

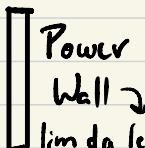
Resumo La Prova - / Computacões Paralela

Sophia Carrazza

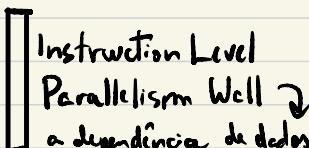
Arquiteturas Paralelas

- Lei de Moore → n° de transistores em um chip dobrar a cada 18 meses
 - ↳ aumenta extâgios do pipeline
 - ↳ aumenta frequência
 - ↳ aumenta tam. da memória cache

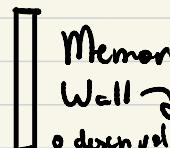
➢ Os três muros



Jim da lei de Dennard em 2006,
(tamanho dos transistores
e consumo de energia era
constante) → passou a ganhar
+ energia



a dependência de dados
entre instruções limita
o aumento dos pipelines (já
mão compensava + aumentar
dados na memória
+ tam.)



o desenvolvimento da
CPU foi mtº maior que
da memória. A busca de
dados na memória
retarda o processamento

★ Multicore → processadores com múltiplos núcleos em um mesmo chip, que cooperam entre si p/ solução de problemas computacionais de forma rápida

↳ o paralelismo para a vez responsabilidade do programador

taxonomia
de
Flann

- MIMD - multiple instruction, multiple data → mem. Compartilhada
- SIMD - single instruction, multiple data → mem. Distribuída
- MISD - não jogado
- SISD - sequencial

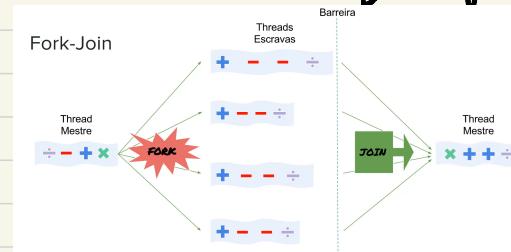
➢ Computação Neuromórfica → abordagem computacional que imita a maneira como o cérebro humano funciona.

↳ "a network of neurosynaptic logic"

Padrões de Programação Paralela

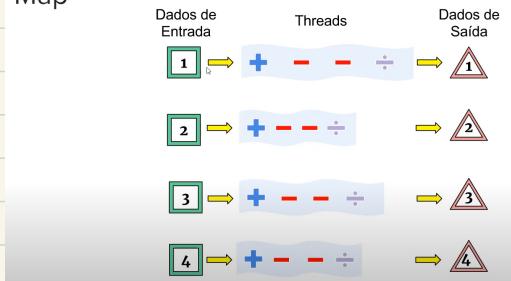
- ▶ programação estruturada → (seqüencial) comporta por estruturas de seleções, repetições, recursões, junções, etc.
 - ↳ serialização do código. Tem um ordenamento temporal e é determinística.
 - ↳ um for tem uma serialização → parallel-for

- ▶ programação paralela estruturada → comporta por estruturas
 - ↳ **fork-join** = permite que uma seção de código seja executada em paralelo por vários threads trabalhadores. Uma thread mestre divide para (fork) várias threads e no final de cada uma, elas se sincronizam (join).



↳ **map** = replica uma função sobre cada elemento de um conjunto de índices (em dados diferentes).
É necessário que as funções (iterações) sejam independentes
→ ex: parallel-for

Map

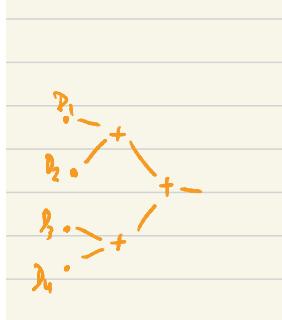
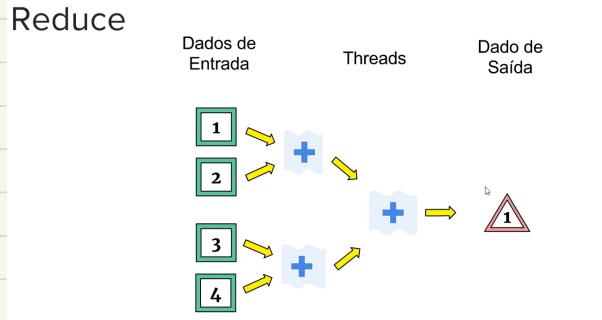


