Sistemas HPC e topologias

1. Explique os três níveis de paralelismo (**fine, medium, coarse-grained**) e indique um exemplo de ferramenta apropriada para cada um.

O paralelismo fine-grained refere-se à execução simultânea de operações sobre vários dados com uma única instrução, como acontece em SIMD, comum em GPUs com CUDA. O paralelismo medium-grained ocorre em ambientes de memória partilhada, com múltiplas threads executando em paralelo, como com o uso de std::thread em C++. Já o paralelismo coarse-grained envolve processos independentes que comunicam via mensagens em memória distribuída, como com MPI.

2. O que se entende **por largura de banda e latência numa topologia de interligação**? Como influenciam o desempenho em sistemas de computação paralela?

A largura de banda refere-se à quantidade de dados que podem ser transmitidos por unidade de tempo entre dois nós. A latência refere-se ao tempo necessário para iniciar a transferência de dados entre dois nós, ou seja, o atraso inicial. Quanto mais a largura de banda melhor o desempenho do sistema principalmente se a quantidade de dados a ser transmitida for grande. Quanto menor a latência, melhor, uma vez que permite respostas mais rápidas.

3. Explica o funcionamento da topologia em árvore gorda (**fat-tree**) e justifica porque é considerada adequada para sistemas HPC.

A topologia fat-tree é uma rede hierárquica que tenta manter a largura de banda constante em todas as bisseções da rede. Cada nó requer apenas uma ligação e através do uso de switches com k portas, é possível ligar até k^3/4 nós, o que a torna muito escalável e eficiente para grandes sistemas HPC.

4. O que se entende por granularidade (**granularity**) na decomposição paralela? Compara os níveis fine-grained e coarse-grained, referindo vantagens e desvantagens. (2 valores)

Granularidade é o nível de detalhe do paralelismo aplicado a um algoritmo, ou seja, refere-se ao tamanho das unidades de trabalho em que um programa é dividido para ser executado em paralelo. O nível fine-grained realiza operações complexas a nível de variáveis e, por isso, permite maios paralelismo, no entanto, como a comunicação é mais frequente o overhead também é maior o que prejudica o desempenho. O nível coarse-grained realiza operações a nível de processo, tem um menor overhead e pode limitar o paralelismo.

5. O que distingue uma máquina com memória **partilhada** de uma máquina com memória **distribuída**? Indica uma vantagem e uma desvantagem de cada modelo.

Numa máquina com memória partilhada, todos os processadores têm acesso a um espaço de memória comum. Numa máquina com memória distribuída, cada processador tem o seu espaço próprio de memória local, e os processadores comunicam entre si através de mensagens. Uma vantagem e uma desvantagem da memória partilhada inclui o facto da comunicação ser feita de forma rápida e simples, no entanto, poderá haver concorrência no acesso à memória. Já na memória distribuída, uma vantagem é a escalabilidade, a facilidade em aumentar o número de processadores sem sobrecarregar o sistema. A desvantagem é a razão da comunicação ser mais complexa e lenta devido ao envio de mensagens.

6. O que é uma topologia em hipercubo (**hypercube**)? Que vantagem apresenta em relação à topologia **mesh**?

A topologia hypecube apresenta nós numa estrutura de cubo binário, em que os seu nós têm log2(n) ligações (n é potência de base 2) e a comunicação entre eles também é dada por esse resultado. É altamente escalável e eficiente para dados mais intensivos. As vantagens em comparação com a topologia mesh são inúmeras uma vez que, nesta topologia, todos os nós têm 4 ligações e a sua comunicação é dada por raiz quadrada de n, sendo n o número de ligações e, posto isto, não é tão escalável e eficiente. Na topologia mesh a largura de banda aumenta drasticamente à medida que a distância entre os nós aumenta também.

