Introdução à Programação Paralela e Hands-on Pthreads e OpenMP

Lucas de Sousa Rosa e Alfredo Goldman

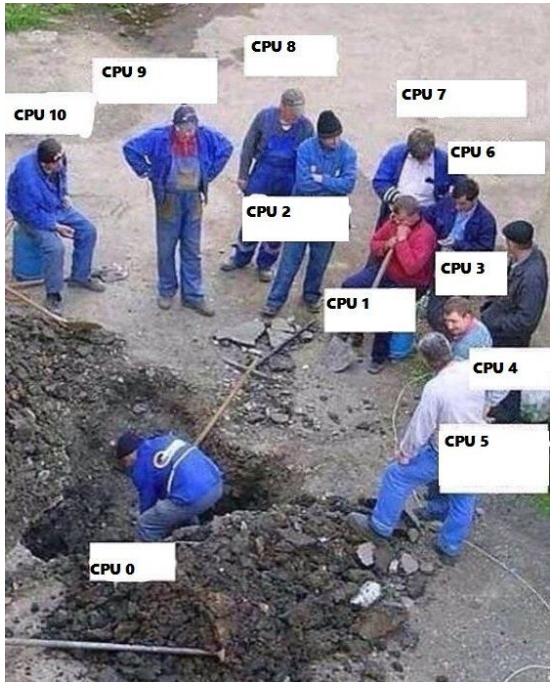
MAC0219 - Programação Concorrente e Paralela

