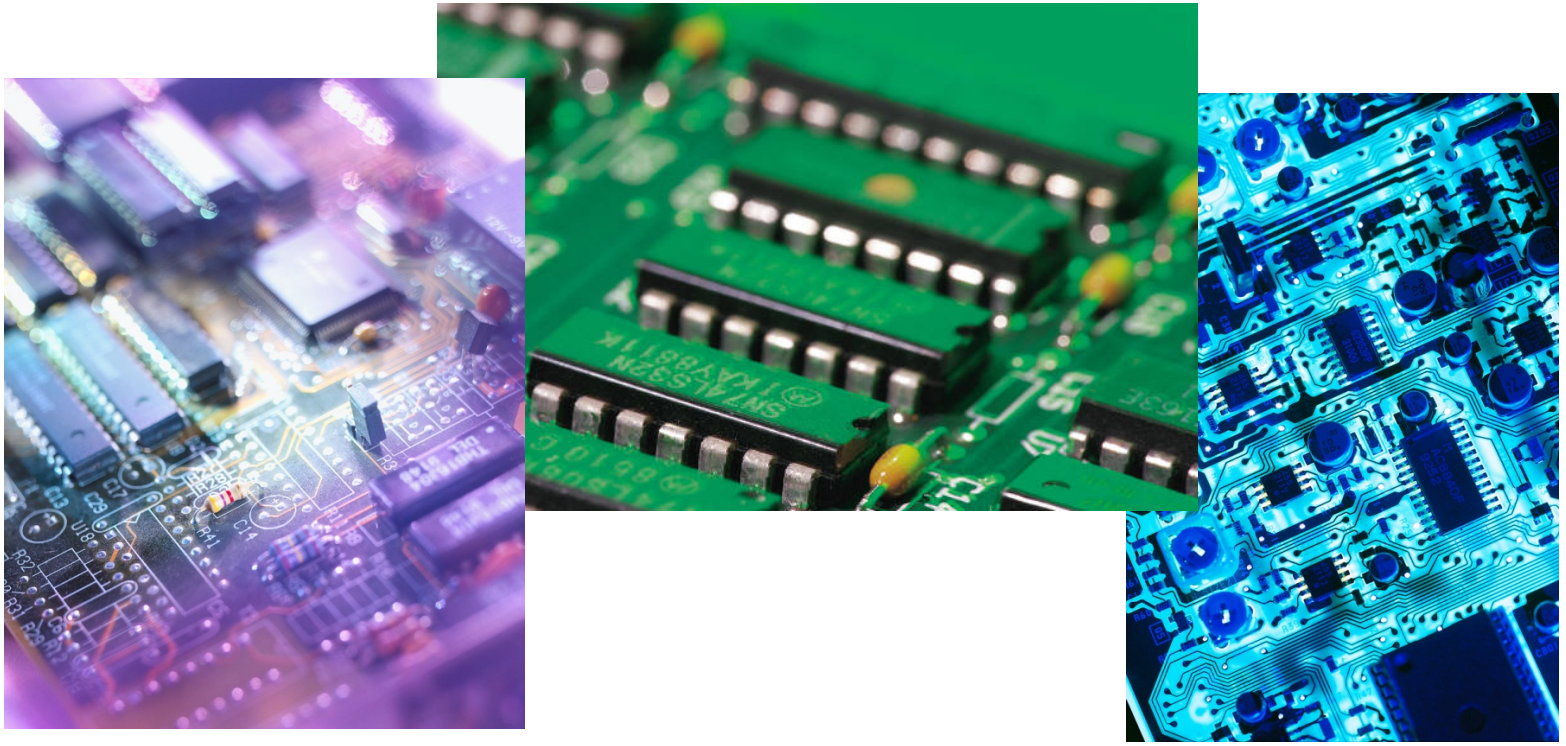
MO644/MC900
Conceitos Básicos
Hardware/Software Paralelos

Prof. Guido Araujo
www.ic.unicamp.br/~guido

Roadmap

- Background
- Hardware paralelo
- Software paralelo
- Entrada e saída
- Desempenho
- Projeto de programas paralelos
- Escrevendo e executando programas paralelos
- O que estamos assumindo



BACKGROUND

Arquitetura de von Neumann

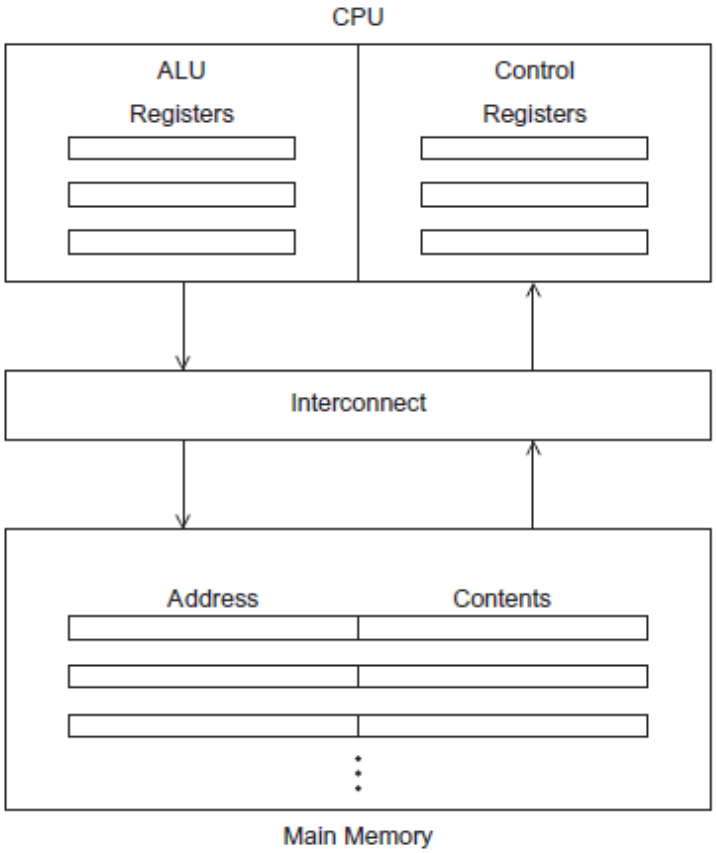
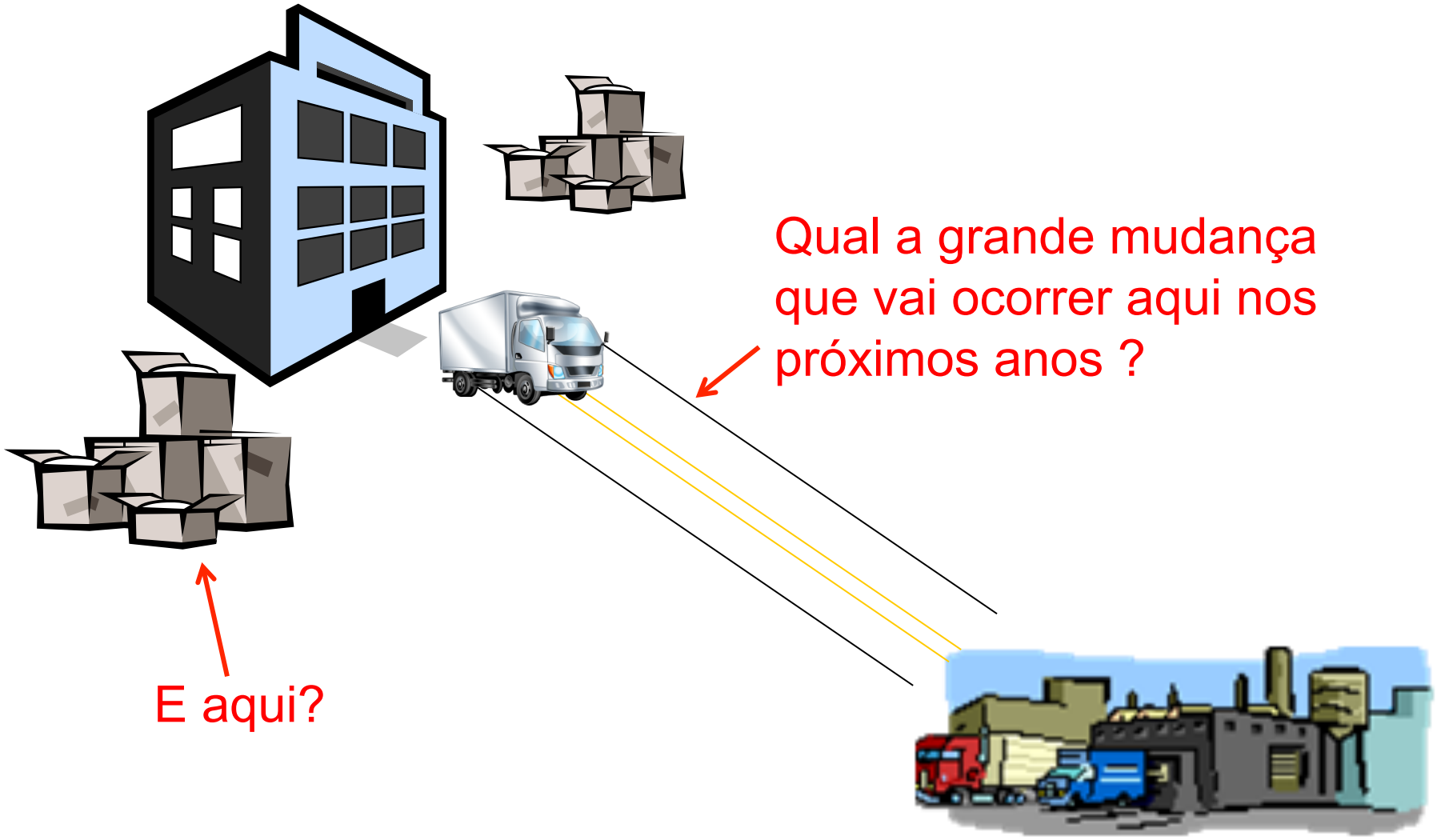


Figure 2.1

Gargalo de von Neumann



Um “processo” do Sistema Operacional

- Uma instância de um programa de computador que está sendo executada.
- Componentes de um processo:
  - Programa em código de máquina executável.
  - Um bloco de memória.
  - Descritor dos recursos que o SO alocou para o processo.
  - Informações de segurança.
  - Informações sobre o estado do processo.

Multitasking

- Cria a ilusão que um único processador está executando múltiplos programas simultaneamente.
- Na verdade cada processo executa, por vez, alternadamente. (**time slice**)
- Depois que o seu tempo acabou, ele espera até que a sua vez chegue novamente.

Threading

- *Threads* estão contidas dentro de processos.
- Elas permitem o programador dividir o seu programa em tarefas (mais ou menos) independentes.
- A esperança é que se uma thread bloquear esperando por um recurso, alguma outra terá trabalho a fazer enquanto isto.

Um processo e duas threads

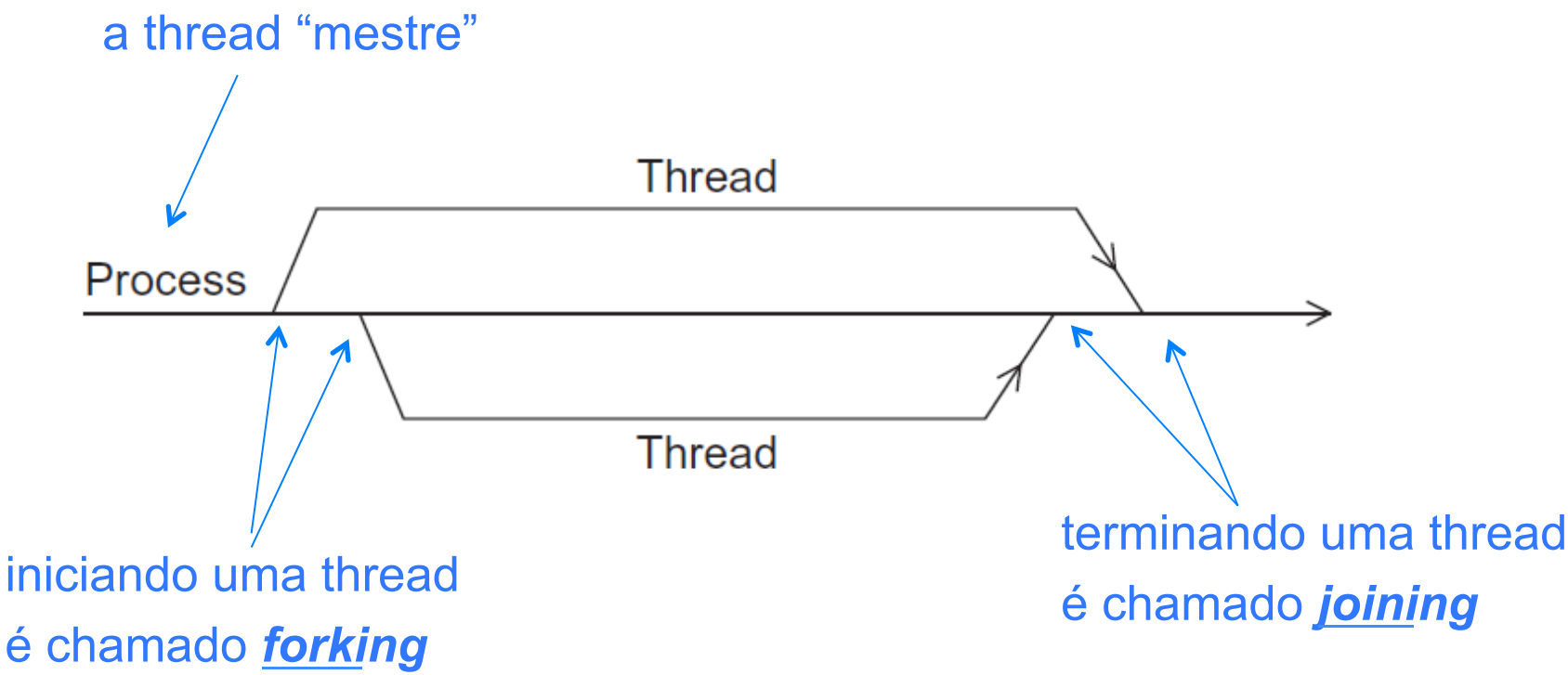
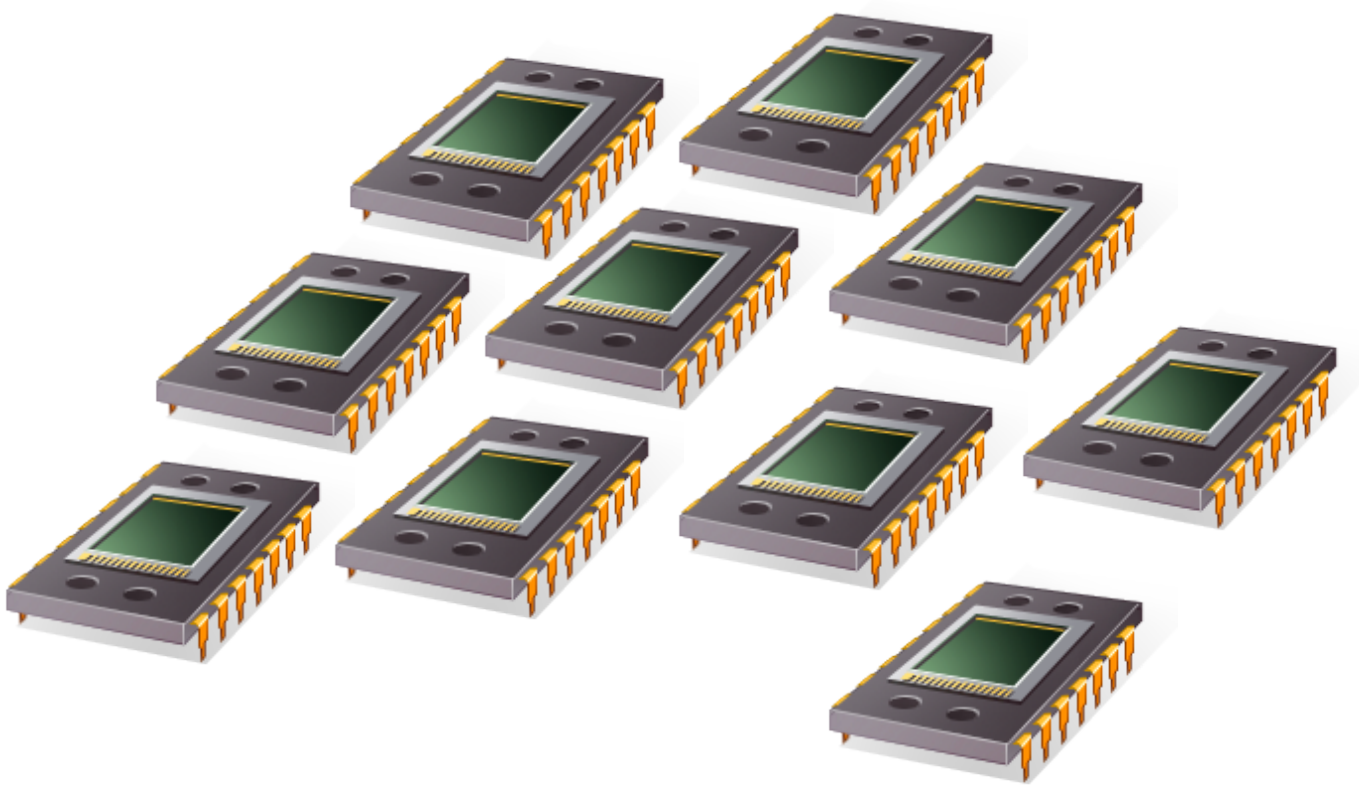


