dev.to

TAXONOMIA DE FLYNN: Computação Paralela

Isaac Alves Pinheiro

100–137 minutos

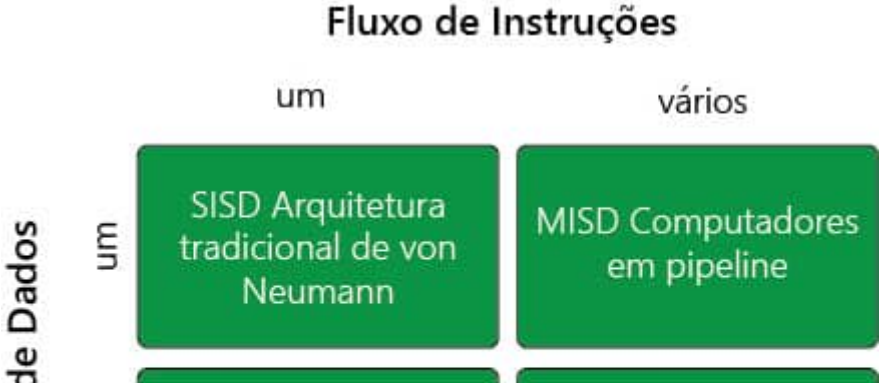
A **computação paralela** (Parallel Computing) é uma computação em que os trabalhos são divididos em partes discretas que podem ser executadas simultaneamente. Cada parte é subdividida em uma série de instruções. As instruções de cada parte são executadas simultaneamente em CPUs diferentes. Os **sistemas paralelos** lidam com o uso simultâneo de vários recursos de computador que podem incluir um único computador com vários processadores, vários computadores conectados por uma rede para formar um **cluster de processamento paralelo** ou **uma combinação de ambos**.

> Os **sistemas paralelos** são mais difíceis de programar do que os

> computadores com um único processador, porque a arquitetura dos computadores paralelos varia de acordo com os recursos disponíveis, sendo assim, os processos de várias CPUs devem ser coordenados e sincronizados.

A primeira descrição formal desse tipo de abordagem foi a **taxonomia de Flynn**. Essa taxonomia foi desenvolvida em 1966 e ligeiramente expandida em 1972:

- É uma metodologia para classificar formas gerais de operação paralela disponíveis em um processador.
- Propõe uma abordagem para esclarecer os tipos de paralelismo suportados no hardware por um sistema de processamento ou disponíveis em uma aplicação.
- Sua classificação é baseada na visão da máquina ou do aplicativo pelo programador de linguagem de máquina (machine code).



A taxonomia de Flynn é uma categorização de formas de arquiteturas de computador paralelas.

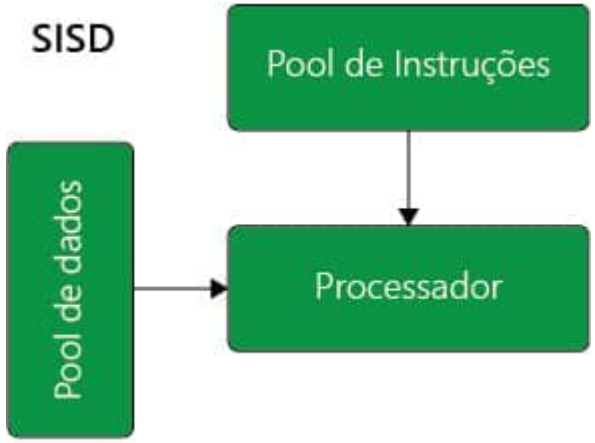
Do ponto de vista do programador de linguagem assembly (linguagem de montagem), os **computadores paralelos** são classificados pela simultaneidade em **sequências de processamento** (ou **fluxos**), **dados** e **instruções**. Isso resulta em quatro classes: **SISD** (instrução única, dados únicos), **SIMD** (instrução única, dados múltiplos), **MISD** (instrução múltipla, dados únicos) e **MIMD** (instrução múltipla, dados múltiplos).

A taxonomia de Flynn pode ser sumarizada de forma ilustrativa segundo a imagem a seguir.

SISTEMAS DE INSTRUÇÃO ÚNICA E DADOS ÚNICOS (SINGLE-INSTRUCTION, SINGLE-DATA – SISD)

Um **sistema de computação de instrução única e dados únicos** (**Single-Instruction**, **Single-Data – SISD**) é uma máquina de **um**

processador que é capaz de executar **uma única instrução**, **operando em um único fluxo de dados**, como se observa na imagem seguinte.

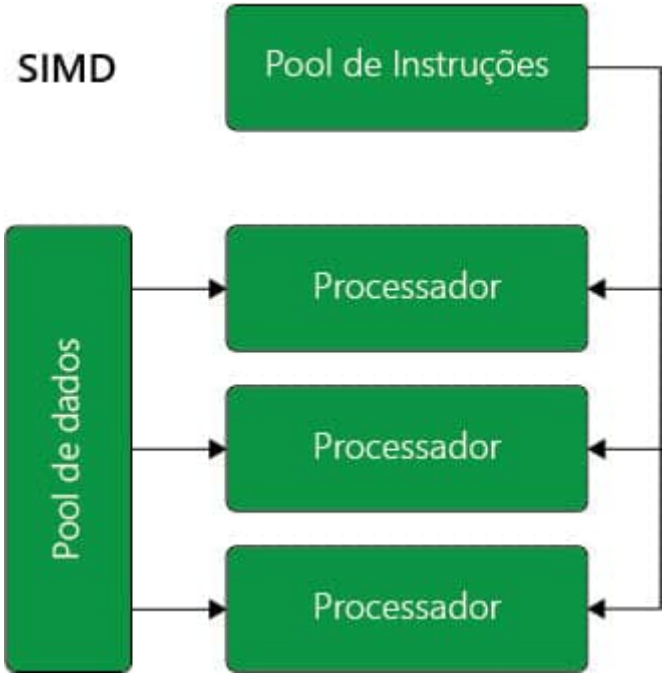


No **SISD**, as instruções de máquina são **processadas de maneira sequencial**, e os computadores que adotam esse modelo são popularmente chamados de **computadores sequenciais**. A maioria dos computadores convencionais derivados da proposição de Von Neumann possui arquitetura SISD. Todas as instruções e dados a serem processados devem ser armazenados na memória primária.

A velocidade do elemento de processamento no modelo SISD é limitada (dependente) pela taxa por meio da qual o computador pode transferir informações internamente. Os sistemas SISD

representativos dominantes são **IBM PC** e **estações de trabalho**, entre outros.

SISTEMAS DE INSTRUÇÃO ÚNICA E DADOS MÚLTIPLOS (SINGLE-INSTRUCTION, MULTIPLE-DATA – SIMD)



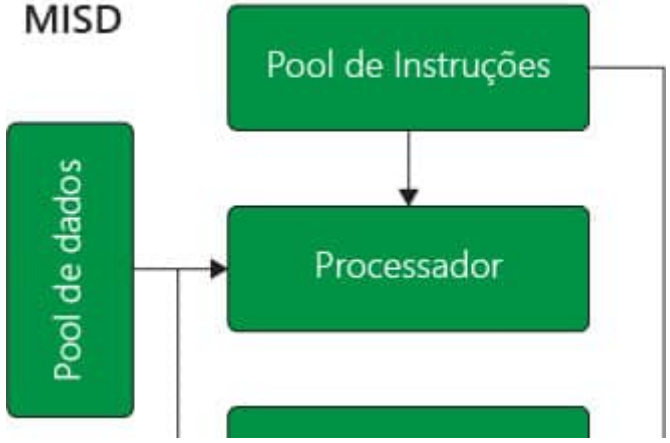
Um **sistema SIMD** é uma **máquina multiprocessada** capaz de executar a **mesma instrução em todas as CPUs**, mas **operando em diferentes fluxos de dados**.

As máquinas baseadas em um modelo SIMD são adequadas para

computação científica, pois envolvem muitas operações de vetor e matriz. Para que a informação possa ser passada para todos os elementos de processamento (Processing Elements – PEs), os elementos de dados organizados dos vetores podem ser divididos em vários conjuntos (N-conjuntos para sistemas N PE) e cada PE pode processar um conjunto de dados. Veja a imagem a seguir.

Os sistemas SIMD representativos são, por exemplo, **máquinas de processamento vetorial da Cray** e as **Unidades de Processamento Gráfico**, as famosas **placas de vídeo** (Graphical Processing Unit – GPU).

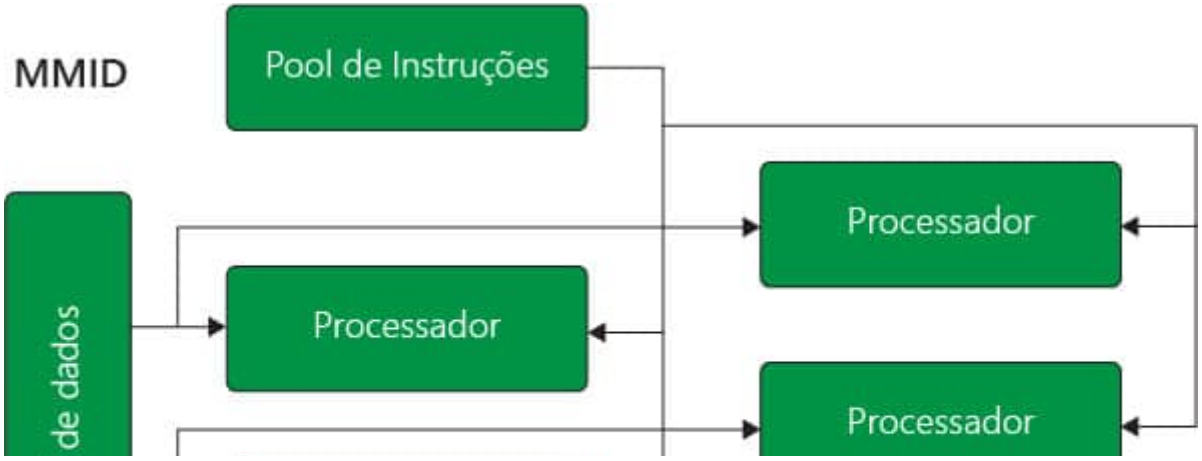
SISTEMAS DE INSTRUÇÃO MÚLTIPLA E DADOS ÚNICOS (MULTIPLE-INSTRUCTION, SINGLE-DATA – MISD)

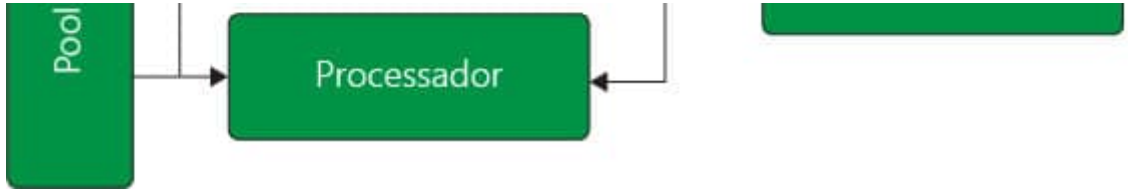


Um **sistema de computação MISD** é uma **máquina multiprocessada** capaz de executar diferentes instruções em diferentes PEs, mas todas operando no mesmo conjunto de dados, conforme a próxima imagem.

O sistema executa diferentes operações no mesmo conjunto de dados. As máquinas construídas usando o modelo MISD não são úteis na maioria das aplicações, algumas máquinas são construídas, mas nenhuma delas está disponível comercialmente.

SISTEMAS DE MÚLTIPLAS INSTRUÇÕES E MÚLTIPLOS DADOS (MULTIPLE-INSTRUCTION, MULTIPLE-DATA – MIMD)





Um **sistema de múltiplas instruções e múltiplos dados** (**MIMD**) é **uma máquina com multiprocessador** capaz de executar **várias instruções em vários conjuntos de dados**, conforme a imagem a seguir.

Cada PE no modelo MIMD tem instruções e fluxos de dados separados, portanto, as máquinas construídas a partir desse modelo suportam qualquer tipo de aplicação. Ao contrário das máquinas SIMD e MISD, os PEs em máquinas MIMD funcionam de forma assíncrona.

> **Atenção**: As **máquinas MIMD** são amplamente categorizadas em MIMD de memória compartilhada e MIMD de memória distribuída com base na maneira como os PEs são acoplados à memória principal.

No modelo MIMD de **memória compartilhada** (sistemas multiprocessadores **fortemente** acoplados), todos os PEs são conectados a uma única memória global e todos têm acesso a ela. A comunicação entre os PEs nesse modelo ocorre por meio da

memória compartilhada, e a modificação dos dados armazenados na memória global por um PE é visível para todos os outros PEs. Os sistemas MIMD representativos dominantes de memória compartilhada são as **máquinas Silicon Graphics** e **SMP** (Symmetric Multi-Processing) da **Sun** / **IBM**.

Em máquinas MIMD de **memória distribuída** (sistemas multiprocessadores **fracamente** acoplados), todos os PEs têm uma memória local. A comunicação entre PEs nesse modelo ocorre por meio da **rede de interconexão** (o canal de comunicação entre processos, ou **IPC**). A rede que conecta os PEs pode ser configurada em **árvore**, **malha** ou de acordo com o requisito.

> **Árvore** (Tree): A topologia de rede em árvore é uma topologia que possibilita a visualização da interligação de várias redes e sub-redes, caracterizando-se pela presença de um concentrador que interliga todos os **nodos** (nós - nodes) (computadores, servidores, PEs etc.) de uma rede local, enquanto outro concentrador interliga as demais redes, interligando dessa forma um conjunto de redes locais (LAN) e fazendo com que estas sejam dispostas no formato de árvore.

> **Malha** (Mesh): A topologia de rede em malha consiste em uma topologia de rede de computadores onde cada nodo da rede (computador, servidor, PE etc.) está conectado aos demais diretamente, o que possibilita que todos os nodos da rede sejam capazes de trocar informações diretamente com todos os demais. Nessa topologia, a informação pode ser transmitida da origem ao destino por diversos meios ou caminhos.

A arquitetura MIMD de memória compartilhada é mais fácil de programar, mas é menos tolerante a falhas e mais difícil de estender em relação ao modelo MIMD de memória distribuída.

> Falhas em um **MIMD de memória compartilhada** afetam todo o sistema, ao passo que este não é o caso do modelo distribuído, no qual cada um dos PEs pode ser facilmente isolado.

Além disso, as arquiteturas MIMD de memória compartilhada têm menos probabilidade de serem escaláveis porque a adição de mais PEs leva à contenção de memória, situação que não ocorre no caso da memória distribuída, em que cada PE possui sua própria memória. Por causa dos resultados práticos e requisitos do usuário, a arquitetura de memória distribuída MIMD é superior aos outros modelos existentes.

LEI DE AMDAHL E SPEEDUP

Quando buscamos o paralelismo da execução de instruções, desejamos que o nosso sistema realize suas tarefas da forma mais rápida possível. Entretanto, existe um limite teórico para a execução dessas tarefas, e ele é determinado pela **lei de Amdahl**.

Que no caso é a resolução de um simples problema, nós temos o nosso ganho, que é 1 dividido pelo Sequencial do programa + a diferença (1 - Sequencial, ou seja o percentual do meu programa que é paralelizável), dividido pela quantidade de processadores.

> **Atenção**: A lei de Amdahl afirma que podemos paralelizar e/ou distribuir nossos cálculos tanto quanto quisermos, ganhando em desempenho à medida que adicionamos recursos de computação. No entanto, nosso código não pode ser mais rápido do que a velocidade de suas partes sequenciais combinadas (ou seja, não paralelizáveis) em um único processador.