30 de agosto de 2024

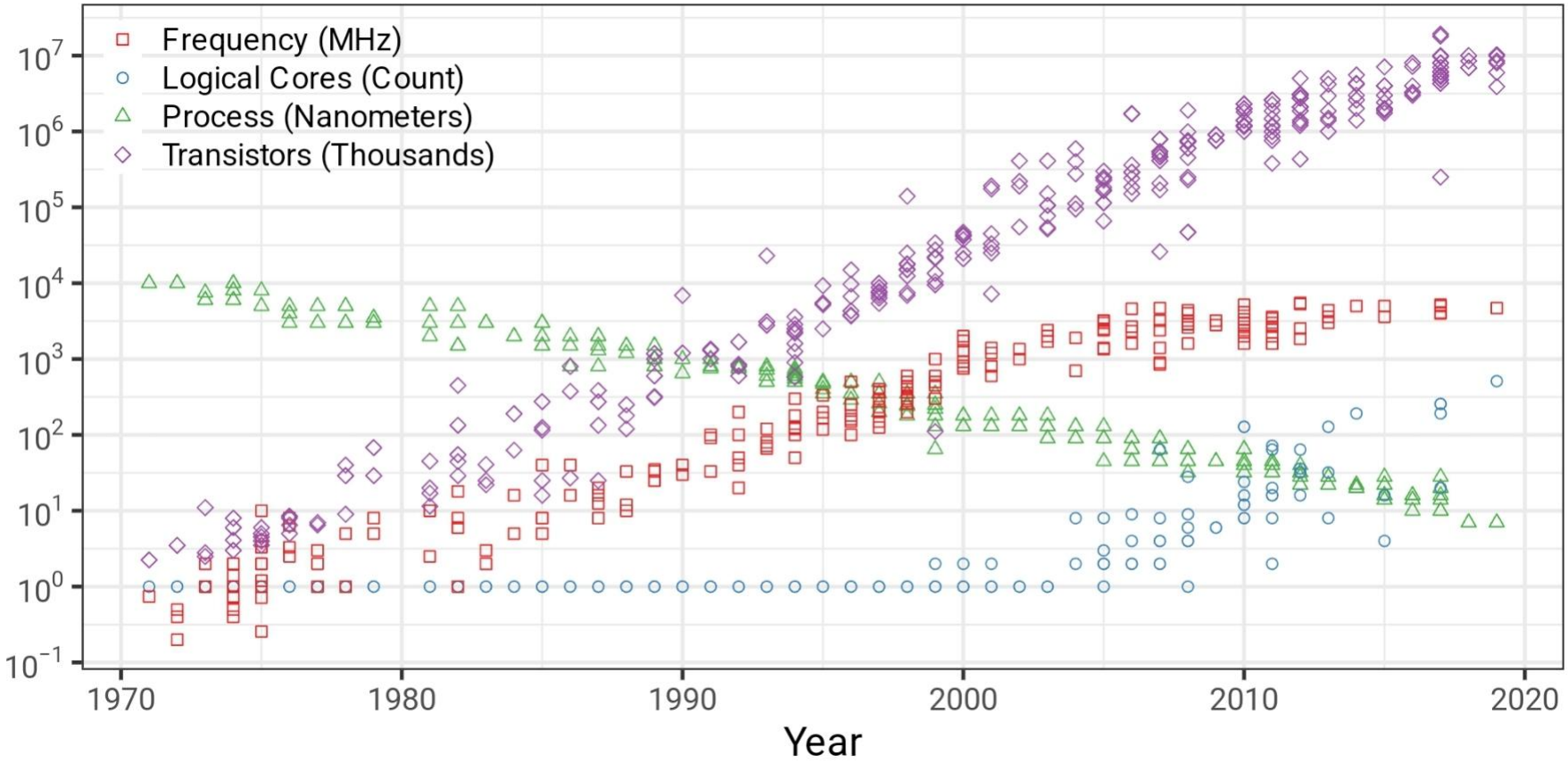
Motivação

Introdução

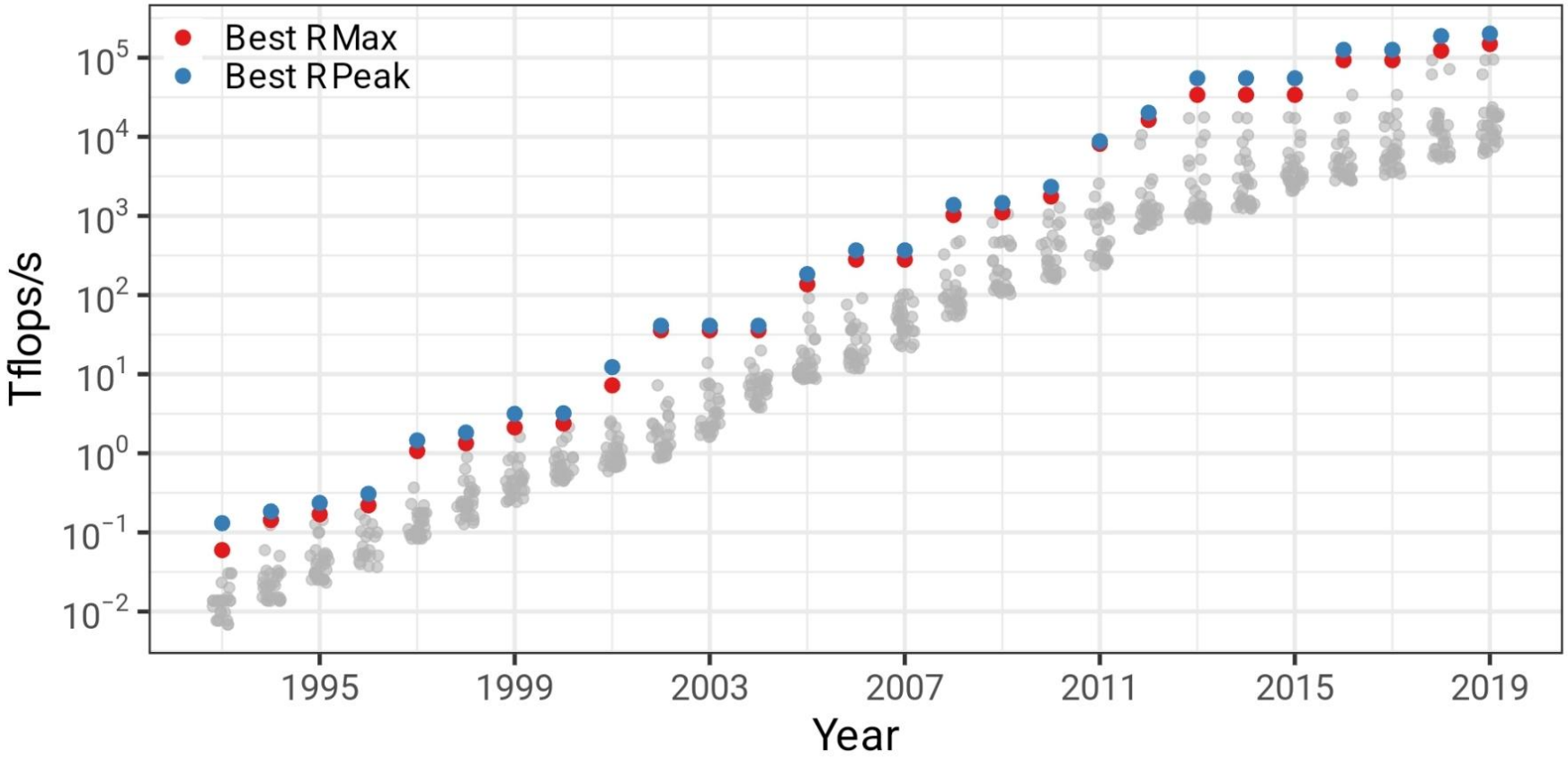
- *Software* era (parece ainda ser) escrito para computadores seriais.
  - Algoritmo = um fluxo serial de instruções.
  - Essas instruções são executadas em uma unidade central de processamento (CPU).
  - Apenas uma instrução pode ser executada por vez.
- Para certos problemas essa abordagem não é eficiente.
  - Paralelismo natural
  - Tarefas independentes.