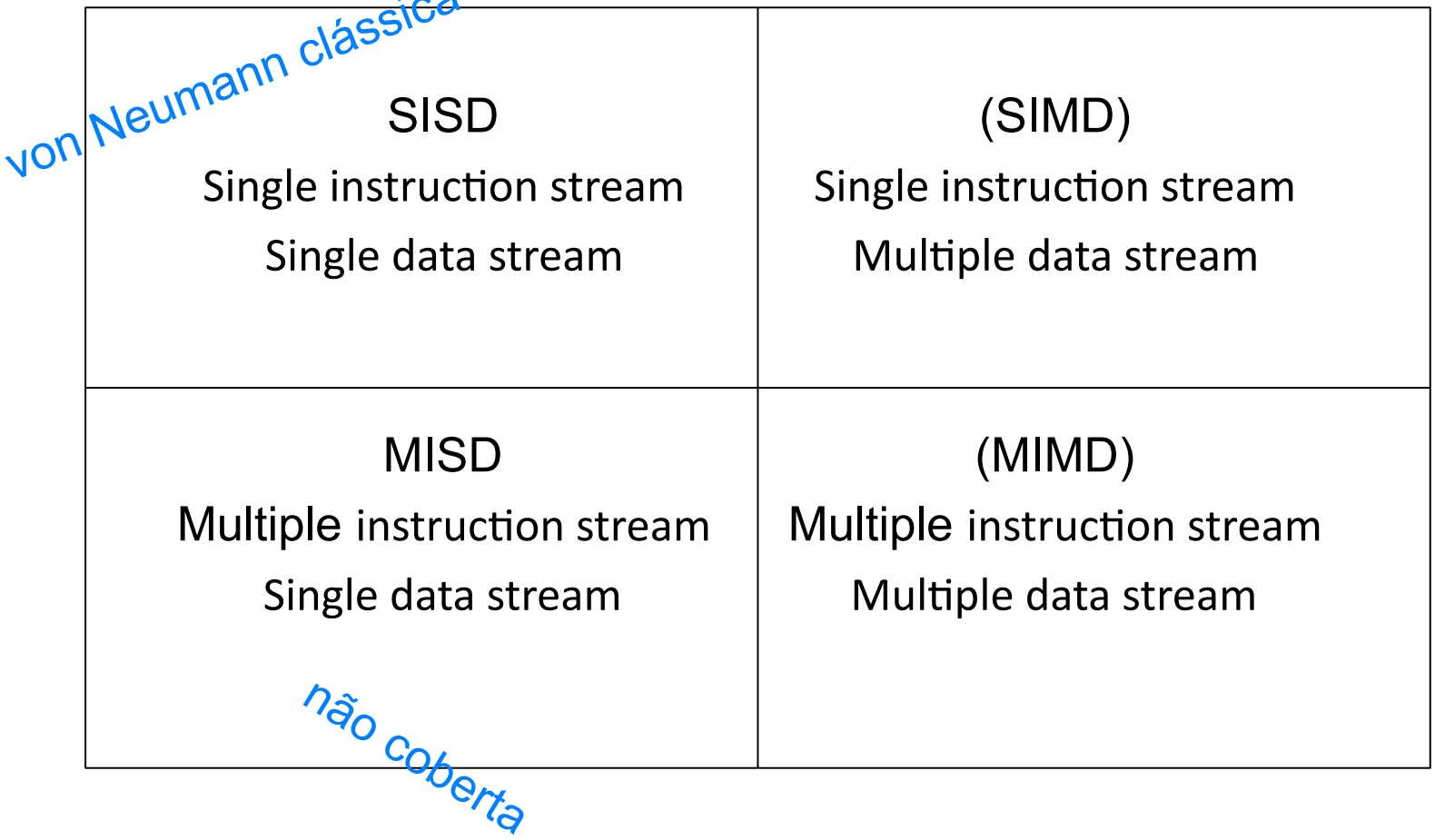
Figure 2.2



Um programador pode escrever código para explorar.

HARDWARE PARALELO

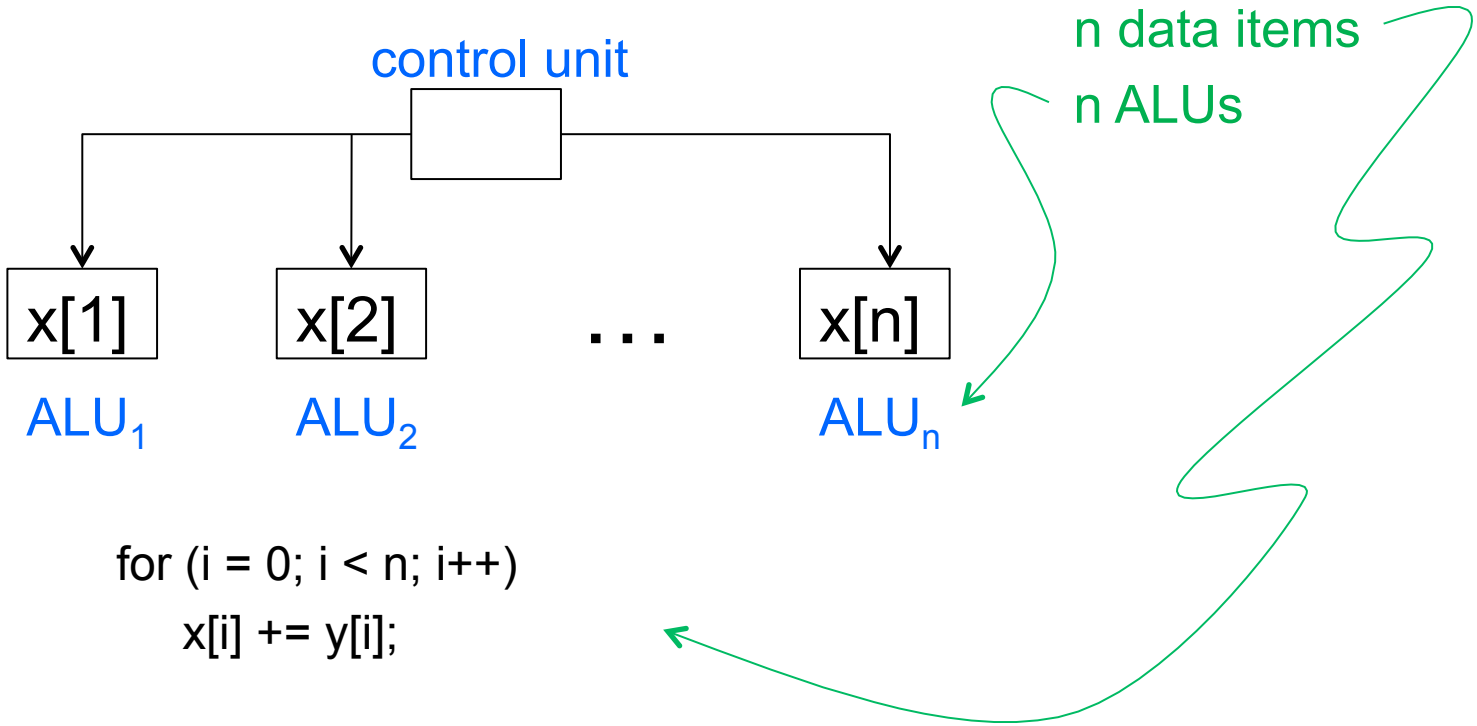
Taxonomia de Flynn



SIMD

- Paralelismo é obtido através da divisão de dados entre os processadores.
- Aplica a mesma instrução a vários dados.
- Chamado data parallelism.

SIMD example



SIMD

- E se não tivermos tantas ALUs quanto dados?
- Divide o trabalho e processa iterativamente.
- Ex. m = 4 ALUs e n = 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]	

Desvantagens de SIMD

Qual a vantagem disto?

- Todas as ALUs executam a mesma instrução, ou permanecem ociosas.
- Em um projeto clássico elas devem executar sincronizadas.
- Solução eficiente para problemas *data parallel* grandes, mas não é útil para problemas paralelos mais complexos.

Processadores Vetoriais (1)

- Operam em arrays ou vetores de dados, enquanto CPUs convencionais operam em dados individuais ou escalares.

- Registradores vetorias.
  - Capazes de armazenar um vetor de operandos e operar simultaneamente no seu conteúdo.

Processadores Vetoriais (2)

- Usam unidades funcionais vetoriais e "pipelined".
  - A mesma operação é aplicada a cada elemento do vetor (ou pares de elementos).

- Instruções vetoriais.
  - Opera em vetores ao invés de escalares.

Exemplo (1)

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

    load v1, &x[0]
    load v2, &y[0]
    add v1, v1, v2
```

Exemplo (2)

Lidando com condições

v1 : [-2, 3, -1, 10]

v2 : [5, 1, 4, 2]

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

    load v1, &x[0]
    load v2, &y[0]
    cmp v1 > 0x0, m1
    addc v1, v1, v2, m1
```

O que faz esta instrução?

m1 : [0, 1, 0, 1]

E esta?

v1 : [-2, 4, -1, 12]

Processadores Vetoriais - Prós

- Veloz.
- Fácil de usar.
- Compiladores são bons em identificar vetores e extrair paralelismo.
- Compiladores são bons em fornecer informações sobre código que não pode ser vetorizado.
  - Ajuda o programador a re-avaliar o código.
- Alta largura de banda de memória.
- Usa cada item da cache.

Processadores Vetoriais- Contras

- Não conseguem lidar com estruturas de dados irregulares tão bem quanto outras arquiteturas paralelas .

- Possuem um limite claro na sua habilidade de lidar com problemas maiores. (escalabilidade)

GPUs

- Baseada em paralelismo estilo SIMD.
  - Apesar de que não são sistemas SIMD puros.
- Divide um grande número de threads idênticas entre vários núcleos
- Cada thread executa a mesma instrução, mas permite fluxos diferentes simultâneos

GPU (Exemplo)

```
1: if (a[i] < 4)
2:     z += 1 ;
3: else
4:     z -= 1;
```

Assume: [0, 1, 7, 13]

T0 (i=1):

```
if (a[i] < 4)
    z += 1;
    nop;
```

T1 (i=2):

```
if (a[i] < 4)
    nop;
    z -= 1;
```

T2 (i=3):

```
if (a[i] < 4)
    nop;
    z -= 1;
```

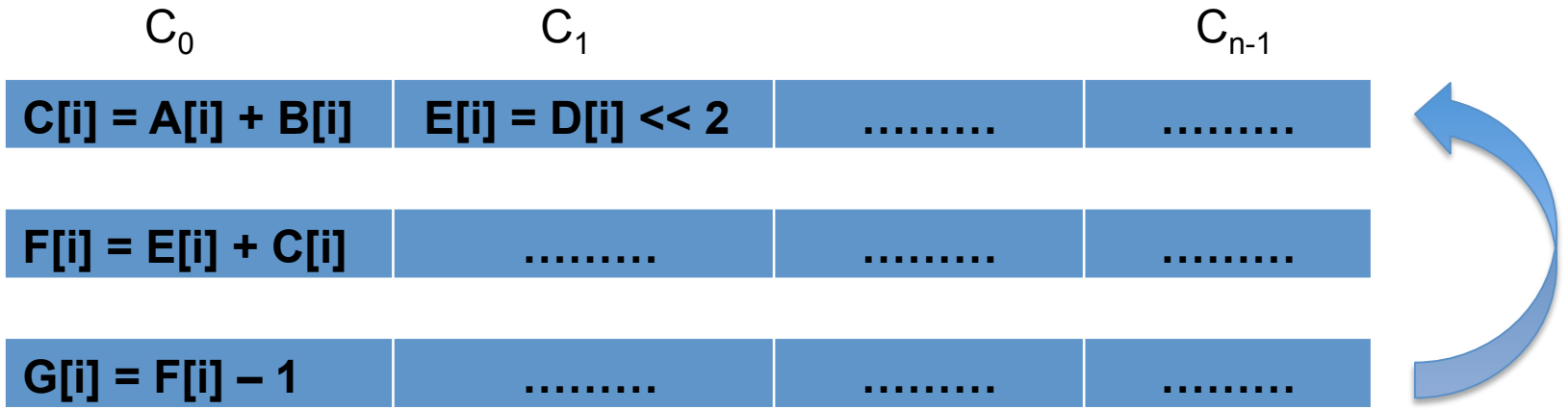
MIMD

- Permitem a execução simultanea de vários fluxos de instruções em vários fluxos de dados.

- Consiste tipicamente de um conjunto de unidades de processamento ou núcleos, independentes, cada com a sua própria unidade de controle e ALU.

Exemplo

```
for (i = 0; i < N; i++) {
        C[i] = A[i] + B[i];
        E[i] = D[i] << 2;
        F[i] = E[i] + C[i];
        G[i] = F[i] – 1;
    }
```

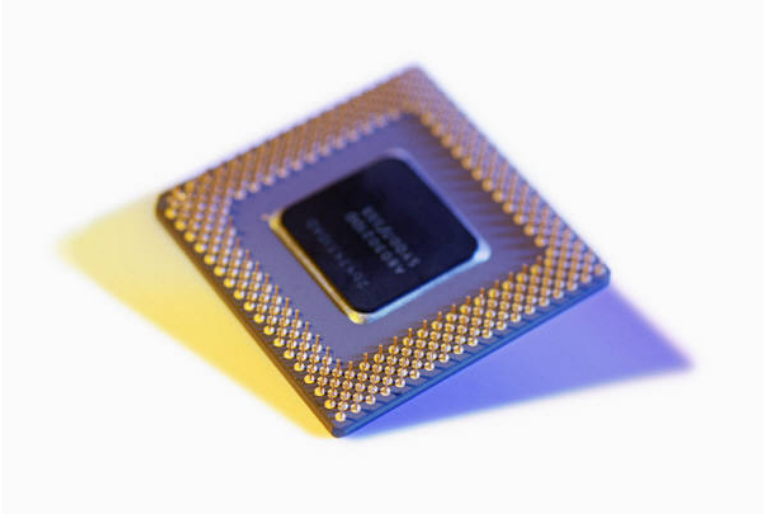


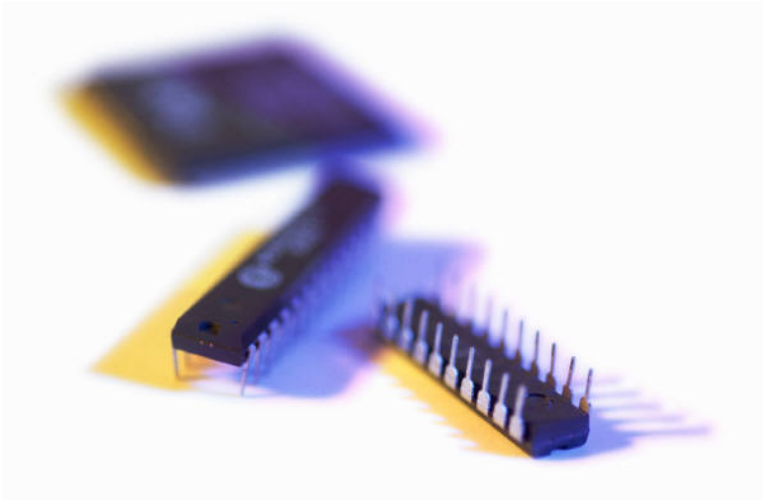
Sistema de Memória Compartilhada (1)

- Um conjunto de processadores autônomos conectados a um sistema de memória através de uma rede de interconexão.
- Cada processador pode acessar cada um dos locais da memória.
- Os processadores usualmente comunicam-se, implicitamente, através do acesso de estruturas de dados compartilhadas.

