar os passos paralelos dessa solução para se executar adequadamente a uma plataforma de execução, seja esta um único computador (com múltiplos núcleos de processamento ou múltiplas placas aceleradoras) ou um *cluster* de computadores (vários nós computacionais interconectados por uma rede de interconexão). Enfim, as decisões na fase de projeto da aplicação paralela acabam sendo determinantes para se obter um bom desempenho e um uso eficiente dos recursos de processamento.

Este curso aborda de maneira ampla os conceitos de **pensamento computacional paralelo** e **programação paralela**, com o objetivo de instigar nos participantes os reflexos importantes necessários para quebrar a solução de um problema grande em um conjunto de problemas menores que podem ser calculados de maneira concorrente, efetivamente identificando o paralelismo da solução. Ainda que esta etapa seja independente da plataforma computacional subjacente, pretendemos também abordar escolhas importantes na identificação de paralelismo quando se define uma plataforma alvo específica, assim como as tendências tecnológicas na área de processamento paralelo. Enfim, espera-se que tais intuições do pensamento computacional paralelo levem ao desenvolvimento de soluções adequadas para problemas atuais da sociedade e possam contribuir para o avanço societal.

A Seção 3.2 apresenta o método PCAM [Foster 1995] para exercitar o pensamento computacional paralelo. A Seção 3.3 traz as principais tendências tecnológicas para o futuro do processamento paralelo. A Seção 3.4 traz uma conclusão com reflexões futuras.

3.2. Pensamento Computacional Paralelo: Método PCAM

O método **PCAM**, ilustrado na Figura 3.1, envolve as etapas: particionamento, comunicação, aglomeração e mapeamento. As duas primeiras etapas tem um enfoque na escalabilidade de uma solução para um problema, ou seja, procuramos definir algoritmos capazes de resolver o problema de maneira mais paralela possível, com maior concorrência entre as unidades de processamento. Nas duas últimas etapas, de aglomeração e mapeamento, a preocupação do projetista se foca na preocupação com o desempenho

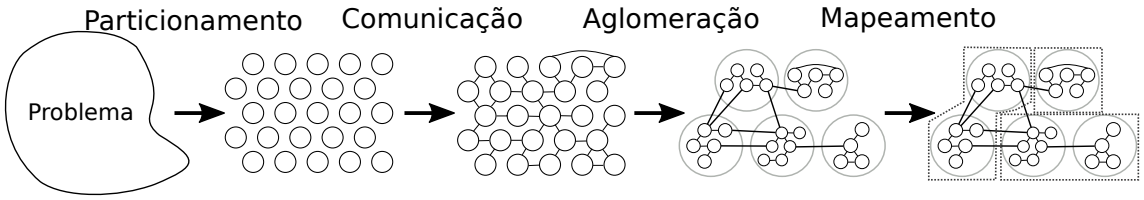

Figura 3.1. Metodologia PCAM com suas quatro etapas.

computacional (tempo de execução). São nestas duas últimas etapas que o conhecimento da configuração da plataforma de execução se torna vital.

Segue uma breve descrição de cada uma das etapas. No **Particionamento**, as operações que resolvem um determinado problema a devem ser quebradas em pedaços pequenos. O objetivo principal é detectar o paralelismo nestas operações. Na **Comunicação**, devemos definir quais são as atividades de comunicação necessárias para que a resolução de um problema, já dividida em pedaços, funcione de maneira apropriada e sem erros. Na **Aglomeração**, devemos avaliar se a solução respeita requisitos de desempenho computacional e custos de implementação. Enfim, no **Mapeamento**, devemos atribuir as tarefas, estática ou dinamicamente, às unidades de processamento. Aqui devemos maximizar o uso de recursos computacionais e minimizar atividades de comunicação.

Em um cenário ideal, espera-se que uma aplicação paralela criada no âmbito do processo metodológico PCAM seja capaz de explorar de maneira eficiente uma quantidade indeterminada de unidades de processamento. Mesmo assim, observa-se frequentemente a aplicação do método especificamente para uma plataforma computacional, como aquelas que serão abordadas na Seção 3.3 onde veremos tendências tecnológicas. Criar uma aplicação com portabilidade de desempenho é uma tarefa bastante difícil pois envolve utilizar algoritmos adaptativos em função da execução e do ambiente. Este tópico permanece como objeto de investigação.

3.2.1. Particionamento

A etapa de particionamento envolve a descoberta de oportunidades para execução paralela. A ideia é identificar o maior número possível de pequenas tarefas. Com isso, procura-se estabelecer qual o menor subproblema possível, consistindo de operações que são executadas sequencialmente. Este menor subproblema possível é então chamado de tarefa. Como consequência desse esforço, esse grão pequeno permitirá uma maior flexibilidade para a criação de algoritmos paralelos que suportem uma variadade maior de plataformas computacionais.

Normalmente, um tamanho de tarefa demasiadamente pequeno pode incutir em uma perda de desempenho no que diz respeito a quantidade de comunicações e ao gerenciamento da enorme quantidade de tarefas pequenas resultantes. Isso leva naturalmente a uma junção das operações de várias tarefas pequenas, efetivamente mudando a *granularidade* das tarefas. No âmbito do modelo PCAM, esta reflexão é relegada para mais tarde, na etapa de aglomeração.

Existem dois tipos de particionamento: de dados e de operações. O particionamento de dados é mais comum, sendo conhecida também por **decomposição de domínio**. Ela tem

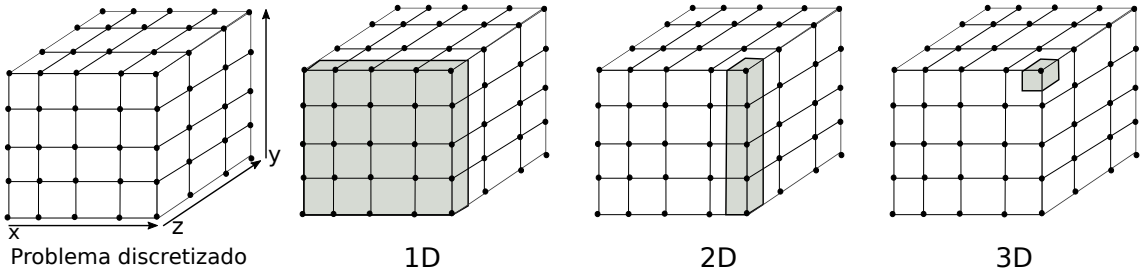

Figura 3.2. Três tipos de decomposição de domínio para o problema tridimensional à esquerda, com uma (1D), duas (2D) e três (3D) dimensões. O tamanho do dado na abordagem tridimensional é o menor possível (representado pelo conjunto de pontos na grade), pois envolve apenas um ponto na grade.

por objetivo quebrar o problema em pedaços suficientemente pequenos. Por simplicidade, esse processo é conduzido de forma a obter pedaços que sejam também de tamanhos idênticos, de forma a facilitar as etapas seguintes do método PCAM. Cada pedaço terá portanto os dados, resultante da partição pelo método, e as operações associadas. É importante quantificar o custo destas operações de forma que elas sejam consideradas, ainda que de maneira secundária, na definição do tamanho da partição de dados. O segundo tipo de particionamento envolve os tipos de operações (instruções) que devem ser computadas pela aplicação paralela, sendo conhecida por **decomposição funcional**. Neste caso, o projetista deve identificar partes no futuro código da aplicação que são funcionalmente independentes, prevendo sua execução de maneira concorrente. Ainda que idealmente o processo de particionamento possa se preocupar com a divisão dos *dados* e das *operações* conjuntamente, é comum adotar um ou outro tipo de decomposição de maneira independente.

A Figura 3.2 demonstra exemplos de decomposição de domínio de um problema tridimensional que já foi discretizado conforme ilustração na esquerda da figura. Esta discretização é representada pelos pontos do espaço tridimensional, com cinco coordenadas no eixo `x` e no eixo `y` e quatro no eixo `z`. O problema foi portanto discretizado em 100 pontos. Esta discretização pode então ser particionada em uma dimensão (1), com planos ao longo do eixo `z`, ou através de colunas (2D) ao longo do eixo `y`, ou através de uma partição tridimensional (3D) que envolve apenas um ponto. É esta última opção que permite a maior flexibilidade nas próximas etapas PCAM pois o tamanho da partição engloba um único ponto do domínio discretizado.

3.2.2. Comunicação

É comum, em um programa paralelo, que as tarefas necessitem trocar informações para realizar suas operações. Em PCAM, a etapa de comunicação envolve justamente o projeto destas atividades de troca de dados. Cenários onde as tarefas são independentes, portanto sem a necessidade de comunicação, são chamados de soluções *trivialmente paralelizáveis*. Nestes casos, basta realizar o particionamento e as etapas de aglomeração e mapeamento de PCAM.

