Introdução à Programação de Alto Desempenho

Ricardo de la Rocha Ladeira

Introdução

Nesta aula, além de fazer uma breve apresentação do que será trabalhado e de como será a disciplina, estão apresentados alguns conceitos básicos sobre *Programação de Alto Desempenho* e *Programação Concorrente*, importantes para o andamento da disciplina e presentes na ementa. O objetivo principal é permitir que o(a) estudante se aproprie destes conceitos. Ao final, exercícios serão disponibilizados para possibilitar a aplicação dos conhecimentos adquiridos e o aprofundamento da compreensão sobre a Programação Concorrente.

Motivações para o estudo da Programação de Alto Desempenho (baseadas em [1])

1) **Melhorar a eficiência de programas**: A Programação de Alto Desempenho visa otimizar o desempenho de programas, tornando-os mais eficientes em termos de tempo de execução, utilização de recursos computacionais e consumo de energia. Isso é especialmente importante para lidar com tarefas computacionalmente intensivas, grandes volumes de dados ou aplicações que exigem tempo de resposta rápido.
2) **Aproveitar o potencial de hardware avançado**: Com o avanço da tecnologia, os computadores estão se tornando cada vez mais poderosos, com processadores *multicore*, GPUs (*Graphics Processing Units*) e arquiteturas paralelas. A Programação de Alto Desempenho permite aproveitar todo o potencial desses hardwares avançados, distribuindo e paralelizando tarefas para obter um melhor desempenho.
3) **Resolver problemas complexos**: Muitos problemas na ciência, engenharia e outros campos exigem soluções eficientes e rápidas. A Programação de Alto Desempenho fornece as ferramentas e técnicas necessárias para enfrentar problemas complexos,

permitindo o processamento rápido de grandes conjuntos de dados, simulações complexas e modelagem computacional avançada.

4) **Otimizar o uso de recursos**: Em muitos casos, os recursos computacionais são limitados e caros. A Programação de Alto Desempenho ajuda a otimizar o uso desses recursos, reduzindo o tempo de execução de programas, aumentando a capacidade de processamento e minimizando o desperdício de recursos.
5) **Melhorar a escalabilidade de sistemas**: A Programação de Alto Desempenho é fundamental para construir sistemas escaláveis, capazes de lidar com cargas de trabalho crescentes. Ao aproveitar técnicas como a paralelização e a distribuição, é possível projetar sistemas que podem lidar com grandes volumes de dados e aumentar sua capacidade de processamento de forma eficiente.
6) **Avançar na pesquisa científica**: A Programação de Alto Desempenho é crucial para avançar na pesquisa científica e acadêmica. Permite a execução de simulações complexas, modelagem computacional, análise de dados em larga escala e a resolução de problemas científicos desafiadores.
7) **Desenvolver aplicações de tempo real**: Muitas aplicações exigem respostas rápidas e em tempo real, como jogos, processamento de áudio e vídeo, sistemas de controle e monitoramento. A Programação de Alto Desempenho é essencial para garantir que essas aplicações atendam a esses requisitos, fornecendo uma execução rápida e responsiva.

Pré-requisitos

Para estudar Programação de Alto Desempenho, é importante ter uma base sólida em conceitos fundamentais de Ciência da Computação, tais como:

- Linguagens de Programação
- Algoritmos e Estruturas de Dados
- Arquitetura de Computadores
- Sistemas Operacionais
- Matemática e Cálculo

Tecnologias e recursos de Programação de Alto Desempenho

A Programação de Alto Desempenho é voltada para otimizar o desempenho e a eficiência de programas e sistemas computacionais, especialmente em cenários que envolvem o processamento intensivo de dados.

Nesse domínio, costuma-se utilizar linguagens de programação de baixo nível, como C e C++, que oferecem um controle mais direto sobre a manipulação de memória e a execução do código. Outras linguagens de programação utilizadas no contexto da Programação de Alto Desempenho são Python, Mojo e Rust.

Além das linguagens, bibliotecas especializadas são importantes na Programação de Alto Desempenho. A OpenMP (*Open Multi-Processing*) é uma biblioteca que possibilita a programação paralela em arquiteturas *multicore*, permitindo que tarefas sejam divididas entre vários núcleos de CPU para acelerar a execução. A biblioteca MPI (*Message Passing Interface*) é amplamente usada para a comunicação entre processos em sistemas distribuídos. CUDA (*Compute Unified Device Architecture*), da NVIDIA, são utilizados no desenvolvimento de código paralelo para GPUs. NVIDIA, aproveitando sua capacidade de processamento paralelo. Isso é particularmente útil em aplicações que envolvem simulações científicas, aprendizagem de máquina e computação gráfica. BLAS (*Basic Linear Algebra Subprograms*, ou *Subprogramas Básicos de Álgebra Linear*, em português), é uma biblioteca que contém operações otimizadas de álgebra linear, aproveitando as características do hardware subjacente, como processadores *multicore*.

