

Introdução ao Hardware e ao Software Paralelos

MCZA020-13 - Programação Paralela

Emilio Francesquini

e.francesquini@ufabc.edu.br

2025.Q2

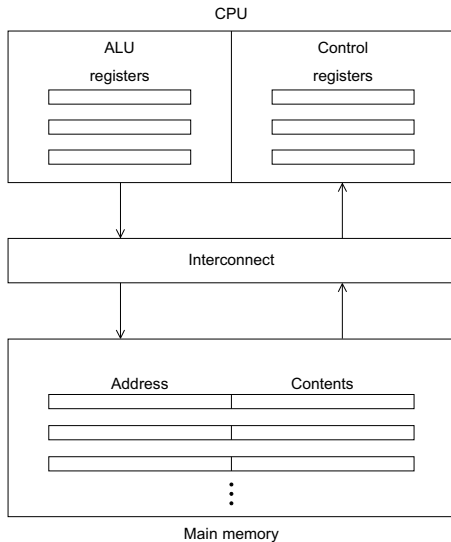
Centro de Matemática, Computação e Cognição
Universidade Federal do ABC



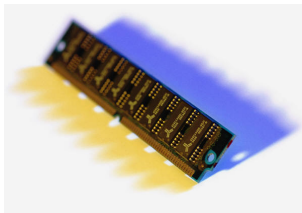
- Estes slides foram preparados para o curso de **Programação Paralela na UFABC**.
- Estes slides são baseados naqueles produzidos por Peter Pacheco como parte do livro *An Introduction to Parallel Programming* disponíveis em:
<https://www.cs.usfca.edu/~peter/ipp/>
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.

Fundamentos





- É uma coleção de posições, cada uma capaz de armazenar **tanto instruções como dados**
- Cada posição consiste de um **endereço** (usado para acessar aquela posição) e seu **conteúdo**



- A **CPU** (*Central Processing Unit*) pode ser dividida em duas partes
- **Unidade de controle (Control Unit)** – é responsável por decidir quais instruções devem ser executadas. (*O chefe*)
- A **Unidade Lógica e Aritimética - ALU (Arithmetic and Logic Unit)**– é responsável por, de fato, executar as instruções. (*O Trabalhador*)



- **Registrador (*Register*)** – Memória com altíssimo desempenho, parte integrada à CPU
- **Contador de Programa - PC (*Program Counter*)** – Armazena o endereço da próxima instrução a ser executada
 - ▶ Processadores Intel usam o nome *Instruction Pointer (IP)*
- **Barramento (*Bus*)** – hardware que conecta a CPU à memória e aos demais dispositivos.



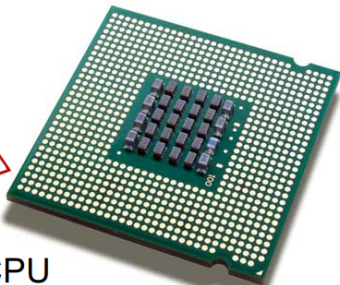
memória



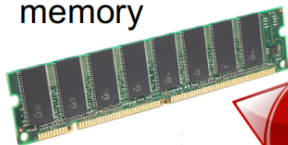
busca (fetch) / leitura (read)



CPU



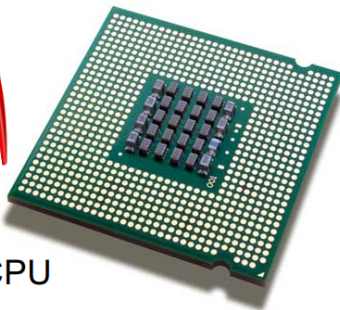
memory

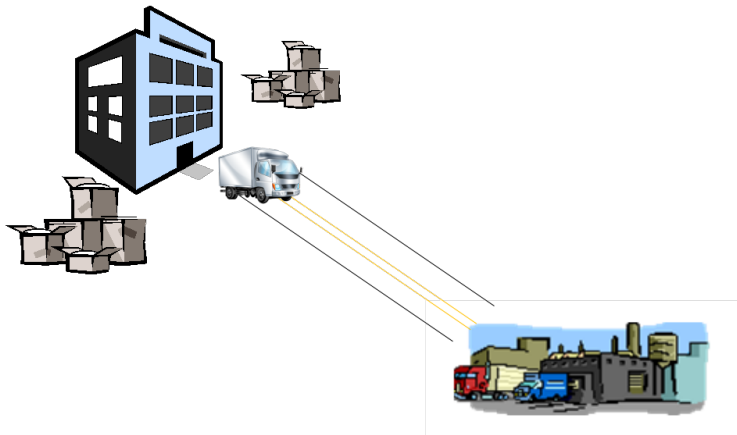


**escrita (write) /
armazena (store)**



CPU

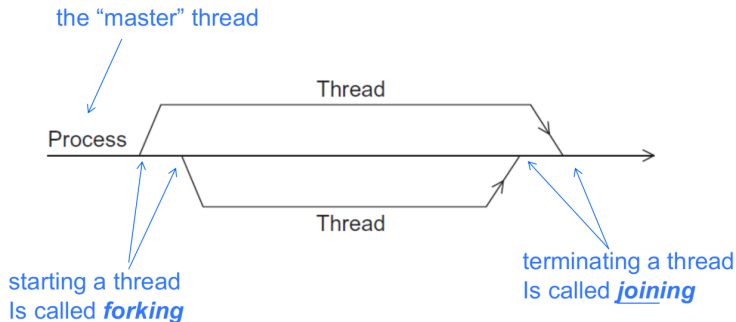




- Um **processo** do **sistema operacional - SO** (*operating system*) é um programa que está sendo executado
- Componentes de um processo
 - ▶ O programa em linguagem de máquina
 - ▶ Regiões da memória
 - ▶ Descritores dos recursos alocados pelo SO ao processo
 - ▶ Informações de segurança
 - ▶ Informações sobre o estado do processo

- **Multitasking** cria a ilusão de que um processador simples, com apenas um **núcleo (core)** de processamento, está rodando múltiplos programas simultaneamente
- Cada processo se reveza no uso do processador (Conceitos: **time share** e **time slice**)
- Quando o tempo alocado a um processo acaba ele é colocado em uma fila e espera novamente pela sua vez. Nestes casos dizemos que o processo está bloqueado (*blocked*)
 - ▶ A entidade responsável por fazer este trabalho é o **escalonador (scheduler)** do SO

- **Threads** elementos que compõem um processo
- Threads permitem os programadores dividirem os seus programas em tarefas virtualmente autônomas e independentes
- A ideia é de que quando um thread bloqueia por estar esperando um recurso, outro thread estará esperando por sua vez
 - ▶ A intenção é promover uma utilização mais otimizada da CPU



Modificações no Modelo de Von Neumann

- **Cache** – Uma coleção de posições de memória que pode ser acessada pelo processador de maneira **muito mais rápida que outras posições de memória**
- A cache de uma CPU é tipicamente localizada **no mesmo chip** ou em uma memória que pode ser acessada **muito rapidamente**
 - ▶ Alguns processadores como o Power8 tem até L4



