

MO401

Arquitetura de Computadores I

2006

Prof. Paulo Cesar Centoducatte

ducatte@ic.unicamp.br

www.ic.unicamp.br/~ducatte

MO401

Arquitetura de Computadores I

Multiprocessadores e Paralelismo em Nível de Thread

**"Computer Architecture: A Quantitative
Approach" - (Capítulo 6)**

Multiprocessadore e Thread

Sumário

- Introdução
- Níveis de Paralelismo
- Multiprocessadores?
- Classificação de Flynn
- Máquinas MIMD
 - UMA
 - NUMA
 - Multicomputadores
- Lei de Ahmdahl
- Métricas para Desempenho
- Arquiteturas Paralela
- Modelos: Shared Adress e Message Passing
- Coerência
 - Definição
 - Soluções

Computadores Paralelos

- Definição:

“A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast.”

Almasi and Gottlieb, *Highly Parallel Computing* ,1989

Computadores Paralelos: Introdução

- **Questões que devem ser respondidas:**
 - Qual o tamanho da coleção de elementos de processamento?
 - Os elementos de processamento são realmente úteis? (ou o quão útil é cada elemento de processamento?)
 - Como eles cooperam entre si e como comunicam-se?
 - Como os dados são transmitidos?
 - Qual o tipo de interconexão utilizada?
 - Quais são as primitivas de HW e SW visíveis ao programador?
- **Proporciona ganho de desempenho?**

Computadores Paralelos: Introdução

- Desde os anos 1950s:
replicar processadores para adicionar desempenho
vs. projetar processadores mais rápidos
- Devemos inovar na organização talhada para um particular modelo de programação já que mono-processamento não poderá avançar sempre
 - a velocidade de um processador não aumentará indefinidamente devido o seu limite ser a velocidade da luz: 1972, ... , 1989
 - (e agora que o projeto de processadores mais rápidos está encontrando uma barreira que não a velocidade da luz?)
 - em 1990s aparecem máquinas comerciais: Thinking Machines, Kendall Square, ...

Paralelismo, em que Nível?

- **Bit level parallelism: 1970 a ~1985**
 - Microprocessadores de 4 bits, 8 bits, 16 bits, 32 bits
- **Instruction level parallelism (ILP): ~1985 até os dias atuais**
 - Pipelining
 - Superscalar
 - VLIW
 - Out-of-Order execution
 - Limits dos benefícios de ILP?
- **Process Level ou Thread level parallelism (processamento de proposito geral)**
 - Servidores são paralelos
 - Desktop dual processor PC, são paralelos?

Por que Multiprocessadores?

1. Os Micro-processadores são as CPUs mais rápidas
 - Redesenhar uma CPU é mais difícil que coloca-las em uma coleção
2. Complexidade dos micro-processadores atuais
 - Novas idéias, em quantidade, para sustentar 1.5X/ano?
3. Aumento de desempenho baixo com softwares paralelos (aplicações científicas, bancos de dados, SO)
4. O mercado emergente de embedded e servidores tem levado a adição de microprocessadores nos desktops
 - Paralelismo de funções embedded (modelo produtor-consumidor)
 - O mérito dos Servidores tem se baseado em tarefas por horas (throughput) não por latência

Computadores Paralelos

Introdução

- Idéias mais antigas: escalar o número de processadores objetivando desempenho
- Máquinas "atuais": Sun Enterprise 10000 (2000)
 - 64 400 MHz UltraSPARC® II CPUs, 64 GB SDRAM, 868 18GB disk, tape
 - \$4,720,800 total
 - 64 CPUs 15%, 64 GB DRAM 11%, disks 55%, gabinete 16% (\$10,800 por processador ou ~0.2% por processador)
 - Mínimo E10K - 1 CPU, 1 GB DRAM, 0 disks, tape ~\$286,700
 - \$10,800 (4%) por CPU, mais \$39,600 board/4 CPUs (~8%/CPU)
- Máquinas "Atuais": Dell Workstation 220 (2001)
 - 866 MHz Intel Pentium® III (Minitower)
 - 0.125 GB RDRAM, 1 10GB disk, 12X CD, 17" monitor, nVIDIA GeForce 2 GTS, 32MB DDR Graphics card, 1yr service
 - \$1,600; para processador extra + \$350 (~20%)

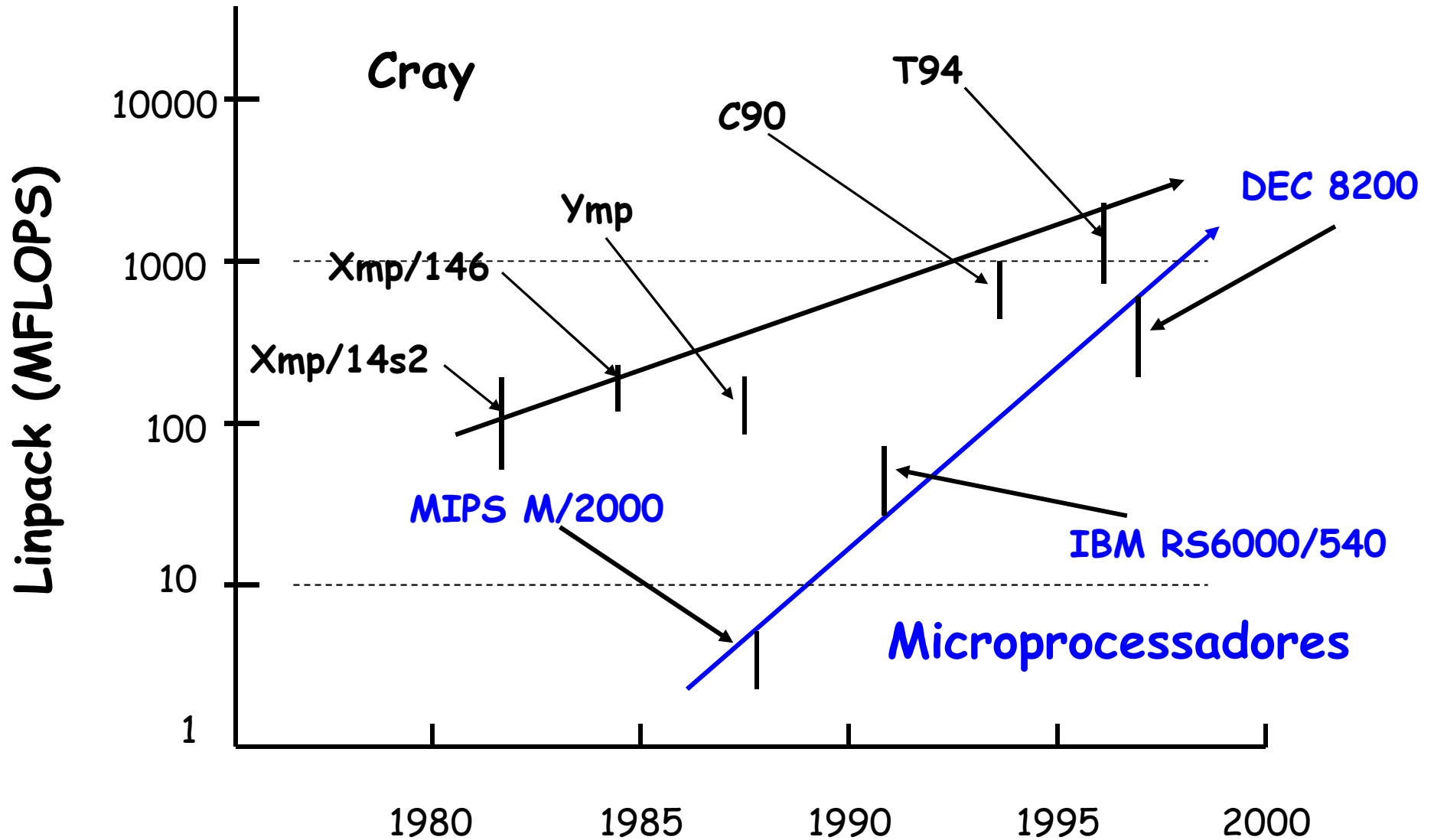
Para onde vão os Supercomputadores

- Linpack (algebra linear) para **Vector Supercomputers** vs. **Microprocessadores**
(próximo slide)
- 1997: 500 máquinas mais rápidas do mundo:
 - 319 Massively Parallel Processors (MPP), 73 bus-based shared memory (SMP), 106 parallel vector processors (PVP)
- 2000: 500 máquinas mais rápidas do mundo:
 - 381: 144 IBM SP (~cluster), 121 Sun (bus SMP), 62 SGI (NUMA SMP), 54 Cray (NUMA SMP)

Dados => **Parallel computer architecture : a hardware/ software approach**, David E. Culler, Jaswinder Pal Singh, with Anoop Gupta. San Francisco : Morgan Kaufmann, c1999.

