🔹 1. Introdução à HPC (High-Performance Computing)

- HPC = uso de sistemas computacionais poderosos para resolver problemas complexos.
- Utiliza **vários processadores** (CPUs/GPUs) interligados para execução paralela.
- Envolve hardware, software, ferramentas e paradigmas de programação paralela (MPI, OpenMP, CUDA...).

🔹 2. Importância da HPC

Usada em áreas como:

- Astrofísica, bioquímica, IA/ML, meteorologia, ciências da Terra, segurança global, fusão nuclear.
- Permite **avanços científicos e tecnológicos** e inovação computacional.

🔹 3. Evolução da HPC

- Antigamente: processadores únicos e vetoriais.
- Atual: arquiteturas heterogéneas CPU-GPU → maior paralelismo.
- Exigem novas formas de programação (CUDA, paralelismo explícito).

🔹 4. Situação Atual da HPC

- Sistemas exascale ($\geq 10^{18}$ operações/segundo).
- Integração com IA/ML e uso via cloud.
- Foco em **eficiência energética** e **escalabilidade**.

Top 3 supercomputadores (novembro 2024):

1. **El Capitan (EUA)** – 1.742 Petaflops
2. **Frontier (EUA)** – 1.353 Petaflops
3. **Aurora (EUA)** – 1.012 Petaflops

🔹 5. Arquitetura Paralela e Escalabilidade

- Sistemas HPC → **memória distribuída**, organizados em clusters de *processing nodes (PN)*.
- Arquitetura projetada para escalar bem à medida que mais nós são adicionados.

Tipos de Topologia de Interligação:

1. **Mesh**: comunicação local eficiente.
2. **Torus**: malha com bordas conectadas → reduz latência.
3. **Hypercube**: altamente escalável, ótimo para grandes volumes de dados.
4. **Tree / Fat-Tree**: hierárquica, com mais largura de banda em níveis superiores.
5. **Star / Ring / Fully Connected**: usados em sistemas pequenos ou específicos.

🔹 **6. Nós de Processamento**

- Compostos por **multicore CPUs** e **manycore GPUs**.
- CPU → coordena tarefas, prepara dados.
- GPU → executa operações paralelas em massa.
- Computação heterogénea: **CPU + GPU complementares**.

🔹 **7. Decomposição Paralela**

- Baseada em dados → divide dados em blocos para processamento simultâneo.
- Pipeline de T etapas com subtarefas independentes.

Níveis de Paralelismo:

- **Fine-grained (SIMD)**: instrução única em vários dados.
- **Medium-grained (MIMD - memória partilhada)**: múltiplas threads.
- **Coarse-grained (MIMD - memória distribuída)**: múltiplos processos.

🔹 **8. Lei de Amdahl**

Fórmula do speedup teórico:

Soverall=1(1−P)+PNS_{overall} = \frac{1}{(1 - P) + \frac{P}{N}}

- P = fração paralelizável, N = número de unidades de processamento.
- O ganho é limitado pela parte **não paralelizável**.
- Em sistemas distribuídos, a **latência de comunicação (C)** reduz ainda mais o ganho:

Soverall=1(1−P)+C+PNS_{overall} = \frac{1}{(1 - P) + C + \frac{P}{N}}

🔹 **9. Ferramentas de Programação Paralela**

- **C/C++** com:
    - std::thread → memória partilhada.
    - MPI → memória distribuída.
    - CUDA → GPU, paralelismo fino.

🖥️ 1. Computer Architecture

Componentes principais:

- **CPU (processador):** Executa instruções e controla operações.
- **Memória Principal (RAM):** Armazena dados temporários (volátil).
- **Armazenamento Massivo:** Guarda dados de forma permanente.
- **I/O:** Comunicação com dispositivos externos.
- **Interconexão do Sistema:** Liga os componentes, normalmente via **bus**.

Arquitetura de von Neumann:

- Instruções e dados partilham a mesma memória.
- Execução cíclica: **Busca → Decodificação → Execução.**

Pipeline:

- Divide uma tarefa em subtarefas (estágios), aumentando o desempenho.
- **Speedup ideal ≈ número de estágios (N)**, se m (objetos) for grande.

