Arquitetura de Processadores, Taxonomia de Flynn e Hierarquia de Memória

João Marcelo Uchôa de Alencar
joao.marcelo@ufc.br
UFC-Quixadá

Introdução

Arquitetura de Processadores e Tendências

A Taxonomia de Flynn

Organização de Memória de Computadores Paralelos

Paralelismo a Nível de *Thread*

Arquitetura de Processadores *Multicore*

Redes de Interconexão

Roteamento e *Switching*

Caches e Hierarquia de Memória

Consistência de Memória

Glossário de Termos

- Tarefas (*tasks*): computações que constituem uma aplicação;
- granularidade: tamanho da tarefa (em termos de esforço computacional);
- paralelismo potencial: características do algoritmo da aplicação que podem guiar a decomposição em tarefas;
- processos e *threads*;
- escalonamento (*scheduling*: alocação de tarefas a processos ou *threads*;
- mapeamento: alocação de processos ou *threads* a processadores ou núcleos;
- sincronização de tarefas;
- memória compartilhada;
- memória distribuída;
- operações de comunicação;
- barreiras;
- tempo de execução paralela;
- balanceamento de carga;
- *speedup* e eficiência.

Arquitetura de Processadores

- Processadores são feitos de transistores;
- lei de Moore: o número de transistores em um *chip* processador dobra a cada 18-24 meses;
- mais transistores permitem adicionar unidades funcionais, *caches* e registradores;
- o resultado é o aumento da velocidade do *clock*;
- *benchmarks*: `www.spec.org`

No momento atual, a lei de Moore está esgotada. Mas mesmo antes do seu fim, projetistas utilizaram paralelismo para aprimorar seus processadores.

Paralelismo a Nível de Palavra

Precisão e endereçamento de memória.

- Processadores evoluíram de palavras de 4 para 8, 16 e 32 *bits*;
- a indústria adotou 64 *bits* como o padrão;
- suficiente para representação de números em ponto flutuante;
- capaz de endereçar quantidades absurdas de memória.

Alguns processadores específicos utilizam palavras maiores.

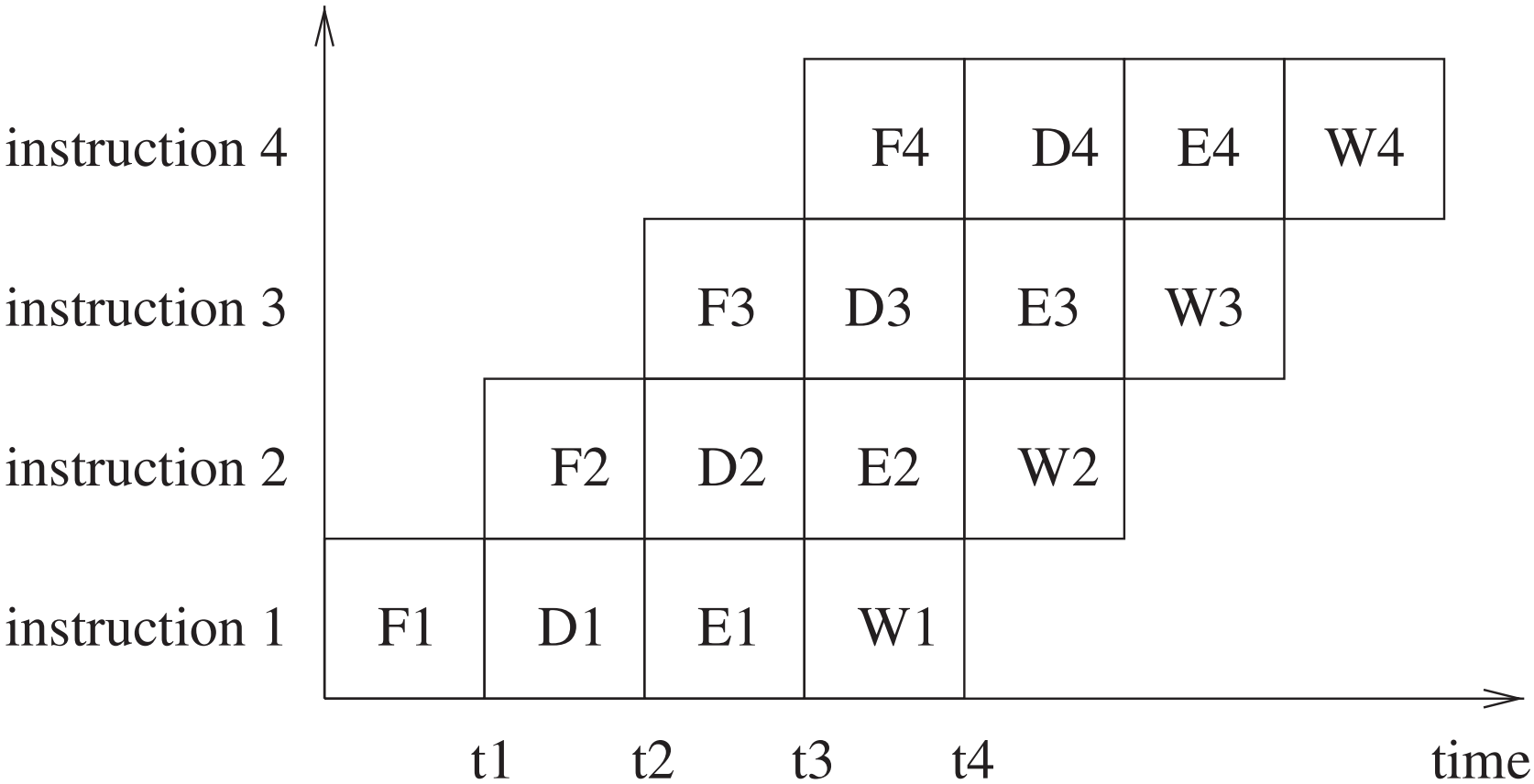
Paralelismo por *Pipeline*

Dividir a execução de cada instrução em várias etapas

- *fetch*: recuperar instrução da memória;
- *decode*: decodificar a instrução;
- *execute*: recuperar os operandos da memória e executar a instrução;
- *write-back*: armazenar o resultado no registrador alvo.

Para otimizar o projeto, o ideal é que cada estágio do *pipeline* tenha a mesma duração.

Paralelismo por *Pipeline*



Paralelismo por *Pipeline*

- **Nível de paralelismo**: quantidade de estágios no *pipeline*;
- em geral, temos algo entre 2 ou 26 estágios;
- paralelismo a nível de instrução;
- existe um limite em o quanto uma instrução pode ser dividida.

Paralelismo por Múltiplas Unidades Funcionais

- Um processador tem várias unidades funcionais:
  - Unidade lógica e aritmética;
  - unidade para operações em ponto flutuante;
  - unidades de interface com a memória (*load/store*);
  - dentre outras...
- essas unidades podem atuar em paralelo, cada uma executando uma instrução ao mesmo tempo;
- processadores superescalares replicam unidades para aumentar ainda mais o paralelismo;

A dependência entre as instruções limita a escalabilidade desse projeto.

Paralelismo ao Nível de Processo/*Thread*

- As soluções anteriores trabalham no fluxo sequencial de instruções gerado pelo compilador;
- com mais de um núcleo ou processador, o desenvolvedor precisa definir fluxos para cada *chip*;
- processadores **multicore**;
- o acesso à memória é compartilhado, precisam ser coordenados.

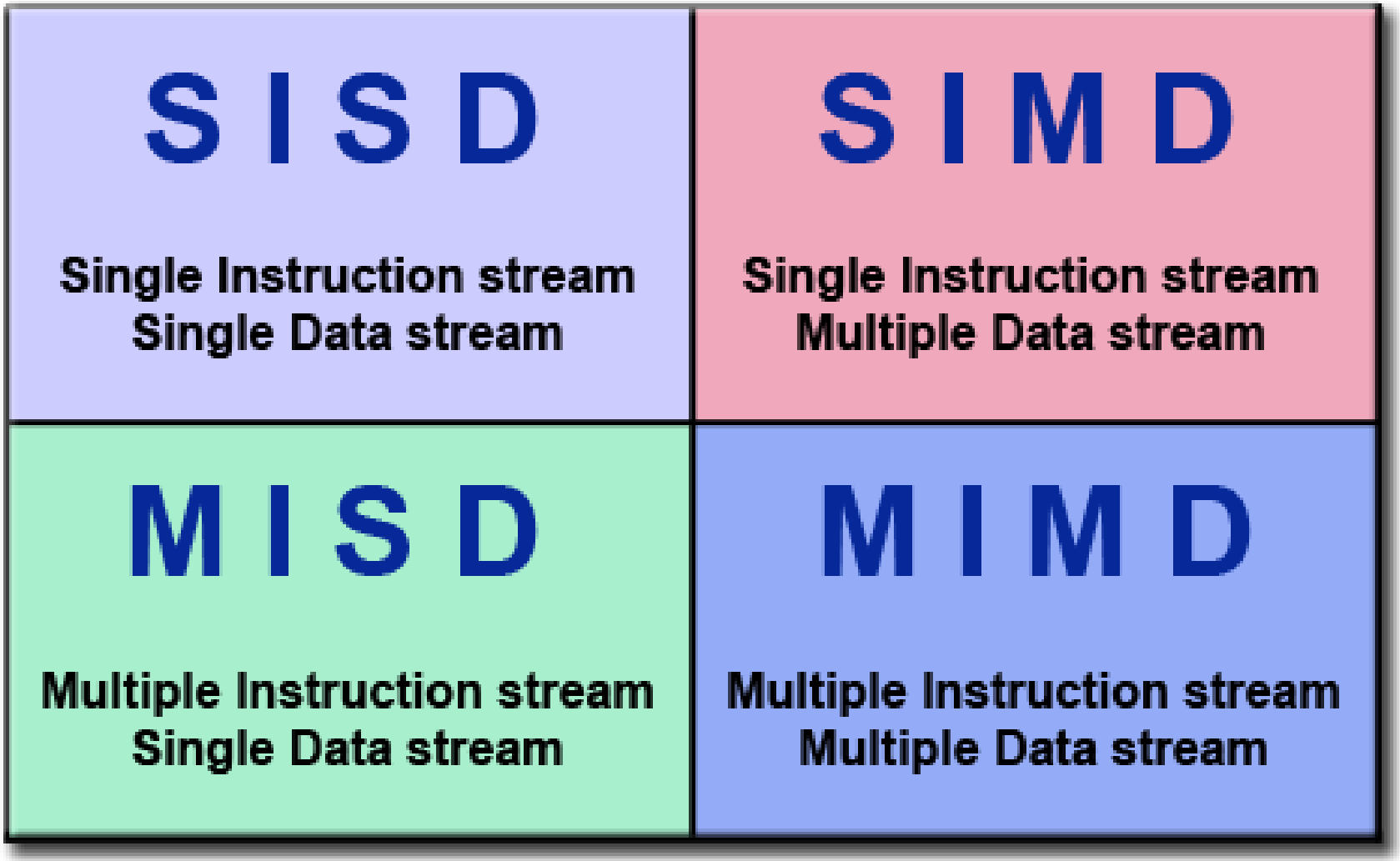
Taxonomia de Flynn

Computador Paralelo

Uma coleção de elementos de processamento que podem se comunicar e cooperar para resolver grandes problemas.

- Número e complexidade dos elementos de processamento?
- estrutura da rede de interconexão?
- coordenação dos elementos?

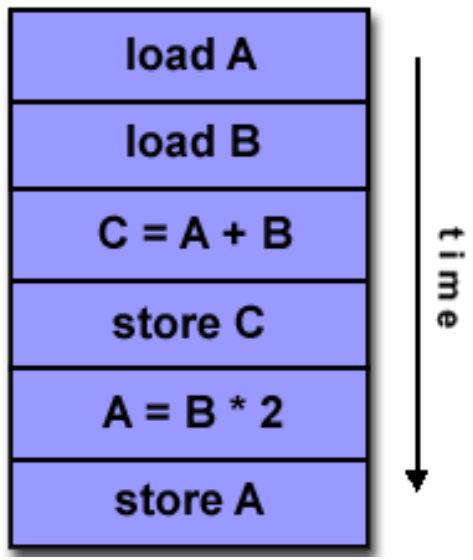
A **taxonomia de Flynn** caracteriza computadores paralelos de acordo com o controle global e o fluxo de dados e controle.



https://computing.llnl.gov/tutorials/parallel_comp/

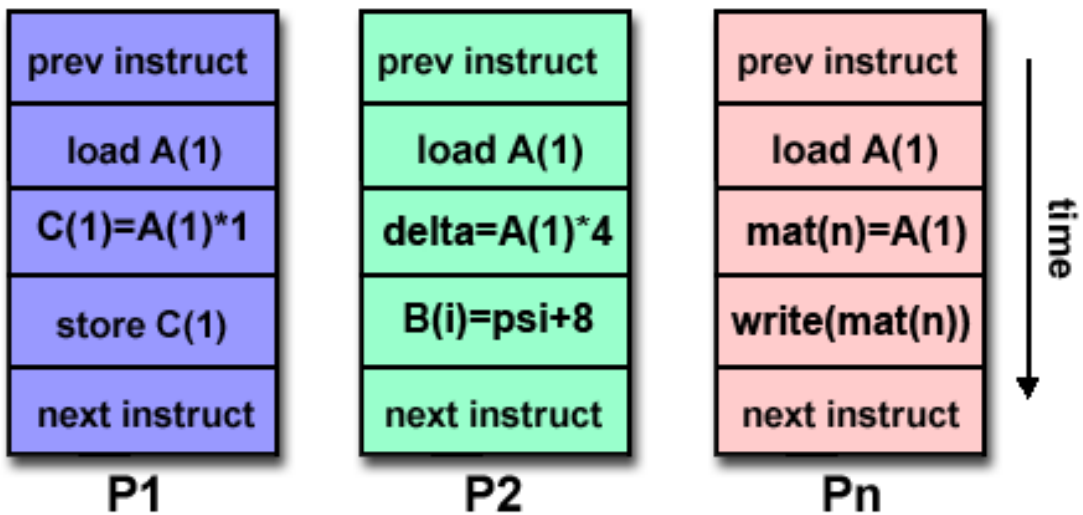
Single-Instruction, Single-Data (SISD)

- Um computador serial, não paralelo;
- modelo de *von Neumann*;
- a cada etapa, uma instrução é carregada, executada e o resultado armazenado na memória.



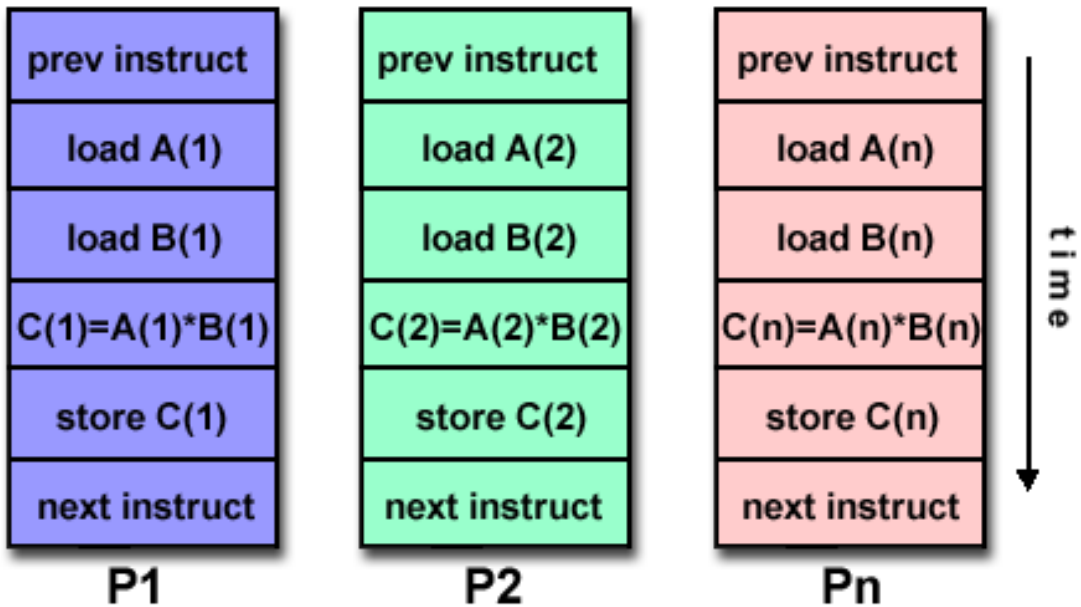
Multiple-Instruction, Single-Data (MISD)

- Vários elementos de processamento;
- cada um executando código diferente;
- atuando na mesma unidade de dados.
- exemplo pouco utilizado.