LINPACK

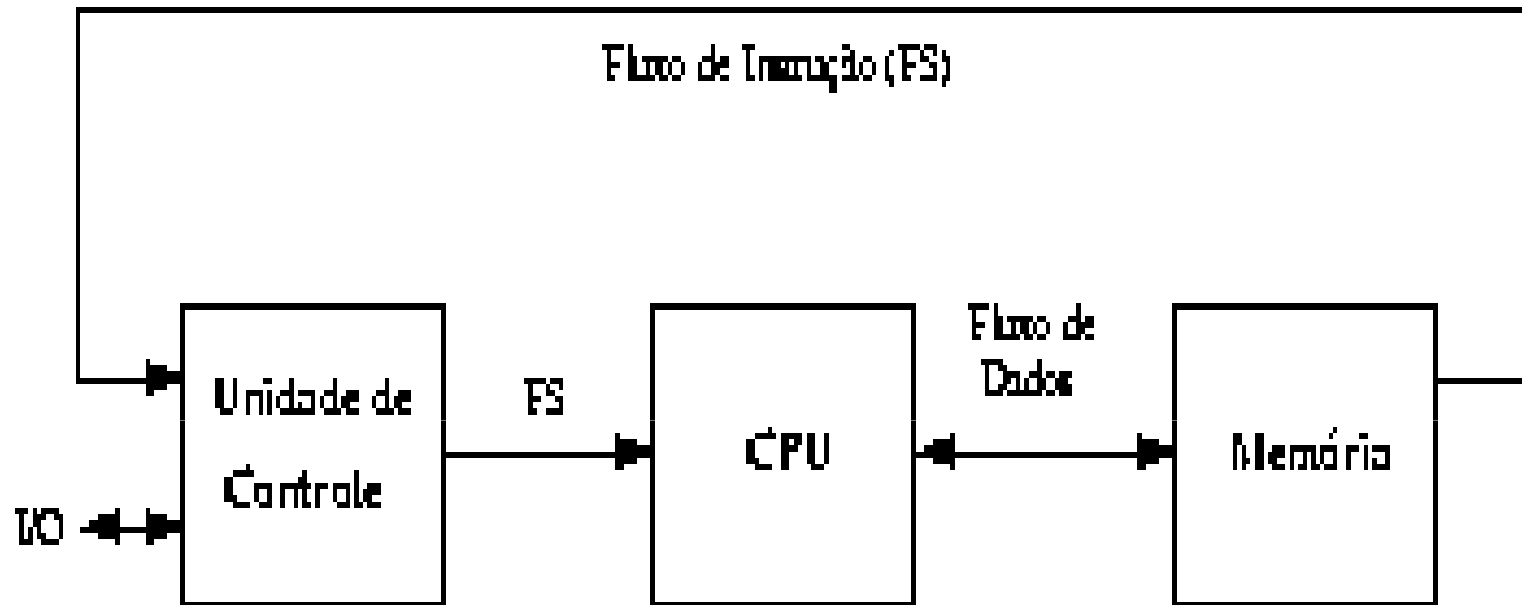
matrizes 100x100 e 1000x1000



Classificação de Flynn

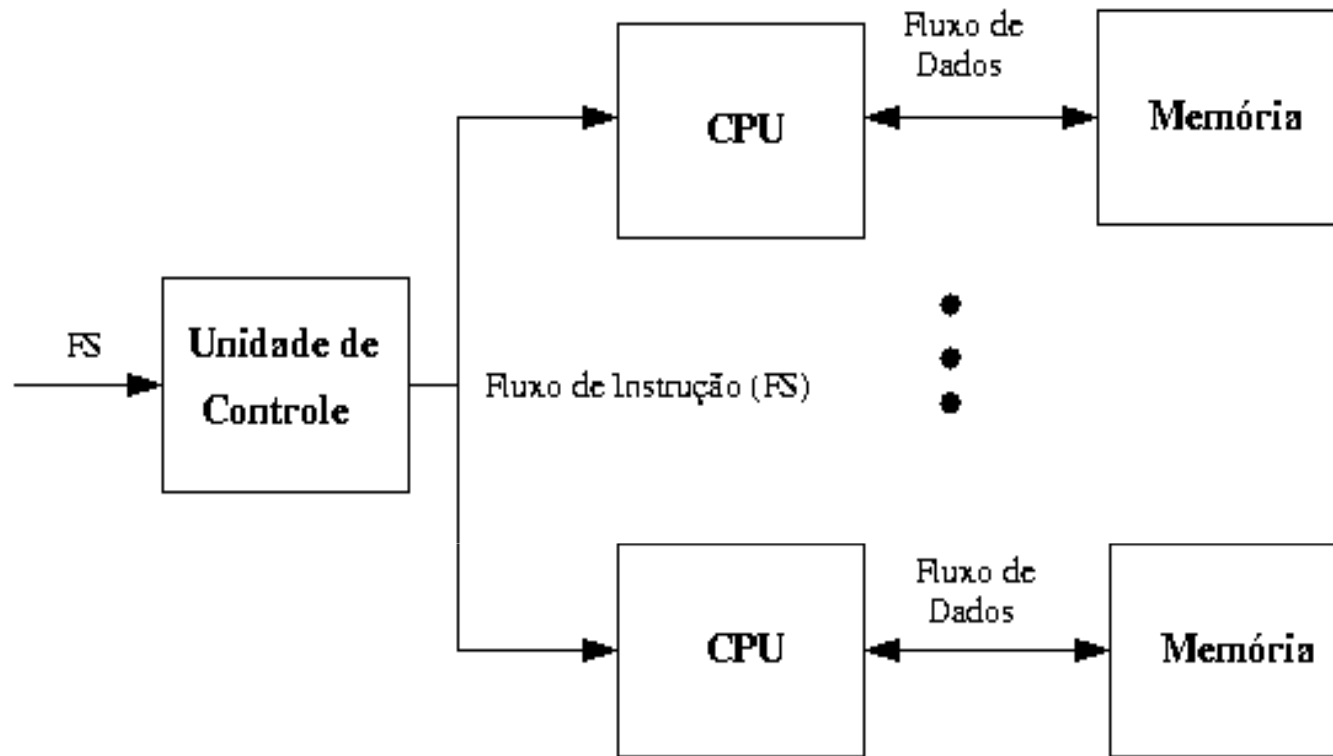
- Classifica as várias arquiteturas de computadores baseado nos fluxos de **Instruções** e de **Dados** que ocorrem no interior dos computadores
 - SISD - *Single Instruction, Single Data stream*
 - SIMD - *Single Instruction, Multiple Data stream*
 - MISD - *Multiple instruction, Single Data stream*
 - MIMD - *Multiple Instruction, Multiple Data stream*

SISD



Exemplos: Estações de trabalho e Computadores pessoais com um único processador

SIMD



Modelo de programação
simples;
Baixo overhead;
Flexibilidade

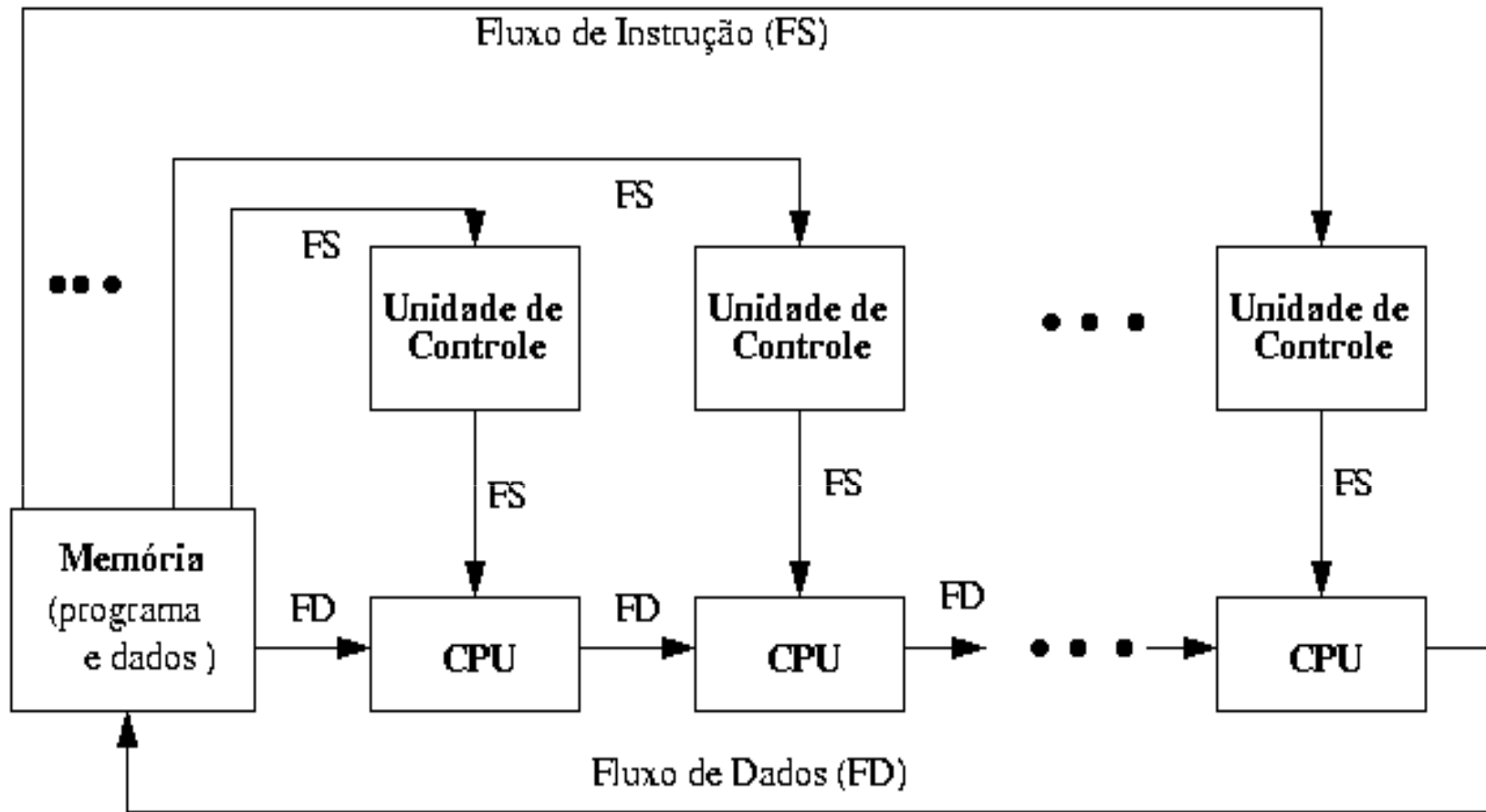
Exemplos: ILIAC IV, MPP, DHP,
MASPAR MP-2 e CPU Vetoriais

MO401-2007
Revisado

Extensões multimídia são
Consideradas como uma forma
de paralelismo SIMD

MO401
11.14

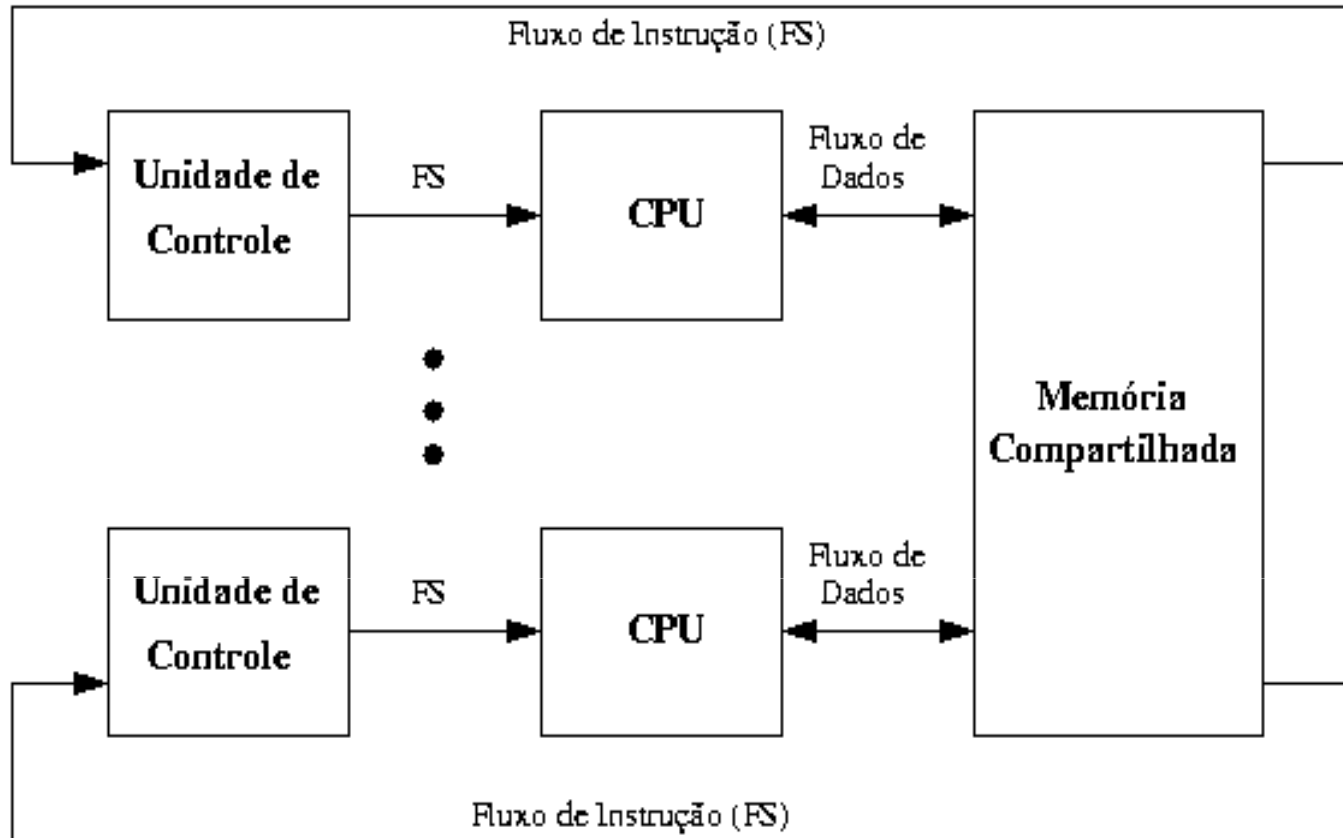
MISD



Exemplos: Array Sistólicos

Não há versões comerciais deste tipo de multiprocessadores

MIMD



Exemplos: Cosmic Cube,
nCUBE 2, iPSC, FX-2000,
SGI Origin, Sun Enterprise
5000 e Redes de
Computadores

MO401-2007
Revisado

Mais difundida
Memória Compartilhada
Memória Distribuída

Flexível
Usa microprocessadores off-the-shelf

MO401
11.16

Resumo

OBS.: O modelo de Flynn é um modelo de referência, na prática alguns multiprocessadores são híbridos em relação a essas categorias

- **MIMD**

- Maioria dos sistemas paralelos existentes
- Mais adequado à computação paralela de "propósito geral"
- Com Hw e SW corretos, alcança bom desempenho e pode ser construído com uma boa relação custo-desempenho

- **SIMD**

- Muitos dos primeiros multiprocessadores, com exceção dos processadores vetoriais, na prática "não existem mais".