Paralelismo a nível de instrução (ILP):

- **Várias instruções em paralelo.**
- Técnicas: múltiplas emissões, execução fora de ordem, prefetching, etc.

Hierarquia de Memória:

- L1 → L2 → L3 → RAM → Disco.
- Problema: **coerência de cache** em sistemas multicore.
- Solução: **Política write-through** nas caches L1 e L2.

Exemplo de otimização:

- Alterar a ordem dos ciclos for pode ter impacto enorme no desempenho por causa do **acesso à memória**.

📄 2. Program vs. Process

Programa vs Processo:

- **Programa:** Conjunto de instruções.
- **Processo:** Execução ativa de um programa.

Componentes de um processo:

- Espaço de endereçamento, contexto do processador, contexto de I/O, estado de execução.

Estados de um processo:

- **Running:** a ser executado.
- **Ready:** pronto para ser executado.
- **Blocked:** espera por evento externo.

Transições de estados:

- Ex: dispatch, timer-run-out, yield, sleep, wake-up, exit.

📝 3. Processes vs. Threads

Diferença:

- **Processo:** Tem espaço de memória próprio, contexto, etc.
- **Thread:** Entidade executável dentro de um processo — partilha recursos com outras threads.

Multithreading:

- Permite **concorrência** e **partilha eficiente de recursos**.
- Cada thread tem: contador de programa, registos e stack local.

Vantagens:

- Maior modularidade.
- Menor sobrecarga do sistema operativo (comparado com processos).
- Em sistemas SMP (vários núcleos), threads podem correr em paralelo.

Organização:

- Threads associadas a funções específicas.
- Partilham uma estrutura de dados global.

- O programa principal é uma thread também.

Estados da Thread:

- Igual ao processo: created, ready, run, blocked, terminated.

Tipos de Threads:

- **ULT (User-Level Threads):**
  - Geridas em espaço do utilizador (rápidas mas pouco eficientes).
  - Problema: uma thread bloqueada → todo o processo bloqueado.
- **KLT (Kernel-Level Threads):**
  - Geridas pelo kernel (mais lentas, mas permitem execução paralela real).
  - Threads podem continuar mesmo que outra bloqueie.

⚙️ 1. Princípios Gerais da Concorrência

Tipos de interação entre processos:

- **Processos Independentes:**
  - Não interagem diretamente.
  - Concorrência implícita: competem por recursos.
  - Ex: programas de utilizadores distintos.
- **Processos Cooperativos:**
  - Partilham dados ou comunicam explicitamente.
  - **Memória partilhada** ou **IPC** (pipes, mensagens, sockets, etc.).

🔐 2. Regiões Críticas e Exclusão Mútua

Regiões críticas:

- Trecho de código que acede a recurso partilhado.
- Deve evitar **condições de corrida** (race conditions).
- Requer **exclusão mútua**: só um processo pode aceder de cada vez.

Problemas possíveis:

- **Deadlock / Livelock:** processos ficam presos à espera de acesso.

- **Starvation:** um processo nunca consegue aceder ao recurso.

Requisitos de uma boa solução:

- Exclusão mútua garantida.
- Independente do número/velocidade dos processos.
- Sem interferência de outros processos.
- Sem adiamento indefinido.
- Tempo de execução finito na região crítica.

🧩 3. Recursos e Alocação

Tipos de recursos:

- **Físicos:** CPU, memória, I/O.
- **Lógicos:** estruturas de dados partilhadas, canais de comunicação, etc.

Alocação:

- **Pré-emptíveis:** podem ser retirados (CPU, RAM).
- **Não pré-emptíveis:** não podem ser interrompidos (impressora, buffer partilhado).

🔁 4. Deadlock (Interbloqueio)

Só envolve recursos não pré-emptíveis.

Condições necessárias (as 4 têm de ocorrer para haver deadlock):

1. **Exclusão Mútua**
2. **Espera com retenção** (hold and wait)
3. **Sem preempção** (no preemption)
4. **Espera circular** (circular wait)

🧠 Fórmula lógica:

- Se ocorrer deadlock ⇒ todas as 4 condições são verdadeiras.
- Para **prevenir**, basta negar **pelo menos uma**.

🚫 5. Prevenção de Deadlocks

Estratégias:

1. Negar "Hold and Wait"

- Processo pede **todos os recursos de uma vez**.
- Se não houver todos, **espera** sem reservar nada.
- Pode causar starvation → usar **política de aging** (prioridade aumenta com o tempo).

