Capítulo 3

Programação Paralela Híbrida: MPI + OpenMP Offloading

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

Abstract

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

Resumo

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

3.1. Introdução

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

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

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

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

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

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

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

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

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

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

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

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

3.2. Arquitetura de Computadores Paralelos

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3. Aceleradores GP-GPUs

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

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

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

Figura 3.1: Arquitetura NVIDIA Kepler

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

Principais características da arquitetura Kepler

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

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

Figura 3.2: Multiprocessador de Fluxo (SM)

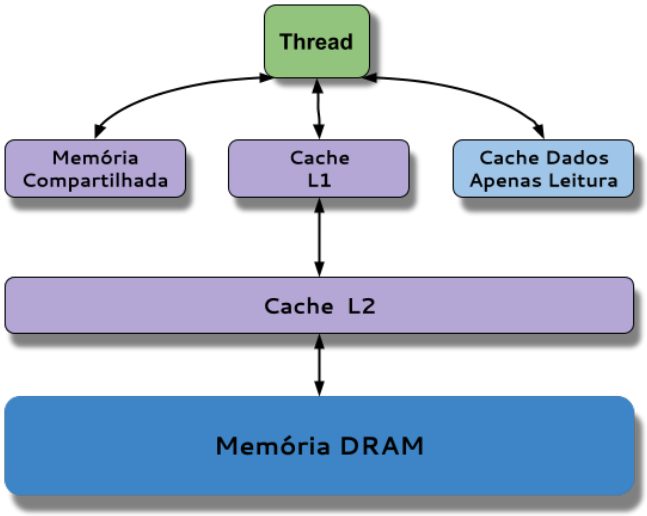
Para melhorar ainda mais o desempenho, a arquitetura Kepler apresenta uma instrução de *shuffle* que permite às *threads* dentro de uma mesma *warp* compartilhar dados. Anteriormente, o compartilhamento de dados entre *threads* demandava o acesso à memória compartilhada, com operações de *load* e *store*,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

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

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

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

Figura 3.3: Hierarquia de Memória



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

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

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

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

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

3.4. Modelos de programação

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

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

3.4.1. Paralelismo com MPI

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

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

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

Cálculo de Pi somente com MPI

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

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

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

Exemplo 3.1: Cálculo do PI em diversos processos com MPI

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

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

3.4.2. Paralelismo com OpenMP

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

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

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

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

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

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

Cálculo Pi somente com OpenMP

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

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

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

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

3.5. Programação Híbrida MPI + OpenMP

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

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

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

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

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

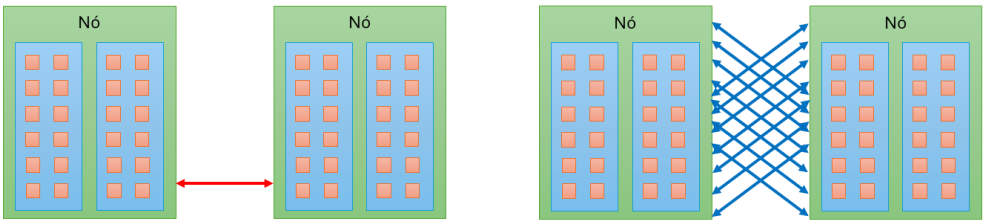
- **Redução do uso de memória**, já que haverá uma diminuição no número de cópias de estruturas de dados replicadas, além de uma menor quantidade de dados em regiões "halo", que são necessárias em algumas operações de comunicação entre processos.
- **Exploração de níveis adicionais de paralelismo**,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

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

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

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

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



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

Figura 3.4: Comunicação de Programação Hibrida

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

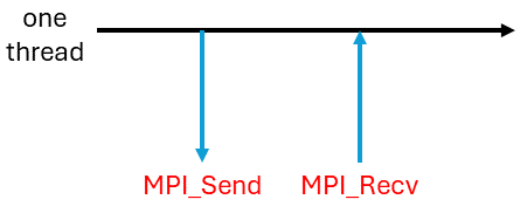
É importante manter essa imagem em mente ao explorar este tópico. Lembre-se de que apenas que na Figura 3.4b à direita as mensagens MPI estão sendo enviadas em paralelo entre múltiplas *threads* em dois (ou mais) ranques MPI.