Sistema de Memória Compartilhada (2)

- Os sistemas de memória compartilhada mais conhecidos possuem um ou mais processadores multicore.
  - (múltiplos núcleos em um único chip)





Sistema de Memória Compartilhada

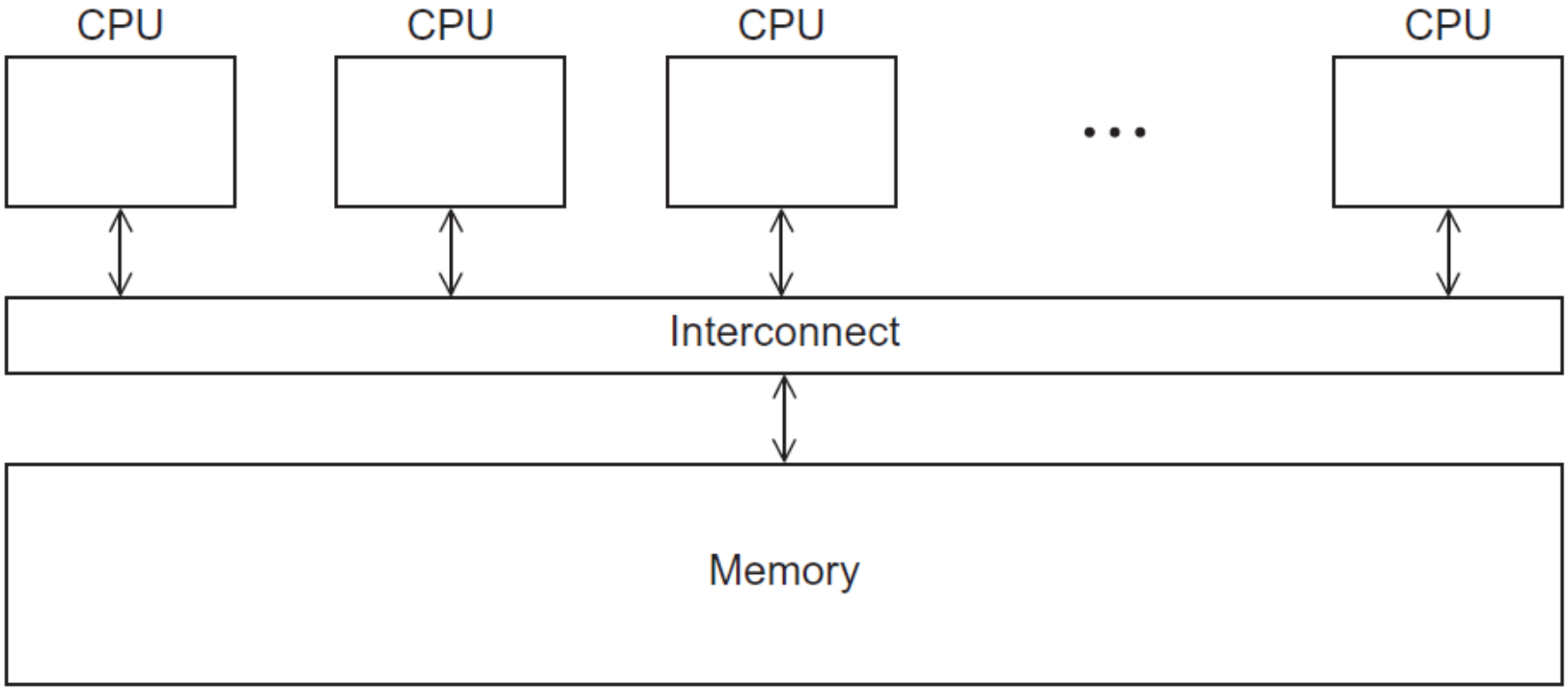
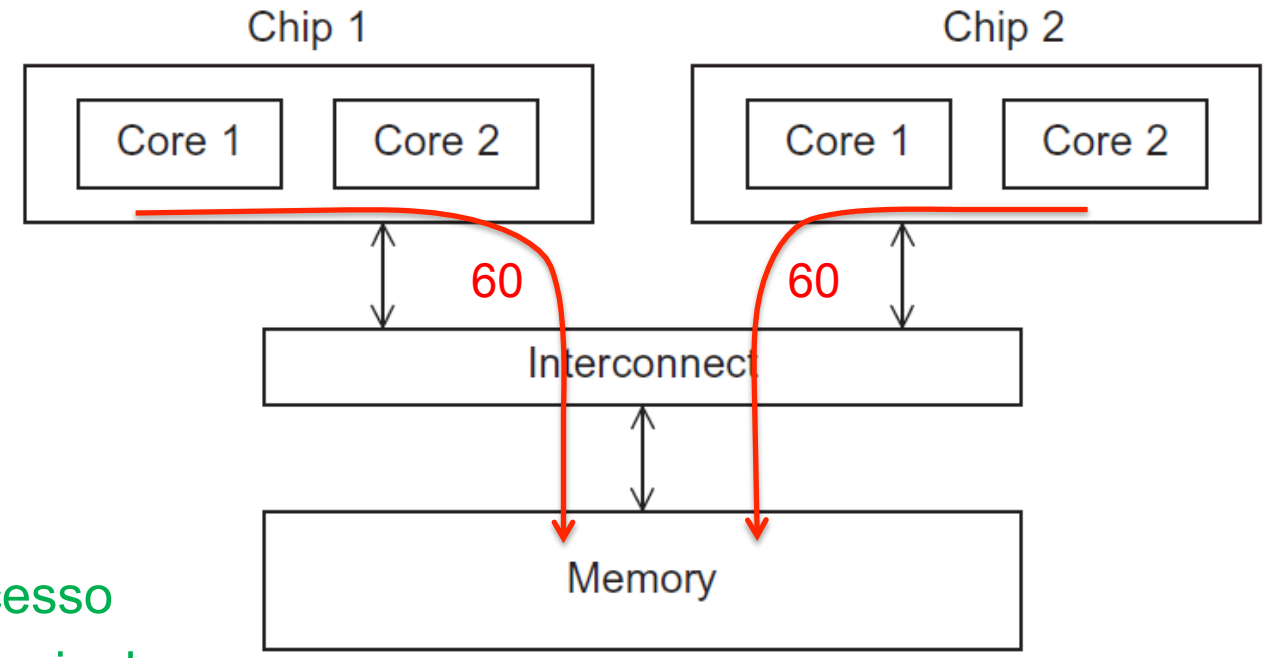


Figura 2.3

UMA multicore system



Tempo de acesso a todos os locais da memória é o mesmo para todos os núcleos.

Figura 2.5

NUMA multicore system

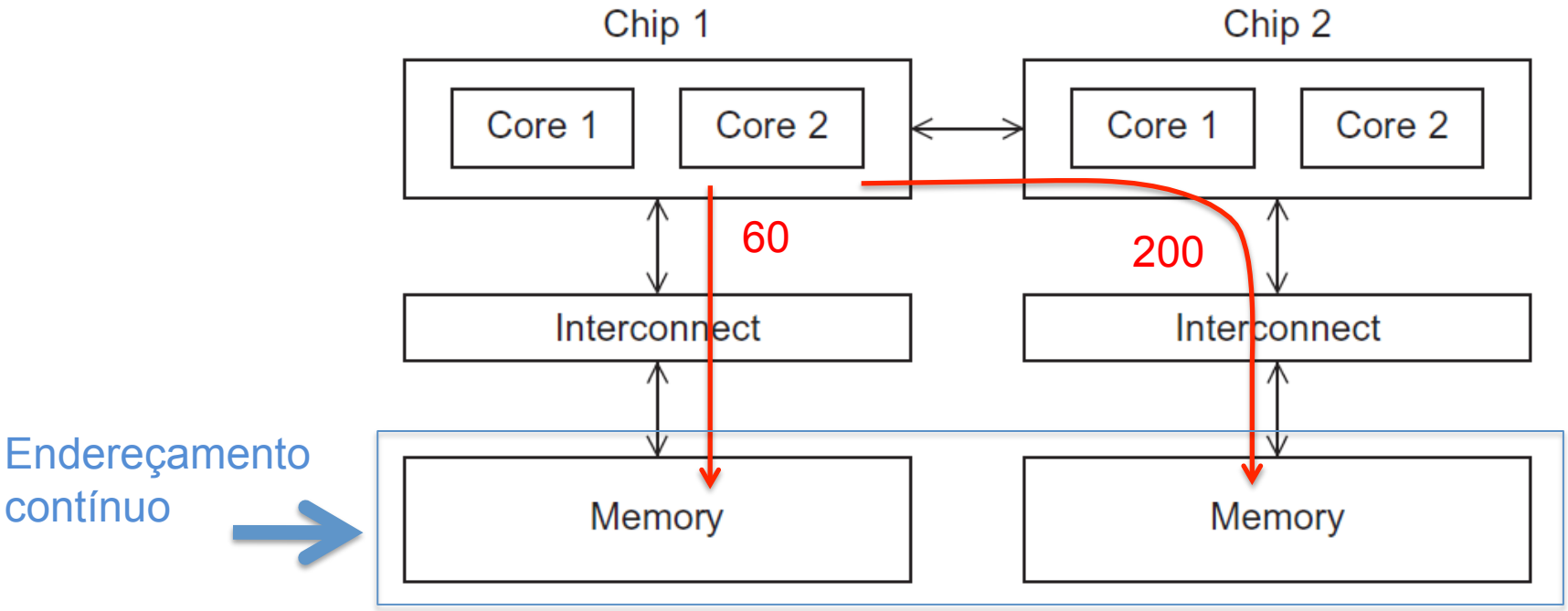


Figura 2.6

Locais da memória, aos quais o núcleo está diretamente conectado podem ser acessados mais rápidamente que locais da memória que são acessados de outro chip.

Sistema de Memória Distribuída

- Clusters (mais popular)
  - Uma coleção de sistemas de prateleira.
  - Conectado a uma rede de interconexão de prateleira.
- Nodes de um cluster são unidades de computação individuais unidas por uma rede de interconexão.

Sistema de Memória Distribuída

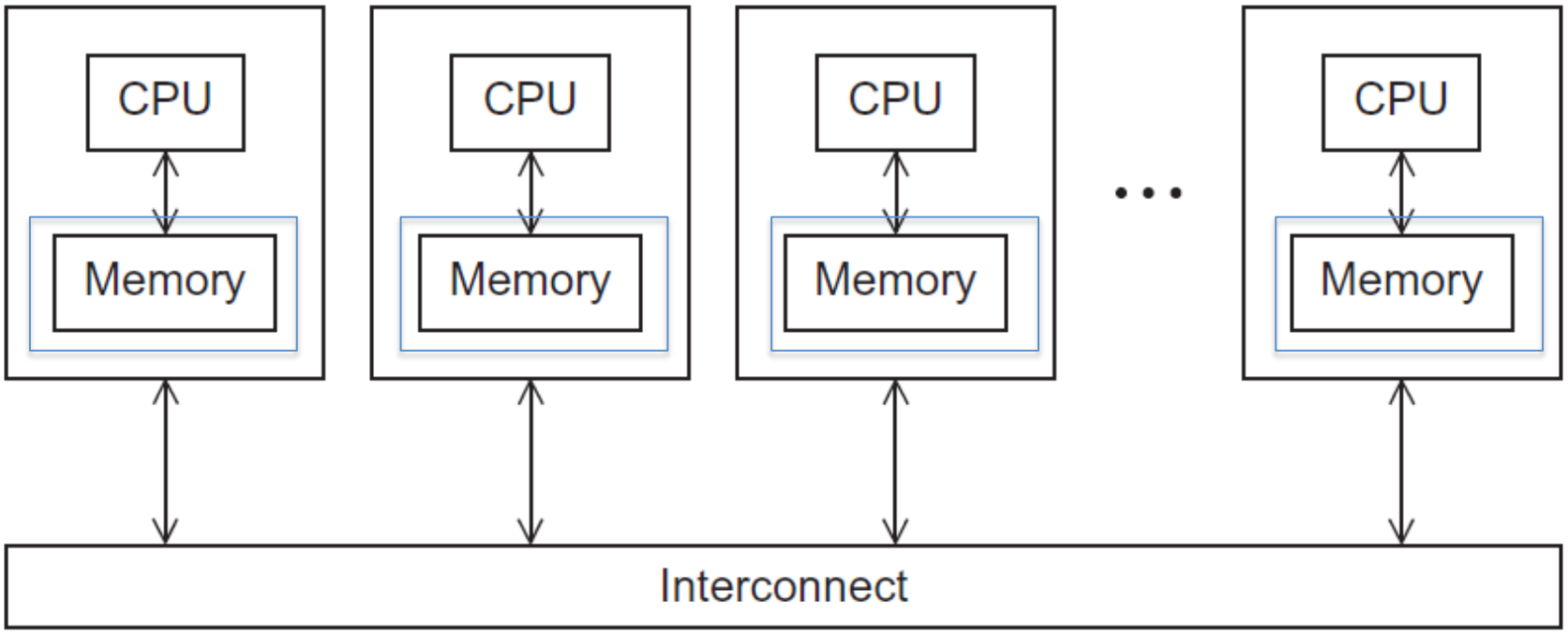


Figura 2.4

Endereçamento independente

Redes de Interconexão

- Afeta o desempenho tanto de sistemas de memória distribuída como compartilhada.

- Duas categorias:
  - Interconexão de memória compartilhada
  - Interconexão de memória distribuída

Interconexão de memória compatilhada

- Barramento
  - Uma coleção de fios de comunicação paralelos combinado com hardware que controla acesso.
  - Os fios de comunicação são compartilhados entre os dispositivos que estão conectados a eles.
  - Com o aumento do número de dispositivos conectados ao barramento, contenção pelo uso do barramento aumenta e o desempenho diminui.

Interconexões para memória compartilhada

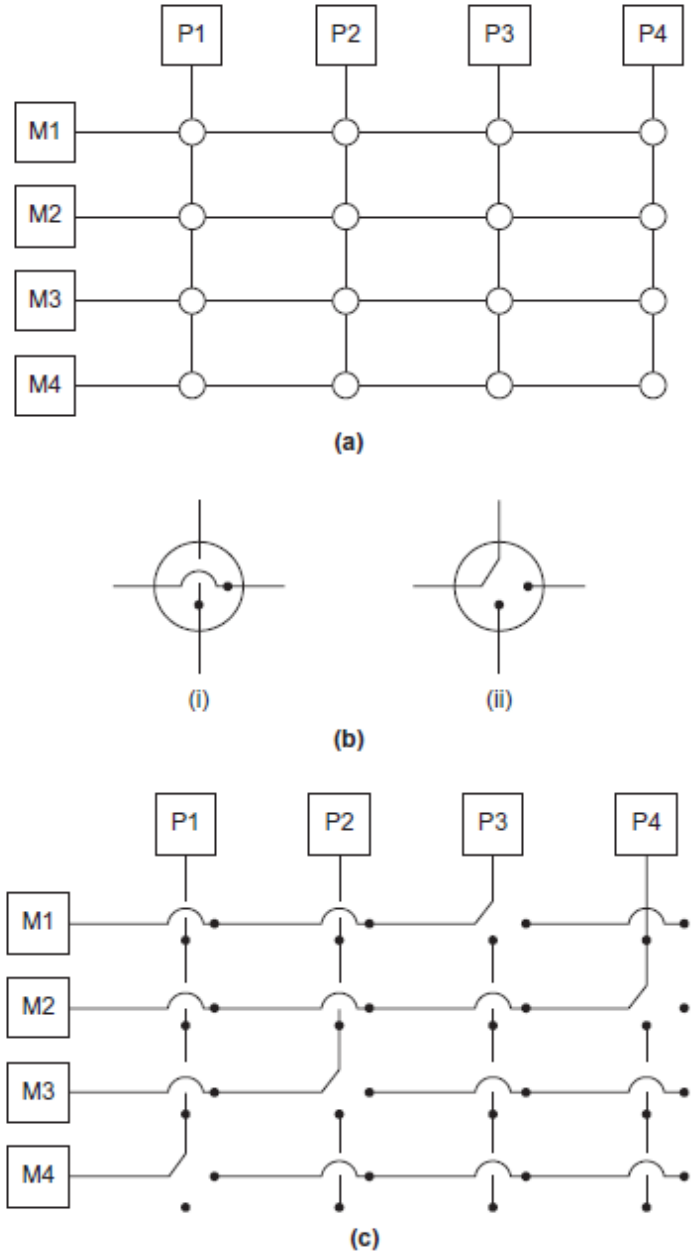
- Interconexão chaveada
  - Usa chaves para conectar o roteamento dos dados entre os dispositivos conectados.
  - Crossbar –
    - Permite a communicação simultânea entre dispositivos diferentes.
    - Mais rápido que barramentos.
    - Mas o custo das chaves e *links* é alto.

