Programação Paralela Híbrida: MPI + OpenMP Offloading

Calebe, Evaldo e **Gabriel**

Escola Supercomputador Santos Dumont - 2025

Introdução

- Neste minicurso é esperado que os alunos tenham um conhecimento básico de MPI e OpenMP, como apresentado na referência:
  - https://www.amazon.com.br/Programação-Paralela-Distribuída-computação-desempenho-ebook/dp/B0B27Z9R3N/
- Repositório utilizado neste minicurso:
  - https://github.com/Programacao-Paralela-e-Distribuida/HIBRIDA

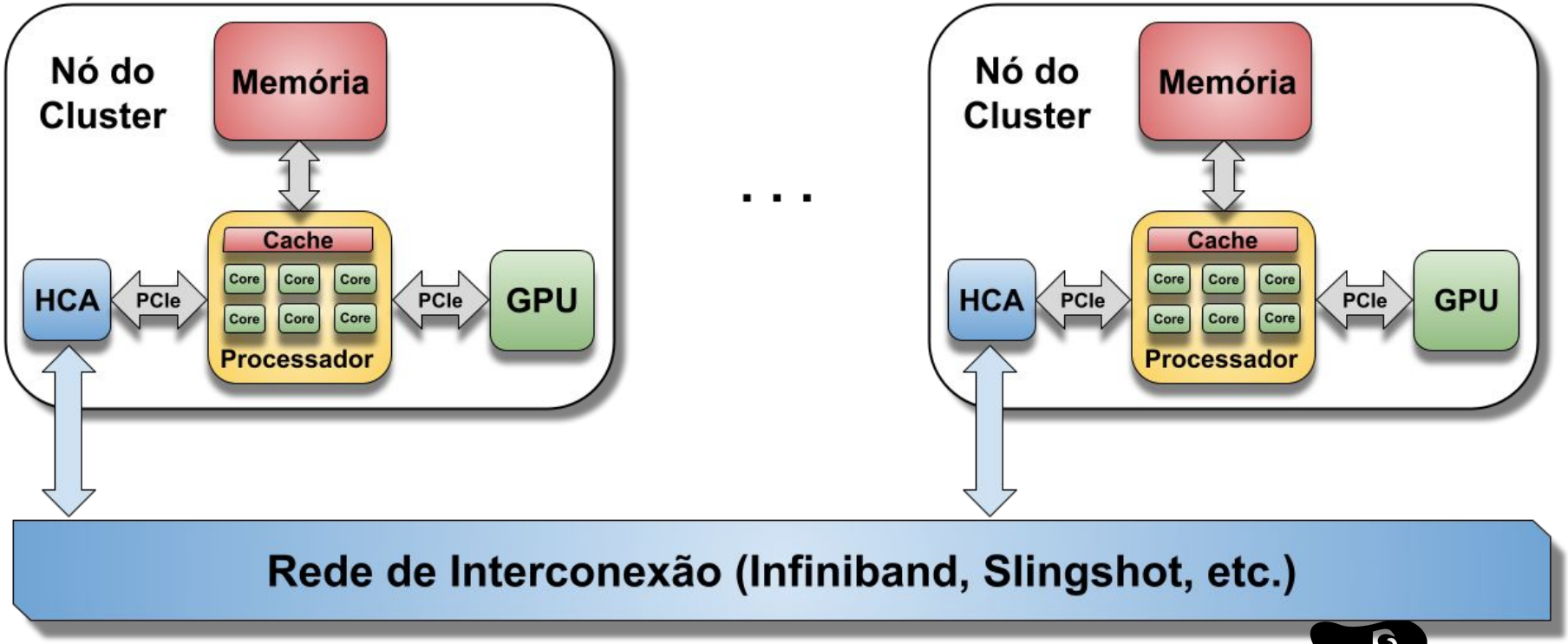
Introdução

- O uso de técnicas eficientes de paralelismo é fundamental para otimizar o uso dos recursos dos modernos computadores.
- A programação híbrida **MPI + OpenMP *offloading*** é uma abordagem eficiente para explorar paralelismo em *clusters* com múltiplos nós com aceleradores (como GPUs), que é aplicada especialmente em computação científica e simulação.
- O MPI é utilizado para comunicação entre nós do *cluster*, dividindo o trabalho e permitindo a comunicação via redes de alta velocidade (ex: Infiniband).

Introdução

- O OpenMP *offloading* facilita a paralelização *multithreaded* e a descarga de tarefas intensivas para as GPUs, maximizando o desempenho computacional.
- Vantagens da abordagem híbrida, com a combinação de MPI e OpenMP para:
  - Melhor aproveitamento dos núcleos de CPUs quanto os aceleradores disponíveis (GPUs, FPGAs).
  - Melhor uso da memória.
  - Exploração adicional do paralelismo.
  - Balanceamento eficiente da carga de trabalho.

Cluster

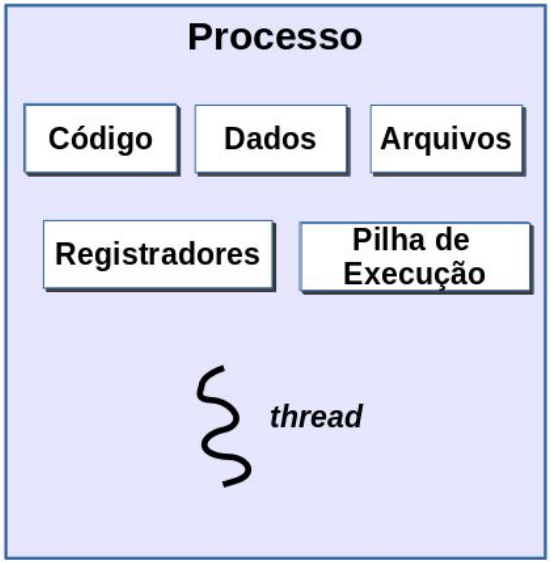

Nó do Cluster
Memória
Cache
Core
Core
Core
Core
Core
Core
Processador
HCA
PCIe
PCIe
GPU
. . .
Nó do Cluster
Memória
Cache
Core
Core
Core
Core
Core
Core
Processador
HCA
PCIe
PCIe
GPU
Rede de Interconexão (Infiniband, Slingshot, etc.)

Conteúdos Abordados

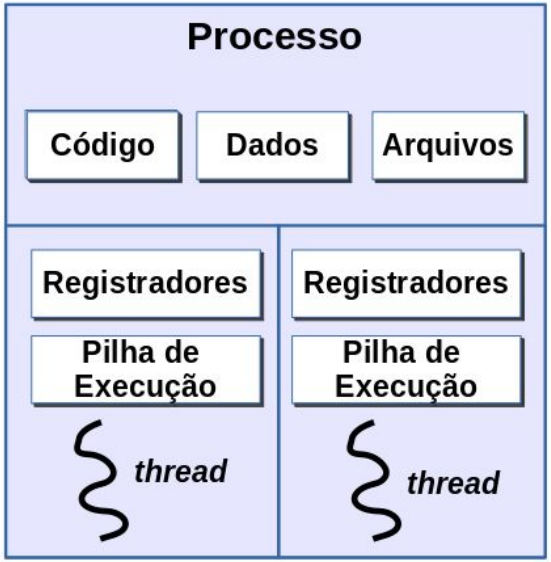
- Modelos de programação e arquitetura dos aceleradores (GPUs).
- Apresentação dos diferentes níveis de suporte a threads no MPI (single, funneled, serialized, multiple) e suas implicações na programação híbrida.
- Uso das diretivas de descarga de trabalho para aceleradores, considerando:
  - Hierarquia de memória e *hardware*.
  - Movimentação eficiente de dados entre o hospedeiro e o dispositivo.
- Exemplos práticos de divisão eficiente do trabalho entre processadores e aceleradores em *clusters*, otimizando a execução de códigos em sistemas heterogêneos.