Os *frameworks* Apache Hadoop, Apache Spark e TensorFlow são frequentemente utilizados para Programação de Alto Desempenho, principalmente em problemas de Computação Distribuída e de Aprendizagem de Máquina. O Apache Hadoop oferece uma plataforma escalável para processar grandes volumes de dados distribuídos em clusters de computadores, utilizando o modelo MapReduce [2]. O Apache Spark proporciona um processamento em memória mais veloz, permitindo análises complexas e em tempo real. O TensorFlow é uma biblioteca de Aprendizagem de Máquina que capacita a criação eficiente de modelos de redes neurais e aprendizagem profunda. Esses *frameworks* estão intrinsecamente ligados à Programação Paralela e de Alto Desempenho, uma vez que tiram proveito da distribuição de tarefas em múltiplos processadores ou nós de computação para

acelerar o processamento e alcançar resultados mais eficientes em diversas aplicações, desde análises de *big data* até treinamento de modelos de inteligência artificial.

Introdução à Programação de Alto Desempenho

A Programação de Alto Desempenho envolve a aplicação de técnicas e estratégias para melhorar o desempenho de programas, visando utilizar eficientemente os recursos computacionais disponíveis. Essa abordagem é especialmente relevante em grandes computadores, como clusters, que são sistemas compostos por múltiplos nós de processamento interconectados.

Clusters são amplamente utilizados em uma variedade de aplicações que exigem alta capacidade de processamento e armazenamento de dados, tais como:

- **Análise de *Big Data***: Clusters são frequentemente utilizados para processar e analisar grandes conjuntos de dados, como dados de redes sociais, registros de atividades de servidores, dados climáticos, informações genômicas e muitos outros. Através da distribuição de tarefas em diferentes nós do cluster, é possível acelerar o processamento e realizar análises complexas em grandes volumes de dados.
- **Simulações Científicas**: Áreas como física, química, biologia, engenharia e outras ciências se beneficiam do uso de clusters para executar simulações computacionais complexas. Essas simulações podem envolver cálculos intensivos, como modelagem molecular, dinâmica de fluidos, simulações de climas e previsões meteorológicas, otimização de designs, entre outros.
- **Aprendizagem de Máquina e Inteligência Artificial**: O treinamento de modelos de aprendizado de máquina e algoritmos de inteligência artificial geralmente requer um grande poder de processamento. Clusters são usados para acelerar o treinamento de modelos complexos, permitindo que algoritmos explorem grandes quantidades de dados para aprender padrões e fazer previsões mais precisas.
- **Renderização de Imagens e Animações**: Indústrias de entretenimento, como filmes, animações e jogos, fazem uso de clusters para renderizar imagens e gráficos computacionalmente intensivos. Ao distribuir a carga de trabalho entre os nós do cluster, é possível acelerar a renderização e reduzir o tempo necessário para gerar sequências de imagens ou animações complexas.
- **Computação Distribuída**: Clusters são fundamentais para a construção de sistemas de computação distribuída, onde múltiplos nós trabalham em conjunto para

realizar tarefas paralelas e distribuídas. Isso inclui, entre outros, sistemas de processamento distribuído, sistemas de arquivos distribuídos e bancos de dados distribuídos.

Em clusters, a Programação de Alto Desempenho é utilizada para aproveitar o poder de processamento distribuído dos nós. Os programas são projetados para serem executados em paralelo em diferentes nós do cluster, dividindo as tarefas computacionais entre eles. Essa abordagem permite que grandes volumes de dados sejam processados de forma eficiente e rápida, além de lidar com problemas computacionalmente intensivos.

No cerne da Programação de Alto Desempenho está a Programação Concorrente, uma vez que permitir a execução simultânea de tarefas independentes dentro de um programa é uma estratégia válida para melhorar o desempenho de programas. Ao utilizar *threads* ou processos concorrentes, é possível distribuir e balancear a carga de trabalho em múltiplos núcleos de processamento e processadores. Os principais conceitos da Programação Concorrente estão citados na sequência.