- O **princípio da localidade** é uma heurística pela qual os projetistas de hardware esperam que o acesso a uma posição de memória seja seguido por acessos a posições em sua vizinhança
- Há dois tipos principais de localidade:
 - ▶ **Localidade Espacial** Acessos ocorrerão em posições de memória próximas.
 - ▶ **Localidade Temporal** Acessos ocorrerão em um futuro próximo.
- As caches de um processador aproveitam-se desses dois casos

```
1 float z[1000];  
2 ...  
3 sum = 0.0;  
4 for (i = 0; i < 1000; i++)  
5     sum += z[i];
```

smallest & fastest



L1

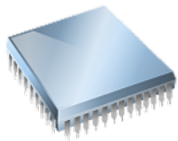


L2

L3



largest & slowest

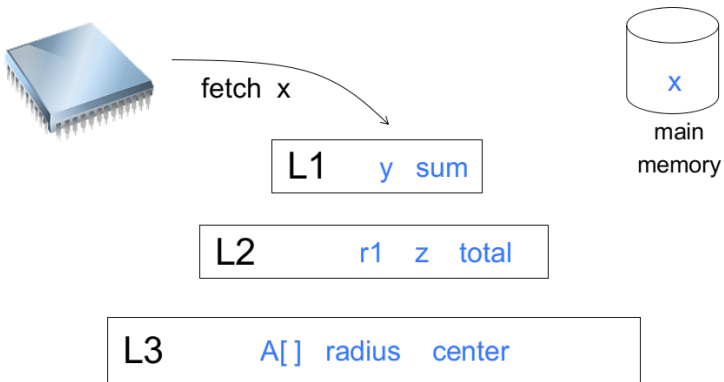


fetch x

L1 x sum

L2 y z total

L3 A[] radius r1 center



- Quando uma CPU escreve dados na cache, o valor daquele dado pode ficar inconsistente, ou defasado (*stale*) em relação aos dados na memória principal.
- **write-through** os dados são escritos na cache e imediatamente enviados para a memória principal
- **write-back** a cache controla os dados armazenados por ela como **sujos** (*dirty*). Quando a linha de cache é substituída por uma nova linha a linha suja é enviada para a memória principal.

- **Full associative, completamente associativa** – uma nova linha pode ser colocada em qualquer posição da cache.
- **Direct mapped, mapeamento direto** – cada linha da cache tem uma posição única onde ela pode ser colocada.
- **n-way set associative, associativa de n vias** – cada linha da cache pode ser colocada em n diferentes posições na cache.

- Quando mais de uma linha na memória pode ser mapeada para diversas posições diferentes, acaba aparecendo a necessidade de definir uma **política de substituição** (*replacement policy/eviction policy*) para decidir qual das linhas precisa ser substituída (*replaced/evicted*)



Memory Index	Cache Location		
	<i>Fully Assoc</i>	<i>Direct Mapped</i>	<i>2-way</i>
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

Associações de uma memória de 16 linhas a uma cache de 4 linhas.

```
double A[MAX][MAX], x[MAX], y[MAX];  
.  
.  
.  
/* Initialize A and x, assign y = 0 */  
.  
.  
.  
/* First pair of loops */  
for (i = 0; i < MAX; i++)  
    for (j = 0; j < MAX; j++)  
        y[i] += A[i][j]*x[j];  
.  
.  
.  
/* Assign y = 0 */  
.  
.  
.  
/* Second pair of loops */  
for (j = 0; j < MAX; j++)  
    for (i = 0; i < MAX; i++)  
        y[i] += A[i][j]*x[j];
```

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

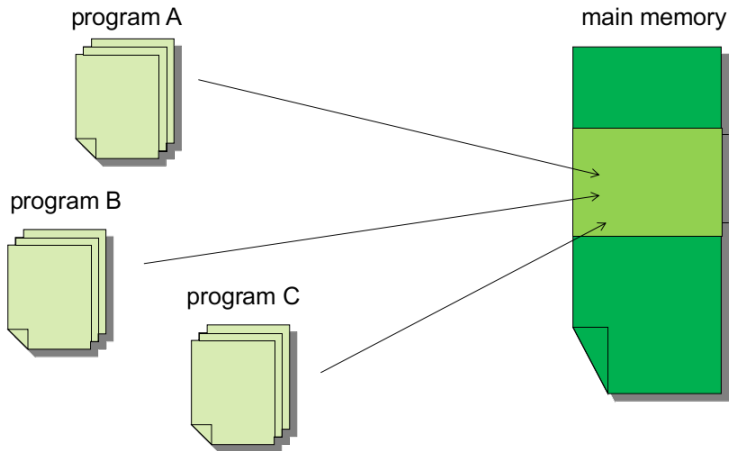
Exemplo: Suponha cache de duas linhas direct mapped.

Memória Virtual

- Se executarmos um programa grande o suficiente ou um programa que acesse muitos dados, pode acontecer da memória principal não ser grande o suficiente.
- A **memória virtual** funciona como uma cache para o dispositivo de armazenamento secundário.

- A memória virtual explora o princípio de localidade espacial e temporal
- Mantém apenas as partes ativas de programas em execução na memória

- **Swap space** - As partes da memória ociosas são transferidas para o espaço de armazenamento secundário
- **Páginas** - Blocos de dados/instruções
 - ▶ Normalmente são relativamente grandes
 - ▶ A maior parte dos sistemas tem um tamanho de páginas fixo que varia entre 4 a 16 kilobytes
 - Alguns sistemas, como o Linux, são capazes de utilizar **huge pages**, que dependendo do processador, podem ser tão grandes quanto 2 megabytes



- Quando um programa é compilado, as suas páginas são associadas à **números de páginas virtuais**
- Quando um programa é executado, uma tabela que mapeia a página virtual aos endereços físicos é criada
- Uma **tabela de páginas** é usada para traduzir um endereço virtual em um endereço físico

Table 2.2 Virtual Address Divided into Virtual Page Number and Byte Offset

Virtual Address									
Virtual Page Number					Byte Offset				
31	30	...	13	12	11	10	...	1	0
1	0	...	1	1	0	0	...	1	1

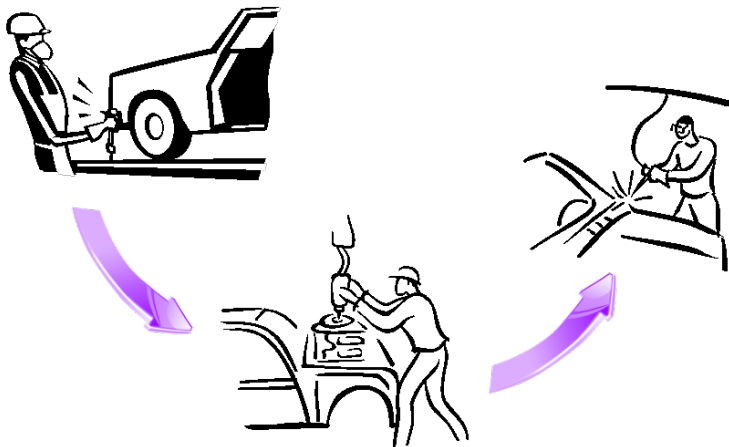
- O simples uso de uma tabela de tradução de endereços poderia causar uma severa perda de desempenho na execução de um programa
- A TLB é uma cache especial cujo propósito é acelerar a tradução de endereços no processador