Conceitos Básicos

Processos vs Threads



a) Processo com uma única *thread*



a) Processo com várias *threads*

Paradigma Memória Distribuída

- Os processos/threads não dispõem de um espaço de memória em comum para operações de comunicação e sincronização, sendo que essas operações são feita por meio de troca de mensagens.
- Os processos dependem de uma rede de interconexão e de uma biblioteca de comunicação, como MPI, para realizar o envio e recebimento de mensagens.
- A programação paralela por troca de mensagens é altamente escalável, o que significa que é possível adicionar mais processadores para aumentar o desempenho da aplicação.

MPI

- Comunicação ponto a ponto e coletiva, com suporte para interação entre processos de forma direta ou em grupo, conforme a necessidade da aplicação.
- Modelos de comunicação variados, com compatibilidade com comunicação síncrona ou assíncrona, atendendo a diferentes requisitos de desempenho e sincronização.
- Topologias de rede virtuais, com suporte para estruturas como anel, árvore e malha, permitindo a organização eficiente de processos em sistemas paralelos.

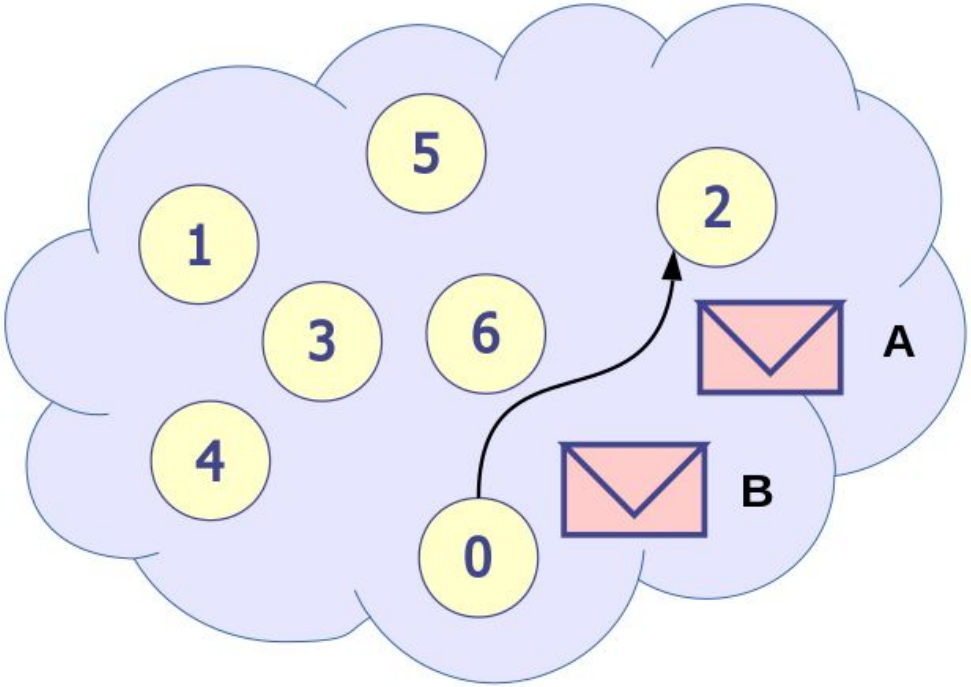
MPI

- Programação por troca de mensagens, pressupõe que um processo não tem acesso à memória do outro.
- São utilizadas funções como MPI_Send() para envio e MPI_Recv() para recepção das mensagens ponto a ponto.
- Possui também um conjunto de rotinas coletivas, como MPI_Bcast(), MPI_Reduce(), entre outras.
- Veja a referência para maiores detalhes.

MPI

- Tipos de dados básicos e estruturados, com flexibilidade para manipular diferentes formatos de dados na troca entre processos.
- Operações de comunicação coletiva, com implementações eficientes para difusão, redução e dispersão de dados, fundamentais em computação de alto desempenho.
- Portabilidade, já que oferece compatibilidade com diversas arquiteturas de computação paralela e sistemas operacionais, garantindo ampla adoção e facilidade de integração.

MPI – Message Passing Interface



MPI

- O MPI pode ser combinado com outros modelos de programação. Assim, podemos combinar MPI com OpenMP *threads*, que apresenta vantagens em alguns casos, como por exemplo:
  - Códigos com escalabilidade MPI limitada, quer seja pelo algoritmo ou pelas rotinas de comunicação coletiva utilizadas.
  - Códigos limitado pelo tamanho de memória, tendo muitos dados replicados em cada processo MPI.
  - Códigos com problemas de desempenho pela ineficiência da implementação da comunicação intra-nó em MPI.
- O MPI também pode ser combinado com OpenMP Offloading, OpenACC ou CUDA para uso de aceleradores.

Paradigma Memória Compartilhada

- Todos os processos/threads compartilham uma memória global comum, como consequência, toda comunicação entre as tarefas é feita através de variáveis na memória.
- Mecanismos de sincronização, como semáforos ou regiões críticas, são utilizados para evitar conflitos e garantir a correta ordenação de acesso à memória compartilhada.
- Como o acesso à memória compartilhada pode ser limitado, problemas de escalabilidade podem surgir com o aumento do número de processadores.
- As bibliotecas de programação mais utilizadas: OpenMP, pthreads.

OpenMP

- O OpenMP (Open Multi-Processing) permite a obtenção de paralelismo por meio da execução simultânea de múltiplas *threads* dentro de regiões paralelas definidas no código.
- O ganho real de desempenho depende da disponibilidade de processadores na arquitetura que possam executar essas regiões efetivamente em paralelo.
- Além disso, as iterações de um laço `for` podem ser distribuídas entre as *threads*. Caso não existam dependências de dados entre as iterações, essas podem ser executadas simultaneamente, maximizando a eficiência do processamento.

OpenMP Threads

- Utiliza diversas *threads*, normalmente executadas por vários núcleos do mesmo processador, para obter ganhos de desempenho, em aplicações paralelas que compartilham a mesma memória.
- O OpenMP faz uso conjunto de diretivas passadas para o compilador, assim como de algumas funções, para explorar o paralelismo no código em linguagem C ou FORTRAN.

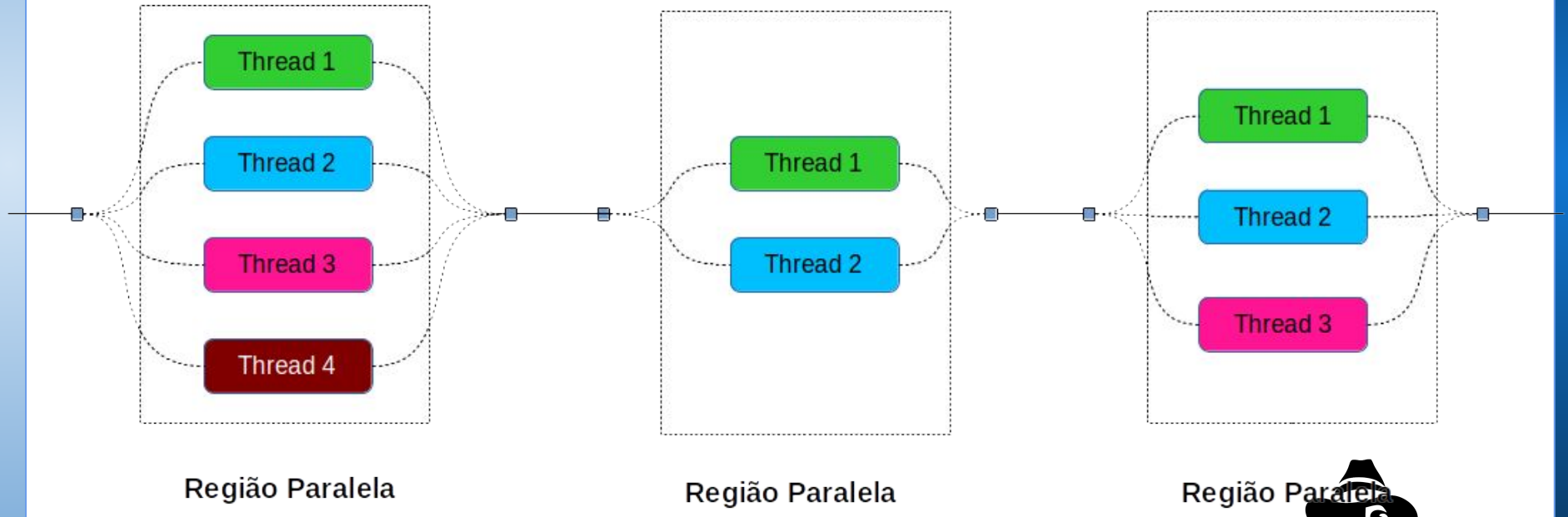
OpenMP Threads

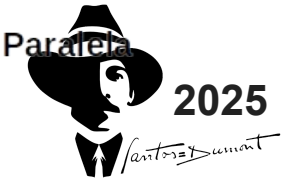
- Uma diretiva é uma linha especial de código fonte com significado especial apenas para determinados compiladores.
- Uma diretiva se distingue pela existência de uma sentinela no começo da linha.
- As sentinelas do OpenMP são:
  - C/C++:

    ```
    #pragma omp
    ```
  - Fortran:

    ```
    !$OMP  (ou C$OMP ou *$OMP)
    ```

OpenMP Threads





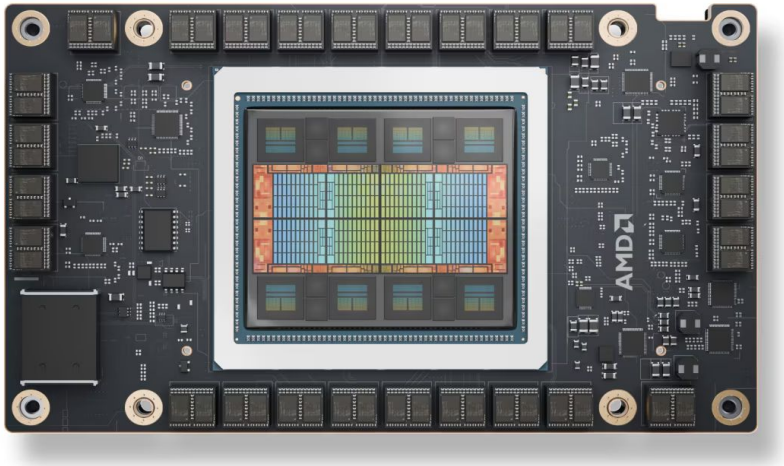
Programação com Aceleradores

- Os laços de computação mais intensiva são transferidos e executados em aceleradores, que podem ser FPGAs ou GPUs.
- Normalmente, as memórias do hospedeiro e do acelerador são separadas, sendo interconectadas por um barramento.
- A sincronização entre esses componentes é realizada por meio de rotinas específicas, que variam de acordo com a biblioteca utilizada.
- Entre as bibliotecas mais comuns destacam-se OpenACC, OpenMP, CUDA e OpenCL. No entanto, a programação com CUDA e OpenCL pode apresentar maior complexidade devido à sintaxe e ao nível de controle necessário sobre o *hardware*.
- Exemplo de acelerador: AMD Instinct Server Accelerators e NVIDIA

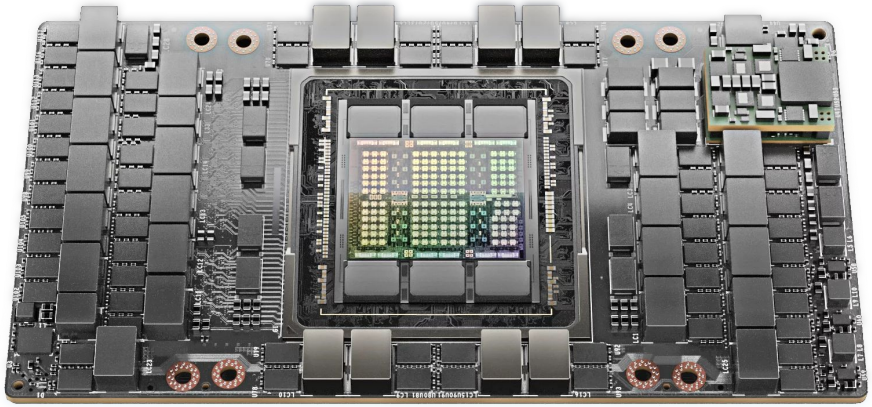
OpenMP Offloading

- O OpenMP *offloading* é uma extensão do OpenMP que permite descarregar (*offload*) tarefas computacionalmente intensivas para dispositivos aceleradores, como GPUs ou FPGAs, enquanto mantém parte da execução no hospedeiro (*host*).
- Essa funcionalidade aproveita o paralelismo massivo desses dispositivos para otimizar o desempenho.
- O programador utiliza diretivas OpenMP específicas para marcar trechos do código que serão executados no acelerador.
- No acelerador, as *threads* são organizadas em equipes, permitindo maior flexibilidade para explorar o paralelismo em larga escala.

Aceleradores (GPUs)



AMD Instinct™ MI325X



NVIDIA H-100

Programação Híbrida com MPI

- É uma abordagem eficiente para explorar paralelismo em clusters com múltiplos nós, cada um com diversos núcleos e um ou mais aceleradores (como GPUs).
- MPI
  - Utilizado para comunicação entre nós de um cluster.
  - Divisão do trabalho.
  - Redes de alta velocidade.
- OpenMP
  - Paralelização *multithreading*.
  - Descarga de tarefas intensivas para GPUs.
  - Maximização do desempenho computacional.

Programação Híbrida com MPI

- Vamos dividir o nosso estudo nas seguintes etapas:
  - MPI + OpenMP *threads*
  - OpenMP *offloading* (OpenMP + Aceleradores)
  - MPI + OpenMP *offloading* (MPI + OpenMP + Aceleradores)
  - MPI + OpenMP *threads* + OpenMP *offloading* (Balanceado)

MPI + OpenMP threads

MPI + OpenMP *threads*

- A combinação MPI + OpenMP com uso de *threads* convencionais pode ser feito de diversas maneiras, dependendo do nível de paralelismo suportado pelo ambiente de execução do MPI.
- Para esse fim, foi criada uma nova rotina de inicialização MPI_Init_thread () que define o nível de paralelismo desejado, em substituição à rotina MPI_Init() convencional.

MPI + OpenMP *threads*

- 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 o MPI tenha suporte para *threads*).
  - As chamadas MPI são feitas apenas da *thread* principal — quer 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.

MPI + OpenMP threads

- Mensagens *multithreaded*: 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).

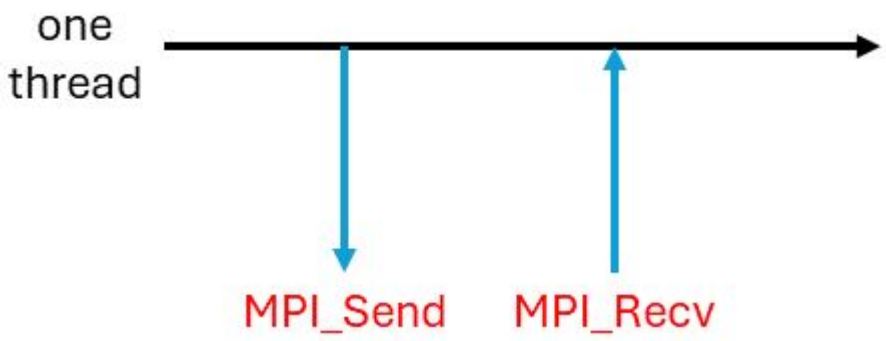
MPI_Init_thread

```
int MPI_Init_thread(int *argc, char ***argv, int required, int *provided)
```

- **required**: o nível de suporte de *thread* desejado (inteiro).
- **provided**: nível fornecido (inteiro), que pode ser menor que o solicitado.
- Essa rotina deve ser chamada antes de qualquer outra rotina MPI.
- Uma chamada para MPI_Init() tem o mesmo efeito de chamar MPI_Init_thread() com required = MPI_THREAD_SINGLE