Conceitos sobre Programação Concorrente

Segundo o dicionário [3], concorrência é

1. ato ou efeito de concorrer
2. competição
3. disputa

Trazendo este conceito para a Computação, a execução concorrente de um programa acontece se partes desse programa são executadas apenas conceitualmente em paralelo, mas competindo por recursos físicos [4].

Programação Concorrente refere-se à execução simultânea de tarefas independentes em um programa. Ela envolve a criação e o gerenciamento de *threads*, que são fluxos de controle de execução separados dentro de um programa. O objetivo da Programação Concorrente é melhorar a eficiência e o desempenho do programa, aproveitando os recursos disponíveis, que podem ser (ou não) processadores múltiplos.

É importante enfatizar que a Programação Concorrente não depende de processadores múltiplos. Ela é útil quando há tarefas que podem ser executadas de forma independente,

independentemente de estarem em um único processador ou em vários processadores. Isso faz da Programação Concorrente diferente da Programação Paralela, e é importante conceituar ambas nesta primeira aula.

A Programação Paralela refere-se à execução simultânea de partes independentes de um programa em diferentes processadores ou núcleos de processamento, com o objetivo de acelerar a execução e lidar com problemas computacionalmente intensivos. A Programação Paralela é uma forma específica de Programação Concorrente, onde diferentes partes do programa são executadas em paralelo em hardware separado.

Embora os termos *Programação Concorrente* e *Programação Paralela* sejam frequentemente usados como sinônimos, é importante destacar a diferença — sutil — entre eles. A Programação Concorrente pode envolver a execução de tarefas independentes dentro de um único processador, enquanto a Programação Paralela envolve a execução simultânea de partes independentes do programa em múltiplos processadores ou núcleos. A Figura 1 apresenta uma analogia que facilita a compreensão acerca da diferença desses dois conceitos.