- A TLB funciona *cacheando* um pequeno número de entradas na tabela de páginas (tipicamente de 16 a 512) da memória principal em uma memória com um desempenho superior.
- **Page Fault** / Falha de página - Ocorre quando um programa em execução tenta acessar um endereço válido na memória física (pós-tradução) mas que não está mais localizado na memória (foi feita a troca/swap para disco)

Instruction Level Parallelism (ILP)

- Tentativa de melhorar o desempenho de um processador através do uso de mais de uma unidade funcional (FU)
 - ▶ A ideia é que se mais de uma FU trabalhar ao mesmo tempo pode-se diminuir o tempo total de execução

- **Pipelining** - as unidades funcionais são organizadas em estágios.
- **Mutiple issue** - múltiplas instruções podem ter a sua execução iniciada ao mesmo tempo



- Somar os números 9.87×10^4 e 6.54×10^3

Time	Operation	Operand 1	Operand 2	Result
0	Fetch operands	9.87×10^4	6.54×10^3	
1	Compare exponents	9.87×10^4	6.54×10^3	
2	Shift one operand	9.87×10^4	0.654×10^4	
3	Add	9.87×10^4	0.654×10^4	10.524×10^4
4	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
5	Round result	9.87×10^4	0.654×10^4	1.05×10^5
6	Store result	9.87×10^4	0.654×10^4	1.05×10^5

```
1 float x[1000], y[1000], z[1000];  
2 ...  
3 for (i = 0; i < 1000; i++)  
4     z[i] = x[ i ] + y[i];
```

- Assuma que cada operação leve um nanossegundo (10^{-9} s)
- Este loop vai levar aproximadamente 7000 nanossegundos

- Divida o *FP adder* em 7 unidades funcionais distintas
- A primeira unidade recupera (*fetch*) os dois operandos, a segunda unidade compara os expoentes, etc...
- Organize as unidades funcionais de modo que a saída de uma seja a entrada da próxima

Table 2.3 Pipelined Addition. Numbers in the Table Are Subscripts of Operands/Results

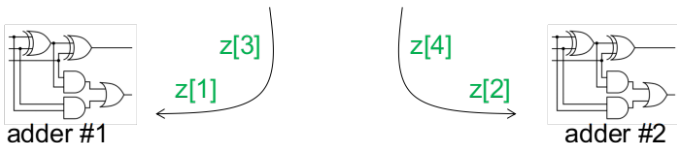
Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

- Uma operação de ponto flutuante **continua** levando 7 nanossegundos
- Mas agora, 1000 operações de ponto flutuante levam **1006 nanossegundos**

- Processadores com *multiple issue* tem unidades funcionais replicadas e tentam executar simultaneamente diferentes instruções de um programa

for (i = 0; i < 1000; i++)

z[i] = x[i] + y[i];



- Multiple issue **estático** - o uso das unidades funcionais é definido em tempo de compilação
- Multiple issue **dinâmico** - o uso das unidades funcionais é determinado em tempo de execução
 - ▶ Processadores com esta característica são chamados de processadores **superescalares**

- Para fazer uso do recurso de *multiple issue*, o processador precisa "achar" instruções para serem executadas de maneira simultânea
- Durante a **especulação**, o compilador ou o processador **chutam** a próxima instrução e torcem para dar certo

Se o sistema especular errado ele deve desfazer os efeitos de todas as instruções especuladas incorretamente.

```
1  z = x + y;  
2  if (z > 0)  
3      w = x;  
4  else  
5      w = y;  
6  
7  //Outro exemplo  
8  z = x + y;  
9  w = *a_p; /* a_p é um ponteiro*/
```



- Nem sempre há boas oportunidades para a execução simultânea de múltiplos threads
- *Hardware multithreading* provê uma solução para os sistemas continuarem a execução e fazer trabalho útil quando a tarefa atualmente sendo executada é *stalled* (temporariamente impedida de executar)
 - ▶ Exemplo: aguardando por um acesso à memória

- **Grão fino** - o processador troca automaticamente de tarefa a cada instrução executada e pula threads que estão *stalled*
 - ▶ **Prós:** potencial de evitar desperdício de tempo do processador devido a stalls
 - ▶ **Contras:** um thread que estiver pronto para executar uma longa sequência de instruções pode ter que esperar para executar cada uma delas

- **Grão grosso** - o processador só troca os threads caso eles tenham entrado em stall
 - ▶ **Prós:** a troca entre threads não é feita a todo momento
 - ▶ **Contras:** o processador pode entrar em ociosidade em stalls curtos e o custo da troca de contexto entre os threads pode ser maior que o ganho

- **Simultaneous Multithreading** (SMT) - é uma variação do multithreading de granularidade grossa
- Permite múltiplos threads fazerem uso de múltiplas unidades funcionais

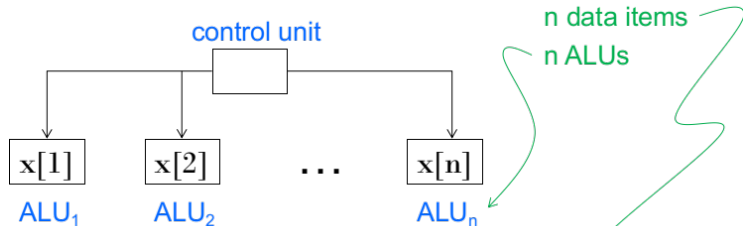
Hardware Paralelo

classic von Neumann

SISD Single instruction stream Single data stream	(SIMD) Single instruction stream Multiple data stream
MISD Multiple instruction stream Single data stream	(MIMD) Multiple instruction stream Multiple data stream

not covered

- Paralelismo é alcançado dividindo os dados entre os processadores
- Aplica a **mesma** instrução a múltiplos dados
- Também chamado de **paralelismo de dados**



```
for (i = 0; i < n; i++)  
    x[i] += y[i];
```

- O que ocorre caso não tenhamos ALUs suficientes?
- Divide-se o trabalho e processa iterativamente
- Exemplo: 4 ALUs e 15 dados

Round3	ALU ₁	ALU ₂	ALU ₃	ALU ₄
1	X[0]	X[1]	X[2]	X[3]
2	X[4]	X[5]	X[6]	X[7]
3	X[8]	X[9]	X[10]	X[11]
4	X[12]	X[13]	X[14]	

- Todas as ALUs precisam executar a **mesma** instrução (ou permanecer ociosa)
- Nos desings mais clássicos, também precisam operar de maneira síncrona
- As ALUs não têm espaço para armazenamento de instruções
- Muito eficiente para problemas grandes e paralelos mas não é eficiente para outros tipos de problemas mais complexos

- Operam em *arrays* ou vetores de dados enquanto processadores convencionais operam em cada um dos dados individualmente (escalares)
- **Registradores vetoriais**
 - ▶ Capazes de armazenar um vetor de operandos e operar simultaneamente em todos os seus elementos

- Unidades funcionais com pipelines e vetorizadas
 - ▶ A mesma operação é aplicada a cada elemento do vetor (ou par de elementos)
- Instruções vetoriais
 - ▶ Operam em vetores em vez de escalares

```
1  for (i = 0; i < n; i++)  
2      x[i] += y[i];
```

- Memória intercalada (*interleaved*)
 - ▶ Múltiplos bancos de memória que podem ser acessados de maneira quase que independente
 - ▶ Distribuem os elementos de um vetor entre diversos bancos de memória o que diminui o tempo para carregar/armazenar diversos elementos
- *Strided memory access* e operações de *scatter* e *gather* por hardware
 - ▶ O programa acessa elementos de um vetor a intervalos fixos.