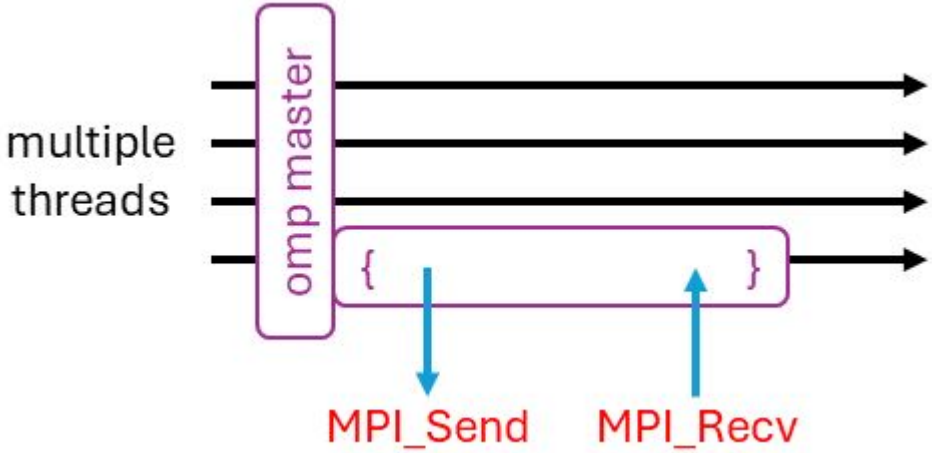
MPI_Init_thread( )

- **MPI_THREAD_SINGLE**: apenas uma *thread* será executada pelo processo e fará a comunicação.



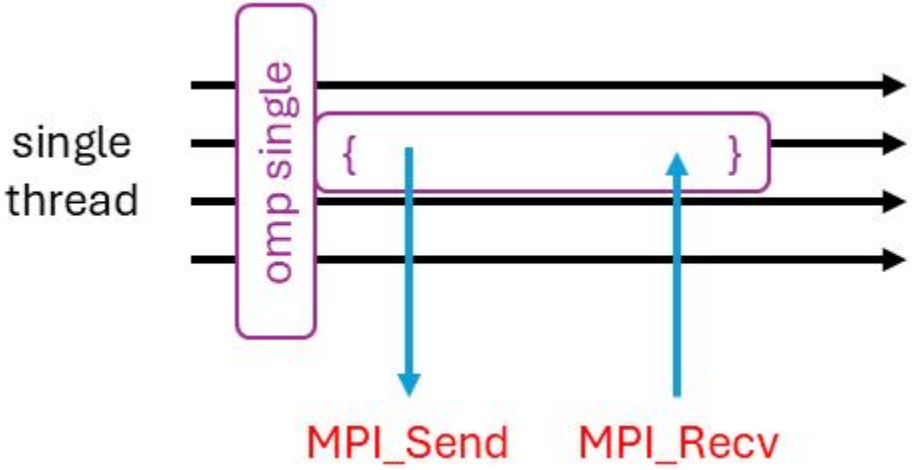
MPI_Init_thread( )

- **MPI_THREAD_FUNNELED**: o processo pode ser *multithreaded*, mas apenas a *thread* principal irá fazer as chamadas MPI (todas as chamadas MPI são afuniladas na *thread* principal).



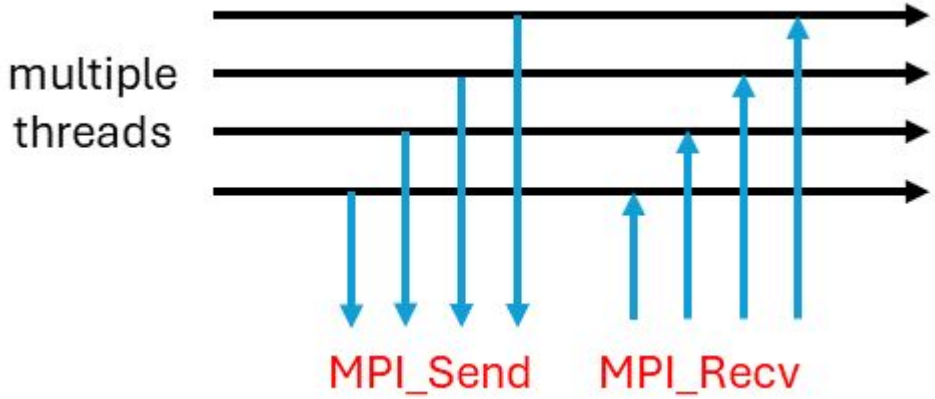
MPI_Init_thread( )

- **MPI_THREAD_SERIALIZED**: o processo pode ser *multithreaded*, e várias *threads* podem fazer chamadas MPI, mas apenas uma de cada vez: as chamadas MPI não são feitas concorrentemente por duas *threads* distintas (todas as chamadas MPI são serializadas).

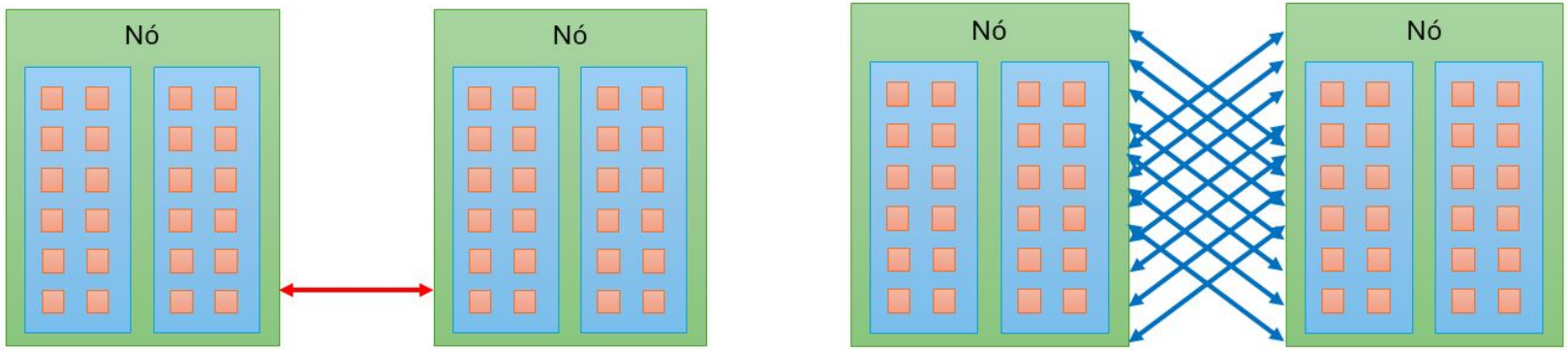


MPI_Init_thread( )

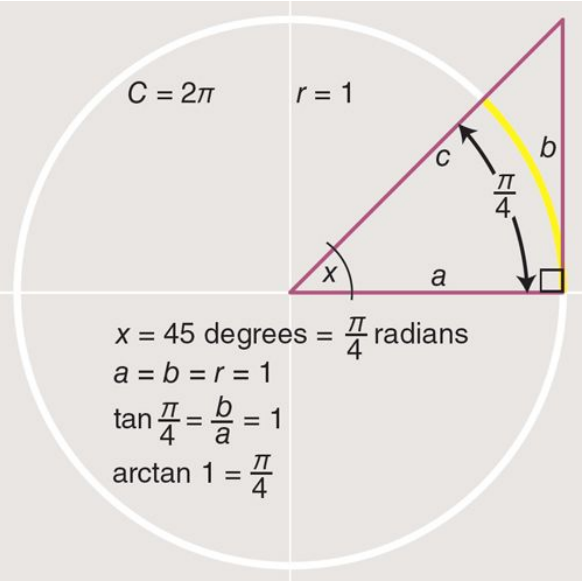
- **MPI_THREAD_MULTIPLE**: várias *threads* podem chamar o MPI, sem restrições.