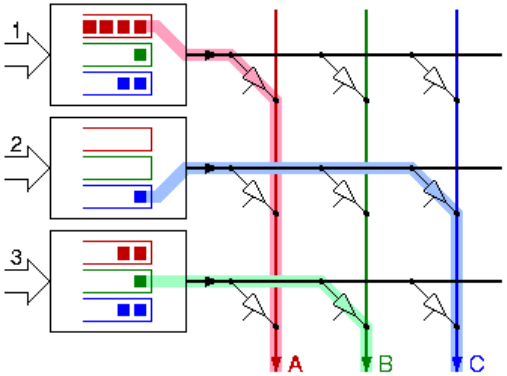
Figure 2.8

(a)
Uma interconexão em crossbar conecta 4 processadores (Pi) à 4 memórias (M_j)

(b)
Configuração das chaves internas de uma crossbar

(c) Acesso simultaneos à memória pelos processadores



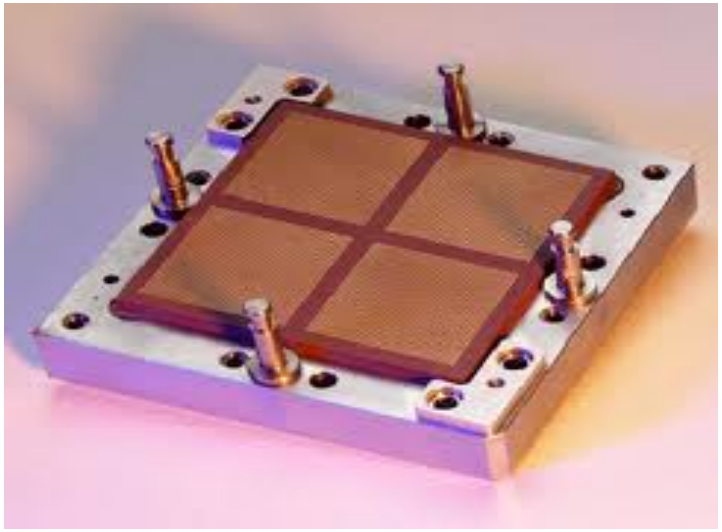


Crossbar

Convex Exemplar

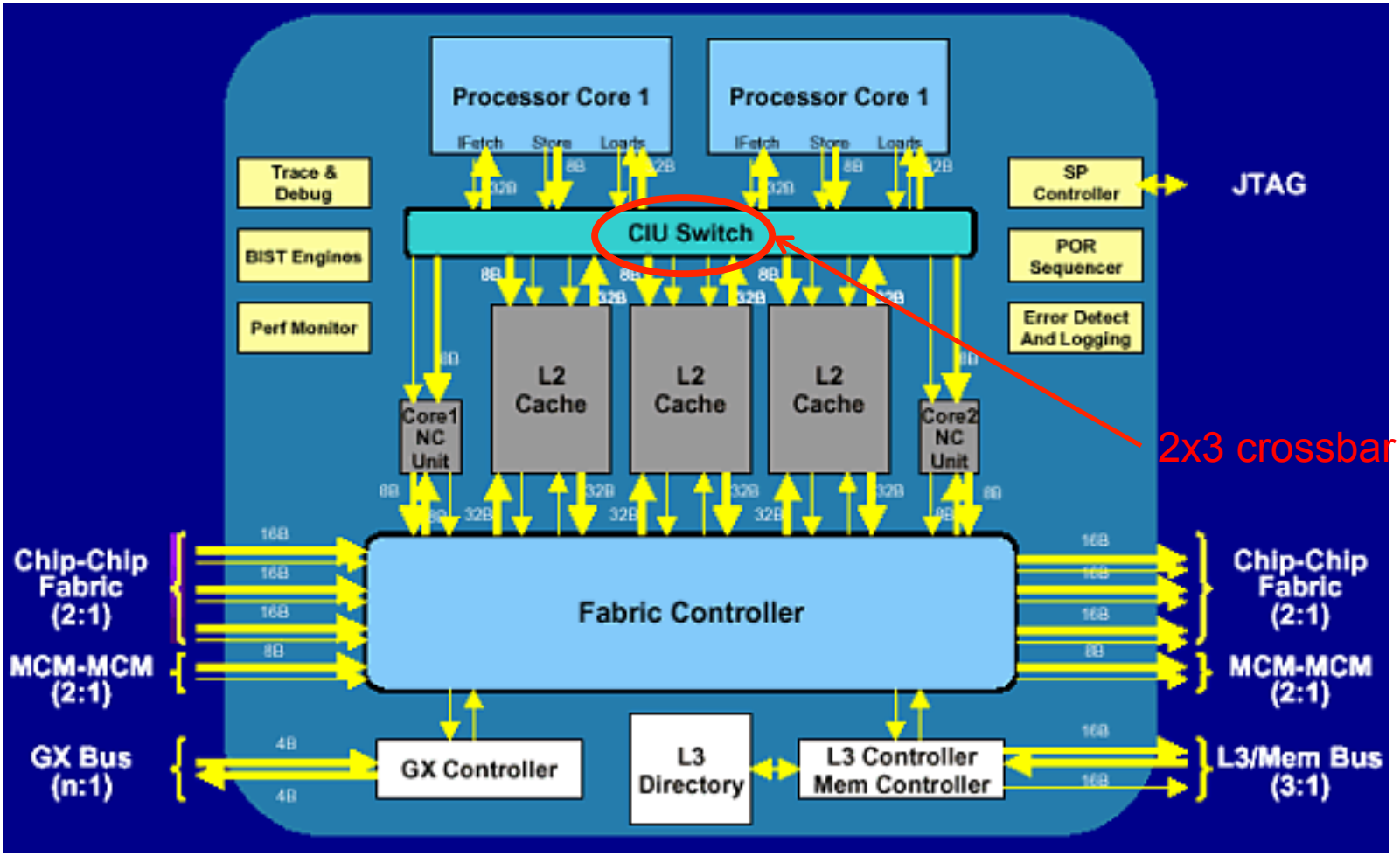
(1994 – later HP)

IBM POWER4



Baer p. 266

Crossbar no IBM POWER4

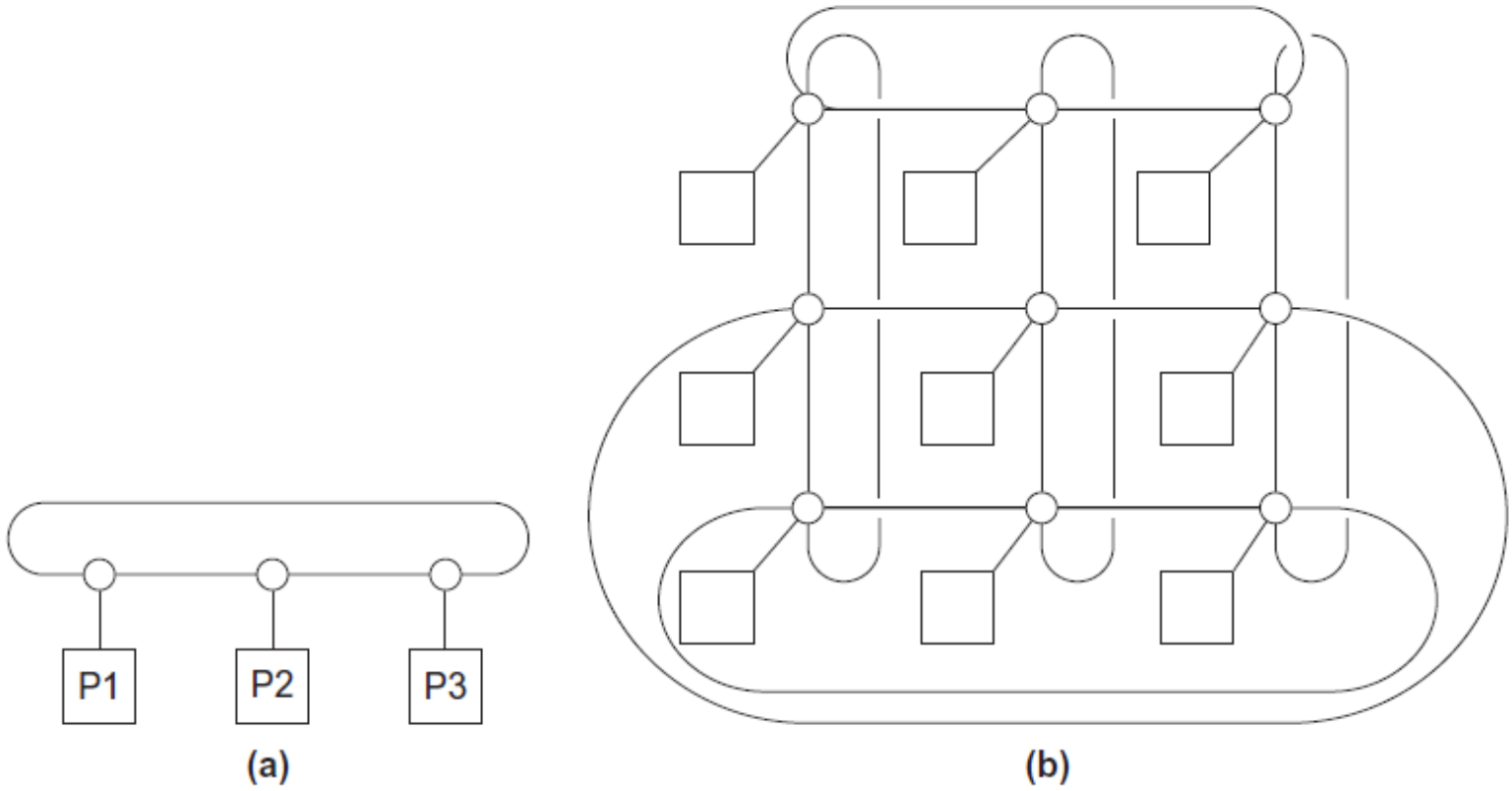


http://ixbtlabs.com/articles/ibmpower4/

Interconexões de Memória Distribuída

- Dois grupos
  - Interconexão direta
    - Cada chave é ligada diretamente a um par processador-memória e as chaves são conectadas entre si.
  - Interconexão indireta
    - As chaves podem não estar ligadas diretamente ao par processador-memória.

Interconexões diretas



anel

Malha toroidal

Figure 2.8

Largura de Biseção

- Uma medida do “número de comunicações simultâneas” ou “conectividade”.
- Quantas comunicações simultâneas podem ocorrer na “seção” entre duas metades?

Rede Completamente Conectada

- Cada chave é conectada diretamente a cada chave.

	Ring	Toroidal	Fully	Hypercube
Links			$\frac{p^2}{2} - \frac{p}{2}$	
Bisection			$\frac{p^2}{4}$	

Largura da biseção = $p^2/4$

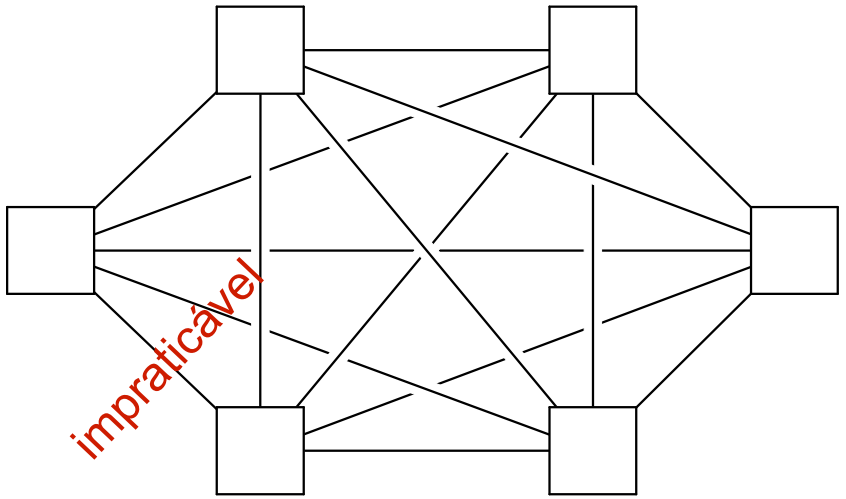


Figura 2.11

Duas biseções de um anel

	Ring	Toroidal	Fully	Hypercube
Links	p		$\frac{p^2}{2} - \frac{p}{2}$	
Bisection	2		$\frac{p^2}{4}$	

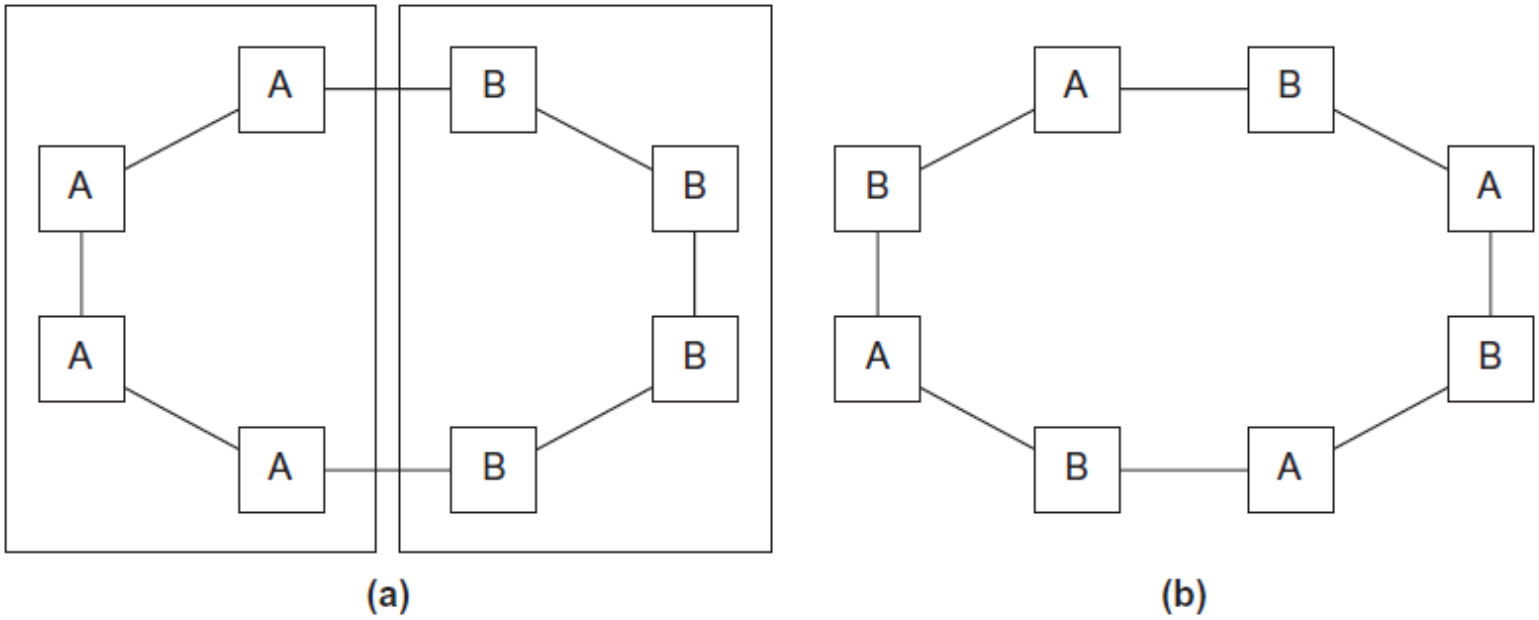


Figura 2.9

Biseção de uma malha toroidal

	Ring	Toroidal	Fully	Hypercube
Links	p	$2p$	$\frac{p^2}{2} - \frac{p}{2}$	
Bisection	2	$2\sqrt{p}$	$\frac{p^2}{4}$	