Uma das principais preocupações com as atividades de comunicação é que elas possam ocorrer da maneira mais concorrente possível. Esse estado ideal pode ser atingido de

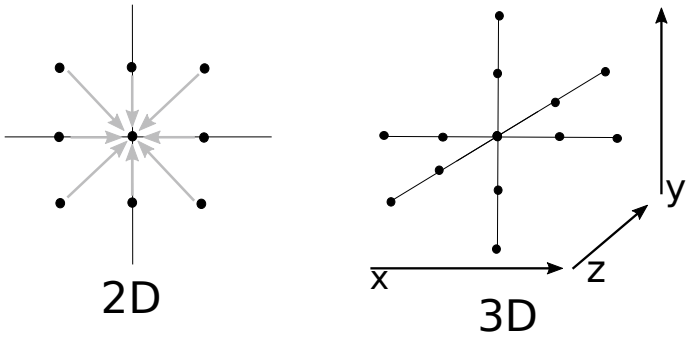

Figura 3.3. Comunicação local em particionamentos 2D (esquerda) e 3D (direita).

diferentes formas, através de variados padrões de comunicação. Uma classificação destes padrões pode seguir os seguintes eixos: comunicação local ou global, estruturada ou não, estática ou dinâmica, e síncrona ou não. Estes eixos são detalhados abaixo.

Local e Global. Uma comunicação local é obtida quando a operação de computação de uma determinada tarefa necessita dados de um pequeno número de tarefas vizinhas. A operação é conhecida por *stencil*, podendo ser configurada para um ambiente de variadas dimensões, de acordo com o domínio do problema. Por exemplo, na Figura 3.2 a quantidade de vizinhos imediatos no 3D é de seis, quatro de cada lado, um acima e outro abaixo. Neste caso, seis comunicações seriam necessárias antes da operação de cálculo. Quando o particionamento é bidimensional, como representado na ilustração 2D da Figura 3.3, a comunicação de todos os vizinhos pode ser necessária para um passo de simulação. Neste caso, existem oito operações de comunicações necessárias antes de efetuar as operações de cálculo da tarefa central. A localidade das comunicações podem variar bastante em função da complexidade da aplicação. Alguns cálculos podem exigir, por exemplo, dados de vizinhos de segunda ordem, conforme a ilustração 3D da Figura 3.3.

Uma operação de comunicação global pode envolver muitas tarefas, potencialmente todas aquelas que participam da aplicação paralela. Operações frequentemente reconhecidas como globais são aquelas de difusão massiva (*broadcast*) onde uma tarefa envia um dado para todas as outras, ou uma tarefa de redução, onde os dados de todas as tarefas são reduzidos por intermédio de um operador binário até uma única tarefa. Uma comunicação global pode ser implementada através de um algoritmo mestre-trabalhador. Neste caso, uma determinada tarefa fica responsável por recepcionar os dados de todas as outras que participam da computação. Este algoritmo tem duas características que tornam-no incapaz de atingir uma boa escalabilidade. Ele é centralizado e sequencial, uma vez que a tarefa mestre recebe as informações em uma determinada ordem. Uma forma mais eficiente de obter a mesma funcionalidade deste algoritmo é empregar uma árvore N-ária para difundir, ou receber, o dado mais rapidamente. Neste caso, as comunicações nas folhas e nós intermediários da árvore podem acontecer simultaneamente. O resultado é uma estrutura de comunicação regular na qual cada tarefa se comunica com poucos vizinhos próximos.

Estruturada ou não estruturada. As situações apresentadas até o momento são exemplos de uma estrutura de comunicação estática, ondes as tarefas tem vizinhos claramente definidos e imutáveis em função do particionamento estabelecido na etapa anterior. Neste

contexto as comunicações são frequentemente fixas, ditas estruturadas, e não evoluem ao longo da execução da aplicação paralela. Em outros casos, a grade de discretização pode seguir padrões mais complexos, conhecidos como não estruturados e irregulares. Por exemplo, um objeto irregular tal como o pulmão de uma pessoa pode ser melhor modelado por uma grade composta por formas simples, tais como triângulos, tetraedros, etc. Estas grades podem ser descritas por grafos, onde os vértices representam as tarefas e as arestas representam comunicação. Nestes casos, as atividades de comunicação entre as tarefas são mais complexas, envolvendo por vezes mais vizinhos em determinadas regiões.

Estática ou dinâmica. Grades de discretização podem ser regulares (veja exemplo na Figura 3.2) ou irregulares (exemplo na Figura 3.4), tais como um objeto modelado por formas como triângulos, etc. Uma diferença fundamental que pode afetar as demais etapas do projeto PCAM é se tais grades são estáticas ou dinâmicas. No caso de grades estáticas, a discretização é fixa desde a concepção na etapa de particionamento do projeto até a execução do código. No caso de grades dinâmicas, a discretização pode mudar em função da execução da aplicação paralela. Programas complexos podem aumentar a fidelidade de simulação em torno de objetos móveis em uma simulação ou em determinadas regiões de interesse, como bordas ou objetos relevantes.

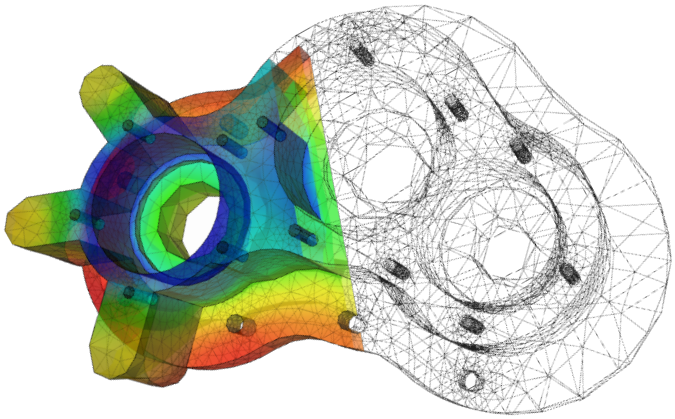

Figura 3.4. Grade de particionamento irregular tridimensional, representada por um grafo onde cada nó é uma tarefa e cada aresta é uma operação de comunicação (Artigo da Wikipedia em Alemão sobre Elementos Finitos).

Padrões irregulares de comunicação normalmente não afetam a etapa de particionamento. No exemplo da Figura 3.4, pode-se observar que algumas regiões tem uma intensidade de tarefas maior (pela proximidade física) que outras. O particionamento de um grafo como este pode implicar que cada nó de um grafo se torne uma tarefa e suas arestas se tornem comunicações. No entanto, uma grade irregular pode complicar bastante a condução das etapas de aglomeração e mapeamento. Por exemplo, ainda que a grade seja estática, a etapa de aglomeração pode ser complicada pois envolve antecipar o custo computacional das tarefas e a quantidade de dados da comunicação, de forma a criar grupos de tarefas que tenham por um lado um peso similar e que minimizem as comunicações. No caso da grade ser dinâmica, os algoritmos que realizam a aglomeração podem ser necessários durante a execução do programa incutindo em sobrecargas que devem ser pesados contra os benefícios trazidos por um melhor agrupamento de tarefas.

Síncrona ou Assíncrona. Até agora, vimos conceitos que consideram comunicação síncrona, onde as duas tarefas envolvidas na troca de dados estão cientes quando a operação acontece. Na comunicação assíncrona, por outro lado, as tarefas que possuem os dados, e que são responsáveis pelo envio, não estão cientes do momento quando as tarefas receptoras precisarão efetivamente dos dados da comunicação. Sendo assim, as tarefas receptoras devem registrar a necessidade de um dado que eventualmente será satisfeito, de maneira assíncrona, pela tarefa responsável por enviar o dado.

A grande vantagem de comunicações assíncronas é que elas podem ser utilizadas para esconder as comunicações. Em *middlewares* sofisticados de comunicação, a troca de dados de maneira assíncrona pode ser implementada de uma forma que a aplicação não fica bloqueada em nenhum momento esperando a conclusão do envio/recepção. Sendo assim, a aplicação paralela pode antecipar o registro da necessidade de um dado de forma que quando ele for necessário esse dado já tenha sido recebido. Esse conceito serve também do lado do envio, onde o gerador do dado registra o envio para quem precisa do dado assim que ele for gerado, potencializando a comunicação assíncrona.

3.2.3. Aglomeração

As etapas precedentes da metodologia PCAM permitem o particionamento e a definição das comunicações necessárias para a resolução paralela de um problema. O resultado destas etapas é um algoritmo abstrato que contém potencialmente muitas tarefas, tendo em vista que o objetivo é identificar a menor operação possível que possa ser executada concorrentemente com as demais. Esse algoritmo abstrato, distante da realidade, é normalmente ruim por ter tarefas demais, visto que somente o gerenciamento dessa quantidade enorme de tarefas é prejudicial para o desempenho. A etapa de *aglomeração* tem por objetivo tornar o algoritmo abstrato das etapas precedentes em algo mais realista, de acordo com os limites impostos pela configuração da plataforma de execução alvo. O objetivo principal é obter um programa eficiente nesta plataforma. Para atingir tal objetivo, é importante avaliar, analítica ou experimentalmente, o benefício da aglomeração de tarefas através do seu impacto em diretivas de comunicação e no tempo de execução. Abaixo são apresentados tópicos relacionados a granularidade de tarefas, a relação entre superfície e volume no particionamento, e uma discussão sobre replicação de cálculo e formas de evitar a comunicação.

Granularidade de tarefas. Na etapa de particionamento o objetivo é baseado na premissa de quanto mais tarefas melhor, ou seja, deve-se encontrar o menor conjunto de operações que possa ser executada sequencialmente. No entanto, observa-se que esse tipo de particionamento fino pode levar a elevados custos de comunicação que prejudicam o desempenho da aplicação, tendo em vista que a unidade de processamento para de executar código útil para se ocupar de envios e recepção de dados. A aglomeração permite então tornar as tarefas maiores, e na medida que isso ocorre, pode haver um efeito positivo da redução do custos de comunicação.