$D_1 \xrightarrow{f(x)} .$
 $D_2 \xrightarrow{f(x)} .$

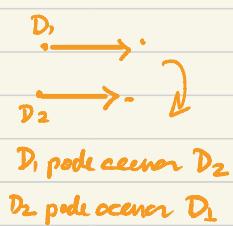
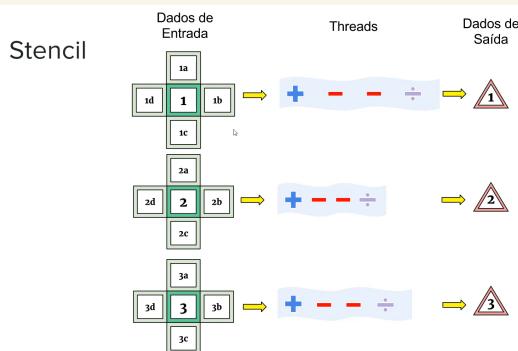
↳ **reduce** = combina cada elemento de uma coleção, geralmente em pares, utilizando um operador, até reduzi-las a um único elemento.

↳ ex: somatório de inteiros, achar o maior valor, etc

Reduce

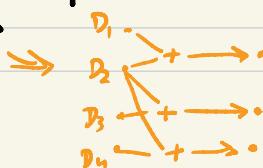


↳ **Stencil** = é uma generalização do MAP, onde cada função processa mais do que um único elemento, mas tb seus vizinhos.
↳ convoluções de imagens, detecção de bordas, etc

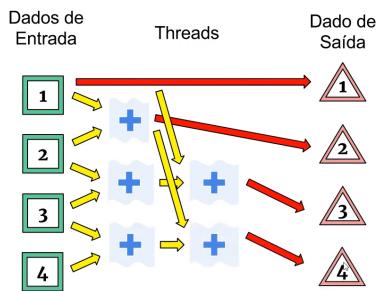


↳ **scan** = computa todos os reduções parciais de uma coleção de elementos, ou seja, p/ cada saída, uma redução parcial é computada até o elemento atual.

↳ somatórios parciais



Scan



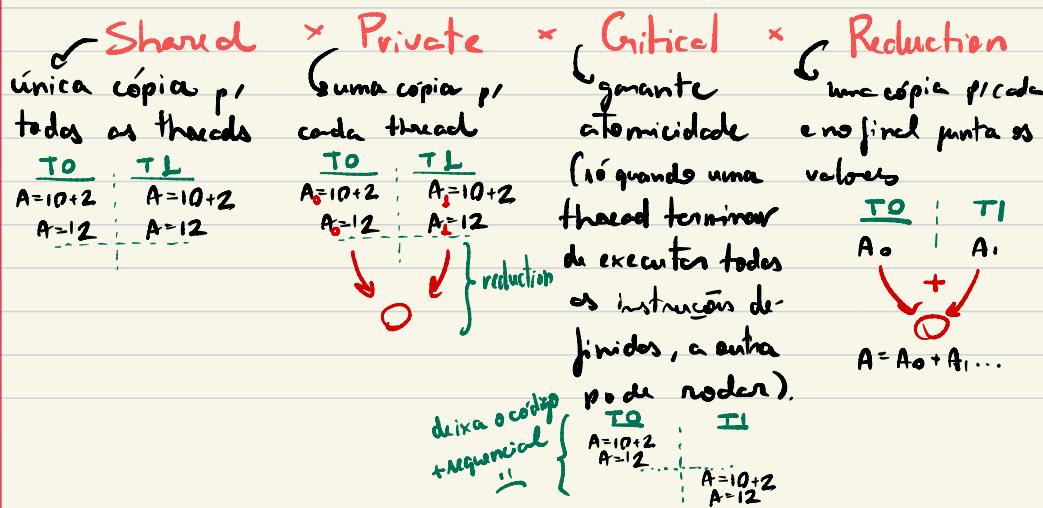
Suporte aos Padrões pelos Modelos de Programação

- Open Multi-processing (OpenMP)
 - Fork-Join, Map e Reduce
- Compute Unified Device Architecture (CUDA)
Open Computing Language (OpenCL)
 - Map
- Intel Thread Building Blocks (TBB)
 - Fork-Join, Map, Reduction, Pipeline, Scan e outros
- Message Passing Interface (MPI)
 - Map, Gather, Scatter, Reduce e outros



Open MP (memória compartilhada)

- Fork - Join - execução do mesmo código em threads ≠.
- MAP - distribui as iterações entre threads
- Reduce - soma resultados parciais entre threads
- API padronizada p/ computação paralela



processo parado → memória local e privada
mensagens → utilização de rede de comunicação.