- Rápidos
- Fáceis de se utilizar
- Compiladores modernos são capazes de extrair vetorização do código automaticamente
 - ▶ Também podem avisar sobre códigos que não puderam ser vetorizados e porquê!
- Alta banda de acesso à memória
- Usa todos os itens da linha do cache

- Não são capazes de lidar com estruturas de dados irregulares tão bem quanto outras estruturas de hardware paralelas
- Tem um limite na sua capacidade de trabalhar com problemas cada vez maiores (limite de **escalabilidade**)

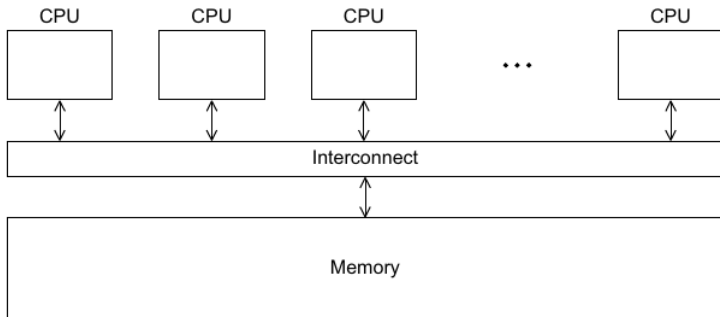
- APIs para geração de gráficos em tempo real usam pontos, linhas, triângulos, ... para representar o estado de um objeto
- Um pipeline gráfico converte a representação interna em uma array de pixels que é enviada à tela
- Vários estágios do pipeline (chamados de **shader functions**) podem ser programados
 - ▶ Tipicamente isto não exige mais do que algumas linhas de código em C

- Shader functions também são implicitamente paralelas uma vez que elas podem ser aplicadas a múltiplos elementos simultaneamente no stream gráfico
- GPUs tipicamente oferecem uma aceleração na execução pelo uso de paralelismo SIMD
 - ▶ Contudo a atual geração de GPUs não é, rigorosamente, um sistema SIMD puro

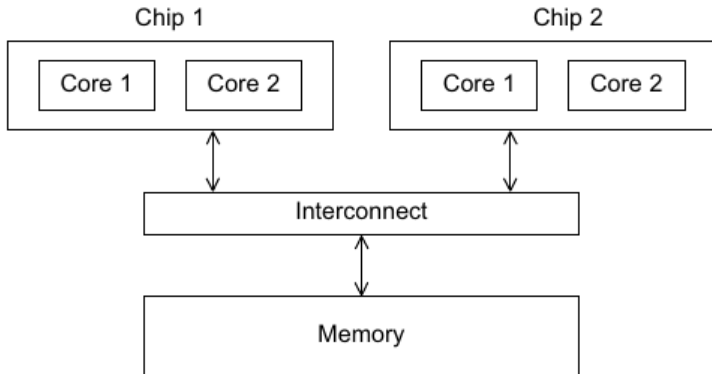
- Dá suporte a múltiplas linhas de execução de instruções simultâneas
- Consiste, tipicamente, de um conjunto de processadores (ou cores) totalmente independentes cada qual com a sua própria unidade de controle e ALU

- Um conjunto de processadores autônomos conectados a um sistema de memória por uma rede de interconexão
- Cada processador pode acessar cada uma das posições de memória
- Os processadores se comunicam, tipicamente, implicitamente pelo acesso a estruturas de dados compartilhadas na memória

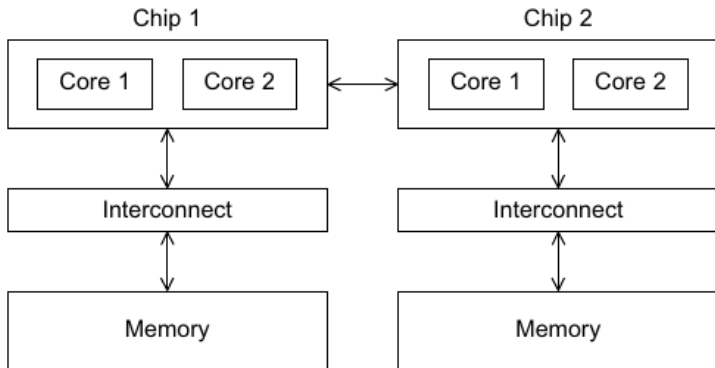
- Os sistemas com memória compartilhada mais comuns oferecem um ou mais processadores multicore
 - ▶ Um ou mais cores ou processadores em um único chip



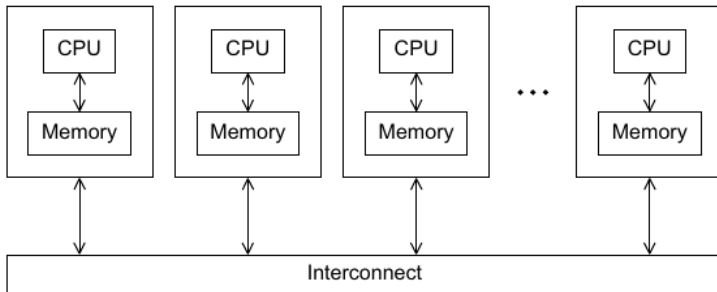
Uniform Memory Access - O tempo para acessar cada uma das posições de memória é sempre o mesmo, independentemente do core ou do endereço



Non-Uniform Memory Access - O tempo de acesso pode variar entre cada um dos cores ou dependendo do endereço



- **Clusters** - Modelo mais popular
 - ▶ Um conjunto de computadores comuns (com memória compartilhada)
 - ▶ Conexão usando redes
- **Nós/Nodos/Nodes** de um cluster são cada uma das unidades computacionais ligadas por uma rede de interconexão
- Também chamados de **sistemas híbridos**