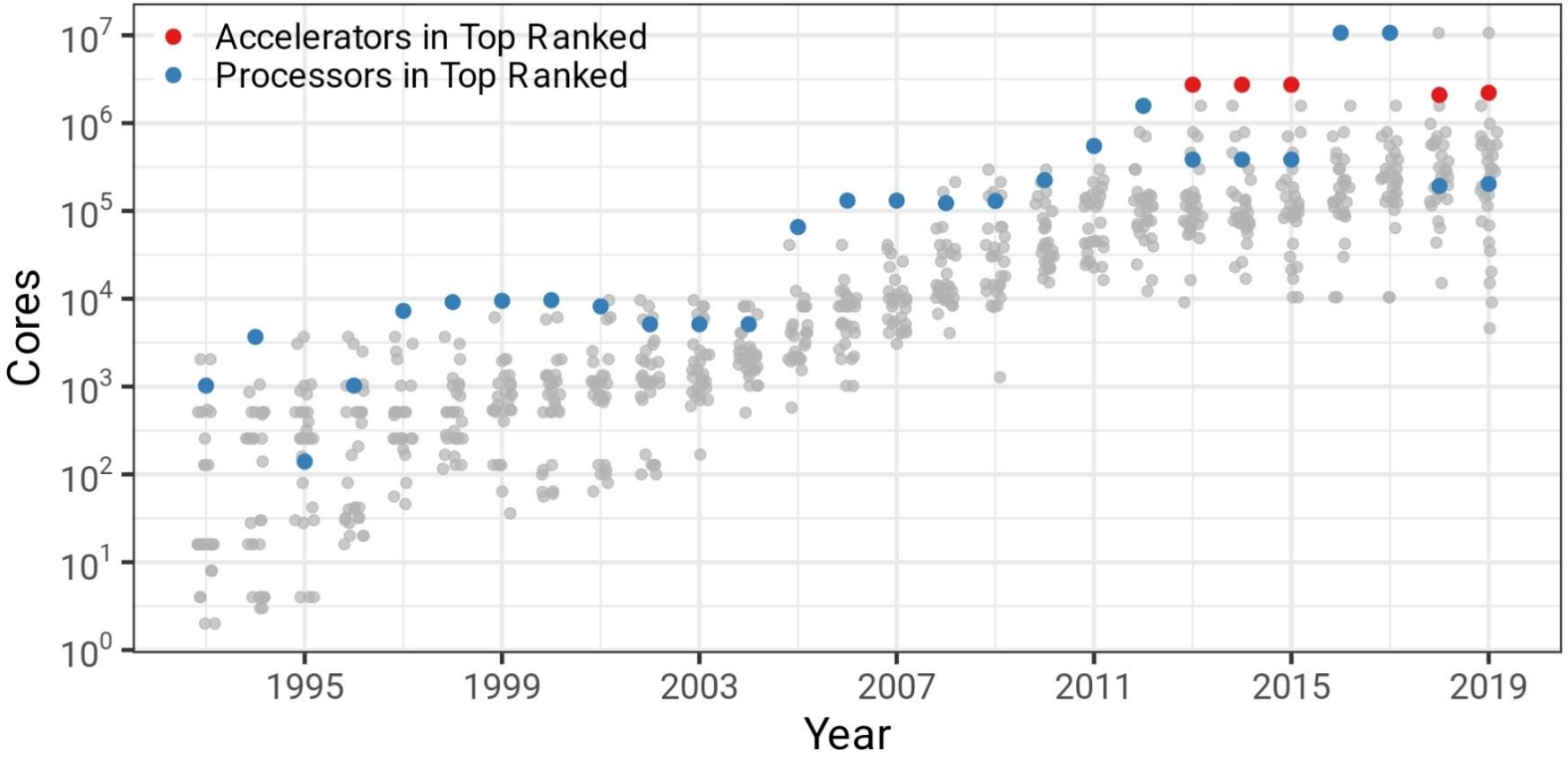
49 anos de tendências em microprocessadores