A Figura 3.5 mostra exemplos de aglomeração de tarefas a partir do particionamento original com uma tarefa por ponto, ilustrada na esquerda da figura. As quatro opções de aglomeração, ilustradas no centro esquerda, ilustram uma aglomeração de pontos horizontal e vertical (na parte superior da figura), e dois planos possíveis (na parte inferior).

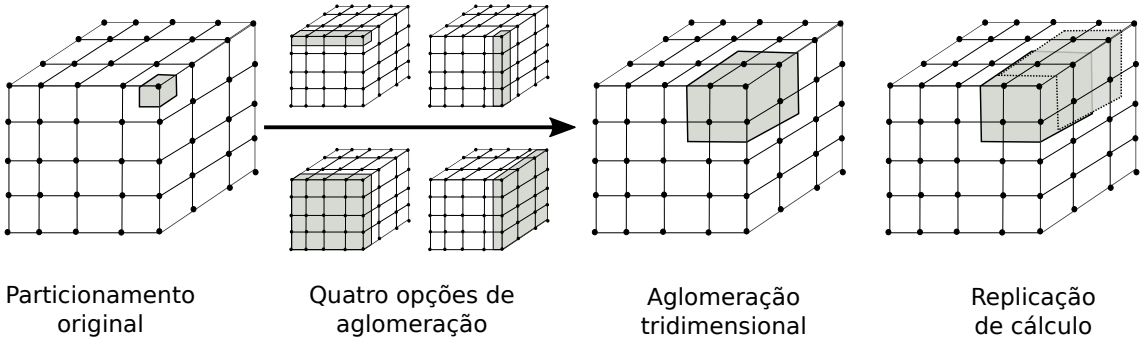

Figura 3.5. Exemplos de aglomeração de tarefas a partir do particionamento original (esquerda), e replicação de cálculo através da sobreposição de dois conjuntos aglomerados na decomposição de domínio (direita).

Enfim, uma aglomeração mais efetiva é a aquela tridimensional, como na ilustração do centro direita onde o bloco aglomera pontos em todas as três dimensões, reduzindo o perímetro do bloco, ou seja, as bordas que exigem comunicação com as tarefas vizinhas.

A aglomeração serve sobretudo para escolher o nível certo de concorrência que extrai o máximo de desempenho da plataforma de execução. Isso envolve então a redução das comunicações mas também pode ser influenciada por outras estratégias. Por exemplo, pode-se agregar dados a serem comunicados de forma que o envio seja feito com uma única operação ao invés de múltiplos envios. Isso permite evitar a latência da rede de interconexão. Pode-se também encontrar a menor quantidade de tarefas que maximize o desempenho, tendo em vista que que o excesso de tarefas pode ser penalizado pelo custo de gerenciamento das mesmas.

Outro ponto relevante na etapa de aglomeração é que a decisão sobre a granularidade das tarefas deve ser configurável. O programa deve ser capaz de permitir uma certa adaptabilidade tendo em vista a evolução dos computadores que podem tornar obsoleta uma determinada decisão de aglomeração.

Relação entre superfície e volume. A etapa de aglomeração traz o benefício de poder reduzir a quantidade de comunicação, como ilustrado no exemplo da Figura 3.6. Do lado esquerdo nós temos um particionamento fino de 8×8, com um total de 64 tarefas (representadas pelos círculos). Considerando que cada tarefa deve enviar 1 dado para cada um dos quatro vizinhos imediatos (conforme ilustrado nas tarefas em tons de cinza mais escuro), nós temos um total de 256 operações de comunicação cada uma com 1 dado. Ao realizar uma aglomeração bidimensional com um fator de 16 para 1, obtemos uma grade como aquela ilustrada na direita da figura, com quatro tarefas. Nesta configuração, são reduzidas não somente a quantidade de operações de comunicação para apenas 16 (pois cada uma das quatro tarefas se comunica com quatro vizinhos), mas também a quantidade de dados comunicados, pois envolve apenas o perímetro dos dados bidimensionais gerenciados por uma tarefa (os quadrados em tons cinza escuro na figura). Essa relação entre superfície e volume, ilustrada na figura através de um exemplo 2D, permite reduzir as necessidades de comunicação.

O efeito da etapa de aglomeração em grades não-estruturadas, como aquela exemplificada na Figura 3.4, são mais complexas de serem realizadas. Existem técnicas especializadas

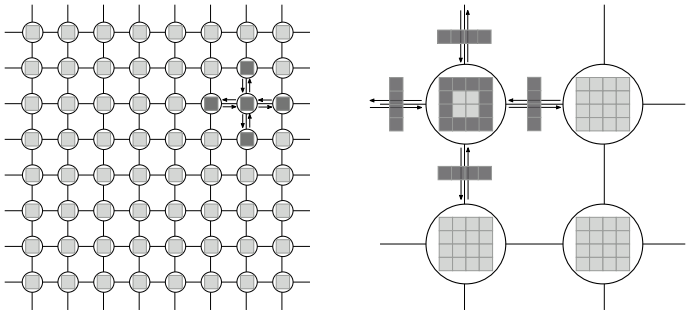

Figura 3.6. Efeito do aumento da granularidade nos custos de comunicação: em uma grade 8×8 (esquerda), o custo total de comunicação é de 256 mensagens (cada tarefa realiza 4 comunicação) cada uma com 1 dado (256 dados no total); em uma grade 2×2 com 4 tarefas (direita), apenas 16 comunicações são necessárias, cada uma com 4 dados para um total de 64 dados.

que tentam equalizar o peso das partições ao mesmo tempo que reduzem as bordas de comunicação. Essas técnicas permitem portanto o balanceamento da carga computacional e são rapidamente apresentadas na Seção 3.2.4 sobre Mapeamento.

3.2.4. Mapeamento

A quarta e última etapa da metodologia PCAM, chamado mapeamento, consiste em definir onde cada tarefa será executada. O mapeamento em si em um problema difícil pois precisa ser explícito em supercomputadores de alto desempenho. Inexiste um método automático para realizar o mapeamento, embora soluções simples possam ser aplicadas, sem que o desempenho seja o melhor possível. Os requisitos fundamentais na atividade explícita de mapeamento envolve (a) colocar tarefas concorrentes em unidades de processamento diferentes, de forma que tais tarefas sejam de fato executadas em paralelo e (b) alocar tarefas que se comunicam frequentemente em locais próximos na topologia de interconexão, tanto física quanto lógica. Estas duas condições podem entrar em conflito. Por exemplo, se considerarmos apenas a localidade podemos ser levados a colocar todas as tarefas em uma unidade de processamento, algo que certamente não é bom visto que as tarefas competiriam pelo mesmo recurso.

Em alguns casos a etapa de mapeamento é simples. Por exemplo, aplicações que possuem tarefas homogêneas entre si e ao longo da execução são apropriadas para mapear em supercomputadores com capacidade homogênea (todos as unidades de processamento são idênticas). Nestes casos, pode-se inclusive aglomerar as tarefas de maneira que tenhamos apenas uma tarefa por processador.

Por outro lado, a etapa de mapeamento se torna mais complexa quando as tarefas tem custos computacionais diferentes, ainda que estes custos sejam estáticos ao longo do tempo. Nestes cenários, algoritmos de *balanceamento de carga* podem ser úteis para equilibrar os custos entre os recursos computacionais. Métodos descentralizados de balanceamento de carga tem mais chance de se adaptar a evolução dos supercomputadores, especialmente no quesito de escalabilidade, pois não há uma única entidade controladora. O cenário mais complexo para a etapa de mapeamento ocorre quando a carga computacional é heterogênea tanto entre as tarefas quanto ao longo da execução, ou seja, o custo de uma tarefa evolui conforme a execução da aplicação avança. Neste caso, deve-se aplicar pre-

ferencialmente algoritmos de *balanceamento de carga dinâmicos*, capazes de monitorar a evolução da carga computacional ao longo do tempo. Algoritmos que exigem apenas um conhecimento local são preferíveis pois não requerem possíveis comunicações coletivas globais entre todas as tarefas.

Enfim, os algoritmos oriundos de decomposição funcional tem uma abordagem diferente de mapeamento. Eles podem ser mapeados preferencialmente por algoritmos de escalonamento de tarefas, antecipando tempo de ociosidade em processadores.

Balanceamento de carga. Algoritmos de balanceamento de carga são também conhecidos por algoritmos de particionamento. Eles tem por objetivo aglomerar tarefas finas (oriundas da etapa de particionamento) de uma partição inicial até encontrar uma tarefa cujo tamanho seja apropriado para uma determinada plataforma de execução. Existem quatro técnicas principais de balanceamento de carga: métodos baseados em bisseção recursiva, algoritmos locais, métodos probabilistas e mapeamento cíclicos.

Os métodos baseados em *bisseção recursiva* particionam o domínio do problema de maneira iterativa, com informações globais, sempre levando-se em conta o custo de um subdomínio e a minimização da comunicação entre as partições. Esses métodos são considerados algoritmos de divisão e conquista. Um exemplo é o algoritmo de Barnes-Hut [Barnes and Hut 1986]. Existem várias variantes dos métodos de bisseção recursiva. A forma mais simples, que não considera o custo e a quantidade das comunicações, consista em realizar a bisseção recursiva unicamente baseada nas coordenadas do domínio: sempre se divide a coordenada mais larga. Uma segunda variante do método de bisseção se chama de método desbalanceado pois tem um enfoque unicamente no controle das comunicações, reduzindo o perímetro das partições. Enfim, uma terceira variante mais sofisticada e mais geral é a bisseção recursiva de grafo, útil para grades não-estruturadas (veja Figura 3.4). Esta variante usa a informação de conectividade do grafo para reduzir o número de arestas que cruzam a fronteira entre dois subdomínios.