- **MISD**

- Modelos para computação específica (não há implementações comerciais)

- **SISD**

- Computação Seqüencial

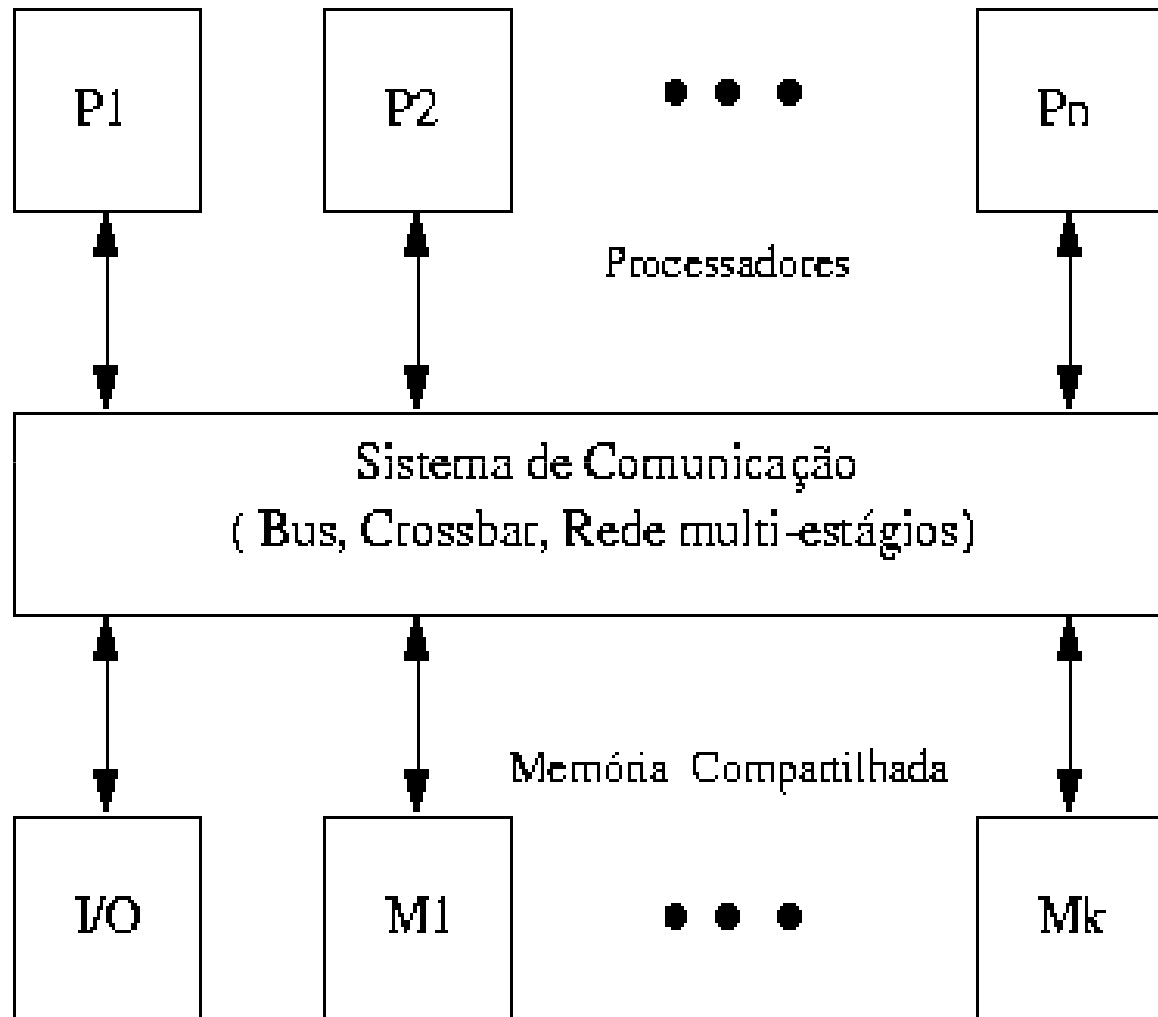
Máquinas MIMD

- Multiprocessadores de Memória Compartilhada (Memória Centralizada)
(*shared-memory multiprocessors*)
 - UMA (*Uniform-Memory-Access*)
 - NUMA (*NonUniform-Memory-Access*)
- Espaço de Endereçamento Único

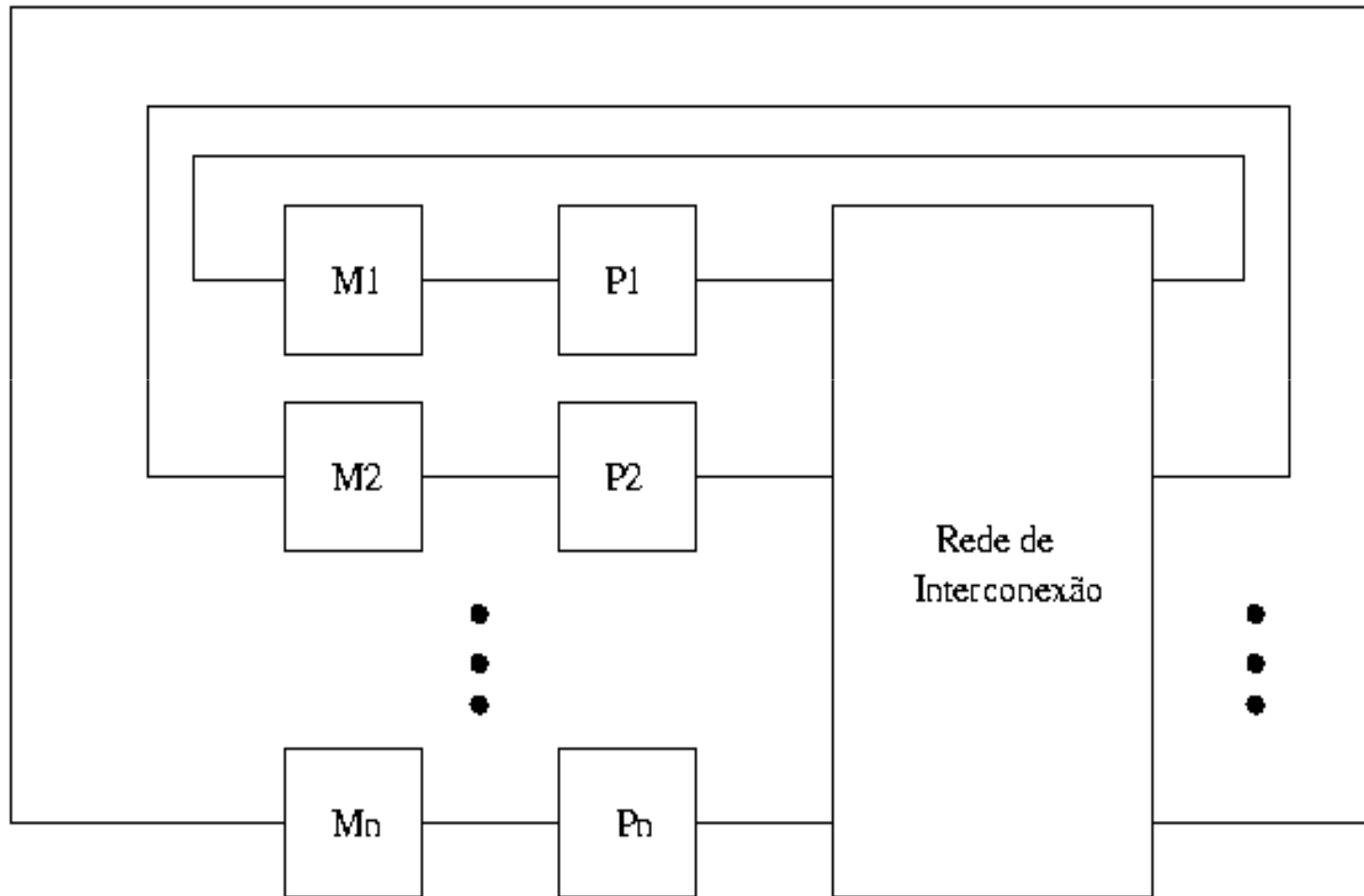
Máquinas MIMD

- Multicomputadores (Memória Descentralizada)
(*message-passing multicomputers*)
 - Espaço de endereçamento separados por processador
 - Possui maior bandwidth para as memórias e menor latência
 - Desvantagens:
 - » latência de comunicação grande
 - » Modelo de programação mais complexo
 - Pode chamar software com "Remote Procedure Call" (RPC)
 - Uso de bibliotecas, como MPI: **Message Passing Interface**
 - Também chamado de "Comunicação Síncrona" já que a comunicação sincroniza os processos que estão comunicando-se