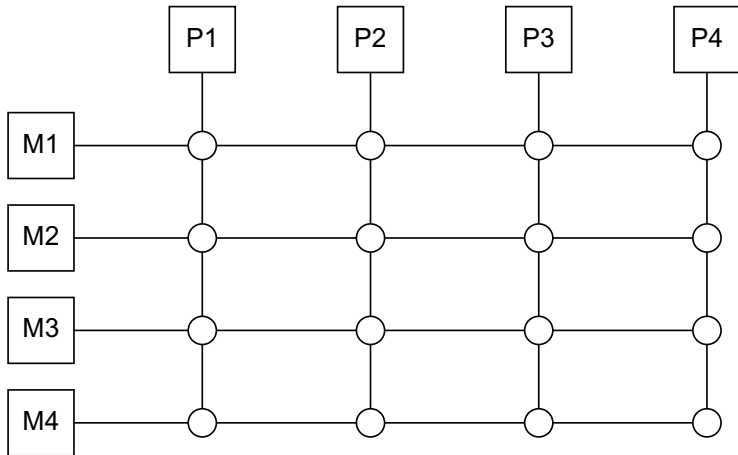
- Afetam o desempenho tanto de sistemas distribuídos quanto de memória compartilhada
- Duas categorias
 - ▶ Interconexões de memória compartilhada
 - ▶ Interconexões de memória distribuída

■ Barramento / *Bus*

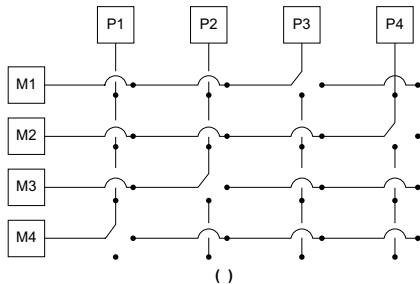
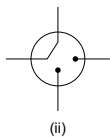
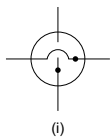
- ▶ Uma coleção de ligações (fios e cabos) paralelos em conjunto com um hardware que controla o acesso ao barramento
- ▶ As vias de conexão são compartilhadas entre os dispositivos conectados
- ▶ Conforme o número de dispositivos aumenta, também aumenta a contenção e, conseqüentemente, há uma diminuição no desempenho

■ Interconexão chaveada / *Switched interconnect*

- ▶ Utiliza chaveadores (*switches*) para controlar o fluxo e o roteamento dos dados entre os dispositivos
- ▶ **Crossbar**
 - Permite comunicação simultânea entre vários dispositivos diferentes
 - Mais rápido que barramentos
 - Contudo, o custo é mais elevado do que barramentos



Crossbar conectando 4 processadores e 4 módulos de memória



Acesso simultâneo à memória por vários processadores

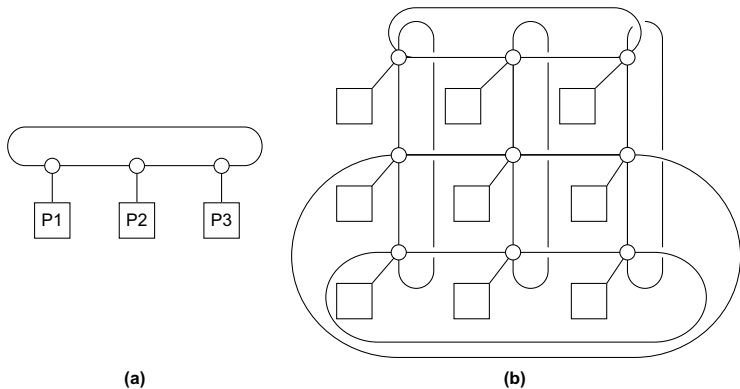
■ Dois grupos

▶ Interconexão direta

- Cada switch é diretamente conectado a um par processador/memória e os switches são conectados uns aos outros

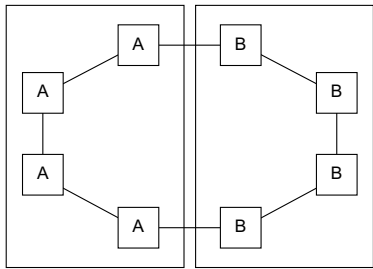
▶ Interconexão indireta

- Os switches podem não estar conectados diretamente a um processador

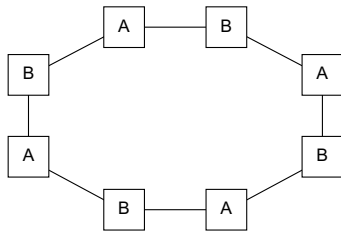


Conexão em (a) anel e (b) toroidal

- Medida do "número de comunicações simultâneas" ou da conectividade
- Quantas comunicações podem ocorrer ao mesmo tempo através de cada parte?

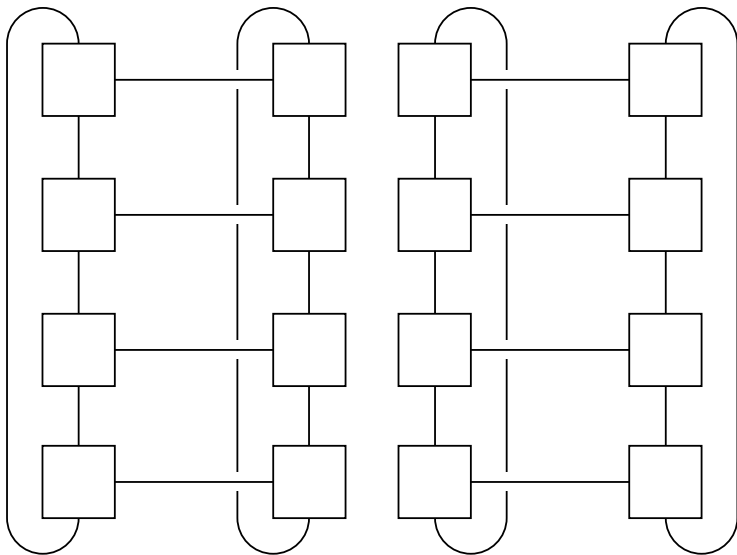


(a)



(b)

(a) 2 conexões (b) 4 conexões



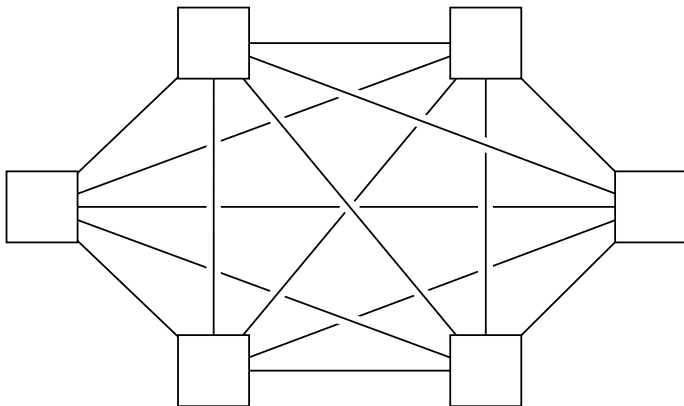
■ Largura de banda / *Bandwidth*

- ▶ Taxa de transmissão de um link
- ▶ Normalmente dada em megabits/gigabits ou megabytes/gigabytes por segundo

■ Largura de banda da bisseção

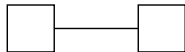
- ▶ Medida da qualidade da rede
- ▶ Em vez de contar o número de links que juntam as duas metades, soma a largura de banda dos links