Uma técnica alternativa com menor intrusão consiste nos algoritmos de *balanceamento de carga locais*. Eles são relativamente baratos pois necessitam apenas de informações da tarefa em questão e de seus vizinhos. Isso possibilita também uma execução paralela, ou seja, todas as tarefas podem executar o algoritmo local simultaneamente. No entanto, a falta de coordenação global leva em geral a um particionamento pior daquele obtido por particionadores globais.

O terceiro tipo de método de balanceamento de carga consiste em *métodos probabilistas*. Com um custo baixo de execução e uma boa escalabilidade, ao mesmo tempo que ignora completamente o custo e quantidade de comunicações, algoritmos probabilistas alocam as tarefas nos recursos computacionais de maneira aleatória. Essa abordagem funciona melhor quando há muitas tarefas, pois as chances são maiores de fazer com que os recursos computacionais recebam carga de trabalho similar.

Enfim, os *mapeamentos cíclicos* são uma quarta forma de realizar o mapeamento da carga nos recursos computacionais. Baseado em um particionamento já definido na primeira etapa do método PCAM, a técnica distribui aos recursos, de maneira cíclica, as partições. Parte-se do princípio que existem bastante tarefas, que os custos com comunicação são baixos através de uma localidade de operações e dados reduzidas. Um mapeamento cí-

clico pode ser realizado utilizando como entrada os blocos de tarefas, criados na etapa de aglomeração.

Escalonamento de tarefas. Os algoritmos de escalonamento de tarefas podem ser usados em situações com requisitos de localidade fracos. Em geral, eles consideram as tarefas como um conjunto de problemas que devem ser resolvidos, sendo colocados em uma "piscina" de subproblemas. Uma heurística de escalonamento deve então decidir, durante a execução e de maneira dinâmica, em qual recurso computacional um determinado problema, recuperado da piscina, será alocado. O desenvolvimento de uma heurística que englobe de um lado a necessidade de reduzir os custos de comunicação e de outro o conhecimento global do sistema (para efetuar um bom balanceamento de carga) é o principal desafio. Embora heurísticas centralizadas tem um bom conhecimento da plataforma, em geral eles não são escaláveis para centenas de unidades de processamento. Para mitigar esse problema, existem heurísticas que criam uma estrutura hierárquica de gerenciadores, permitindo uma alternativa mais escalável com uma visão semi-global do estado da plataforma. Enfim, no outro extremo existem heurísticas totalmente descentralizadas: cada processador mantém uma "piscina" de tarefas e trabalhadores podem requisitar mais tarefas quando se tornam ociosos. Heurísticas probabilistas, potencialmente hierárquicas de acordo com a topologia da plataforma computacional, e associadas a roubo de tarefas se enquadram nesta classe de algoritmos.

3.3. Tendências Tecnológicas

Esta seção aborda as tendências tecnológicas modernas que moldam o presente e o futuro da área de processamento paralelo. Serão abordadas desdes conceitos estabelecidos, tais como computação de alto desempenho na nuvem, até conceitos mais inovadores como computação neuromórfica de alto desempenho.

3.3.1. Computação de Alto Desempenho na Nuvem

A computação em nuvem é uma forma de disponibilizar recursos de computação, como armazenamento, processamento, rede, software, entre outros, por meio da internet. Ela permite o acesso a uma infraestrutura escalável, flexível, econômica e segura para executar aplicações paralelas de alto desempenho. Alguns dos benefícios da computação em nuvem para a computação paralela de alto desempenho incluem a redução de custos operacionais e de manutenção; aumento da disponibilidade e confiabilidade; adaptação dinâmica à demanda e ao desempenho; acesso a tecnologias avançadas e inovadoras; entre outros. Alguns dos exemplos de serviços de computação em nuvem voltados para a computação paralela de alto desempenho disponíveis compreendem a *Amazon Web Services* (AWS), *Google Cloud Platform* (GCP), *Microsoft Azure*, *IBM Cloud*, entre outros.

Neste cenário, a computação de alto desempenho está sendo empregada como um serviço na nuvem (*HPCaaS*), conforme ilustrado na Figura 3.7. Ela mostra um exemplo de infraestrutura para execução de cargas de trabalho paralelas na nuvem, onde os usuários finais submetem a carga de trabalho através da *Internet* e o ambiente de nuvem (*HPCaaS*) é responsável por alocar máquinas e implantar a carga de trabalho para execução em sistemas computacionais de alto desempenho [Paillard et al. 2015][Navaux et al. 2023].

Um dos principais desafios associados à computação de alto desempenho em ambientes

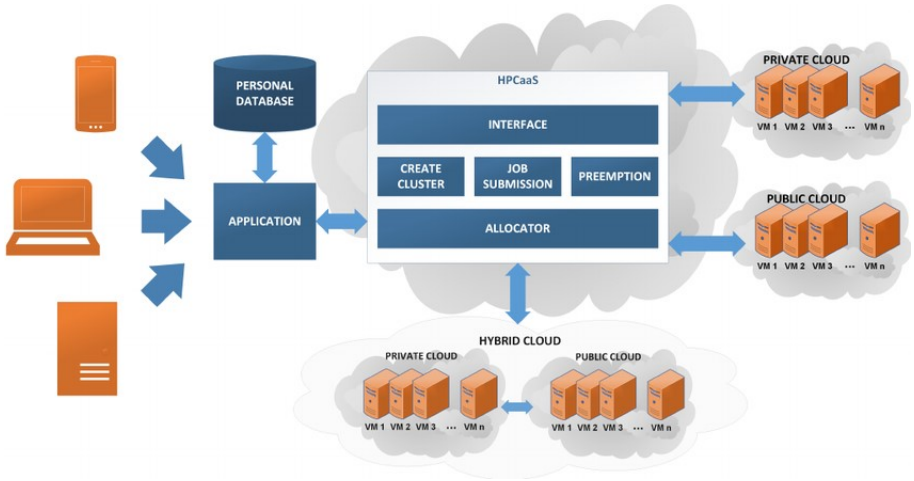

Figura 3.7. Computação de Alto Desempenho *as a service* na Computação na Nuvem [Paillard et al. 2015]

de nuvem está relacionado à eficiência do desempenho. Embora a nuvem ofereça escalabilidade em termos de recursos computacionais, a necessidade de virtualização e compartimentalização pode introduzir uma sobrecarga adicional, levando a alta latência e perda de desempenho. Consequentemente, a otimização das cargas de trabalho de HPC para uma execução eficaz em ambientes virtualizados exige ajustes e otimizações específicas para mitigar essa sobrecarga.

De maneira similar, a gestão de recursos é outra área desafiadora. Enquanto ambientes HPC tradicionais têm controle detalhado sobre os recursos do sistema, permitindo ajustes precisos para otimizar o desempenho, a natureza compartilhada da nuvem pode resultar em conflitos de recursos e competição entre diferentes instâncias de máquinas virtuais. Portanto, uma alocação e monitoramento eficazes dos recursos são e continuarão sendo essenciais para garantir que as cargas de trabalho HPC possam acessar a capacidade de computação necessária.

Dado que as aplicações HPC frequentemente dependem de comunicação intensiva entre nós de processamento, a latência da rede no contexto da nuvem pode também se tornar um problema crítico para o desempenho das aplicações. Nesse sentido, o projeto de uma arquitetura de rede de alto desempenho na nuvem, caracterizada por baixa latência e alta largura de banda, torna-se fundamental para atender às exigências de comunicação das aplicações HPC. Além disso, garantir que o processo de comunicação durante a migração de dados e cargas de trabalho para a nuvem não comprometa a privacidade dos usuários é um desafio adicional, especialmente considerando que as aplicações HPC frequentemente lidam com dados sensíveis ou confidenciais.

Uma tendência adicional é a proliferação significativa de provedores de computação em nuvem de alto desempenho. Consequentemente, a escolha de um provedor específico pode restringir a interoperabilidade e portabilidade das aplicações HPC, dificultando a transferência de cargas de trabalho entre diferentes nuvens ou para ambientes locais. Além disso, uma vez que as aplicações HPC normalmente são desenvolvidas para ambientes específicos, sua adaptação para aproveitar a escalabilidade e recursos da nuvem

pode se tornar um processo complexo e demorado devido à diversidade de recursos heterogêneos disponíveis.

Resumidamente, a integração da computação de alto desempenho em ambientes de nuvem oferece diversas oportunidades, mas também apresenta desafios substanciais. Otimização de desempenho, alocação eficaz de recursos, segurança sólida e adaptação de aplicações são elementos críticos que requerem abordagens cuidadosas para garantir o sucesso da computação de alto desempenho na nuvem. Além disso, à medida que a tecnologia evolui, soluções inovadoras e adaptativas continuarão a ser desenvolvidas para superar esses desafios e permitir que as aplicações HPC alcancem todo o seu potencial na nuvem.