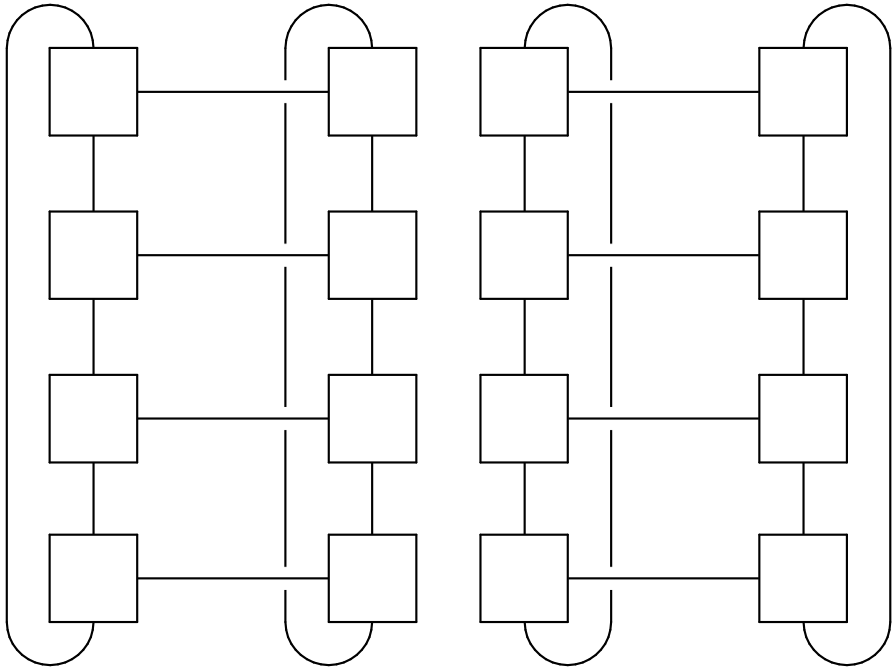
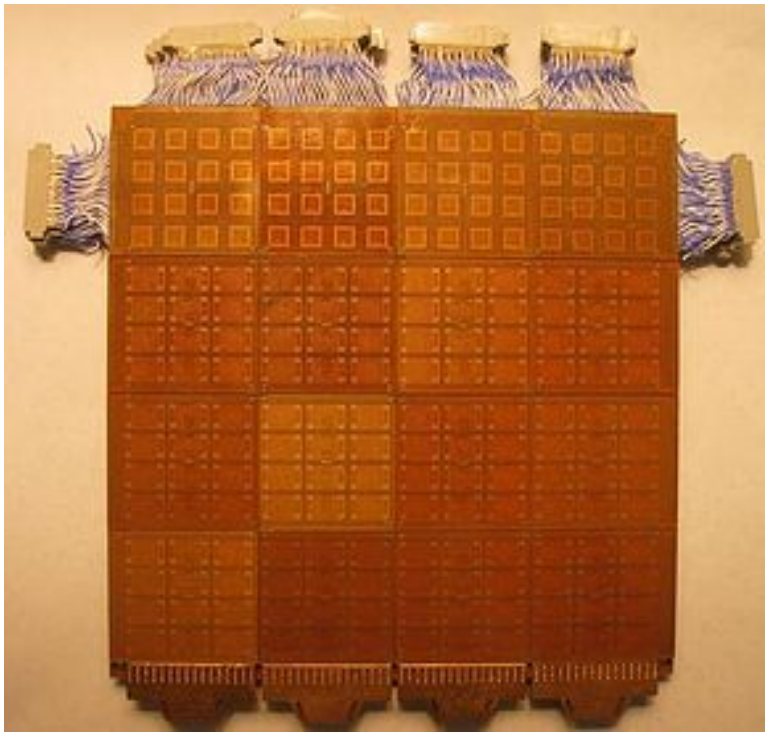


Figura 2.10

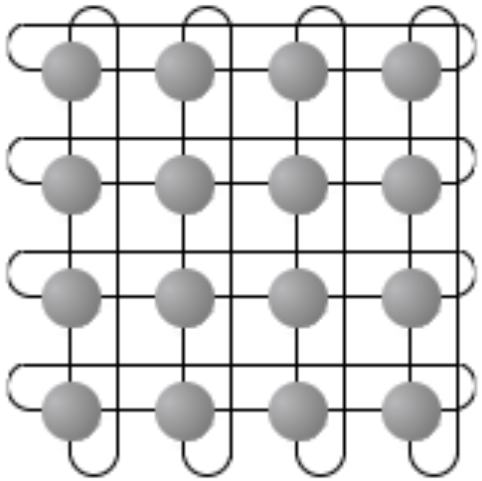


Cray 3 Module

2D Torus usado no Cray 3TD e

Cray 3TE

Cray 3 “brick”



2D Torus

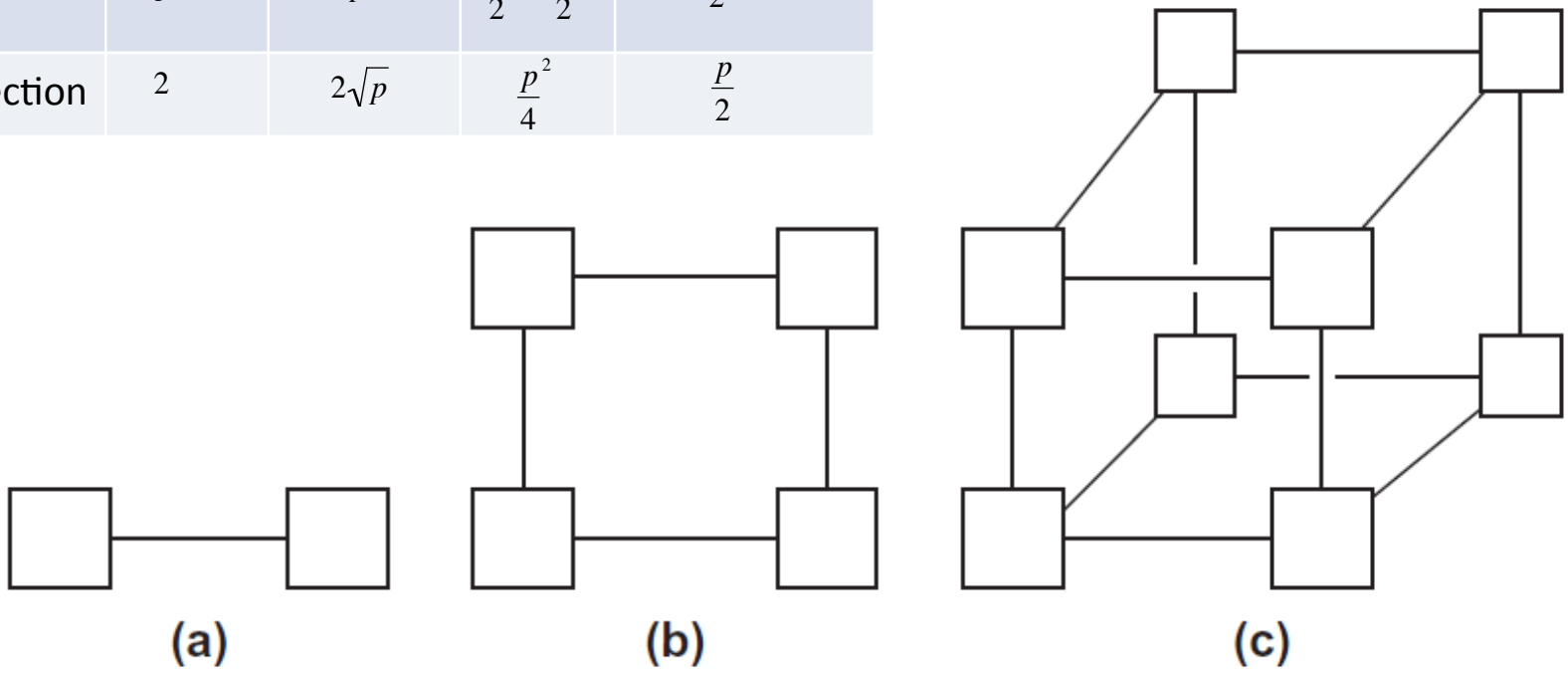
Baer p. 268

Hipercubo

- Interconexão direta altamente conectada.
- Construída usando indução:
  - Um hipercubo uni-dimensional é um sistema completamente conectado contendo dois processadores.
  - Um hipercubo bi-dimensional pode ser construído juntando-se as chaves "correspondentes" de dois hipercubos uni-dimensionais.
  - De maneira semelhante, um hipercubo tri-dimensional pode ser construído juntando-se as chaves "correspondentes" de dois hipercubos bi-dimensionais.

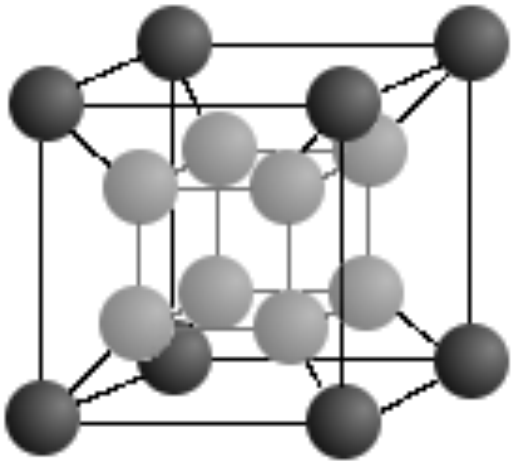
Hipercubos

	Ring	Toroidal	Fully	Hypercube
Links	p	$2p$	$\frac{p^2}{2} - \frac{p}{2}$	2^d
Bisection	2	$2\sqrt{p}$	$\frac{p^2}{4}$	$\frac{p}{2}$

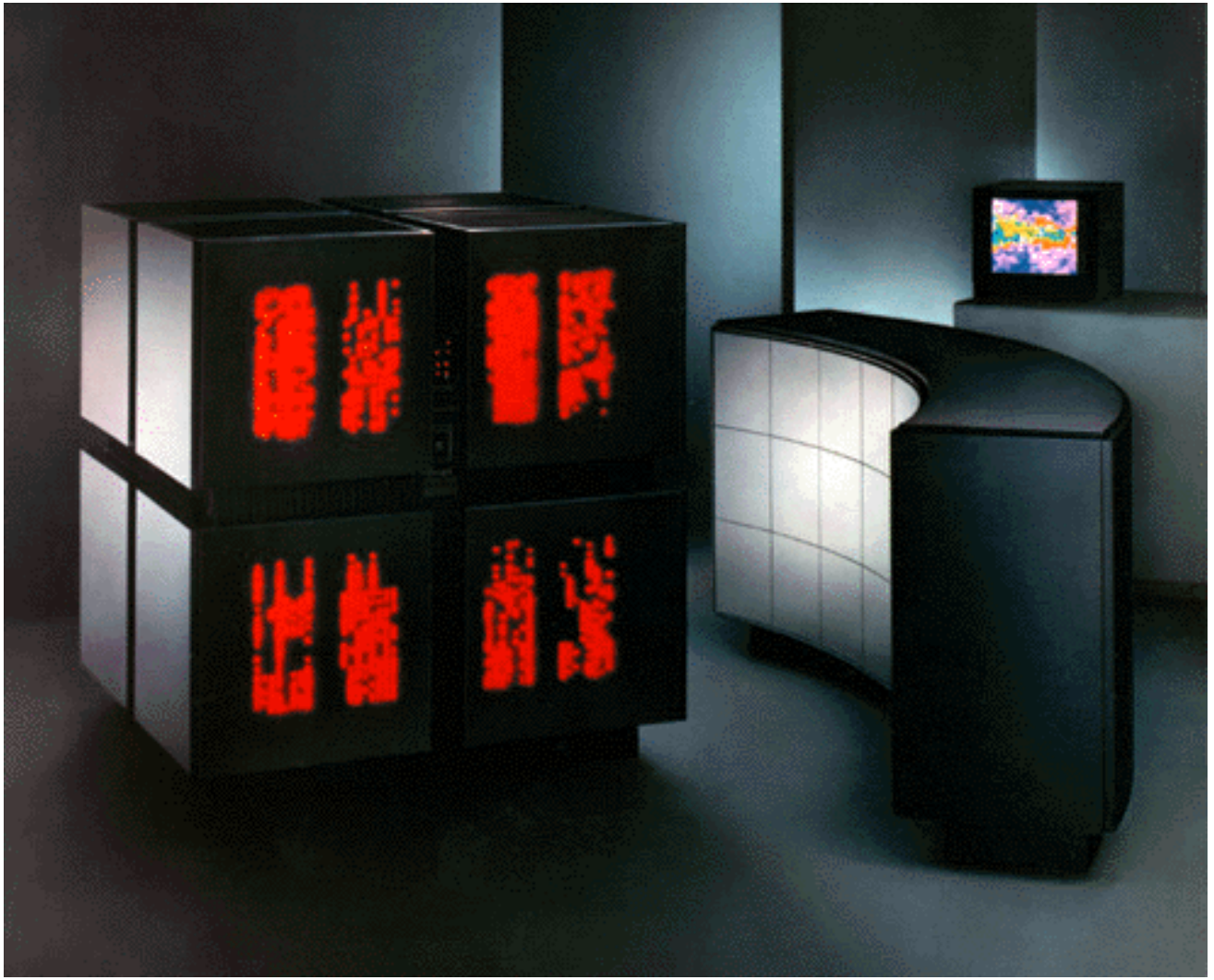


uni- bi- tri-dimensional

Figura 2.12



3D Hypercube



Connection Machine CM-2 usou um 12-D hypercube

Baer p. 268

Definições

- Largura de banda
  - Taxa na qual um link pode transmitir dados.
  - Dada usualmente em megabits ou megabytes por segundo.
- Largura de banda da biseção
  - Medida da qualidade da rede.
  - Ao invés de contar o número de links unindo as metades, soma a largura da banda dos links.

Interconexões indiretas

- Exemplos simples de redes indiretas:
  - Crossbar
  - Omega network
- Construídas normalmente usando:
  - Pares de processadores-memória, cada par com um link de entrada e um de saída uni-direcional
  - Uma rede de chaves interconexctando os links.