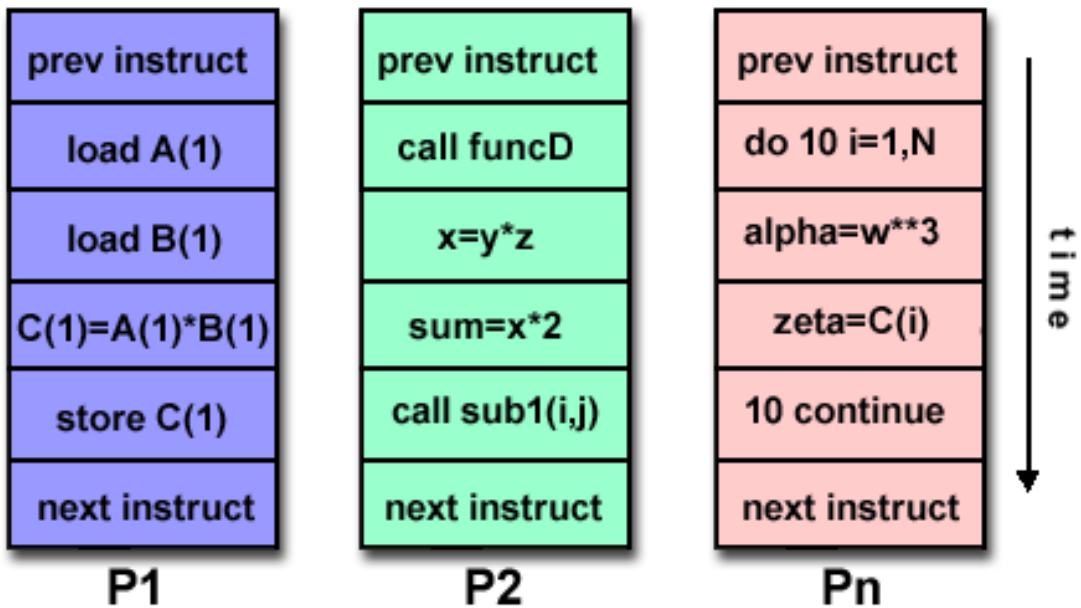
Single-Instruction, Multiple-Data (SIMD)

- Vários elementos de processamento;
- todos executando o mesmo código;
- atuando em unidades de dados diferentes;
- uma mesma instrução é executada de forma síncrona em seções diferentes dos dados disponíveis;
- bastante utilizado na prática.



Multiple-Instruction, Multiple-Data (MIMD)

- Vários elementos de processamento;
- cada um executando código diferente;
- em unidades de dados diferentes;
- são os processadores *multicore* de hoje em dia.



Considerações sobre a Taxonomia

Considere o código abaixo:

```
if (b == 0)
   c = a;
else
   c = a/b;
```

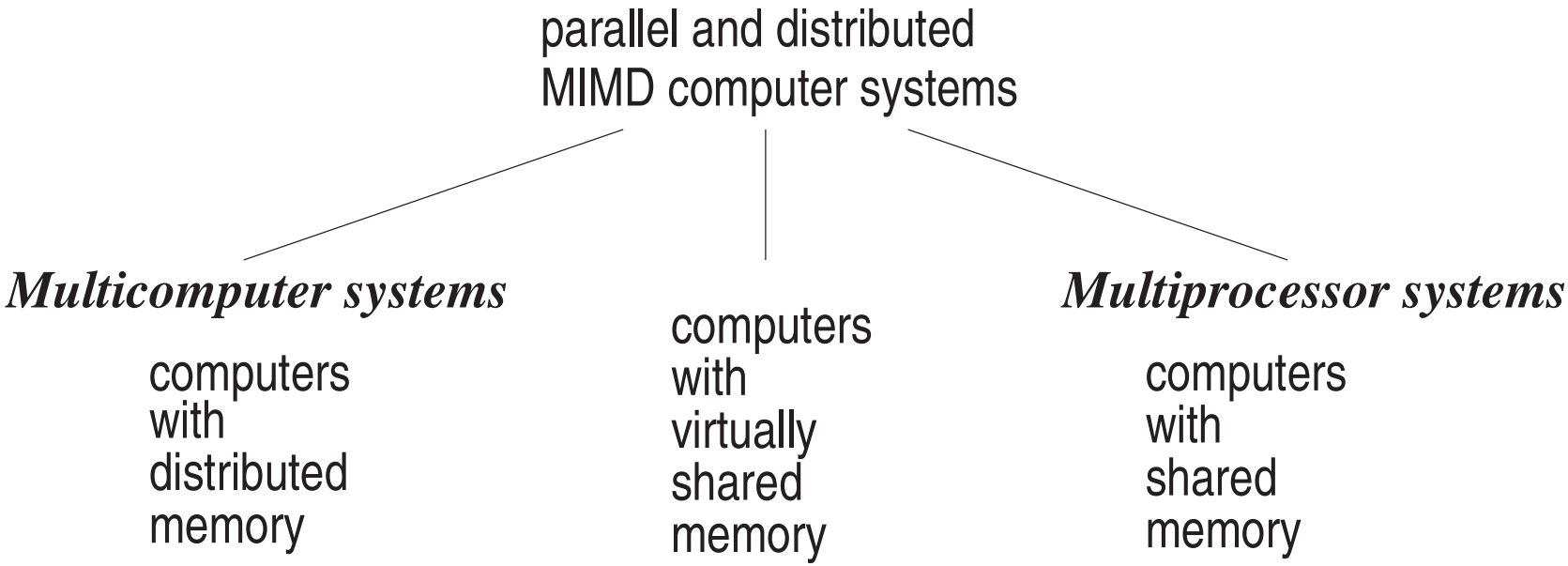
Existe alguma diferença na execução em uma máquina SIMD e outra MIMD?

Organização de Memória de Computadores Paralelos

A maioria dos computadores paralelos de propósito geral são MIMD. Podemos classificá-los de acordo com sua organização de memória:

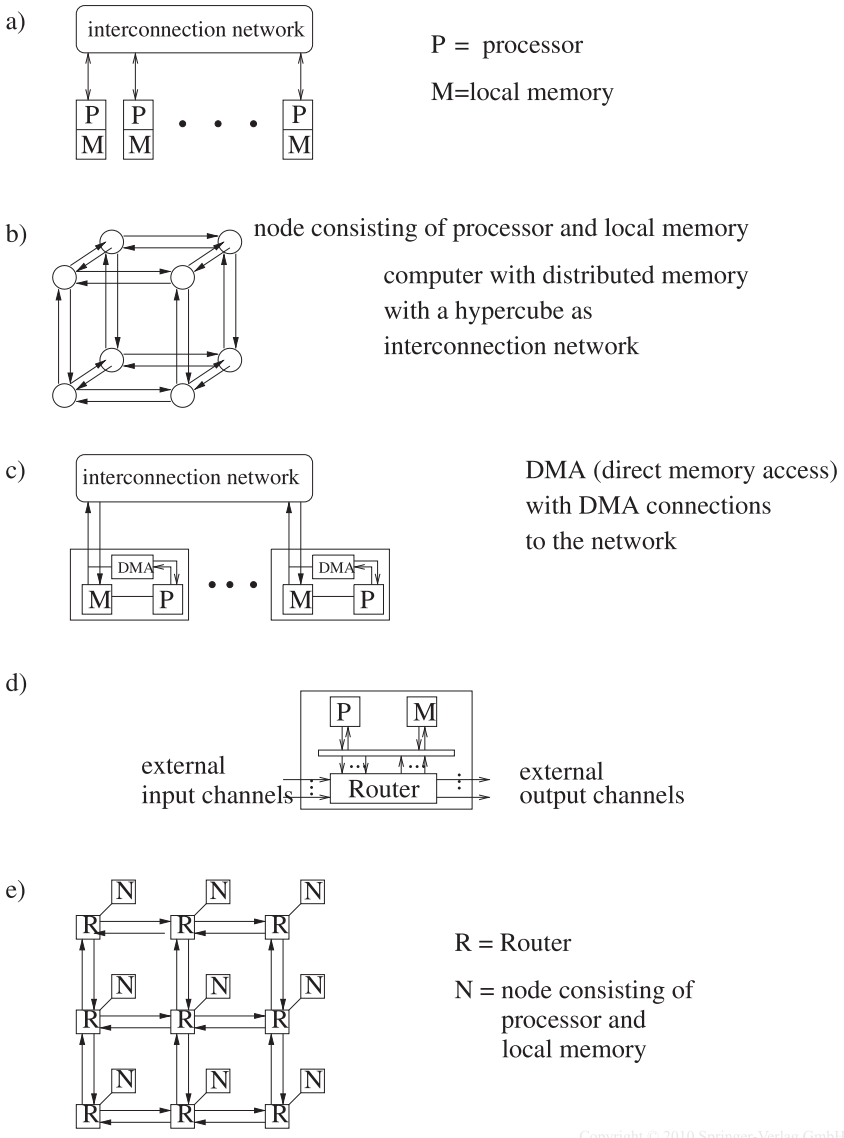
- A organização física dos bancos de memória:
  - Memória compartilhada;
  - memória distribuída.
- a visão que o programador tem da memória:
  - Espaço de endereçamento compartilhado;
  - espaço de endereçamento distribuído.

Organização de Memória de Computadores Paralelos



Computadores com Organização de Memória Distribuída

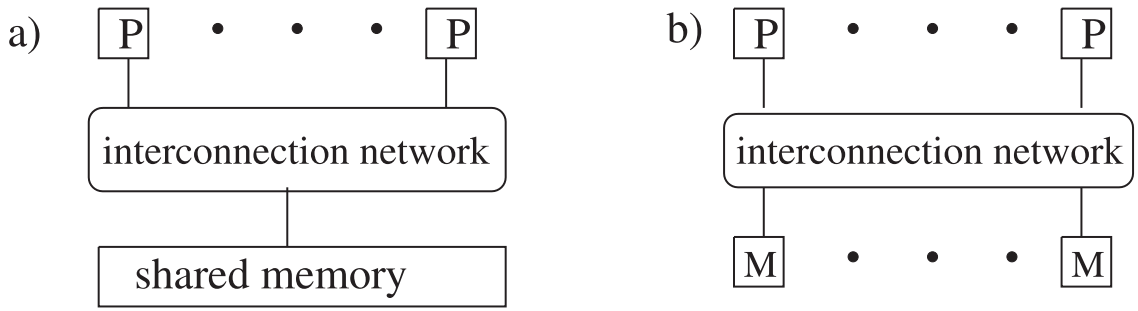
- Elementos de processamentos (máquinas ou **nós**) independentes;
- cada um executa uma imagem, instância de Sistema Operacional;
- cada um possui bancos de memória próprios, processadores, periféricos, etc;
- toda a memória é privada, apenas o processador(es) local tem acesso direto;
- para acessar memória de um outro elemento, um nó precisa recorrer a **troca de mensagem**.



- Conexões ponto-a-ponto;
- controladores DMA;
- roteadores:
  - rotas mais complexas;
  - *buffers*.

Computadores com Organização de Memória Compartilhada

- Memória global compartilhada entre vários processadores em um mesmo elemento;
- dados podem ser compartilhados por variáveis compartilhadas (condições de corrida);
- espaço de endereçamento comum;
- a interconexão é fornecida por um barramento interno;
- (*Symmetric Multiprocessor* - SMP): acesso uniforme a qualquer endereço a partir de qualquer processador;



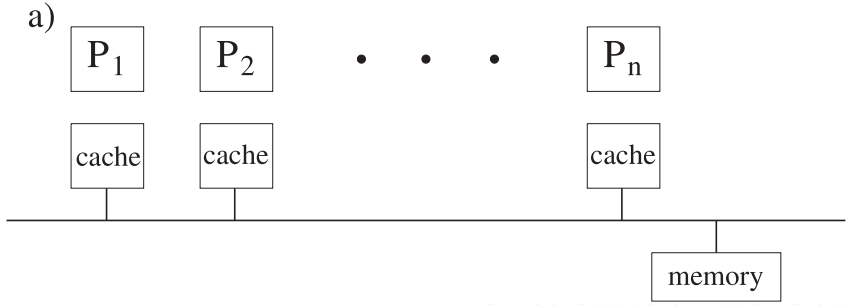
Computadores com Organização de Memória Compartilhada

Non-Uniform Memory Access - NUMA

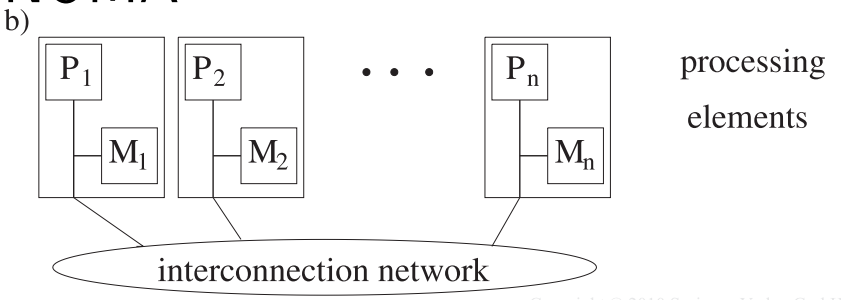
- A memória é fisicamente distribuída, mas o espaço de endereçamento é único;
- processos podem se comunicar por variáveis compartilhadas;
- mas o tempo de acesso não é uniforme;
- acessar a memória mais *próxima* é mais rápido do que acessar a memória de outro processador.

Computadores com Organização de Memória Compartilhada

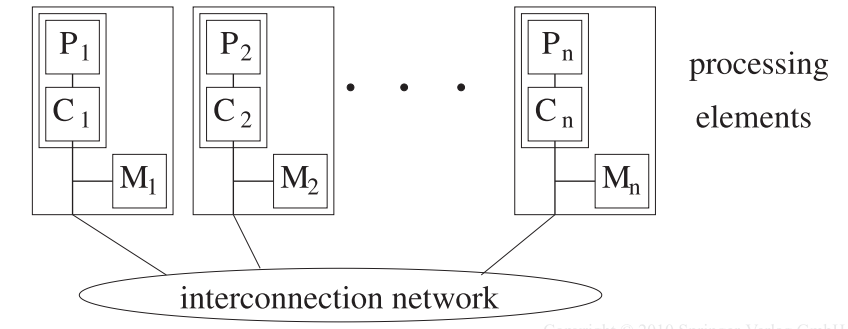
SMP



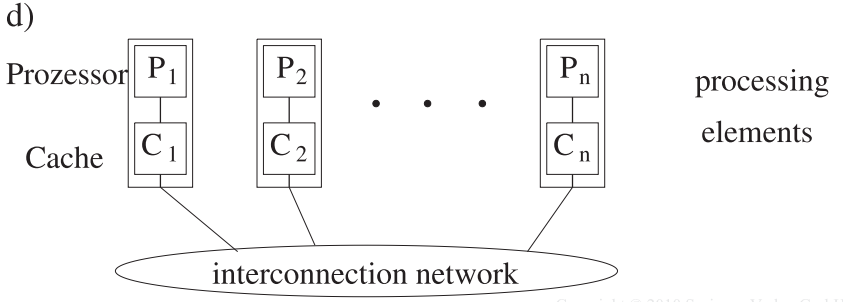
NUMA



CC-NUMA



COMA



Reduzindo Tempo de Acesso a Memória

Tradicionalmente, as CPUs são bem mais rápidas do que os *chips* de memória.