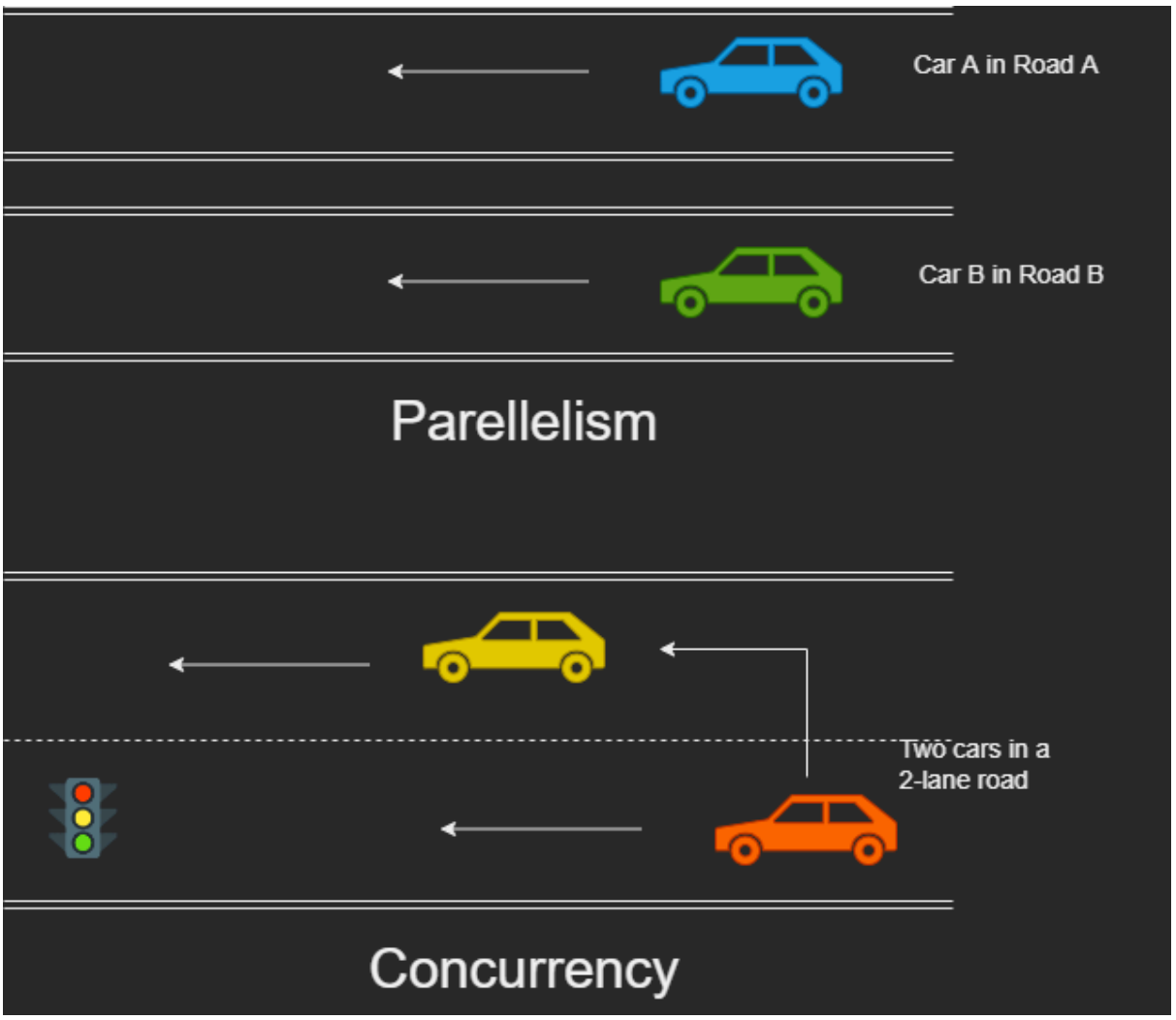

Figura 1. Concorrencia-Paralelismo.png [5]. Disponível em https://drive.google.com/file/d/1-LgjzZ_5yirfYxWW6Q5sGKfkwNcxPGJJ/view?usp=sharing.

A execução de um programa ocorre de forma paralela se seus comandos são executados em paralelo fisicamente, e ocorre de forma concorrente se seus comandos são executados conceitualmente em paralelo [4].

As duas abordagens têm como objetivo melhorar o desempenho e a eficiência de programas, mas a Programação Paralela é especialmente adequada para lidar com problemas que podem ser divididos em partes independentes que podem ser executadas em paralelo.

O estudo da Programação Concorrente envolve a compreensão de alguns conceitos:

- **Processo**: É uma instância de um programa em execução, representando uma entidade ativa que tem espaço de endereçamento, recursos e fluxo de controle próprios. Um processo pode conter uma ou mais *threads*, além de dados, código

executável, arquivos. Cada processo é isolado e protegido de outros processos, garantindo que eles não acessem diretamente os recursos uns dos outros. Os processos são gerenciados pelo sistema operacional. Se um processo tiver várias *threads* de controle, ele poderá executar mais de uma tarefa por vez. A Figura 2 ilustra a diferença entre um processo tradicional de uma *thread* e um processo com múltiplas *threads* [6].

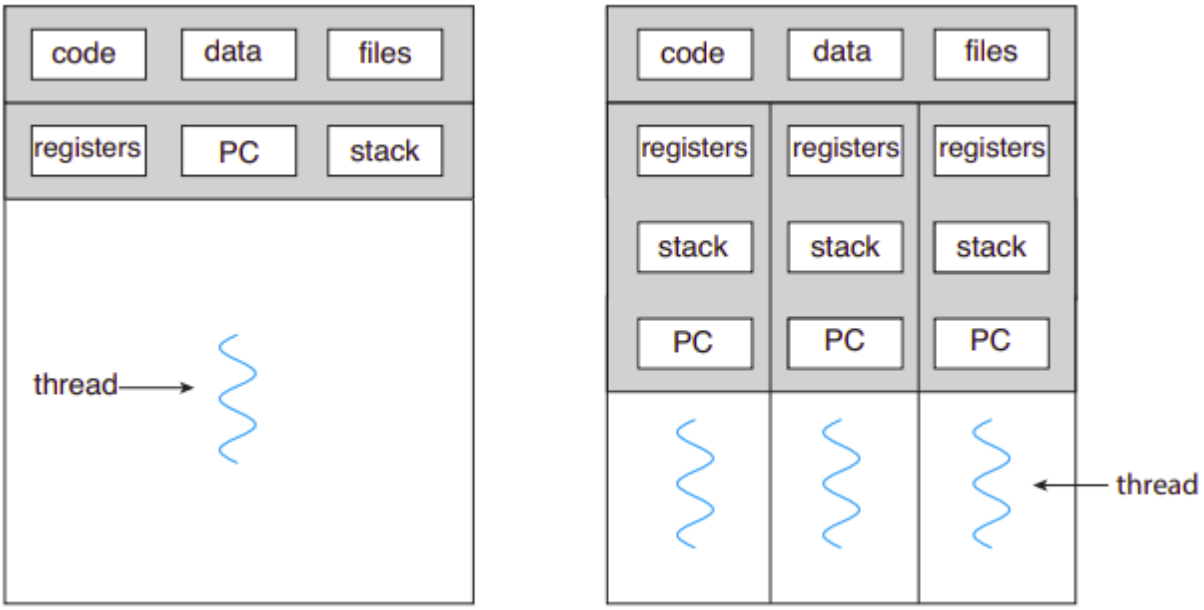

Figura 2. Thread-single-multi.png [6]. Disponível em https://drive.google.com/file/d/1QbjO3Oi5UNC-NtqG3OT6lrgwlB7VKB_R/view?usp=sharing.