Top500: Rpeak e RMax



Top500: Núcleos de processador e acelerador



Conceitos Básicos

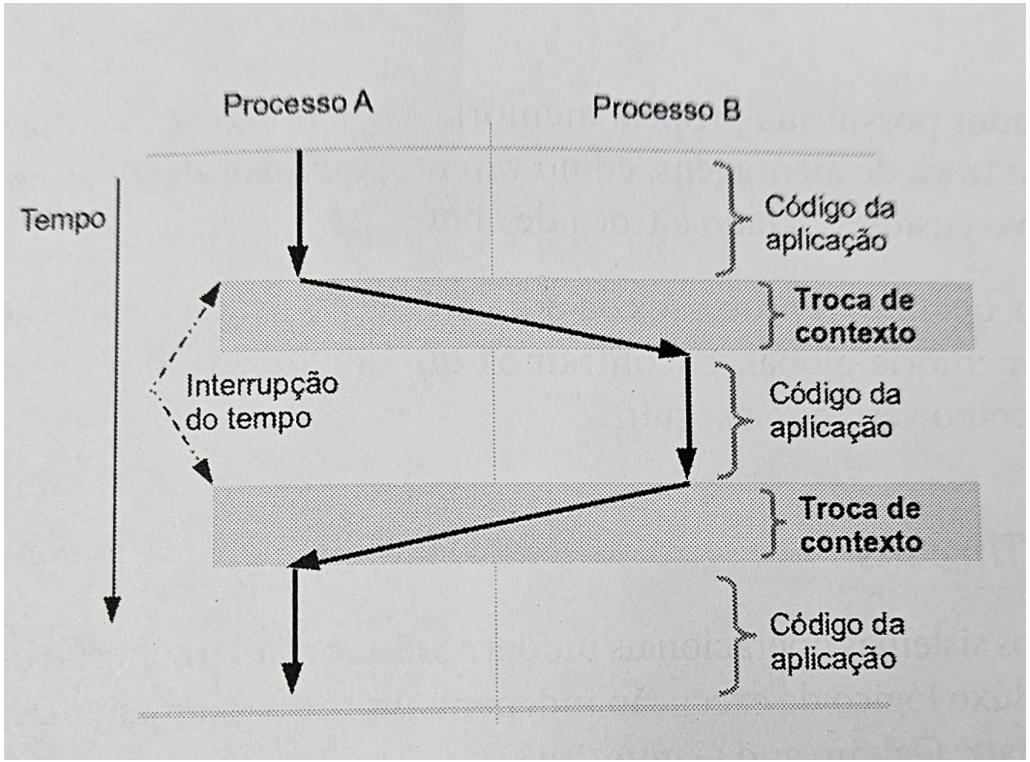
Sistemas Operacionais

- Sistema operacional faz a ponte entre o *hardware* e as aplicações do usuário.
- Três abstrações fundamentais:
  - Arquivos
  - Memória virtual
  - Processos.
- Os **arquivos** simplificam o acesso a dispositivos de E/S permitindo armazenar dados de forma permanente.
- A **memória virtual** gerencia o acesso a memória principal e secundária.
- Os **processos** encapsulam os recursos necessários à execução de um programa.
  - Pode conter vários fluxos independentes de execução, ou **threads**.

Processos

- Processos possuem um **contexto** com informações necessárias para executar o programa.
  - Código do programa
  - Dados armazenados na memória
  - Pilha de execução
  - Conteúdo dos registradores
  - Variáveis de ambiente
  - Descritores de arquivos abertos.
- **Trocas de contexto** e compartilhamento do tempo de CPU.
  - Processos são interrompidos para execução de novos processos.
  - Trocas de contexto são **caras**: precisa salvar o contexto atual e carregar o novo contexto.

Processos

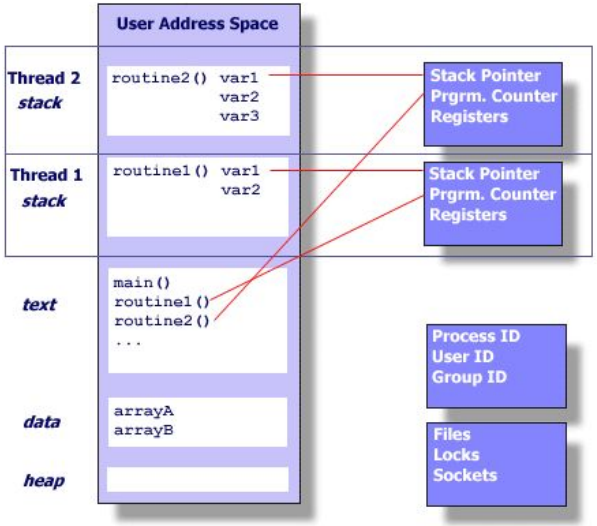

Processo A
Processo B
Tempo
Código da aplicação
Troca de contexto
Interrupção do tempo
Código da aplicação
Troca de contexto
Código da aplicação

Processos

- Podemos explorar o paralelismo de uma aplicação usando diversos processos.
  - Trocar de contexto é caro.
  - O contexto de cada processo é isolado. Qualquer comunicação exige **mecanismos de comunicação interprocessos**.
  - Funciona bem em sistemas distribuídos, já que a troca de contexto deixa de ser um problema.
- Se os processos compartilham o mesmo espaço de memória, há uma alternativa mais eficaz.
  - Threads!!

Threads

- Thread = um “procedimento” que é executado independentemente do processo principal.
  - Usa e existe dentro dos recursos de um processo.
  - Pode ser agendada pelo sistema operacional.
  - Duplica apenas os recursos essenciais para ser executada. É mais leve.
- Threads de um mesmo processo compartilham recursos.
  - Ler e escrever nos mesmos locais de memória é possível (requer sincronização explícita pelo programador).
  - Recursos compartilhados pelo sistema são visíveis para todas as threads.
  - Dois ponteiros com o mesmo valor apontam para os mesmos dados.



Threads versus processos

É mais barato: tempo para a criação de 50.000 processos (`fork()`) e uma thread (`pthread_create()`)