Rede de interconexões genérica

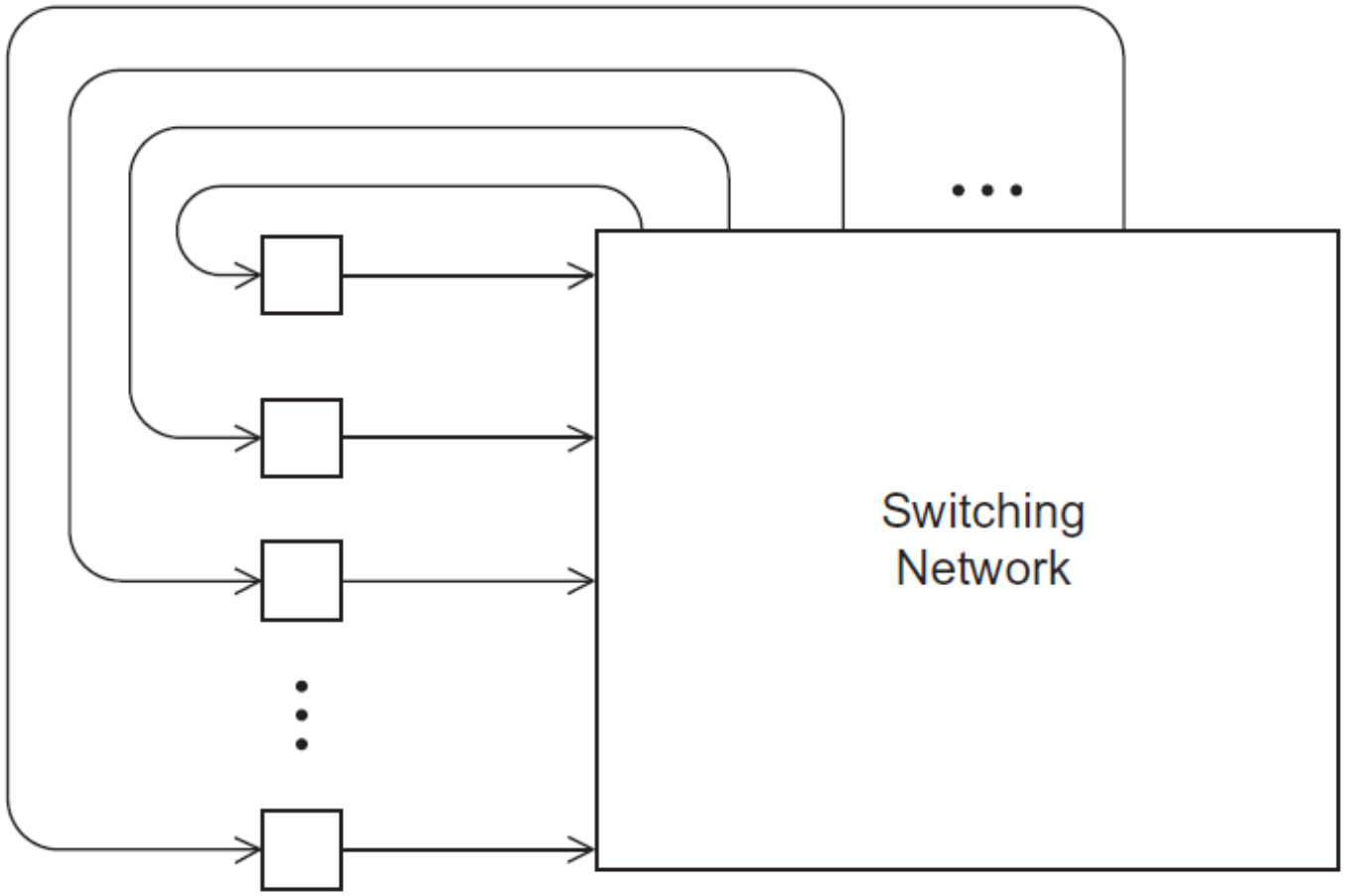


Figura 2.13

Crossbar para memória distribuída

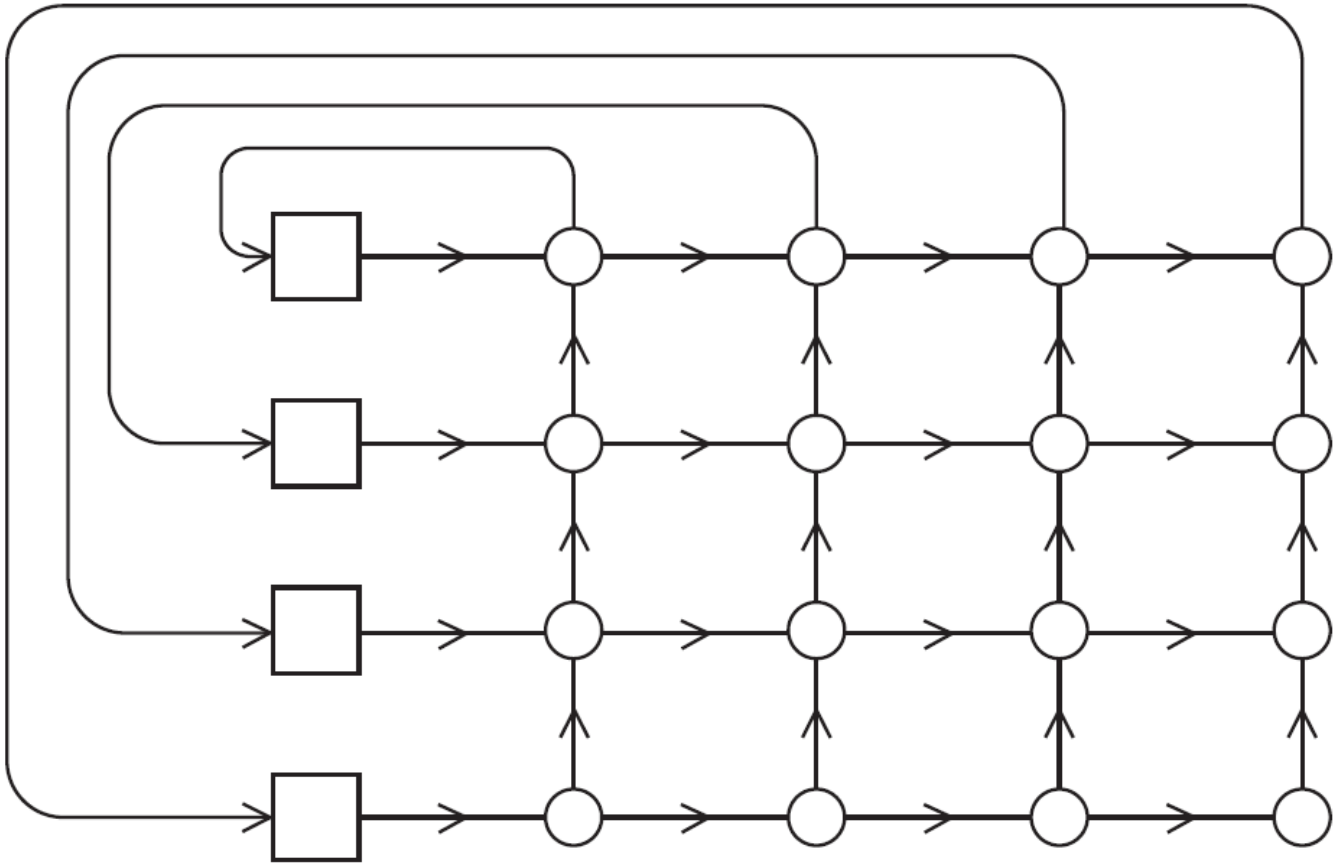


Figura 2.14

Qual a restrição de comunicação?

Uma rede omega

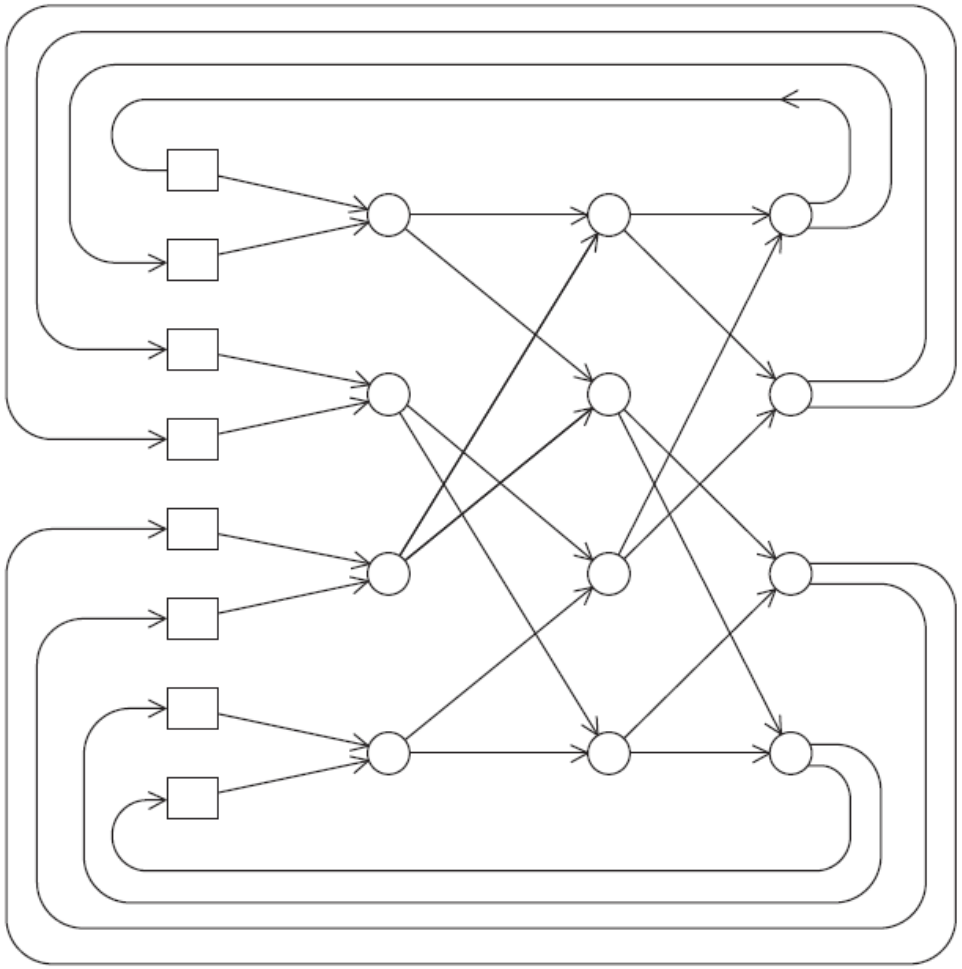


Figura 2.15

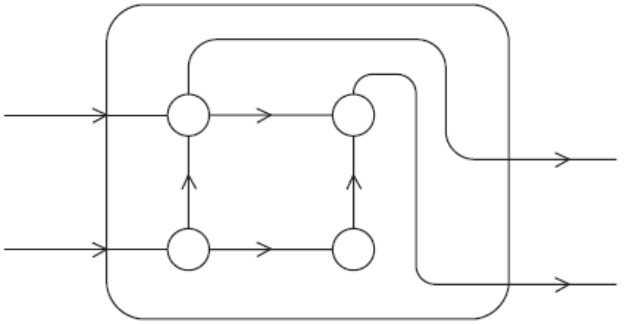
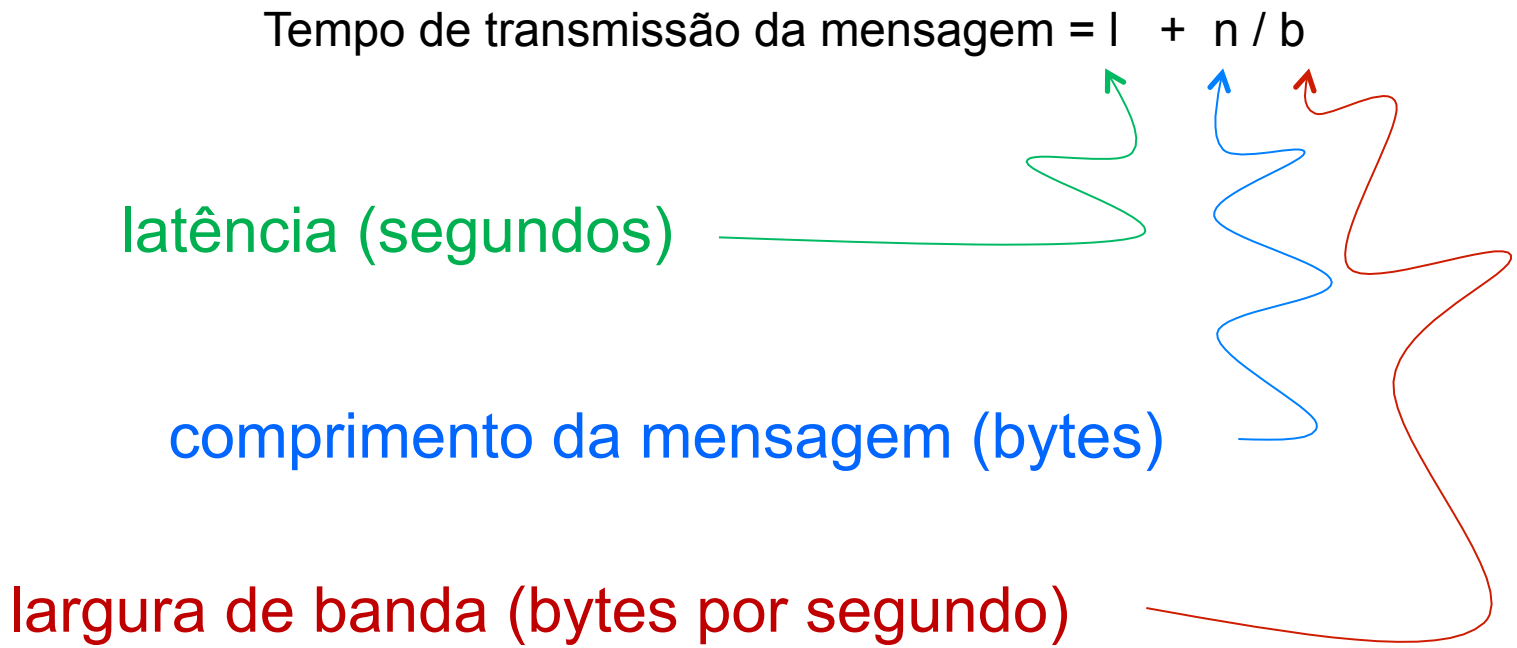


Figura 2.16

Qual a restrição de comunicação?

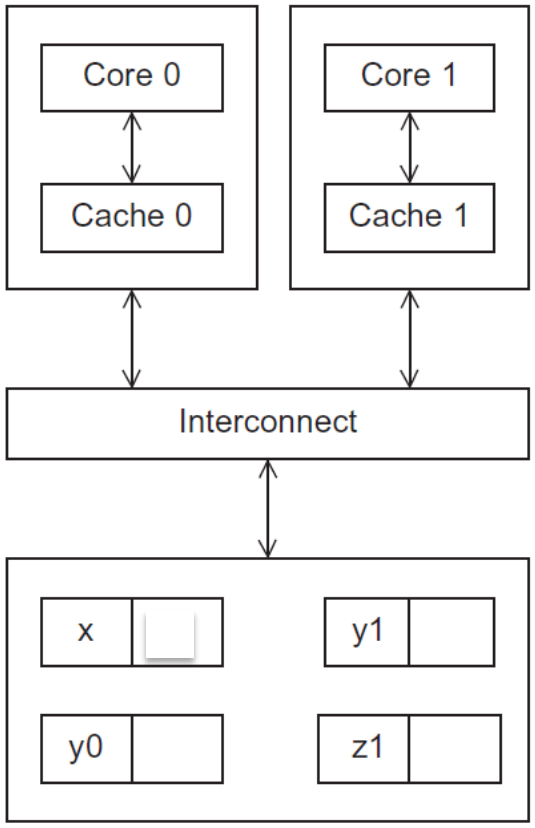
Mais definições

- Toda vez que um dado é transmitido, estamos interessados em saber quanto tempo levará para ele chegar ao seu destino.
- Latência
  - Tempo que passa entre o momento em que o dado começou a ser transmitido e o momento em que ele começou a ser recebido.
- Largura de banda
  - Taxa na qual o destino recebe o dado logo após ele começar a receber o primeiro byte.



Coerência de cache

- Programadores não tem qualquer controle sobre a cache ou quando ela é atualizada.



Um sistema de memória com dois núcleos e duas caches

Figura 2.17

Relação entre cache e memória

y0 propriedade privada do Core 0

y1 e z1 propriedade privada do Core 1

x = 2; /* variável compartilhada */

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

y0 termina sendo = 2

y1 termina sendo = 6

Qual o valor de z1?

O que ocorre quando

(a) cache write-through?

(b) cache write-back?

O que é preciso?

Manter as caches coerentes

Coerência de cache usando (*snooping*)

- Os núcleos partilham um barramento.
- Qualquer sinal transmitido no barramento pode ser “visto” por todos os núcleos a ele conectados.
- Quando o núcleo 0 atualiza a cópia de x que armazenada em sua cache ele transmite isto para todos os núcleos no barramento.
- Se o núcelo 1 está “snooping” o barramento, ele vai ver que x foi atualizada e pode marcar sua cópia de x como inválida.

Directory Based Cache Coherence

- Usa uma estrutura de dados chamada diretório que armazena o status de cada linha da cache.

- Quando uma variável é atualizada, o diretório é consultado, e os controladores de cache de cada núcleo que possuem a linha de cache daquela variável, invalidam as suas respectivas linhas.

Problemas com coerência

- Imagine agora que a e b estão na mesma cache line.
- O que ocorre com estas threads?

x	z		

T0:

```
for (i = 0; i < n; i++)
    if (i%2 + 1)
       x += 1;
```

T1:

```
for (i = 0; i < n; i++)
    if (i%2)
       z += 1;
```

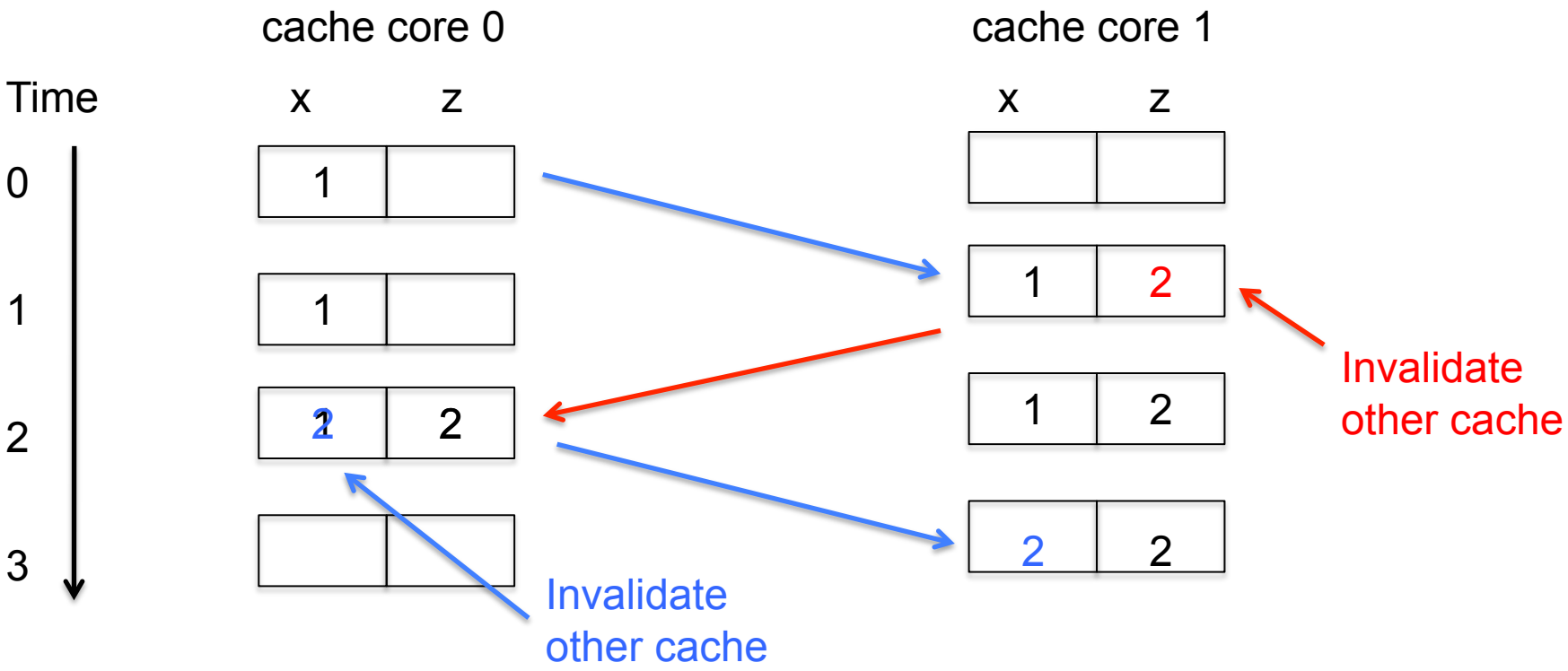
False sharing

T0:

```
for (i = 0; i < n; i++)
    if (i%2 == 0)
      x += 1;
```

T1:

```
for (i = 0; i < n; i++)
    if (i%2 != 0)
      z += 1;
```



SOFTWARE PARALELO

O trabalho está no software

- Hardware e compiladores podem acompanhar o ritmo.
- De agora em diante…
  - Em programas de memória compartillhada:
    - Inicie um único processo e dispare *threads*.
    - Threads executam as tarefas.
  - Em programas de memória distribuída:
    - Inicie múltiplos processos.
    - Processos executam as tarefas.