7. Explica o conceito de **pipeline paralela** e como pode ser utilizada para melhorar o desempenho em HPC.

Pipeline paralela refere-se à distribuição de tarefas em etapas sequencias, onde cada etapa é executada por uma unidade de processamento distinta. A execução é feita em paralelo, ou seja, enquanto uma unidade processa a fase atual de um dado, a próxima unidade já pode iniciar o processamento do dado anterior. Pode ser usada para melhorar o throughput, reduz a latência de sistemas mais complexos e permitir paralelismo.

8. O que se entende por **topologia de interligação**? Refere duas propriedades essenciais que devem ser tidas em conta para garantir escalabilidade em HPC.

Topologia de interligação refere-se à forma como os nós de processamentos estão interligados entre si numa rede de um sistema paralelo. De modo a garantir escalabilidade é necessário que a topologia tenha uma largura de banda alta pois quanto maior, melhor será o desempenho visto que a largura de banda representa a quantidade de dados que podem ser transferidos de um nó para outro; também é

necessário que tenha uma baixa latência, ou seja, que o tempo de início de comunicação de dados seja reduzido.

9. Compara a topologia **mesh** com a topologia **torus** em termos de número de ligações por nó, latência e escalabilidade.

Vou aproveitar e falar de todas as topologias. A topologia mesh tem uma estrutura em malha em que cada nó tem 4 ligações, o seu número de ligações é de raiz quadrada de n (n é o número de nós), é simples porém, à medida que a distância entre os nós aumenta, a latência também aumenta. A topologia torus tem também um estrutura em malha porém, o que a difere da topologia mesh é que os seus nós extremos estão interligados entre si o que melhora a latência, o número de ligações é igual e é tem melhor escalabilidade, pois mantém distâncias médias mais curtas. A topologia hypercube apresenta nós numa estrutura de nós binário, em que têm log2(n) ligações (n é uma potência de base 2), é altamente escalável e eficiente para dados mais intensivos. Por fim, a topologia fat-tree é representada por um rede hierárquica que tenta manter a largura de banda constate em todas as bisseções da rede. Cada nó requer apenas uma ligação e através de switches de k portas é possível ligar até k^3/4 nós, o que a torna muito escalável e eficiente para grandes sistemas HPC.

10. Define o conceito de **decomposição paralela orientada a dados**. Como é que este tipo de decomposição se relaciona com a estrutura pipeline?

A decomposição paralela orientada a dados consiste na distribuição dos dados de entrada em problemas menores, que podem ser executados em paralelo por diferentes processos. A estrutura pipeline refere-se à distribuição de tarefas por etapas sequenciais que são executadas por uma unidade de processamento distinta.

11. O que distingue uma topologia em malha (**mesh**) de uma topologia em árvore gorda (**fat-tree**)? Compara ambas em termos de escalabilidade, latência e número de ligações por nó.

A topologia em mesh tem um estrutura de grade/rede, cada nó tem 4 ligações, o seu número de ligações é dado pela raiz quadrada de n (n é o número de nós); escala razoavelmente no entanto quanto maior a distância entre nós, maior a latência. A topologia fat-tree é uma rede hierárquica que tenta manter constante a largura de banda a cada bissecção da rede. Cada no tem apenas uma ligação e, através de switches com k portas, é possível interligar até k^3/4 nos, o que a torna bastante escalável e eficiente para grandes sistemas HPC.

12. Define o conceito de **scalability**. De que forma a topologia influencia a escalabilidade de um sistema de computação paralela?

Escalabilidade corresponde à capacidade do sistema continuar com um bom desempenho à medida que são adicionados novos nós à topologia. A estrutura da topologia influencia a escalabilidade devido à forma como os nos são organizados e o número de ligações entre eles.

13. Explica o papel dos processamento em **pipeline** e da **granularidade** no desempenho de sistemas paralelos.

O processamento em pipeline consiste na distribuição de tarefas em problemas menores para serem executados por unidades de processamento distintos. A granularidade refere-se ao nível de detalhe nas operações dos processos e pode ser fine, medium ou coarse-grained. Cada um destes tipos opera de forma diferente. O fine-grained faz operações complexas com variáveis, o medium com threads e o coarse com processos.