- ***Thread***: Uma *thread* é uma sequência de instruções que pode ser executada de forma independente dentro de um programa. De acordo com [6], *thread* é uma unidade básica de uso da CPU, e contém um ID de *thread*, um contador de programa (PC), um conjunto de registradores e uma pilha. Ela representa um fluxo de controle de execução separado que pode ser executado concorrentemente com outras *threads*. As *threads* podem compartilhar recursos e dados dentro do programa e podem ser usadas para realizar tarefas paralelas. *Threads* são também a maneira mais fácil de aproveitar o poder de computação dos sistemas multiprocessadores, e à medida que o número de processadores aumenta, a exploração efetiva da simultaneidade se tornará cada vez mais importante [6].
- **Sincronização**: A sincronização é um mecanismo necessário para garantir que as *threads* cooperem corretamente e evitem problemas como condições de corrida, onde duas ou mais *threads* acessam e modificam os mesmos dados simultaneamente. Mecanismos de sincronização, como mutexes, semáforos e

barreiras, são usados para controlar o acesso compartilhado a recursos e garantir a consistência dos dados.

- **Comunicação**: A comunicação é fundamental para permitir a troca de informações entre *threads* concorrentes. Existem diferentes mecanismos de comunicação, como variáveis compartilhadas, filas de mensagens, canais de comunicação, entre outros. Esses mecanismos garantem que as *threads* possam cooperar e trocar informações de forma segura e coordenada.
- **Atomicidade**: Propriedade de uma operação ser executada de forma indivisível, ou seja, como uma única unidade atômica. Isso significa que a operação é executada sem ser interrompida por outras *threads* concorrentes, garantindo que ela seja concluída completamente antes que qualquer outra *thread* acesse ou modifique os mesmos recursos. A atomicidade é importante para evitar problemas de consistência de dados, como condições de corrida. Se uma operação não for executada atomicamente, pode ocorrer uma situação em que uma *thread* lê um valor intermediário modificado por outra *thread*, levando a resultados inconsistentes ou indesejados. Ela é alcançada através de algum mecanismo de sincronização.
- **Condição de corrida (*race condition*)**: Problema que ocorre na Programação Concorrente quando o resultado da execução de um programa depende da ordem em que as operações são executadas por múltiplas *threads* concorrentes. Em uma race condition, duas ou mais *threads* acessam e manipulam um recurso compartilhado simultaneamente, resultando em um comportamento indefinido ou incorreto do programa. Isso acontece quando as *threads* não estão sincronizadas adequadamente e não coordenam seus acessos ao recurso compartilhado. Ela acaba ocorrendo porque o resultado final depende da maneira como as instruções são intercaladas entre as *threads*, e essa intercalação pode variar a cada execução do programa. Portanto, o comportamento do programa pode ser imprevisível e inconsistente, o que não é desejável. A Figura 3 exibe um exemplo do que seria uma condição de corrida em um caso hipotético em que duas pessoas tentam depositar uma unidade monetária em uma mesma conta bancária.