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

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

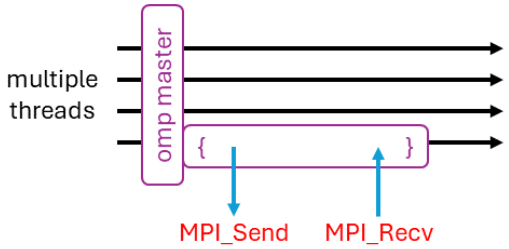
- **MPI_THREAD_SINGLE** (*single*): apenas uma *thread* existe em cada processo MPI — sem *multithreading* - veja a Figura 3.5. Em termos práticos, o código não pode conter diretivas OpenMP com regiões paralelas (*#pragma omp parallel*).

Figura 3.5: Modelo *Single*



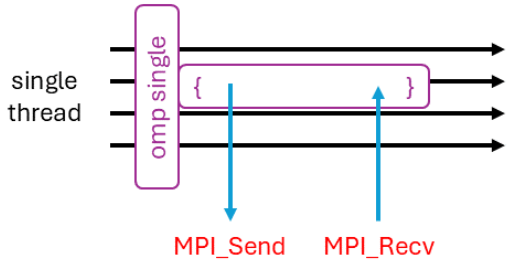
- **MPI_THREAD_FUNNELLED** (*funneled*): apenas a *thread* principal pode fazer chamadas MPI; em regiões *multithread*, todas as chamadas MPI devem ser canalizadas através da *thread* principal - veja a Figura 3.6. Ou seja, o programa pode conter diretivas OpenMP com regiões paralelas, mas apenas a *thread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

Figura 3.6: Modelo *Funneled*



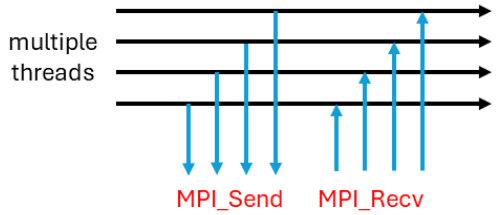
- **MPI_THREAD_SERIALIZED** (*seralized*): múltiplas *threads* podem chamar o MPI, mas de forma sincronizada, de modo que apenas uma chamada MPI esteja em progresso por vez - veja a Figura 3.7. Para evitar que chamadas MPI de uma *thread* sejam sobrepostas com chamadas de outras *threads*, as construções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

Figura 3.7: Modelo *Serialized*



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

Figura 3.8: Modelo *Multiple Threads*



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

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

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

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

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

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

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

  MPI_Finalize();
}
```

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

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

Cálculo de Pi com MPI + OpenMP

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

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

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

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

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

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

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

Considerações sobre o desempenho MPI + OpenMP

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

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

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

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

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

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

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

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

3.6. Programação em GPU com OpenMP *Offloading*

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

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

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

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

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

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

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

3.6.1. Modelo *host/device* do OpenMP

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Exemplo 3.5: Exemplo de código OpenMP com offloading

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

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

Vantagens do modelo *host/device* no OpenMP

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

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

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

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

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

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

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

3.6.3.1. Diretiva target – Cláusula map

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

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

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

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

Nela,

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

O seguintes tipos de mapeamento são admitidos:

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

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

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

Exemplo 3.6: Exemplo de uso da cláusula 'map'

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

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

Uso avançado da cláusula 'map'

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

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

Exemplo 3.7: Uso avançadao da cláusula 'map'

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

Mapeamento de Subregiões de Arrays

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

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

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

Boas Práticas ao Usar 'map'

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

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

3.6.3.2. Diretiva target data

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

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

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

Exemplo 3.8: Diretiva 'target data'

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

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

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

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

3.6.3.3. Diretiva teams

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

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

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

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

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

3.6.3.4. Diretiva distribute

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

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

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

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

Uso Combinado de 'teams' e 'distribute'

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

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

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

Diferença entre 'distribute' e 'for'

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

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

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

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

Cálculo de Pi somente com OpenMP *offloading*

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

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

Exemplo 3.10: Pi só com OpenMP *offloading*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Cálculo de Pi com MPI + OpenMP com *offloading*

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

Exemplo 3.11: Pi com MPI + OpenMP com offloading

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

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

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

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

3.8. Conclusão

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

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

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

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

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

Referências

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

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

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

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

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

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

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

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

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

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

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

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

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

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