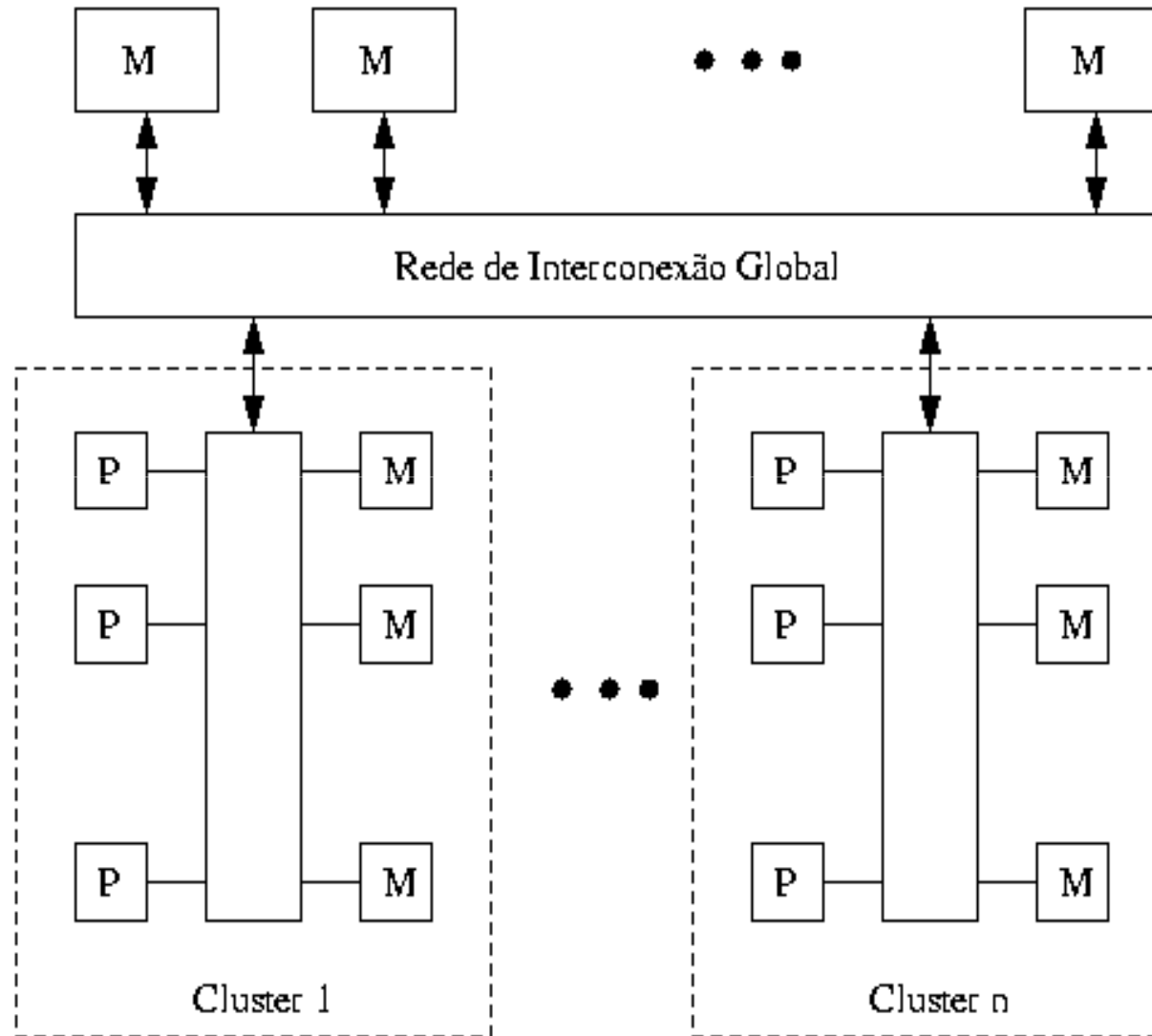
Multiprocessadores - UMA



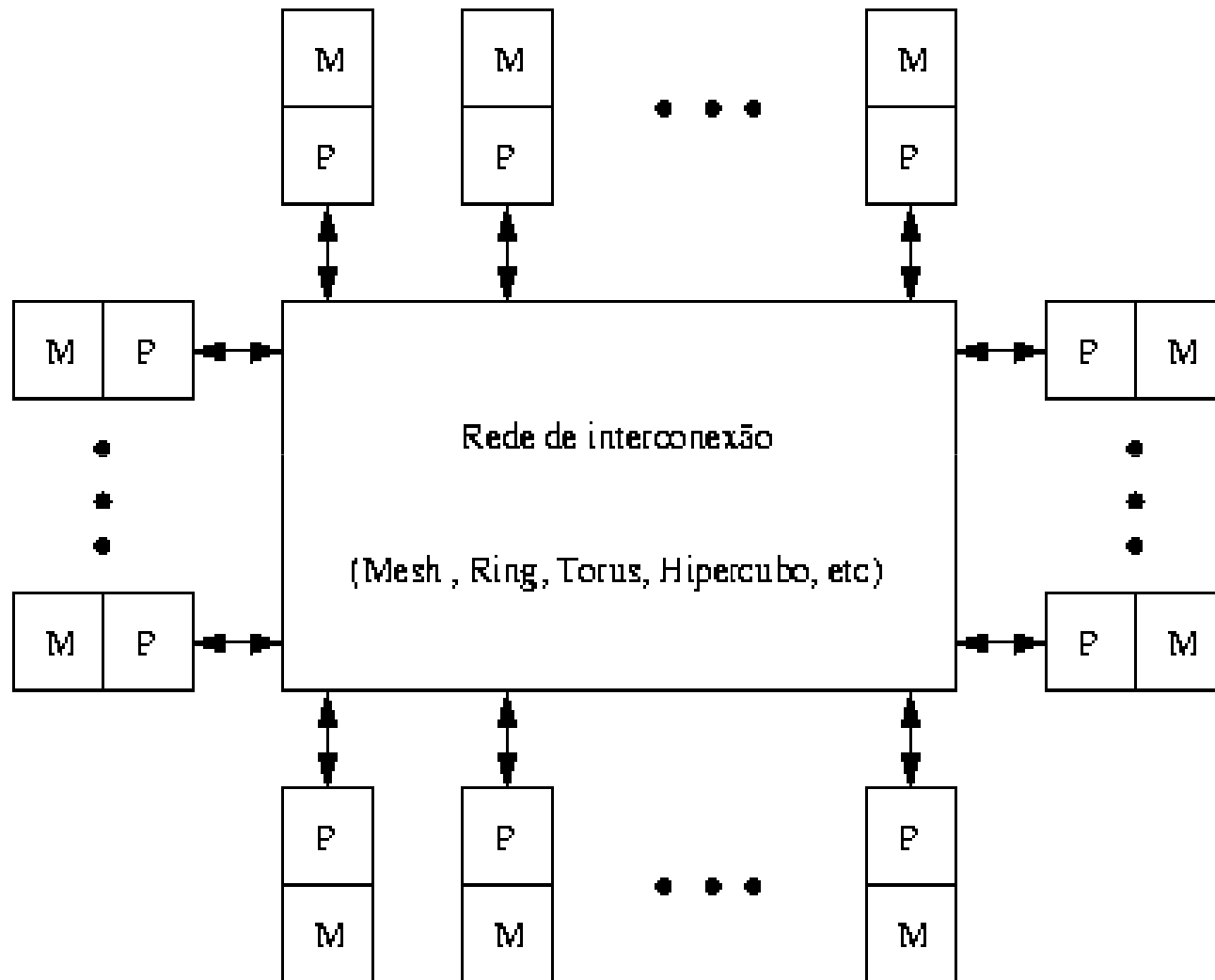
Multiprocessadores - NUMA



Multiprocessadores - NUMA



Multicomputadores

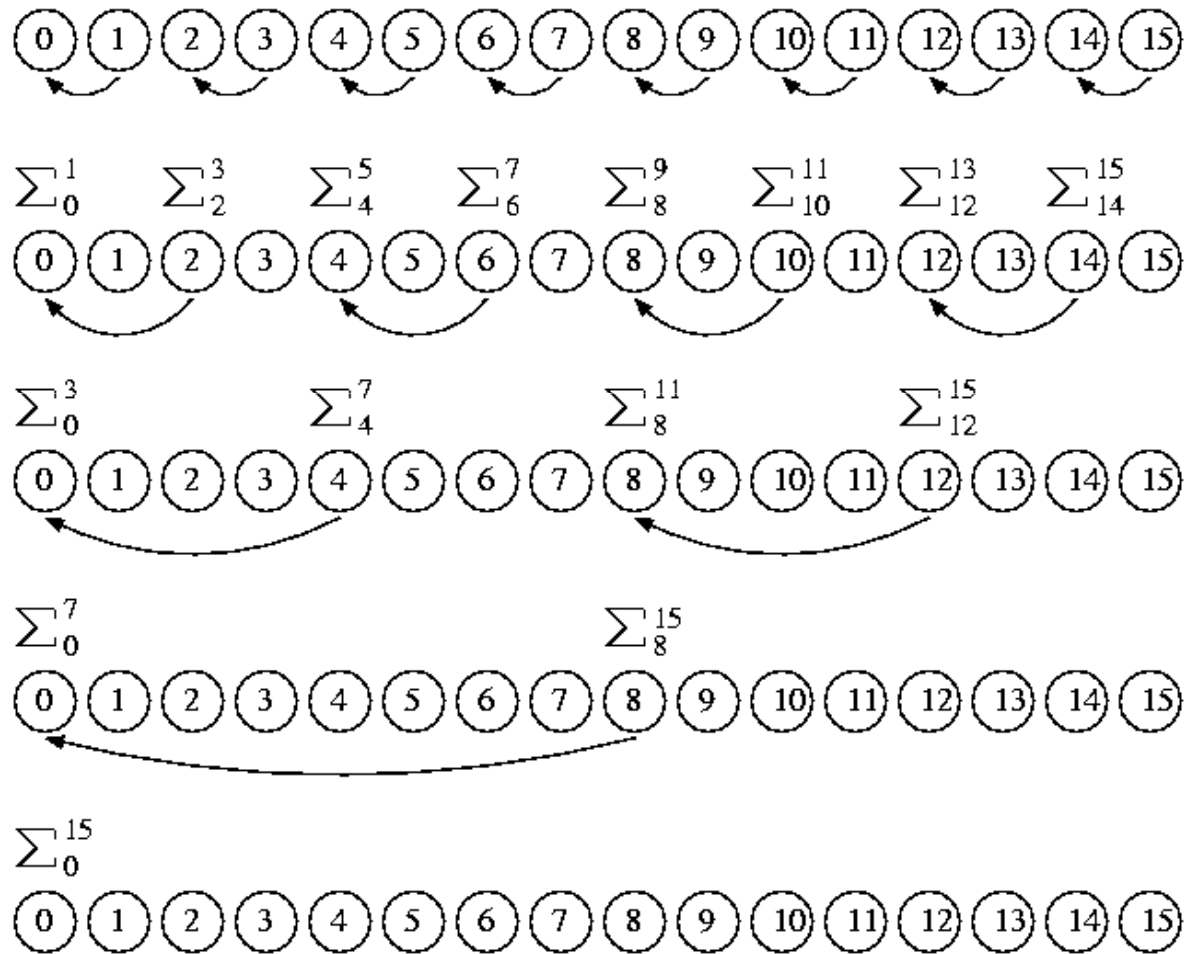


Programação Paralela

Exemplo

- Somar 16 valores, utilizando-se 16 processadores
- Quantas operações soma são realizadas?
- Qual o ganho em relação à solução usando um único processador?

Programação Paralela Uma Solução



Programação Paralela

Speedup

- Somar 16 valores, utilizando-se 16 processadores
- Quantas operações soma são realizadas?
 - Solução seqüencial = 15 operações de Soma
 - Solução paralela = 4 operações de Soma
- Qual o ganho em relação à solução usando um único processador?

$$ganho = \frac{15}{4} = 3.75$$

OBS.: 15 Comunicações?
4 Comunicações?

Desempenho

- *Speedup* - Ganho apresentado pela máquina paralela em relação a uma máquina seqüencial
- Qual o comportamento do *speedup* com o aumento do número de processadores?
 - *Ideal: N*
 - *Realidade: menor que N*

Lei de Ahmdahl

$$speedup = \frac{1}{(1-f) + \frac{f}{P}}$$

Onde: f - fração melhorada (paralelizável)

P - número de processadores

- Exemplo: Qual a fração paralelizável necessária para se alcançar um *speedup* de 200 usando-se 256 processadores?

$$200 = \frac{1}{(1-f) + \frac{f}{256}} = \frac{1}{\frac{256 - 256f + f}{256}}$$

Lei de Ahmdahl

$$200 = \frac{1}{\frac{256 - 255f}{256}}$$

$$200 = \frac{256}{256 - 255f}$$

$$f = 0.9989 = 99.89\%$$

Métricas para Desempenho

- **Bandwidth**
 - Alto bandwidth na comunicação
 - Define os limites da rede, memória e processador
 - Desafio é compatibilizar a velocidade da interface de rede com o bandwidth da rede
- **Latência**
 - Afeta o desempenho, uma vez que o processador deverá esperar
 - Afeta a complexidade de programação, já que requer soluções para sobrepor comunicação com processamento
 - Overhead de comunicação é um problema em várias máquinas
- **Esconder a Latência!**
 - Como um mecanismo pode esconder a latência?
 - Aumenta a responsabilidade do sistema de programação
 - Exemplos: overlap de envio de mensagens com computação, prefetch de dados, trocar a tarefa em execução

Arquiteturas Paralela

- Arquiteturas paralelas estende a arquitetura de computadores “tradicional” com **arquiteturas de comunicação**
 - Abstração (HW/SW interface)
 - Estrutura Organizacional que realiza a abstração de forma eficiente

Framework

- **Layers:**

- **Modelo de Programação:**

- » **Multiprogramming** : muito trabalho, sem comunicação
 - » **Shared address space**: comunicação via memória
 - » **Message passing**: send/receive de mensagens
 - » **Data Parallel**: vários agentes operando em diversos conjuntos de dados simultaneamente e trocando informações globais e simultaneamente (shared ou message passing)

- **Abstração de Comunicação:**

- » **Shared address space**: exp.: load, store, atomic swap
 - » **Message passing**: exp.: send, receive library calls

Modelo: Shared address

- Cada processador pode ter acesso a qq posição física na máquina
- Cada processo pode usar qq dado que ele compartilha com outros processos
- Transferencia de dados via load e store
- Data size: **byte**, **word**, ... ou **cache blocks**
- Usa memória virtual para mapear endereços virtuais a endereços locais ou remotos
- Modelo de hierarquia de memória é aplicável: comunicação move dados para cache local ao processador (como load move dados da memória para a cache)
 - Latência, BW, escalabilidade em relação à comunicação?