3.3.2. Computação Quântica de Alto Desempenho

A computação quântica é uma forma de explorar os fenômenos da mecânica quântica para realizar operações que seriam impossíveis ou muito lentas em computadores clássicos. A computação quântica para alto desempenho é um campo que busca aproveitar as vantagens dos computadores quânticos para resolver problemas complexos e desafiadores que exigem muitos recursos computacionais. Os computadores quânticos são dispositivos que usam as propriedades da física quântica, como a superposição e o emaranhamento, para manipular unidades de informação chamadas *qubits*. Os *qubits* podem representar simultaneamente os valores 0 e 1, o que permite que os computadores quânticos realizem operações paralelas e explorem um espaço de soluções muito maior do que os computadores clássicos.

Ela tem a promessa de revolucionar diversos campos da ciência, tecnologia e negócios, como criptografia, otimização, aprendizado de máquina, química, física, medicina, entre outros. Embora ainda esteja em desenvolvimento, a computação quântica já vem mostrando avanços significativos e desafiando os limites da computação tradicional. Conforme destacado pelos autores em [Fu et al. 2016], um computador quântico irá sempre consistir de componentes de computação convencional e quântica pois algoritmos quânticos consistirão de partes clássicas e quânticas e, portanto, serão executadas por seus respectivos blocos de computação. Além disso, o computador quântico requer monitoramento muito próximo e, se necessário, correção pela lógica clássica. A Figura 3.8 ilustra uma visão geral da pilha do sistema de um computador quântico, onde as camadas superiores representam os algoritmos para os quais as linguagens e compiladores precisam ser desenvolvidos para que aplicações possam explorar o hardware quântico subjacente. Neste ponto, os *qubits* são definidos como *qubits* lógicos. A próxima camada é o conjunto de instruções da arquitetura (*Q Instruction Set Architecture* – QISA). Assim, o compilador irá traduzir instruções lógicas em instruções físicas que pertence a QISA. A camada de correção de erro (*Quantum Error Correction*) é responsável pela detecção e correção de erro, onde recebe dados para identificar possíveis erros e irá realizar as correções necessárias.

Existem vários desafios para desenvolver e implementar essa tecnologia, como a fragilidade dos *qubits*, a correção de erros, a escalabilidade, a programação e a integração com sistemas clássicos. Algumas empresas e instituições estão trabalhando para superar esses desafios e oferecer soluções de computação quântica para alto desempenho. Por exemplo, a IBM anunciou recentemente a criação de um processador quântico avançado chamado

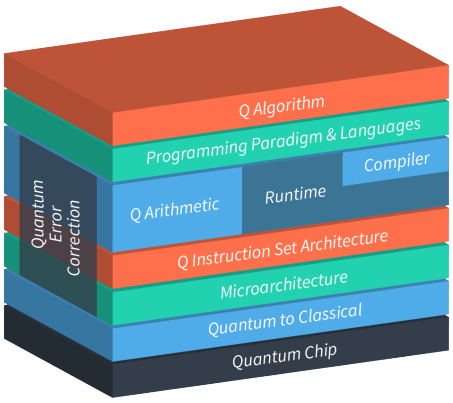

Figura 3.8. Visão geral da pilha do sistema de um computador quântico [Fu et al. 2016]

Eagle, que tem 127 *qubits* e ultrapassa o limite de processamento existente em máquinas quânticas [Chow et al. 2021]. A Microsoft oferece uma plataforma aberta chamada Azure Quantum, que permite aos usuários acessar diferentes tipos de hardware quântico e desenvolver algoritmos quânticos. Outras empresas que estão na vanguarda da computação quântica para alto desempenho são *Google*, *Amazon*, *Intel*, *Alibaba*, *Fujitsu* e *D-Wave*.

As interfaces de programação paralela que podem ser utilizadas para programar processadores quânticos dependem do tipo de hardware quântico e do modelo de computação quântica que se deseja usar. Existem diferentes paradigmas de computação quântica, como o modelo de circuitos quânticos, o modelo de computação adiabática, o modelo de máquinas de Turing quânticas e o modelo de computação topológica. Cada um desses modelos requer uma forma diferente de representar e manipular os *qubits* e os algoritmos quânticos. Assim, algumas das interfaces de programação paralela mais conhecidas e usadas para programar processadores quânticos são:

- *Q#*: É uma linguagem de programação quântica desenvolvida pela Microsoft, que faz parte do Quantum Development Kit[1]. O Q# é baseado no modelo de circuitos quânticos e permite a criação de programas quânticos híbridos, que combinam código quântico e clássico. O Q# também oferece ferramentas para simular, depurar, testar e otimizar os programas quânticos. Ele pode ser usado com o Azure Quantum, que é uma plataforma aberta para executar programas quânticos em diferentes tipos de hardware quântico.
- *Qiskit*: É um framework de software para programação quântica desenvolvido pela IBM, que faz parte do IBM Quantum Experience[2]. O *Qiskit* é baseado no modelo de circuitos quânticos e permite a criação de programas quânticos usando linguagens de programação como *Python* e *C++*. O *Qiskit* também oferece ferramentas para simular, visualizar, analisar e otimizar os programas quânticos. O Qiskit pode ser usado com o

[1]https://github.com/microsoft/Quantum
[2]https://www.ibm.com/quantum

IBM Quantum Cloud, que é uma plataforma que fornece acesso a diferentes tipos de hardware quântico.

- *Cirq*: É um framework de software para programação quântica desenvolvido pelo Google, que faz parte do Google Quantum AI[3]. O *Cirq* é baseado no modelo de circuitos quânticos e permite a criação de programas quânticos usando linguagens de programação como *Python*. Ele também oferece ferramentas para simular, depurar, testar e otimizar os programas quânticos. O *Cirq* pode ser usado com o Google Quantum Cloud, que é uma plataforma que fornece acesso a diferentes tipos de hardware quântico.

- Existem outras interfaces de programação paralela para programar processadores quânticos, como o *Quil* (desenvolvido pela Rigetti[4]), o *Ocean* (desenvolvido pela D-Wave[5]), o *Strawberry Fields* (desenvolvido pela Xanadu[6]) e o *ProjectQ* (desenvolvido pela ETH Zurich[7]). Cada uma dessas interfaces tem suas próprias características, vantagens e desvantagens, dependendo do tipo de problema que se quer resolver e do tipo de hardware quântico que se quer usar.

Portanto, a programação paralela para computadores quânticos apresentará desafios únicos à medida que essa tecnologia continua a avançar. Um desafio central será a complexidade inerente das operações quânticas, que envolvem estados superpostos e entrelaçados. Assim, a efetiva divisão e coordenação de tarefas entre qubits para realizar cálculos paralelos precisará levar em consideração as propriedades delicadas dos *qubits* e os requisitos específicos das operações quânticas. Além disso, a natureza não determinística dos computadores quânticos pode tornar a sincronização de processos paralelos mais desafiadora, exigindo novas abordagens para garantir resultados consistentes. A escalabilidade também será um fator limitante, já que a adição de mais *qubits* a um sistema pode aumentar as complexidades de comunicação e controle. Por fim, a heterogeneidade dos dispositivos quânticos disponíveis, com diferentes tempos de coerência e taxas de erro, exigirá estratégias de programação paralela adaptativas e otimizadas para garantir o melhor desempenho em um ambiente quântico.

3.3.3. Computação Neuromórfica de Alto Desempenho

A computação neuromórfica é uma forma de inspirar-se na estrutura e no funcionamento do cérebro humano para projetar e construir sistemas de computação paralela de alto desempenho. Ela tem o objetivo de criar dispositivos que possam aprender, adaptar-se e interagir com o ambiente de forma autônoma e eficiente. A computação neuromórfica também busca superar as limitações dos sistemas convencionais em termos de consumo de energia, latência, precisão, robustez, entre outros. Alguns dos exemplos de aplicações da computação neuromórfica incluem reconhecimento de padrões, visão computacional, processamento de linguagem natural e controle robótico. As plataformas de computação

[3]https://quantumai.google/
[4]https://github.com/quil-lang/quil
[5]https://www.dwavesys.com/solutions-and-products/ocean/
[6]https://strawberryfields.ai/
[7]https://github.com/ProjectQ-Framework/ProjectQ

neuromórfica têm evoluído nos últimos anos. Exemplos destas arquiteturas incluem *Intel Loihi* [Davies et al. 2018], *IBM TrueNorth* [Akopyan et al. 2015] e *SpiNNaker* [Mayr et al. 2019], conforme discutidos a seguir.

O processador *Intel Loihi 2* é uma evolução do *Loihi*, com a promessa de ser até 10 vezes mais rápido e mais econômico em energia. Além disso, o *Loihi 2* foi fabricado com a tecnologia de 4 nanômetros da Intel, sendo um dos primeiros chips a usar esse nó de produção. Este processador possui 128 núcleos de processamento neuromórficos com 8x mais neurônios e sinapses que a versão anterior. Cada núcleo tem 192KB de memória flexível e o neurônio pode ser completamente programável, assim como um FPGA. O processador *IBM TrueNorth* é composto por 4096 núcleos, cada um contendo 256 neurônios artificiais e 256 milhões de sinapses. Esses elementos são interconectados por uma infraestrutura de roteamento baseada em eventos. O *TrueNorth* foi fabricado com a tecnologia de 28 nanômetros da IBM, e consome apenas 65 miliwatts de energia.