Biblioteca MPI (memória distribuída)

→ modelo de passagem de mensagens: permite a programação de máquinas de memória distribuída.

↳ programação explícita baseada em troca de mensagens com duas operações:

- `Send()`
- `receive()`

→ MPI = Message Passing Interface = conjunto de processos parados que se comunicam por meio de mensagens.
processo parado → memória local e privada
mensagens → utiliza redes de comunicação

→ SPMD = Single Program Multiple Data = todos os processos vão executar no mesmo programa e cada processo é identificado por um ID (um rank).

→ São processos, e não threads! (não compartilham a mesma memória). Eles podem estar na mesma ou em máquinas diferentes.

→ Comandos básicos:

- `MPI_Init(&argc, &argv)` = inicializa o MPI
- `MPI_Comm_rank(&r, communicator)` = retorna o ID (rank) em "int r"
- `MPI_Comm_size(&p, communicator)` = retorna o nº de processos
- `MPI_Send (muitos argumentos)`
- `MPI_Recv (muitos argumentos)`
- `MPI_Finalize()` = finaliza
- `MPI_COMM_WORLD` = constante comunicadora básica que tem todos os processos.

→ Mensagem em MPI:

- MPI_Send(buffer, tamanho, tipo, destino, tag, comunicador)
pontiro p/ o buffer
buffer
ex: MPI.INT

- MPI_Recv(buffer, tamanho, tipo, remetente, tag, comunicador, status).
info re mensagem foi recebida corretamente, etc.

Operações Coletivas em MPI:

comunicação coletiva
paralelos

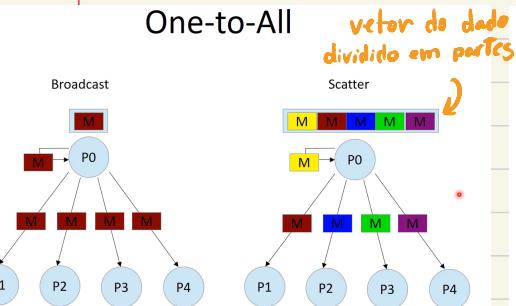
→ permite que todos os processos troquem mensagens ao mesmo tempo (não precisa fazer ponto a ponto).

→ há 3 categorias:

- One-to-All = broadcast, scatter
- All-to-One = reduce, gather
- All-to-All = all-gather, all-to-all

→ Toda operação coletiva funciona como uma barreira de sincronização.

One-to-All



vetor de dado dividido em partes

Broadcast

Scatter

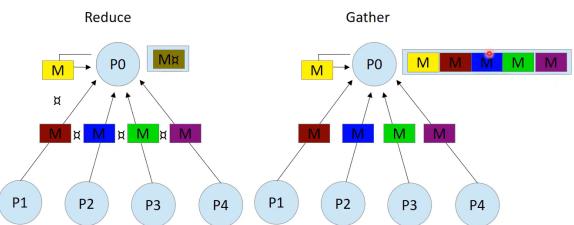
envia a mesma mensagem p/ todo mundo

envia mensagens distintas p/ cada um (cada processo recebe um pedaço)

All-to-One

Reduce

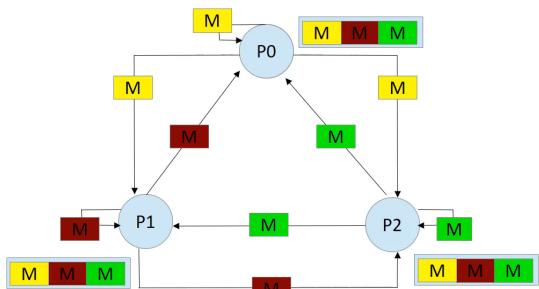
Gather



aplica uma operação sobre os dados enviados por todos os processos

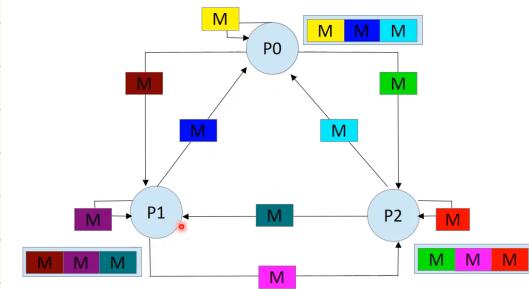
só coleta os dados de todos os processos p/ um único raiz concatenados em uma sequência.