2. Negar "No Preemption"

- Se processo não conseguir todos os recursos, **liberta os que tem** e tenta de novo.
- Bloqueia em vez de usar "busy waiting".
- Também requer políticas justas (e.g., aging).

3. Negar "Circular Wait"

- Impor **ordem linear** nos recursos.
- Processos só podem pedir recursos por ordem crescente.
- Previne ciclos, logo, previne deadlock.

📝 Thread Pools – Conceitos Básicos

O que é um Thread Pool?

- Um **conjunto fixo de threads** (chamadas *worker threads*) prontas para executar tarefas.
- Em vez de criar/destruir threads repetidamente, **reutiliza** threads existentes.

Vantagens:

- Reduz overhead de criação/destruição de threads.
- Melhora a eficiência do agendamento de tarefas.
- Evita sobrecarga do sistema.
- Ideal para aplicações de alto desempenho (e.g., servidores web, jogos, big data).

⚙️ Como Funciona um Thread Pool

Componentes principais:

1. **Worker Threads:** threads persistentes à espera de tarefas.

2. **Task Queue:** fila onde as tarefas são armazenadas antes de serem executadas.
3. **Thread Manager:** componente que atribui tarefas às threads.
4. **Primitivas de sincronização:** mutex, condition_variable, etc., para garantir segurança concorrente.

Ciclo de execução:

1. Tarefa é adicionada à fila.
2. Uma thread livre pega na tarefa e executa.
3. Ao terminar, a thread volta ao pool.

🛠️ Implementação em C++

Bibliotecas utilizadas:

- <thread> – gestão de threads.
- <queue> – fila de tarefas.
- <functional> – armazena funções executáveis.
- <atomic> – variáveis atómicas para sincronização.
- <mutex> e <condition_variable> – controlo da concorrência.

Estrutura base (ThreadPool):

Contém:

- std::vector<std::thread> workers
- std::queue<std::function<void()>> queue
- mutexes, condition_variables
- enqueue(), dequeue_task(), wait() etc.

Exemplo:

thread_pool.init(2); // Inicia com 2 threads

thread_pool.enqueue([i] { /* tarefa */ });

thread_pool.wait(); // Espera que todas as tarefas terminem

✅ Resumo Final e Aplicações

Benefícios:

- Reutilização de threads poupa recursos.
- Execução paralela eficiente com menos overhead.
- Fácil de escalar e controlar o número de tarefas concorrentes.

Aplicações típicas:

- Servidores web com muitos clientes.
- Computação paralela científica.
- Processamento de grandes volumes de dados (e.g., MapReduce).
- Motores de jogos (tarefas em background).

🔁 Message Passing – Computação em Larga Escala

🧠 Princípio Geral

- **Troca de mensagens** é um método de comunicação que não precisa de espaço de endereçamento partilhado.
- Funciona em ambientes de processador único, multiprocessador e distribuídos.
- Um processo **PF (forwarder)** envia uma mensagem por um canal de comunicação.
- O processo **PR (recipient)** recebe a mensagem acedendo ao canal.

⏳ Sincronização

1. Sincronização Não Bloqueante (Non-blocking)

- Cada processo trata da sua própria sincronização.
- **Envio**: A função devolve logo, mesmo sem confirmação de receção.
- void msgSendNB(unsigned int destid, MESSAGE msg);
- **Receção**: Devolve sempre, mesmo que não tenha chegado mensagem.
- void msgReceiveNB(unsigned int srcid, MESSAGE *msg, bool *msg_arrival);

2. Sincronização Bloqueante (Blocking)

- O envio e receção ficam bloqueados até a mensagem ser recebida/enviada.
- **Envio bloqueia** até a mensagem ser recebida.

- **Receção bloqueia** até chegar uma mensagem.
- void msgSend(unsigned int destid, MESSAGE msg);
- void msgReceive(unsigned int srcid, MESSAGE *msg);

Tipos de sincronização bloqueante:

- **Rendezvous**: Ambos os processos devem estar prontos para troca; sem armazenamento intermédio.
- **Remote**: O remetente espera confirmação; pode haver armazenamento intermédio (ex: canais partilhados).

