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.

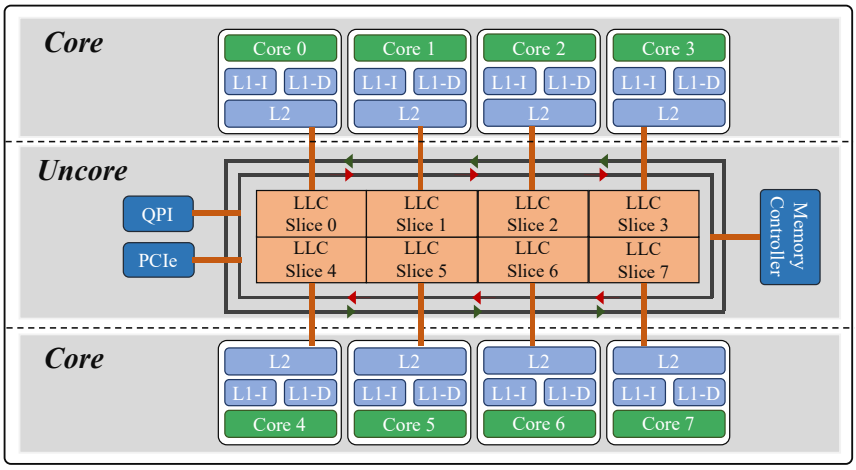


Figura 3.1: Exemplo do *package* de um processador com a parte do *Core* e *Uncore*

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

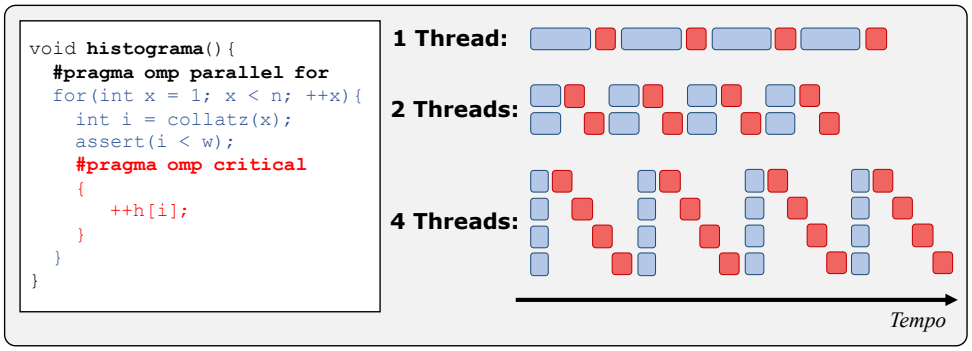


Figura 3.3: Escalabilidade de uma região crítica

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:

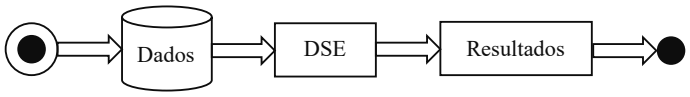
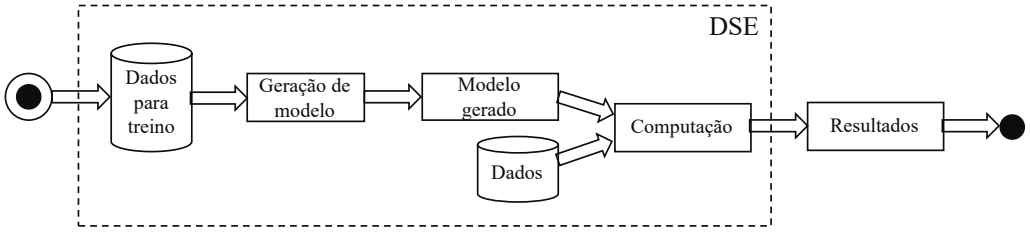
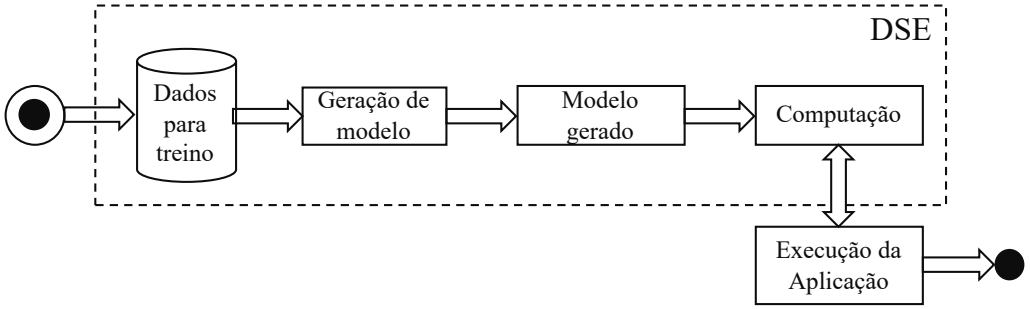