All-Gather



Cada processo envia um bloco de dados. No final, todos os processos recebem a concatenação dos blocos de todos os processos

All-to-All



Cada processo possui N blocos e envia o K-ésimo p/ o K-ésimo processo. No final, cada processo recebe blocos diferentes.

Broadcast / Reduce

- MPI - Bcast (buffer, tam, tipo, remetente, comunicador)
 - ↳ se $for = 0$, estou enviando
 - ↳ se $for \neq 0$, estou recebendo
 - MPI - Reduce (send-buffer, recv-buffer, tam, tipo, operação, destino, comunicador).

Contagem de Números

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

#define N 10
#define MAX 4
#define NUMBER 3

int main(int argc , char **argv)
{
    int size, rank, i, buffer[N], partial, total, number = NUMBER;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    if(rank == 0) {
        srand(time(NULL));
    }

    for(i=0; i < N; i++){
        buffer[i] = rand()%MAX;
    }

    Procurar quantas vezes
    o número 3 é encontrada
    no buffer com N posições

    Processo de
    um vetor com
    valores aleatórios menor que MAX
```

Contagem de Números

MPI_Bcast(buffer,N,MPI_INT,0,MPI_COMM_WORLD)

```
partial = 0;
for(i=rank; i < N; i+=size){
    if(buffer[i] == number)
        partial++;
}
```

MPI_Reduce(&partial, &total, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

```
if(rank==0){  
    printf("Input array\\n");  
    for(i=0; i < N; i++){  
        printf("%d ",buffer[i]);  
    }  
}
```