- Reduzir a latência de memória pode *preencher* melhor a CPU com instruções;
- *Multithreading*
  - Duplicar algumas das estruturas internas do processador;
  - permitir mais de um *thread* associado ao processador;
  - enquanto um *thread* executa, os dados do próximo *thread* são carregados da memória.
- *Cache*
  - Memória de alto desempenho contendo um subconjunto da memória principal;
  - associatividade por conjunto;
  - hierarquia de *caches*;
  - coerência de cache.

Paralelismo a Nível de *Thread*

- Já discutimos como o *threading* pode aprimorar o acesso à memória;
- incentivo para desenvolvedores criarem programas com paralelismo explícito;
- será que existe algum suporte na arquitetura para otimizar ainda mais a execução de *threads*?
- *simultaneous multithreading* - SMT.

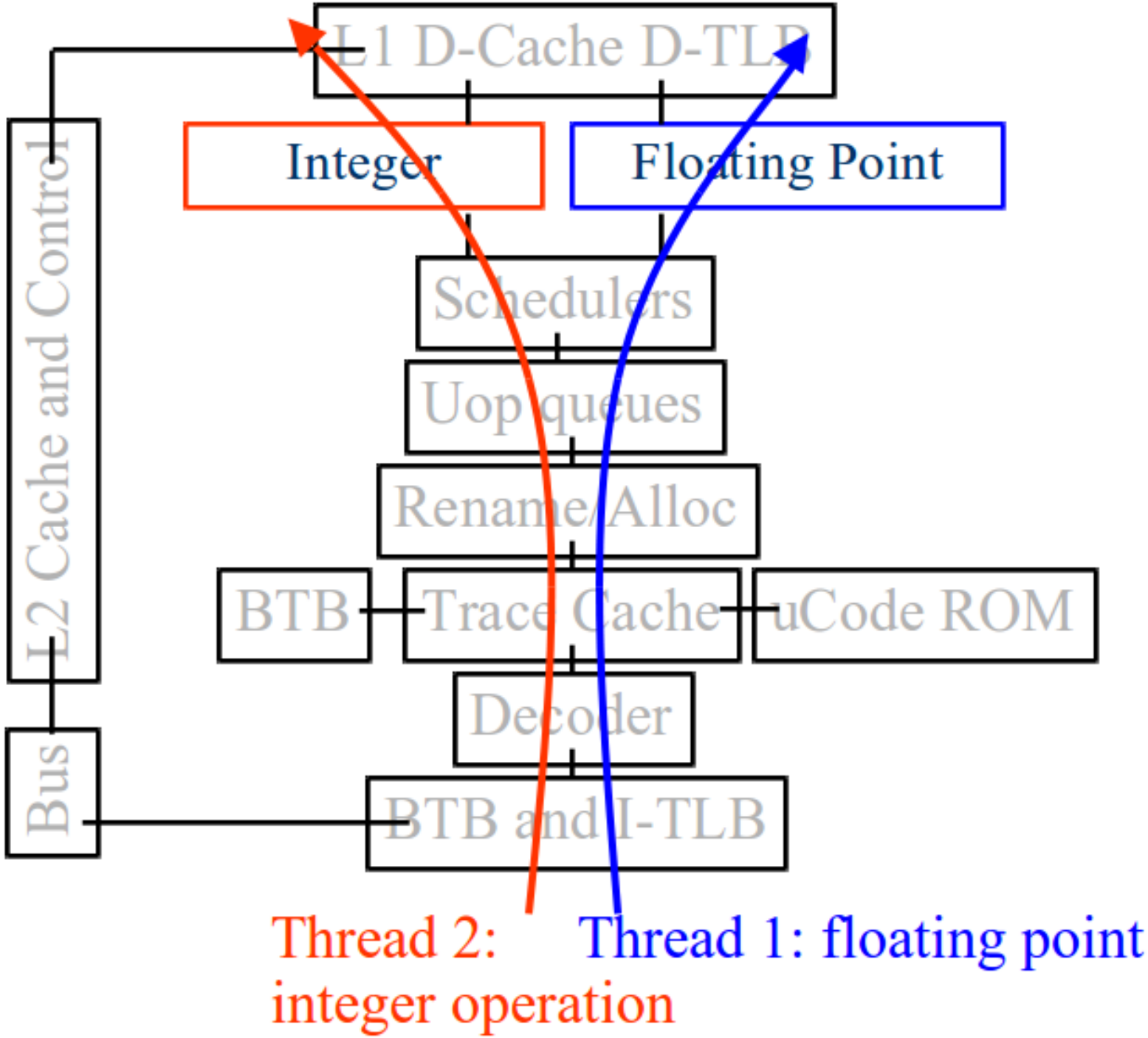
Também conhecido como *HyperThreading* nos *chips* Intel. Não se trata de um processador *multicore*, mas um único *core* com melhorias. Considerem um meio termo.

Simultaneous Multithreading - SMT

Existe um único fluxo de instruções, mas as instruções são originadas de vários *threads*.

- Replicação de áreas do *chip*
  - contador de programa;
  - registradores de controle;
  - controlador de interrupção.
- cria a ilusão de *processador lógico* ao sistema operacional.
- *caches*, barramentos, unidades de controle e função (ULA) são compartilhadas entre os processadores lógicos.

Quando um *thread* em um processador lógico é bloqueado, outro processador lógico pode executar, minimizando o tempo de troca de contexto.

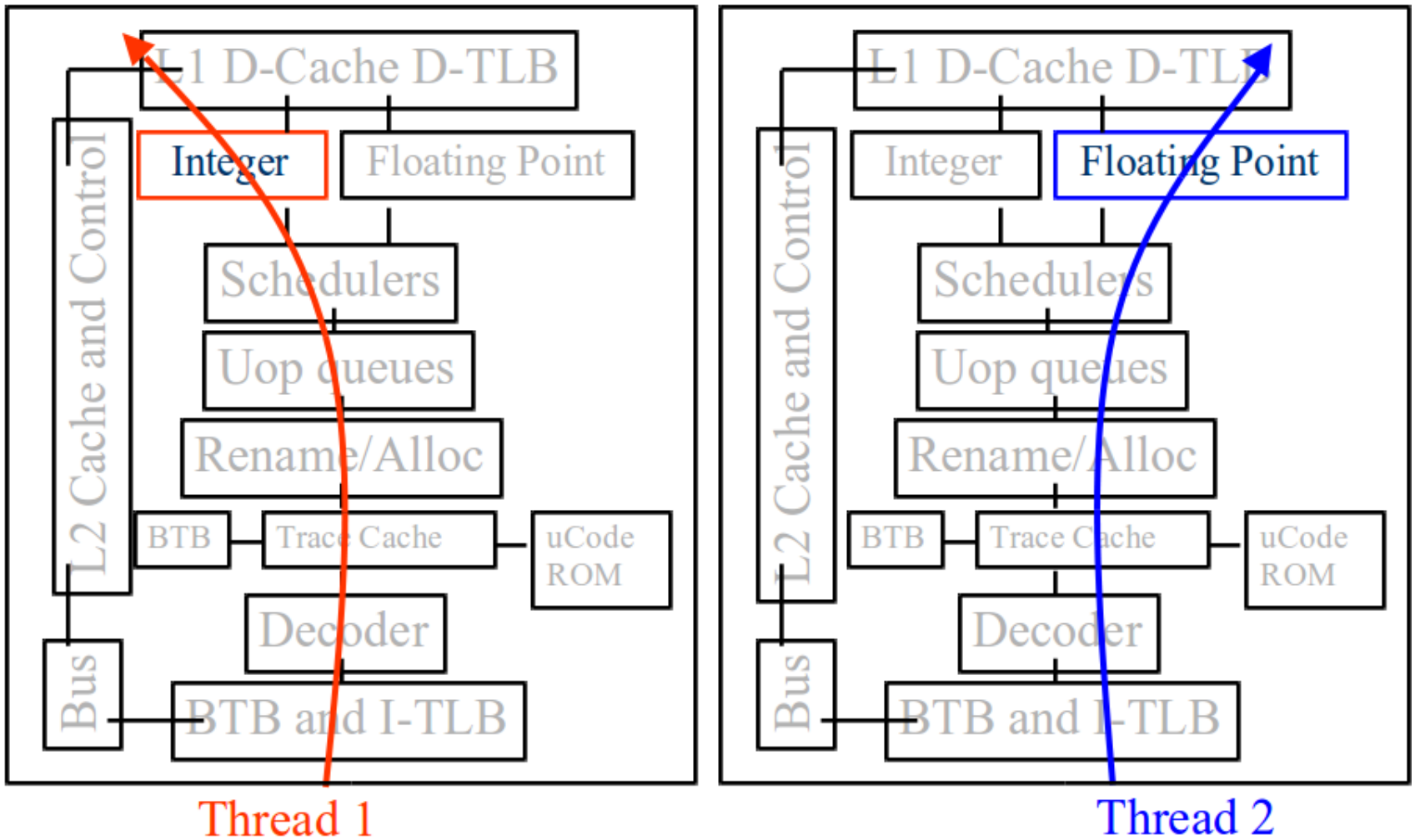

L1 D-Cache D-TLB
Integer
Floating Point
L2 Cache and Control
Schedulers
Uop queues
Rename/Alloc
BTB
Trace Cache
uCode ROM
Decoder
Bus
BTB and I-TLB
Thread 2: integer operation
Thread 1: floating point

Processadores *Multicore*

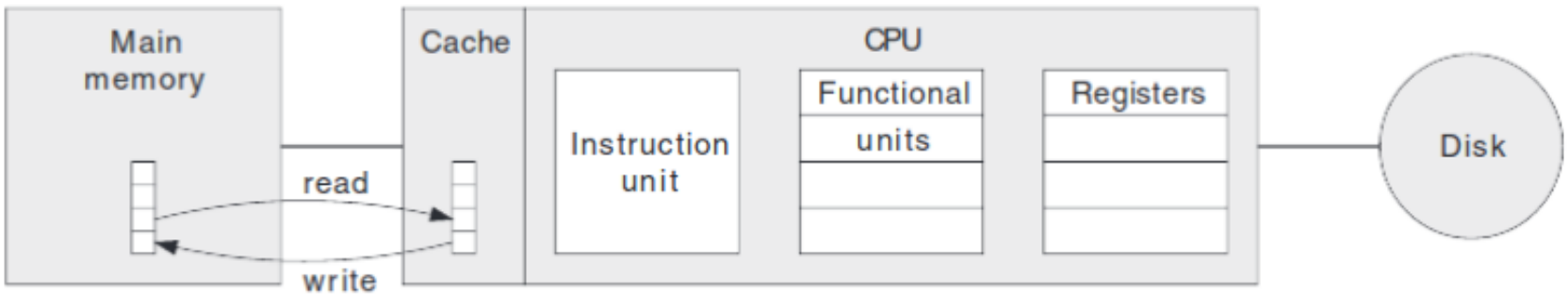
Limites para a melhoria do processador clássico *monocore*:

- Lei de Moore;
- um único fluxo de instruções é incapaz de manter um grande número de unidades funcionais ocupadas;
- a memória não evolui na mesma velocidade;
- limites físicos de aquecimento e consumo de energia.

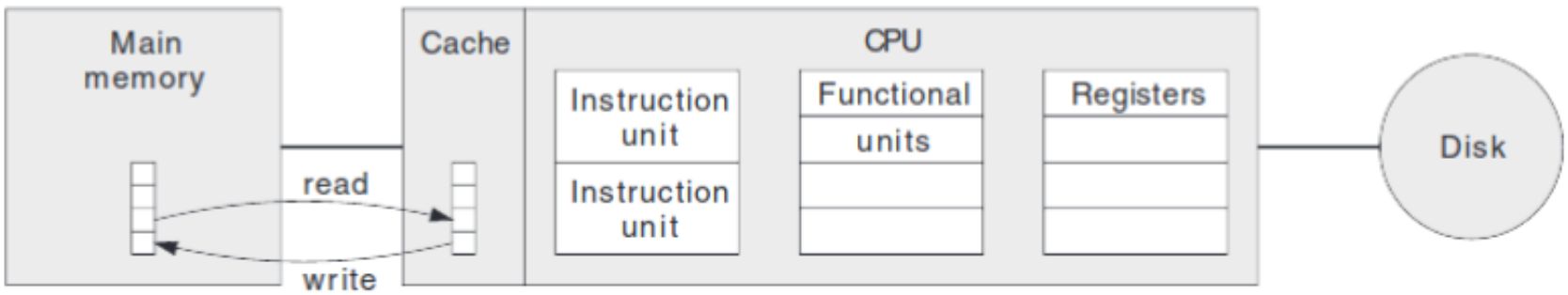
Processadores *multicore* (*Symmetric Multiprocessing* - SMP) replicam o núcleo de processamento por completo.