Figura 3.4: Processo de exploração dos parâmetros de configuração



(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

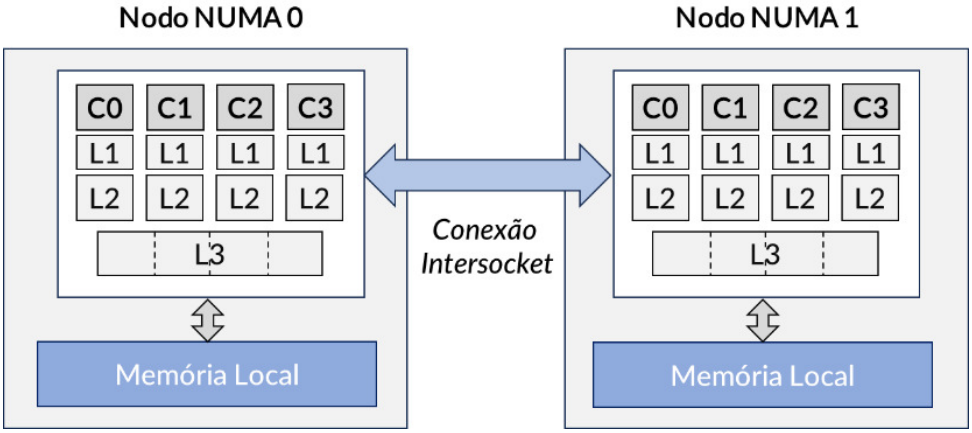


Figura 3.6: Representação de uma máquina com dois nodos NUMA

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]

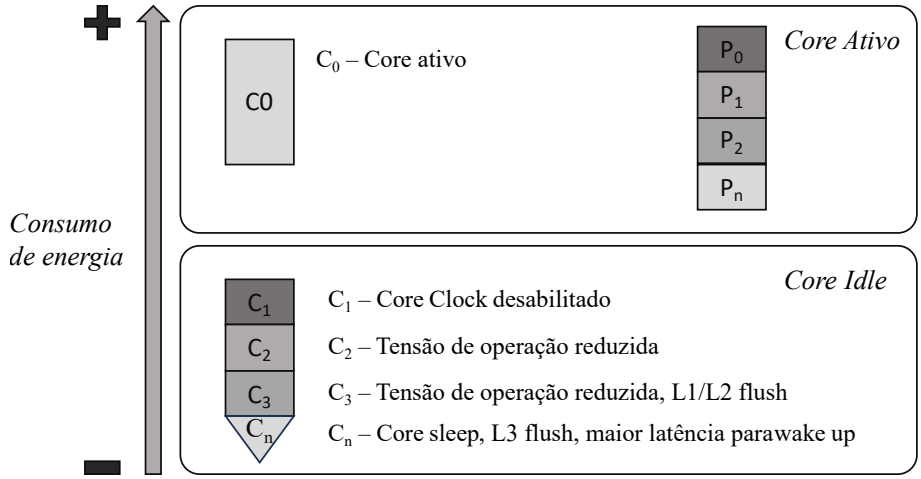


Figura 3.7: Representação dos *P-states* e *C-states*.

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]*.