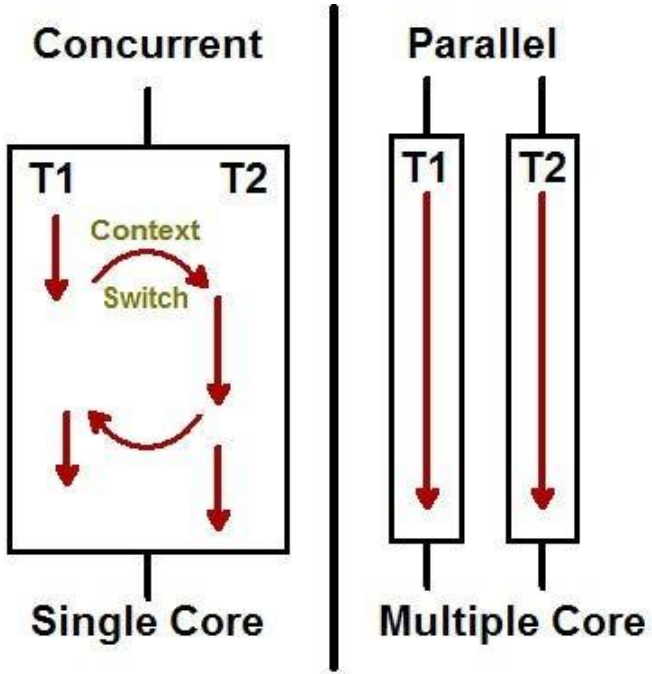
Outro motivo: threads compartilham o mesmo espaço de endereço (não há transferência de dados)

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

Programação Paralela

Só mais alguns conceitos

- Geralmente escrevemos código de forma sequencial.
- Concorrência versus paralelismo.
  - Operações paralelas podem ser executadas ao mesmo tempo.
  - Operações que foram iniciadas e não foram ainda completadas.
  - Paralelismo é uma forma da implementação de concorrência.
- Dois paradigmas para escrever programas paralelos:
  - Programação com **memória compartilhada**.
  - Programação por **troca de mensagens** (veremos no futuro).



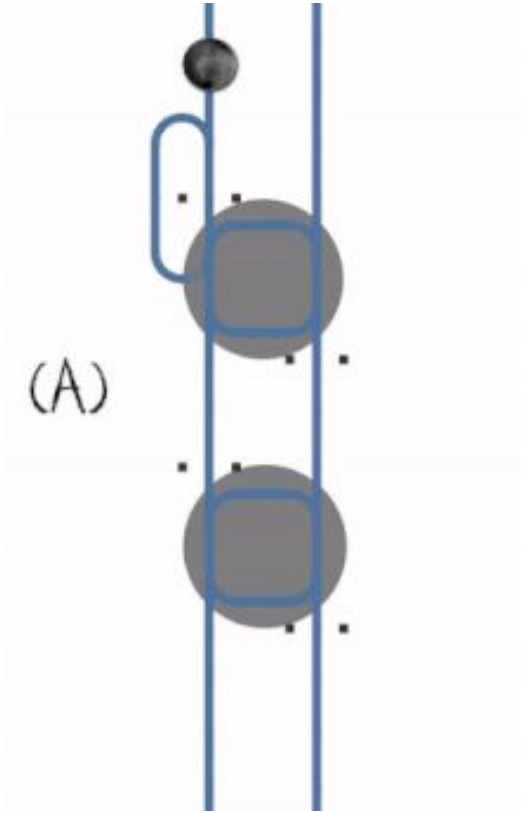
https://42bits.medium.com/concorr%C3%AAncia-vs-paralelismo-585fc0636c64

Programação com memória compartilhada

- A comunicação entre as tarefas é feita por meio de variáveis armazenadas em um espaço de memória compartilhada.
  - Podemos usar tanto processos, quanto threads para mapear as tarefas.
  - Threads são mais adequadas e menos complexas para esse paradigma.
- A comunicação pode gerar **condições de corrida**.
  - Leituras e escritas simultâneas onde o resultado pode depender da ordem de execução das threads.
  - Exemplo - incrementando uma variável global.
- Proteger **regiões críticas** para evitar condições de corrida.
  - Usamos **sincronização por exclusão mútua** para protegê-las.
- Sincronização excessiva ou incorreta pode causar ***deadlocks***.
  - Ocorre quando cada tarefa em um conjunto de tarefas é bloqueada à espera de um recurso pertencente a outra tarefa no conjunto.
  - Condições para ocorrência de *deadlocks*: condições de Coffman.

Condições de Coffman

1. **Exclusão mútua**: um recurso só pode ser usado por um processo de cada vez.
2. **Contenção de recursos (hold and wait):** um processo que já possui recursos pode solicitar e aguardar por outros.
3. **Nenhuma preempção**: um recurso só pode ser liberado voluntariamente pelo processo que o detém, não podendo ser tomado à força.
4. **Espera circular**: existe um ciclo de processos, onde cada um espera por um recurso que está sendo detido pelo próximo processo no ciclo.