Modelo: Shared Address (Memória)

- Comunicação via Load e Store
 - Modelo mais antigo e usado
- Baseado em **timesharing**: processos em múltiplos processadores vs. compartilhamento em um processador
- **processo**: um espaço de endereçamento virtual e ~ 1 thread de controle
 - Múltiplos processos podem sobrepor (share), mas todas as **threads** compartilham o espaço de processamento do processo
- Escritas, por uma thread, em um espaço compartilhado são visíveis, para leitura, às outras threads
 - Modelo Usual: share code, private stack, algum shared heap, algum private heap
- Interconexão: processador-memória; I/O
- Bus: acesso a todas as memórias com mesmo tempo (UMA)

Modelo: Message Passing

- Os computadores (CPU, memory, I/O devices) comunicam-se por operações explícitas de I/O
- Send especifica o buffer local + processo destino no computador (nó) remoto
- Receive especifica o processo origem no nó remoto + buffer local para colocar o dado
 - Synch: quando é completado um send, quando o buffer free, quando o request é aceito, quando o receive espera por um send

Modelo: Data Parallel

- Operações podem ser executadas em paralelo em cada elemento de uma estrutura de dados regular, ex.: array
- 1 Processador de Controle: broadcast para todos PEs
- Flags de condição nos PEs podem inibir uma operação
- Dados distribuídos em cada memória

Vantagens: Modelo de Comunicação Memória Compartilhada

- Compatibilidade com o hardware SMP
- “Fácil” de programar, principalmente quando o padrão de comunicação é complexo ou altera durante a execução
- Abilidade para desenvolver aplicações usando modelos SMP, a atenção voltada somente ao desempenho de acessos críticos
- Overhead de comunicação baixo,

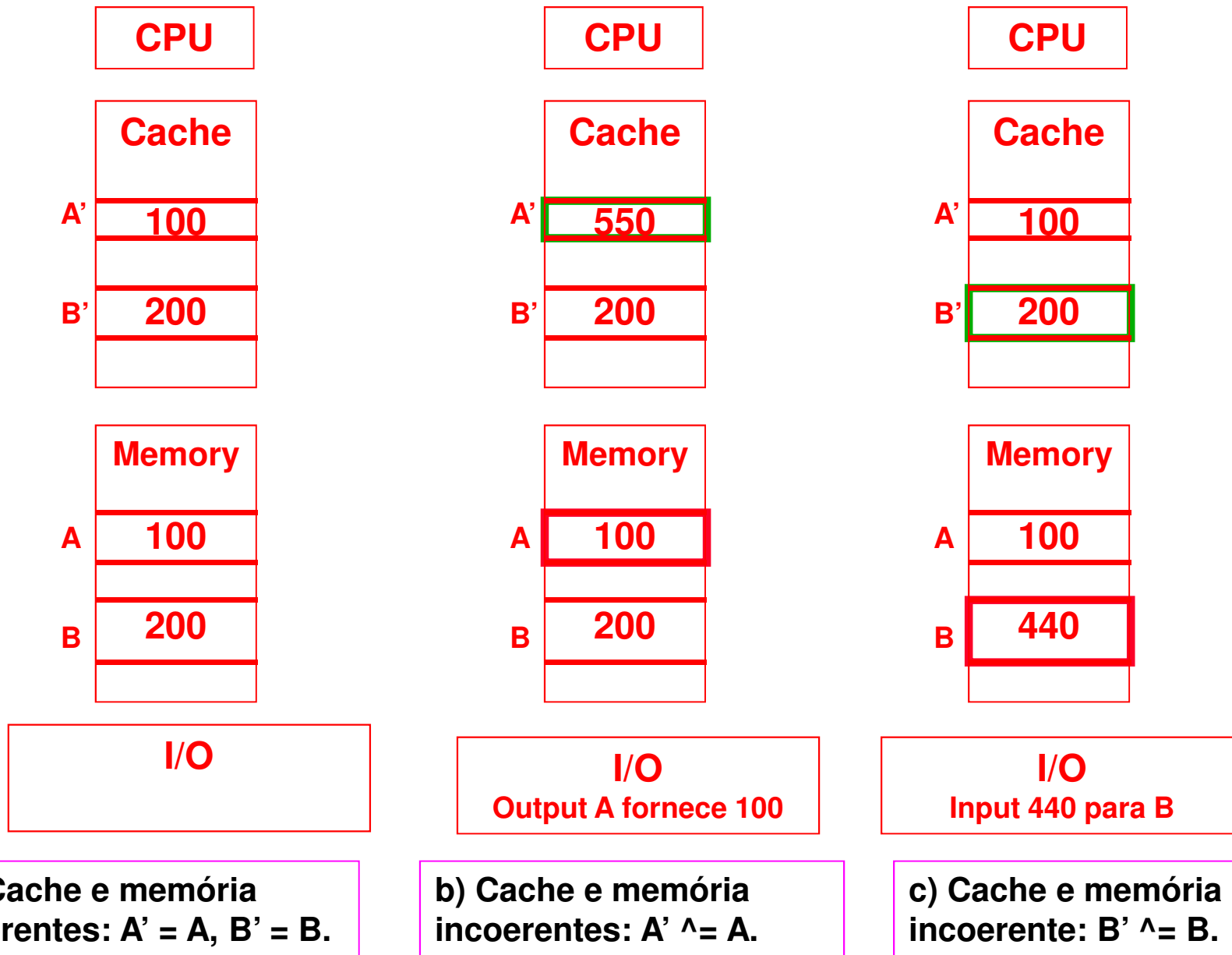
Vantagens: Modelo de Comunicação message-passing

- Hardware mais simples
- Comunicação explícita => fácil de ser entendida;
- Comunicação explícita coloca o foco das atenções nos aspectos de custo da computação paralela
- A sincronização é naturalmente associada com os envios de mensagens, reduzindo as possibilidades de erros introduzidas por sincronização incorreta

Coerência?

- Informalmente:
 - “Qualquer leitura deve retornar o valor escrito mais recente”
 - Mais difícil de ser implementado
- Melhor (para “implementação”):
 - “toda escrita deve ser vista por uma leitura”
 - Todas as escritas devem ser vistas em ordem (“serialização”)
- Duas Regras para Garantir Coerência:
 - “Se P escreve x e P1 lê x , As escritas de P serão vistas por P1 se as leituras e escritas estão suficientemente isoladas
 - Escritas em uma mesma posição deve ser serializadas:
(Vista em uma dada ordem)
 - » Será vista a última escrita
 - » Caso contrário as escritas serão vistas em uma ordem sem lógica (valor antigo no lugar do novo valor)

Coerência



Caches

Mecânismo	Operação	Desempenho	Coerência
Write Back	Write modifica o dado na cache e na memória somente quando necessário	Bom, não exige do bandwidth da memória.	Pode ter problemas com várias cópias tendo valores diferentes.
Write Through	Write modifica o dado na cache e memória juntos.	Não é Bom, usa muito bandwidth da memória.	Modifica os valores sempre que são escritos; os dados são coerentes.

Diferentes Tipos de Dados nas Caches

Qual o tipo de informação que está na Cache?

```
Test_and_set(lock)
shared_data = xyz;
Clear(lock);
```

TYPE	Shared?	Escrito?	Com Coerência?
Código	Shared	Não	Não é Preciso.
Dado Privado	Exclusive	Sim	Write Back
Dado Shared	Shared	Sim	Write Back *
Interlock Data	Shared	Sim	Write Through **

- * Write Back proporciona bom desempenho, e se usarmos write through, haverá degradação no desempenho?
- ** Write through aqui significa que o estado de lock é visto por todos imediatamente.

Possíveis Soluções para Coerência

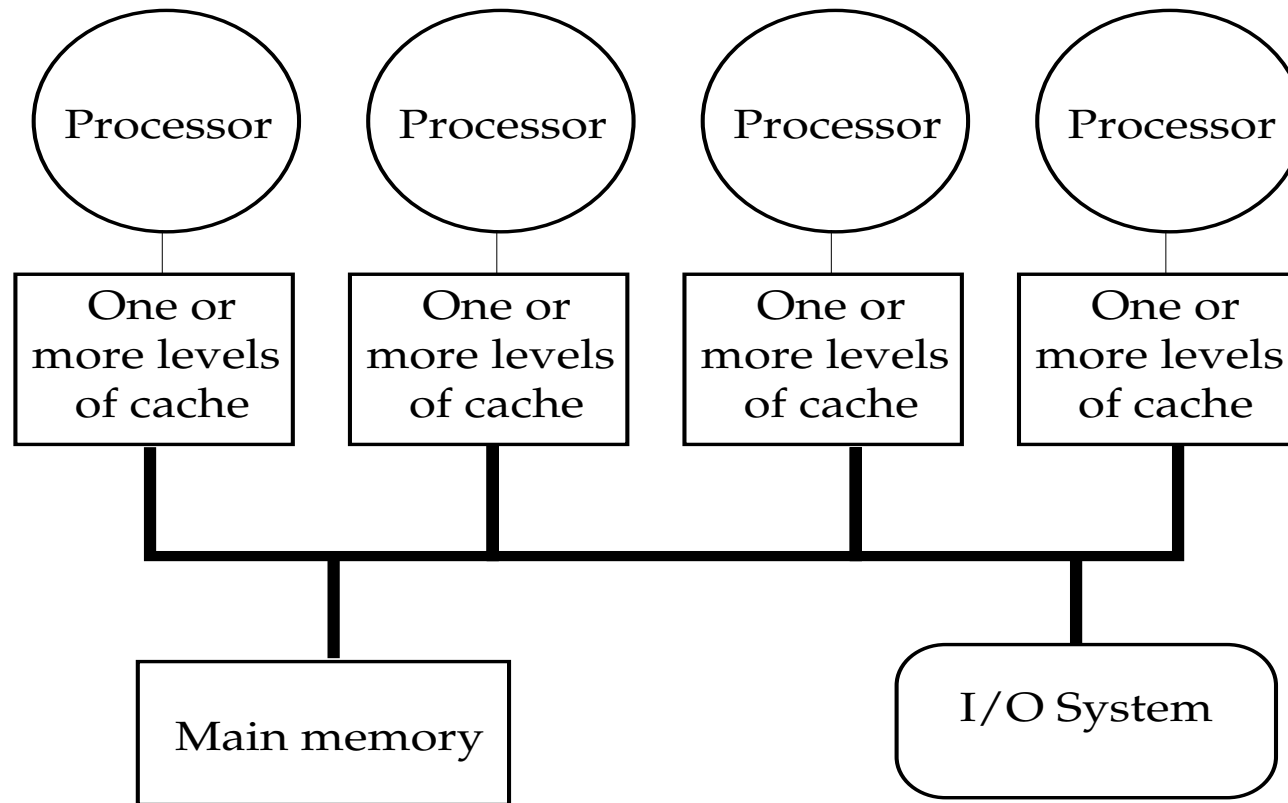
- Solução “Snooping” (Snoopy Bus):
 - Envia todos os **requests** de dados para todos os processadores
 - Processadores verificam se eles têm uma cópia e respondem
 - Requer **broadcast**, já que informações “cacheadas” estão nos processadores
 - Trabalha bem com barramento (**broadcast** é natural)
 - Domina as implementações para máquinas pouco escaláveis (maioria no mercado)

Possíveis Soluções para Coerência

- Directory-Based Schemes
 - Mantem um “relatório” do que está sendo compartilhado em um local centralizado (**lógico**)
 - Memória Distribuída => distribui o diretório para alcançar escalabilidade (evita gargalos)
 - Envia **requests** ponto-a-ponto para os processadores via network
 - Escala melhor que **Snooping**
 - É anterior aos esquemas baseados em **Snooping**

Topologia do Bus Snooping

- Memória: centralizada com tempo de acesso uniforme (“UMA”) e interconexão por barramento
- Exemplos: Sun Enterprise 5000 , SGI Challenge, Intel SystemPro



Protocolos: Snopy Básicos

- Write Invalidate Protocol:
 - Múltiplas leituras, uma escrita
 - Escrita em dado compartilhado: é enviado um "invalidate" para todas as caches com snoop e são invalidadas todas as cópias
 - Read Miss:
 - » **Write-through**: a memória está sempre atualizada
 - » **Write-back**: **snoop** nas caches para encontrar o dado mais recente
- Write Broadcast Protocol (tipicamente write through):
 - Escrita em dado compartilhado: **broadcast** no barramento, processadores **snoop**, e atualizam todas as cópias
 - Read miss: a memória está sempre atualizada
- Write Serialization: o bus serializa as requisições
 - O barramento é o único local de arbitragem