Concorrência e sincronização

1. Distinga entre **programa, processo e thread**. Quais são as principais vantagens do uso de multithreading?

Um programa é um conjunto estático de instruções, enquanto um processo é a instância em execução desse programa, com seu próprio espaço de memória e contexto de execução. Uma thread, por sua vez, é uma unidade de execução dentro de um processo, que partilha os mesmos recursos com outras threads do mesmo processo. O uso de multithreading traz vantagens como melhor aproveitamento dos recursos do sistema, redução da sobrecarga do sistema operativo e maior modularidade no desenvolvimento de aplicações paralelas.

2. O que é uma **região crítica**? Descreva os requisitos essenciais de uma solução para garantir exclusão mútua.

Uma região crítica é um trecho de código que acede a um recurso partilhado e que, por isso, precisa garantir exclusão mútua para evitar condições de corrida, senão poderá levar a perdas de informação. Uma boa solução para garantir exclusão mútua é assegurar que apenas um processo ou thread possa aceder ao recurso partilhado de cada vez, independentemente do número ou velocidade dos processos.

3. Quais são as principais causas de um **deadlock**? Apresente uma estratégia para prevenir este problema.

Os quatro requisitos para que ocorra um deadlock são: mutual exclusion, hold and wait, no preemption e circular wait. Se todas estas condições estiverem presentes, o sistema pode entrar num estado de deadlock. Uma forma de prevenir deadlocks

é negar a condição de hold and wait, exigindo que os processos requisitem todos os recursos de uma só vez.

4. Define **starvation**. Em que se distingue de um **deadlock**? Que técnica pode ser usada para evitar starvation?

Starvation ocorre quando um processo fica ativo porém em espera indefinida por recursos porque outros processos com prioridade mais alta estão em execução. Esta problema pode ser mitigado usando a técnica de aging que consiste em aumentar a prioridade do processo "esquecido" consoante o tempo vai passando. A distinção entre starvation e dealock reside no facto do processo em starvation estar ativo e no deadlock estar bloqueado permanentemente.

5. O que é um **monitor** e como **garante exclusão mútua**? Explica o papel das primitivas wait() e signal() com um exemplo simples.

Um monitor é um módulo que encapsulado dados partilhados e, assim, assegura automaticamente exclusão mútua, uma vez que apenas uma thread pode estar dentro de um monitor. A primitiva wait() serve para bloquear uma thread até que uma condição seja verdadeira e a primitiva signal() acorda uma thread bloqueada numa condição.

6. Quando várias threads acedem a uma região crítica com sincronização, por que razão se deve evitar o uso de **busy waiting**? Que alternativa pode ser usada?

Busy waiting ocorre quando uma thread fica num ciclo ativo a verificar repetidamente se pode entrar na região crítica, consumindo CPU sem realizar trabalho útil. Isso leva a desperdício de recursos computacionais, pois a CPU está ocupada com uma thread que só está à espera, impedindo outras threads ou processos de serem executados eficientemente. Outras alternativas são o uso de mutexes, semáforos ou monitores.

7. Define o que é uma **região crítica**. Que problemas podem surgir se várias threads entrarem na mesma região crítica sem controlo?

Um região crítica é um trecho de código que acede a um recurso partilhado e que, por isso, precisa de garantir exclusão mútua de forma a evitar condições de corrida. Caso haja entrada de várias threads numa região crítica sem controlo, os resultados podem acabar por ficar incorretos ou poderá haver corrupção de dados.

8. O que é uma **race condition**? Que mecanismos podem ser utilizados para a evitar? Dá um exemplo simples.

Uma race condition é quando duas ou mais threads acedem ou modificam a mesma variável ou recurso partilhado. Para evitá-la, utilizam-se mecanismos como mutexes, semáforos ou monitores.