- Cada switch está conectado a todos os outros switches
- Impraticável para valores grandes de p

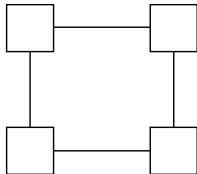


- Largura de bisseção = $\frac{p^2}{4}$

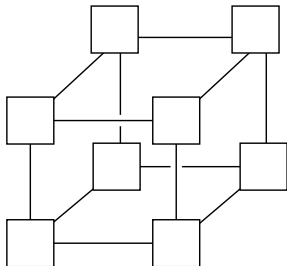
- Interconexão direta e altamente conectada
- Construída por indução
 - ▶ Um **hipercubo unidimensional** é um sistema completamente conectado com dois processadores
 - ▶ Um **hipercubo bidimensional** é construído a partir de dois hipercubos unidimensionais pela junção dos switches "correspondentes"
 - ▶ Da mesma maneira, um **hipercubo tridimensional** é um hipercubo construído a partir de dois hipercubos bidimensionais



(a)



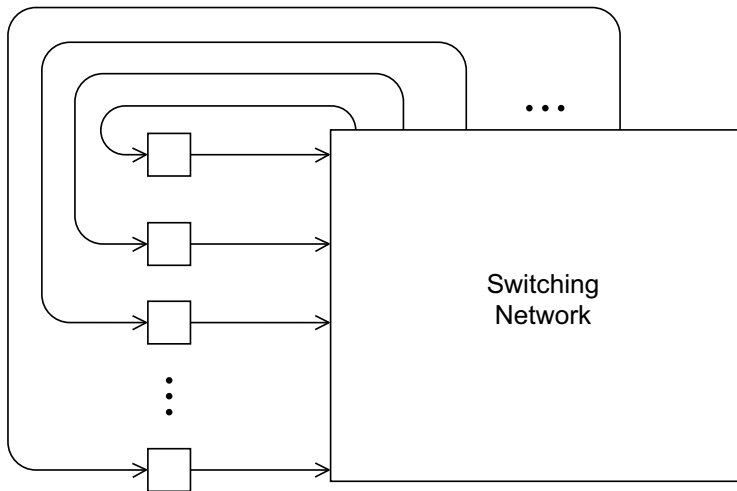
(b)

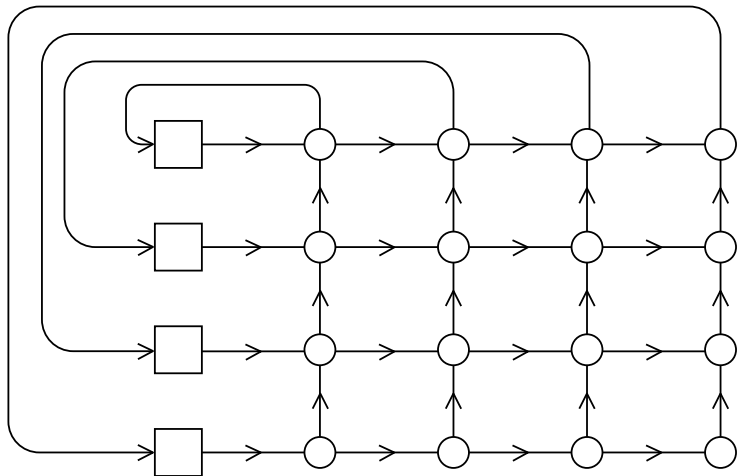


(c)

(a) 1D (b) 2D (c) 3D

- Exemplos mais simples de redes de interconexão indiretas
 - ▶ Crossbar
 - ▶ Omega Network
- Geralmente são mostradas como redes com links unidirecionais e com uma coleção de processadores, cada um deles com um link de entrada e outro de saída, e uma rede chaveada



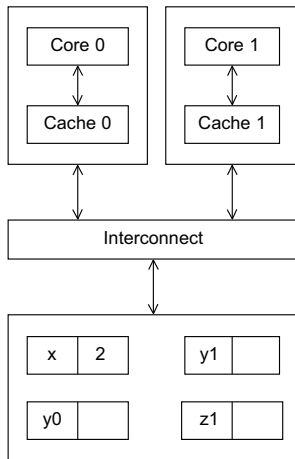


- Para toda troca de dados é importante sabermos quanto tempo que os dados alcancem o seu destino
- **Latência** - O tempo que leva entre a origem **começar** a transmitir os dados e o destino **começar** a receber o primeiro byte
- **Largura de banda** - A taxa pela qual o destinatário recebe os dados após ter começado a receber o primeiro byte

Tempo de transmissão

$$\text{Tempo} = \text{Latência (s)} + \text{Tamanho (bytes)} / \text{Banda (bytes/s)}$$

- Programadores não tem quase nenhum controle sobre o funcionamento das caches
 - ▶ Conteúdo
 - ▶ Política de substituição
 - ▶ Algumas ISAs permitem marcar regiões da memória como não-cacheáveis ou ainda fixar algumas linhas na cache.}



- Suponha:
 - ▶ `y0` mantida privada pelo Core 0
 - ▶ `y1` e `z1` mantidas privadas pelo Core 1
 - ▶ `x = 2;` Variável compartilhada

Time	Core 0	Core 1
0	<code>y0 = x;</code>	<code>y1 = 3*x;</code>
1	<code>x = 7;</code>	Statement(s) not involving x
2	Statement(s) not involving x	<code>z1 = 4*x;</code>

- `y0` termina com 2
- `y1` termina com 6
- `z1` termina valendo?

- Baseado no fato de que os cores compartilham o barramento
- Qualquer sinal transmitido no barramento pode ser "visto" por todos os cores
- Quando o Core 0 atualiza a sua cópia de x armazenada em sua cache, ele também avisa (*broadcast*) esta informação no bus
- Se o Core 1 estiver "escutando"/"snooping" o barramento, ele percebe que x foi atualizado e marca a sua cópia local como inválida

- Usa uma estrutura de dados chamada **diretório** que armazena o status de cada uma das linhas de cache
- Quando uma variável é atualizada, o diretório é consultado e os controladores de cache dos cores que tiverem uma cópia daquela variável em seus caches têm as suas linhas invalidadas

Software paralelo

- Mudança na abordagem comum até o início dos anos 2000
 - ▶ Compiladores e software vão ter que por a mão na massa
- De agora em diante
 - ▶ Em programas de memória compartilhada
 - Inicie um único processo e várias threads
 - Threads cuidam das tarefas
 - ▶ Em programas de memória distribuída
 - Inicie vários processos
 - Processos cuidam das tarefas

- Um programa SPMD consiste em um executável único que pode se comportar de maneira diferente (como se fossem múltiplos programas) através do uso de condicionais

```
if (I'm thread process i)  
    do this;  
else  
    do that;
```



- 1 Divida o trabalho entre os processos/threads
 - ▶ De modo que cada processo/thread receba aproximadamente o mesmo tanto de trabalho
 - ▶ E de modo que a comunicação seja minimizada
- 2 Organize o código para que os processos/threads se sincronizem
- 3 Organize o código para que os processos/threads se comuniquem