Comunicação de Programação Híbrida



Cálculo de Pi



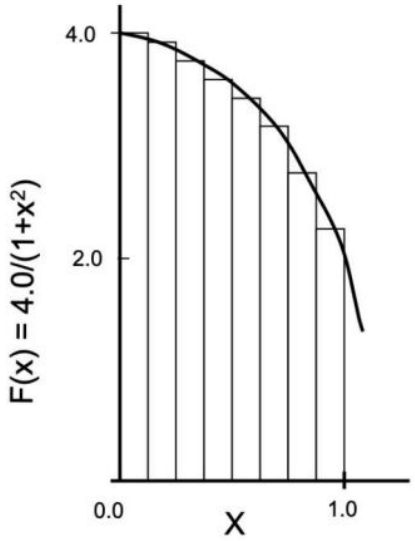
$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \simeq \frac{1}{N}\sum_{i=0}^{N-1} \frac{4}{1+x_i^2} \tag{4.1}$$

$$arctan(1) = \pi/4 \tag{4.2}$$

2025

Cálculo de Pi

Exemplo 1 – Cálculo e Pi



Nosso primeiro exemplo será o cálculo de Pi por integração numérica

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

Pode ser aproximado pela soma das áreas dos retângulos:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

https://web.inf.ufpr.br/erad2022/wp-content/uploads/sites/35/2022/08/Minicurso-OpenMP-GPU-V-17_04_22_compressed.pdf

Código Sequencial

```c
#include <stdio.h>
#include <time.h>
static long num_steps = 10000000000;
int main(int argc, char *argv[ ]) {
long int i;
double x, pi, sum = 0.0, step;
clock_t start_time, end_time;
     step = 1.0 / (double)num_steps;
     start_time = clock();     // Início da execução
     for (i = 0; i < num_steps; i++) {
          x = (i + 0.5) * step;
          sum += 4.0 / (1.0 + x * x);
     }
     pi = step * sum;
     end_time = clock();       // Tempo final
     double run_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
     printf("pi = %2.15f, %ld passos em %lf segundos\n", pi,
             num_steps,  run_time);
     return 0;
}
```

Pi - Versão MPI

```
int main(int argc, char *argv[]) {
  …
     rank = MPI_Comm_rank(MPI_COMM_WORLD);
     size = MPI_Comm_size(MPI_COMM_WORLD);
     step = 1.0 / (double)num_steps;
     start_time = MPI_Wtime(); // Tempo de início da execução
     for (i = rank; i < num_steps; i += size) {
          x = (i + 0.5) * step;
          sum += 4.0 / (1.0 + x * x);
     }
     MPI_Reduce(&sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);
     if (rank == 0) {
          pi = step * global_sum; // O processo 0 calcula o valor final de Pi
          run_time = MPI_Wtime() - start_time;
          printf("pi = %2.15f, %ld passos  em %lf segundos\n", pi, num_steps,
                 run_time);
     }
  …
}
```

Pi - Versão OpenMP threads

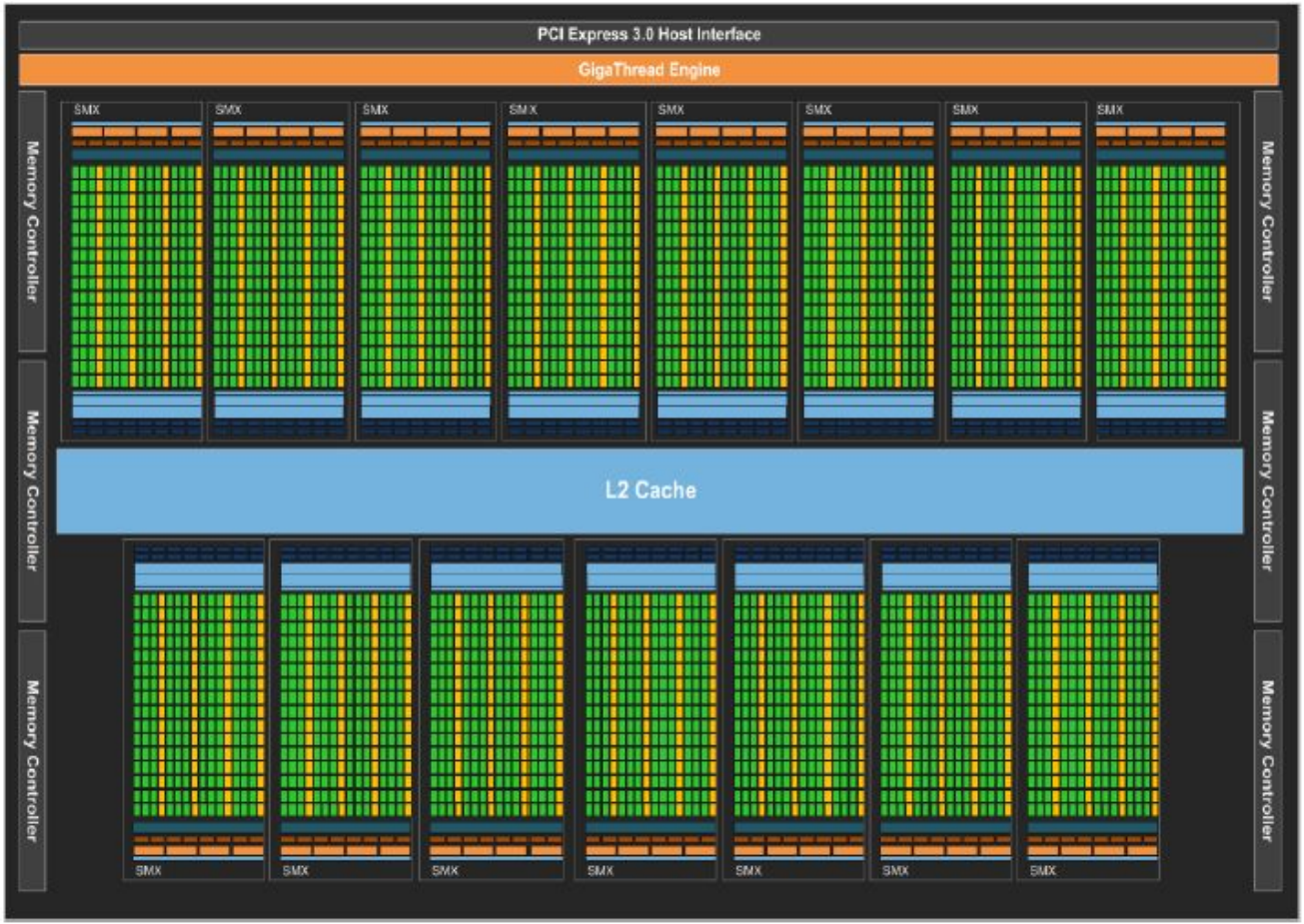
```
int main() {
static long num_steps = 10000000000;
double step = 1.0 / (double) num_steps;
double sum = 0.0, pi = 0.0, begin, end;
    begin = omp_get_wtime();
    #pragma omp parallel shared(sum, step, num_steps)
    #pragma omp for reduction(+:sum)
        for (long int i = 0; i < num_steps; i++) {
                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: %3.15f. O tempo de execução
            foi %3.15f \n", pi, end-begin);
    return 0;
}
```

Pi - Versão MPI + OpenMP threads

```
int main(int argc, char *argv[ ]) {
…
    rank = MPI_Comm_rank(MPI_COMM_WORLD);
    size = MPI_Comm_size(MPI_COMM_WORLD);
    start_time = omp_get_wtime();
    sum = 0.0;
    #pragma omp parallel shared(sum, step, num_steps, h, rank, size)
    #pragma omp for reduction(+:sum)
    for (i = rank; i <= num_steps; i += size) {
            double x = step * (double) (i+0.5);
            sum += 4.0 / (1.0 + x * x);
    }
    mypi = step * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    run_time = omp_get_wtime() - start_time;
    if (rank == 0)
            printf("pi = %lf, %ld passos em %lf segundos, diferença
%lf\n", pi, num_steps, run_time, pi - PI);
… }
```