L1 D-Cache D-TLB
Integer
Floating Point
Schedulers
Uop queues
Rename/Alloc
BTB
Trace Cache
uCode ROM
Decoder
BTB and I-TLB
L2 Cache and Control
Bus
Thread 1
L1 D-Cache D-TLB
Integer
Floating Point
Schedulers
Uop queues
Rename/Alloc
BTB
Trace Cache
uCode ROM
Decoder
BTB and I-TLB
L2 Cache and Control
Bus
Thread 2

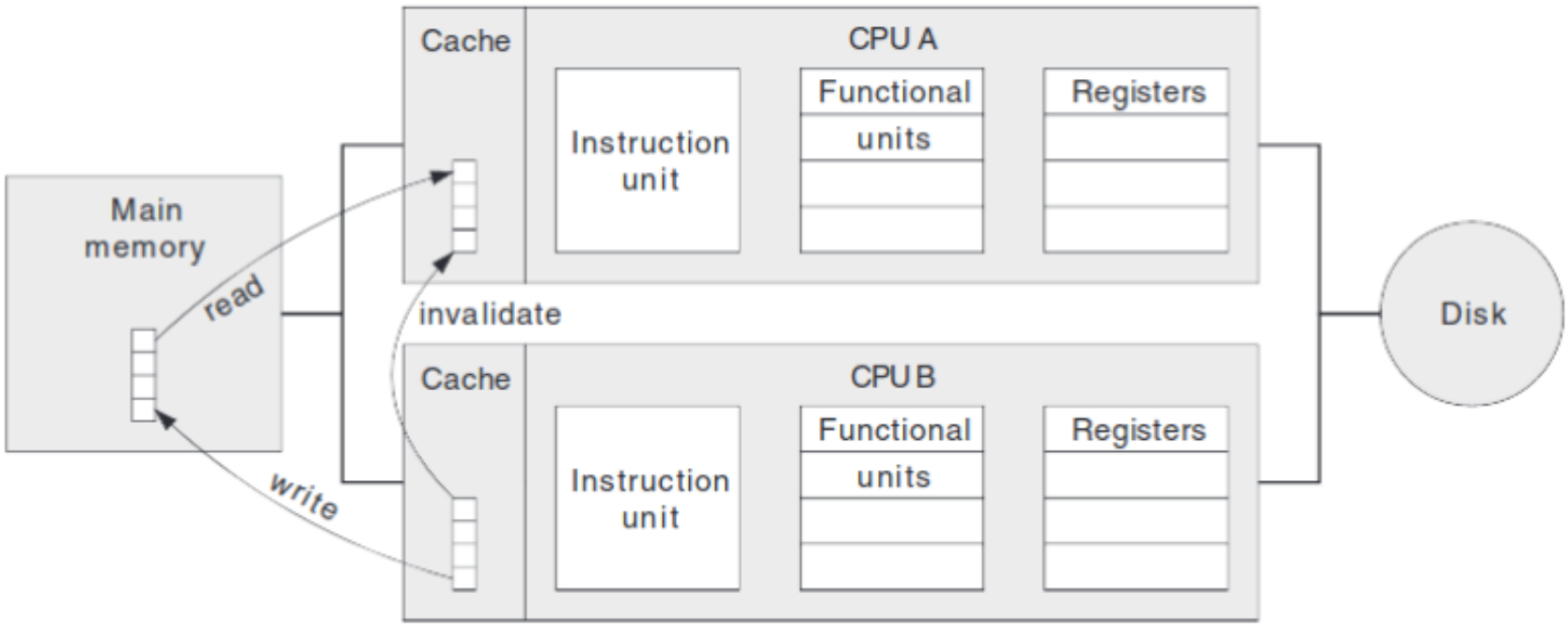
SMT vs. SMP


Main
memory
Cache
CPU
Instruction
unit
Functional
units
Registers
Disk
read
write

SMT vs. SMP


Main memory
Cache
CPU
Instruction unit
Instruction unit
Functional units
Registers
Disk
read
write

SMT vs. SMP

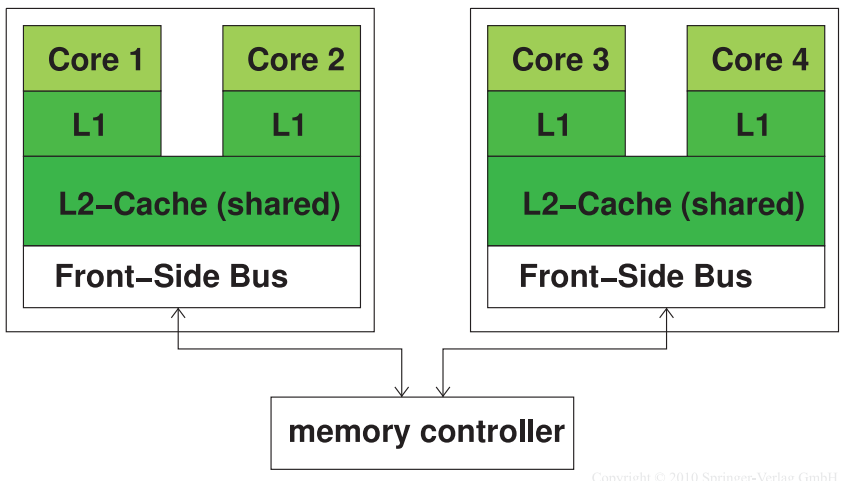

Cache
CPU A
Instruction unit
Functional units
Registers
Main memory
read
invalidate
Cache
CPU B
Instruction unit
Functional units
Registers
write
Disk

Arquitetura de Processadores *Multicore*

- Os processadores modernos utilizam tanto SMT quanto SMP;
- variam em tamanho de *cache*, como os núcleos acessam a *cache* e a presença de unidades funcionais especializadas;
- três tipos básicos de arquitetura podem ser enumerados:
  - Projeto hierárquico;
  - projeto em *pipelines*;
  - projeto em rede.
- na prática, os três tipos de projetos podem estar presentes em um processador híbrido.

Projeto Hierárquico

- Vários *cores* compartilham várias *caches*;
- estrutura em árvore;
- a raiz é a memória externa, principal;
- cada *core* tem uma memória *cache* individual;
- entre a memória principal e a *cache*, pode existir diversas *caches* intermediárias compartilhadas.

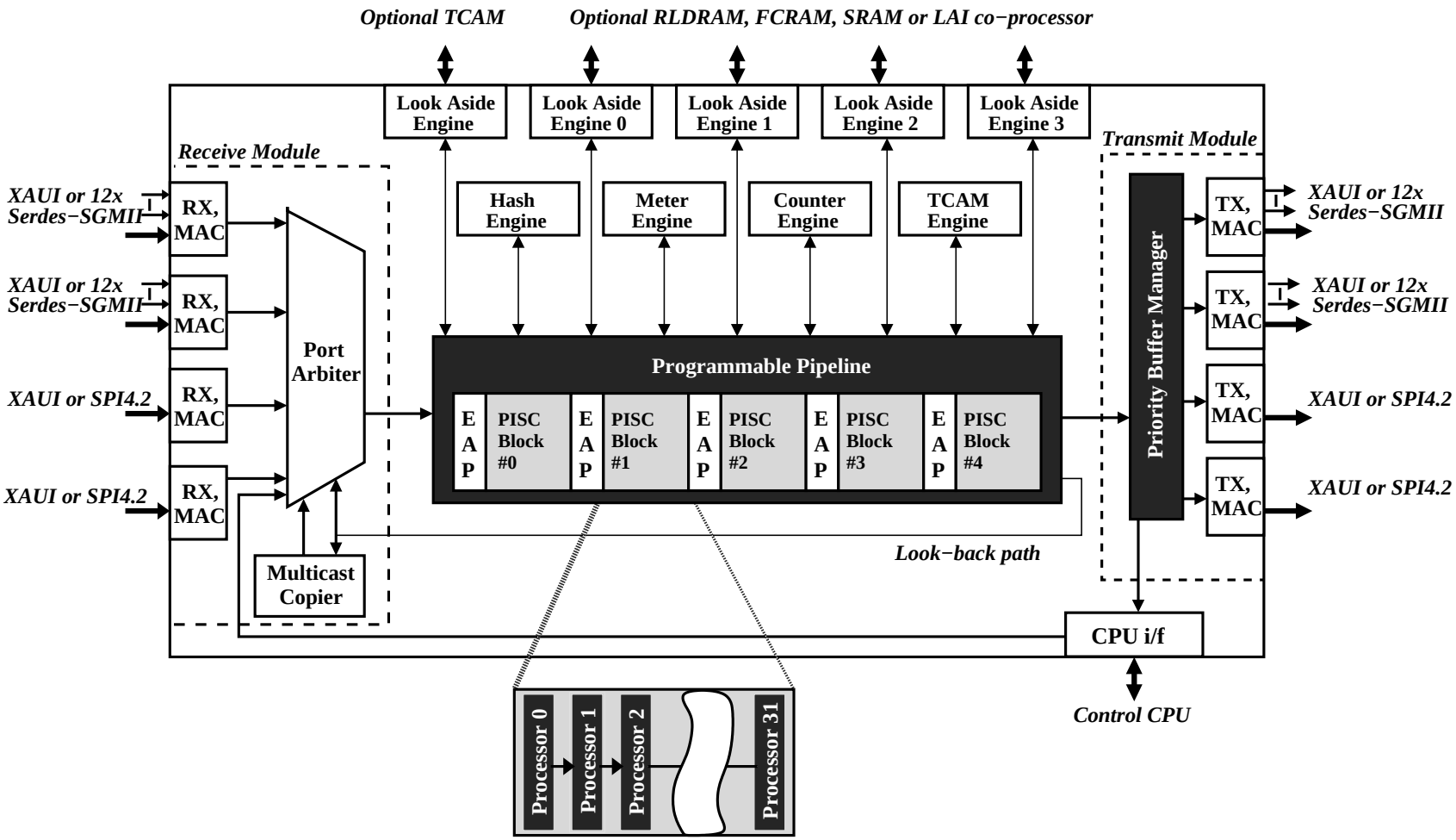


Projeto em *Pipelines*

- Os dados entram no processador por uma porta específica;
- são tratados sucessivamente por diferentes núcleos;
- cara núcleo faz um processamento específico;
- o resultado da computação é escrito em uma porta de saída.

Esse projeto é usado em processadores de roteadores ou processamento gráfico.

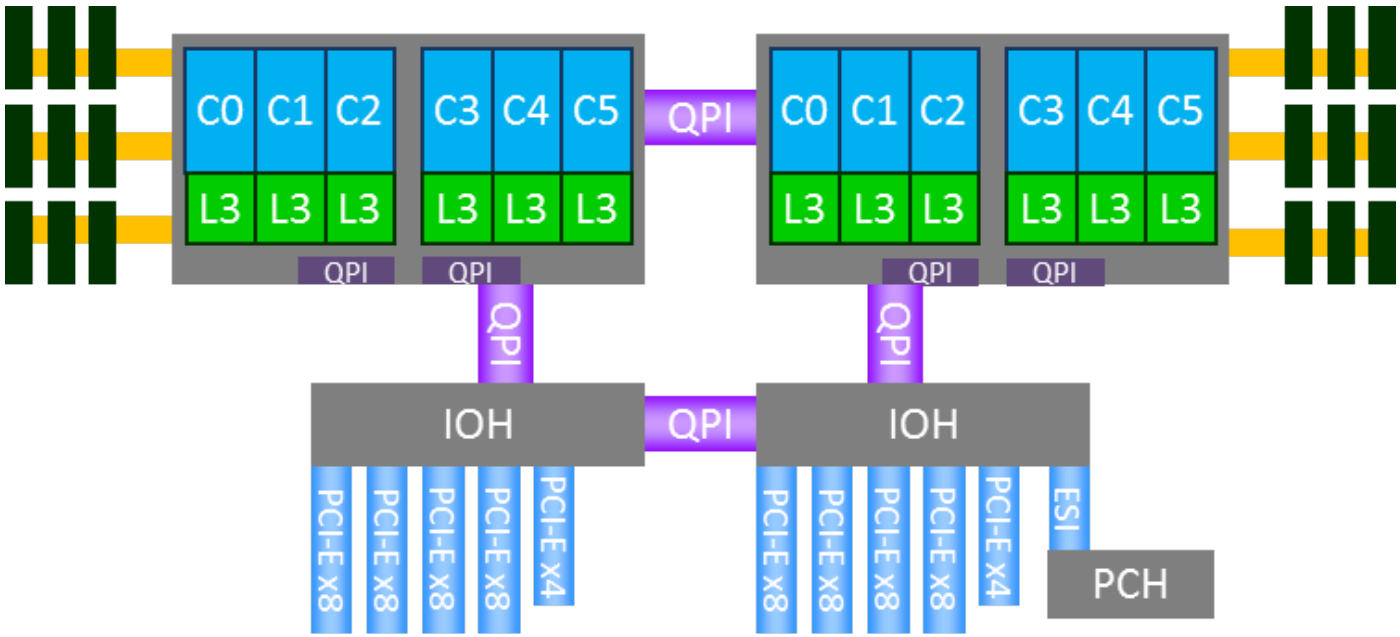
Projeto em *Pipelines*



Projeto em Rede

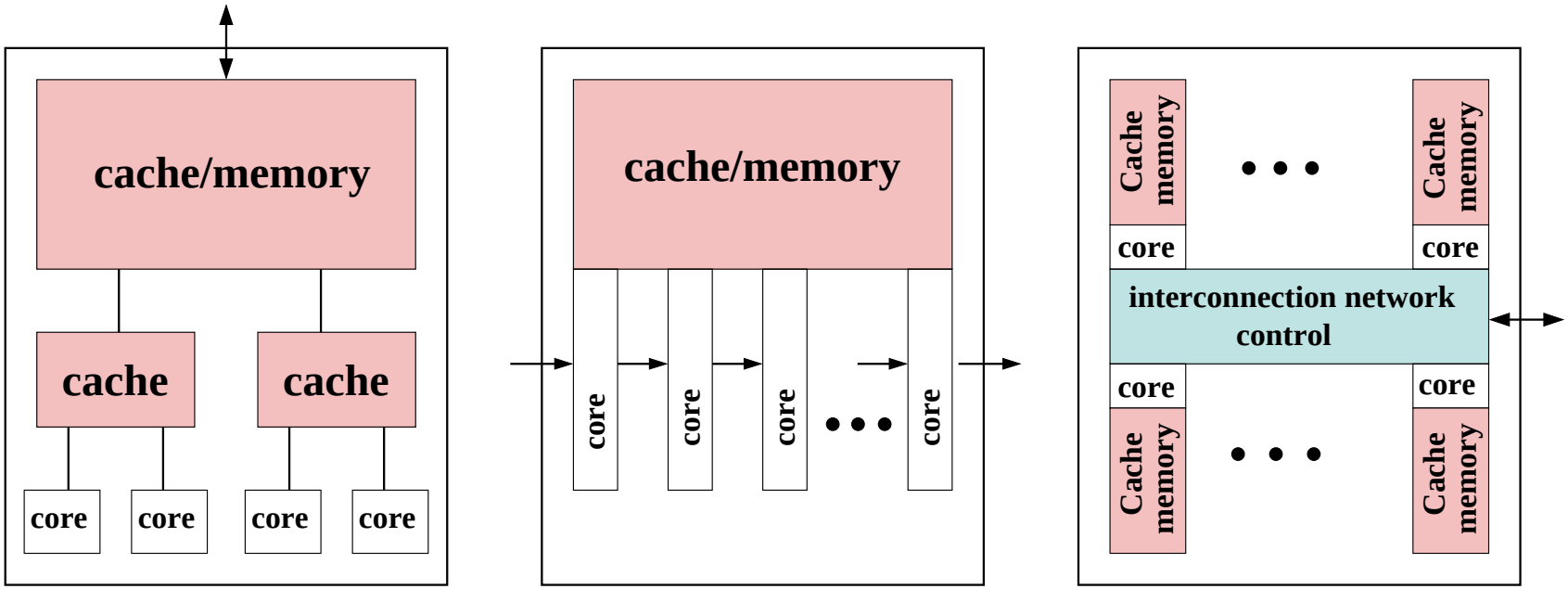
- Os *cores* e suas as *caches* são ligados a outros *cores* por uma rede de interconexão;
- além de transferência de dados, a rede pode ajudar na sincronização dos *cores*;
- projeto similiar a organização de memória em sistemas NUMA.