9. Que situações conduzem a **deadlock**? Refere e explica as quatro condições necessárias para que ele ocorra.

O deadlock acontece quando todas as quatro situações acontecem ao mesmo tempo, são elas, exclusão mútua, hold and wait, circular wait e no preemption. Na exclusão mútua apenas um thread deve ter acesso a recursos partilhados; na hold and wait um processo fica com um recurso a aguardar por outros recursos que estão a ser usados por outros processos; no no-preemption, o recurso não pode ser retirado à força de um processo sendo apenas possível libertá-lo pelo processo que o detém; e, por fim, no circular wait os processos estão num ciclo fechado onde cada processo está à espera de um recurso detido pelo próximo processo.

10. O que são **synchronization primitives**? Refere duas e explica os seus efeitos.

Synchronization primitives são primitivas para garantir que as threads estejam sincronizadas em certo ponto do código e que, por isso, garantem que não haja condições de corrida. Exemplos de synchronization primitives podem ser mutexs, monitores e semáforos.

11. O que é **busy waiting**? Porque é considerado ineficiente?

Busy waiting refere-se a uma thread que esteja num ciclo ativo a verificar repetidamente se pode entrar numa região crítica, o que consome CPU sem realizar trabalho útil. É ineficiente porque o CPU está ocupado com uma thread que apenas está a espera impedindo assim que outros threads ou processos sejam executados.

12. Um programa concorrente pode entrar em **deadlock**. Que técnica pode ser usada para evitar que ocorra a condição **hold and wait**? Explica.

Para um programa concorrente entrar em deadlock, é preciso que quatro condições sejam verificadas em simultâneo, são elas, exclusão mútua, hold and wait, no-preemption e circular wait. Para evitar que ocorra deadlock pode se negar a condição hold and wait fazendo com que os recursos sejam todos acedidos de uma vez só.

MPI

1. O que distingue a comunicação **bloqueante** da **não bloqueante** em MPI? Dê exemplos práticos de aplicação de cada uma.

Na comunicação bloqueante em MPI, funções como MPI_Send e MPI_Recv fazem com que o processo aguarde até que a operação esteja concluída, o envio espera que a mensagem seja copiada e a receção bloqueia até que uma mensagem chegue. Já na comunicação não bloqueante, com MPI_Isend e MPI_Irecv, o processo continua a sua execução imediatamente após o pedido de envio ou receção.

2. Descreva os modos de comunicação coletiva **MPI_Scatter** e **MPI_Gather**. Em que cenários são mais vantajosos do que **MPI_Send** e **MPI_Recv**?

MPI_Scatter é uma operação coletiva em que um processo root envia partes diferentes de um vetor para cada processo do grupo, sendo ideal para distribuir dados a serem processados localmente. MPI_Gather, por outro lado, recolhe dados de todos os processos e junta-os num só vetor no processo root. Estas operações são mais eficientes do que múltiplas chamadas ponto-a-ponto (MPI_Send e MPI_Recv) porque reduzem a sobrecarga de gestão de comunicação e utilizam otimizações internas de MPI, sendo especialmente vantajosas em cenários de inicialização e recolha de resultados paralelos.

3. Compare os modos de comunicação ponto-a-ponto **MPI_Send** / **MPI_Recv** com **MPI_Isend** / **MPI_Irecv**.

MPI_Send e MPI_Recv são operações bloqueantes, ou seja, o processo aguarda até que a operação termine. Já MPI_Isend e MPI_Irecv são versões não bloqueantes: retornam imediatamente e a comunicação continua em segundo plano. Estas últimas permitem sobrepor computação com comunicação, sendo úteis quando se deseja reduzir tempos de espera em programas com forte dependência entre processos.

4. Descreva como funciona a operação **MPI_Reduce** e apresente um exemplo de uso prático.

A função MPI_Reduce agrega valores provenientes de todos os processos de um grupo, aplicando uma operação (como soma ou máximo), e envia o resultado final para um processo root. Por exemplo, num programa que calcula somas locais de cada processo, MPI_Reduce pode ser usada para somar todos esses valores e entregar o total ao processo 0. Isto é útil para cálculos de totais, médias ou qualquer agregação global.

5. Explique a diferença entre **endereçamento direto** e **indireto** em comunicação por mensagens. Dê exemplos práticos.