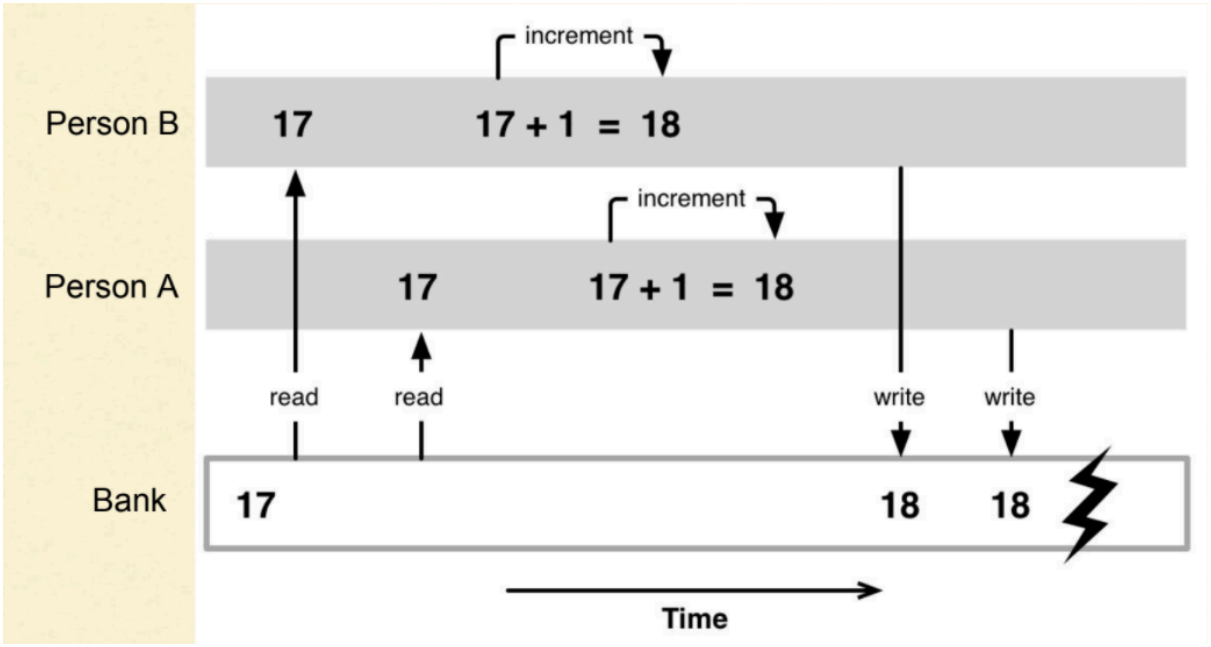

Figura 3. Race-Condition.png [7]. Disponível em https://drive.google.com/file/d/1O8xbXHOz388-Xp6mdxyFL66eEcHkVGGx/view?usp=sharing.

- **Multiprocessamento**: Refere-se à capacidade de um sistema ter mais de um processador ou núcleo de processamento para executar tarefas simultaneamente. Com o multiprocessamento, é possível realizar a execução paralela de múltiplos processos ou *threads* em diferentes processadores ou núcleos, aumentando a capacidade de processamento e melhorando o desempenho do sistema.

Ao projetar e implementar programas concorrentes, o programador seguramente precisará conhecer todos os conceitos acima. Eles serão mais detalhados no decorrer da disciplina.

Exemplos de uso

- **Sistemas Operacionais**: os sistemas operacionais fazem uso extensivo da Programação Concorrente para gerenciar e executar tarefas simultâneas. Por exemplo, o escalonador de processos e *threads* é responsável por decidir quais tarefas devem ser executadas e quando. A Programação Concorrente é essencial para lidar com a execução simultânea de múltiplos processos, gerenciando o acesso aos recursos compartilhados e garantindo a sincronização adequada.
- **Servidores *web***: os servidores *web* são um exemplo comum de aplicação que utiliza a Programação Concorrente para lidar com várias solicitações simultâneas de clientes. Cada solicitação é tratada como uma tarefa concorrente, onde *threads* são

usadas para processar as requisições e fornecer as respostas aos clientes. A Programação Concorrente permite que o servidor *web* atenda a vários clientes simultaneamente, melhorando o desempenho e a capacidade de resposta.

- **Aplicações de processamento de imagens e vídeos**: muitas aplicações que envolvem o processamento de imagens e vídeos fazem uso da Programação Concorrente para melhorar o desempenho. Por exemplo, algoritmos de processamento de imagens podem ser paralelizados para acelerar a aplicação de filtros, detecção de bordas ou reconhecimento de padrões em imagens. Da mesma forma, a decodificação e a reprodução de vídeos podem ser tratadas de forma concorrente para garantir uma exibição suave e sem interrupções.
- **Computação científica**: a Programação Concorrente é utilizada na computação científica para resolver problemas complexos que exigem uma grande quantidade de cálculos. Por exemplo, simulações numéricas, modelagem computacional e análise de dados em larga escala são áreas que se beneficiam da Programação Concorrente. Ao distribuir as tarefas em diferentes *threads* ou processos, é possível acelerar o processamento e obter resultados mais rapidamente.
- **Jogos e aplicações interativas**: jogos e outras aplicações interativas podem se beneficiar da Programação Concorrente para lidar com múltiplas tarefas simultâneas, como animações, física do jogo, inteligência artificial de personagens e processamento de entrada do usuário. A Programação Concorrente permite que essas tarefas sejam executadas em paralelo, proporcionando uma experiência mais fluida e responsiva para o usuário.