Projeto em Rede

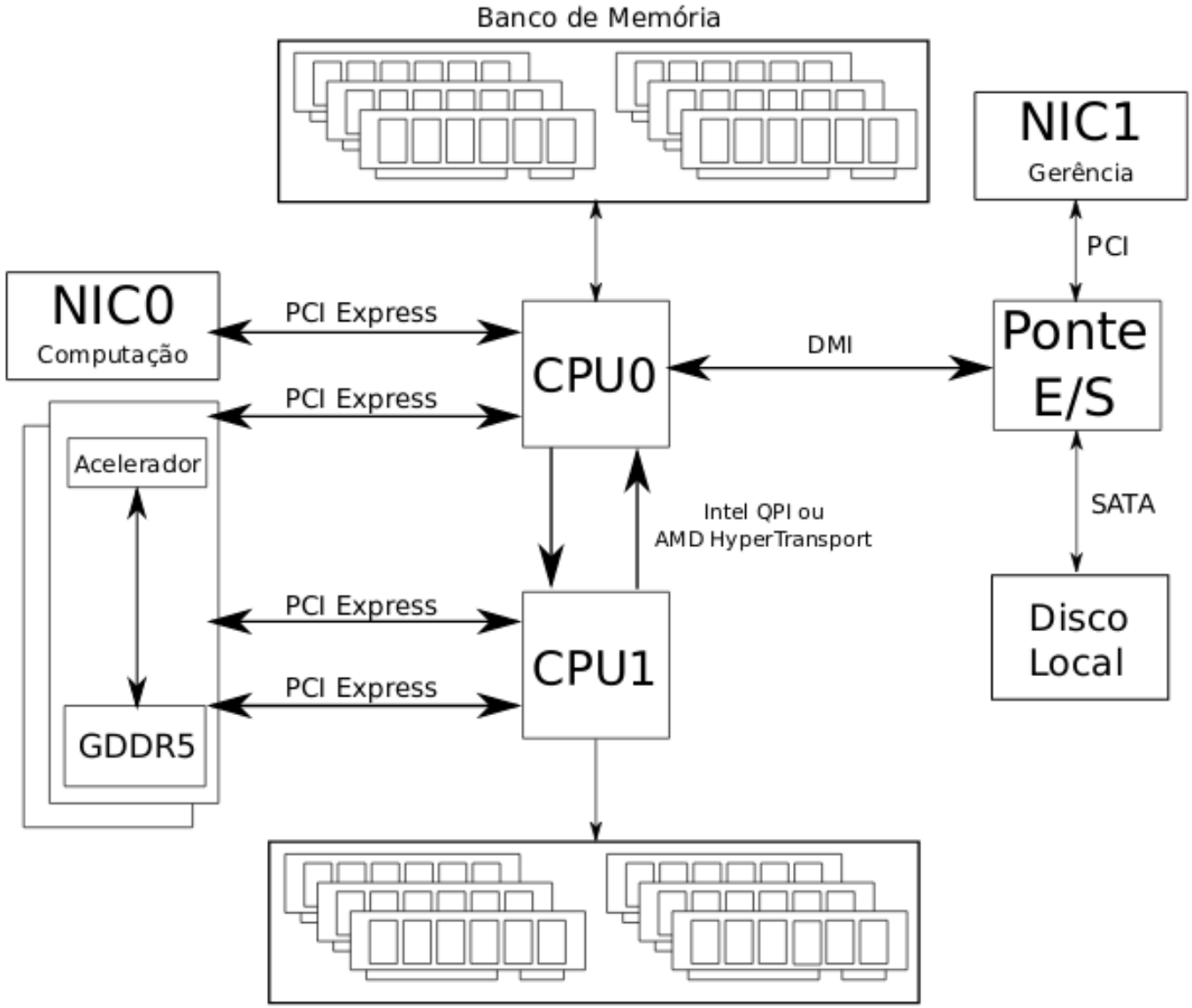


http://www.qdpma.com/SystemArchitecture/SystemArchitecture_QPI.html

Projetos de Processadores



Visão Geral de um Processador de Alto Desempenho Moderno



Redes de Interconexão

Já vimos a importância das redes de interconexão para conectar processadores ou *cores* na hierarquia de memória e na arquitetura dos processadores.

- *Links* e *switches*;
- sistemas com vários computadores:
  - A rede conecta computadores ou processadores;
  - mensagens para coordenação, sincronização ou troca de dados.
- sistemas multiprocessados:
  - A rede conecta processadores a módulos de memória;
  - o acesso a endereços de memória é através da interconexão.

Transmitir uma mensagem de um processador a um destino. A mensagem pode conter dados ou uma requisição de memória. O destino pode ser outro processador ou memória.

Redes de Interconexão

- Topologia:
  - descreve a estrutura da rede;
  - grafos;
  - estática ou dinâmica;
- técnica de roteamento:
  - caminho da mensagem do remetente ao destinatário;
  - algortimo de roteamento;
  - estratégia de *switching*.

O desempenho da rede é resultado direto do algoritmo de roteamento, estratégia de *switching* e topologia da rede.

Propriedades das Redes de Interconexão

Considerando redes estáticas:

$$\text{Grafo } G = (V, E)$$

- Os vértices em V são *switches*, processadores, núcleos ou bancos de memória;
- as arestas em E representam uma conexão entre os vértices;
- se há uma conexão entre $u \in V$ e $v \in V$, então $(u, v) \in E$.

Caminho entre u e v

Sequência de vértices $(v_0, ..., v_k)$ de tamanho k na qual cada par $(v_i, v_{i+1}) \in E$ para todo $0 \leq i < k$.

Propriedades das Redes de Interconexão

O diamêtro $\delta(G)$ de uma rede G é definido como a maior distância entre dois pares de vértices:

$$\delta(G) = \max_{u,v \in E} \min_{\substack{\text{caminho } \varphi \\ \text{de } u \text{ para } v}} \{k | k \text{ é o tamanho do caminho } \varphi \text{ de } u \text{ para } v\}$$

O grau $g(G)$ de uma rede G é o grau máximo de um vértice da rede, na qual o grau de um vértice n é o número de vizinhos diretos de n:

$$g(G) = max\{g(v) | g(v) \text{ grau de } v \in V\}$$

Propriedades das Redes de Interconexão

A largura de banda de bissecção $B(G)$ de uma rede G é o número mínimo de arestas que devem ser removidas para particionar a rede em duas partes de tamanho igual, sem nenhuma conexão entre elas. Para uma quantidade ímpar de vértices, o tamanho das partes deve diferir no máximo em 1.

$$B(G) = \min_{\substack{U_1, U_2 \text{ partições de } V \\ ||U_1| - |U_2|| \leq 1}} |\{(u, v) \in E | u \in U_1, v \in U_2\}|$$

Considere $|A|$ como o número de elementos de A. O que acontece com uma rede quando existem $B(G) + 1$ mensagens?

Propriedades das Redes de Interconexão

Quantos elementos devo retirar para obter duas partes da rede, de quaisquer tamanho, sem ligação?

Conectividade de Vértices

Quantos vértices devem falhar para para desconectar a rede?

Conectividade de Arestas

Quantas arestas devem falhar para para desconectar a rede?

Considere $G_{V \setminus M}$ o grafo restante da remoção de todos os vértices $M \subset V$, assim como as arestas adjacentes.

$$G_{V \setminus M} = (V \setminus M, E \cap ((V \setminus M) \times (V \setminus M)))$$

Propriedades das Redes de Interconexão

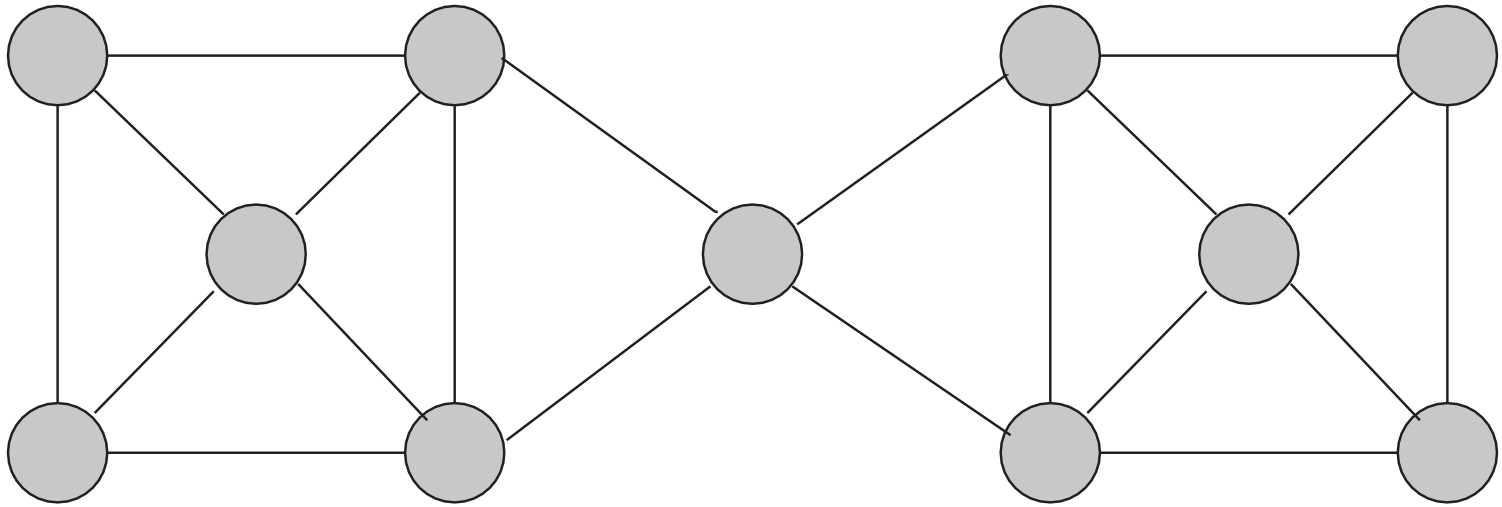
Conectividade de Vértices

$$nc(G) = \min_{M \subset V} \{|M| | \text{existem } u, v \in V \backslash M, \text{ tal que não existem caminhos em } G_{V \backslash M} \text{ de } u \text{ para } v\}$$

Conectividade de Arestas

$$ec(G) = \min_{F \subset E} \{|F| | \text{existem } u, v \in V, \text{ tal que não existem caminhos em } G_{E \backslash F} \text{ de } u \text{ para } v\}$$

Propriedades das Redes Redes de Interconexão



- Conectividade de vértices?
- conectividade de arestas?
- grau?

Embarcando Redes

Considere duas redes $G = (V, E)$ e $G' = (V', E')$.

- Mapear cada vértice de G' em um vértice de G;
- vértices diferentes de G' são mapeados em vértices diferentes de G;
- arestas entre dois vértices em G' também estão presentes nos vértices mapeados de G.

Podemos definir embarcar redes como um mapeamento $\sigma : V' \to V$ de forma que:

1. Se $u \neq v$ para $u, v \in V'$, então $\sigma(u) \neq \sigma(v)$;
2. se $(u, v) \in E'$, então $(\sigma(u), \sigma(v)) \in E$.

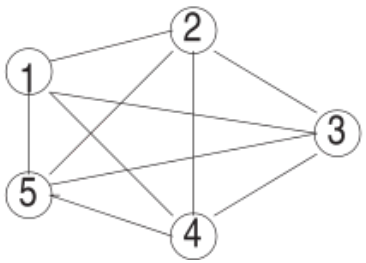
Se uma rede G' pode ser embarcada em G, então G é pelo menos tão flexível quanto G'.

Propriedades Desejáveis em uma Topologia

- Diâmetro pequeno;
- grau pequeno;
- grande largura de bissecção;
- alta conectividade;
- possibilidade de embarcamento em várias redes;
- fácil extensão.