```
printf("\n\nTotal of number \"%d\" found = %d\n",number,total);
```

Envia o buffer para todos os processos.

Cada processo, vasculha um pedaço do vetor.
Ex: se size = 4,
processo 0 → 0, 4, 8, 12 etc

Realiza um reduce das contagens parciais

Observações

- MPI não se limita a programas Mestre/Escravo
 - Padrão frequente: processos de rank par fazem uma coisa, os ímpares fazem outra coisa.
- Os tags devem ser predefinidos pelo programador.
 - O conjunto (source, tag, dest) identifica a mensagem, então cuidado com possíveis confusões.
- O buffer é uma área de memória contígua
 - Deve-se serializar dados não-contíguos para realizar a comunicação.
- A mensagem tem tamanho fixo, e o recipiente deve conhecê-lo para efetuar o Recv.
 - Normalmente, manda-se primeiro o tamanho, e depois os dados.

dados recebidos
pelo MPI Recv()

Modos de Comunicação (p/ envio de mensagens)

- MPI Standard: A mensagem é enviada e o emissor segue a execução do código. Pode haver buffering temporário de mensagens pequenas e sincronização de mensagens grandes.
- MPI Synchronous: A comunicação é realizada após confirmação do receptor.
 - Qual o problema? 😊
- MPI Ready: A comunicação acontece havendo ou não receptor preparado para receber. Problema? 😞
- MPI Buffered: Para evitar a dependência de sincronização com o receptor, o emissor trabalha com um buffer para armazenamento temporário antes do envio da mensagem. Assim, o emissor pode trabalhar normalmente.
 - MPI_Buffer_attach (void* buf, int size)
 - Informação para criação de buffering local.
 - MPI_Buffer_detach (void **buf, int *size)
 - Informação para não utilização de buffer local.

😊 (misto) dito

MPI_Send()

MPI_Ssend()

MPI_Rsend()

MPI_Bsend()

resultado
+ Nyung ↗

bloqueante
a execução do programa é
interrrompida enquanto a
comunicação não é finalizada.

Comunicação ↗

+ desempenho
não-bloqueante ↗

a execução do programa con-
tinua normalmente, indepen-
dente do resultado da comunicação

Uma aplicação MPI precisa encontrar o menor elemento em um vetor de inteiros gerado no processo mestre. Qual(is) a(s) operação(es) MPI necessária(s) para realizar esta tarefa?

- L- scatter pois o vetor deve ser dividido e enviado p/ os processos
2- cada processo faz a busca pelo menor localmente
 scatter e reduce → 3- o reduce coleta os menores elementos "parciais" e encontra o mínimo global.
- broadcast e gather
 reduce
 allgather
 alltoall

multicore (usa memória compartilhada p/ as threads se comunicarem).

OpenMP na prática: diretivos:

#pragma omp parallel → p/ paralelizar um código sequencial em OpenMP.

- cria uma thread p/ cada núcleo presente que executam o mesmo código. (um for)
(FORK-JOIN)
- o valor das variáveis é compartilhado!

REPLICA O CÓDIGO!

variável

#pragma omp parallel private(i)

↓

- transforma variáveis compartilhadas em privadas.

* É possível que ele gere resultados diferentes, pois não se pode garantir a ordem de execução de threads em paralelo.

#pragma omp parallel for

↓

- distribui as iterações do for entre as threads. Cada iteração do for é executada 1 vez, mas cada thread executa 1 iter. diferente
(MAP)

Divide o trabalho!

* quando threads compartilham uma mesma variável, o valor dela pode ficar imprevisível

* estruturas de repetição não são melhores candidatos p/ a paralelização

* resultados incorretos geralmente indicam conflitos de variáveis compartilhadas.

#pragma omp critical



- cria uma **região crítica** sobre o comando, que impede que dois threads executem um mesmo código ao mesmo tempo.

(REMOVE DISPUTA)

*pragma omp parallel for reduction (op : var)

- Soma os resultados parciais de cada thread até gerar um único valor.

(REDUCE)

* variáveis compartilhadas que acumulam resultados podem ser mapeados no padrão REDUCE, evitando colocar regiões críticas que sequencializam muito o código.
pode incluir um private (var) aqui

operações

variável onde a operação op é aplicada

*pragma(... reduction(... schedule (pol, chunk))



política de distribuição de trabalho

tamanhos das tarefas

- especifica a política de distribuição de trabalho utilizada. As opções são:
(política de escalonamento)

static

chunkSize fixo e definido é distribuído igualmente pelos threads.

CHUNK = 2



T0	T1
1, 2	3, 4
5, 6	7, 8



Loops balanceados, pouca variabilidade

dynamic

chunks fixos, mas não atribuídos dinamicamente às threads conforme eles terminarem suas tarefas "first-come, first-served"

CHUNK = 2



T0	T1
1, 2	3, 4
5, 6	7, 8



Loops com cargas irregulares

guided

começa com pedaços grandes e diminui o tamanho dos chunks exponencialmente até o limite



T0	T1
5, 6, 7, 8	3, 4
2	1



Balanceamento com menor overhead

#pragma omp parallel sections private (i,j)
#pragma omp section



- especifica um escopo onde **vários trechos de códigos diferentes** são executados em paralelo (inclusive laços de repetição)

memória distribuída \Rightarrow usam pacotes de memória p/ os processos se comunicarem.

MPI na prática

↳ Message Passing Interface

`MPI_Init (&argc, &argv);`

`MPI_Comm_rank (MPI_COMM_WORLD, &rank);`

`MPI_Comm_size (MPI_COMM_WORLD, &size);`

`MPI_Finalize();`

comunicador básico (constante)

`MPI_Send ();` } definidos lá em cima no resumo
`MPI_Recv ();`

Escalabilidade

- (capacidade do sistema de manter ou melhorar a eficiência conforme o n° de processadores ou recursos ↑).

FORTE

mede a capacidade de aumentar o desempenho (\downarrow tempo exec) ao aumentar o n° de processadores
- tamanho do problema mantém ↑ threads

vs

FRACA

mede a capacidade de manter a eficiência conforme o n° de processadores aumenta.
↑ tamanho do problema ↑ threads

↳ verificamos o quanto o tempo de exec. diminuiu

↳ verificamos o quanto a eficiência se manteve

Programação híbrida → Open MP + MPI

- Escalabilidade por meio de nós SMP
- exploração eficiente da memória compartilhada em cada nó de um cluster
- + eficiência, desempenho e escalabilidade.

- MPI permite comunicação entre nós.
- MPI facilita o envio de estruturas complexas entre nós.
- Sincronização é explícita.
- Cenário de execução comum.
- Um único processo MPI é iniciado em cada nó do cluster.
- Cada processo dispara N threads no nó do cluster.
- A thread master de cada nó comunica com threads masters de outros nós.
- As threads pertencentes a cada processo são executadas até encontrarem pontos de sincronismo ou finalizarem.