Vamos ver um exemplo de problema: Suponhamos que 80% do nosso programa não podemos paralelizarm então:

Colocado de modo mais formal, a lei de Amdahl tem a seguinte formulação.

Dado um algoritmo que é parcialmente paralelo, vamos chamar *P* sua **fração paralela** e *S* sua **fração serial** (*S* + *P* = 100%). Além disso, chamemos *T*(n) o **tempo de execução** (em segundos) do algoritmo ao usar *n* processadores.

Então, a seguinte relação se mantém:

T(n) ≥ S . T(1) + P . T(1) / n

A relação anterior afirma o seguinte:

O tempo de execução do algoritmo descrito aqui em n processadores é igual — e geralmente maior — do que o tempo de execução de sua parte serial em um processador (isto é, S . T(1)) mais o tempo de execução de sua parte paralela em um processador (ou seja, P . T(1)) dividido por n (número de processadores).

À medida que aumentamos o número n de processadores usados por nosso código, o segundo termo da equação fica cada vez menor, tornando-se insignificante em relação ao primeiro termo. Nesses casos, a relação anterior simplesmente se torna esta:

$T(\infty) \approx S.T(1)$

A tradução dessa relação pode ser interpretada da seguinte forma:

O tempo de execução do algoritmo descrito aqui em um número infinito de processadores (ou seja, um número realmente grande de processadores) é aproximadamente igual ao tempo de execução de sua parte serial em um único processador (ou seja, `S.T(1)`).

Agora, vamos parar por um segundo e pensar sobre as implicações da lei de Amdahl. O que temos aqui é uma observação bastante simples: muitas vezes, não podemos paralelizar totalmente nossos algoritmos.

O que significa que, na maioria das vezes, não podemos ter `S = 0` nas relações anteriores. As razões para isso são inúmeras:

- Copiar dados e/ou código para onde os vários processadores serão capazes de acessá-los.
- Dividir os dados em pedaços e mover esses pedaços pela rede.
- Coletar os resultados de todas as tarefas simultâneas e executar algum processamento adicional nelas, e assim por diante.

> **Atenção**: Seja qual for o motivo, se não pudermos paralelizar

> totalmente o algoritmo, eventualmente o tempo de execução do código será dominado pelo desempenho da fração serial. Não apenas isso, mas mesmo antes que aconteça, começaremos a ver acelerações cada vez menores do que o esperado.

Como uma observação lateral, algoritmos que são totalmente paralelos são geralmente chamados de "**embaraçosamente paralelos**" e oferecem propriedades de escalabilidade impressionantes (com acelerações geralmente lineares com o número de processadores). É claro que não há nada de constrangedor nesses softwares. Porém, infelizmente, eles não são tão comuns quanto gostaríamos.

Como a execução paralela dos algoritmos pode realmente fazer com que a tarefa seja executada de forma mais rápida, temos de mensurar esse ganho, o que chamamos de **speedup**.

> O **speedup** é definido como a razão entre o tempo de execução serial do melhor algoritmo sequencial para resolver um problema e o tempo gasto pelo algoritmo paralelo para resolver o mesmo problema com p processadores.

É simplesmente a razão entre o tempo gasto com uma execução serial dividido pelo tempo gasto pela execução paralela, conforme

podemos ver na fórmula a seguir.

Speedup = $Tempo_{serial}$ / $Tempo_{Paralelo}$ = T_8 / T_p

Vamos tentar visualizar toda a lei de Amdahl, bem como o speedup associado com alguns números.

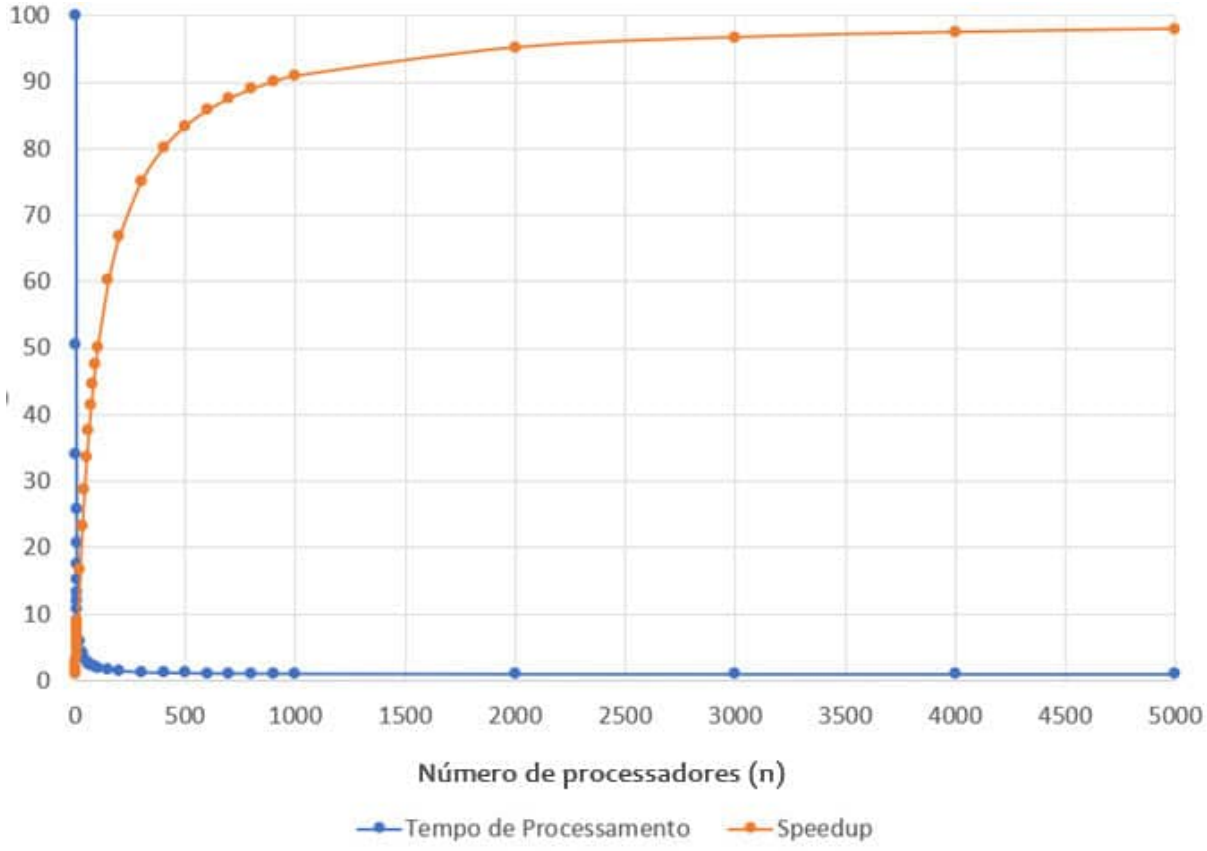
- Suponha que o algoritmo leve 100 segundos para ser executado em um único processador.
- Vamos supor também que podemos paralelizar 99% disso, o que seria uma façanha incrível, na maioria das vezes.
- Podemos tornar o código mais rápido aumentando o número de processadores que usamos, conforme esperado.

  Veja o cálculo:

  T(1) = 100s
  T(10) ≈ 0,01 . 100s + 0,99*100s / 10 = 10,9s => Speedup = 100/10,9 = 9,2x
  T(100) ≈ 1s + 0,99s = 1,99s ===> Speedup = 100/1,99 = 50,2x
  T(1000) ≈ 1s + 0,099s = 1,099s => Speedup = 100/1,099 = 91x

  A partir dos números anteriores, vemos que o aumento da aceleração com valores crescentes de n é bastante decepcionante.

Começamos com um aumento de velocidade realmente incrível de 9,2x usando 10 processadores, e então caímos para apenas 50x ao usar 100 processadores e um insignificante 91x ao usar 1.000 processadores.



A imagem anterior mostra a aceleração de melhor caso esperada para o mesmo algoritmo (calculado até). Não importa quantos processadores usamos; não podemos obter um aumento de

velocidade maior que , o que significa que o mais rápido que o código rodará é um segundo, que é o tempo que sua fração serial leva em um único processador, exatamente como foi previsto pela lei de Amdahl.

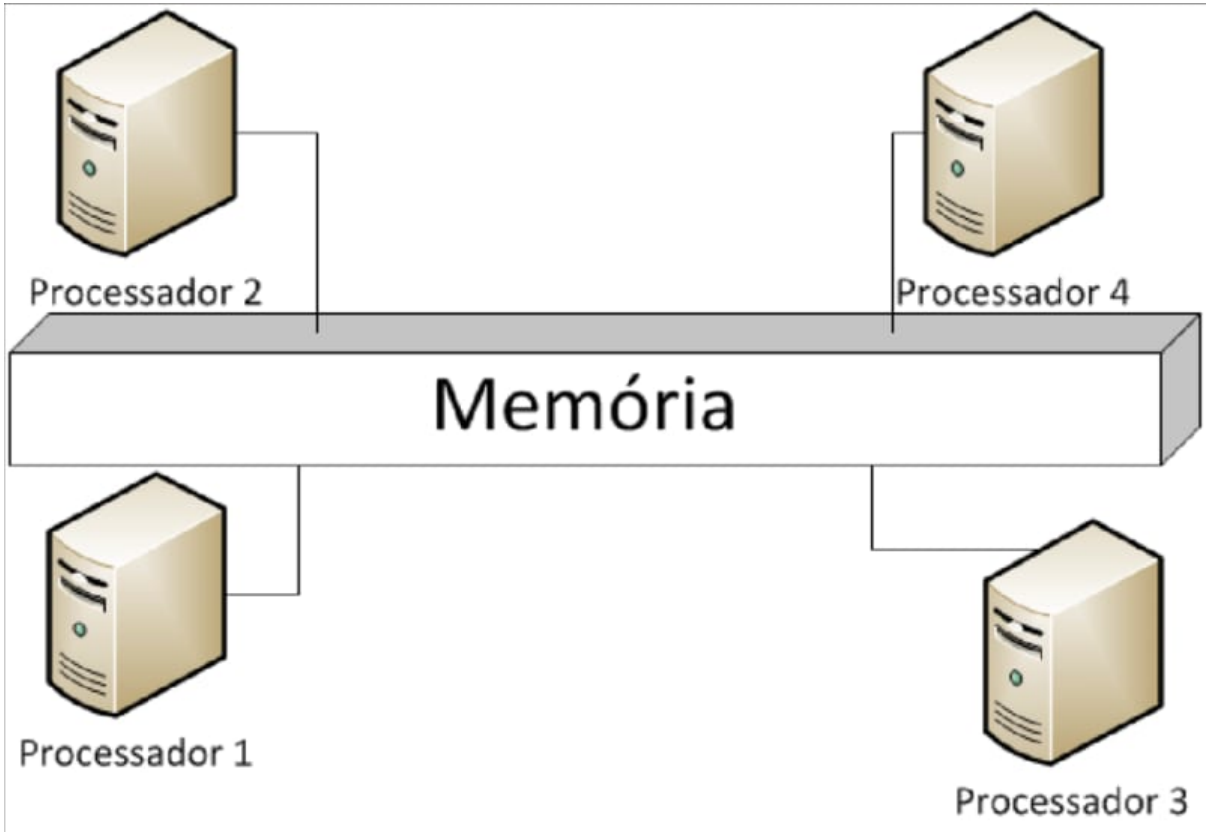
A lei de Amdahl nos diz duas coisas:

> A primeira é o quanto de aceleração podemos razoavelmente esperar na melhor das hipóteses e quando parar de adicionar hardware ao sistema devido aos retornos decrescentes.
>
> A segunda é que a lei de Amdahl se aplica igualmente a sistemas distribuídos e **sistemas híbridos distribuídos**, ou seja são aqueles que combinam diferentes tipos de aspectos arquiteturais, paralelamente. Nesses casos, n refere-se ao número total de processadores em computadores disponíveis no sistema.

Um aspecto que deve ser mencionado nesse ponto é que à medida que os sistemas que podemos usar se tornam mais poderosos, nossos algoritmos distribuídos levarão cada vez menos tempo para rodar, se puderem fazer uso dos ciclos extras.

- https://player.vimeo.com/video/549004821?h=68a286301b&app_id=122963

Memória Compartilhada



Para se aproveitar do paralelismo disponível nos sistemas modernos, os programas podem ser desenvolvidos para utilizar mais de um core (núcleos) simultaneamente. Isso significa que um programa com essa capacidade, sendo executado em um servidor, por exemplo, com 8 cores, poderia ser executado até oito vezes mais rápido (já vimos de acordo com a lei de Amdahl que isso não

é verdade). Para isso, o programa precisa ser explicitamente programado com essa capacidade, usando bibliotecas ou técnicas específicas.

A **memória compartilhada** é a memória que pode ser acessada simultaneamente por vários programas com a intenção de fornecer comunicação entre eles. Além de ser uma forma de comunicação entre os diversos programas, possibilita a economia de recursos, evitando cópias redundantes. Dentro desse contexto, os programas podem ser executados em um único processador ou mesmo em vários processadores separados.

O conceito de memória compartilhada pode ser aplicado tanto em hardware como em software. Sua aplicação em hardware não é o foco do nosso estudo neste tema, assim, passaremos apenas a considerar sua aplicação em software.

Em termos de software, a memória compartilhada pode ser um método de comunicação entre processos (Interprocess Communication – IPC), ou seja, uma maneira de trocar dados entre programas em execução ao mesmo tempo. Um processo criará uma área na RAM a qual outros processos podem acessar.

Como dissemos anteriormente, o uso de memória compartilhada

permite a conservação de recursos de memória, evitando cópias de dados de uma mesma instância, usando mapeamentos de memória virtual ou com suporte explícito do programa em questão.

Uma vez que ambos os processos podem acessar a área de memória compartilhada como memória de trabalho regular, essa é uma forma muito rápida de comunicação, utilizando-se, via de regra, a comunicação síncrona nesses sistemas. Por outro lado, é menos escalável, pois os processos de comunicação devem estar rodando na mesma máquina, ficando limitado aos recursos desta. Existem ainda outros problemas, pois se os processos que compartilham a mesma memória compartilhada estiverem sendo executados em CPUs separadas, podem surgir problemas de coerência de cache.

Passemos a um exemplo onde todos os cores de CPUs utilizados estão sempre no mesmo servidor, e tem acesso a mesma memória, chamamos essa técnica de **paralelização de memória compartilhada**. São termos relacionados a esse mecanismo: SMP, Threads, OpenMP, os quais não serão objetos de nosso estudo.

> **Thread** é uma forma como um processo/tarefa de um programa de computador é divido em duas ou mais tarefas que podem ser

> executadas concorrentemente. Os threads criados ocupam a CPU do mesmo modo que o processo criador, e também são escalonadas pelo próprio processo.
>
> **Exemplo**: Programas em C/C++ podem ser “facilmente” paralelizados para esse modelo usando a biblioteca **OpenMP**. Para isso, são necessárias as inclusões de algumas diretivas de compilação (pragmas) no código e a utilização de um compilador compatível.

A seguir, temos um exemplo de memória compartilhada utilizando a linguagem C. Claro que para utilizá-lo, a **biblioteca OMP** já deve estar instalada em seu computador.

Inicialmente, temos que incluir o cabeçalho OpenMP para nosso programa junto com os arquivos de cabeçalho padrão.

memoriaCompartilhada.c

```
//OpenMP header
#include <omp.h>
```

No **OpenMP**, precisamos mencionar a região que vamos fazer como paralela usando a palavra-chave `pragma omp parallel`.

O `pragma omp parallel` é usado para bifurcar threads adicionais para realizar o trabalho paralelo na região determinada. O encadeamento original será indicado como o encadeamento mestre com ID de encadeamento 0.