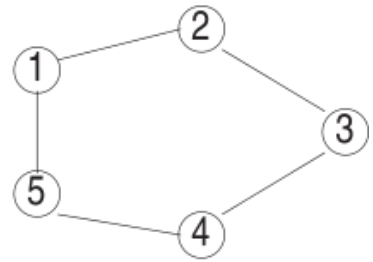
Exemplos de Topologia

Grafo Completo



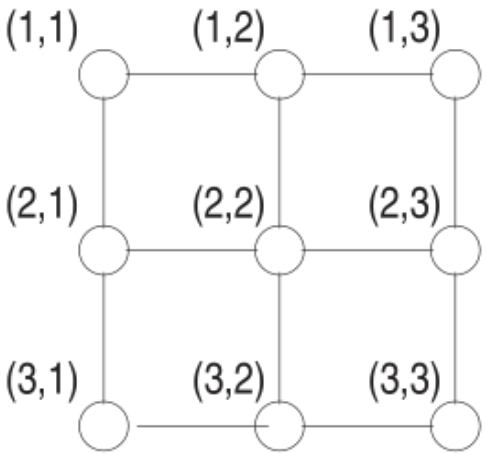
Array Linear

Anel

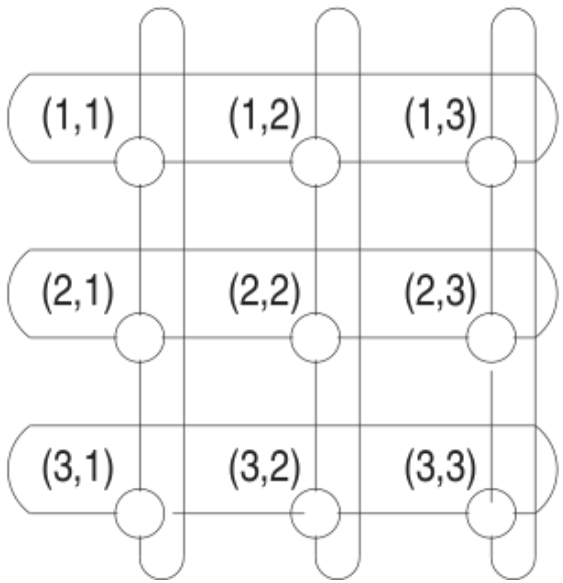


Exemplos de Topologia

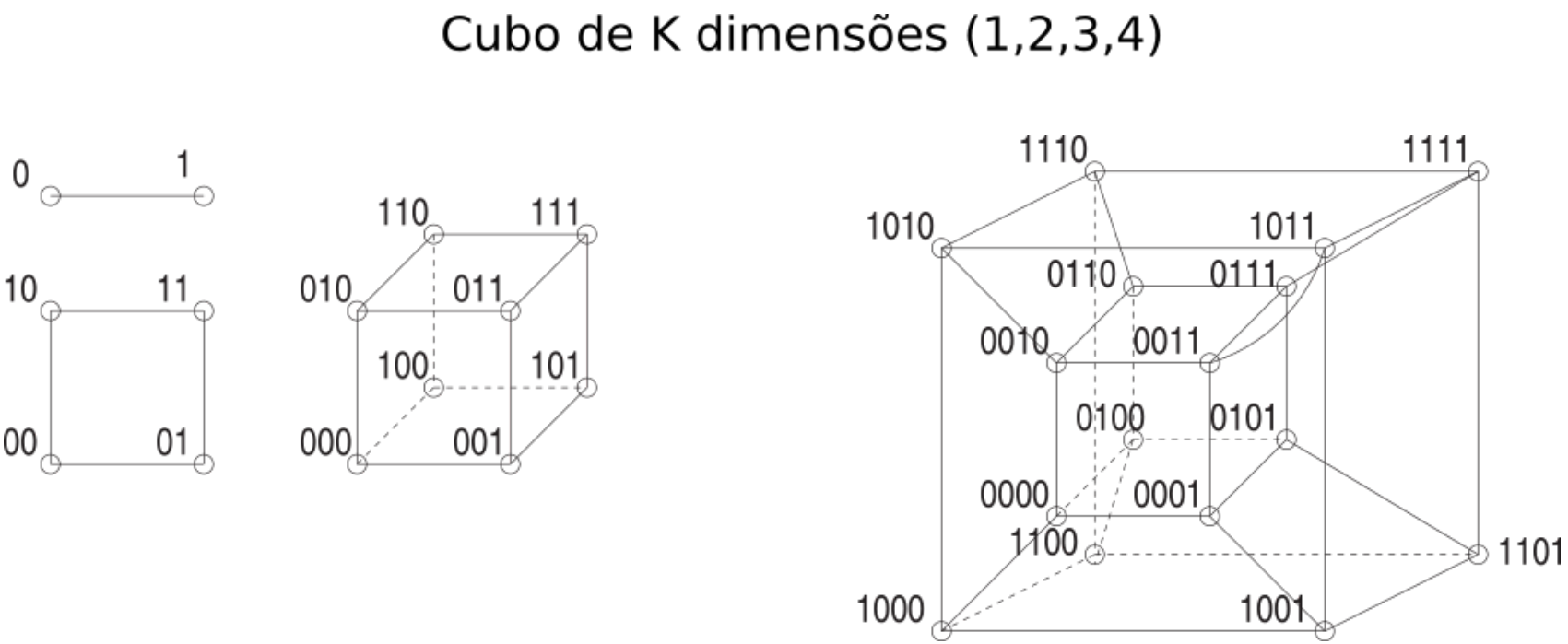
Mesh de 2 dimensões



Torus de 2 dimensões

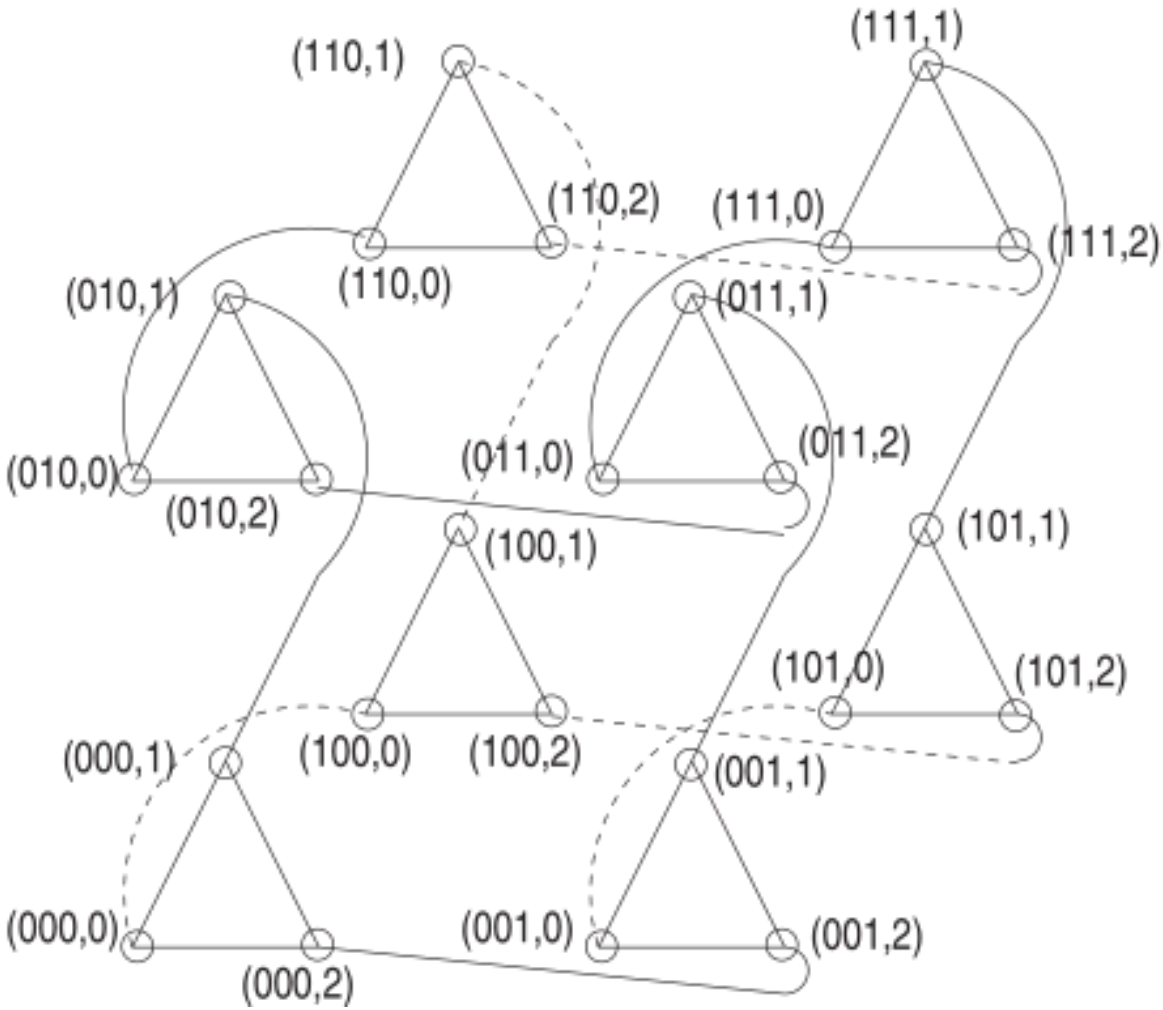


Exemplos de Topologia



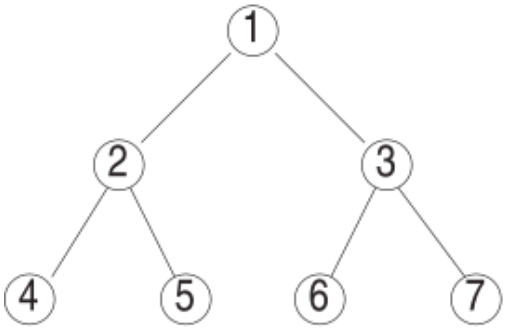
Exemplos de Topologia

Rede de Cubos Conectada por Ciclos (k = 3)

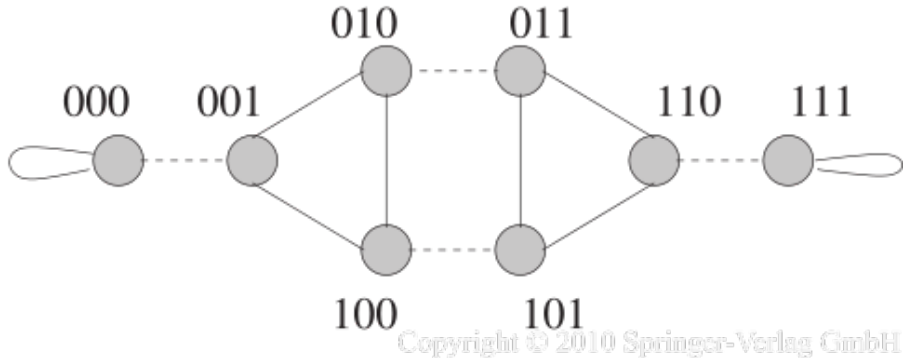


Exemplos de Topologia

Árvore Binária



Rede *Suffle-Exchange*



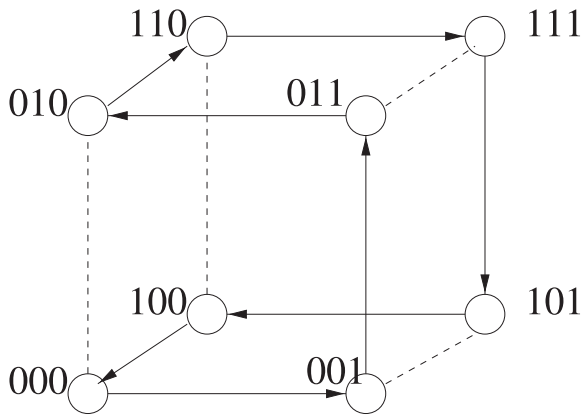
Embarcamento de Redes entre Topologias

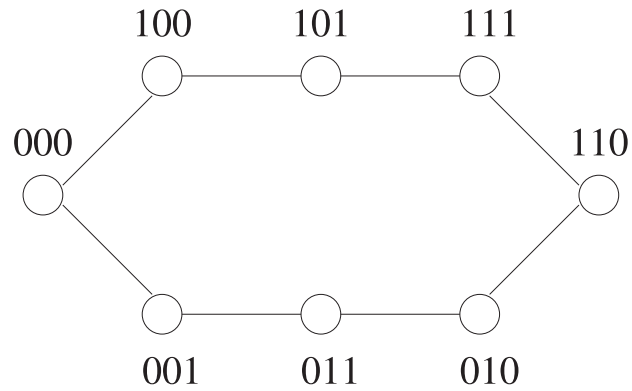
A maioria das topologias podem ser mapeadas para a rede baseada em cubos (*hypercube*) através do código RGC (*Ray Gray Code*):

- RGC de 1 *bit*: $RGC_1 = (0, 1)$;
- RGC de 2 *bits*: adiciona 0 e 1 na frente de cada cadeia em RGC_1, inverte a segunda cadeia e concatena o resultado: $RGC_2 = (00, 01, 11, 10)$;
- Para $k \geq 2$, constrói RGC_{k-1}, anexa 0 ao ínicio de cada uma de suas sequências, anexa 1 ao início de cada uma de suas sequências, inverte o segundo grupo, concatena, temos RGC_k.

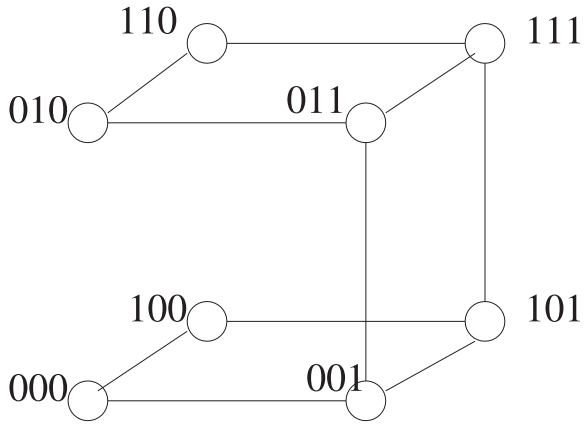
Embarcamento de uma Anel e de um *Mesh*

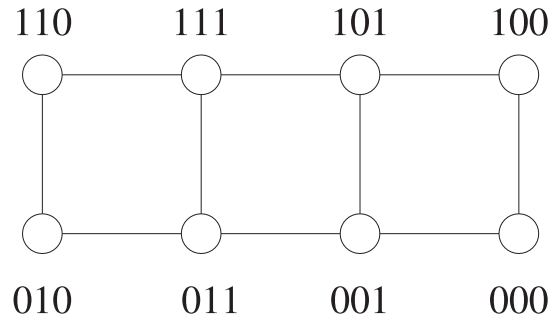
a)





b)

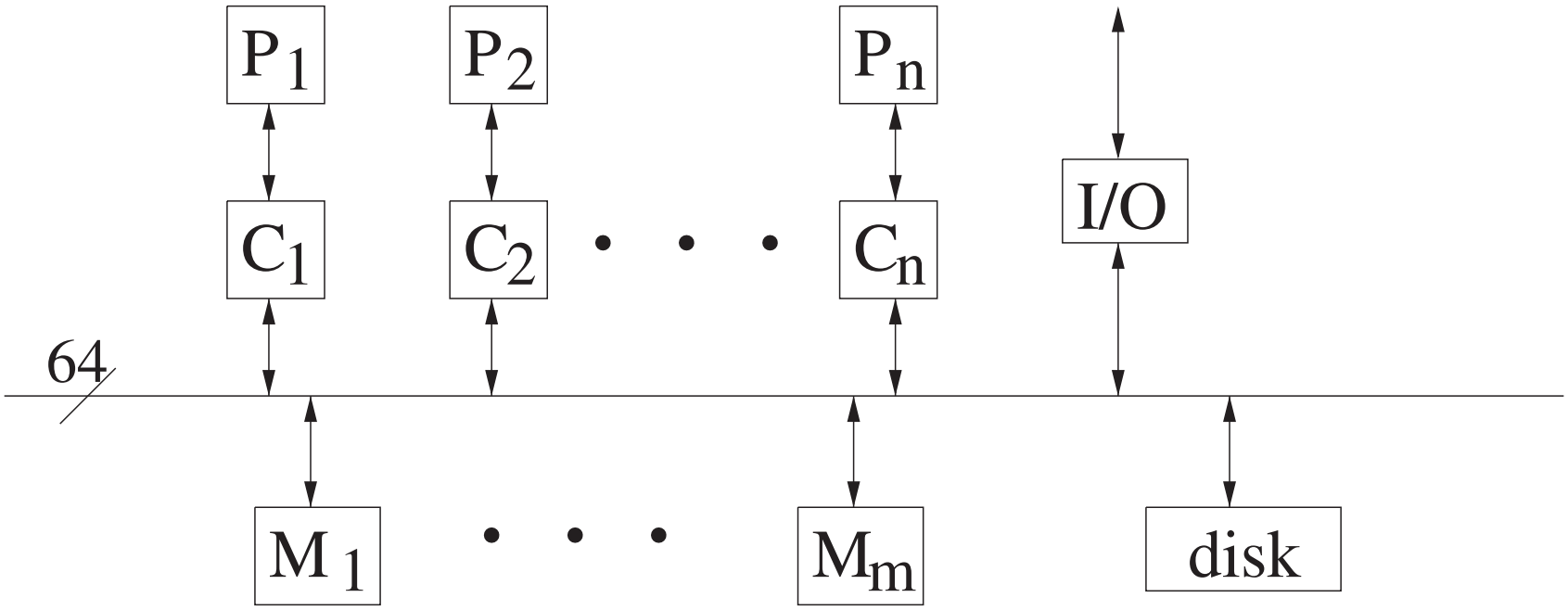




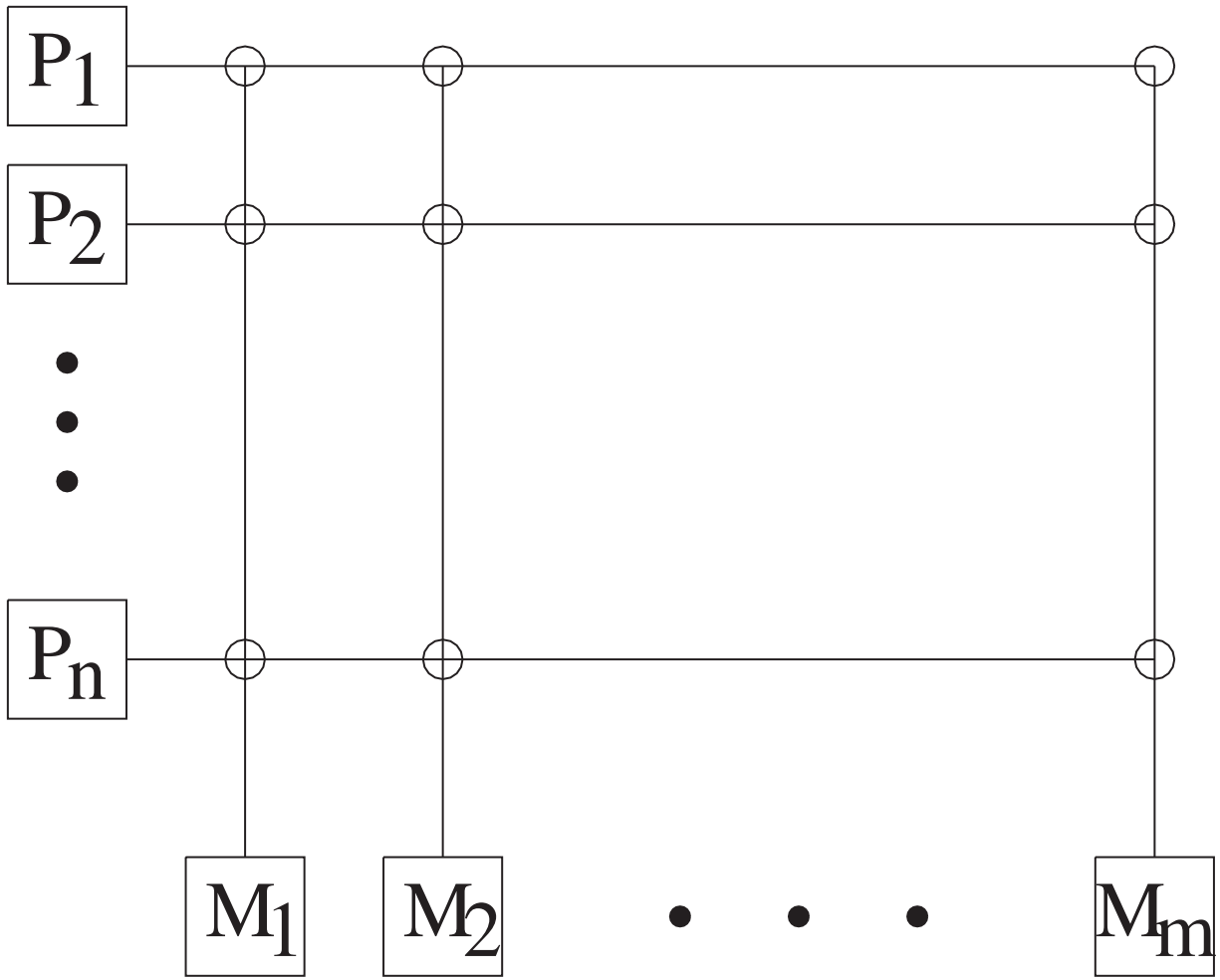
Redes de Interconexão Dinâmica

- Uso de *switches* e roteadores, sem conexxão direta entre processadores ou núcleos;
- os processadores não sabem detalhes da rede: vizinhos, topologia, etc;
- a interconexão é entre os *switches*;
- esses devem ser configurados dinamicamente, ou seja, durante a execução da máquina.