Resumo

Neste texto, abordamos uma introdução aos conceitos de Programação de Alto Desempenho e Programação Concorrente, as motivações para estudar este tópico, os pré-requisitos necessários para sua compreensão, as tecnologias envolvidas e exemplos práticos em que esse tipo de programação é adequado.

Os conceitos apresentados são essenciais para o trabalho do programador de alto desempenho. Os processos são entidades fundamentais na Programação Concorrente, representando instâncias em execução de um programa. Eles podem conter uma ou mais *threads*, que são fluxos de execução independentes. O conhecimento sobre *threads* permitirá criar blocos de produção independentes, permitindo o trabalho concorrente. Os mecanismos de comunicação precisarão ser estudados para garantir que as *threads*

possam trocar informações de forma consistente e segura. Da mesma forma, conhecer as diferentes abordagens que garantem a sincronização das *threads* ao acessarem recursos compartilhados será fundamental para evitar resultados anômalos e erros de condição de corrida. É importante também considerar a atomicidade das operações, ou seja, a garantia de que uma operação seja executada de forma indivisível, sem interrupções de outras *threads*. Além disso, o conceito de multiprocessamento é relevante para entender como os programas são executados em ambientes com múltiplos processadores ou núcleos. A Programação Concorrente permite aproveitar o multiprocessamento, criando *threads* e/ou processos que podem ser executados paralelamente em diferentes processadores ou núcleos, visando melhorar o desempenho e a eficiência do programa.

Portanto, a Programação Concorrente envolve a consideração cuidadosa dos conceitos abordados a fim de garantir corretude, eficiência e segurança em programas executados em ambientes com múltiplas *threads* concorrentes.

Exercícios

1) Qual é a diferença fundamental entre Programação Concorrente e Programação Paralela?

a) A Programação Concorrente envolve a execução simultânea de tarefas independentes em um programa, enquanto a Programação Paralela refere-se à execução simultânea de partes independentes de um programa em diferentes processadores ou núcleos.

b) A Programação Concorrente e a Programação Paralela são termos diferentes para se referir à mesma abordagem de programação.

c) A Programação Concorrente refere-se à execução sequencial de tarefas em um único processador, enquanto a Programação Paralela envolve a execução simultânea de tarefas independentes.

d) A Programação Concorrente envolve a execução simultânea de tarefas dependentes em um programa, enquanto a Programação Paralela refere-se à execução de tarefas independentes.

2) O que é uma *race condition* na Programação Concorrente?

a) Uma situação em que duas ou mais *threads* tentam acessar e modificar um recurso compartilhado simultaneamente, resultando em um comportamento indefinido ou incorreto do programa.

b) Uma condição em que uma única *thread* executa várias tarefas independentes de forma concorrente.
c) Um problema que ocorre quando *threads* estão sincronizadas adequadamente e coordenam seus acessos ao recurso compartilhado.
d) Uma propriedade que garante que as operações executadas por várias *threads* sejam concluídas de forma indivisível.

3) Na Programação Sequencial, as tarefas são executadas de forma linear, uma após a outra. Em comparação à Programação Concorrente, em que situações essa abordagem é
   3.1) vantajosa
   3.2) desvantajosa

4) Leia o capítulo 3[1] de [6] para resolver esta questão. Sobre a arquitetura de multiprocessamento do navegador *web* Google Chrome.
   4.1) Resuma como funciona esta arquitetura.
   4.2) Quais são os benefícios dessa arquitetura?

5) Em conjunto, aprimore o resumo do capítulo 3 (*Processos*) até a seção 3.3 (*Operações em Processos*) de [6]. Escreva o resumo colaborativamente.

Gabarito

1) a
2) a
3)

3.1) **Tarefas com dependências sequenciais**: Se as tarefas do programa têm dependências estritas entre si, ou seja, uma tarefa precisa ser concluída antes que outra possa iniciar, a programação concorrente pode introduzir complexidade desnecessária. Nesses casos, a programação sequencial é mais simples e direta, já que as tarefas são executadas em uma ordem determinística.
Problemas com baixa granularidade: Se as tarefas individuais do programa são muito pequenas ou possuem pouca computação, a sobrecarga introduzida pela programação concorrente, como a criação e a sincronização de *threads* ou processos, pode piorar o desempenho da aplicação em termos de tempo e recursos utilizados.