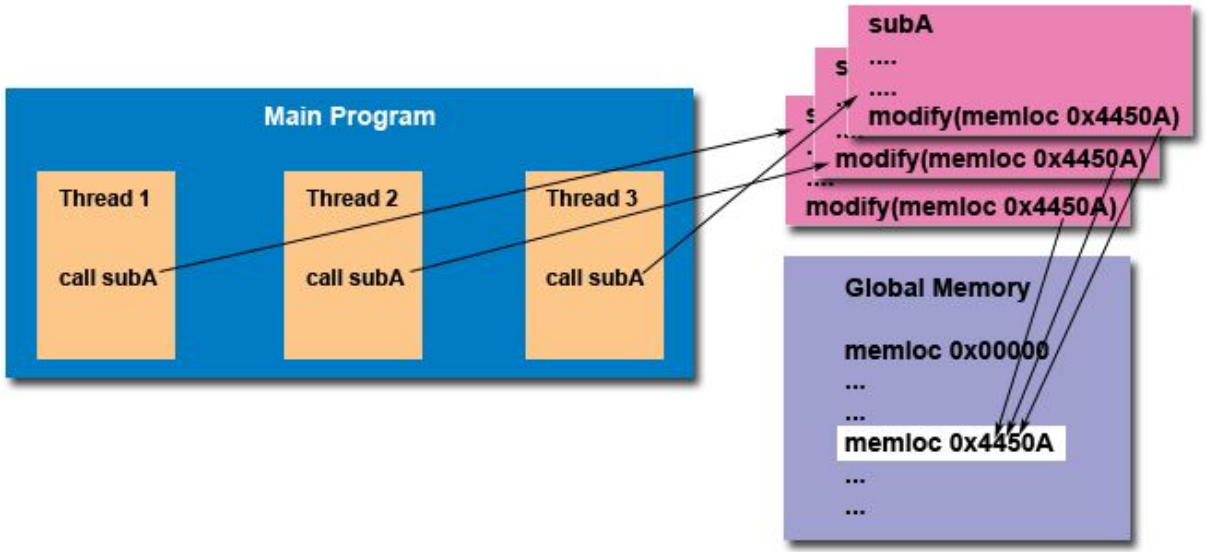
https://en.wikipedia.org/wiki/Deadlock_(computer_science)

Projetando programas multithread

- O problema deve ser decomponível em **tarefas independentes** que possam ser executadas em **paralelo**.
- Modelos mais comuns:
  - **Manager/worker**: o manager cuida dos inputs e divide o trabalho entre os workers. Pode ser atribuído de forma estática ou dinâmica.
  - **Pipeline**: parecido com uma linha de montagem de automóveis.
  - **Peer**: mesmo modelo do manager/worker. O manager também trabalha.
- Alguns cuidados para tomar:
  - Memória compartilhada (sincronização e proteção é por conta do programador).
  - Limite máximo de threads e o tamanho da pilha da thread pode variar entre sistemas.
  - Thread-safeness.

Projetando programas multithread

- Cuidado ao usar bibliotecas ou outros objetos que não garantam explicitamente **thread-safeness**.
- Em caso de dúvida, assuma que eles não são seguros até que se prove o contrário.



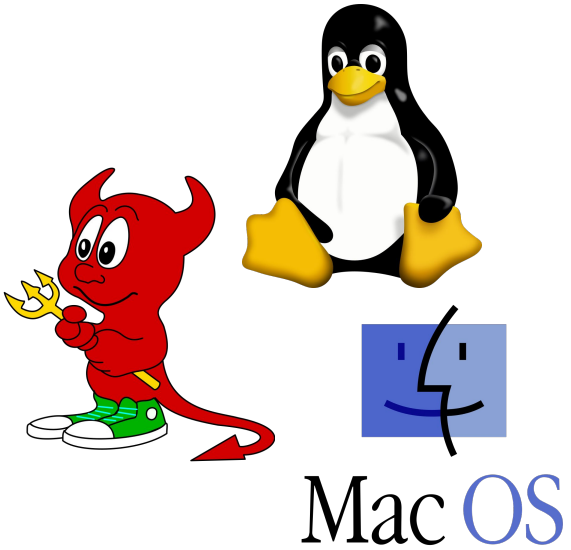
Glossário

- Condição de corrida
  - Situação onde ocorre a manipulação simultânea de um recurso por duas ou mais threads.
  - Pode levar a resultados incorretos.
- Exclusão mútua
  - Propriedade do controle de concorrência que assegura **acesso exclusivo** a um recurso compartilhado.
  - Impede condições de corrida.
- Seção crítica
  - Trecho do código que coordena o acesso a um recurso compartilhado. Proteção contra acessos simultâneos.
  - Requer exclusão mútua de acesso.
- Deadlock
  - É uma situação em que um grupo de entidades fica parado porque cada uma precisa que outra faça algo antes de poder continuar, criando um ciclo de espera que impede qualquer progresso.

Hands-on Pthreads e OpenMP

O que são Pthreads?

- Pthreads é um conjunto de tipos e de chamadas procedurais da linguagem C.
  - Geralmente faz parte de bibliotecas padrão como a `libc`.
  - Implementada em `/usr/include/pthread.h`.
- Pthreads deriva do conjunto de normas POSIX.
  - Padrão IEEE POSIX 1003.1c (1995).
  - **P**ortable **O**perating **S**ystem **I**nterface. O X deriva da herança dos sistemas UNI**X**.
  - POSIX define um conjunto de normas para compatibilidade de sistemas operacionais como APIs, shells de linha de comando e interfaces utilitárias.



Gerenciamento de Threads

Criando e terminando as threads (`pth_exemplo00.c` e `pth_exemplo01.c`)

- `pthread_create()`
  - `thread`: um "identificador exclusivo" para a nova thread.
  - `attr`: atributos da thread ou `NULL` para valores padrão.
  - `start_routine`: o código que a thread executará depois de criada.
  - `arg`: um único argumento que pode ser passado para `start_routine`. Ele deve ser passado por referência como `(void *)`.
- `pthread_exit()`
  - Adicionar no final da `main()` evita maiores problemas.