SPMD – single program multiple data

- Programas SPMD consistem de um único executável que pode se comportar como se fosse vários programas diferentes através do uso de desvio condicional.

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

Escrevendo programas paralelos

1. Divida o trabalho entre os processos/threads de modo que:
   (a) cada processo/thread receba +/- mesma carga
   (a) a comunicação seja minimizada.
2. Organize a sincronização de cada processo/thread
3. Organize a comunicação entre os processos/threads.

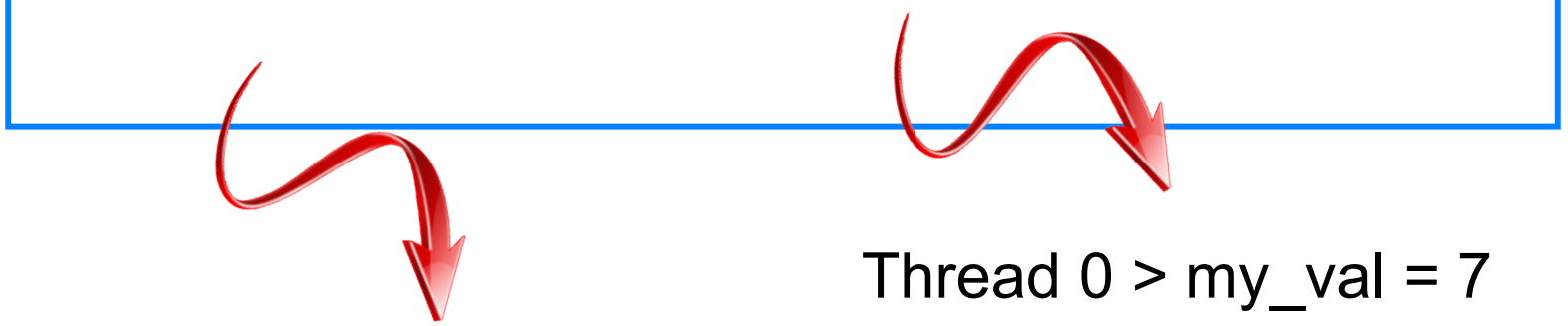
```
double x[n], y[n];
…
for (i = 0; i < n; i++)
    x[i] += y[i];
```

Memória Compartilhada

- Dynamic threads
  - A thread mestre espera pelo trabalho, dispara novas threads, e quando elas concluírem termina.
  - Uso eficiente dos recursos, mas a criação e o término da thread é custoso.
- Static threads
  - Um *pool* de threads é criado, o trabalho alocado, mas não termina até a limpeza final.
  - Produz um desempenho melhor mas pode desperdiçar recursos do sistema.

Não determinismo

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



Thread 0 > my_val = 7

Thread 1 > my_val = 19

Thread 1 > my_val = 19

Thread 0 > my_val = 7

Não determinismo

my_val = Compute_val (my_rank) ;

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`

Locks

- Condição de corrida
- Seção crítica
- Exclusão mútua
- Exclusão mútua usando lock (mutex, ou lock)

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

busy-waiting

```
my_val = Compute_val ( my_rank ) ;
if ( my_rank == 1)
    while ( ! ok_for_1 ) ;  /* Busy−wait loop */
x += my_val ;  /* Critical section */
i f ( my_rank == 0)
    ok_for_1 = true ;  /* Let thread 1 update x */
```

Passagem de mensagem

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

Partição do espaço de endereçamento

```
shared int n = . . . ;
shared double x [n] , y [n] ;
private int i , my_first_element , my_last_element ;
my_first_element = . . . ;
my_last_element = . . . ;
/ * Initialize x and y  */
. . .
for ( i = my_first_element ; i <= my_last_element ; i++)
    x [ i ] += y [ i ] ;
```

Input and Output

- Em programas de memória distribuída ou compartilhada, somente o processo 0 pode acessar *stdin*.

- Em programas de memória distribuída ou compartilhada todos os processos/threads podem acessar *stdout* and *stderr*.

Input and Output

- No entanto, dado o indeterminismo na ordem de saída para *stdout*, na maioria dos casos somente um único processo/thread é usado para acionar *stdout*, que não seja saída de depuração.

- Saídas de depuração devem sempre incluir o *rank* ou *id* do processo/thread que está gerando a saída.

Input and Output

- Somente um único processs/thread irá tentar acessar qualquer arquivo outro que não seja *stdin*, *stdout*, or *stderr*.

- Por exemplo, cada processo/thread pode abrir, de maneira privada, um arquivo para leitura/escrita, mas nunca dois processos podem abrir o mesmo arquivo.

PERFORMANCE

Speedup

- Número de núcleos = p
- Tempo de execução serial = T_{serial}
- Tempo de execução paralelo = $T_{parallel}$

$$T_{parallel} = T_{serial} / p$$

speedup linear

$$S = \frac{T_{serial}}{T_{parallel}}$$

Eficiência de um programa paralelo

$$E = \frac{S}{p} = \frac{\left(\frac{T_{serial}}{T_{parallel}}\right)}{p} = \frac{T_{serial}}{p \bullet T_{parallel}}$$

O que E está medindo?

Speedups e eficiências de um programa a paralelo

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

Com quantos cores temos o melhor speed-up?

Qual a solução mais eficiente considerando-se o custo?

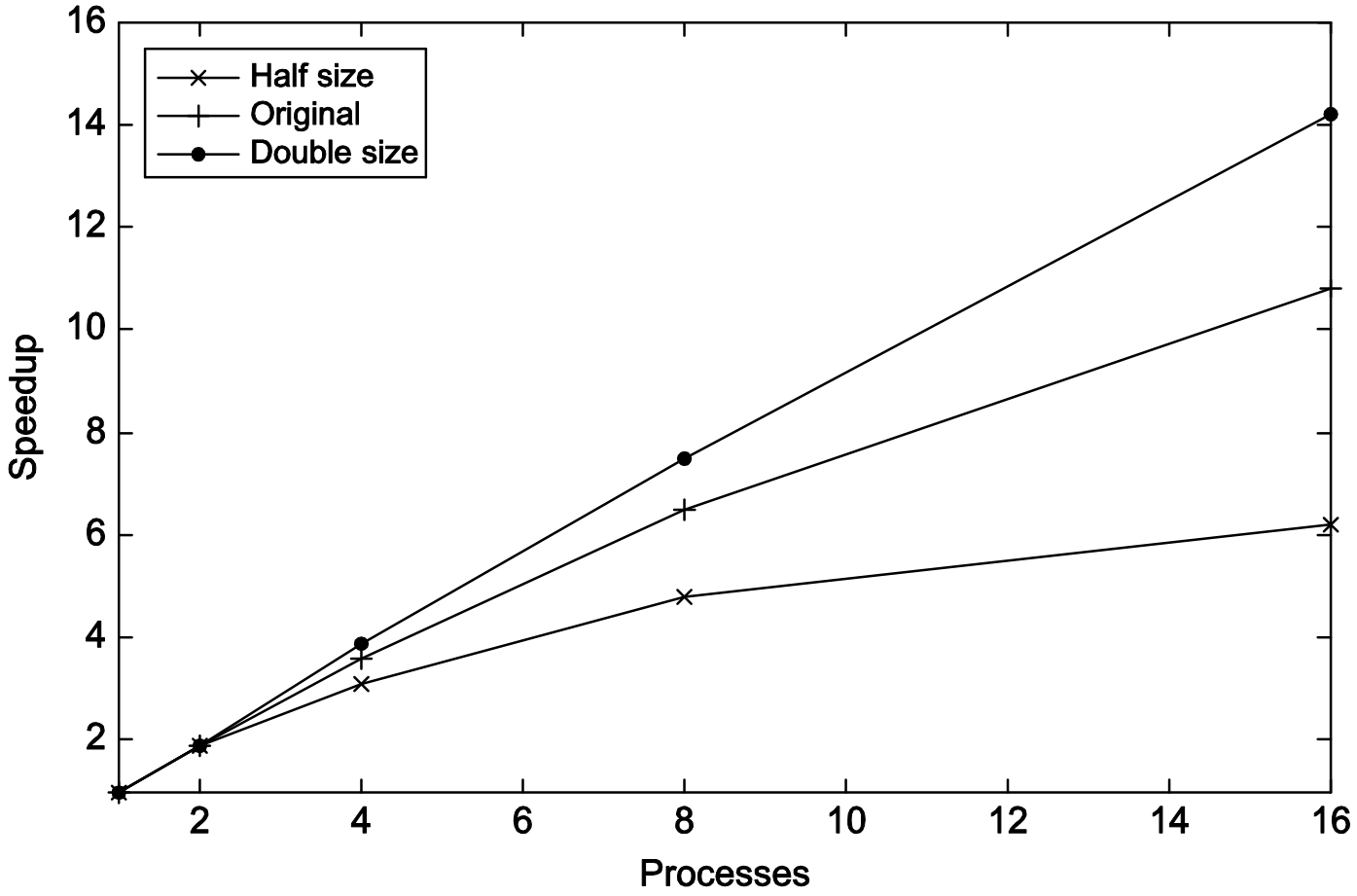
Variação de speedups e eficiências de um programa paralelo com o tamanho do programa

	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

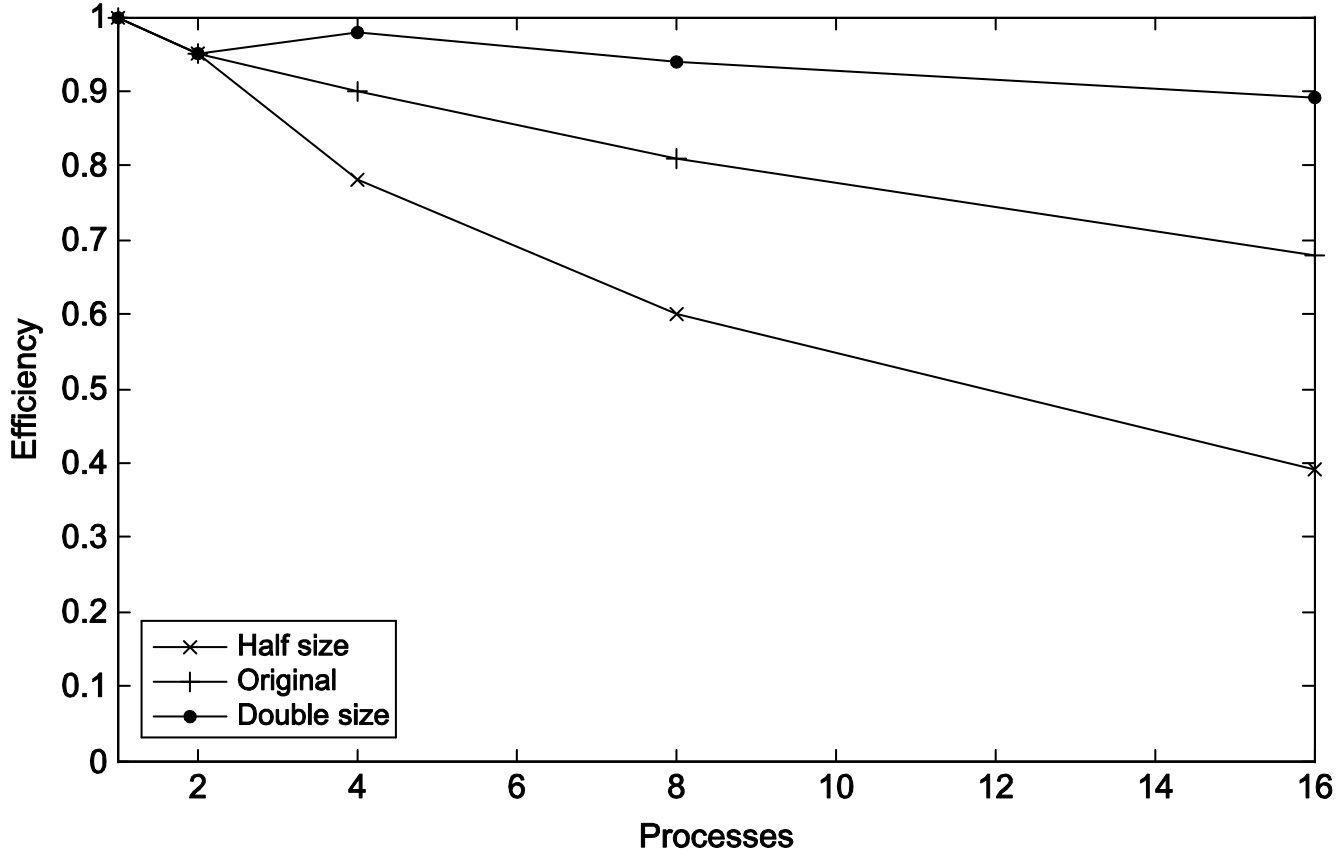
Aumentam!

Como variam S e E com o tamanho do problema?

Speedup



Efficiency



O Mundo dos Porques….

- Por que S e E aumentam com o tamanho?

  Tamanho do problema esconde o overhead!

$$T_{parallel} = T_{serial} / p + T_{overhead}$$

- Por que E vai diminuido e S vai saturando com aumento de p?

  Para o mesmo tamanho do problema, se p aumenta pode aumentar a comunicação e portanto o overhead

Lei de Amdahl

- Ao não ser que todo um programa serial possa ser paralelizado, o speedup possível será bem limitado – independente do número de núcleos disponíveis.

Exemplo

- Podemos paralelizar 90% de um programa serial.
- A paralelização é “perfeita” independente do número *p* de núcleos que usarmos.
- Assuma um programa em que $T_{serial} = 20$ seconds

Exemplo (cont.)

- Tempo de execução da parte não “paralelizável” é

$$0.1 \times T_{serial} = 2$$

- Tempo de execução da parte paralelizável é:

$$0.9 \times T_{serial} / p = 18 / p$$

- Tempo de execução total é:

$$T_{parallel} = 0.9 \times T_{serial} / p + 0.1 \times T_{serial} = 18 / p + 2$$

Exemplo (cont.)