Protocolos: Snopy Básicos

- **Write Invalidate vs Broadcast:**
 - **Invalidate** requer uma transação por operação de escrita
 - **Invalidate** usa localidade espacial: uma transação por bloco
 - **Broadcast** possui menor latência entre escritas e leituras

Protocolo Snoopy: Exemplo

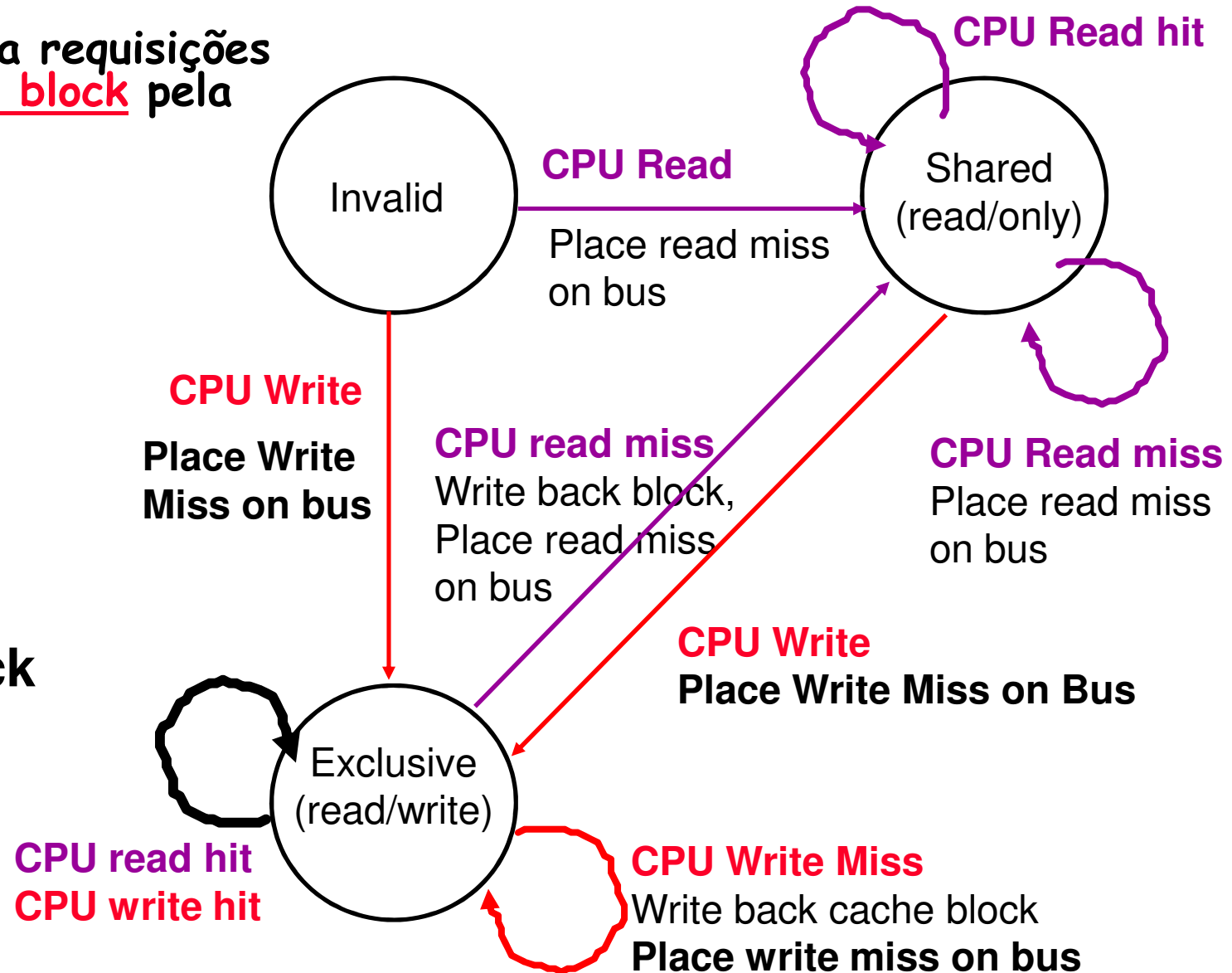
Protocolo Invalidation, Cache write-back

- Cada bloco de (na) memória está em um dos estado:
 - **Clean** em todas as caches e **up-to-date** na memória (**Shared**)
 - Ou **Dirty** em exatamente uma cache (**Exclusive**)
 - Ou Não está em nenhuma cache
- Cada bloco da (na) cache está em um dos estados:
 - **Shared** : bloco pode ser lido
 - Ou **Exclusive** : a cache tem uma cópia, ele é **writeable**, e **dirty**
 - Ou **Invalid** : o bloco não contém dado
- **Read misses**: faz todas as caches **snoop** o bus
- Writes em uma **clean line** são tratados como os misses

Snoopy-Cache: Máquina de Estados I

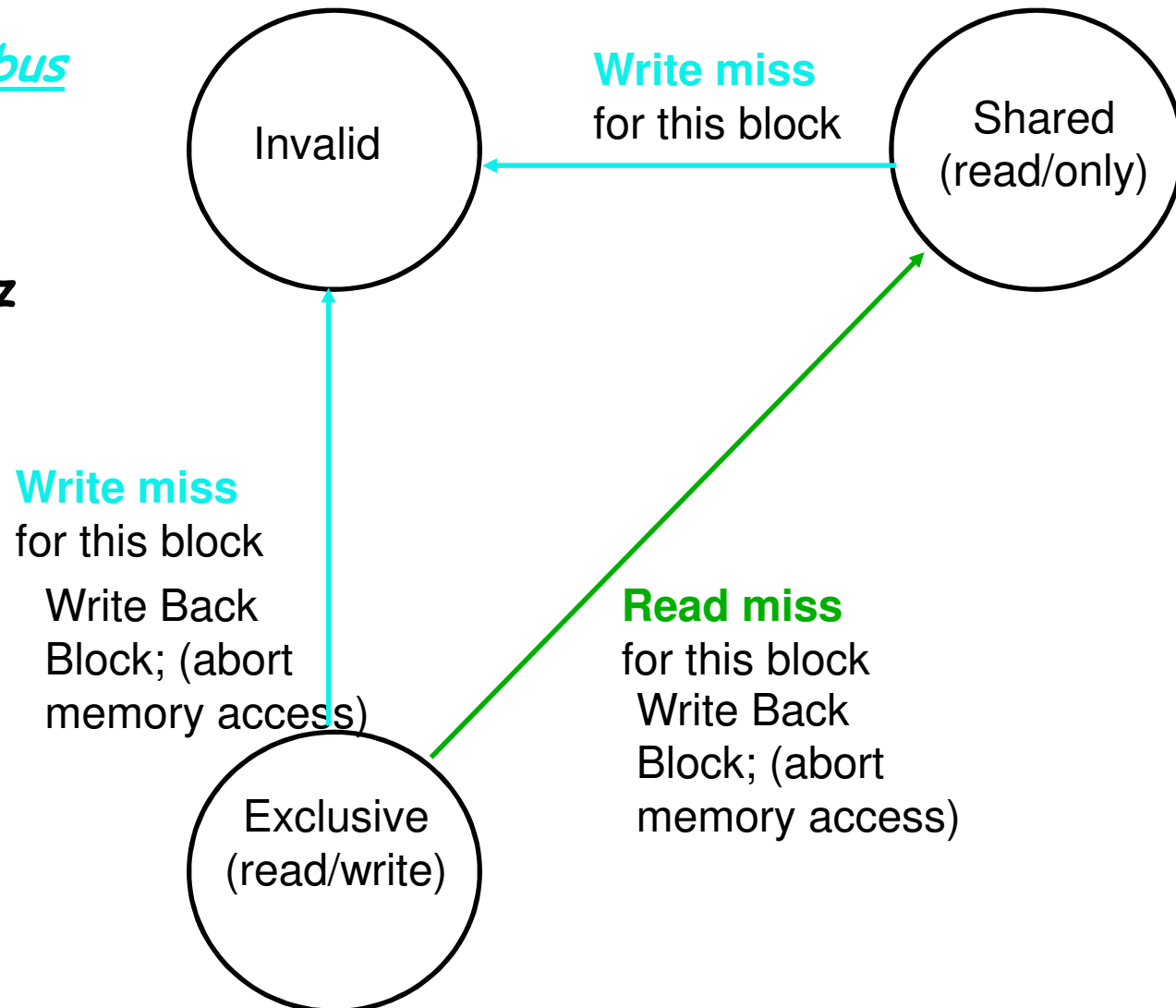
- FSM para requisições de cache block pela CPU

Estados da Cache Block



Snoopy-Cache State Machine-II

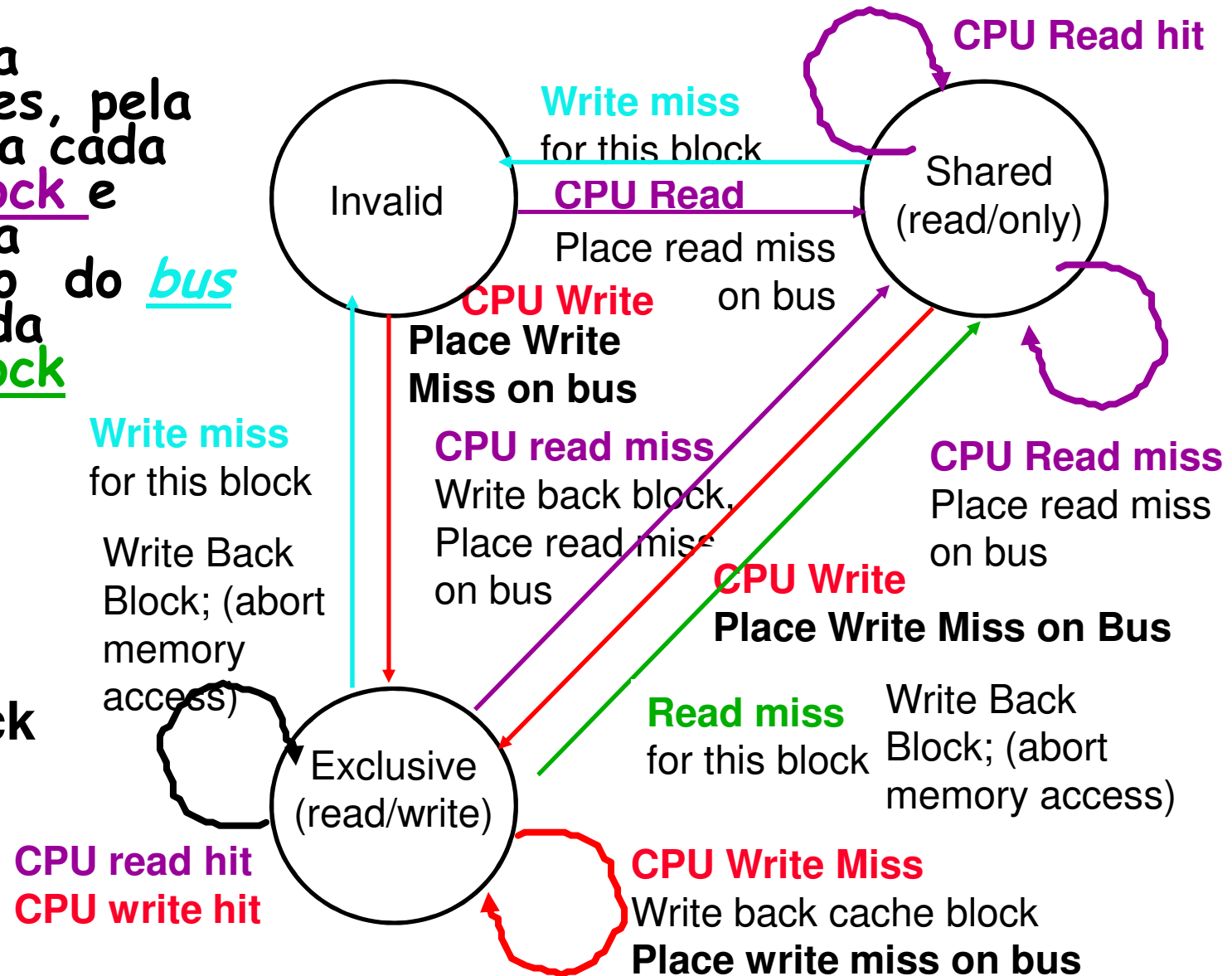
- FSM para requisições do bus para cada cache block
- Apêndice E traz detalhes sobre bus requests