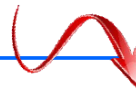
```
double x[n], y[n];  
...  
for (i = 0; i < n; i++)  
    x[i] += y[i];
```

- Threads dinâmicos
 - ▶ Thread principal (*master*, *main*) espera por trabalho, cria novos threads, espera a sua conclusão, finaliza
 - ▶ Uso eficiente dos recursos, contudo criar e descartar threads é custoso
- Threads estáticos
 - ▶ *Pool* de threads - Threads são criados e colocados para trabalhar, mas não são descartados até o final da execução
 - ▶ Melhor desempenho, mas pode ser uma fonte de desperdício de recursos

```
...  
printf ( "Thread %d > my_val = %d\n" ,  
        my_rank , my_x ) ;  
...
```



Thread 1 > my_val = 19
Thread 0 > my_val = 7



Thread 0 > my_val = 7
Thread 1 > my_val = 19

```
1 my_val = Compute_val(my_rank);  
2 x += my_val;
```

Time	Core 0	Core 1
0	Finish assignment to my_val	In call to Compute_val
1	Load x = 0 into register	Finish assignment to my_val
2	Load my_val = 7 into register	Load x = 0 into register
3	Add my_val = 7 to x	Load my_val = 19 into register
4	Store x = 7	Add my_val to x
5	Start other work	Store x = 19

- Condições de corrida (*race condition*)
- Seção crítica
- Exclusão mútua
- Trava de exclusão mútua (*mutual exclusion lock, mutex*)

```
my_val = Compute_val ( my_rank ) ;  
Lock(&add_my_val_lock ) ;  
x += my_val ;  
Unlock(&add_my_val_lock ) ;
```

```
1 my_val = Compute_val(my_rank);
2 if (my_rank == 1)
3     while (!ok_for_1); /* Busy-wait loop */
4 x += my_val ; /* Critical section */
5 if (my_rank == 0)
6     ok_for_1 = true; /* Let thread 1 update x */
```

```
1 char message [100];
2 ...
3 my_rank = Get_rank();
4 if (my_rank == 1) {
5     sprintf( message , "Greetings from process 1");
6     Send(message, MSG_CHAR, 100 ,0);
7 } else if (my_rank == 0) {
8     Receive(message, MSG_CHAR, 100, 1);
9     printf("Process 0 > Received: %s\n", message);
10 }
```

```
1  shared int n = ...;
2  shared double x[n] , y[n];
3  private int i, my_first_element, my_last_element;
4  my_first_element = ...;
5  my_last_element = ...;
6  /* Initialize x and y */
7  ...
8  for (i = my_first_element; i <= my_last_element; i++)
9      x [i] += y[i];
```

- Em um programa com memória distribuída, apenas o processo 0 vai acessar a **stdin**
- Em um programa com memória compartilhada, apenas o thread principal vai acessar a **stdin**
- Em ambos os casos, todos os processos/threads podem acessar a **stdout** e a **stderr**

- Contudo, devido ao não-determinismo da ordem da saída para **stdout** tipicamente restringe-se o uso do **stdout** da mesma maneira que fazemos para a **stdin**
 - ▶ Exceção notável: depuração de código
- Saída de depuração deve sempre incluir o número do processo/thread para permitir identificação do gerador da mensagem

- Apenas um processo deve tentar acessar um arquivo por vez (excluindo-se `stdin`, `stdout` e `stderr`)
- Por exemplo, cada processo/thread pode abrir seu próprio arquivo de modo privado para leitura e escrita, mas dois processos distintos deveriam abrir o mesmo arquivo ao mesmo tempo

Desempenho

- Número de cores/processadores = p
- Tempo de execução sequencial = $T_{\text{sequencial}}$
- Tempo de execução paralelo = T_{paralelo}

$$\text{Se } T_{\text{paralelo}} = \frac{T_{\text{sequencial}}}{p} \rightarrow \text{speedup linear}$$

Qual é o tempo sequencial que deve ser usado?

- Melhor implementação sequencial?
- Implementação base da paralelização?

$$S = \frac{T_{\text{sequencial}}}{T_{\text{paralelo}}}$$

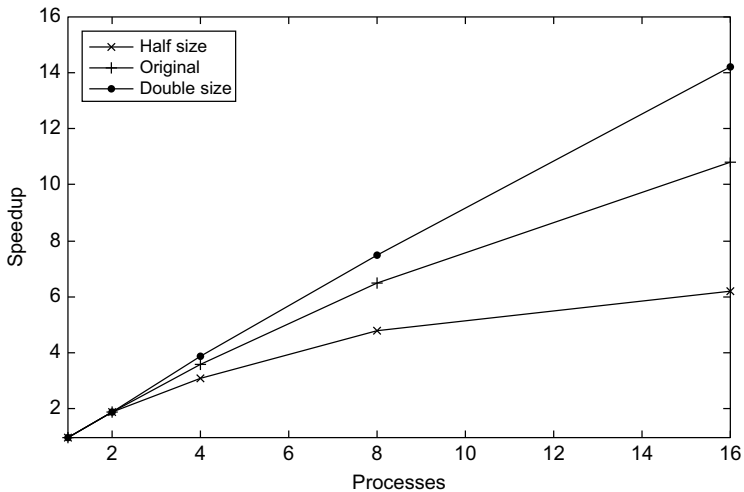
$$E = \frac{S}{P} = \frac{\frac{T_{\text{sequencial}}}{T_{\text{paralelo}}}}{p} = \frac{T_{\text{sequencial}}}{p \cdot T_{\text{paralelo}}}$$

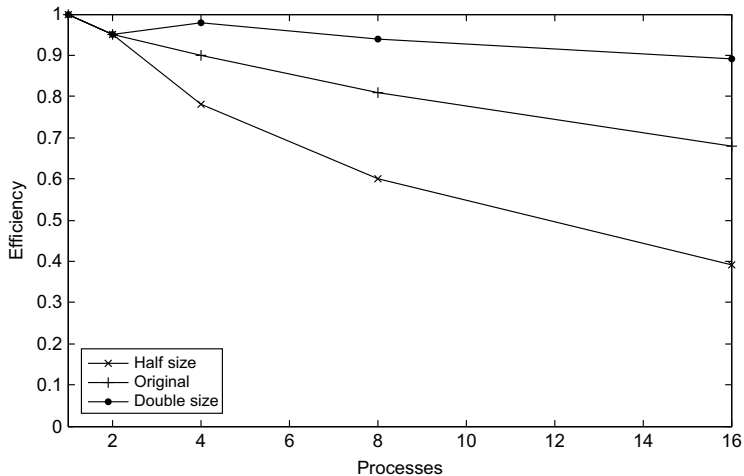
Table 2.4 Speedups and Efficiencies of a Parallel Program

p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
$E = S/p$	1.0	0.95	0.90	0.81	0.68

Table 2.5 Speedups and Efficiencies of a Parallel Program on Different Problem Sizes

	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89





$$T_{\text{paralelo}} = \frac{T_{\text{sequencial}}}{p} + T_{\text{overhead}}$$

- Um programa paralelo passa a ter overheads de comunicação, divisão do trabalho, controle para entrada em regiões críticas,
- São raríssimas as ocasiões onde é possível ter um speedup perfeitamente linear (ou superlinear)

- Observação feita na década de 60 por Gene Amdahl
 - ▶ Mais tarde passou a ser conhecida como A lei de Amdahl
- Para explicar, pense em um exemplo:
 - ▶ $T_{\text{sequencial}} = 20$
 - ▶ Parte paralelizável do programa 90%
- Qual é o speedup **máximo** (independentemente do número de processadores) que podemos alcançar?