🧭 **Endereçamento (Addressing)**

- **Endereçamento Direto**: o remetente indica diretamente o destino (ex: pelo ID do processo).
- **Endereçamento Indireto**: usa-se um canal comum, como uma **mailbox**.

Mailboxes:

- Canal com armazenamento intermédio.
- Mensagens são guardadas por ordem de chegada.
- Pode existir:
    - Uma **mailbox** por processo;
    - Um **PO BOX** (grupo de mailboxes).

🔀 **Tipos de Comunicação**

1. Contextos de Comunicação

- **One-to-One**: entre dois processos.
- **One-to-Many**:
    - **Broadcast**: para todos os processos.
    - **Multicast**: para um grupo específico.

2. Mensagens Compostas

- **Scatter**: divide uma mensagem e envia partes diferentes para processos distintos.
- **Gather**: junta partes recebidas de vários processos numa só mensagem.

💁 **Produtor/Consumidor com Mailbox**

Exemplo com envio não bloqueante:

```
void main (unsigned int p) {
  while (1) {
    produceValue(&msg.info);
    msgSendNB(com, msg);  // Envio não bloqueante
    doSomeThingElse();
  }
}
```

Exemplo com receção bloqueante:

```
void main (unsigned int c) {
  while (1) {
    msgReceive(com, &msg);  // Espera por mensagem
    consumeValue(msg.info);
    doSomeThingElse();
  }
}
```

- **com** é o identificador da mailbox partilhada.
- Múltiplos produtores e consumidores podem usá-la.
- Capacidade da mailbox: até K mensagens.

💬 **Introdução ao MPI – Message Passing Interface**

📌 **O que é o MPI?**

- É uma **especificação de biblioteca** para programação com troca de mensagens (**message-passing**).
- Não é uma linguagem, mas sim uma **API padrão** para C/C++ e Fortran (usa #include <mpi.h>).
- Permite comunicação entre processos **sem memória partilhada**.
- Suporta escalabilidade e desempenho, podendo usar otimizações específicas de hardware.

- Todas as funções/constantes começam com MPI_ (ex: MPI_Init, MPI_Send).

🧱 Anatomia de um Programa MPI

Exemplo básico:

```
#include <mpi.h>

int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);          // Início
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Número do processo
  MPI_Comm_size(MPI_COMM_WORLD, &size); // Total de processos
  MPI_Finalize();                  // Fim
}
```

Compilação e execução:

```
$ mpic++ -Wall -O3 hello.cpp -o hello
$ mpiexec -n 4 ./hello
```

⚠️ Tratamento de Erros em MPI

- Todas as funções MPI retornam um código de erro.
- **Handlers de erro predefinidos:**
  - MPI_ERRORS_ARE_FATAL (padrão): termina todos os processos se houver erro.
  - MPI_ERRORS_RETURN: devolve o erro ao utilizador e continua a execução.

Exemplo com MPI_ERRORS_RETURN:

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
```

🔢 Tipos de Dados

- MPI define os seus próprios tipos:
  - MPI_INT, MPI_FLOAT, MPI_DOUBLE, etc.
- Usados para garantir **portabilidade e segurança de tipos** nas mensagens.
- Tipos especiais:
  - MPI_BYTE: byte bruto (como unsigned char).

    - MPI_PACKED: útil para empacotamento manual de estruturas complexas.

👥 Conceito de Comunicador (Communicator)

- Um **comunicador** define um **contexto isolado** onde ocorrem comunicações.
- Exemplo: MPI_COMM_WORLD inclui todos os processos lançados.
- Cada processo tem um **rank único** no grupo, entre 0 e size - 1.

✉️ Formato de uma Mensagem MPI

Cabeçalho (Envelope):

- Contexto de comunicação (comunicador).
- Rank de origem e destino.
- Tag (etiqueta) para identificar o tipo da mensagem.

Conteúdo (Payload):

- Tipo de dados (MPI_INT, etc.).
- Ponteiro para o buffer.
- Número de elementos.

🔁 Comunicação Ponto-a-Ponto (P2P)

Operações base:

MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);

MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);

Parâmetros:

- buf: ponteiro para os dados.
- count: número de elementos.
- datatype: tipo MPI (ex: MPI_INT).
- dest / source: rank do destino/origem.
- tag: etiqueta (usada para distinguir mensagens).
- comm: comunicador (ex: MPI_COMM_WORLD).

- status: contém info sobre origem, tag, erros (pode-se usar MPI_STATUS_IGNORE).

⚠️ **São bloqueantes por padrão.**

📦 **Exemplo de envio e receção:**