Snoopy-Cache: Máquina de Estados III

- FSM para requisições, pela **CPU**, para cada **cache block** e para cada requisição do **bus** para cada **cache block**

Estados da Cache Block



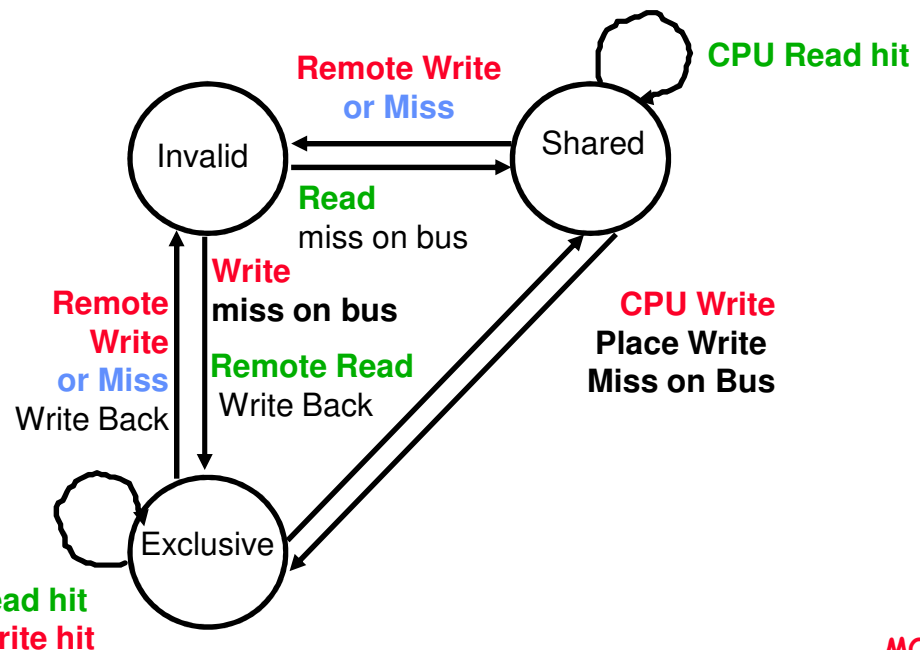
Exemplo

	Processor 1			Processor 2			Bus			Memory		
	<i>P1</i>			<i>P2</i>			<i>Bus</i>				<i>Memory</i>	
<i>step</i>	<i>State</i>	<i>Addr</i>	<i>Value</i>	<i>State</i>	<i>Addr</i>	<i>Value</i>	<i>Action</i>	<i>Proc.</i>	<i>Addr</i>	<i>Value</i>	<i>Addr</i>	<i>Value</i>
P1: Write 10 to A1												
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assuma que A1 e A2 são mapeados para o mesmo bloco da cache, o estado da cache inicial é **Invalid** e $A1 \neq A2$

Arco ativo: 

Cache de P1.

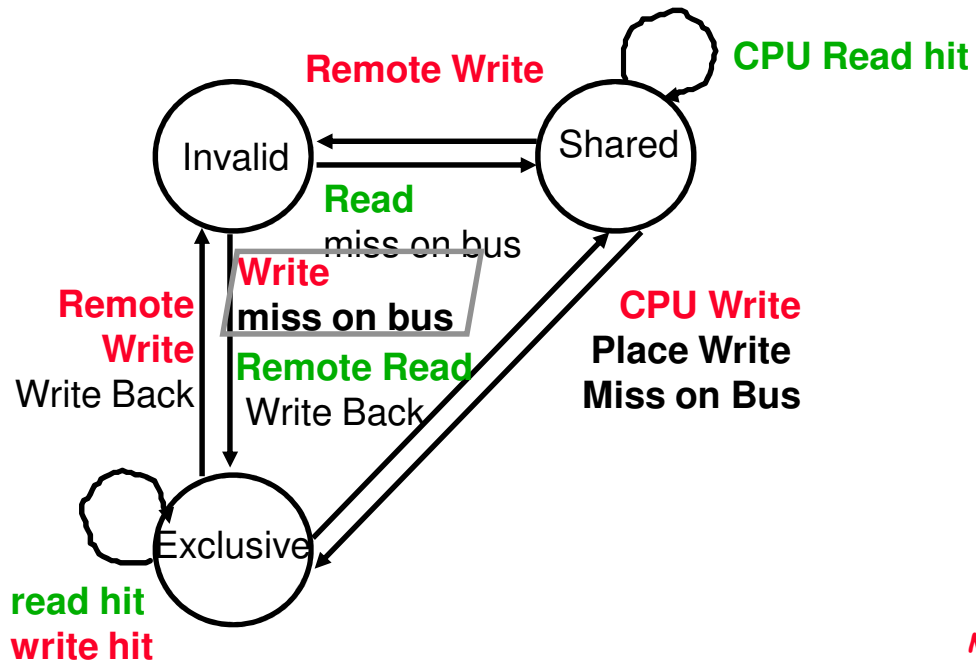


Exemplo: passo 1

	P1			P2			Bus			Memory		
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	<u>Excl.</u>	<u>A1</u>	<u>10</u>				<u>WrMs</u>	P1	A1			
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assuma que A1 e A2 são mapeados para o mesmo bloco da cache, o estado da cache inicial é **Invalid** e $A1 \neq A2$

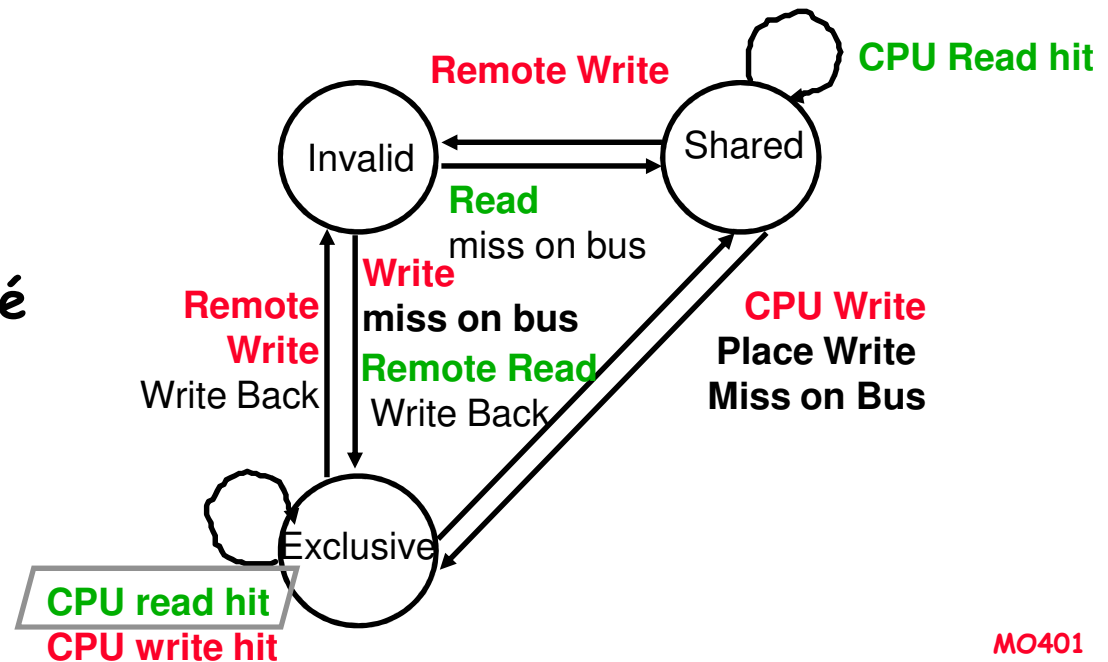
Aresta Ativa:



Exemplo: passo 2

step	P1			P2			Bus				Memory	
	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	<u>Excl.</u>	<u>A1</u>	<u>10</u>				<u>WrMs</u>	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

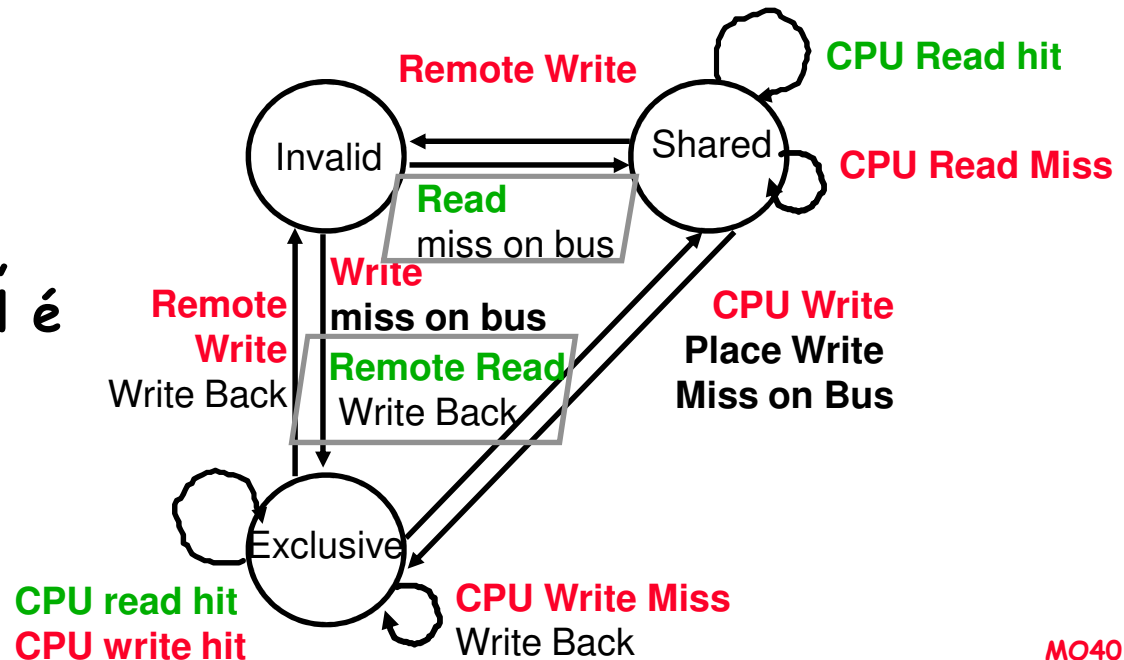
Assuma que A1 e A2
são mapeados para
o mesmo bloco da cache,
o estado da cache inicial é
Invalid e $A1 \neq A2$



Exemplo: passo 3

	P1			P2			Bus			Memory		
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	<u>Excl.</u>	<u>A1</u>	<u>10</u>				<u>WrMs</u>	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				<u>Shar.</u>	<u>A1</u>		<u>RdMs</u>	P2	A1			
	<u>Shar.</u>	A1	10				<u>WrBk</u>	P1	A1	10	<u>A1</u>	<u>10</u>
				Shar.	A1	<u>10</u>	<u>RdDa</u>	P2	A1	10		10
P2: Write 20 to A1												
P2: Write 40 to A2												