A Figura 3.9 destaca a arquitetura *TrueNorth* [Akopyan et al. 2015], inspirada pela estrutura e função do (*a*) cérebro humano e (*b*) coluna cortical, uma pequena região de neurônios densamente interconectados e funcionalmente relacionados que formam a unidade canônica do cérebro. De forma análoga, o (*c*) núcleo neurossináptico é o bloco básico de construção da arquitetura *TrueNorth*, contendo 256 axônios de entrada, 256 neurônios e uma barra transversal sináptica de 64k. Composto por computação fortemente acoplada (neurônios), memória (sinapses) e comunicação (axônios e dendritos), um núcleo ocupa $240x390\mu m$ de área de sílicio (*d*). 4096 núcleos neurossinápticos organizados em uma matriz 2-D forma um chip *TrueNorth* (*e*), que ocupa $4,3cm^2$ em um processo CMOS de 28nm e consome apenas 65mW ao executar uma aplicação típica de visão computacional (*f*). O resultado é uma aproximação eficiente da estrutura cortical dentro das restrições de um subtrato de silício (*g*).

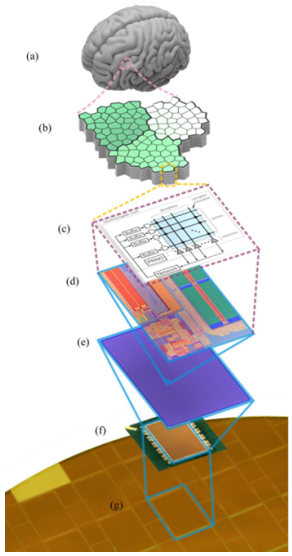

Figura 3.9. Arquitetura TrueNorth, inspirada pela estrutura e funções do cérebro humano [Akopyan et al. 2015]

O processador *SpiNNaker* é uma plataforma para computação massivamente paralela. Ele é composto por meio milhão de elementos computacionais simples, que imitam as

sinapses, controlados pelo seu próprio software. Ele pode realizar até 200 milhões de ações por segundo e tem mais 100 milhões de peças móveis. Durante o seu projeto, três dos axiomas do projeto de máquinas paralelas (coerência de memória, sincronicidade e determinismo) foram descartados sem comprometer a capacidade de realizar cálculos significativos.

Para programar em paralelo para esses processadores, é preciso levar em conta algumas características e técnicas específicas, como por exemplo, o uso de linguagens de programação funcionais que permitem expressar algoritmos de forma mais abstrata e declarativa, facilitando a paralelização neste tipo de arquitetura; estruturas de dados imutáveis para garantir que os dados não serão modificados por outros processos paralelos, evitando problemas de sincronização e consistência; e modelos de programação específicos para processadores neuromórficos.

Estes modelos de programação específicos oferecem abstrações e construções adequadas para processadores neuromórficos. Por exemplo, a linguagem *PyNN* (Python for Neural Networks [Davison et al. 2009]) permite definir modelos de neurônios e sinapses, criar redes neurais e executá-las em diferentes plataformas neuromórficas, como o *SpiNNaker*. A linguagem *Nengo*[8] também permite criar e simular redes neurais em arquiteturas neuromórficas, usando um paradigma baseado em fluxo de dados.

Outra maneira de programar paralelo para arquiteturas neuromórficas é usar bibliotecas ou *frameworks* que facilitam a integração entre as aplicações e as plataformas neuromórficas. Por exemplo, o framework *NeuCube* permite desenvolver aplicações de aprendizado de máquina baseadas em redes neurais esparsas, que podem ser executadas em arquiteturas neuromórficas como o *TrueNorth* [Kasabov 2014]. O framework *NeuGen* permite gerar modelos detalhados de circuitos neuronais, que podem ser simulados em arquiteturas neuromórficas como o *Neurogid* [Eberhard et al. 2006].

No entanto, a programação paralela para computadores neuromórficos apresenta desafios distintos à medida que essa tecnologia evolui. Um desafio central reside na tradução eficiente de algoritmos tradicionais para modelos neuroinspirados, aproveitando a capacidade desses sistemas de processar informações de maneira semelhante ao cérebro humano. A programação eficaz também exigirá a exploração das arquiteturas altamente paralelas e distribuídas desses sistemas, garantindo que a computação ocorra de maneira otimizada e que as interconexões complexas entre neurônios artificiais sejam devidamente coordenadas. Além disso, lidar com a variabilidade inerente dos componentes neuromórficos e as taxas de falha associadas requer estratégias de tolerância a falhas e adaptação dinâmica. A coexistência de múltiplos tipos de neurônios e sinapses em um único chip também exige uma alocação eficiente de recursos e uma programação que leve em conta as características específicas dos diferentes elementos. Em última análise, a programação paralela bem-sucedida para computadores neuromórficos dependerá da colaboração entre a comunidade de neurociência computacional e a comunidade de ciência da computação para criar abordagens que aproveitem totalmente o potencial desses sistemas inovadores.

[8]https://www.nengo.ai/

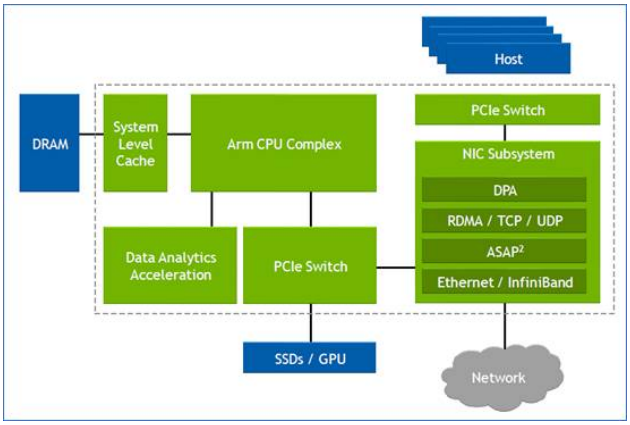

Figura 3.10. Diagrama de blocos da DPU Bluefield-3 da NVidia [Burstein 2021]

3.3.4. Computação Paralela em Adaptadores de Rede Inteligentes

Os adaptadores de rede inteligentes, também conhecidos como *SmartNICs*, são dispositivos que possuem unidades de processamento de dados (DPUs – *data processing units*) integradas capazes de executar funções de rede e segurança no próprio hardware, sem depender da CPU do sistema. Estes dispositivos têm sido bastante úteis para cenários de computação de alto desempenho, como ambiente de nuvem, aprendizado de máquina e *big data*.

Diferentes são as maneiras de empregar *SmartNICs* para computação de alto desempenho. Uma delas consiste no uso das DPUs para executar funções de rede e segurança no hardware. Outra forma consiste em usar as tecnologias de transferência direta de dados entre os dispositivos de armazenamento e as GPUs, como RoCE e GPUDirect Storage[9]. Essas tecnologias permitem que os dados sejam processados em paralelo pelas GPUs, sem passar pela CPU ou pela memória do sistema, reduzindo a latência, o consumo de energia e a complexidade do sistema. Além disso, os *SmartNICs* podem fornecer um serviço de sincronização extremamente preciso para as aplicações de centro de dados e infraestrutura subjacente. Isso pode facilitar a coordenação e a comunicação entre os processos paralelos que dependem de sincronização de tempo, como os algoritmos distribuídos e os sistemas de consenso.

A Figura 3.10 exemplifica os componentes de *hardware* do adaptador de rede inteligente *Bluefild-3* (em verde), da NVIDIA. Este adaptador está conectado à internet (*Network*) e a máquina *host*. Adicionalmente, ele possui conexões diretas com a memória DRAM e unidades de armazenamento para acelerar o acesso aos dados sem a necessidade de interferência da CPU do *host*. Esta arquitetura possui 16 núcleos ARM-A78 (*ARM CPU Complex*) e aceleradores de dados com 16 núcleos e 256 *threads* (*Data Analytics Acceleration*). Deste modo, para programar em paralelo para *SmartNICs* tais como *Bluefield-3*, é preciso conhecer os recursos e as limitações das unidades de processamento de dados que eles possuem, bem como as bibliotecas e as ferramentas que permitem acessar e controlar essas DPUs. Assim, diferentes bibliotecas surgem como alternativas:

[9]https://docs.nvidia.com/gpudirect-storage/overview-guide/index.html

- ***NVIDIA DOCA***[10] é um framework de desenvolvimento de software para *SmartNICs* baseados em DPUs NVIDIA BlueField. Ele oferece uma interface de programação para criar e gerenciar aplicações de rede e segurança que executam nas DPUs. Ele também fornece um ambiente de execução (*runtime*) que abstrai os detalhes de baixo nível do hardware e do sistema operacional.
- ***Mellanox SmartNIC SDK***[11] é um kit de desenvolvimento de software para *SmartNICs* baseados em DPUs *Mellanox ConnectX*. Ele permite o desenvolvimento de aplicações personalizadas que executam nas DPUs, usando linguagens como C, C++ e Python. Ele também inclui exemplos de código, documentação e ferramentas de depuração.
- ***Intel Data Plane Development Kit (DPDK)*** é um conjunto de bibliotecas e *drivers* para acelerar o processamento de pacotes em plataformas baseadas em processadores Intel. Ele pode ser usado para programar *SmartNICs* baseados em DPUs Intel I/O Processing Units (IOPUs), usando linguagens como C e *Rust*. Ele também oferece suporte a várias arquiteturas de rede, como memória compartilhada, troca de mensagens e RDMA [Zhu 2020].