O código para a criação de uma região paralela seria:

memoriaCompartilhada.c

```
#pragma omp parallel
{
// As variáveis criadas aqui serão compartilhadas
por todas
// as theads em execução
//Parallel region code
}
```

Na região entre colchetes, todas as variáveis ali declaradas serão compartilhadas por todos os threads em execução. Em outras palavras, todas as variáveis criadas nesse espaço de memória serão compartilhadas por todas as instâncias de thread criadas.

O comando poderia ser, por exemplo:

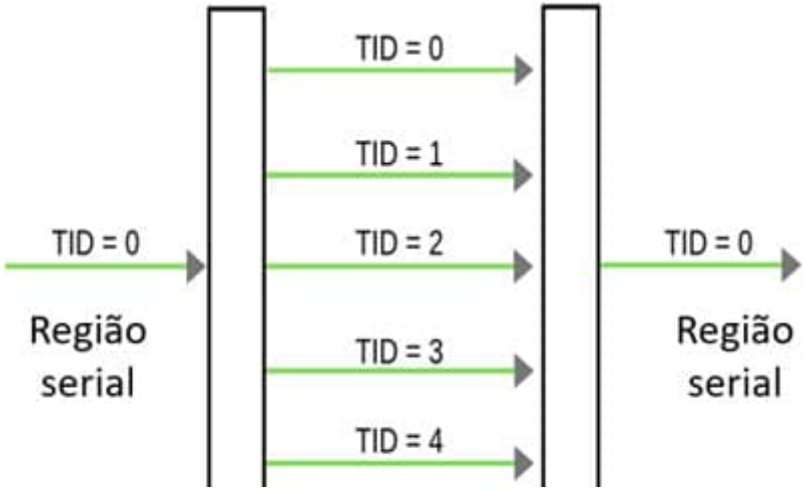
memoriaCompartilhada.c

```
#pragma omp parallel
{
  printf("Ola Mundo... da thread = %d\n",
omp_get_thread_num());
}
```

Defina o número de threads para executar o programa usando a variável externa.

memoriaCompartilhada.c

```
export OMP_NUM_THREADS=5
```



Região paralela

De acordo com a imagem anterior, uma vez que o compilador encontra o código das regiões paralelas, o thread mestre (thread que tem o id de thread 0) será bifurcado no número especificado de threads.

Aqui, ele será dividido em 5 threads porque inicializaremos o número de threads a serem executados como 5, usando o comando `export OMP_NUM_THREADS = 5`.

Todo o código dentro da região paralela será executado por todos os threads simultaneamente, bem como todas as variáveis eventualmente declaradas nesse código serão compartilhadas. Uma vez que a região paralela terminar, todos os threads serão mesclados no thread mestre.

> **Comentário**: Não se preocupe em executar este programa no seu computador. A biblioteca **OpenMP** não será o foco do nosso estudo. Mas, caso se interesse no exemplo atual, siga os comandos abaixo descritos.

Compile o programa no ambiente Linux, utilizando o seguinte

comando:

```
gcc -o hello -fopenmp hello.c
```

Execute o programa com o seguinte comando:

```
./hello
```

A seguir, está o programa completo descrito anteriormente.

memoriaCompartilhada.c

```
// Usando o OpenMP para executar o Hello World
compartilhado
// utilizando a linguagem C

// OpenMP header
#include <omp.h>

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
```

```
{

    // Inicio da Regiao Paralela
    #pragma omp parallel
    {

        printf("Ola Mundo... da thread = %d\n",
omp_get_thread_num());
    }
    // Fim da regiao paralela
}
```

Como especificamos, o número de **threads** a serem executados como 5, estes 5 `threads` executarão a mesma instrução de impressão ao mesmo tempo. Aqui, não podemos garantir a ordem de execução dos threads, ou seja, a ordem de execução da instrução na região paralela não será a mesma para todas as execuções.

Na imagem a seguir, durante a primeira execução do programa, o encadeamento 1 é concluído primeiro, sendo que na segunda

execução do programa, o encadeamento `0` é o primeiro a ser concluído. `omp_get_thread_num ()` retornará o número do thread (tarefa) associado ao segmento.

Veja a saída de uma execução do programa.

```
aluno@ubuntu: ~$ export OMP_NUM_THREADS=5
aluno@ubuntu: ~$ gcc -o hello -fopenmp hello.c
aluno@ubuntu: ~$ ./hello
Ola Mundo... da thread = 1
Ola Mundo... da thread = 0
Ola Mundo... da thread = 4
Ola Mundo... da thread = 3
Ola Mundo... da thread = 2
```

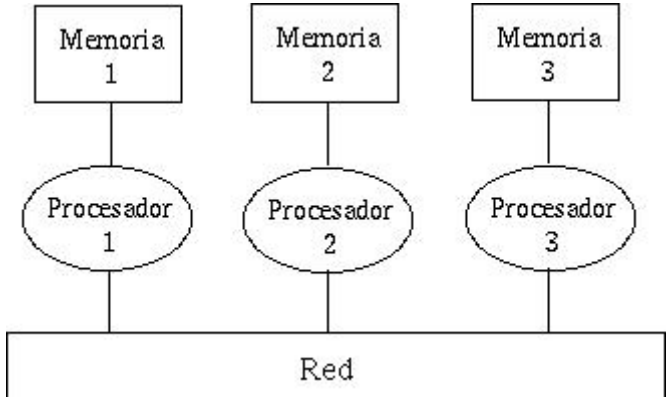
Quando executado por várias vezes: a ordem de execução dos threads muda a cada vez.

```
aluno@ubuntu: ~$ ./hello
Ola Mundo... da thread = 1
Ola Mundo... da thread = 0
Ola Mundo... da thread = 4
Ola Mundo... da thread = 3
Ola Mundo... da thread = 2
```

```
aluno@ubuntu: ~$ ./hello
Ola Mundo... da thread = 0
Ola Mundo... da thread = 4
Ola Mundo... da thread = 3
Ola Mundo... da thread = 2
Ola Mundo... da thread = 1
```

> A memória compartilhada pode ser um método de comunicação entre processos. A memória compartilhada não é escalável, pois os processos compartilham a mesma memória no mesmo computador, dificultando a expansão da quantidade de memória disponível. Por compartilharem o mesmo espaço de memória, ocorrem problemas de coerência, pois os processos estão disputando o mesmo espaço de memória.

Memória Distribuída



A **memória distribuída** refere-se a um sistema de computador multiprocessador no qual cada processador tem sua própria memória privada. As tarefas computacionais só podem operar em dados locais e, se forem necessários dados remotos, a tarefa computacional deve se comunicar com um ou mais processadores remotos.

Em contraste, como vimos anteriormente, um multiprocessador de memória compartilhada oferece um único espaço de memória usado por todos os processadores. Os processadores não precisam estar cientes de onde os dados residem, exceto de que pode haver penalidades de desempenho e que as **condições de corrida** devem ser evitadas.

> **Condições de corrida** são situações caracterizadas pelo acesso simultâneo de dois ou mais processos a dados compartilhados, em um processamento ou sistema cujo resultado final depende da ordem de execução de seus processos.

Em um sistema de memória distribuída, normalmente, há um processador, uma memória e alguma forma de interconexão que permite que os programas em cada processador interajam uns com os outros. A interconexão pode ser organizada com enlaces ponto

a ponto ou o hardware separado pode fornecer uma rede de comutação.

A topologia da rede é um fator chave para determinar como a máquina multiprocessadora é dimensionada. Os enlaces entre os nós podem ser implementados usando algum protocolo de rede padrão (onde a comunicação via placas Ethernet é a mais comum) ou algum outro tipo de comunicação. Como a comunicação, geralmente, é por protocolos de rede, pode existir uma grande latência nestas operações, que em operações bloqueantes (síncronas) pode não ser apropriado. Assim, as comunicações assíncronas (em regra as não bloqueantes) são as mais utilizadas nesse tipo de sistema.

O principal problema na programação de sistemas de memória distribuída é como distribuir os dados pelas memórias. Dependendo do problema resolvido, os dados podem ser distribuídos estaticamente ou podem ser movidos através dos nós. Os dados podem ser movidos sob demanda ou podem ser enviados para os novos nós com antecedência.

> A principal vantagem da memória compartilhada é que ela oferece um espaço de endereço unificado no qual todos os dados podem

> ser encontrados, além dos dados serem acessados com maior rapidez.

A vantagem da memória distribuída é que ela **exclui condições de corrida**, e a principal preocupação do programador é pensar sobre a distribuição de dados. Da mesma forma, a memória distribuída é muito mais escalável do que a memória compartilhada, bastando acessar novos nós a rede. Por outro lado, em que pese a ocultar o mecanismo de comunicação, não é possível deixar de considerar a latência de comunicação para acessar os dados.

Um programa que usa as funcionalidades de memória distribuída tem de ser explicitamente desenvolvido com essa capacidade. Geralmente, isso é feito usando uma biblioteca **MPI** (no caso da memória compartilhada, usamos no nosso exemplo o **OpenMP**). Não se preocupe em executar este código no seu computador, alguns conceitos não foram tratados, mas irá permitir que você tenha uma visão do emprego da memória distribuída, conforme veremos em nosso vídeo: https://player.vimeo.com/video/549006057

memoriaDistribuida.c

```
// Bibliotecas do MPI necessarias
```

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[])
{
    int  numtasks, rank, len, rc;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    // initializando o MPI
    MPI_Init(&argc,&argv);
    // obtendo o numero de tasks
    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    // obtendo o rank
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    // obtendo o nome do processador
    MPI_Get_processor_name(hostname, &len);
    printf ("Number of tasks= %d My rank= %d
Running on %s\n", numtasks,rank,hostname);
    // finalizando o MPI
    MPI_Finalize();
}
```

> A memória distribuída é altamente escalável e a vantagem da memória distribuída é que ela exclui condições de corrida. O programador deve pensar sobre a distribuição de dados. Os processadores não precisam estar cientes de onde os dados residem, exceto de que pode haver penalidades de desempenho e que as condições de corrida devem ser evitadas.

PARALELISMO DE DADOS

O **paralelismo de dados** é a paralelização dos dados entre vários processadores em ambientes de computação paralela:

- Concentra-se na distribuição dos dados em nós diferentes, que operam nos dados em paralelo.
- Pode ser aplicado em **estruturas de dados regulares**, como **arrays** e **matrizes**, trabalhando em cada elemento em paralelo.

  Um **trabalho paralelo de dados em uma matriz** de n elementos pode ser dividido igualmente entre todos os processadores.

  Vamos supor que queremos somar todos os elementos da matriz fornecida e o tempo para uma única operação de adição é

unidades de tempo T_a.

No caso de execução sequencial, o tempo gasto pelo processo será $n \times T_a$ unidades de tempo, pois soma todos os elementos de uma matriz.

Por outro lado, se executarmos esse trabalho como um trabalho paralelo de dados em 4 processadores, o tempo gasto seria reduzido para $(n \times T_a)/4$, desconsiderando eventuais atrasos de execução e operações obrigatoriamente serializáveis já citados na lei Amdahl. Nesse caso, a **execução paralela** resulta em uma aceleração de 4 em relação à execução sequencial.

> **Atenção**: Uma questão interessante a se notar é que a localidade das referências de dados desempenha um papel importante na avaliação do desempenho de um modelo de programação paralela de dados.

A localidade dos dados depende de dois fatores:

1. Acessos à memória realizados pelo programa.
2. Tamanho do cache.

Em um sistema multiprocessador que executa um único conjunto de instruções (SIMD), arquitetura de dados que já explanamos, o

paralelismo de dados é obtido quando cada processador executa a mesma tarefa em diferentes dados distribuídos.

Em algumas situações, um único thread de execução controla as operações em todos os dados.

Em outras, diferentes threads controlam a operação, mas executam o mesmo código.

> Exemplo: Considere a multiplicação e adição de matrizes de maneira sequencial. Veja a seguir o pseudocódigo sequencial para multiplicação e adição de duas matrizes onde o resultado é armazenado na matriz C. O pseudocódigo para multiplicação calcula o produto escalar de duas matrizes A e B e armazena o resultado na matriz de saída C.
>
> Se os programas a seguir forem executados sequencialmente, o tempo necessário para calcular o resultado seria $0(n^3)$, assumindo que os comprimentos de linha e de coluna de ambas as matrizes são n e $0(n)$ para multiplicação e adição, respectivamente.

C multiplicacaoDeMatrizes.c

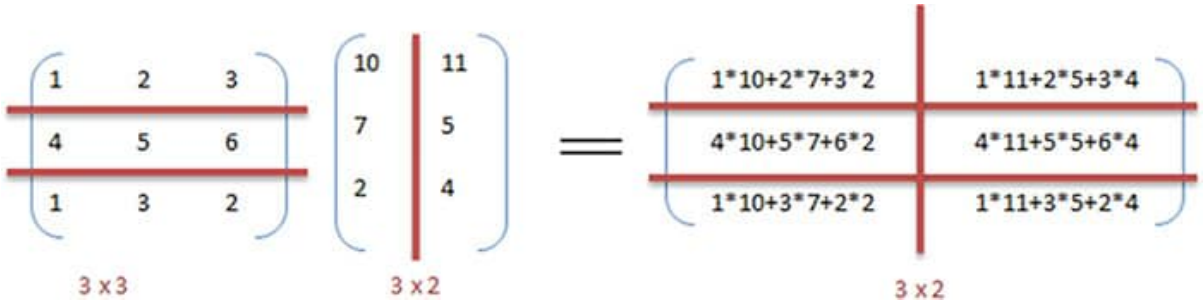
```
// Multiplicacao de matrizes
for (i = 0; i < row_length_A; i++)
{
    for (k = 0; k < column_length_B; k++)
    {
        sum = 0;
        for (j = 0; j < column_length_A; j++)
        {
            sum += A[i][j] * B[j][k];
        }
        C[i][k] = sum;
    }
}
// Adicao de arrays
for (i = 0; i < n; i++) {
    c[i] = a[i] + b[i];
}
```

Podemos explorar o paralelismo de dados no código anterior para executá-lo mais rapidamente, pois a aritmética é independente em relação ao loop.

A paralelização do código de multiplicação da matriz é obtida usando o **OpenMP**. Uma diretiva OpenMP, "`omp parallel for`" instrui o compilador a executar o código no loop `for` em paralelo.

Para multiplicação, podemos dividir a matriz A e B em blocos ao longo de linhas e colunas, respectivamente. Isso nos permite calcular cada elemento na matriz C individualmente, tornando a tarefa paralela.

> **Exemplo**: Matriz A versus Matriz B
>
> `A[m x n].B[n x k]` pode ser finalizado em `O(n)` em vez de `O(m*n*k)` quando executado em paralelo usando `m * k` processadores.



C multiplicacaoDeMatrizesEmParalelo.c

```
// Multiplicação de matrizes em paralelo
#pragma omp parallel for schedule(dynamic,1)
collapse(2)
```