```
if (rank == 0) {
  char data[] = "Hello!";
  MPI_Send(data, strlen(data), MPI_CHAR, 1, 0, MPI_COMM_WORLD);
} else if (rank == 1) {
  char recData[100] = {0};
  MPI_Recv(recData, 100, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
  std::cout << "Recebido: " << recData << std::endl;
}
```

🤝 **Comunicação Coletiva em MPI (Message Passing Interface)**

A comunicação coletiva envolve **vários processos ao mesmo tempo**, em vez de um para um (ponto-a-ponto). Ela facilita a **distribuição estruturada e eficiente** de dados entre processos.

📣 **Broadcast (MPI_Bcast)**

- Envia a **mesma mensagem M** de um processo *root* para todos os outros, **incluindo ele próprio**.
- **Bloqueante**: todos os processos esperam até a mensagem ser recebida.
- Internamente: o root faz um envio; os outros fazem receção.

Sintaxe:

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm
comm);
```

📦 **Scatter (MPI_Scatter)**

- O *root* envia **partes diferentes** dos dados para cada processo.
- Cada processo recebe uma **parte distinta** (ex: linhas de uma matriz).
- **Bloqueante**.

Sintaxe:

```
int MPI_Scatter(
  void *sendbuf, int sendcount, MPI_Datatype sendtype,
  void *recvbuf, int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm
);
```

Variante: MPI_Scatterv

- Para **tamanhos desiguais** de dados por processo.
- Usa arrays sendcounts[] e displs[] (deslocamentos).

📥 **Gather (MPI_Gather)**

- Cada processo envia a sua mensagem para o *root*.
- O *root* junta todos os dados num só buffer.
- Útil para recolher resultados computados paralelamente.
- **Bloqueante**.

Sintaxe:

```
int MPI_Gather(
  void *sendbuf, int sendcount, MPI_Datatype sendtype,
  void *recvbuf, int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm
);
```

Variante: MPI_Gatherv

- Para receber **diferentes quantidades** de dados de cada processo.

➕ **Reduce (MPI_Reduce)**

- Cada processo contribui com dados.
- O *root* aplica uma **operação (como soma, max, min)** elemento a elemento.
- Só o *root* recebe o resultado final.
- **Bloqueante**.

Sintaxe:

```
int MPI_Reduce(
  void *sendbuf, void *recvbuf, int count,
  MPI_Datatype datatype, MPI_Op op,
  int root, MPI_Comm comm
);
```

Operações pré-definidas:

- MPI_SUM, MPI_PROD, MPI_MAX, MPI_MIN
- MPI_BAND, MPI_BOR, MPI_BXOR (bitwise)

🧪 **Exemplos Implementados**

- Cada função (MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Reduce) tem exemplos em C++:
    - Leitura clara dos *buffers*.
    - Uso de std::vector para armazenar os dados.
    - Saída por std::cout com identificação do processo via rank.

🧠 **Resumo de Computação em Larga Escala - MPI Parte II**

🟦 **Comunicação Não Bloqueante (Non-Blocking Communication)**

- **MPI_Isend** e **MPI_Irecv**:
    - Enviam e recebem dados **sem bloquear** o programa.
    - Permitem que o processo continue a executar código enquanto a comunicação ocorre.
- **Funções principais**:
    - MPI_Isend(buf, count, datatype, dest, tag, comm, request)
    - MPI_Irecv(buf, count, datatype, source, tag, comm, request)
    - MPI_Wait(request, status) → Espera até a operação terminar.
    - MPI_Test(request, flag, status) → Verifica se terminou, **sem bloquear**.
- **Exemplo**:

- Um processo envia um número e faz outras tarefas enquanto espera.
- Outro processo recebe e também faz trabalho paralelo.
- Usa MPI_Test num loop para ver se a receção acabou.