Um desafio central da programação paralela em *SmartNICs* é a exploração eficiente das capacidades de processamento altamente paralelo desses controladores, que são projetados para acelerar funções de rede e processamento de dados em níveis próximos ao hardware. A programação precisa encontrar o equilíbrio entre a distribuição de tarefas em vários núcleos de processamento e a otimização do uso dos recursos específicos do *SmartNIC* para obter um desempenho máximo. Além disso, a coordenação entre os núcleos de processamento e as unidades de rede especializadas exige abordagens eficazes para evitar gargalos de comunicação e latência excessiva. Lidar com a heterogeneidade de cargas de trabalho e requisitos variados de aplicativos também é um desafio, visto que diferentes cenários de uso demandarão estratégias de programação personalizadas. A garantia de segurança e isolamento, especialmente quando várias funções de rede são executadas no mesmo dispositivo, também será crucial. Em última análise, a programação paralela bem-sucedida para *SmartNICs* dependerá da compreensão profunda da arquitetura desses dispositivos e da adaptação das técnicas de programação existentes para tirar o máximo proveito de suas capacidades específicas

3.3.5. Computação de Alto Desempenho em Arquiteturas Massivamente Paralelas

Arquiteturas massivamente paralelas são projetos de sistemas de computação que se destacam pela capacidade de executar um grande número de tarefas ou instruções simultaneamente, aproveitando um grande conjunto de unidades de processamento interconectadas. Essa abordagem visa a obtenção de um alto nível de desempenho computacional para lidar com cargas de trabalho intensivas e complexas, como simulações científicas, análise de grandes conjuntos de dados e processamento de inteligência artificial.

Essas arquiteturas se diferenciam das abordagens convencionais, que contam com um ou alguns poucos núcleos de processamento central. Em vez disso, as arquiteturas massivamente paralelas incorporam centenas ou mesmo milhares de núcleos de processamento

[10] https://developer.nvidia.com/networking/doca

[11] https://docs.nvidia.com/networking/display/NEOSDKv25

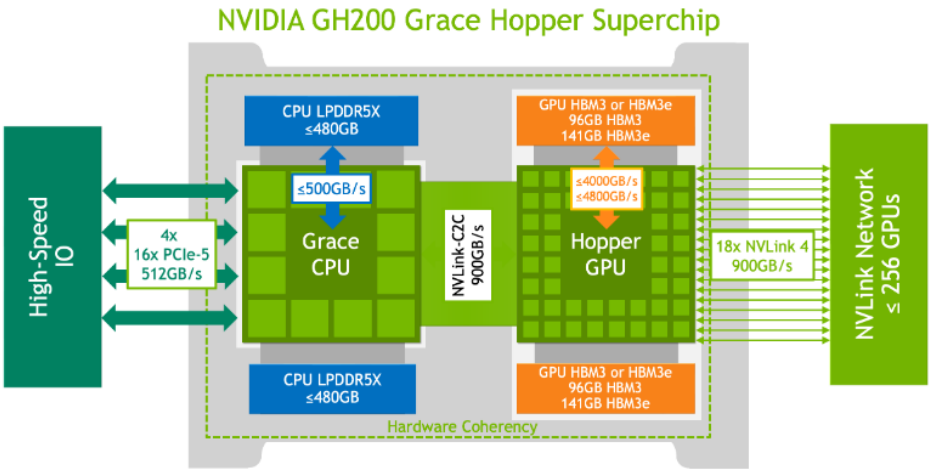

Figura 3.11. Superchip Grace Hopper da NVIDIA

independentes, trabalhando em conjunto para acelerar a execução de tarefas. Um exemplo notável é o uso de unidades de processamento gráfico (GPUs - *graphical processing units*), projetadas originalmente para renderização gráfica. A arquitetura altamente paralela dessas unidades se mostrou extremamente eficaz em tarefas de computação intensiva, como aprendizado de máquina e simulações científicas. Logo, é natural que diferentes empresas têm voltado seus projetos para o desenvolvimento de arquiteturas massivamente paralelas, como o caso da NVIDIA, Intel e AMD.

Recentemente, a NVIDIA projetou a arquitetura *GH200 Grace Hopper*. Ela consiste de uma CPU acelerada projetada do zero para computação de alto desempenho e inteligência artificial. Conforme ilustrado na Figura 3.11[12], ela combina as arquiteturas *Grace* e *Hopper* usando a tecnologia *NVIDIA NVLink-C2C* para oferecer um modelo de memória coerente de CPU e GPU com alta largura de banda (até 900GB/s). Considerando estações de trabalho que demandam alto desempenho gráfico, a Intel projetou a família de GPUs Intel Arc, com suporte a *Ray Tracing* acelerado por *hardware*, codificação de vídeo com o *codec* AV1 e outras características. De modo similar, a GPU AMD ROCm é uma plataforma de software aberto para programação de GPUs, suportando ambientes em vários fornecedores e arquiteturas de aceleradores. Ela oferece suporte às principais estruturas de aprendizado de máquina para ajudar os usuários a acelerar as cargas de trabalho de IA.

No entanto, aproveitar ao máximo essa capacidade computacional disponível requer uma mudança na forma como os programas são projetados e implementados. A programação paralela nestas arquiteturas massivamente paralelas visa aproveitar as vantagens de cada dispositivo para resolver problemas complexos de forma eficiente e escalável. Assim, diversas interfaces de programação permitem o desenvolvimento de aplicações paralelas para tais arquiteturas, cada uma com suas próprias características, vantagens e desvantagens. Algumas das mais populares são:

- *CUDA*: É uma linguagem baseada em C/C++ que permite a programação direta de GPUs da NVIDIA. É uma linguagem de baixo nível que oferece um controle fino sobre

[12]https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper

os recursos da GPU, mas também exige um conhecimento profundo da arquitetura e do modelo de execução.

- *OpenCL*: É uma linguagem baseada em C/C++ que permite a programação de diversos tipos de dispositivos, como CPUs, GPUs, FPGAs, entre outros. É uma linguagem de baixo nível que oferece flexibilidade e portabilidade, mas também exige um cuidado maior com a compatibilidade e a otimização do código para cada dispositivo.
- *OpenMP*: É uma interface de programação paralela para C/C++ que permite a programação paralela por meio da inserção de diretivas e funções que controlam a criação e a sincronização de threads em um código sequencial. Diferentemente de *OpenCL* e *CUDA*, ela visa exploração de paralelismo objetivando simplicidade e produtividade.
- *OpenACC*: É uma extensão da linguagem C/C++ que permite a programação paralela por meio de diretivas que indicam quais regiões do código devem ser executadas em arquiteturas multicore e dispositivos aceleradores, como GPUs e FPGAs. Assim com o *OpenMP*, é uma linguagem de alto nível que oferece abstração e portabilidade, mas também depende da qualidade do compilador para gerar código eficiente para cada dispositivo.
- *Intel OneAPI*: é um modelo de programação unificado, baseado em padrões abertos, que permite o desenvolvimento de aplicações que podem ser executadas em diferentes tipos de arquiteturas aceleradoras, como CPUs, GPUs e FPGAs. Seu objetivo é oferecer uma experiência de desenvolvimento que aumenta a produtividade, desempenho e inovação dos desenvolvedores.

3.3.6. Computação de Alto Desempenho em Processadores de Inteligência Artificial

Nos últimos anos, avanços notáveis na convergência entre a IA e HPC tem resultado em uma simbiose promissora que impulsiona a fronteira da capacidade computacional. Assim, o uso de processadores otimizados para IA em ambientes de HPC está se tornando uma tendência proeminente, trazendo consigo a promessa de acelerar não apenas as cargas de trabalho tradicionais de HPC, mas também possibilitando abordagens inovadoras para problemas complexos. Esses processadores, muitas vezes equipados com unidades especializadas de hardware para tarefas intensivas de IA, como treinamento e inferência de redes neurais profundas, demonstraram a capacidade de lidar com a crescente demanda por processamento de dados complexos em campos que variam desde a pesquisa científica até a análise de grandes conjuntos de dados. Deste modo, esta fusão entre IA e HPC promete catalisar a descoberta, resolução de problemas complexos e tomada de decisões mais informadas, moldando o futuro da computação de alto desempenho.