```
for (i = 0; i < row_length_A; i++){
    for (k = 0; k < column_length_B; k++){
        sum = 0;
        for (j = 0; j < column_length_A; j++){
            sum += A[i][j] * B[j][k];
        }
        C[i][k] = sum;
    }
}
```

Pode-se observar a partir do exemplo que, conforme os tamanhos das matrizes continuam aumentando, serão necessários muitos processadores. Manter o tempo de execução baixo é a prioridade, mas, na medida em que o tamanho da matriz aumenta, nos deparamos com outras restrições, como a complexidade de tal sistema e seus custos associados. Portanto, restringindo o número de processadores no sistema, podemos ainda aplicar o mesmo princípio e dividir os dados em blocos maiores para calcular o produto de duas matrizes.

Para adição de matrizes em uma implementação paralela de

dados, vamos supor um sistema mais modesto com duas CPUs A e B. A CPU A poderia adicionar todos os elementos da metade superior das matrizes, enquanto a CPU B poderia adicionar todos os elementos da metade inferior das matrizes. Como os dois processadores funcionam em paralelo, a tarefa de realizar a adição do array levaria metade do tempo de realizar a mesma operação em série usando apenas uma CPU.

> O paralelismo de dados está muito em voga nos dias atuais, e talvez as placas com tecnologia GPU sejam o sinônimo desta tecnologia. Esse tipo de abordagem não será o foco do nosso estudo.
>
> Utiliza-se computação síncrona no paralelismo de dados. A mesma tarefa é executada em diferentes subconjuntos de dados e a quantidade de paralelização é proporcional ao tamanho da entrada.

Para saber mais sobre os assuntos explorados neste tema, pesquise:

- Os livros mais adotados para sistemas distribuídos, que são os de COULOURIS et al., 2013 e TANENBAUM e STEEM, 2006. Neles, podem ser encontrados os principais conceitos referentes a esses

assuntos.

- O tutorial de OpenMP do Lawrence Livermore National Laboratory, onde se encontram orientações bem detalhadas sobre programação paralela e distribuída.
- Um exemplo completo e detalhado da multiplicação de matrizes, disponível em Matrix Multiplication with OpenMP e Mxm Openmp c.

PARALELISMO DE TAREFAS

O **paralelismo de tarefas** (também conhecido como **paralelismo de função** e **paralelismo de controle**) é uma forma de paralelismo de código de computador em vários processadores em ambientes de computação paralela. Concentra-se na distribuição de tarefas — executadas simultaneamente por processos ou threads — em diferentes processadores. Em contraste com o paralelismo de dados, que envolve a execução da mesma tarefa em diferentes componentes de dados, distingue-se pela execução de muitas tarefas diferentes ao mesmo tempo nos mesmos dados.

> **Atenção**: Em um sistema multiprocessador, o paralelismo de tarefas é obtido quando cada processador executa um thread diferente (ou processo) nos mesmos dados ou em dados

> diferentes, diferentemente do que vimos no paralelismo de dados, onde os threads executam a mesma tarefa.

Os threads podem executar o mesmo código ou código diferente. No caso geral, diferentes threads de execução se comunicam uns com os outros enquanto funcionam, mas isso não é um requisito. A comunicação geralmente ocorre passando dados de um thread para o próximo como parte de um fluxo de trabalho. Essa comunicação entre threads pode ser executada inclusive utilizando mecanismos de memória distribuída.

> **Exemplo**: Se um sistema está executando o código em um sistema de dois processadores (CPUs "a" e "b") em um ambiente paralelo e queremos fazer as tarefas "A" e "B", é possível dizer CPU "a" para fazer a tarefa "A" e CPU "b" para fazer a tarefa "B" simultaneamente, reduzindo assim o tempo de execução. As tarefas podem ser atribuídas usando instruções condicionais.

O paralelismo de tarefas enfatiza a natureza distribuída (paralelizada) do processamento (ou seja, threads), em oposição aos dados (paralelismo de dados).

O **paralelismo de nível de thread** (**TLP**) é o paralelismo inerente a um aplicativo que executa vários threads de uma vez. Esse tipo de

paralelismo é encontrado principalmente em aplicativos escritos para servidores comerciais, como bancos de dados. Ao executar muitos threads de uma vez, esses aplicativos são capazes de tolerar as altas quantidades de input e output e latência do sistema de memória em que suas cargas de trabalho podem incorrer - enquanto um thread está atrasado esperando por um acesso à memória ou disco, outros threads podem fazer um trabalho útil.

A exploração do paralelismo de nível de thread também começou a fazer incursões no mercado de desktops com o advento dos microprocessadores multicores. Nos computadores de mesa, isso ficou evidenciado com o advento do Windows NT e do Windows 95, em que pese a essas tecnologias, já eram possíveis em outros sistemas operacionais, como o SunOS e o Solaris. Isso ocorreu porque, por várias razões, tornou-se cada vez mais impraticável aumentar a velocidade do clock ou as instruções por clock de um único núcleo.

> **Comentário**: Se essa tendência continuar, novos aplicativos terão de ser projetados para utilizar vários threads para se beneficiar do aumento potencial do poder de computação. Isso contrasta com as inovações anteriores do microprocessador, nas quais o código existente era automaticamente acelerado ao ser executado em um

> computador mais novo / mais rápido.
>
> O exemplo que mostramos em memórias compartilhada também serve como exemplo de uma tarefa distribuída, onde cada thread executava a função de imprimir o seu Id. Uma forma simples seria colocar uma função dentro do espaço reservado para o paralelismo.

- https://player.vimeo.com/video/549007806

> As tarefas diferentes podem ser executadas nos mesmos dados que foram disponibilizados para processamento. A quantidade de paralelização é proporcional ao número de tarefas independentes realizadas e como cada processador executará um thread, ou processo, diferente no mesmo conjunto de dados ou em um conjunto diferente de dados, a aceleração é menor.

Fundamentos de Computação Paralela

O desenvolvimento dos sistemas distribuídos nos permitiu acessar diversos tipos de aplicações, disponibilizadas na internet. Essas aplicações mudaram a forma de estudarmos, de trabalharmos, de nos divertimos e até a maneira como as pessoas se relacionam.

Uma aplicação distribuída, porém, tem algumas complexidades.

Por ser distribuída, existem vários processos que vão compô-la e, consequentemente, há a necessidade de uma coordenação entre esses processos.

Vários processos sendo executados simultaneamente dão origem ao conceito de programação paralela – divisão de uma tarefa computacional em instâncias (processos) independentes e que podem ser executados de forma paralela.

Os processos que são executados em paralelo, no entanto, podem trazer algumas situações inconvenientes durante sua execução. Imagine, por exemplo, que existe um recurso – como uma impressora – que deve ser acessado por esses processos que estão executando em paralelo. Se todos quiserem acessar simultaneamente, teremos problemas para definir quem irá imprimir primeiro.

É necessário conhecer os fundamentos da computação paralela para entender os problemas que existem nesse paradigma computacional e quais mecanismos podemos utilizar para evitar ou, pelo menos, minimizar os seus efeitos.

Por fim, conheceremos diversas plataformas que podem ser utilizadas para o desenvolvimento de programas paralelos.

- https://player.vimeo.com/video/558221759

  O thread mestre normalmente espera por solicitações de trabalho – por exemplo, em uma rede –, e, quando um novo pedido chega, ele bifurca/inicia um thread de trabalho.

  Esse thread executa a solicitação e, quando conclui o trabalho, termina e se junta ao thread mestre. Esse tipo de modelo faz uso eficiente dos recursos do sistema, uma vez que os recursos exigidos por um thread só estão sendo usados enquanto este está realmente em execução.

VARIÁVEIS COMPARTILHADAS

Antes de mais nada, saiba que:

> Em programas de memória compartilhada, as **variáveis** podem ser **compartilhadas** ou **privadas**.

Em um ambiente onde um processo possui vários **threads**, as **variáveis compartilhadas podem ser lidas ou escritas por qualquer um deles**, e **variáveis privadas normalmente podem ser acessadas apenas por um thread**. Quando a comunicação entre os threads é feita por meio de variáveis compartilhadas, ela é implícita, não explícita – como por soquetes (sockets) ou named

pipes.

> **Comentário**: Em muitos ambientes, os programas de memória compartilhada usam threads dinâmicos. Nesse modelo, geralmente há um thread mestre e, em qualquer momento, uma coleção (possivelmente vazia de início) de threads de trabalho.

O **thread mestre** normalmente espera por solicitações de trabalho – por exemplo, em uma rede –, e, quando um novo pedido chega, ele bifurca/inicia um thread de trabalho.

Esse thread executa a solicitação e, quando conclui o trabalho, termina e se junta ao thread mestre. Esse tipo de modelo faz uso eficiente dos recursos do sistema, uma vez que os recursos exigidos por um thread só estão sendo usados enquanto este está realmente em execução.

PARADIGMA DE THREAD ESTÁTICO

Uma alternativa ao paradigma dinâmico é o **paradigma de thread estático**. Nesse paradigma, todos os threads são bifurcados/iniciados após qualquer configuração necessária pelo thread mestre e os threads são executados até que todo o trabalho seja concluído. Depois que os threads se juntam ao thread mestre, este

pode fazer alguma limpeza (por exemplo, liberação de memória) e, então, também finaliza esses threads.

> **Atenção**: Em termos de uso de recursos, isso pode ser menos eficiente: se um thread estiver ocioso, seus recursos (por exemplo: pilha, contador de programa etc.) não podem ser liberados.

No entanto, bifurcar e unir threads podem ser operações bastante demoradas. Portanto, se os recursos necessários estiverem disponíveis, o paradigma de thread estático tem potencial de melhor desempenho do que o paradigma dinâmico. Ele também tem a virtude de estar mais próximo do paradigma mais amplamente usado para programação de memória distribuída, então, parte da memória que é usada para um tipo de sistema é preservada para o outro.

> Assim, o paradigma de thread estático é o mais frequente.

CÁLCULO NÃO DETERMINÍSTICO

Em qualquer sistema **MIMD**, Instrução Múltipla - Dados Múltiplos – em inglês, *Multiple Instruction Multiple Data*, no qual os processadores são executados de forma assíncrona, é provável que haja não determinismo das tarefas. Um cálculo é não

determinístico se uma entrada específica pode resultar em saídas diferentes. Se vários threads estiverem executando de forma independente, a taxa relativa na qual eles concluirão as instruções varia de execução para execução e, portanto, os resultados do programa podem ser diferentes a cada execução.

Exemplo:

Suponha que temos dois threads, um com `id 0` e o outro com `id 1`. Suponha também que cada um está armazenando uma variável privada `x`, o valor de `x` na thread `0` é `5` e na thread `1` é `8`. Além disso, imagine que ambos os threads executem o seguinte código:

```
printf("Thread %d > val = %dnn", x);
```

Então, a saída pode ser:

```
Thread 0 >  val = 5

Thread 1 >  val = 8
```

Mas pode ser também:

```
Thread 1 > val = 8


Thread 0 > val = 5
```

> Na verdade, as coisas podem ser ainda piores: a saída de um thread pode ser interrompida pela saída de outro thread. No exemplo anterior, estamos apenas tratando de uma saída.

No entanto, como os threads estão executando de forma independente e interagindo com o sistema operacional, o tempo que leva para um thread completar um bloco de instruções varia de execução para execução. Portanto, a ordem em que essas instruções são concluídas não pode ser prevista.

> **Comentário**: Em muitos casos, o não determinismo não é um problema. Em nosso exemplo, uma vez que rotulamos a saída com a classificação do tópico, a ordem em que a saída aparece provavelmente não importa. No entanto, também existem muitos casos em que o não determinismo, especialmente em programas de memória compartilhada, pode ser desastroso, porque pode facilmente resultar em erros de programa.

Vejamos um exemplo simples com dois threads: suponha que cada

thread calcule um valor inteiro, que ele armazena em uma variável privada `val`, e que queremos adicionar os valores armazenados em `val` em um local de memória compartilhada x que foi inicializado em `0`. Ambos os threads, portanto, desejam executar um código semelhante a este:

```
val = Compute_val(my_rank);
x += val;
```

Agora, lembre-se de que uma adição normalmente requer o carregamento dos dois valores a serem adicionados aos registradores, a adição dos valores e, por fim, o armazenamento do resultado. Para manter o exemplo relativamente simples, vamos presumir que os valores são carregados da memória principal diretamente nos registradores e armazenados na memória principal diretamente dos registradores. Na tabela a seguir, visualizamos uma sequência possível de execução desses threads utilizando os valores 5 e 8 do exemplo anterior:

Tempo	Núcleo 1	Núcleo 2
0	Carrega o valor de val	Chama a função

		Compute_val
1	Carrega x = 0 para o registrador	Carrega o valor de val
2	Carrega val = 5 para o registrador	Carrega x = 0 para o registrador
3	Adiciona val a x	Carrega val = 8 para o registrador
4	Armazena x = 5	Adiciona val a x
5	Inicia outro trabalho	Armazena x = 8

Claramente, não é isso que queremos, pois o resultado deveria considerar que estamos incrementando o valor de x, e o resultado esperado deveria ser 13. Assim, é fácil imaginar outras sequências de eventos que resultam em um valor incorreto para x.

> O não determinismo aqui é resultado do fato de que duas threads estão tentando atualizar mais ou menos simultaneamente à localização da memória x.

Quando threads ou processos tentam acessar simultaneamente um recurso, e os acessos podem resultar em erro, costumamos dizer que o programa tem uma condição de corrida, porque os threads ou processos estão em uma “corrida de cavalos”. Ou seja, o resultado do cálculo depende de qual thread ganha a corrida. Em nosso exemplo, os threads estão em uma corrida para executar x `+ = val`. Nesse caso, a menos que um encadeamento conclua x `+ = val` antes do outro encadeamento iniciar, o resultado será incorreto.

> **Você sabia**: Um bloco de código que só pode ser executado por um thread de cada vez é chamado de **seção crítica** (veremos isso em mais detalhes adiante), e, geralmente, é nosso trabalho como programadores garantir acesso mutuamente exclusivo à seção crítica. Em outras palavras, precisamos garantir que, se um encadeamento estiver executando o código na seção crítica, os outros encadeamentos serão excluídos.

MUTEX

O mecanismo mais comumente utilizado para garantir a exclusão mútua é um **bloqueio de exclusão mútua** ou **mutex** ou **bloqueio**.

> **Atenção**: A ideia básica é que cada seção crítica seja protegida por um “tipo de fechadura”. Antes que um thread possa executar o código na seção crítica, ele deve "obter" essa fechadura, no caso o mutex, chamando uma função mutex e, quando terminar de executar o código na seção crítica, deve "abandonar" o mutex, chamando uma função de desbloqueio.

Enquanto um thread "possui" o bloqueio – isto é, retornou de uma chamada para a função de bloqueio, mas ainda não chamou a função de desbloqueio –, qualquer outro thread que tentar executar o código na seção crítica aguardará em sua chamada para a função de bloqueio.

Para garantir que nosso código funcione corretamente, podemos modificá-lo para que se pareça com isto:

```
val = Compute_val(my_rank);
Lock(&add my_val_lock);
x += val;
Unlock(&add my_val_lock);
```

Assim, garante-se que apenas um thread de cada vez possa executar a instrução `x + = val`. Observe que o código não impõe

nenhuma ordem predeterminada nos threads que executam o código. Tanto o thread `0` quanto o thread 1 podem executar `x += val` primeiro.

Observe também que o uso de um mutex impõe a serialização da seção crítica. Como apenas um thread por vez pode executar o código na seção crítica, esse código é efetivamente serial. Portanto, queremos que nosso código tenha o menor número possível de seções críticas, e que nossas seções críticas sejam as mais curtas possíveis.

> Existem alternativas para seções críticas além do mutex. Na espera ocupada, por exemplo, um thread entra em um loop cujo único propósito é testar uma condição.

Suponha que haja uma variável compartilhada ok que foi inicializada como `falsa`. Então, algo como o código a seguir pode garantir que o thread 1 não atualize `x` até que o thread `0` o tenha atualizado:

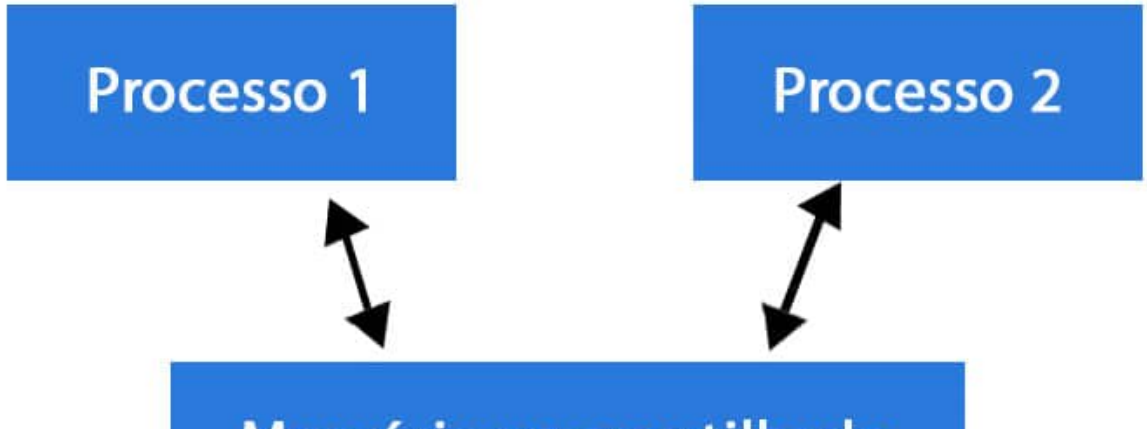
```
val = Compute_val(my_rank);
while (ok != 1); // Espera ocupada
ok = 0 // Bloqueia a entrada de outros thread
x += my val; // Executa a seção crítica
```

```
ok = 1 // Desbloqueia a variável
```

Até que o thread `0` seja executado, a variável ok passa para o valor `0`, bloqueando a entrada de outros threads. Assim, o thread 1 ficará preso no loop while (`ok != 1`); – chamado de "espera ocupada", porque o thread pode estar muito ocupado esperando a condição.

> **Saiba mais**: Simples de entender e implementar, isso, no entanto, pode ser um grande desperdício de recursos do sistema, porque mesmo quando um encadeamento não está fazendo nenhum trabalho útil, o núcleo que o executa verifica repetidamente se a seção crítica pode ser acessada.

MEMÓRIA COMPARTILHADA



Memória compartilhada

Tratamos, até aqui, do uso simplificado de memória compartilhada utilizando threads dentro de um mesmo processo. Entretanto, é possível fazer o uso de memória compartilhada usando processos diferentes, como você pode observar na imagem a seguir:

Sabemos que, para nos comunicarmos entre dois ou mais processos, usamos memória compartilhada. Mas, antes de usar a memória compartilhada, o que precisa ser feito com as chamadas de sistema, tendo uma sequência de operações para isso?

1. Devemos criar o segmento de memória compartilhada. No Linux, isso pode ser feito com a chamada `shmget()`.
2. Devemos, posteriormente, anexar o segmento de memória anteriormente criado ao nosso programa com a chamada `shmget()`.
3. Fazemos nossas operações sobre a memória, como escrita e leitura, com a chamada `shmget()`.
4. Vale ressaltar que devemos ter todos os cuidados necessários, decorrentes da programação concorrente, citados neste conteúdo.

5. Por fim, devemos desanexar o segmento de memória compartilhada de nosso programa com a chamada `shmget()`.

   O uso de memória compartilhada em processos distintos também está disponível nos diversos sistemas operacionais, como, por exemplo, o Windows.

- https://player.vimeo.com/video/558216109

CONDIÇÕES DE CORRIDA

Uma **condição de corrida** ou **risco de corrida**, como vimos anteriormente, é a condição na qual o comportamento do sistema depende da sequência ou do tempo de outros eventos incontroláveis.

Torna-se um **bug** quando um ou mais dos comportamentos possíveis são indesejáveis.

Uma **condição de corrida** surge no software quando um programa de computador, para operar corretamente, depende da sequência ou do tempo dos processos ou threads do programa.

> **Saiba mais**: **Condições críticas de corrida** – que frequentemente acontecem quando os processos ou threads dependem de algum

> estado compartilhado – causam execução inválida e bugs de software. As operações em estados compartilhados são feitas em seções críticas que devem ser mutuamente exclusivas. O não cumprimento dessa regra pode corromper o estado compartilhado.

Uma condição de corrida pode ser difícil de reproduzir e depurar, pois o resultado não é determinístico e depende do tempo relativo entre threads interferentes. Problemas dessa natureza podem, então, desaparecer durante a execução no modo de depuração, adicionando log extra ou anexando um depurador. Os erros que desaparecem durante as tentativas de depuração são geralmente chamados de *heisenbug*. Portanto, é melhor evitar condições de corrida por meio de um projeto de software cuidadoso.

Suponha que cada um de dois threads incremente o valor de uma variável inteira global em 1. Idealmente, a seguinte sequência de operações ocorreria:

Thread 1	**Thread 2**		**Valor da variável**
			0
Ler valor		←	0

Incrementar valor			0
Escrever valor		→	1
	Ler valor	←	1
	Incrementar valor		1
	Escrever valor	→	2

Nesse caso, o valor final é 1, em vez do resultado correto de 2. Isso ocorre porque aqui as operações de incremento não são **mutuamente exclusivas**.

> **Mutuamente exclusivas** operações mutuamente exclusivas são aquelas que não podem ser interrompidas durante o acesso a algum recurso, como um local de memória.
>
> **Comentário**: Nem todos consideram as corridas de dados como um subconjunto das condições de corrida. A definição precisa de corrida de dados é específica para o modelo de simultaneidade formal que está sendo usado.

Normalmente, porém, refere-se a uma situação na qual uma operação de memória em um encadeamento de threads pode

tentar acessar um local de memória ao mesmo tempo que uma operação de memória em outro encadeamento está escrevendo para esse local da memória, em um contexto no qual isso é perigoso. Isso implica que uma corrida de dados é diferente de uma condição de corrida, pois é possível haver não determinismo devido ao tempo, mesmo em um programa sem corrida de dados. Por exemplo, em um programa no qual todos os acessos à memória usam apenas **operações atômicas**.

Isso pode ser perigoso, porque, em muitas plataformas, se dois threads gravam em um local da memória ao mesmo tempo, é possível que o local da memória acabe mantendo um valor que é uma combinação arbitrária e sem sentido dos bits que representam os valores que cada thread estava tentando escrever. Com isso, pode haver uma corrupção de memória, se o valor resultante for um que nenhum thread tentou gravar (às vezes, isso é chamado de "**gravação interrompida**").

Da mesma forma, se um thread lê um local enquanto outro thread está gravando nele, é possível que a leitura retorne um valor que é uma combinação arbitrária e sem sentido dos bits que representam o valor que o local da memória mantinha antes da gravação e dos bits que representam o valor que está sendo escrito.

> **Atenção**: Em muitas plataformas, **operações de memória especiais** são fornecidas para **acesso simultâneo**; nesses casos, normalmente o acesso simultâneo é seguro, mas o acesso simultâneo usando outras operações de memória é perigoso. Às vezes, essas **operações especiais** (que são seguras para acesso simultâneo) são chamadas de **operações atômicas** ou de **sincronização**, enquanto as **operações comuns** (que não são seguras para acesso simultâneo) são chamadas de **operações de dados**, sendo esse termo denominado de **corrida de dados**.

SINCRONIZAÇÃO DE PROCESSOS (Operações Atômicas)

A **sincronização** é a tarefa de coordenar a execução de processos, de forma que dois deles não possam ter acesso aos mesmos dados e recursos compartilhados.

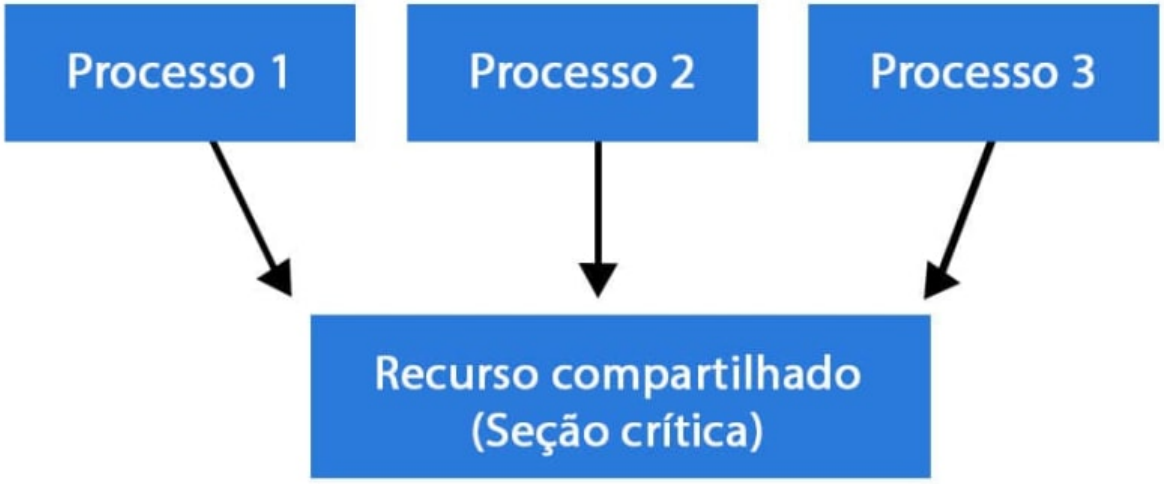
> **Atenção**: É especialmente necessária em um sistema de vários processos, quando vários deles estão sendo executados juntos e mais de um processo tenta obter, simultaneamente, acesso ao mesmo recurso ou dado compartilhado, o que pode levar à inconsistência de dados.

Isso porque a mudança feita por um processo não se reflete necessariamente quando outros acessam os mesmos dados compartilhados. Por exemplo: o processo A altera os dados em um local da memória enquanto outro processo B tenta ler os dados do mesmo local da memória. Há grande probabilidade de que os dados lidos pelo segundo sejam errôneos. **Portanto, para evitar esse tipo de inconsistência de dados, os processos precisam ser sincronizados entre si**.

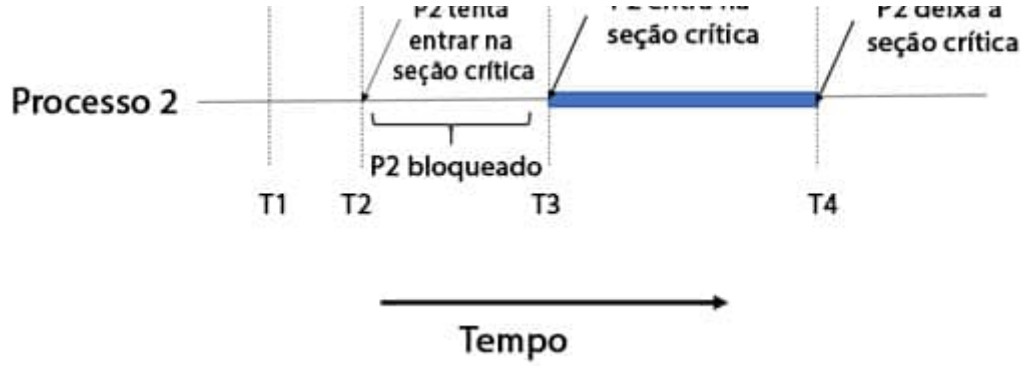
> A **sincronização de threads** é definida como um mecanismo que garante que dois ou mais processos ou threads simultâneos não executem simultaneamente algum segmento de programa específico conhecido como **seção crítica**.

O **acesso dos processos** à seção crítica é controlado por meio de técnicas de sincronização. Quando um thread começa a executar a seção crítica (segmento serializado do programa), o outro thread deve esperar até que o primeiro termine. Se as técnicas de sincronização adequadas não forem aplicadas, isso pode causar uma condição de corrida na qual os valores das variáveis podem ser imprevisíveis, dependendo dos tempos de troca de contexto dos processos ou threads.

Suponha que haja três processos: 1, 2 e 3. Todos os três estão executando simultaneamente e precisam compartilhar um recurso comum (seção crítica), conforme mostrado na imagem a seguir:



A sincronização deve ser usada aqui para evitar quaisquer conflitos para acessar esse recurso compartilhado. Portanto, quando os Processos 1 e 2 tentam acessar esse recurso, ele deve ser atribuído a apenas um processo por vez. Se for atribuído ao Processo 1, o Processo 2 precisa esperar até que o primeiro libere esse recurso. Veja na próxima imagem:



Precisamos satisfazer quatro condições para chegar a uma solução:

1. Dois processos nunca podem estar simultaneamente em suas regiões críticas.
2. Nada pode ser afirmado sobre a velocidade ou sobre o número de CPUs.
3. Nenhum processo executando fora de sua seção crítica pode bloquear outros processos.
4. Nenhum processo deve esperar eternamente para entrar em sua seção crítica.

Outro requisito de sincronização a ser considerado é a ordem na qual determinados processos ou threads devem ser executados. Não é possível, por exemplo, embarcar em um avião antes de comprar uma passagem ou verificar e-mails antes de validar as

credenciais apropriadas (nome de usuário e senha). Da mesma forma, um caixa eletrônico não fornecerá nenhum serviço até que receba um PIN correto.

Além da exclusão mútua, a sincronização também lida com o seguinte:

Deadlock (impasse)

> Conforme veremos mais detalhadamente adiante, deadlocks ocorrem quando muitos processos estão esperando por um recurso compartilhado (seção crítica) que está sendo mantido por algum outro processo. Nesse caso, os processos simplesmente continuam esperando e não são mais executados. Deadlocks também podem ocorrer entre máquinas.
>
> Por exemplo, diversos escritórios têm redes locais com vários computadores conectados entre elas. Muitas vezes, dispositivos como scanners e impressoras são conectadas a essas redes como recursos compartilhados, disponíveis a qualquer usuário em qualquer máquina. Se esses dispositivos puderem ser reservados remotamente, o mesmo tipo de deadlock poderá ocorrer, como descrito anteriormente.

Starvation (fome)

> Ocorre quando um processo está esperando para entrar na seção crítica, mas outros processos a monopolizam, e o primeiro é forçado a esperar indefinidamente.

Inversão de prioridade

> Ocorre quando um processo de alta prioridade está na seção crítica e é interrompido por um de média prioridade. Essa violação das regras de prioridade pode acontecer em certas circunstâncias, e pode levar a sérias consequências em sistemas de tempo real.

Espera ocupada

> Ocorre quando um processo pesquisa frequentemente para determinar se tem acesso a uma seção crítica. Essa pesquisa frequente rouba o tempo de processamento de outros processos.

Um dos desafios para o projeto do algoritmo em ambientes distribuídos é minimizar ou reduzir a sincronização – que pode levar mais tempo do que a computação, especialmente na computação distribuída. A redução da sincronização atraiu a atenção dos cientistas da computação por décadas e, à medida

que a lacuna entre a melhoria da computação e a latência aumenta, isso se torna um problema cada vez mais significativo.

PROBLEMAS CLÁSSICOS DE SINCRONIZAÇÃO

A seguir, alguns problemas clássicos de sincronização:

PROBLEMA PRODUTOR-CONSUMIDOR (PROBLEMA DE BUFFER LIMITADO)

Nesse problema, há um **produtor** (que está produzindo algo) e um **consumidor** (que está consumindo esses produtos produzidos). Os **produtores** e **consumidores** compartilham o mesmo buffer de memória de tamanho fixo.

O trabalho do **produtor é gerar os dados, colocá-los no buffer** (Em ciência da computação, buffer de dados é uma região de memória física utilizada para armazenar temporariamente os dados enquanto eles estão sendo movidos de um lugar para outro.) e, novamente, começar a gerar dados. Enquanto o trabalho do **consumidor é consumir os dados do buffer**.

O produtor deve produzir dados apenas quando o buffer não estiver cheio. Se estiver cheio, o produtor não deve ter permissão

para colocar nenhum dado no buffer.

O consumidor, por sua vez, deve consumir dados apenas quando o buffer não está vazio. Se estiver vazio, o consumidor não deve ter permissão para retirar nenhum dado do buffer. O produtor e o consumidor não devem acessar o buffer ao mesmo tempo.

PROBLEMA DOS LEITORES-ESCRITORES

O problema dos leitores-escritores está relacionado a um objeto, como um arquivo que é compartilhado entre vários processos. Alguns desses processos são **leitores** (querem apenas ler os dados do objeto) e alguns são **escritores** (querem escrever no objeto).

O problema dos leitores-escritores é usado para gerenciar a sincronização, de forma que não haja intercorrências com os dados do objeto.

Se dois leitores, por exemplo, acessarem o objeto ao mesmo tempo, não há problema. No entanto, se dois escritores ou um leitor e um escritor o acessarem ao mesmo tempo, pode haver problemas. Para resolver essa situação, um escritor deve obter acesso exclusivo a um objeto. Ou seja, quando um escritor está

acessando o objeto, nenhum leitor ou escritor pode acessá-lo. No entanto, vários leitores podem acessar o objeto ao mesmo tempo.

PROBLEMA DO JANTAR DOS FILÓSOFOS

O problema do jantar dos filósofos afirma que há cinco filósofos compartilhando uma mesa circular, e eles comem e pensam alternativamente. Há uma tigela de arroz para cada um dos filósofos, além de cinco hashis. Um filósofo precisa de dois hashis para comer. Um filósofo faminto só pode comer se houver os dois talheres disponíveis. Do contrário, ele abaixa o talher e começa a pensar novamente. O grande problema nesse algoritmo é que, se todos os filósofos pegarem somente um hashi, todos ficarão parados para sempre, aguardando o segundo talher ficar disponível, gerando um **deadlock** (ou impasse).

Esses problemas são usados para testar quase todos os esquemas de sincronização ou primitivos recentemente propostos.

EXCLUSÃO MÚTUA

Na ciência da computação, a **exclusão mútua** é uma propriedade do controle de concorrência, instituída com o objetivo de prevenir

condições de corrida.

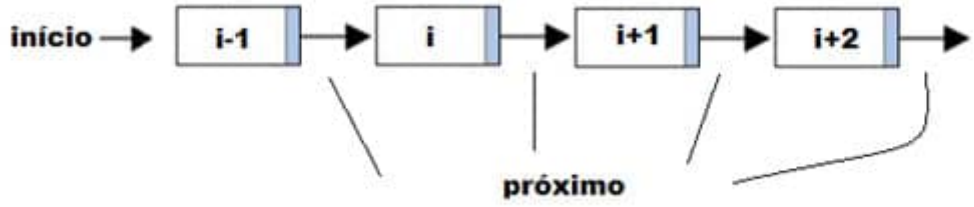
É o requisito de que um thread de execução nunca entre em uma seção crítica enquanto um thread simultâneo de execução já a está acessando. Refere-se a um intervalo de tempo durante o qual um thread de execução acessa um recurso compartilhado, como memória e objetos de dados compartilhados.

> **Comentário**: Conforme vimos, a seção crítica é um objeto de dados que dois ou mais threads simultâneos estão tentando modificar (no qual duas operações de leitura simultâneas são permitidas, mas não são permitidas duas operações de gravação simultâneas ou uma leitura e uma gravação, pois isso leva à inconsistência dos dados).

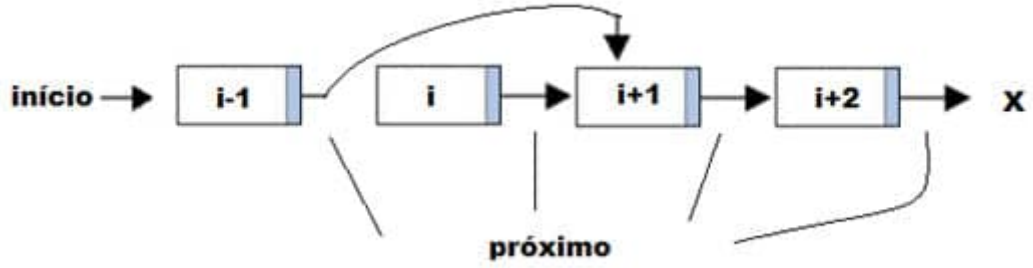
O **algoritmo de exclusão** garante, portanto, que, se um processo já estiver executando a operação de gravação em um objeto de dados, nenhum outro tem permissão para acessar e modificar o mesmo objeto ou seção crítica até que o primeiro tenha concluído a gravação e liberado o objeto para outros processos lerem e escreverem. Em outras palavras, a seção crítica passa a ser de uso exclusivo do processo que a acessou inicialmente.

Um exemplo simples de porque a exclusão mútua é importante na

prática pode ser visualizado usando uma lista unida de quatro itens, na qual o segundo e o terceiro devem ser removidos, como pode ser exemplificado na imagem a seguir: `Lista unida de quatro itens`.



A remoção de um nó que fica entre dois outros é realizada mudando o ponteiro próximo do nó anterior para apontar para o próximo nó. Em outras palavras, se o nó está sendo removido, então o ponteiro próximo de nó é alterado para apontar para o nó , removendo assim da lista encadeada qualquer referência ao nó . Observe a imagem a seguir: `Lista unida após movimentação do ponteiro próximo de n-`.



Quando tal lista encadeada está sendo compartilhada entre vários threads de execução, dois threads de execução podem tentar remover dois nós diferentes simultaneamente – um thread de execução mudando o ponteiro próximo do nó `i - 1` para apontar para o nó `i + 1`, enquanto outro thread de execução muda o ponteiro próximo do nó `i` para apontar para o nó `i + 2`. Embora ambas as operações de remoção sejam concluídas com sucesso, o estado desejado da lista vinculada não é alcançado: nó `i + 1` permanece na lista, porque o ponteiro próximo do nó `i - 1` aponta para o nó `i + 1`.

Esse problema (chamado de condição de corrida) pode ser evitado usando o requisito de exclusão mútua para garantir que atualizações simultâneas na mesma parte da lista não ocorram.

> **Saiba Mais**: O termo exclusão mútua também é usado em referência à gravação simultânea de um endereço de memória por um thread/processo, enquanto o endereço de memória mencionado anteriormente está sendo manipulado ou lido por um ou mais threads/processos.

O problema que a exclusão mútua aborda é de compartilhamento

de recursos:

> Como um sistema de software pode controlar o acesso de vários processos a um recurso compartilhado, quando cada processo precisa do controle exclusivo desse recurso enquanto faz seu trabalho?

Para isso, a solução de exclusão mútua torna o recurso compartilhado disponível apenas enquanto o processo está em um segmento de código específico (seção crítica), com acesso exclusivo, impedindo que outros processos acessem essa área de código. Ele controla o acesso ao recurso compartilhado controlando cada execução mútua daquela parte do programa na qual o recurso seria usado.

Uma solução bem-sucedida para esse problema deve ter pelo menos duas propriedades:

- **Deve implementar a exclusão mútua**: Apenas um processo pode estar na seção crítica de cada vez.
- **Deve estar livre de deadlocks**: Se os processos estão tentando entrar na seção crítica, um deles deve ser capaz de fazê-lo com sucesso, desde que nenhum processo permaneça na seção crítica permanentemente.

A liberdade de deadlock pode ser expandida para implementar uma ou ambas as propriedades. A liberdade de bloqueio garante que qualquer processo que deseje entrar na seção crítica será capaz de fazê-lo eventualmente. Isso é diferente da prevenção de deadlock, que requer que algum processo em espera seja capaz de obter acesso à seção crítica, mas não requer que cada processo tenha sua vez. Se dois processos trocarem continuamente um recurso entre eles, um terceiro processo pode ser bloqueado e ficar sem recursos, mesmo que o sistema não esteja em um impasse. Se um sistema estiver livre de bloqueios, ele garante que todos os processos possam ser revertidos em algum ponto no futuro.

Uma propriedade de espera limitada por k fornece um compromisso mais preciso do que a liberdade de bloqueio. Isso porque, na liberdade de bloqueio, não dá nenhuma garantia sobre o tempo de espera. Um processo pode ser ultrapassado um número arbitrário ou ilimitado de vezes por outros de prioridade mais alta antes de chegar a sua vez. Sob uma propriedade de espera limitada por k, cada processo tem um tempo de espera máximo finito. Isso funciona estabelecendo um limite para o número de vezes que outros processos podem interromper a fila,

de modo que nenhum deles possa entrar na seção crítica mais de k vezes enquanto outro está esperando.

O programa de cada processo pode ser dividido em quatro seções, resultando em quatro estados. A execução do programa percorre esses quatro estados, em ordem:

1. **Seção não crítica**: A operação está fora da seção crítica; o processo não está usando ou solicitando o recurso compartilhado.
2. **Tentando**: A operação está fora da seção crítica; o processo não está usando ou solicitando o recurso compartilhado.
3. **Seção crítica**: O processo tem permissão para acessar o recurso compartilhado nessa seção.
4. **Saída**: O processo sai da seção crítica e disponibiliza o recurso compartilhado para outros processos.

   Se um processo deseja entrar na seção crítica, ele deve primeiro executar a seção de tentativa e esperar até obter acesso à seção crítica. Após ter executado sua seção crítica e finalizado com os recursos compartilhados, ele precisa executar a seção de saída para liberá-los para uso de outros processos. O processo então retorna à sua seção não crítica.

SOS SEÇÃO CRÍTICA (Região)

Conforme vimos, na programação simultânea, os acessos simultâneos a recursos compartilhados podem levar a um comportamento inesperado ou errôneo. Portanto, partes do programa onde o recurso compartilhado é acessado precisam ser protegidas de maneiras que evitem o acesso simultâneo. Essa seção protegida – chamada de **região** ou **seção crítica** – não pode ser executada por mais de um processo por vez.

Normalmente, a seção crítica acessa um recurso compartilhado, como uma estrutura de dados, um dispositivo periférico ou uma conexão de rede, que não operaria corretamente no contexto de vários acessos simultâneos.

> **Comentário**: Códigos ou processos diferentes podem consistir na mesma variável ou em outros recursos que precisam ser lidos ou escritos, mas cujos resultados dependem da ordem em que as ações ocorrem.

Por exemplo, se uma variável x deve ser lida pelo processo A e o processo B precisa gravar na mesma variável x ao mesmo tempo, o processo A pode obter o valor antigo ou o novo valor de x.