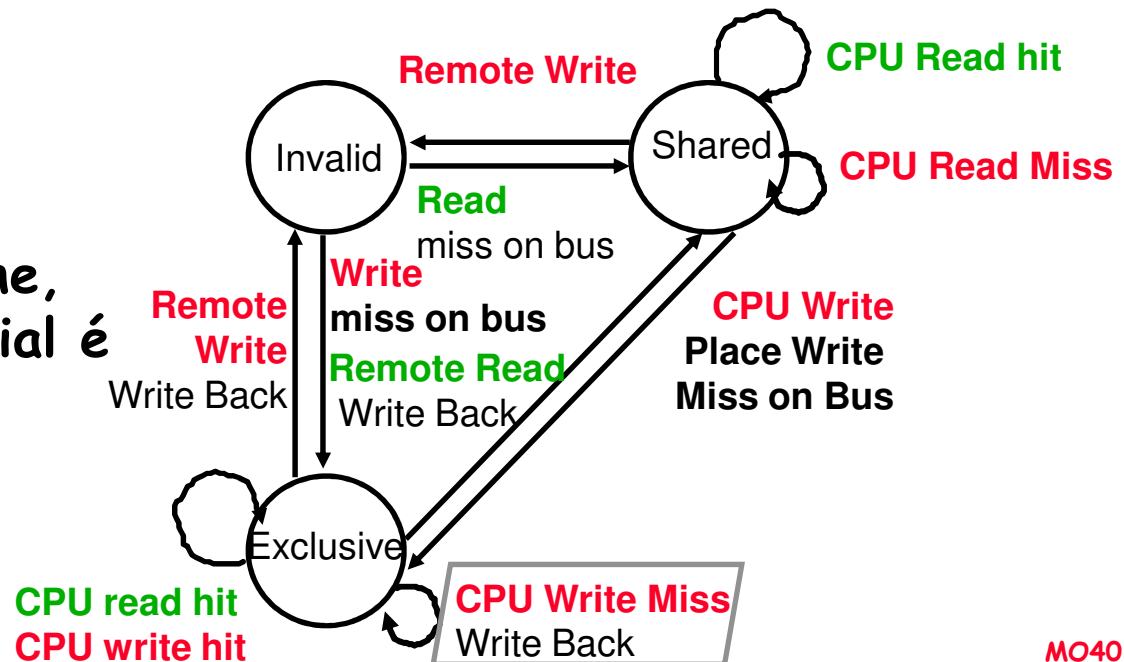
Assuma que A1 e A2 são mapeados para o mesmo bloco da cache, o estado da cache inicial é **Invalid** e $A1 \neq A2$



Exemplo: passo 5

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	<u>Excl.</u>	<u>A1</u>	<u>10</u>				<u>WrMs</u>	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				<u>Shar.</u>	<u>A1</u>		<u>RdMs</u>	P2	A1			
	<u>Shar.</u>	A1	10				<u>WrBk</u>	P1	A1	10	<u>A1</u>	<u>10</u>
				Shar.	A1	<u>10</u>	<u>RdDa</u>	P2	A1	10		10
P2: Write 20 to A1	<u>Inv.</u>			<u>Excl.</u>	A1	<u>20</u>	<u>WrMs</u>	P2	A1			10
P2: Write 40 to A2							<u>WrMs</u>	P2	A2			10
				Excl.	<u>A2</u>	<u>40</u>	<u>WrBk</u>	P2	A1	20	<u>A1</u>	<u>20</u>

Assuma que A1 e A2 são mapeados para o mesmo bloco da cache, o estado da cache inicial é **Invalid** e $A1 \neq A2$



Variações de Snooping Cache

Basic Protocol	Berkeley Protocol	Illinois Protocol	MESI Protocol
Exclusive	Owned Exclusive	Private Dirty	<u>M</u> odified (private, !=Memory)
Shared	Owned Shared	Private Clean	<u>E</u> xclusive (private, =Memory)
Invalid	Shared	Shared	<u>S</u> hared (shared, =Memory)
	Invalid	Invalid	<u>I</u> nvalid

Implementação

Restrições (ou complicações)

- **Write:**
 - Não pode ser atualizado até a aquisição do barramento
 - » Além disso, outro processador pode obter o barramento primeiro e escrever o mesmo bloco!
 - Processo de 2 passos:
 - » Arbitragem do barramento
 - » Informar miss no barramento e completar a operação
 - Se um miss ocorre para o bloco enquanto aguarda pelo barramento, o miss deve ser tratado (pode ser necessário invalidar) e reiniciar.
 - Transações “quebradas” no barramento:
 - » As transações no bus não são atômicas: pode haver múltiplas transações em progresso para o bloco
 - » Múltiplos misses podem ser sobrepostos, permitindo que duas caches prendam o bloco no estado **Exclusive**

Implementando Snooping Caches

- Múltiplos processadores devem usar o barramento (acesso de endereços e dados)
- Adicionar alguns comandos novos para manter a coerência, em adição à leitura e escrita
- Os Processadores continuamente deve monitorar o barramento de endereços
 - Se há "address matches tag", então **invalidate** ou **update**
- Já que toda transação no barramento verifica a cache tag, pode interferir na verificação da CPU:
 - solução 1: duplicar as tags para a cache L1, permitindo a verificação em paralelo com a CPU
 - solução 2: Cache L2, já tem uma duplicata, use essa tag (maior integração entre L1 e L2)
 - » block size, associatividade de L2 afeta L1

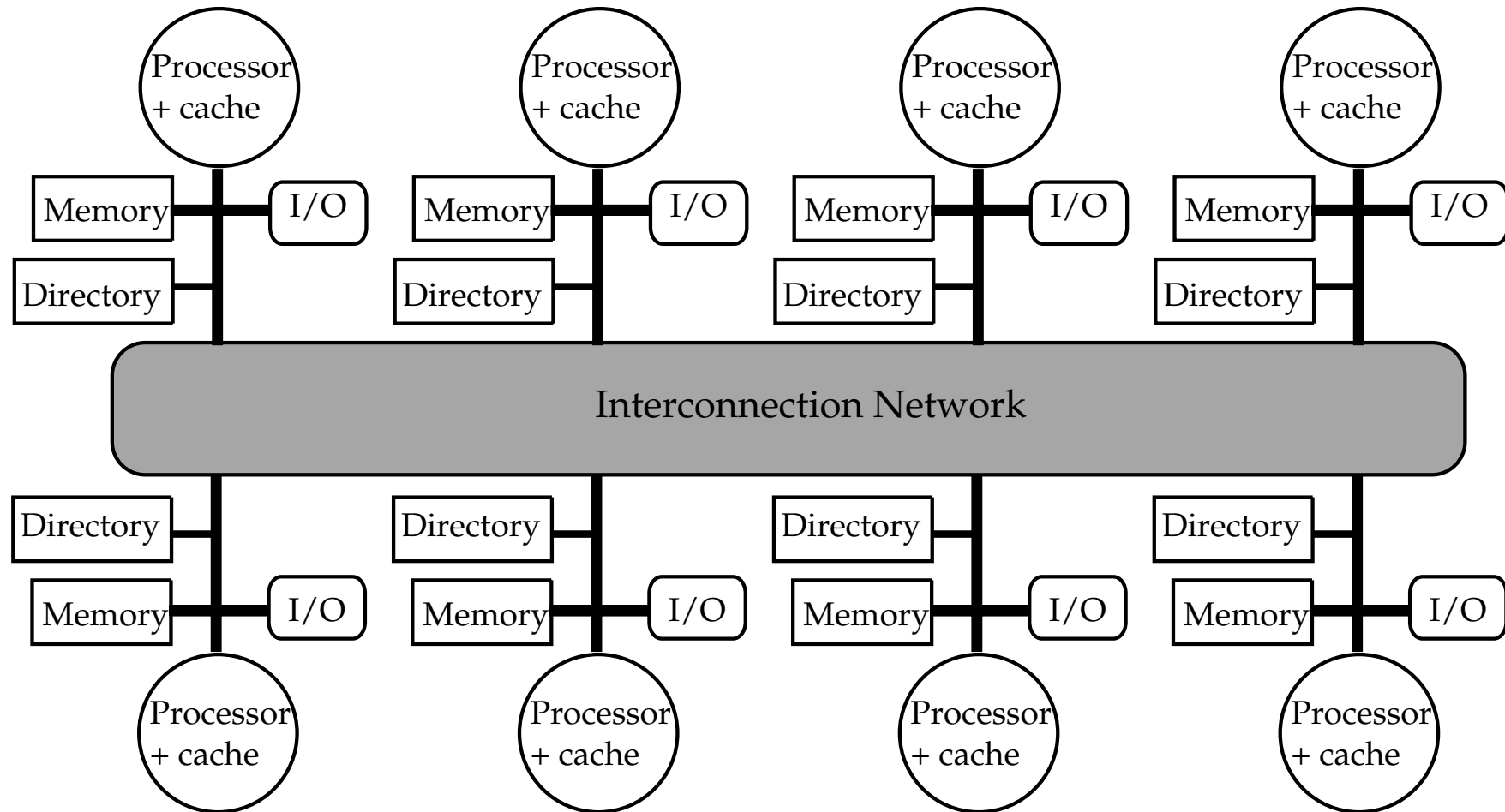
Implementando Snooping Caches

- O barramento serializa os writes, mantendo-se o barramento, nenhuma outra operação pode ser executada na memória
- Adição de bit extra na cache para determinar se compartilhado ou não.
- Adicionar um quarto estado (MESI)

Grandes MPs

- Memórias (separadas) por Processador
- Acesso Local ou Remoto via controlador de memória
- Uma solução: não incluir **Coerência de Cache**
(conceito de dados locais e remotos e dados compartilhados são marcados como Não "Cacheado")
- Alternativa: **diretório** por cache que monitora o estado de todos os blocos em todas as caches
 - Qual cache tem uma cópia do bloco, dirty vs. clean, ...
- Info por bloco de memória vs. por bloco de cache?
- O directório pode se tornar um gargalo !!!
 - Distribuir o diretório pelas memórias

Diretórios Distribuídos em MPs



Directory Protocol => Trabalho

Questões Fundamentais

- 3 questões fundamentais caracterizam as máquinas paralelas

1. Classificação (Nomenclatura)

2. Sincronização

3. Desempenho: Latency and Bandwidth

Classificação

- A Classificação define como resolver o problema
 - Qual dado é compartilhado
 - Como ele é endereçado
 - Quais operações podem ter acesso aos dados
 - Como cada processo interage com os outros processos
- A escolha de uma Classificação afeta a produção do código pelo compilador; via load (é necessário um endereço) ou troca de mensagem (número do processador e endereço virtual)
- A escolha de uma Classificação afeta a replicação de dados; via load na hierarquia de memória ou via replicação por SW e consistência

Classificação

- **Global physical address space:** qualquer processador pode gerar endereços e acessá-los em uma única operação
- **Global virtual address space:** Se o espaço de endereçamento de cada processo pode ser configurado para conter todo os dados compartilhados pelo programa paralelo
- **Segmented shared address space:**
A localização é endereçada por
<process number, address>
e de forma uniforme por todos os processos do programa paralelo

Sincronização

- Para haver cooperação os processos devem ser coordenados
- Troca de Mensagens implementa uma coordenação implícita com o envio e recebimento dos dados
- **Shared address**
=> operações adicionais para implementar uma coordenação explícita

Sincronização

- Sincronização?
 - Necessário para se conhecer quando é seguro, para os diferentes processos, usar o dado compartilhado
- Mecanismo para Sincronização:
 - Operação atômica para buscar e atualizar dados na memória (sem interrupção);
 - Operações de sincronização no nível **Usuário** que usem essas primitivas;
 - Para MPs em larga escala, a sincronização pode ser o(um) gargalo; Técnicas para reduzir a contenção e as latências nos mecanismos de sincronização

Instruções para busca e atualização de dados na memória

- **Atomic exchange**: troca de um valor em um registrador com um valor em memória
 - 0 => sincronismo está livre
 - 1 => sincronismo está bloqueado e indisponível
 - "Seta" registrador em 1 & swap
 - Novo valor no registrador determina o sucesso em obter o "lock"
 - 0 se sucesso em "setar" o lock (é o primeiro)
 - 1 se outro processo já obteve sucesso
 - O importante é que a operação é indivisível
- **Test-and-set**: testa um valor e "seta" se passar no teste
- **Fetch-and-increment**: retorna o valor para a posição de memória e, atomicamente, o incrementa
 - 0 => sincronismo está livre