No endereçamento direto, o remetente indica explicitamente o recetor (ex: pelo rank em MPI), enquanto no indireto a comunicação é feita através de uma estrutura

intermediária, como uma mailbox. O indireto oferece maior desacoplamento entre processos, sendo ideal para cenários com múltiplos produtores e consumidores.

6. O que é **MPI_Barrier** e como ele pode ser usado para sincronizar fases de execução?

MPI_Barrier sincroniza todos os processos de um comunicador, garantindo que nenhum avance antes dos outros. É útil para delimitar fases de execução.

7. O que é o **rank** de um processo MPI? Que função é usada para o obter e para que serve?

Em MPI, rank é o identificar (ID) único de um processo que é usado para obter o número do processo no comunicador através de MPI_Comm_rank.

8. Explica o funcionamento de **MPI_Gather** e **MPI_Reduce**. Em que situações são usados? Dá um exemplo para cada.

A função MPI_Gather serve para recolher dados diferentes mas de tamanhos iguais de vários processos do comunicador e juntá-los num único array no processo root. Pode ser usado quando se quer agregar dados distribuídos de vários processos para análise ou saída centralizada. MPI_Reduce faz o mesmo porém, ao recolher os dados, aplica sobre eles uma operação escolhida pelo utilizador, e retorna o resultado ao processo root. Pode ser usado quando se quer combinar resultados locais para obter um resultado global (ex: soma , valor máximo).

9. Num sistema **com 6 processos** MPI, pretende-se distribuir os dados de uma matriz (6x6) para **multiplicação paralela**.

Para multiplicar paralelamente uma matriz 6x6 com 6 processos MPI, distribui-se a matriz A em linhas usando MPI_Scatter, transmite-se a matriz B completa com MPI_Bcast, cada processo calcula sua parte do produto, e os resultados são reunidos no processo root com MPI_Gather.

10. Explica o que é um **comunicador** MPI e como é usado para organizar a comunicação entre processos. Dá o nome do comunicador predefinido mais comum.

Em MPI, todos os processos são instanciados com o comunicador MPI_COMM_WORLD que é o objeto que define o contexto de comunicação e representa o grupo de todos os processos apresentados pela aplicação.

11. Qual a diferença entre **MPI_Bcast** e **MPI_Scatter**? Em que tipo de situações usarias cada uma?

A função MPI_Bcast envia uma mensagem igual, através do processo root, para todos os processos, incluindo ele mesmo, no comunicador. A função MPI_Scatter

envia partes diferentes de um vetor mas com o mesmo tamanho, através do processo root, para todos os processos, incluindo ele mesmo. São necessários para, por exemplo, mandar uma mensagem a dizer que iniciou ou para distribuir uma mensagem pelos outros processos.

12. Que função MPI permite que todos os **processos recebam a mesma informação**? E qual permite que cada processo receba um subconjunto dos dados? Explica com exemplos.

Em MPI, para todos os processos receberem a mesma informação, dada pelo processo root, a função a usar é a MPI_Bcast. Para receberem um subconjunto de dados, dados pelo processo root, a função é a MPI_Scatter.

13. Define o conceito de **rank** e de **comunicador** em MPI. Que papel têm na organização do programa?

Rank, em MPI, corresponde ao identificador único (ID) de um processo que é usado para obter o número do processo no comunicador através de MPI_Comm_rank. Um comunicador é o objeto que define o contexto da comunicação e representa o grupo de todos os processos apresentados pela aplicação, é dado por MPI_COMM_WORLD.

14. Descreve como usarias **MPI_Scatter** e **MPI_Reduce** para **calcular** o máximo de uma lista de 1000 inteiros usando 5 processos.

O MPI_Scatter dividiria 200 inteiros pelos 5 processos e o MPI_Reduce iria agregar os resultados calculando o máximo e devolvendo-o ao processo root.