```
//Processo a
b = x + 5 // Instrução executada em Tx
//Processo b
x = 3 + z // Instrução executada em Tx
```

Em casos como esses, uma seção crítica é importante. No exemplo anterior, se A precisar ler o valor atualizado de x, executar o Processo A e o Processo B ao mesmo tempo pode não dar os resultados necessários. Para evitar isso, a variável x é protegida por uma seção crítica. Primeiramente, B obtém o acesso à seção. Assim que B terminar de escrever o valor, A terá acesso à seção crítica e a variável x poderá ser lida.

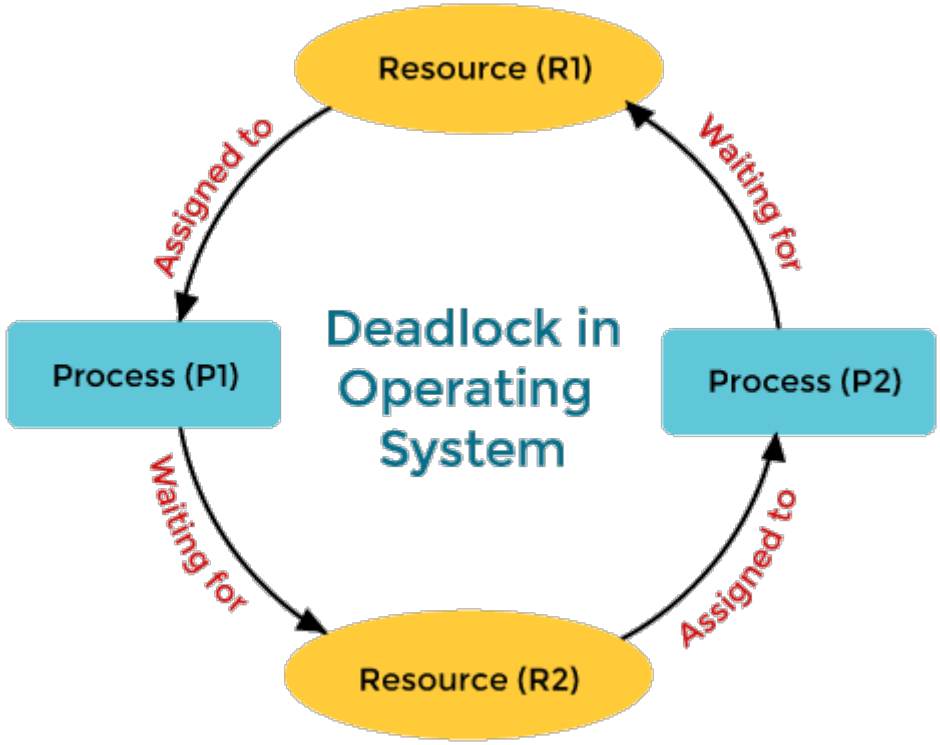
Controlando cuidadosamente quais variáveis são modificadas dentro e fora da seção crítica, o acesso simultâneo à variável compartilhada é evitado.

> Uma seção crítica é normalmente usada quando um programa multithreaded deve atualizar várias variáveis relacionadas sem que um thread separado faça alterações conflitantes nesses dados.

Em uma situação relacionada, uma seção crítica pode ser usada

para garantir que um recurso compartilhado – por exemplo, uma impressora – só possa ser acessado por um processo por vez.

🚙🚙 DEADLOCK (impasse)



Como vimos, na computação simultânea, um **deadlock** é um estado em que cada membro de um grupo espera que outro membro, incluindo ele mesmo, execute uma ação – como enviar uma mensagem ou, mais comumente, liberar um bloqueio.

> **Atenção**: Deadlock são problemas comuns em sistemas de multiprocessamento (sistemas operacionais), banco de dados, computação paralela e sistemas distribuídos, em que bloqueios de software e hardware são usados para arbitrar recursos compartilhados e implementar sincronização de processos.

Em um sistema operacional, um deadlock ocorre quando um processo ou thread entra em um estado de espera porque um recurso de sistema solicitado é mantido por outro processo em espera – que, por sua vez, está esperando por outro recurso mantido por outro processo em espera.

> Se um processo é incapaz de mudar seu estado indefinidamente porque os recursos solicitados por ele estão sendo usados por outro processo em espera, então o sistema é considerado um deadlock.

Em um sistema de comunicação, os deadlock ocorrem principalmente devido a sinais perdidos ou corrompidos, em vez de contenção de recursos. Uma situação de deadlock em um recurso pode surgir se e somente se todas as seguintes condições, conhecidas como as quatro condições de Coffman et al. 1971, forem mantidas simultaneamente em um sistema:

- **Exclusão mútua**: Como vimos anteriormente, pelo menos um recurso deve ser mantido em um modo não compartilhável. Caso contrário, os processos não seriam impedidos de usar o recurso quando necessário. Apenas um processo pode usar o recurso em um determinado instante de tempo.
- **Reter e aguardar ou retenção de recurso**: Um processo está atualmente retendo pelo menos um recurso e solicitando recursos adicionais que estão sendo retidos por outros processos.
- **Sem preempção**: Um recurso só pode ser liberado voluntariamente pelo processo que o detém.
- **Espera circular**: Cada processo deve estar esperando por um recurso que está sendo retido por outro processo, que, por sua vez, está aguardando que o primeiro processo libere o recurso. Em geral, existe um conjunto de processos de espera, `P = {P1, P2, ..., PN}`, tal que `P1` está esperando por um recurso mantido por `P2`, `P2` está esperando por um recurso mantido por `P3` e assim por diante, até que `PN` seja esperando por um recurso mantido por `P1`.

> Um deadlock ocorre se e somente se as quatro condições de Coffman forem satisfeitas:

1. Exclusão mútua.
2. Posse e espera.
3. Não preempção.
4. Espera circular.

Embora essas condições sejam suficientes para produzir um conflito em sistemas de recursos de instância única, elas indicam apenas a possibilidade de conflito em sistemas com várias instâncias de recursos.

A maioria dos sistemas operacionais atuais não pode evitar bloqueios. Quando ocorre um deadlock, diferentes sistemas operacionais respondem a eles de maneiras diferentes e fora do padrão. A maioria das abordagens funciona evitando que uma das quatro condições de Coffman ocorra, especialmente a espera circular.

As principais abordagens são:

- **IGNORANDO IMPASSE**: Nessa abordagem, presume-se que nunca ocorrerá um deadlock. É usada quando os intervalos de tempo entre as ocorrências de deadlocks são grandes e a perda de dados incorrida a cada vez é tolerável. Ignorar deadlocks pode ser

feito com segurança se estes forem formalmente comprovados como nunca ocorrendo.

- **DETECÇÃO**: Na detecção de deadlocks, presumimos que estes podem ocorrer. Em seguida, o estado do sistema é examinado para detectar se ocorreu um conflito e, posteriormente, corrigi-lo. É empregado um algoritmo que rastreia a alocação de recursos e os estados do processo, reverte e reinicia um ou mais processos para remover o impasse detectado. Detectar um deadlock que já ocorreu é facilmente possível, uma vez que os recursos que cada processo bloqueou e/ou solicitou atualmente são conhecidos pelo escalonador de recursos do sistema operacional.

  Depois que um deadlock é detectado, ele pode ser corrigido usando um dos seguintes métodos:

Encerramento do processo

Um ou mais processos envolvidos no deadlock podem ser abortados. Pode-se optar por abortar todos os processos concorrentes envolvidos no impasse, garantindo que o impasse seja resolvido com certeza e velocidade. Entretanto, eventuais processamentos anteriores podem ser perdidos.

O encerramento de processos pode seguir uma sequência, até que o impasse seja resolvido. Essa abordagem, porém, tem alto custo, porque após cada encerramento de processo/thread, um algoritmo deve determinar se o sistema ainda está em deadlock.

Vários fatores devem ser considerados ao escolher um candidato para o encerramento, como prioridade e tempo de execução do processo.

Preempção de recursos

Os recursos alocados a vários processos podem ser sucessivamente eliminados e alocados a outros processos até que o impasse seja resolvido.

A prevenção de deadlock funciona evitando que uma das quatro condições de Coffman ocorra. Remover a condição de exclusão mútua significa que nenhum processo terá acesso exclusivo a um recurso. Isso é impossível para recursos que não podem ser armazenados em **spool**. Mas, mesmo com recursos em spool, o impasse ainda pode ocorrer. Os algoritmos que evitam a exclusão mútua são chamados de **algoritmos de sincronização sem bloqueio**.

> **Spool** é o processo que transfere dados colocando-os em uma área de trabalho temporária, na qual outro programa pode acessá-lo para processá-lo em um tempo futuro.
>
> **Comentário**: As condições de retenção e espera ou retenção de recurso podem ser removidas exigindo que os processos solicitem todos os recursos de que precisarão antes de iniciar (ou antes de iniciar um determinado conjunto de operações). Esse conhecimento prévio é frequentemente difícil de satisfazer e, em qualquer caso, é um uso ineficiente de recursos.

Outra maneira é exigir que os processos solicitem recursos apenas quando não houver nenhum. Primeiro, eles devem liberar todos os seus recursos atualmente mantidos antes de solicitar todos os recursos de que precisarão do zero. Muitas vezes, isso também é impraticável, porque os recursos podem ser alocados e permanecer sem uso por longos períodos. Além disso, um processo que requer um recurso popular pode ter que esperar indefinidamente, já que este pode sempre ser alocado para algum processo, resultando em escassez de recursos. Esses algoritmos, como tokens de serialização, são conhecidos como **algoritmos tudo ou nada**. Eles devem liberar todos os seus recursos atualmente mantidos antes de solicitar todos os recursos de que

precisarão do zero. Muitas vezes, isso também é impraticável, porque os recursos podem ser alocados e permanecer sem uso por longos períodos. Além disso, um processo que requer um recurso popular pode ter que esperar indefinidamente, já que este pode sempre ser alocado para algum processo, resultando em escassez de recursos. Esses algoritmos, como tokens de serialização, são conhecidos como algoritmos tudo ou nada.

A condição de ausência de preempção também pode ser difícil ou impossível de evitar, pois um processo deve ser capaz de ter um recurso por um determinado período de tempo, ou o resultado do processamento pode ser inconsistente. No entanto, a incapacidade de impor a preempção pode interferir com um algoritmo de prioridade.

A preempção de um recurso "bloqueado" geralmente implica uma reversão e deve ser evitada, uma vez que é muito cara em despesas gerais. Os algoritmos que permitem a preempção incluem algoritmos sem bloqueio e sem espera e controle de simultaneidade otimista. Se um processo retém alguns recursos e solicita algum outro que não pode ser alocado imediatamente a eles, a condição pode ser removida liberando todos os recursos retidos atualmente desse processo.

A condição final é a condição de espera circular. Abordagens que evitam esperas circulares incluem desabilitar interrupções durante seções críticas e usar uma hierarquia para determinar uma ordenação parcial de recursos. Se nenhuma hierarquia óbvia existe, até mesmo o endereço de memória dos recursos será usado para determinar a ordem, e os recursos serão solicitados na ordem crescente da enumeração.

Deadlocks distribuídos podem ocorrer em sistemas distribuídos quando transações distribuídas ou controle de simultaneidade estão sendo usados. Eles podem ser detectados pela construção de um gráfico de espera global a partir de gráficos de espera locais em um detector de deadlock ou por um algoritmo distribuído como **perseguição de borda**.

Os sistemas distribuídos podem ter três tipos de deadlocks: **fantasma**, de **recurso** e de **comunicação**. Veja-os a seguir:

Deadlocks fantasmas

São deadlocks falsamente detectados em um sistema distribuído devido a atrasos internos do sistema, mas não existem de fato.

Por exemplo, se um processo libera um recurso R1 e emite uma

solicitação de R2, e a primeira mensagem é perdida ou atrasada, um coordenador (detector de deadlocks) pode concluir falsamente um deadlock (a solicitação de R2 enquanto tem R1 causaria um impasse). Em uma possível situação de deadlock fantasma, é muito difícil detectá-lo, porque vários processos podem liberar alguns recursos ou solicitar outros recursos ao mesmo tempo.

Em alguns casos, as mensagens de liberação chegam depois das mensagens de solicitação para alguns recursos específicos. O **algoritmo de detecção** trata a situação como um deadlock, e pode abortar alguns processos para quebrá-lo. Mas, na realidade, nenhum deadlock estava presente – os algoritmos o detectaram com base em mensagens desatualizadas dos processos devido a atrasos na comunicação. Esse impasse é chamado de **impasse fantasma**.

Deadlocks de recurso

Os processos podem esperar simultaneamente por vários recursos, e não podem prosseguir até que tenham adquirido todos eles. Um conjunto de processos fica em deadlock se cada processo no conjunto solicitar recursos mantidos por outro processo no conjunto. E deve receber todos os recursos solicitados

antes que possa ser desbloqueado. Esse tipo de deadlock é muito parecido com os que observamos anteriormente.

Deadlocks de comunicação

Os processos esperam para se comunicar com outros em um conjunto. Um processo de espera pode ser desbloqueado ao receber uma comunicação de um desses processos.

Há conflito se cada processo no conjunto estiver esperando para se comunicar com outro e nenhum deles iniciar qualquer comunicação adicional até que receba a comunicação pela qual está esperando.

Assista ao vídeo a seguir e conheça as quatro condições necessárias para que ocorra o problema de deadlock.

🚦 Semáforo

Na ciência da computação, um **semáforo** é um tipo de dado variável ou abstrato empregado para controlar o acesso a um recurso comum por vários processos e evitar problemas de seção crítica em um sistema simultâneo, como um sistema operacional multitarefa. Um **semáforo trivial** é uma variável simples que é alterada (por exemplo, incrementada ou decrementada ou alternada), dependendo das condições definidas pelo programador.

> Em ciência da computação, semáforo é uma variável especial protegida que tem como função o controle de acesso a recursos compartilhados num ambiente multitarefa. A invenção desse tipo de variável é atribuída a Edsger Dijkstra, em 1965 e foi utilizado inicialmente no sistema operacional THEOS.
>
> **Atenção**: Os semáforos são uma ferramenta útil na prevenção de condições de corrida; entretanto, seu uso não é de forma alguma uma garantia de que um programa esteja livre desses problemas.

Os semáforos que permitem uma contagem arbitrária de recursos são chamados de **semáforos de contagem**, enquanto os que são restritos aos valores 0 e 1 (ou bloqueado/desbloqueado,

indisponível/disponível) são chamados de **semáforos binários** e são usados para implementar bloqueios.

Exemplo:

Suponha que uma biblioteca tenha dez salas de estudo idênticas, para serem usadas por um aluno por vez. Os alunos devem solicitar na recepção, caso desejem usar uma sala de estudo e, se nenhuma estiver disponível, eles esperam na mesa. Quando um aluno terminar de usar uma sala, ele deve retornar à carteira e indicar que uma das salas ficou livre.

Na implementação mais simples, o funcionário da recepção sabe apenas o número de salas livres disponíveis. Quando um aluno solicita uma sala, o funcionário diminui esse número. Quando um aluno libera uma sala, o secretário aumenta esse número. As salas podem ser usadas por tempo indeterminado, por isso não é possível reservá-las com antecedência.

Nesse cenário, o contador da recepção representa um semáforo de contagem, as salas são o recurso e os alunos representam processos/threads. O valor do semáforo é inicialmente 10, com todas as salas vazias. Quando um aluno solicita uma sala, o valor do semáforo é alterado para 9. Depois que o próximo aluno chega,

ele cai para 8, depois para 7 e assim por diante. Se alguém solicita uma sala e o valor atual do semáforo é 0, eles são forçados a esperar até que uma delas esteja disponível (quando a contagem é aumentada de 0).

Se uma das salas foi liberada, mas há vários alunos esperando, então qualquer método pode ser usado para selecionar aquele que irá ocupá-la. E os alunos precisam informar ao balconista sobre a liberação de sua sala somente depois de realmente deixá-la; caso contrário, pode haver uma situação embaraçosa quando este está em processo de deixar a sala (está empacotando seus livros etc.) e outro aluno entra antes de ele sair.

Os semáforos são muito úteis na sincronização de processos e multithreading. Temos a biblioteca de semáforo **POSIX** em sistemas Linux, e vamos aprender a usá-lo.

> Salientamos que semáforos existem nas mais diversas linguagens que permitem programação paralela, como por exemplo Java, bem como em diferentes sistemas operacionais.

O código básico de um semáforo é simples, conforme apresentado em nosso estudo. Mas esse código não pode ser escrito diretamente, pois as funções precisam ser atômicas, e escrever o

código diretamente levaria a uma troca de contexto sem a conclusão da função e resultaria em uma bagunça.

Para bloquear um semáforo ou esperar, podemos usar a função `sem_wait`:

```
int sem_wait(sem_t *sem);
```

Para liberar ou sinalizar um semáforo, usamos a função `sem_post`:

```
int sem_post(sem_t *sem);
```

Um semáforo é inicializado usando `sem_init` (para processos ou threads) ou `sem_open` (para IPC).

```
sem_init(sem_t *sem, int pshared, unsigned int value);
```

SEÇÃO CRÍTICA: TÉCNICAS DE SINCRONIZAÇÃO

Como vimos no módulo anterior, o acesso dos processos à seção

crítica (segmento serializado do programa) é controlado pelo uso de técnicas de sincronização. Quando um thread começa a executar a seção crítica, o outro thread deve esperar até que o primeiro thread termine.

Se as técnicas de sincronização adequadas não forem aplicadas, isso pode causar uma condição de corrida em que os valores das variáveis podem ser imprevisíveis e variam dependendo dos tempos de alternância de contexto dos processos ou threads.

Vejamos um exemplo de código para estudar problemas de sincronização por meio de uma seção crítica:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
pthread_t tid[2];
int counter;
pthread_mutex_t lock;
void * trythis(void* arg)
{
```

```
    pthread_mutex_lock(&lock);
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);
    for (i = 0; i < (0xFFFFFFFF); i++);
    printf("\n Job %d has finished\n", counter);
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(void)
{
    int i = 0;
    int error;

    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("\n mutex init has failed\n");
        return 1;
    }

    while (i < 2) {
```

```
        error = pthread_create(&(tid[i]), NULL,
&trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created :
[%s]",
                   strerror(error));
        i++;
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

No código anterior, percebemos que:

- Um mutex é inicializado no início da função principal.
- O mesmo mutex está bloqueado na função `trythis ()` ao usar o recurso compartilhado ‘contador’.
- No final da função `trythis ()`, o mesmo mutex é desbloqueado.

- No final da função principal, quando ambas os threads estão concluídos, o mutex é destruído.

  Resultado:

  Job 1 started
  Job 1 finished
  Job 2 started
  Job 2 finished

🚨 MONITOR

Para superar os erros de tempo que ocorrem ao usar o semáforo para sincronização de processos, foi introduzida uma construção de sincronização de alto nível denominada monitor. Um tipo de monitor é um tipo de dados abstrato usado para sincronização de processos.

Sendo um tipo de dados abstrato, o tipo de monitor contém as variáveis de dados que devem ser compartilhados por todos os processos e algumas operações definidas pelo programador, que permitem que os processos sejam executados em exclusão mútua dentro do monitor.

Um processo não pode acessar diretamente a variável de dados

compartilhados no monitor. Ele deve acessá-lo por meio de procedimentos definidos no monitor que permitem que apenas um processo acesse as variáveis compartilhadas em um monitor por vez. Ou seja, um monitor é uma construção em que apenas um processo está ativo por vez. Se outro processo tentar acessar a variável compartilhada no monitor, ele será bloqueado e será alinhado na fila para obter o acesso.

A sintaxe do monitor é a seguinte:

```
monitor monitor_name
{
    //shared variable declarations
    procedure P1 ( . . . ) {
    }
    procedure P2 ( . . . ) {
    }
    procedure Pn ( . . . ) {
    }
    initialization code ( . . . ) {
    }
}
```

Nos monitores, há a introdução do conceito de variáveis condicionais como um mecanismo de sincronização adicional. A variável condicional permite que um processo aguarde dentro do monitor e que um processo de espera seja retomado imediatamente quando o outro libera os recursos.

> **Atenção**: A variável condicional pode invocar apenas duas operações, `wait ()` e `signal ()`. Se um processo P invocar uma operação `wait ()`, ele é suspenso no monitor até que outro processo Q invoque a operação `signal ()`; ou seja, uma operação `signal ()` invocada por um processo retoma o processo suspenso.

DIFERENÇA ENTRE SEMÁFORO E MONITOR

Veja no quadro abaixo algumas diferenças entre semáforo e monitor:

Semáforos	Monitores
O semáforo é uma variável inteira S que indica o número de	O monitor é o tipo de dado abstrato que permite que

recursos disponíveis no sistema	apenas um processo seja executado na seção crítica por vez.
O valor do semáforo pode ser modificado apenas pelas operações `wait()` e `shmget()`	Um monitor, por sua vez, possui as variáveis compartilhadas e os procedimentos apenas por meio dos quais as variáveis compartilhadas podem ser acessadas pelos processos.
No semáforo, quando um processo deseja acessar recursos compartilhados, o processo executa a operação `wait()` e bloqueia os recursos e, quando libera os recursos, executa a operação `signal()`	Em monitores, quando um processo precisa acessar recursos compartilhados, ele deve acessá-los por meio de procedimentos no monitor.

Além disso, o tipo de monitor possui variáveis de condição que o semáforo não possui.

semaforo.c