Gerenciamento de Threads

Passando argumentos para as threads (`pth_exemplo02.c`)

- Usar uma `struct` com todos os argumentos é uma boa solução.
  - Verifique se os dados são seguros - não podem ser alterados por outras threads.

Sincronizando threads com join (`pth_exemplo03.c`)

- `pthread_join()` bloqueia a thread até o seu término.
- `pthread_exit()` pode ser usado para obter o `status` de término da thread.
- Usando o argumento `attr`.

Variável Mutex

Mutex e produto interno (`pth_exemplo04.c` e `pth_exemplo05.c`)

- Uma variável mutex age como um bloqueio protegendo o acesso a um recurso compartilhado.
- `pthread_mutex_init()`
  - `mutex`: um "identificador exclusivo" para o novo mutex.
  - `attr`: atributos do mutex ou `NULL` para valores padrão (semelhante às threads).
- Gerenciando o mutex com `pthread_mutex_lock()` e `pthread_mutex_unlock()`.

O que é OpenMP?

Uma API usada para criar programas multithreaded de memória compartilhada em C/C++ e Fortran.

- Fornece capacidade para paralelizar gradualmente um programa serial.
- A API OpenMP é composta por três componentes:
  - Diretivas de compilação
  - Runtime Library Routines
  - Variáveis de ambiente.
- Disponível no Ubuntu através do pacote `libomp-dev`.
- Escrevemos bem menos código 😀 (versus Pthreads).
  - Veremos no final.

Escrevendo diretivas e construtor parallel

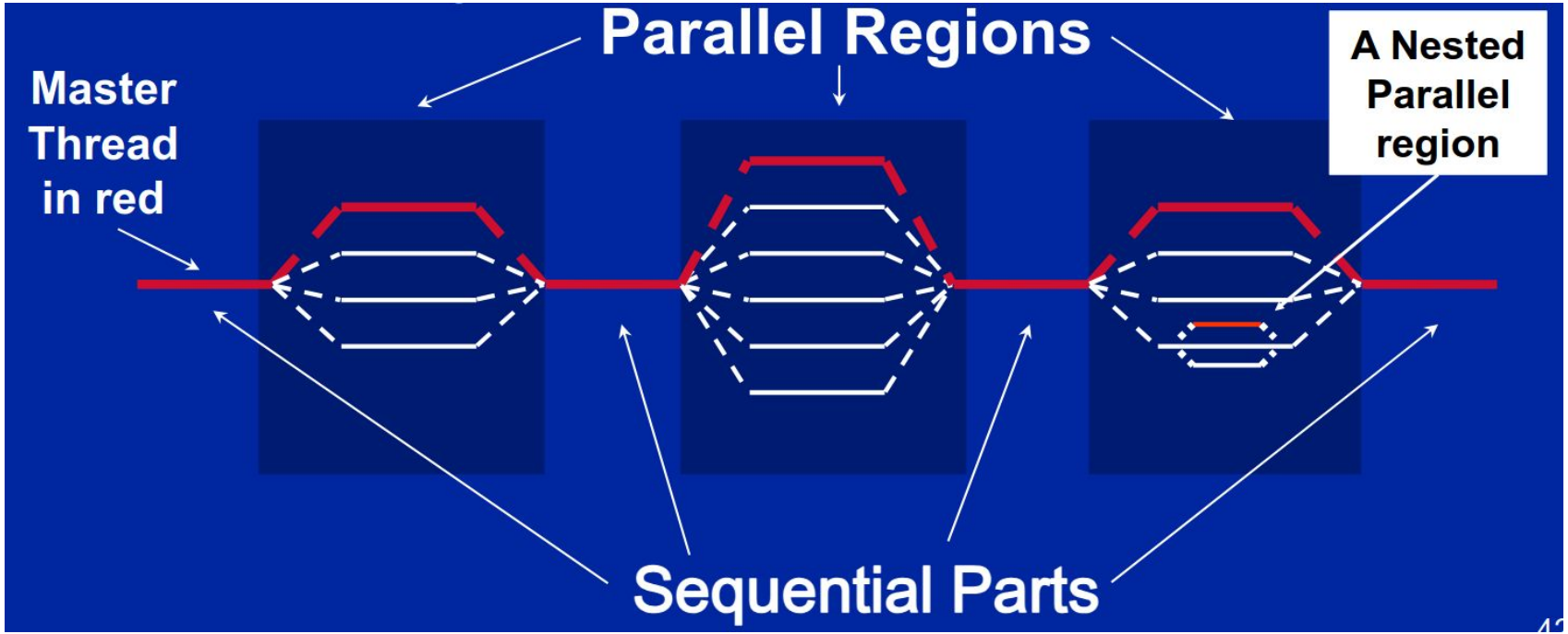
Formato: `#pragma omp directive-name [clause, ...] newline`

1. `#pragma omp`: necessário para todas as diretivas OpenMP C/C++.
2. `directive-name`: uma diretiva válida.
3. `[clause, ...]`: cláusulas são opcionais.
4. `newline`: bloco encapsulado pela diretiva (obrigatório).