🟦 **Sincronização Coletiva (Collective Synchronization)**

- **MPI_Barrier**:
  - Garante que todos os processos **cheguem a um ponto comum antes de continuar**.
  - Útil para:
    - Medir tempos
    - Coordenar fases
    - Garantir estado consistente
- **Exemplo**:
  - Cada processo faz uma quantidade de trabalho (aleatória ou definida).
  - Depois todos usam MPI_Barrier() para esperar.
  - Só depois continuam e o processo 0 pode medir o tempo total.

🟦 **Ordenação com MPI (Sorting With MPI)**

🔹 **Problema:**

- Ordenar grandes volumes de dados (GB ou TB) é lento num só processador.

🔹 **Solução:**

- Usar **vários processos** para acelerar:
  1. **Distribuir dados**
  2. **Ordenar localmente**
  3. **Juntar resultados ordenados**

🔹 **Técnicas comuns:**

- Parallel Merge Sort
- Sample Sort
- Bitonic Sort

- Bucket Sort

🟦 **Parallel Merge Sort (com MPI)**

1. Dividir:

- MPI_Scatter divide os dados entre os processos.

2. Ordenar Localmente:

- Cada processo usa std::sort() ou qsort().

3. Merge em Várias Etapas:

- Usar **pairwise merge** (pares de processos trocam dados e mantêm metade ordenada).
- São feitas várias rondas com distâncias dobradas (1, 2, 4, ...).
- Ex: processo 1 envia para 0, 3 para 2, etc.

4. Merge Final:

- O processo 0 faz o merge final e fica com o resultado ordenado.

5. (Opcional) Recolher dados:

- Usar MPI_Gather se for necessário juntar tudo num só processo.

6. Finalizar:

- Chamar MPI_Finalize() no fim do programa.

🧠 **Resumo de SLURM - Workload Manager**

🟦 **O que é o Slurm?**

- **S**imple **L**inux **U**tility for **R**esource **M**anagement
- Gerenciador de tarefas e recursos para **clusters Linux**
- Usado para **agendar e executar jobs** em sistemas HPC (computação de alto desempenho)

🟦 **Funções principais do Slurm**

1. **Alocação de Recursos**
    - Atribui nós de computação para jobs dos utilizadores
    - Pode ser em modo exclusivo ou partilhado

2. **Lançamento e Monitorização de Jobs**
    - Executa jobs em paralelo
    - Permite verificar estado e desempenho dos jobs
3. **Gestão de Fila (Queue Management)**
    - Mantém fila de jobs
    - Decide a ordem de execução
    - Garante justiça e eficiência no uso dos recursos

🟦 **Componentes Principais do Slurm**

- slurmctld (Controller)
    - Coordena tudo: fila de jobs, agendamento, alocação de recursos
- slurmd (Node Daemon)
    - Corre nos nós de computação
    - Executa os jobs e envia estado de volta
- slurmdbd (opcional)
    - Guarda histórico de jobs e dados de uso
- slurmrestd (opcional)
    - API REST para integrações web

🔐 **Segurança**

- Usa **MUNGE** para autenticação e comunicação segura entre os componentes

📐 **Arquitetura e Configuração**

- Modelo **centralizado/distribuído**
- Ficheiro principal: slurm.conf
    - Define nós, partições (filas), políticas de agendamento, etc.

✳️ **Exemplo de Partições:**

- debug: curta duração (máx 1 hora)
- main: trabalho de produção (até 7 dias)

👨‍💻 **Como usar o Slurm (interação do utilizador)**

🔹 **Comandos principais:**

- sbatch: envia um job em lote (batch job)
- srun: executa um comando diretamente (usado dentro do sbatch)
- salloc: inicia uma sessão interativa

🔹 **Script de job (exemplo):**

```
#!/bin/bash
#SBATCH --job-name=meu_job
#SBATCH --output=saida.txt
#SBATCH --time=0-01:00:00
#SBATCH --partition=debug
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --mem=1G

./meu_programa arg1 arg2
```

📊 **Gestão e Monitorização de Jobs**

- squeue → ver jobs em fila
- sinfo → ver estado do cluster/nós
- scancel <jobid> → cancelar job
- sacct → ver histórico de jobs (se ativo)

🟩 **Slurm + MPI**

- Slurm **suporta MPI nativamente**:
  - Distribui processos (ranks) entre nós
  - Usa srun, mpirun ou mpiexec (dependendo da configuração)