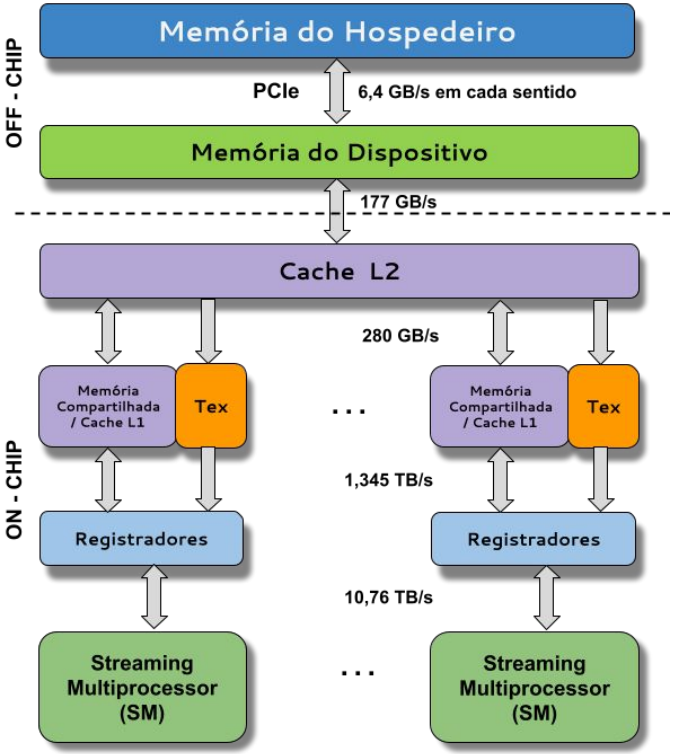
OpenMP offloading

Arquitetura GPU



Arquitetura GPU

Hierarquia de Memória da GPU



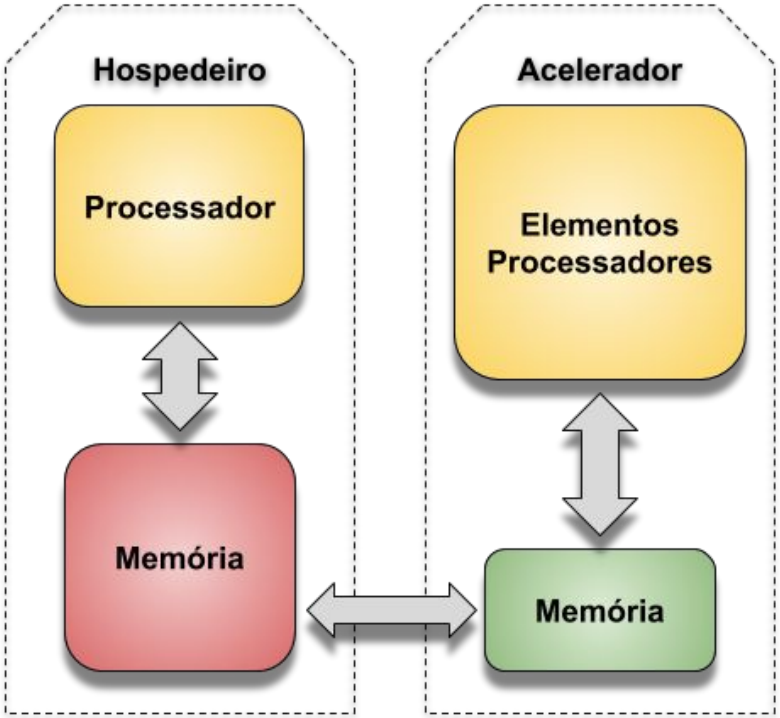
Modelo Hospedeiro + Dispositivo

- O modelo hospedeiro/dispositivo 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.
- Hospedeiro (host): o hospedeiro 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.

Modelo Hospedeiro + Dispositivo

- No contexto do OpenMP, o hospedeiro geralmente inicia a computação paralela e decide quando as tarefas devem ser enviadas para os dispositivos aceleradores.
- Dispositivo (device): o dispositivo 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.

Movimentação de Dados



Modelo Hospedeiro + Dispositivo

- No OpenMP, as regiões de código que são paralelizadas para o dispositivo são transferidas 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.

Compiladores para OpenMP offloading

- Estão disponíveis vários compiladores que são capazes de descarregar o código para execução em aceleradores utilizando o OpenMP.
- Entretanto, algumas combinações de processador, compilador e acelerador não são possíveis.
- Na tabela a seguir listamos algumas combinações que acreditamos serem viáveis.

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)

Principais Diretivas

- `#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 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.

OpenMP offloading

- Sincronização e Retorno dos Dados
  - Após a conclusão da execução no acelerador, os resultados são transferidos de volta para a memória do hospedeiro.
  - A sincronização entre hospedeiro e dispositivo é controlada automaticamente ou por meio de diretivas específicas (`#pragma omp taskwait`, por exemplo).
- O OpenMP permite especificar quais dados serão transferidos, quais vão persistir no acelerador ou serão movidos de volta ao hospedeiro por meio das cláusulas **map** (como `map(to: var)` ou `map(from: var)`).

Principais Diretivas

- `#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**.

Diretiva target -- Cláusula map

- A cláusula **map** especifica como os dados do hospedeiro são transferidos para o dispositivo (GPU) e de volta, facilitando a manipulação de grandes volumes de dados em computações paralelas.
- A sua principal função é gerenciar explicitamente a movimentação de dados entre o hospedeiro (normalmente um processador) e o dispositivo, (normalmente uma GPU).

Diretiva target -- Cláusula map

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

- map (to: variavel)
- map(from: variavel)
- map(tofrom: variavel)
- map(alloc: variavel)}

OpenMP offloading (map to: from:)

```
int main(int argc, char *argv[]) {
int N = 10,  a[N], b[N], c[N];
    for (int i = 0; i < N; i++) { // Inicia vetores no hospedeiro
         a[i] = i;
         b[i] = i * 2;
    }
    #pragma omp target map(to: a, b) map(from: c)
    // Região paralela no dispositivo
    for (int i = 0; i < N; i++)    // Dispositivo (GPU)
            c[i] = a[i] + b[i];

    for (int i = 0; i < N; i++) {  // Hospedeiro
            printf("%d ", c[i]);
     }
    printf("\n");
    return 0;
}
```

Diretiva target data

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

- 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.

OpenMP offloading (target data)

```
int main(int argc, char *argv[ ]) {
int N = 100;
double A[N], B[N];
    for (int i = 0; i < N; i++) {    // Hospedeiro
            A[i] = i * 1.0;
            B[i] = 0.0;
    }
    #pragma omp target data map(to: A[0:N]) map(from: B[0:N])
    #pragma omp target parallel for
    for (int i = 0; i < N; i++) {   // Dispositivo
            B[i] = A[i] * 2.0;
    }
    for (int i = 0; i < N; i++) {    // Hospedeiro
            printf("B[%d] = %f\n", i, B[i]);
    }
    return 0;
}
```

Diretiva *teams*

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

- 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*).
- É especialmente útil para dispositivos que possuem várias unidades de processamento paralelas como GPUs.

Diretiva *distribute*

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

- 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.