CUDA e GPU

1. Explique como a **hierarquia de threads** em CUDA (blocos e grelha) influencia a **organização do paralelismo**.

Em CUDA, as threads são organizadas em blocos (blocks), que por sua vez formam uma grelha (grid). Essa hierarquia permite escalabilidade automática e facilita a divisão do trabalho. Cada bloco contém múltiplas threads, que podem cooperar entre si usando memória partilhada e sincronização. A organização em grelha permite distribuir grandes volumes de trabalho entre milhares de threads, tornando possível aproveitar ao máximo a estrutura paralela da GPU, onde cada SM (Streaming Multiprocessor) pode executar vários blocos simultaneamente.

2. O que significa **SIMT** (Single Instruction, Multiple Threads)? Que estrutura da GPU executa threads neste modelo?

SIMT é uma arquitetura usada em GPUs da NVIDIA para executar milhares de threads em paralelo, com base um apenas um fluxo de instruções. SI refere-se a todas as threads executarem a mesma instrução, dentro do mesmo grupo (warp) e ao mesmo tempo. MT refere-se a cada thread ter o seu próprio contexto, podendo operar em dados diferentes.

3. Como se **calcula o índice global** de uma thread dentro de um kernel CUDA? Indica a fórmula e explica para que serve.

O índice global de uma thread em CUDA é calculado pela fórmula threadIdx.x + blockIdx.x * blockDim.x e serve para identificar unicamente cada thread na grelha, permitindo que aceda a diferentes partes dos dados em paralelismo.

4. Indica **três tipos de memória** em CUDA, descrevendo as suas principais características e diferenças.

Três tipos de memória em CUDA são a global, que é visível por todas as threads de todos os blocos e pelo kernel, está localizada na memória DRAM da GPU o que se reflete numa alta latência; a partilhada que é acessível apenas pelas threads do mesmo bloco, tem baixa latência e a sua comunicação é feita de forma rápida; por fim, a constante que é acessível por todas as threads de todos os blocos, é apenas de leitura e tem valores fixos usados por todo o kernel e tem baixa latência.

5. O que é um **kernel CUDA**? Explica a sintaxe de lançamento e o significado dos parâmetros gridDim e blockDim.

Um kernel CUDA é uma função executada em paralelo por múltiplas threads na GPU. É lançado através da sintaxe nome_kernel<<<gridDim, blockDim>>>, onde gridDim é representado pelo número total dos blocos por grid e blockDim é o número de todas as threads dentro de cada bloco.

6. Para que serve a **função __syncthreads()** dentro de um kernel CUDA? Em que casos deve ser usada?

A função __syncthreads() serve para sincronizar todas as threads de um bloco num certo ponto do código servindo como barreira. Apenas quando todas as threads chegarem àquela função é que poderão todas prosseguir. Deve ser usada em casos onde há partilha de dados na memória partilhada de forma a evitar condições de corrida.

7. Explica o papel da memória partilhada (**shared memory**) em CUDA. Em que se distingue da memória global?

A memória partilhada é visível por todas as threads do mesmo bloco, sendo usada para comunicação rápida e eficiente dentre essas threads. Distingue-se da memória global por ser muito mais rápida uma vez que a global é visível por todas

as threads de todos os blocos e pelo kernel e pela sua localização ser na memória DRAM da GPU; também tem mais latência.

8. O que é o modelo de **computação heterogénea** usado em CUDA? Que papel desempenham a CPU e a GPU?

O modelo de computação heterogénea usado em CUDA baseia-se na divisão de tarefas entre o CPU (host) e o GPU (device), onde o CPU gere o controlo e a lógica do programa e a GPU executa as operações de forma paralela usando milhares de threads. Este modelo permite combinar a alta capacidade de paralelismo da GPU e a capacidade da CPU de gerir a execução, memória e os lançamentos de kernels.

9. Explica por que razão o acesso à **shared memory** é mais eficiente do que à **global memory**.

O tipo de memória partilhada é visível por todas as threads de um único bloco, sendo usada para comunicação rápida e eficiente dentre essas threads. A memória global é menos eficiente uma vez que é visível por todas as threads de todos os blocos e pelo kernel e também pela sua localização ser na memória DRAM do GPU tendo mais latência.