- Speed up

$$S = \frac{T_{serial}}{0.9 \times T_{serial} / p + 0.1 \times T_{serial}} = \frac{20}{18 / p + 2}$$

$$S = \frac{1}{f / p + (1 - f)}, \text{ onde } f = \frac{T_{parallelizable}}{T_{serial}}$$

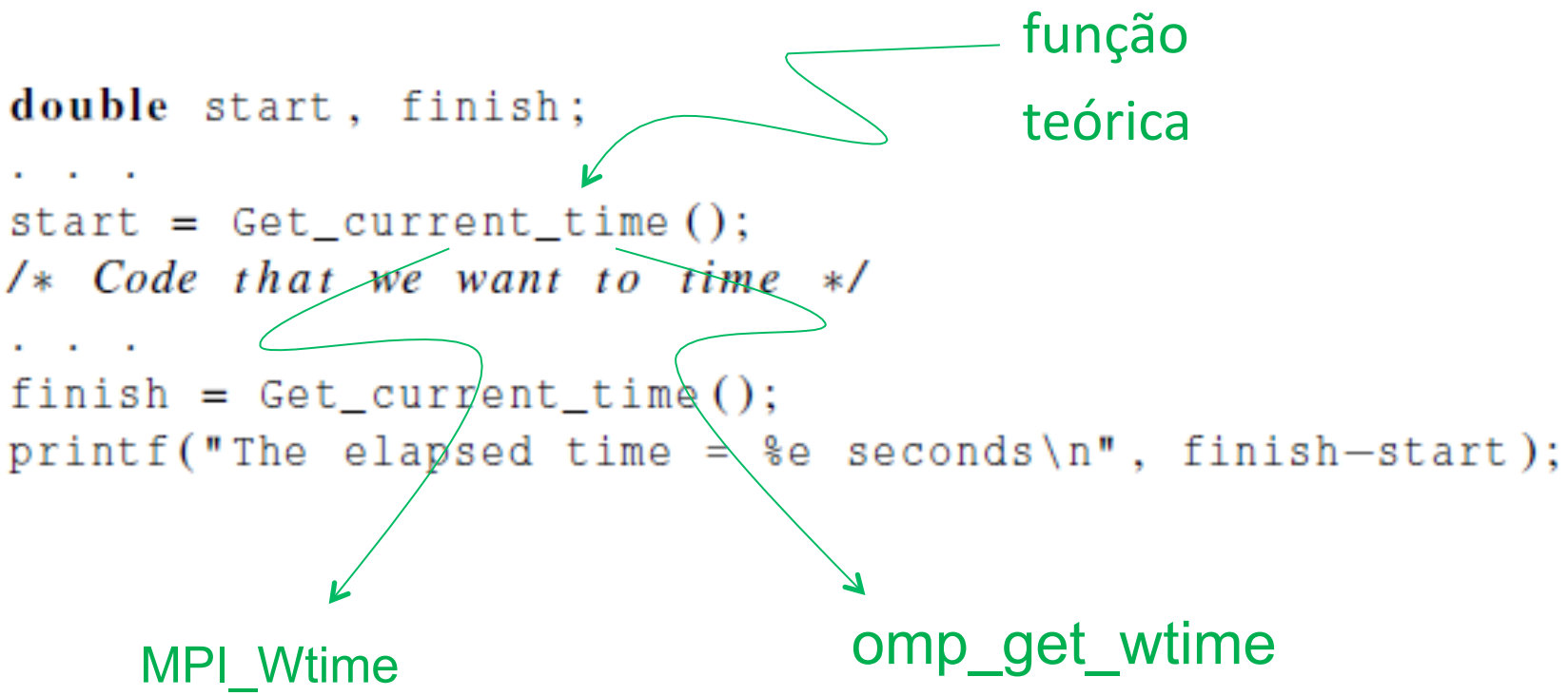
Escalabilidade

- No geral, um problema é *escalável* se ele pode lidar com tamanhos de problema sempre crescentes.
- Se o tamanho do problema é fixo e quando aumentamos o número de processos/threads a eficiência permanece a mesma, então dizemos que o problema é *fortemente escalável*.
- Se aumentando o tamanho do problema, para manter a eficiência fixa precisamos aumentar o número de processos/threads na mesma taxa, então problema é *fracamente escalável*.

Medindo tempos

- Qual é o tempo?
- Do começo até o final?
- Qual o trecho de interesse do programa?
- Usar tempo de CPU?
- Usar o tempo do relógio?

Medindo tempos



Taking Timings

```
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);
```

Medindo tempos

```
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);
```

PROJETO DE PROGRAMAS PARALELOS

Metodologia de Foster

1. Particionamento
2. Comunicação
3. Agregação
4. Mapeamento

Metodologia de Foster

1. Particionando: divida a computação a ser realizada e os dados a serem operados em tarefas menores.

   O foco deve ser em executar tarefas que podem ser executadas em paralelo.

Metodologia de Foster

2. Communicação: determinar que comunicação precisa ser executada entre as tarefas identificadas nos passos anteriores.

Metodologia de Foster

3. Aglomeração ou agregação: combine tarefas e comunicações identificadas no primeiro passo em tarefas maiores.

   Por exemplo, se a tarefa A deve ser executada antes que a tarefa B possa ser executada, faz sentido agregá-las em uma única tarefa composta.

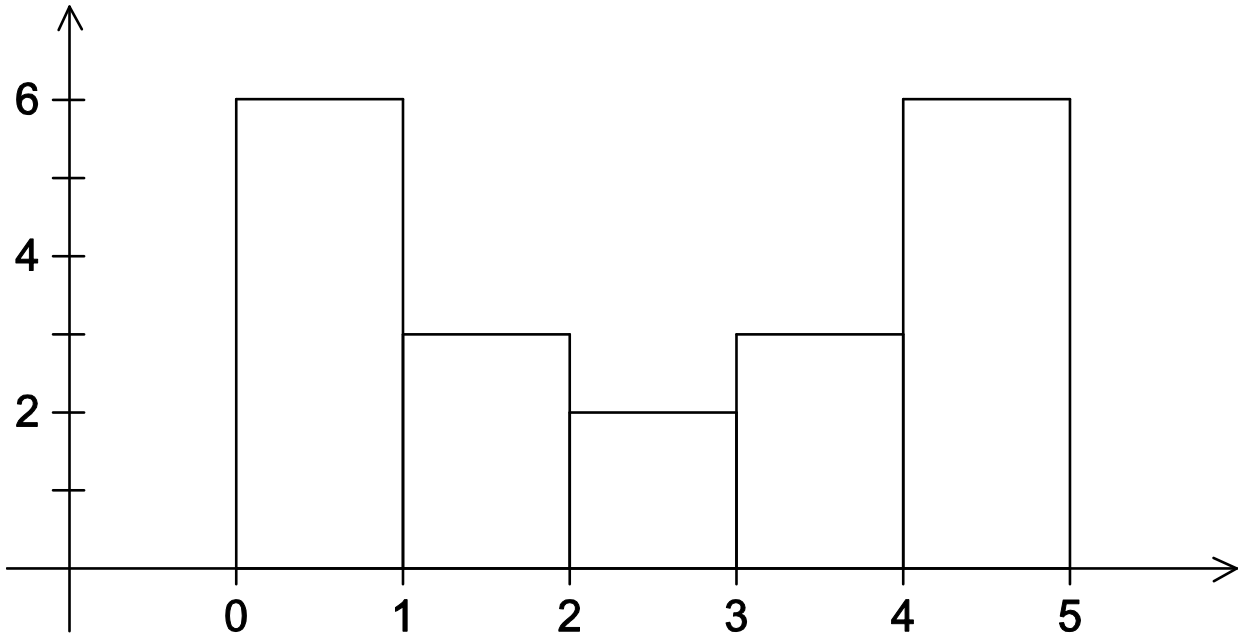
Metodologia de Foster

4. Mapeamento: atribua a tarefa composta identificada na etapa anterior aos processos/ threads.

   Isto deve ser feito de modo a minimizar a comunicação, e cada processo/thread recebe aproximadamente o mesmo volume de trabalho.

Exemplo - histograma

- 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



Serial program - input

1. The number of measurements: data_count
2. An array of data_count floats: data
3. The minimum value for the bin containing the smallest values: min_meas
4. The maximum value for the bin containing the largest values: max_meas
5. The number of bins: bin_count

Programa serial - output

1. bin_maxes : um array com bin_count floats

2. bin_counts : um array com bin_count ints

O que é preciso?

ATENÇÃO: Isto não é um programa!!

```
bin_width = (max_meas – min_meas)/bin_count

for (b = 0; b < bin_count; b++)
    bin_maxes[b] = min_meas + bin_width*(b+1);

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

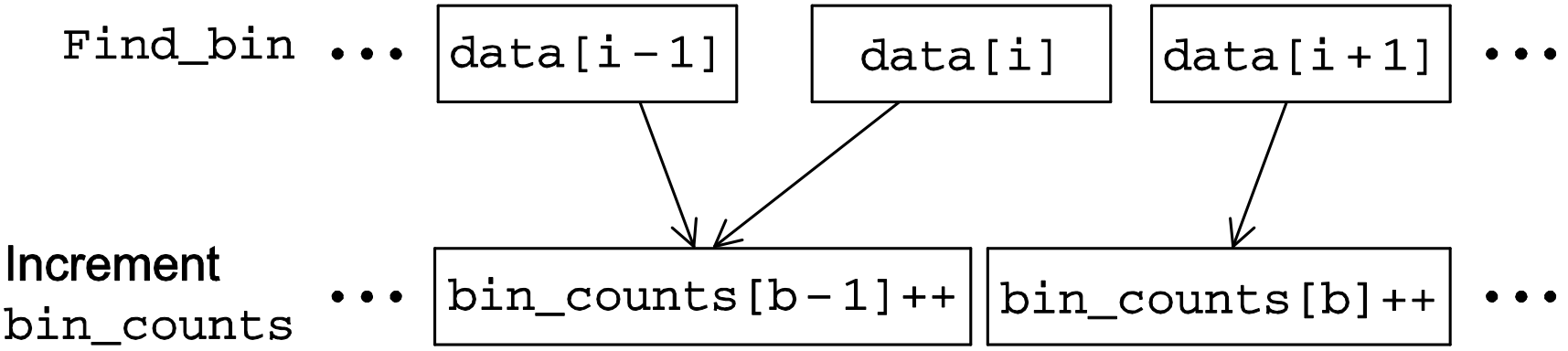
```
bin_maxes[b–1] <= data[i] < bin_maxes[b]

min_meas <= measurement < bin_maxes[0]
```

Primeiras duas etapas da Metodologia de Foster

Quais seriam estas?

1. Particionamento
2. Comunicação



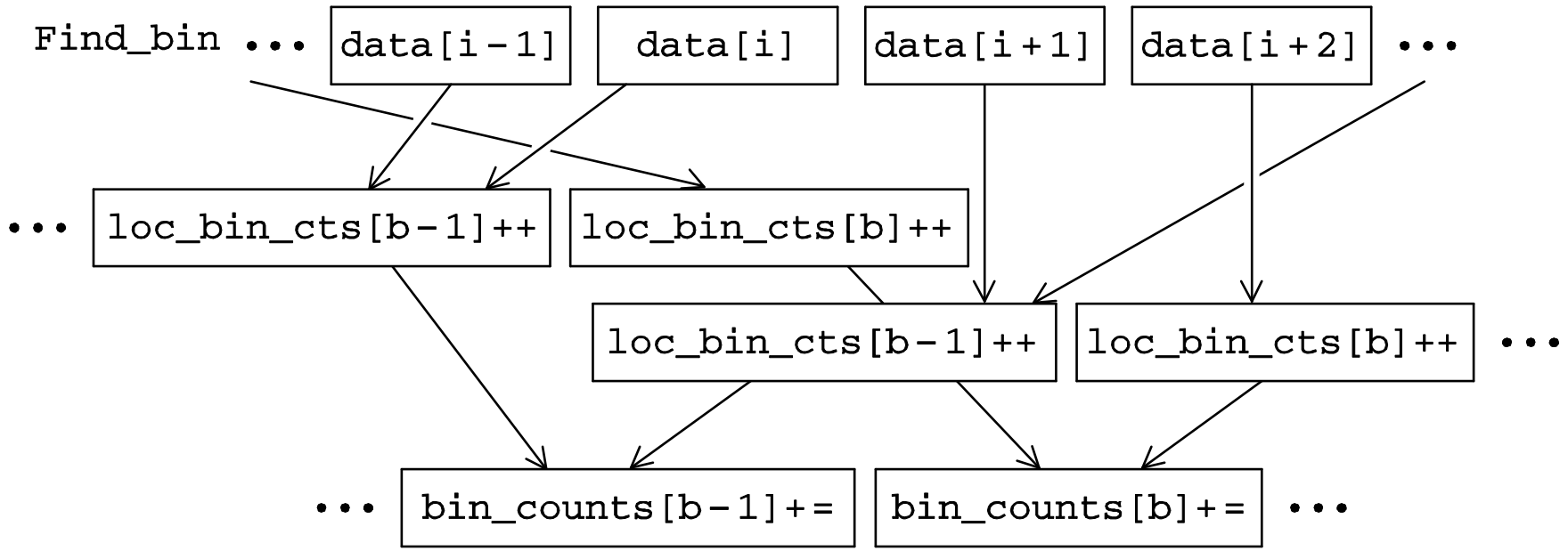
Qual o problema aqui?

Corrida daqueles que vão para o mesmo bin!!

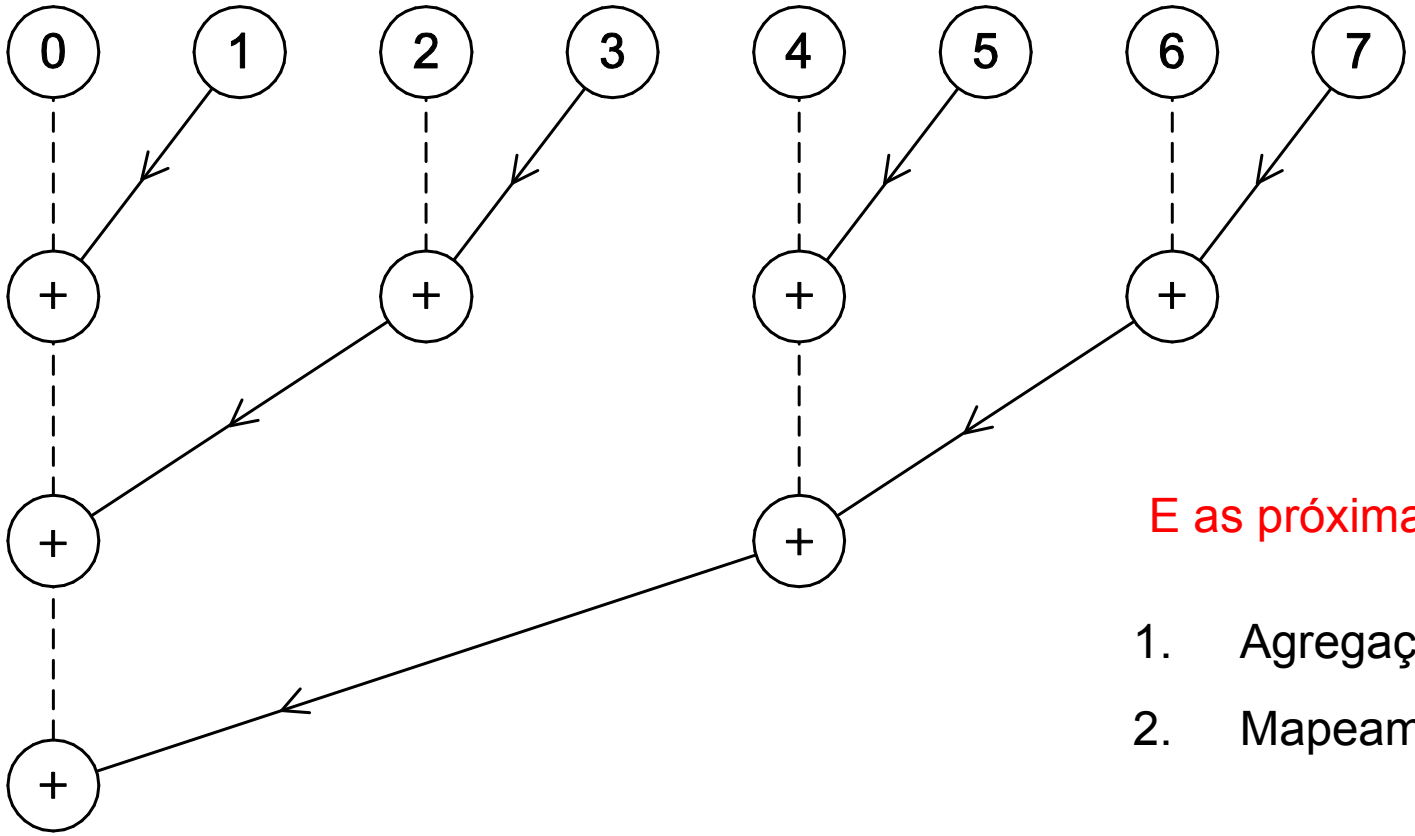
Definições alternativas de tarefas e comunicação

Qual a solução?

Criar vetores locais (privados) para os bins e somá-los no final



Somando arrays locais



E as próximas duas?

1. Agregação
2. Mapeamento

Conclusões (1)

- Sistemas seriais
  - O modelo padrão de arquitetura tem sido o de von Neumann.
- Hardware paralelo
  - Taxonomia de Flynn.
- Software paralelo
  - Nós focamos em software para sistemas MIMD, consistindo de um único programa seria que obtém paralelismo através de *branching*.
  - Programas SPMD.

Conclusões (2)

- Input and Output
  - Nós iremos escrever programas para os quais um processo ou um thread pode acessar stdin e todos os processos podem acessar stdout ou stderr.
  - No entanto, devido a não determinismo, exceto para a saída de depuração, usualmente teremos um único processo ou thread acessando stdout.

Conclusões (3)

- Performance
  - Speedup
  - Eficiência
  - Lei de Amdahl
  - Escalabilidade
- Projeto de programas paralelos
  - Metodologia de Foster