🔹 **Exemplo de configuração para job MPI:**

```
#SBATCH --nodes=4
#SBATCH --ntasks=16        # Total de processos MPI
#SBATCH --ntasks-per-node=4
```

#SBATCH --cpus-per-task=1

srun ./meu_mpi_programa

- srun é o **método preferido**:
    - Automatiza distribuição de processos
    - Garante afinidade CPU, contabilidade, e ligação entre nós

---

🧠 **Resumo de CUDA Programming (NVIDIA)**

🟦 **O que é CUDA?**

- Plataforma e modelo de programação **paralela da NVIDIA**
- Permite usar **GPUs** para acelerar cálculos intensivos
- Baseado em extensões de C/C++ (também suportado em Python, Fortran, Java...)

🟦 **APIs CUDA**

- **Driver API** (baixo nível): Mais controlo, mas mais complexo
- **Runtime API** (alto nível): Mais simples, usa-se normalmente em CUDA C

🔹 **Nota**: **Não se podem misturar** Driver e Runtime API no mesmo programa

🟦 **Modelo de Compilação**

- Usa o compilador nvcc:
    - Separa código **host** (CPU) e **device** (GPU)
    - Kernel é o nome dado à função executada na GPU

🟦 **Abstrações CUDA**

- Hierarquia de **grupos de threads**
- **Memória partilhada** dentro de blocos
- **Sincronização de threads** dentro de um bloco

🟦 **Arquitetura da GPU**

- GPUs têm vários **SMs (Streaming Multiprocessors)**
- Cada SM executa blocos de threads

- O código é **escalável automaticamente** — corre mais rápido em GPUs com mais SMs sem mudar o código

🟦 **Estrutura típica de um programa CUDA**

1. **Aloca memória na GPU**
2. **Copia dados da CPU → GPU**
3. **Chama kernel**
4. **Copia resultados da GPU → CPU**
5. **Liberta memória da GPU**

🔹 Exemplo:

```
__global__ void helloFromGPU() {
 printf("Hello from GPU!\n");
}

int main() {
 helloFromGPU<<<1, 10>>>();
}
```

🟦 **Modelo de Execução CUDA**

- **Kernel** é uma função marcada com __global__
- Lançado com: kernel<<<gridDim, blockDim>>>();
- Cada **thread** tem:
    - ID local → threadIdx
    - Registos próprios e memória privada

🧱 **Indexação de Threads**

- threadIdx: 1D, 2D ou 3D (ex: threadIdx.x, threadIdx.y)
- Fórmulas:
    - 1D: id = x
    - 2D: id = x + y * Dx
    - 3D: id = x + y * Dx + z * Dx * Dy

- blockDim → dimensão do bloco
- blockIdx → posição do bloco na grelha

📝 **Sincronização & Cooperação**

- Threads no mesmo bloco podem:
    - Partilhar dados via **shared memory**
    - Sincronizar com __syncthreads() (barreira)

🧠 **Modelo de Memória CUDA**

- **Local** → privada de cada thread
- **Shared** → comum dentro de um bloco
- **Global** → visível por todas as threads (acesso mais lento)
- **Constant / Texture** → leitura rápida para dados estáticos

🔸 A **gestão de memória é manual** (alocação, cópia, libertação)

🟢 **Memória Unificada (Unified Memory)**

- Alternativa moderna: memória gerida automaticamente
- Usa cudaMallocManaged() para alocar uma única zona acessível pela CPU e GPU
- Vantagens:
    - Simplicidade
    - Boa para aplicações que alternam CPU ↔ GPU

🏷️ **Qualificadores de Funções**

Qualificador	Corre na	Chamado de	Uso
__global__	GPU	CPU	Lançamento de kernel
__device__	GPU	GPU	Funções auxiliares na GPU
__host__	CPU	CPU	(opcional) para CPU
__host__ __device__	Ambos	Ambos	Função acessível dos dois

⚠️ **Limitações dos Kernels**

- Devem ter void como tipo de retorno

- Não podem usar:
    - Ponteiros de função
    - Variáveis estáticas
    - Argumentos variáveis (...)
- Lançamento é **assíncrono** (CPU continua antes do GPU acabar)