A menos que a totalidade de um programa sequencial possa ser paralelizado, o speedup máximo é limitado independentemente do número de cores disponíveis.

- No nosso exemplo o speedup máximo é $10\times$
- De modo geral, o speedup máximo é $\frac{1}{r}$ onde r é a fração do programa que não é paralelizável

E agora, jogamos a toalha?

A realidade dos programas paralelos não é tão sombria.
Quanto maior o problema menor a parte sequencial
(proporcionalmente, ou seja $r \rightarrow 0$)

- Há milhares de exemplos de problemas com speedups altíssimos usados todos os dias
- Nem sempre um speedup de 10 ou 20 é pouco, ainda mais se o esforço para a paralelização for pequeno

- De maneira informal dizemos que um problema (ou a sua solução) é **escalável** (*scalable*), se é possível tratar problemas cada vez maiores
 - ▶ Porém aqui precisaremos de uma definição mais precisa.

- Suponha que rodamos a solução para um problema com um determinado número de processos e tamanho de entrada e obtemos uma eficiência E
- Suponha também que aumentemos o número de processos

Dizemos que o problema é escalável se é possível achar um aumento correspondente no tamanho da entrada de modo que a eficiência E permaneça constante.

Suponha que

- $T_{\text{sequencial}} = n$
- $T_{\text{paralelo}} = \frac{n}{p} + 1$

Este problema é escalável?

$$E = \frac{n}{p(\frac{n}{p}+1)} = \frac{n}{n+p}$$

- Para verificar se o problema é escalável, vamos aumentar o tamanho do problema em um fator de k
- Precisamos encontrar x tal que E permaneça constante

Então

$$E = \frac{n}{n+p} = \frac{xn}{xn+kp}$$

$$E = \frac{n}{n+p} = \frac{xn}{xn+kp}$$

Tomando $x = k$ temos:

$$\frac{xn}{xn+kp} = \frac{kn}{kn+kp} = \frac{n}{n+p} = E$$

O problema é escalável.

Escalabilidade forte

A eficiência é mantida à medida que p cresce **sem** um aumento no tamanho do problema.

Escalabilidade fraca

A eficiência é mantida à medida que p cresce quando também há um aumento proporcional do tamanho do problema.

Qual é o caso do nosso exemplo?

- O que medir?
 - ▶ Tempo de processamento
 - Descarta o tempo de carga, por exemplo?
 - E quanto ao tempo de E/S?
 - ▶ Tempo de comunicação
 - ▶ **Tempo de parede** (*wall-clock time*)
- Como apresentar as medidas?
 - ▶ Média
 - ▶ Mediana
 - ▶ Máximo
 - ▶ Mínimo

```
double start, finish;
. . .
start = Get_current_time();
/* Code that we want to time */
. . .
finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);
```

- Função "teórica"
- Funções reais: `MPI_Wtime`, `omp_get_wtime`

Na verdade é preciso cuidar de muitos detalhes como:

- Variabilidade
- Resolução

E no caso de programas paralelos?

```
private double start, finish;  
.  
.  
.  
start = Get_current_time();  
/* Code that we want to time */  
.  
.  
.  
finish = Get_current_time();  
printf("The elapsed time = %e seconds\n", finish-start);
```

```
shared double global_elapsed;
private double my_start, my_finish, my_elapsed;
. . .
/* Synchronize all processes/threads */
Barrier();
my_start = Get_current_time();

/* Code that we want to time */
. . .

my_finish = Get_current_time();
my_elapsed = my_finish - my_start;

/* Find the max across all processes/threads */
global_elapsed = Global_max(my_elapsed);
if (my_rank == 0)
    printf("The elapsed time = %e seconds\n", global_elapsed);
```

É comum o uso de **barreiras** (*barriers*) em programas paralelos quando precisamos que todas as linhas de execução estejam no mesmo ponto.

Design de programas paralelos

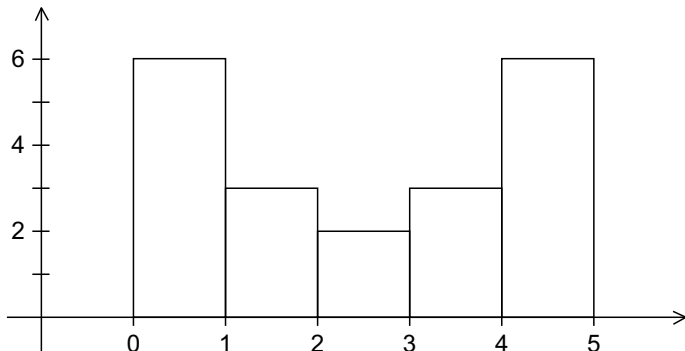
- Precisamos dividir o trabalho em tarefas menores e aproximadamente do mesmo tamanho
- Precisamos minimizar a comunicação

Ian Foster sistematizou estas diretrizes no que ficou conhecido como **Metodologia de Foster**

- 1 **Particionamento** - dividir a computação e os dados sobre os quais devemos trabalhar em pequenas tarefas. O foco principal deve ser em identificar tarefas que podem ser executadas em paralelo.
- 2 **Comunicação** - determinar o que precisa ser comunicado entre as tarefas identificadas no passo anterior

- ③ **Aglomeração/Agregação** - combinar as tarefas e as comunicações identificadas no primeiro passo em tarefas maiores. Por exemplo, se a tarefa A só pode executar depois de B, pode fazer sentido agregá-las em uma única tarefa maior.
- ④ **Mapeamento** - atribuir as tarefas compostas no passo 3 para processos/threads. Isto deve ser feito de maneira a minimizar a comunicação e de modo a equilibrar a quantidade de tarefas para cada trabalhador.

- 1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9



- 1 O número de medidas: `data_count`
- 2 Um vetor de `data_count` floats: `data`
- 3 O valor mínimo para a *bin* contendo os menores valores:
`min_meas`
- 4 O valor máximo para a *bin* contendo os maiores valores:
`max_meas`
- 5 O número de bins: `bin_count`

- ① `bin_maxes`: um vetor de `bin_count` floats
- ② `bin_coutns`: um vetor de `bin_count` ints

Um *hotspot* é um trecho “quente” do código, ou seja, um trecho que gasta boa parte do tempo total de processamento.

```
for (i = 0; i < data_count; i++) {  
    bin = Find_bin(data[i], bin_maxes, bin_count, min_meas);  
    bin_counts[bin]++;  
}
```