Redes Barramento

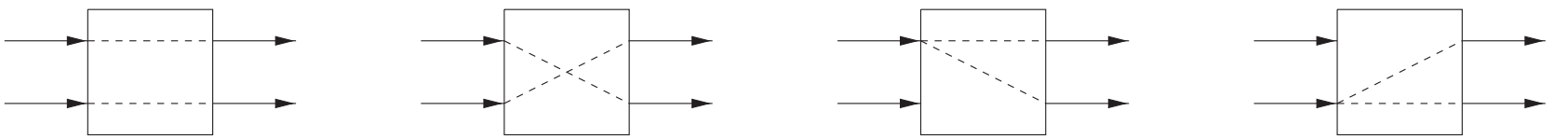


Redes *Crossbar*

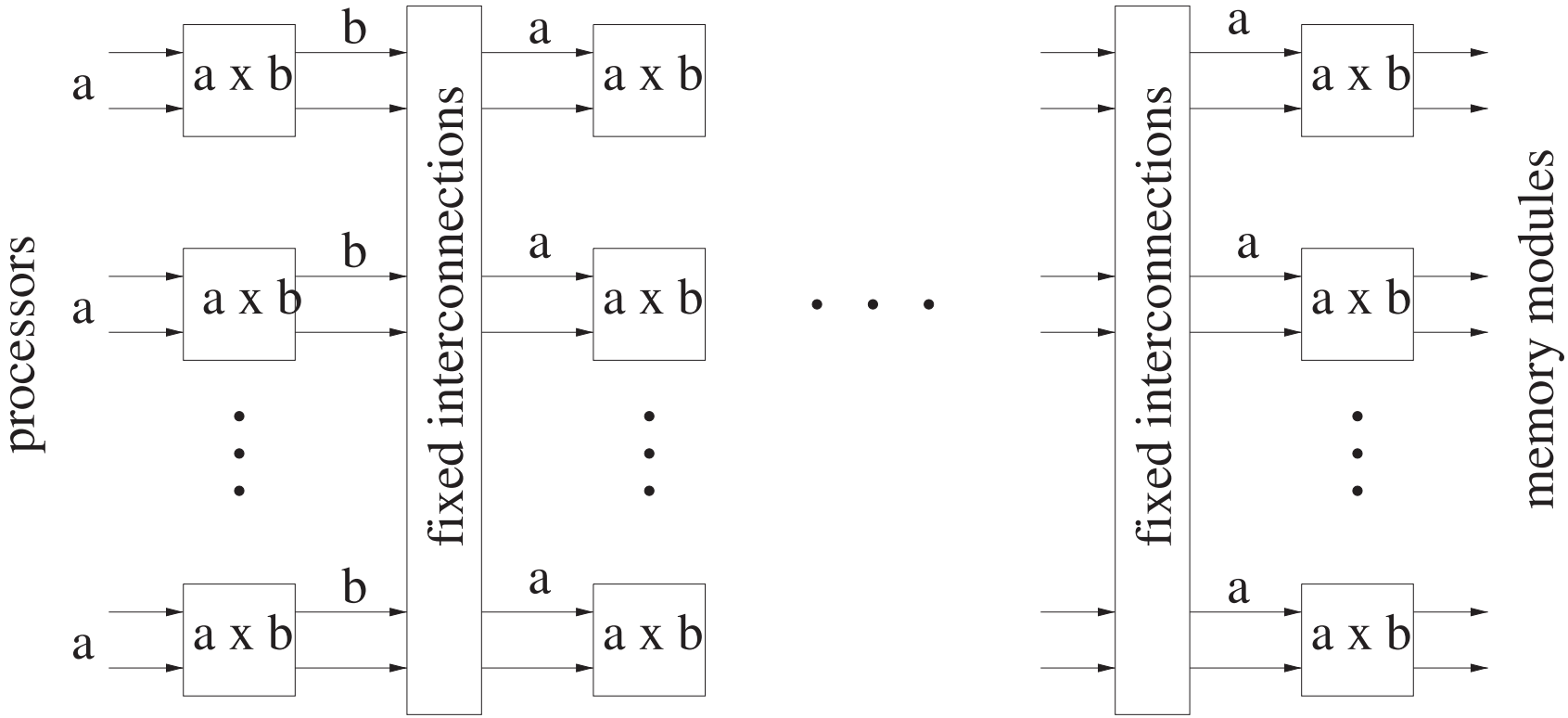


Redes de *Switches* Multi-Estágio

- Várias etapas;
- a primeira e última etapas são processadores, núcleos, memórias, etc;
- as etapas intermediárias são *switches*;
- novamente, as etapas intermediárias podem ser vistas como um grafo, com *switches* como vértices e *links* como arestas;
- a rede é dita regular quanto cada *switch* intermediário tem o mesmo número de entradas e saídas.



Redes de *Switches* Multi-Estágio

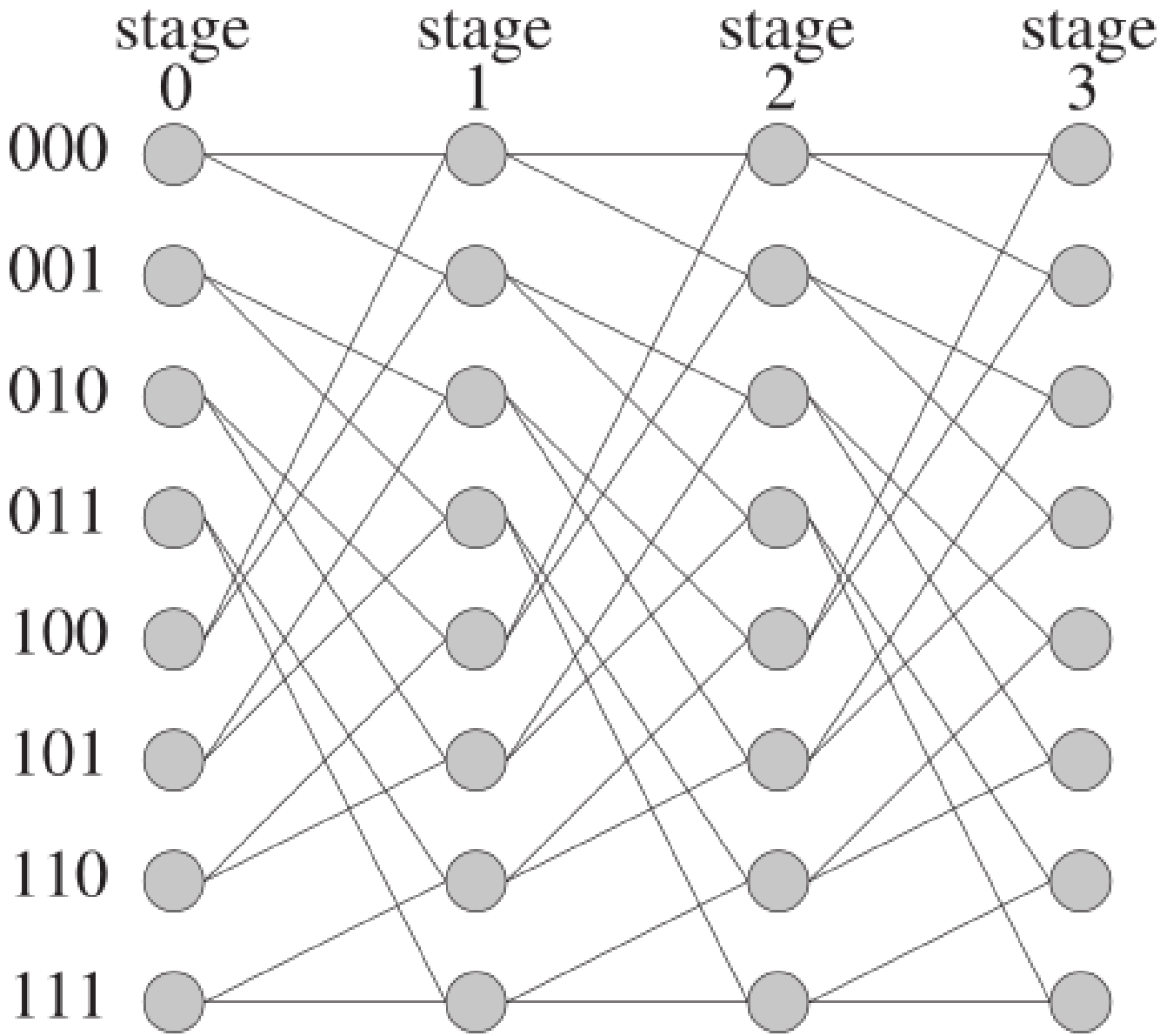


Rede Omega

Uma rede omega tem $n \times n$ elementos organizandos em $log(n)$ estágios. Dados dois vértices (α, i) e $(\beta, i+1)$, α e β binários do número do vértice e i sendo o número do estágio:

- β é uma rotação à esquerda de α ou
- β é uma rotação à esquerda de α com o último *bit* invertido.

Rede Omega

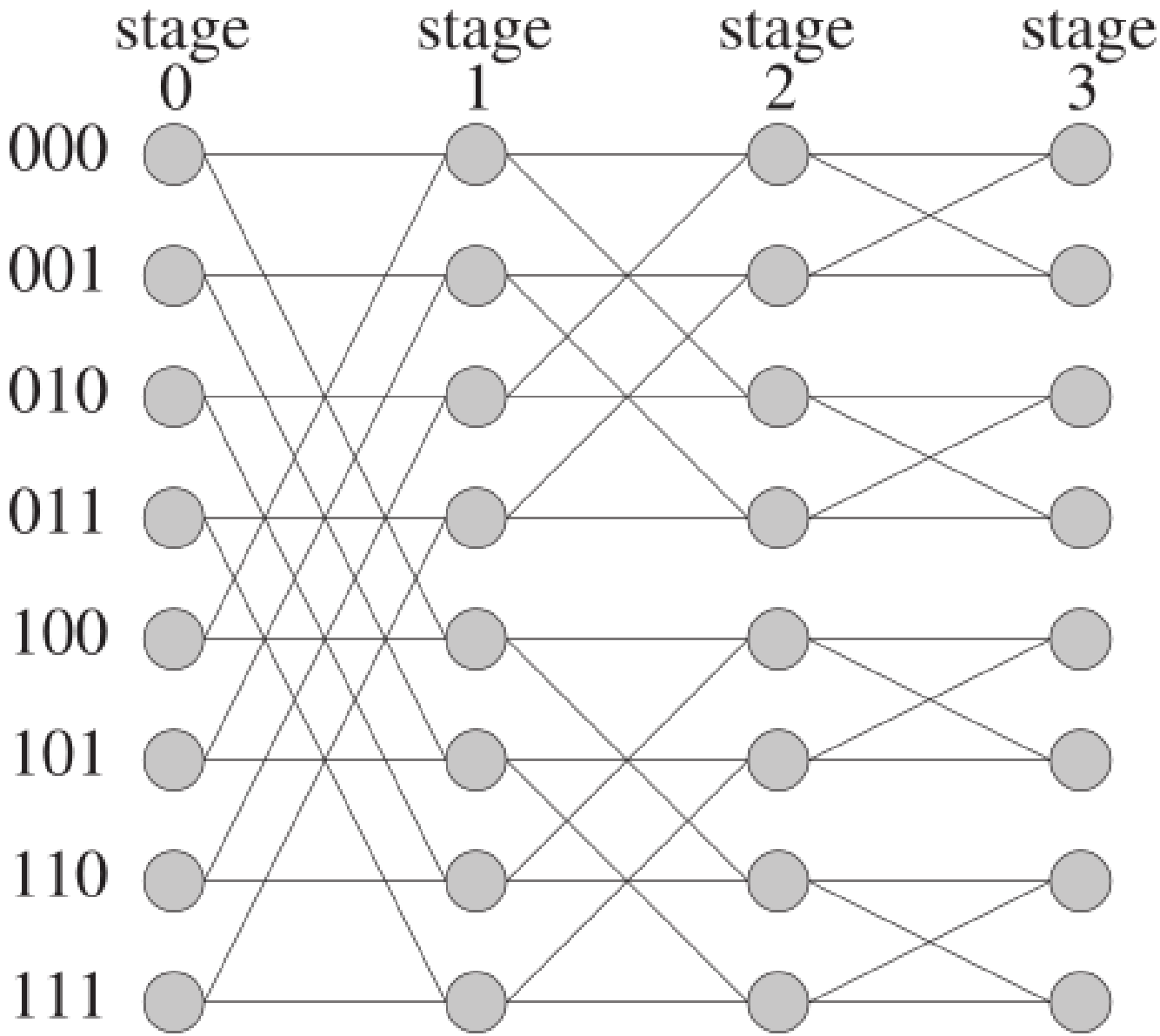


Rede *Butterfly*

Uma rede *butterfly* conecta $n = 2^{k+1}$ elementos a $n = 2^{k+1}$ elementos em $k + 1$ estágios com 2^k *switches* por estágio. Dois nós (α, i) e $(\beta, i + 1)$ são conectados se e somente se:

- α e β são idênticos;
- α e β diferem apenas no *bit* $i + 1$ a partir da esquerda.