Diretiva *teams* + *distribute* + *parallel for*

```
#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).
- No exemplo a seguir apresentamos um programa para cálculo de Pi usando uma abordagem OpenMP + offloading.
- O trabalho é enviado para o acelerador com a diretiva `#pragma omp target teams distribute parallel for`.

Pi - Versão OpenMP offloading

```
int main(int argc, char *argv[])
{
  …
    #pragma omp target data map(tofrom: pi) map(to:num_steps, step)
device(1)
    #pragma omp target teams distribute parallel for reduction(+:pi)
    for (i = 0; i < num_steps; i++) {
            x = (i + 0.5) * step;
            sum += 4.0 / (1.0 + x * x);
    }
    pi = step * sum; // calcula o valor final de Pi
    run_time = omp_get_wtime() - start_time;
    printf("pi = %lf, %ld passos, computados em %lf segundos\n", pi,
    num_steps, run_time);
    return 0;
}
```

MPI + OpenMP offloading

MPI + OpenMP offloading

- Em um *cluster* moderno, dentro do contexto da computação de alto desempenho, cada nó pode ter:
  - Vários processadores com múltiplos núcleos cada um;
  - Uma ou mais GPUs associadas a cada nó;
  - Interconexão de alta velocidade (InfiniBand, por exemplo) para comunicação entre os nós.
- A abordagem híbrida de MPI + OpenMP + GPU permite explorar o paralelismo em diferentes níveis --- tanto entre nós quanto dentro de cada nó.

MPI + OpenMP offloading

- O MPI é utilizado para comunicação e paralelismo entre os nós.
- O OpenMP é utilizado para paralelismo dentro de um nó, com múltiplos núcleos da CPU
- OpenMP offloading ou CUDA é utilizado para enviar computação intensiva para a(s) GPU(s).
- Este uso conjunto pode fornecer enormes ganhos de desempenho, mas requer atenção a vários fatores, como balanceamento de carga, escalabilidade, gerenciamento de memória e otimização de comunicação.

MPI + OpenMP offloading

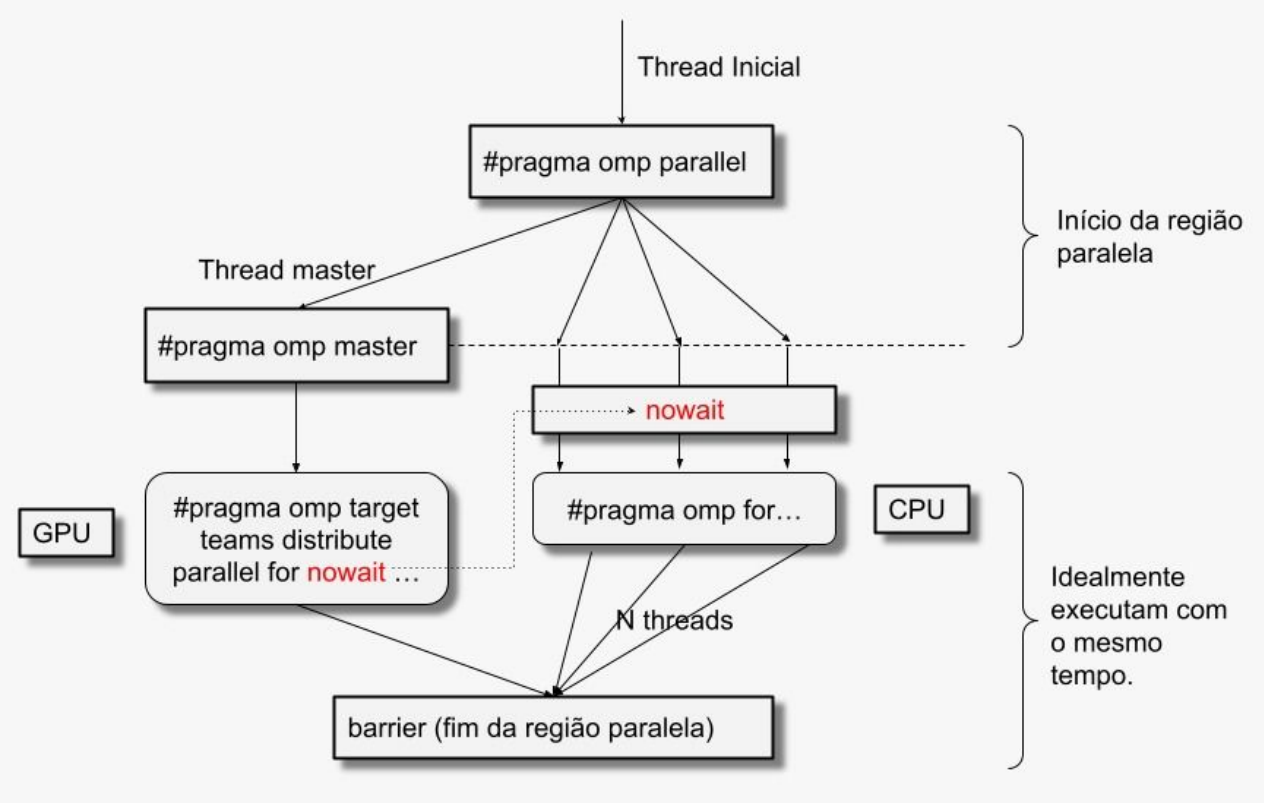
```
int main(int argc, char *argv[])
{ …
   MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
   rank = MPI_Comm_rank(MPI_COMM_WORLD);
   size = MPI_Comm_size(MPI_COMM_WORLD);
   #pragma omp target data map(tofrom:sum) map(to:size, num_steps, rank, step)
device(1)
   #pragma omp target teams distribute parallel for reduction(+:sum)
   for (i = rank; i < num_steps; i += size) { // Saltos de acordo com rank
     x = (i + 0.5) * step;
     sum += 4.0 / (1.0 + x * x);
   }
   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;
     printf("pi = %3.15f, %ld passos em %lf segundos\n", pi, num_steps, run_time);
   }
   MPI_Finalize(); // Finaliza o MPI
   return 0;
}
```

MPI + OpenMP offloading

- O ganho de desempenho da aplicação advém do fato de haver múltiplos aceleradores no *cluster*, permitindo a divisão do trabalho entre as diferentes GPUs.
- No exemplo acima 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.
- A seguir apresentamos um exemplo de código para cálculo de Pi que procura aproveitar toda a capacidade computacional disponível no *cluster*.

MPI + OpenMP offloading (Balanceado)

MPI + OpenMP offloading (Balanceado)



2025

MPI + OpenMP offloading (Balanceado)