[1] Na 10ª edição o trecho em questão encontra-se na página 125.

Restrições de recursos: Em sistemas com recursos limitados, como dispositivos embarcados com baixo poder de processamento, a programação concorrente pode consumir muitos recursos adicionais, como memória e capacidade de processamento. Nesses cenários, a programação sequencial pode ser preferível para evitar a sobrecarga e garantir a eficiência dos recursos disponíveis.

3.2) **Processamento de grandes volumes de dados**: Em aplicações que lidam com análise de *big data*, como processamento de dados em tempo real, mineração de dados ou aprendizado de máquina, a programação concorrente permite distribuir o processamento em várias *threads* ou processos, acelerando o tempo de execução e possibilitando o processamento paralelo de diferentes partes dos dados.

Simulações e modelagem computacional: Em áreas como física, química, biologia e engenharia, onde são realizadas simulações complexas e modelagem computacional avançada, a programação concorrente permite dividir a carga de trabalho entre várias *threads* ou processos, agilizando a obtenção dos resultados e possibilitando a execução de simulações mais detalhadas e precisas.

4)

4.1) A arquitetura de multiprocessamento do Google Chrome é baseada em um modelo chamado *Processo por Site* ou *Processo por Origem*. Resumidamente, a arquitetura funciona da seguinte maneira:

- Cada aba aberta no Google Chrome é executada em seu próprio processo separado, isolado dos outros processos. Isso significa que cada site ou aplicação *web* carregada em uma aba tem seu próprio processo dedicado.
- O processo principal do Google Chrome é responsável por gerenciar a interface do usuário, a barra de endereços, os favoritos e outras funcionalidades do navegador. Ele também coordena a comunicação entre os processos de cada aba.
- Cada processo de aba é responsável por carregar e renderizar o conteúdo do site ou aplicação *web* específica. Isso inclui a interpretação do HTML, CSS e JavaScript e a renderização de elementos visuais.
- Além dos processos de renderização, também existem outros processos especializados, como o Processo da GPU, que cuida do processamento gráfico acelerado por hardware, e o Processo de *Plugin*, que lida com a execução de *plugins* como o Flash.

4.2) Essa arquitetura traz os seguintes benefícios:

- **Isolamento**: cada aba tem seu próprio processo separado, então um problema em um site ou aplicação *web* não afeta as outras abas ou o navegador como um todo. Isso aumenta a estabilidade e a segurança do navegador.

- **Estabilidade**: a arquitetura é capaz de lidar com exceções e erros, como falhas em abas específicas, não afetando as demais abas.
- **Desempenho**: O uso de múltiplos processos permite que o Google Chrome aproveite melhor o hardware disponível, distribuindo a carga de trabalho em vários núcleos de processamento ou processadores, quando eles estiverem disponíveis.
- **Gerenciamento de Memória**: Cada processo de aba tem seu próprio espaço de memória, permitindo um gerenciamento mais eficiente. Se uma aba consome muita memória ou apresenta um vazamento de memória, apenas o processo daquela aba é afetado, não afetando as outras abas ou o navegador como um todo.

5) Livre.

Referências

[1] STERLING, T.; ANDERSON, M.; BRODOWICZ, M.; GROPP, W.. High Performance Computing: Modern Systems and Practices. Morgan Kaufmann: 2020.

[2] DEAN, J. & GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. USENIX Symposium on Operating Systems Design and Implementation, 2004.

[3] OXFORD. Dicionário. Conceito de concorrência.

[4] LEE, Edward A. & SESHIA, Sanjit A. Introduction to Embedded Systems. A Cyber-Physical Systems Approach. Second Edition, MIT Press, ISBN 978-0-262-53381-2, 2017.

[5] FIGURA 1 (Concorrencia-Paralelismo.jpg), originalmente com a legenda "Illustration showing difference between parallelism and concurrency". Disponível em: https://www.freecodecamp.org/news/concurrent-programming-in-go/. Acesso em: 11 jul. 2023.

[6] SILBERSCHATZ, A., GALVIN, P. B., & GAGNE, G. (2018). Operating System Concepts (10ª ed.). Wiley.

[7] FIGURA 3 (Race-Condition.png), originalmente com a legenda "Race Condition". Disponível em: https://cloudxlab.com/blog/race-condition-and-deadlock/. Acesso em: 12 jul. 2023.

[8] GOETZ, B., PEIERLS, T., BLOCH, J., BOWBEER, J., HOLMES, D., & LEA, D. (2006). Java Concurrency in Practice. Addison-Wesley Professional.