```
//---------------------------------------------
// Programa em C para demonstrar como os
Semaforos trabalham
//---------------------------------------------
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
//---------------------------------------------
#define SemaforoLigado 1 // constante com o
semáforo ligado
//---------------------------------------------
typedef struct { // estrutura de dados para
identificar o estado do semáforo
    int identificador;
}thread_arg, *ptr_thread_arg;
//---------------------------------------------
sem_t mutex;    //semaforo nao-binario
//---------------------------------------------
void* thread(void* arg)
```

```
{
    //wait
    if(SemaforoLigado) sem_wait(&mutex);

    printf("\nEntrando na Regiao Critica..\n");

    ptr_thread_arg targ = (ptr_thread_arg)arg;

    printf("A - Thread:  [%d]\n", targ-
>identificador);

    sleep(5); //dormir 5 segundos

    printf("B - Thread:  [%d]\n", targ-
>identificador);
    //signal
    printf("\nSaindo da Regiao Critica...\n");

    if(SemaforoLigado) sem_post(&mutex);
}
//-----------------------------------------------
```

```
int main()
{
    thread_arg argumentos[2];
    if(SemaforoLigado) sem_init(&mutex, 0, 1);
//Iniciar semaforo

    pthread_t t[2];

    argumentos[0].identificador=1;
    argumentos[1].identificador=2;


pthread_create(&t[0],NULL,thread,&(argumentos[0]))
//Criar a Thread 1
    sleep(2);
//dormir 2 segundos

pthread_create(&t[1],NULL,thread,&(argumentos[1]))
//Criar a Thread 2
    pthread_join(t[0],NULL);
//Esperar por um Thread 1
```

```
    pthread_join(t[1],NULL);
//Esperar por um Thread 2

    if(SemaforoLigado) sem_destroy(&mutex);
    return 0;
}
//-----------------------------------------------
```

↕️ CORROTINAS (co-rotinas)

As **corrotinas** são componentes de programas de computador que generalizam sub-rotinas para multitarefa não preemptiva, permitindo que a execução seja suspensa e reiniciada. Os sistemas Windows iniciais não eram preemptivos, e o uso de corrotinas permitia que se alternassem dentro dos processos em execução. O uso de corrotinas permite, por exemplo, a não utilização de mecanismos de primitivas de sincronização, como semáforos e mutex.

> As corrotinas foram uma das primeiras formas de permitir a simulação de um ambiente multitarefa, em especial nos sistemas

> operacionais não preemptivos, sendo ainda bastante úteis e simples.

Elas também são componentes de programas de computador, onde os valores dos dados locais para uma co-rotina persistem em chamadas sucessivas e a execução de uma corrotina é suspensa quando o controle a deixa.

As sub-rotinas são casos especiais de corrotinas. Quando são chamadas, a execução começa no início e, quando uma sub-rotina for encerrada, ela é concluída. Uma instância de uma sub-rotina só retorna uma vez, e não mantém o estado entre as invocações.

> As corrotinas são úteis para implementar máquinas de estado.

As corrotinas, por sua vez, podem sair chamando outras corrotinas, que podem posteriormente retornar ao ponto onde foram chamadas na corrotina original. Do seu ponto de vista, ela não está saindo, mas chamando outra corrotina. Assim, uma instância de corrotina mantém o estado e varia entre as invocações – pode haver várias instâncias de uma determinada corrotina de uma vez.

> A diferença entre chamar outra corrotina por meio de "ceder" a ela e simplesmente chamar outra rotina (que então, também, voltaria ao ponto original), é que a relação entre duas corrotinas que

cedem uma à outra não é a do chamador, mas simétrica.

Uma aplicação de corrotina seria na industria de videogames, onde cada ator tem seus proprios procedimentos, mas eles cedem controle ao escalonador central, que faz a execução dele sequencialmente.

Vejamos um exemplo simples de como as corrotinas podem ser úteis: Suponha que você tenha um relacionamento consumidor-produtor em que uma rotina cria itens e os adiciona a uma fila e outra remove itens da fila e os usa. Por razões de eficiência, você deseja adicionar e remover vários itens de uma vez. O código pode ser assim:

corrotina.c

```
var q := new queu
coroutine produce # produtora
    loop # preenchendo/criando os elementos em
uma fila
        while q is not full
            create some new items
            add the items to q
        yield to consume # Quando ela termina,
```

```
ela manda para a consumidora
coroutine consume # consumidora
    loop
        while q is not empty # consumir os
elementos e remover
            remove some items from q
            use the items
        yield to produce # Quando ele acabar de
consumir, ele envia para a produtora
call produce
```

Um exemplo clássico, seria o de produtor e consumidor, onde a corrotina cria itens e os coloca em uma fila e a outra rotina remove os itens da fila para usá-los e fazer a atividade dela. Perceba que há um processo colaborativo, onde ocorre as execuções e eles ficam se avisando quando ocorre essa execução.

corrotina.py

```
# -*- coding: utf-8 -*-
# Exemplo de co-rotina
def imprimir_nome(nome):
```

```
    print("Procurando Nome:{}".format(nome))
    while True:
        texto_entrada = (yield)
        if nome in texto_entrada:
            print(nome, "foi encontrado no texto:
",texto_entrada)

# Chamando co-rotina. Nada irá acontecer.
co_rotina = imprimir_nome("teste")
# Isso iniciará a execução da co-rotina e
# Imprime a primeira linha "Procurando Nome ..."
# e avançar a execução para "yield"

co_rotina.__next__()

# Enviando entradas
co_rotina.send("teste 1")
co_rotina.send("teste 2")
co_rotina.send("este 3")
```

Temos um exemplo de corrotina implementada na linguagem de

> programação Python, diferente no programa anteriormente, esse tem uma função onde fazemos chamadas, e no resultado dela, foi achado o `teste 1` e `teste 2`, mas não o `este 3`.

A fila é completamente preenchida ou esvaziada antes de ceder o controle para a outra corrotina usando o comando `yield`. As chamadas de corrotinas adicionais começam logo após o rendimento, no `loop` de corrotinas externo.

Embora, esse exemplo seja frequentemente usado como uma introdução ao multithreading, dois threads não são necessários para isso: a instrução `yield` pode ser implementada por um salto direto de uma rotina para a outra.

As vantagens das corrotinas sobre os threads são:

- Elas podem ser usadas em um contexto de tempo real rígido (alternar entre as corrotinas não precisa envolver nenhuma chamada do sistema ou qualquer chamada de bloqueio).
- Não há necessidade de primitivas de sincronização, como mutexes, semáforos etc. para proteger as seções críticas.
- Não há necessidade de suporte do sistema operacional.

É possível implementar corrotinas usando encadeamentos de tarefas agendados preventivamente, de forma que seja transparente para o código de chamada, mas algumas das vantagens (particularmente a adequação para operação em tempo real e relativa economia de alternar entre eles) serão perdidas.

As corrotinas são úteis para implementar o seguinte:

- **MÁQUINAS DE ESTADO**: Dentro de uma única sub-rotina, em que o estado é determinado pelo ponto de entrada e de saída atual do procedimento. Isso pode resultar em um código mais legível em comparação com o uso da diretiva goto, e pode ser implementado por meio de recursão mútua com chamadas finais.
- **MODELO DE ATOR DE SIMULTANEIDADE**: Por exemplo, em videogames, cada ator tem seus próprios procedimentos (isso, novamente, separa logicamente o código), mas eles cedem de forma voluntária o controle ao escalonador central, que os executa sequencialmente (essa é uma forma de multitarefa cooperativa).
- **COMUNICAR PROCESSOS SEQUENCIAIS**: Ocorre quando cada subprocesso é uma corrotina. As entradas/saídas do canal e as operações de bloqueio geram corrotinas, e um planejador as desbloqueia em eventos de conclusão. Alternativamente, cada

subprocesso pode ser o pai daquele que o segue no pipeline de dados (ou o precede, caso em que o padrão pode ser expresso como geradores aninhados).

- **COMUNICAÇÃO REVERSA**: Comumente usada em software matemático, no qual um procedimento como um solucionador, avaliador integral, precisa do processo de uso para fazer um cálculo, como avaliar uma equação ou integrando.

PTHREADS

Um dos conceitos mais importantes relacionados a sistemas operacionais é denominado **processo**. De modo geral, um processo é uma abstração de um programa em execução.

Tal programa possui um espaço de endereçamento e, em sistemas tradicionais, apenas um thread (segmento ou fluxo de controle) de execução. Além disso, o processo contém toda informação relacionada ao seu contexto de execução, por exemplo, contador de programa, apontador de pilha e demais registradores. Ou seja, é um programa com sua função principal, denominada, sendo executado sequencialmente, instrução por instrução.

No entanto, em muitos sistemas operacionais é possível criar mais

de um thread no mesmo processo, isto é, no mesmo espaço de endereçamento. Nesse caso, mais de um fluxo de execução ocorre dentro do mesmo processo. Diante disso, é importante destacar algumas aplicações:

- Tratar atividades que ocorrem “simultaneamente”.
- Dividir a aplicação em tarefas que acessam recursos compartilhados.
- Reduzir o tamanho de uma aplicação, uma vez que threads ocupam menos espaço em relação aos processos.
- São mais fáceis de criar e destruir.
- A sobreposição de tarefas pode acelerar a aplicação.
- Possibilitam paralelismo real em sistemas multicore.

  É importante destacar que threads e processos são conceitos diferentes.

Processo	**Threads**
Como dito anteriormente, o processo é basicamente um agrupador de recursos (código	Os threads são criados no contexto de um processo e compartilham o mesmo

e dados) e possui uma identidade.	espaço de endereçamento.

Em vista disso, threads não são independentes como os processos, pois, embora compartilhem o mesmo espaço de endereçamento dentro de um processo, cada thread possui os mecanismos para gerenciar seu contexto de execução. Assim, threads possuem seu próprio contador de programa, seu apontador de pilha e seus registradores.

Os threads criados ocupam a CPU do mesmo modo que o processo criador, e são escalonados pelo próprio processo. Nesse contexto, quando uma aplicação multithread é executada, esses threads podem estar em qualquer um destes estados: em execução, bloqueado (aguardando), pronto para ser executado ou concluído (finalizado), conforme ilustrado na imagem a seguir:

Para padronizar a utilização de threads em diversos sistemas, o IEEE estabeleceu o padrão POSIX threads (IEEE 1003.1c), ou Pthreads. Esse padrão define mais de 60 funções – definidas na biblioteca **pthreads.h** para criar e gerenciar threads.

1. **Novo Thread**: Ela é criada,

2. **Pronto para ser executado**: pronta para ser executada,
3. **Em execução**: A thread pode passar pelo escalonador para ser executada,
4. **Aguardando**: A thread pode voltar para o estado de pronto, mas também para a operação de I/O,
5. **Pronto para ser executado**: Quando a operação anterior é resolvida, ela volta para o estado de pronto e volta para o ciclo em execução. Até que ela seja finalizada.

Além disso, a biblioteca define **estruturas de dados** e **atributos** para configurar os threads. De modo geral, esses atributos são passados como argumentos para os parâmetros das funções, por exemplo:

Exemplo:

`pthread_t`: Handle para pthread, isto é, um valor que permite identificar o thread.

`pthread_attr_t`: Atributos para configuração de thread.

OPENMP C/C++ 14.0.0

OpenMP é uma interface de programação de aplicativo (API) de memória compartilhada cujos recursos, como acabamos de ver, são baseados em esforços anteriores para facilitar a programação paralela de memória compartilhada.

O OpenMP se destina à implementação em uma ampla gama de arquiteturas SMP.

> **Atenção**: À medida que máquinas multicore e processadores multithreading se espalham no mercado, eles podem ser cada vez mais usados para criar programas para computadores uniprocessadores.

Como seus predecessores, OpenMP não é uma nova linguagem de programação. Em vez disso, é uma notação que pode ser adicionada a um programa sequencial em C ou C++ para descrever como o trabalho deve ser compartilhado entre threads que serão executados em diferentes processadores ou núcleos e

para solicitar acessos a dados compartilhados conforme necessário.

A inserção apropriada de recursos OpenMP em um programa sequencial permitirá que muitos, talvez a maioria, dos aplicativos se beneficiem de arquiteturas paralelas de memória compartilhada – geralmente com modificações mínimas no código. Na prática, muitos aplicativos têm um paralelismo considerável que pode ser explorado.

O sucesso do OpenMP pode ser atribuído a vários fatores:

- Sua forte ênfase na programação paralela estruturada.
- O fato de ser comparativamente simples de usar, uma vez que a responsabilidade de trabalhar os detalhes do programa paralelo é do compilador.
- A grande vantagem de ser amplamente adotado, de modo que um aplicativo OpenMP pode ser executado em muitas plataformas diferentes.

> Mas, acima de tudo, o OpenMP é oportuno. Com o forte crescimento na implantação de SMPs pequenos e grandes e outros hardwares multithreading, a necessidade de um padrão de

> programação de memória compartilhada que seja fácil de aprender e aplicar é aceita em todo o setor. Os fornecedores por trás do OpenMP entregam coletivamente grande fração dos SMPs em uso nos dias atuais. Seu envolvimento com esse padrão de fato garante sua aplicabilidade contínua às suas arquiteturas.

Prós

- Código multithreading portátil (em C/C++ e outras linguagens, normalmente é necessário chamar primitivas específicas da plataforma para obter multithreading).
- Não precisa lidar com a passagem de mensagens como o MPI faz.
- O layout e a decomposição dos dados são controlados automaticamente por diretivas.
- Escalabilidade comparável ao MPI em sistemas de memória compartilhada.
- Paralelismo incremental: pode funcionar em uma parte do programa ao mesmo tempo, sendo que nenhuma mudança drástica no código é necessária.
- Código unificado para aplicativos seriais e paralelos: construções

OpenMP são tratadas como comentários quando compiladores sequenciais são usados.

- As instruções de código original (serial) não precisam, em geral, ser modificadas quando paralelizadas com OpenMP. Isso reduz a chance de introduzir bugs inadvertidamente.
- Tanto o paralelismo de granulação grossa quanto o de granulação fina são possíveis.
- Em aplicações multifísicas irregulares que não aderem apenas ao modo de computação SPMD (Single Program Multiple Data), conforme encontrado em sistemas de partículas de fluido fortemente acoplados, a flexibilidade do OpenMP pode ter grande vantagem de desempenho sobre o MPI.
- Pode ser usado em vários aceleradores, como GPGPU (General Purpose Graphics Processing Unit) e FPGAs (Field Programmable Gate Array).

CONTRAS

- Risco de introdução de bugs de sincronização difíceis de depurar e condições de corrida.

- A partir de 2017, funciona de forma eficiente somente em plataformas de multiprocessador de memória compartilhada.
- Requer um compilador que suporte OpenMP.
- A escalabilidade é limitada pela arquitetura da memória.
- Sem suporte para comparar e trocar.
- Falta um tratamento confiável de erros.
- Carece de mecanismos refinados para controlar o mapeamento do processador de thread.
- Alta chance de escrever código de compartilhamento falso acidentalmente.