Região paralela, cláusula private e número de threads (`omp_exemplo00.c`)

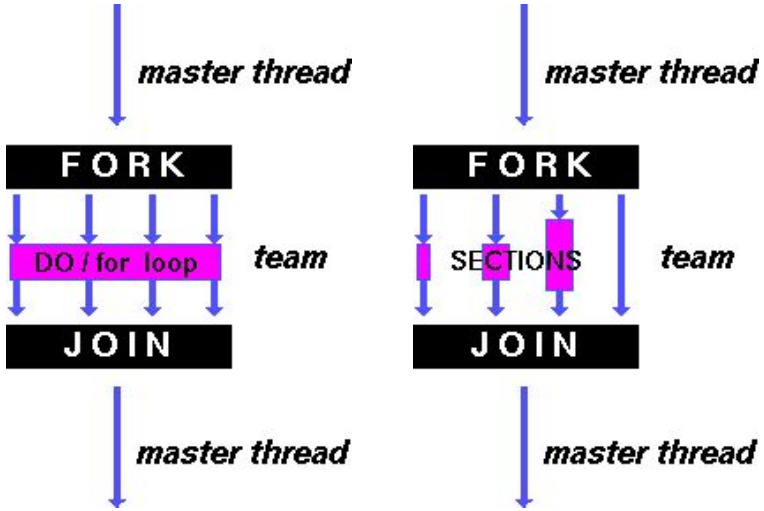
- Uma região paralela é um bloco de código que será executado por várias threads.
  - Há uma barreira implícita no final de uma seção paralela.
- Definindo o número de threads:
  - Variável de ambiente `OMP_NUM_THREADS`.
  - Função `omp_set_num_threads()`.

Construtor parallel



Construtores work-sharing e de sincronização

- Divide a execução da região de código entre as threads.
  - Não cria novas threads.
  - Não tem barreira para entrar, mas tem para sair.
- Diretiva **do/for**:
  - Compartilha iterações de um loop pela equipe (paralelismo de dados).
- Diretiva **sections**:
  - Divide o trabalho em seções separadas e discretas (paralelismo funcional).
  - Cada seção é executada por um thread.
- Eu posso combinar o construtor parallel com os work-sharing.



Construtores work-sharing e de

Algumas diretivas de sincronização são:

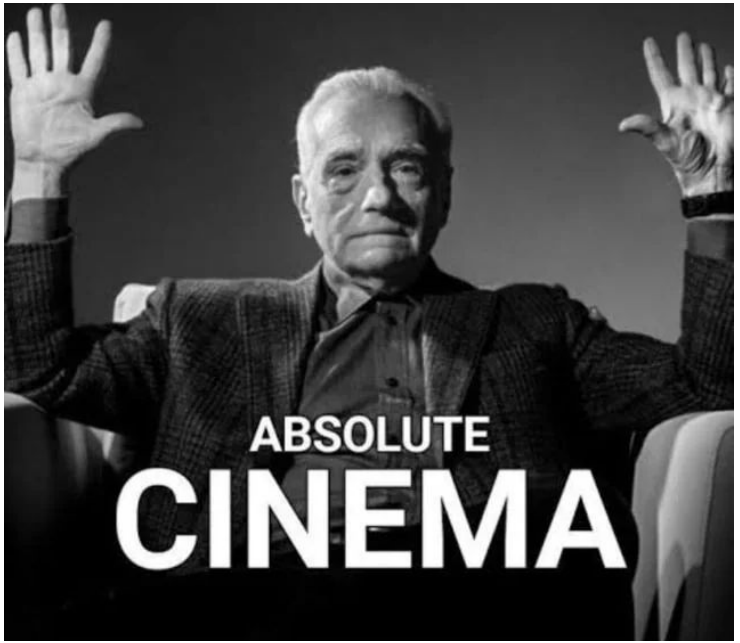
- `Master`: só a thread master pode executar a região.
- `Barrier`: sincroniza todas as threads
- `Critical`: especifica uma região crítica.

Diretiva do/for, cláusulas shared, schedule e nowait (`omp_exemplo01.c` e `omp_exemplo02.c`)

- `schedule`
  - Descreve como as iterações do loop são divididas entre as threads.
- `nowait`
  - Remove a sincronização no final do bloco.

Produto interno e as cláusulas default e reduce (`omp_exemplo03.c`)

Com uma única linha conseguimos paralelizar o cálculo do produto interno



Referências

- Essa aula foi preparada com base nos slides do **Pedro Bruel** (Github @phrb), nos slides do **Tim Mattson** "*Introduction to OpenMP*" nos tutoriais "*POSIX Threads Programming*" e "*OpenMP Tutorial*" do **LLNL** e no livro "*Programação Paralela e Distribuída*" do **Gabriel**, **Calebe** e **Evaldo**.
- Os códigos-fonte utilizados e outras aulas estão disponíveis no repositório do Github **fredgrub/aulas-PPD**.