Diferentes arquiteturas focadas em IA têm sido propostas ao longo dos anos, conforme discutido a seguir. O processador SambaNova é baseado em uma arquitetura chamada *Reconfigurable Dataflow Unit* (RDU), que consiste em um arranjo de unidades aritméticas que podem se comunicar entre si de forma assíncrona e adaptativa, sem a necessidade de um barramento ou uma memória compartilhada . Essa arquitetura permite que o processador se adapte dinamicamente às características e aos requisitos das diferentes aplicações de inteligência artificial, otimizando o uso dos recursos e o consumo de energia. O processador SambaNova pode ser usado tanto para treinamento quanto para inferência de

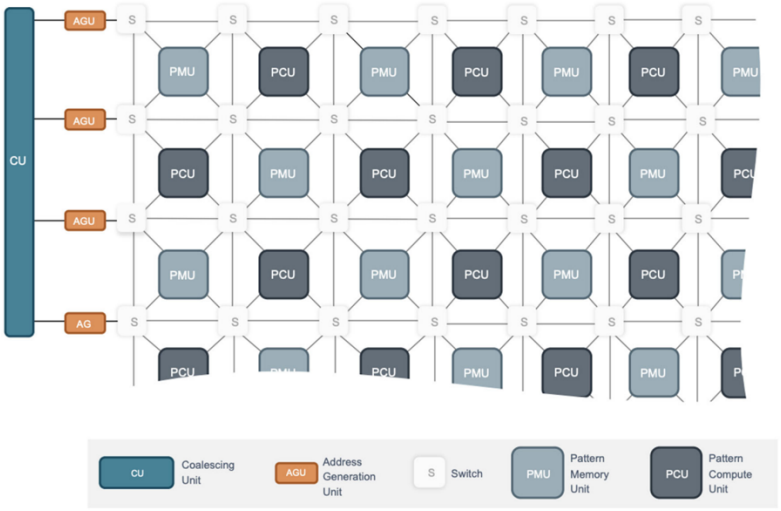

Figura 3.12. Reconfigurable Dataflow Unit (RDU) do sistema SambaNova [Emani et al. 2021]

modelos de inteligência artificial, sendo capaz de executar redes neurais profundas com milhões ou bilhões de parâmetros.

A Figura 3.12 destaca uma pequena parte da arquitetura RDU [Emani et al. 2021]. Ela consiste de uma matriz de unidades de processamento e memória reconfiguráveis conectadas por uma malha de comutação 3-D. Quando uma aplicação inicia a execução, o *software SambaFlow* configura os elementos RDU para executar um fluxo de dados otimizado para a aplicação. A PCU (unidade de computação padrão) foi projetada para executar uma única operação paralela de uma aplicação. Seu caminho de dados é organizado como um pipeline SIMD reconfigurado de vários estágios. A PMU (unidade de memória padrão) consiste de *scrathpads* especializadas que fornecem capacidade de memória e executam uma série de funções específicas para minimizar a movimentação de dados, reduzir a latência e fornecer alta largura de banda. O componente de *Switch* (S) é uma estrutura de alta velocidade que conecta PCUs e PMUs, sendo composta por três redes de comutação: escalar, vetorial e de coontrole. Por fim, as unidades geradoras de endereço (AGU) e unidades coalescentes (CU) fornecem a interconexão entre RDUs e o resto do sistema, incluindo a memória DRAM, outras RDUs, e o processador *host* para o processamento eficiente de problemas maiores.

Cerebras Systems é um processador baseado em uma arquitetura chamada *Wafer Scale Engine* (WSE), que consiste em um único chip que ocupa toda a superfície de um *wafer* de silício. Um *wafer* é um disco de silício usado para fabricar vários chips menores, mas o processador Cerebras Systems usa o wafer inteiro como um chip gigante, com trilhões de transistores e milhares de núcleos otimizados para IA. Outras soluções incluem o processador *Graphcore* e *Groq*. *Graphcore* é baseado em uma arquitetura chamada *Intelligence Processing Unit* (IPU), que consiste em um chip que contém milhares de núcleos de processamento e uma grande quantidade de memória interna [Knowles 2021]. Já o processador *Groq* é baseado em uma arquitetura chamada Tensor Streaming Processor (TSP), que consiste em um chip que contém milhares de unidades de processamento e

uma grande quantidade de memória interna [Gwennap 2020].

As interfaces de programação paralela que podem ser utilizadas para programar tais processadores são as seguintes:

- *SambaNova* oferece uma plataforma de software chamada *SambaFlow*, que é um conjunto de ferramentas e bibliotecas para desenvolver, treinar e implantar modelos de aprendizado de máquina em seu sistema *DataScale*. O *SambaFlow* suporta linguagens de programação como Python, C++ e Java, e frameworks de aprendizado de máquina como *TensorFlow*, *PyTorch* e *MXNet*. Ele também permite a integração com outras ferramentas de software, como *Kubeflow*, *Spark* e *Ray*.

- *Cerebras* oferece uma plataforma de software chamada *Cerebras Software Platform*, um ambiente integrado para desenvolver, treinar e executar modelos de aprendizado de máquina em seu sistema *CS-2*. Esta plataforma suporta linguagens de programação como Python e C++, e *frameworks* de aprendizado de máquina como *TensorFlow*, *PyTorch* e *JAX*. O Cerebras Software Platform também permite a integração com outras ferramentas de software, como *Horovod*, *MPI* e *NCCL*.

- *GraphCore* oferece uma plataforma de software chamada *Poplar*, um framework gráfico para desenvolver, treinar e implantar modelos de aprendizado de máquina em seu sistema IPU. O Poplar suporta linguagens de programação como Python e C++, e frameworks de aprendizado de máquina como *TensorFlow*, *PyTorch* e *ONNX*. O *Poplar* também permite a integração com outras ferramentas de software, como *Kubeflow*, *Spark* e *Dask*.

- *Groq* oferece uma plataforma de software chamada *Groq SDK*, que suporta linguagens de programação como Python e C++, e frameworks de aprendizado de máquina como TensorFlow, PyTorch e ONNX. O *Groq SDK* também permite a integração com outras ferramentas de software, como *TVM*, *MLIR* e *Glow*.

3.4. Conclusão e Reflexões Futuras

O processamento paralelo é uma peça fundamental no desenvolvimento técnico-científico pois é necessário para resolver problemas em inúmeras áreas da ciência e da tecnologia. Torna-se portanto fundamental criar intuições a partir do pensamento computacional paralelo e da programação paralela. Este minicurso procurou trazer as peças básicas necessárias para incutir no leitor os reflexos necessários para projetar soluções paralelas eficientes, além de trazer uma moderno apanhado de tendências tecnológicas que moldam o futuro da área. Espera-se que com esta combinação de conceitos básicos e tendências o leitor possa ser motivado a trabalhar nesta enriquecedora e multidisciplinar área de pesquisa, com problemas concretos e de alto impacto na sociedade.

Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Referências

[Akopyan et al. 2015] Akopyan, F., Sawada, J., Cassidy, A., Alvarez-Icaza, R., Arthur, J., Merolla, P., Imam, N., Nakamura, Y., Datta, P., Nam, G.-J., Taba, B., Beakes, M., Brezzo, B., Kuang, J. B., Manohar, R., Risk, W. P., Jackson, B., and Modha, D. S. (2015). Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557.

[Barnes and Hut 1986] Barnes, J. and Hut, P. (1986). A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446.

[Burstein 2021] Burstein, I. (2021). Nvidia data center processing unit (dpu) architecture. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–20.

[Chow et al. 2021] Chow, J., Dial, O., and Gambetta, J. (2021). Ibm quantum breaks the 100-qubit processor barrier. *IBM Research Blog*, 2.

[Davies et al. 2018] Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S. H., Dimou, G., Joshi, P., Imam, N., Jain, S., et al. (2018). Loihi: A neuromorphic manycore processor with on-chip learning. *Ieee Micro*, 38(1):82–99.

[Davison et al. 2009] Davison, A. P., Brüderle, D., Eppler, J. M., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., and Yger, P. (2009). Pynn: a common interface for neuronal network simulators. *Frontiers in neuroinformatics*, 2:388.

[Eberhard et al. 2006] Eberhard, J. P., Wanner, A., and Wittum, G. (2006). Neugen: a tool for the generation of realistic morphology of cortical neurons and neural networks in 3d. *Neurocomputing*, 70(1-3):327–342.

[Emani et al. 2021] Emani, M., Vishwanath, V., Adams, C., Papka, M. E., Stevens, R., Florescu, L., Jairath, S., Liu, W., Nama, T., and Sujeeth, A. (2021). Accelerating scientific applications with sambanova reconfigurable dataflow architecture. *Computing in Science & Engineering*, 23(2):114–119.

[Foster 1995] Foster, I. (1995). *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Fu et al. 2016] Fu, X., Riesebos, L., Lao, L., Almudever, C. G., Sebastiano, F., Versluis, R., Charbon, E., and Bertels, K. (2016). A heterogeneous quantum computer architecture. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 323–330.

[Gwennap 2020] Gwennap, L. (2020). Groq rocks neural networks. *Microprocessor Report, Tech. Rep., jan*.

[Kasabov 2014] Kasabov, N. K. (2014). Neucube: A spiking neural network architecture for mapping, learning and understanding of spatio-temporal brain data. *Neural Networks*, 52:62–76.

[Knowles 2021] Knowles, S. (2021). Graphcore. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–25.

[Mayr et al. 2019] Mayr, C., Höppner, S., and Furber, S. B. (2019). Spinnaker 2: A 10 million core processor system for brain simulation and machine learning. *CoRR*, abs/1911.02385.

[Navaux et al. 2023] Navaux, P. O. A., Lorenzon, A. F., and Serpa, M. d. S. (2023). Challenges in high-performance computing. *Journal of the Brazilian Computer Society*, 29(1):51â€“62.

[Paillard et al. 2015] Paillard, G. A. L., Coutinho, E. F., de Lima, E. T., and Moreira, L. O. (2015). An architecture proposal for high performance computing in cloud computing environments. In *4th International Workshop on ADVANCEs in ICT INfrastructures and Services (ADVANCE 2015), Recife*.

[Zhu 2020] Zhu, H. (2020). *Data Plane Development Kit (DPDK): A Software Optimization Guide to the User Space-Based Network Applications*. CRC Press.