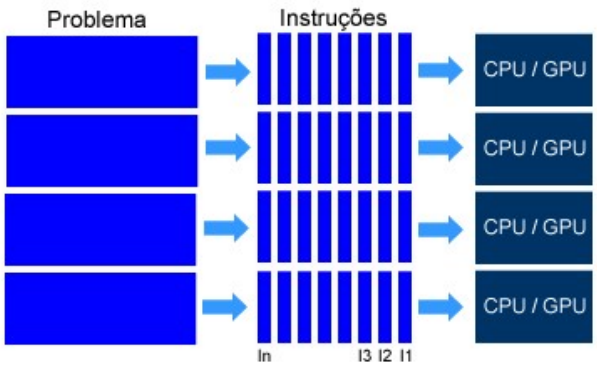
Machine Learning e IA em Ambientes Distribuídos Versão 2.0

Como Funciona o Paralelismo?

A programação paralela sempre foi uma área de grande importância na computação, principalmente na computação de alto desempenho. No entanto, mais recentemente, com o avanço do *hardware* para arquiteturas *multi-core* ou *many-core* ela tornou-se fundamental para quase todos os tipos de aplicações. O surgimento dos processadores com várias unidades de computação em um único *chip* fez com que simples computadores pessoais se tornassem potenciais sistemas em paralelo. Porém, como ocorre na maioria das vezes, o avanço tecnológico traz consigo um grande desafio: reestruturar os sistemas existentes de forma a aproveitar os recursos adicionais de computação. Para tirar proveito do poder computacional e buscar a aceleração de uma aplicação é necessário um esforço adicional a nível de *software.*

Através de linguagens de programação ou APIs com recursos paralelos é possível dividir um problema em rotinas menores e executá-las de forma segura simultaneamente. A figura abaixo ilustra um problema sendo dividido em instruções menores para ser executado por uma unidade processadora, como por exemplo CPU ou GPU.



Uma vez que o problema é dividido em rotinas menores, essas são atribuídas a processos ou *threads* que são enviados às unidades de processamento. *Scheduling* ou agendamento é o termo dado à atribuição das tarefas aos processos ou *threads*, onde também é fixada a ordem de execução das tarefas. O *scheduling* pode ser realizado de três maneiras: (i) explícito no código fonte; (ii) através do ambiente de programação em tempo de compilação ou (iii) dinamicamente em tempo de execução. Depois disso, os processos ou *threads* são

atribuídos às unidades de processamento, procedimento conhecido como mapeamento.

As tarefas de um algoritmo podem ser independentes uma das outras, ou seja, uma tarefa não depende do resultado de outra tarefa. Porém, há casos em que uma tarefa é dependente do resultado de outra tarefa. Quando isso ocorre é necessário que uma tarefa espere pela conclusão da outra tarefa. Como os programas paralelos geram tarefas que concorrem por recursos há a necessidade da coordenação e da sincronização entre os processos ou *threads*, a fim de executar corretamente. Mesmo que um programa paralelo apresente o resultado de forma correta, ele pode deixar de realizar o melhor desempenho se não explorar a concorrência. Cuidados devem ser tomados para garantir que a sobrecarga gerada pela gestão da concorrência não interfira no tempo de execução do programa. No processamento paralelo os métodos de sincronização e coordenação estão fortemente relacionados com a forma como a informação é trocada entre processos ou *threads*.

Em computadores com memória distribuída os dados não podem ser acessados por todos os processadores. Para ocorrer a comunicação entre os processos é necessária a troca (envio e recebimento) de mensagens. Nos dispositivos com memória compartilhada os dados podem ser acessados por todos os processadores ou núcleos. Nessa situação, todas as threads fazem acesso a mesma memória através de rotinas de leitura e escrita. A sincronização entre threads possibilita que o trabalho seja coordenado, de forma que uma thread não leia um dado antes que outra thread encerre a gravação na mesma área de memória. Especificar áreas de barrier (barreiras) também é uma forma de realizar o sincronismo entre processos ou threads. Somente depois que todos os processos ou threads tenham executado o código antes da barreira de sincronização, eles conseguem continuar a execução após a barreira.

O paralelismo por estar intimamente ligado a técnicas de *hardware* e *software* pode ser implementado em diferentes níveis:

a) Nível de dados: O *Data-level Parallelism* (DLP) opera simultaneamente em

múltiplos dados, como por exemplo, adição e multiplicação de números binários e processamento vetorial;

b) Nível de instrução: Compreende a execução simultânea de mais de uma instrução por parte do processador. O *pipelining* é um exemplo do paralelismo em nível de instrução. Também chamado de *Instruction-level parallelism* (ILP);

c) Nível de *thread*: Mais conhecido por *Thread-level parallelism* (TLP). Nesse nível várias *threads* são executadas simultaneamente em um ou mais processadores compartilhando recursos computacionais;

d) Nível de processo: No *Process-Level Parallelism* (PLP) os processos são executados em um ou mais computadores, onde cada processo tem seus próprios recursos computacionais, como por exemplo, memória e registradores.

Referência:

Programming Massively Parallel Processors: A Hands-on Approach
https://www.amazon.com.br/Programming-Massively-Parallel-Processors-Hands-ebook/dp/B01NCENHQQ/ref=sr_1_1?ie=UTF8&qid=1496248368&sr=8-1&keywords=parallel+programming