```
#define GPU_WORKLOAD 70
int main(int argc, char *argv[])
{ …
   MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
   rank = MPI_Comm_rank(MPI_COMM_WORLD);
   size = MPI_Comm_size(MPI_COMM_WORLD);
   long int gpu_trials = num_trials * GPU_WORKLOAD / 100;
   long int cpu_trials = num_trials - gpu_trials;
   #pragma omp parallel num_threads(8)
   {
        #pragma omp master  // Thread master para comunicação e offloading GPU
        #pragma omp target data map(tofrom:local_count_gpu) device(1)
        #pragma omp target teams distribute parallel for nowait
reduction(+:local_count_gpu)
            for (i = rank; i < gpu_trials; i += size) {
                x = (double)rand() / RAND_MAX;
                y = (double)rand() / RAND_MAX;
                z = x * x + y * y;
                if (z <= 1.0) local_count_gpu++; // Verifica se o ponto está
no círculo
```

MPI + OpenMP offloading

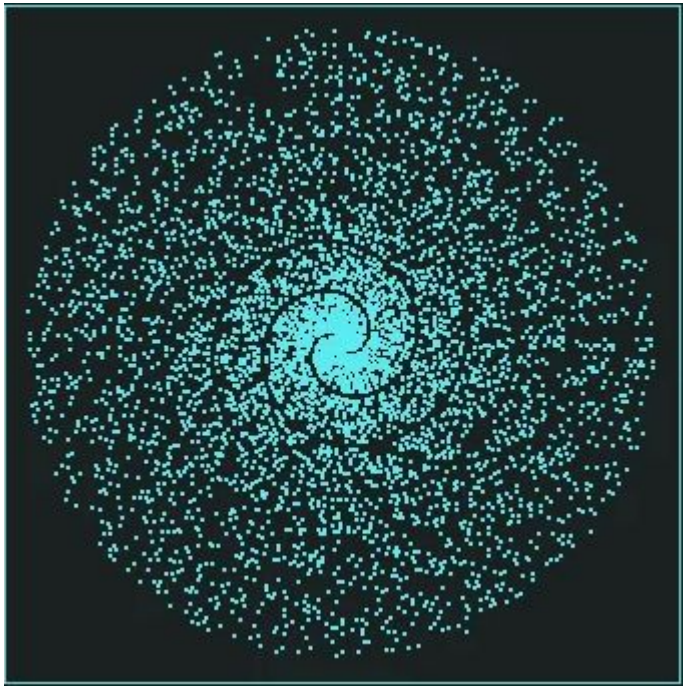
```
        #pragma omp for reduction(+:local_count_cpu)
        for (i = rank; i < cpu_trials; i += size) {
             x = (double)rand() / RAND_MAX;
                y = (double)rand() / RAND_MAX;
                z = x * x + y * y;
                if (z <= 1.0) local_count_cpu++;
         }
    }
    // MPI_Reduce para somar os resultados de todos os processos MPI
    long int local_count_total = local_count_gpu + local_count_cpu;
    MPI_Reduce(&local_count_total, &global_count, 1, MPI_LONG, MPI_SUM, 0,
MPI_COMM_WORLD);
    if (rank == 0) {
        pi = ((double)global_count / (double)num_trials) * 4.0; // Estimação de Pi
        run_time = omp_get_wtime() - start_time;
        printf("pi = %3.15f, %ld amostras em %lf segundos\n", pi, num_trials,
run_time);
    }
```

Estudo de caso - Números primos

Números primos

- O programa em questão serve para determinar a quantidade de números primos entre **0** e um determinado valor inteiro **N**.
- Embora possa parecer um programa trivial a princípio, ele tem algumas particularidades que o tornam um problema interessante.
- Na matemática, o Teorema do Número Primo (TNP) descreve a distribuição assintótica dos números primos entre os inteiros positivos.
- Ele formaliza a ideia intuitiva de que os números primos tornam-se menos comuns à medida que **N** aumenta, quantificando precisamente a taxa em que isso ocorre.

Distribuição dos números primos



2025

Qual a influência em nosso programa?

- Bom, a nossa primeira tentativa de paralelização seria dividir o total de **N** números igualmente entre os **P** processadores disponíveis, ou seja, **N/P** valores para cada uma dos processos ou *threads*.
- No entanto, há uma implicação importante: a distribuição dos números primos não é uniforme entre os inteiros.
- Esse fato mostra que a divisão direta é ineficaz, pois os processos ou *threads* que receberem intervalos com uma maior concentração de números primos terão uma carga de trabalho significativamente maior.

Solução sequencial

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
// Função para verificar se um número é primo
bool is_prime(long int n) {
    if (n <= 3) return true;
    if (n % 2 == 0 || n % 3 == 0) return false;
    for (long int i = 5; i * i <= n; i += 6)
        if (n % i == 0 || n % (i + 2) == 0) return false;
    return true;
}
int main(int argc, char *argv[]) {
long int N, total_primes=0;
    if (argc < 2) {
        printf("Valor inválido! Entre com o valor do maior inteiro\n");
        return 0;
    } else {
        N = strtol(argv[1], (char **) NULL, 10);
    }
    // Calcula só para os números ímpares, começando com 3
    for (int i = 3; i <= N ; i+=2)
        if (is_prime(i)) total_primes++;
    total_primes++;  // O número 2 também é primo
    // Imprime os resultados
    printf("Total de números primos entre 1 e %ld: %ld\n", N, total_primes);
    return 0;
}       You, 4 minutes ago • Uncommitted changes
```

Números primos

- Vamos apresentar apenas as soluções com uso de OpenMP offloading e MPI com OpenMP offloading.
- Na versão com OpenMP offloading apenas, a divisão do trabalho é mais e feita achando-se um ponto de partição entre o trabalho para os núcleos e o trabalho para a GPU.
- Na versão MPI com OpenMP offloading, a distribuição entre os nós é feita de forma que os processos possam ter tanto a mesma quantidade de números “baixos” como a de número mais “altos”.
- A distribuição de carga não é a melhor possível, mas os melhores resultados só podem ser obtidos por tentativa e erro.

Números primos (OpenMP offloading)

```
int total_primes_cpu = 0;
int total_primes_gpu = 0;

// Dividir o trabalho entre CPU e GPU
int split_point = (int)(N * .3); // 30% para CPU, 70% para GPU
if (split_point % 2 != 0)
    split_point--; // Se for ímpar, subtrai 1 para tornar par
double inicio = omp_get_wtime();
```

Números primos (OpenMP offloading)

```
// Criar duas tarefas assíncronas
#pragma omp parallel
{
    #pragma omp master
    {
        // Tarefa 1: Processamento na CPU
        #pragma omp task shared(total_primes_cpu, split_point)
        #pragma omp parallel for reduction(+:total_primes_cpu) num_threads(16)
        for (int                                    t; i +=2) {
            if
                total_primes_cpu++;
        }

        // Tarefa 2: Processamento na GPU
        #pragma omp task shared(total_primes_gpu, split_point)
        #pragma omp target teams distribute parallel for reduction(+:total_primes_gpu) map(tofrom: total_primes_gpu)
        for (int i = split_point + 1; i <= N; i+=2) {
            if (is_prime(i))
                total_primes_gpu++;
        }
        // Aguardar a conclusão das duas tarefas
        #pragma omp taskwait
    }
}
```

int total_primes_cpu

Números primos (OpenMP offloading)

```
// Soma os resultados da CPU e GPU
int total_primes = total_primes_cpu + total_primes_gpu;
total_primes++; // Adiciona o número 2, que não foi verificado
double tempo = omp_get_wtime() - inicio;
printf("Total de números primos entre 1 e %d: %d. Calculado em %f segundos.\n", N, total_primes, tempo);
printf("Primos encontrados na CPU: %d\n", total_primes_cpu);
printf("Primos encontrados na GPU: %d\n", total_primes_gpu);
printf("Ponto de Divisão: %d\n", split_point);
return 0;
```

Números primos (MPI + OpenMP offloading)

```
gpu_end = n  * GPU_WORKLOAD / 100;
if (gpu_end % 2 != 0) {
    gpu_end--; // Se for ímpar, subtrai 1 para tornar par
}
salto = numprocs*2;
start_time = omp_get_wtime();
```

Números primos (MPI + OpenMP offloading)

```
#pragma omp parallel
{
    #pragma omp master
    {
        // Tarefa 1: Processamento na GPU
    #pragma omp task shared(num_primes_gpu, rank, n)
    {
    #pragma omp target data map(tofrom:num_primes_gpu) device(1)
    #pragma omp target teams distribute parallel for reduction(+:num_primes_gpu)
        for (long int i = 3+(rank*2); i <= gpu_end; i += salto) {
            if (is_prime(i)) num_primes_gpu++;
        }
    }
      // Tarefa 1: Processamento na CPU
    #pragma omp task shared(num_primes_cpu, rank, n)
    #pragma omp parallel for reduction(+:num_primes_cpu) num_threads(4)
        for (long int j = gpu_end+(rank*2)+1; j <= n; j += salto) {
            if (is_prime(j)) num_primes_cpu++;
        }
    }
}
```

Números primos (MPI + OpenMP offloading)

```
long int local_num_primes = num_primes_gpu + num_primes_cpu;
printf("Total local %d  %ld\n", rank, local_num_primes);
MPI_Reduce(&local_num_primes, &global_num_primes, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
global_num_primes++; // Adiciona 2, que é o único número par primo
if (rank == 0) {
    end_time = omp_get_wtime();
    printf("Total de primos até %ld: %ld\n", n, global_num_primes);
    printf("Calculado em %lf segundos\n", end_time - start_time);
}
```

Considerações Finais

- O uso dessas técnicas se justifica pela crescente complexidade dos problemas computacionais e pela disponibilidade de hardware avançado, que demandam estratégias eficientes de paralelismo.
- 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.
- Apresentamos exemplos prático de programação híbrida MPI para aceleradores.
- Repositório:

https://github.com/Programacao-Paralela-e-Distribuida/HIBRIDA

Obrigado!

Escola Supercomputador Santos Dumont - 2025