Rede *Butterfly*

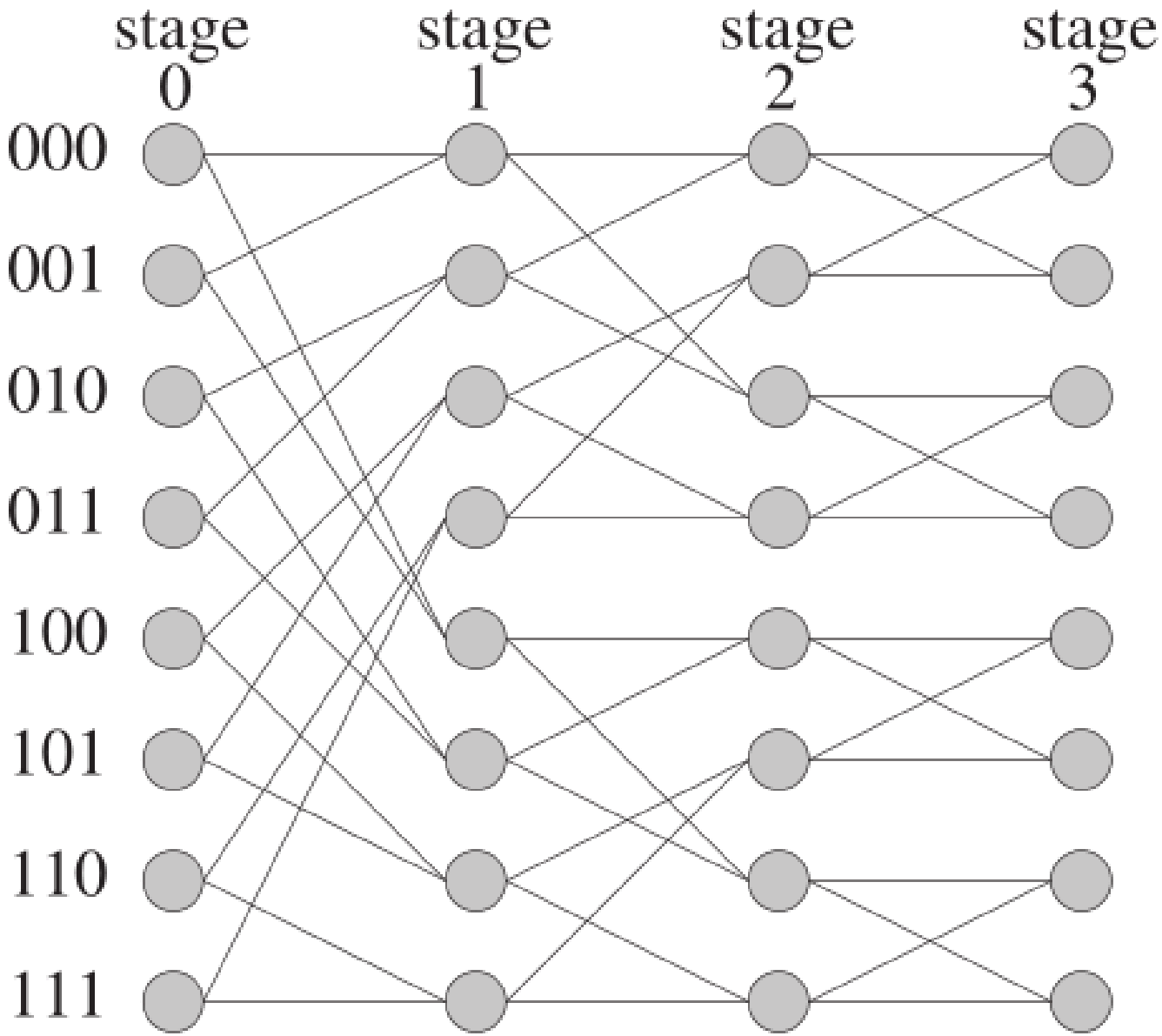


Rede *Baseline*

Semelhante a rede butterfly, porém com as seguintes para conexão dos *switches*:

- β é igual a α rotacionado à direita nos últimos $k - i$ *bits*;
- β é obtido de α invertendo o último *bit* de α e depois realizando uma rotação à direita nos últimos $k - i$ *bits*.

Rede *Baseline*

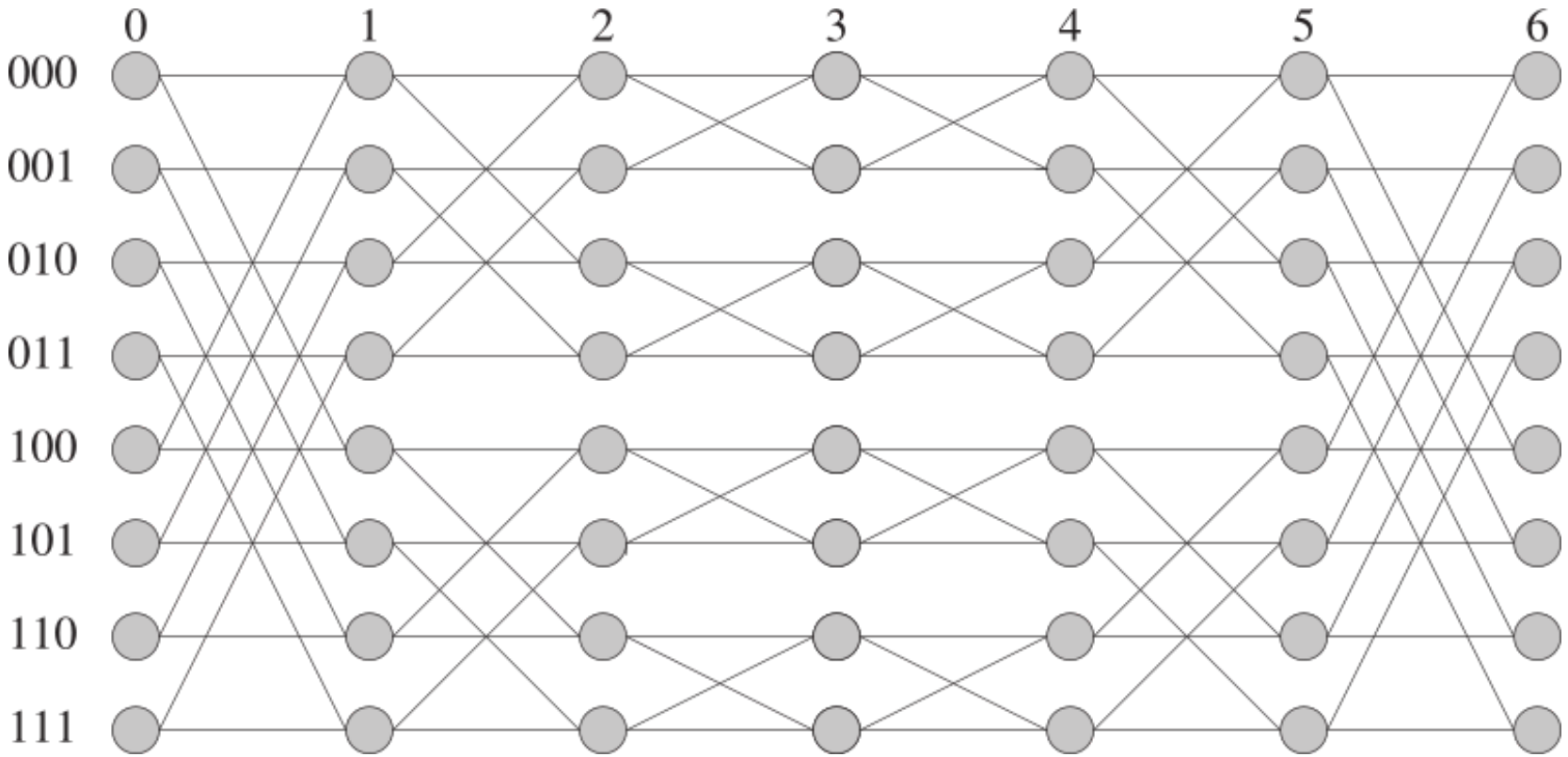


Rede *Benes*

União de duas redes *butterfly*

- Os primeiros $k + 1$ estágios são uma rede *butterfly* normal;
- os últimos $k + 1$ estágios são uma rede *butterfly* invertida.

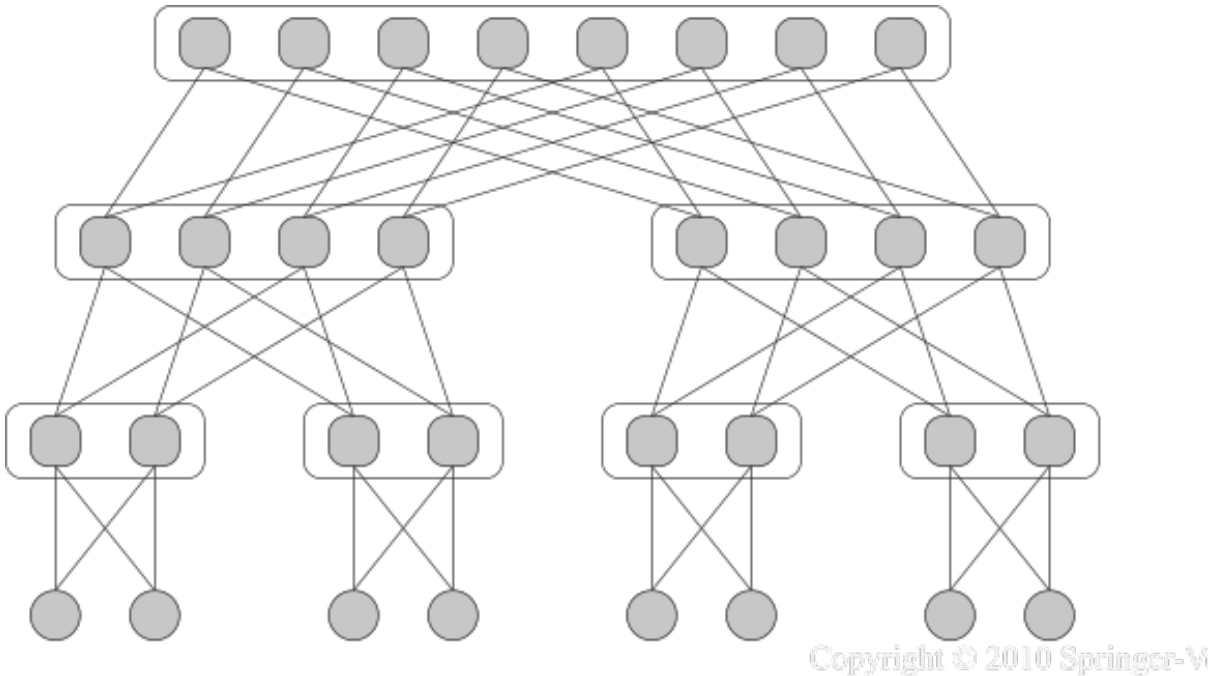
Rede *Benes*



Rede *FatTree*

- É como uma rede binária;
- porém com mais conexões a medida que se aproximamos da raiz;
- o objetivo é reduzir os gargalos.

Rede *FatTree*



Roteamento

No contexto das redes de interconexão:

- **Roteamento** é a escolha do caminho entre dois elementos finais;
- **switching** é como a mensagem é transmitida entre os nós intermediários.

Algoritmos de Roteamento

Um algoritmo de roteamento determina um caminho em uma dada rede entre um elemento A e um destino B. O caminho é uma sequência de elementos tal que nós vizinhos são conectados por uma ligação física. Os seguintes tópicos são importantes para a seleção de caminho:

- **Topologia da rede**: os grafos que já vimos;
- **contenção da rede**: o que fazer quando duas mensagens devem ser transmitidas ao mesmo tempo no mesmo *link*;
- **congestionamento da rede**: *overflow* de *buffers* nos dispositivos de comunicação.

Algoritmos de Roteamento

- Tamanho de caminho mínino vs. tamanho de caminho não-mínino?
- determinístico ou adaptativo?

Roteamento em Ordem de Dimensão - *Meshes*

Para enviar uma mensagem do elemento A com posição (X_A, Y_A) para um elemento B com posição (X_B, Y_B) a mensagem é enviada do elemento de origem na direção X até a coordenada X_B ser atingida. Depois o caminho é feito na direção Y.

Algoritmos de Roteamento

- Roteamento baseado em origem;
- Roteamento baseado em tabelas;
- Roteamento baseado em inversões;
- etc...

Em teoria qualquer técnica de roteamento das redes tradicionais pode ser usada, lembrando que em redes para computação paralela a quantidade de nós é menor e a exigência de velocidade é maior.

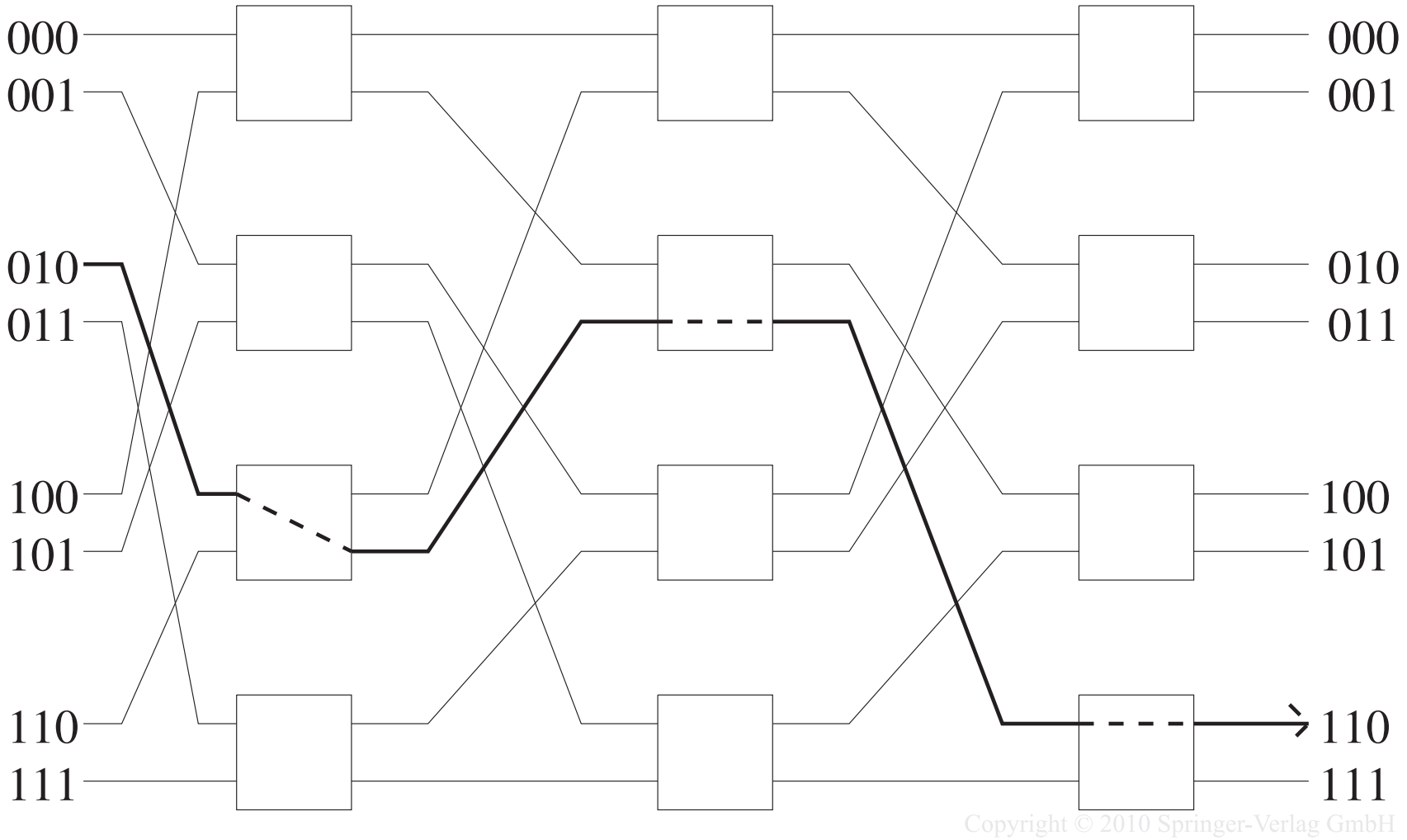
Roteamento na Rede Omega

Considere uma rede omega com n entradas e n saídas:

- Cada elemento é representado por uma cadeia de *bits* com comprimento $log(n)$;
- seja o objetivo rotear (definir caminho) uma mensagem do elemento α para um elemento β, passando por um elemento intermediário etapa k:
  - para o k-ésimo *bit* de β igual a zero, envie para o *link* superior;
  - para o k-ésimo *bit* de β igual a um, envie para o *link* inferior.

Em uma rede omega $n \times n$, no máximo n mensagens de entradas distintas para saídas distintas podem ser enviadas concorrentemente sem colisão.

Roteamento na Rede Omega



Caminho de 010 para 110.

Switching

Switching está relacionado a:

- Como a mensagem é dividida em partes;
- como o caminho definido pelo roteamento é alocado;
- como as mensagens são encaminhadas (*buffers* ou não) dentro de um *switch*.

Enquanto o roteamento cuida de toda o caminho, o *switching* está interessado na transferência em cada ligação.

Protocolo de Comunicação entre dois *Switches*

Elemento emissor:

1. A mensagem é copiada para o *buffer* do sistema;
2. um *checksum* é computado e adicionado ao cabeçalho da mensagem;
3. um temporizador é iniciado e a mensagem é enviada.

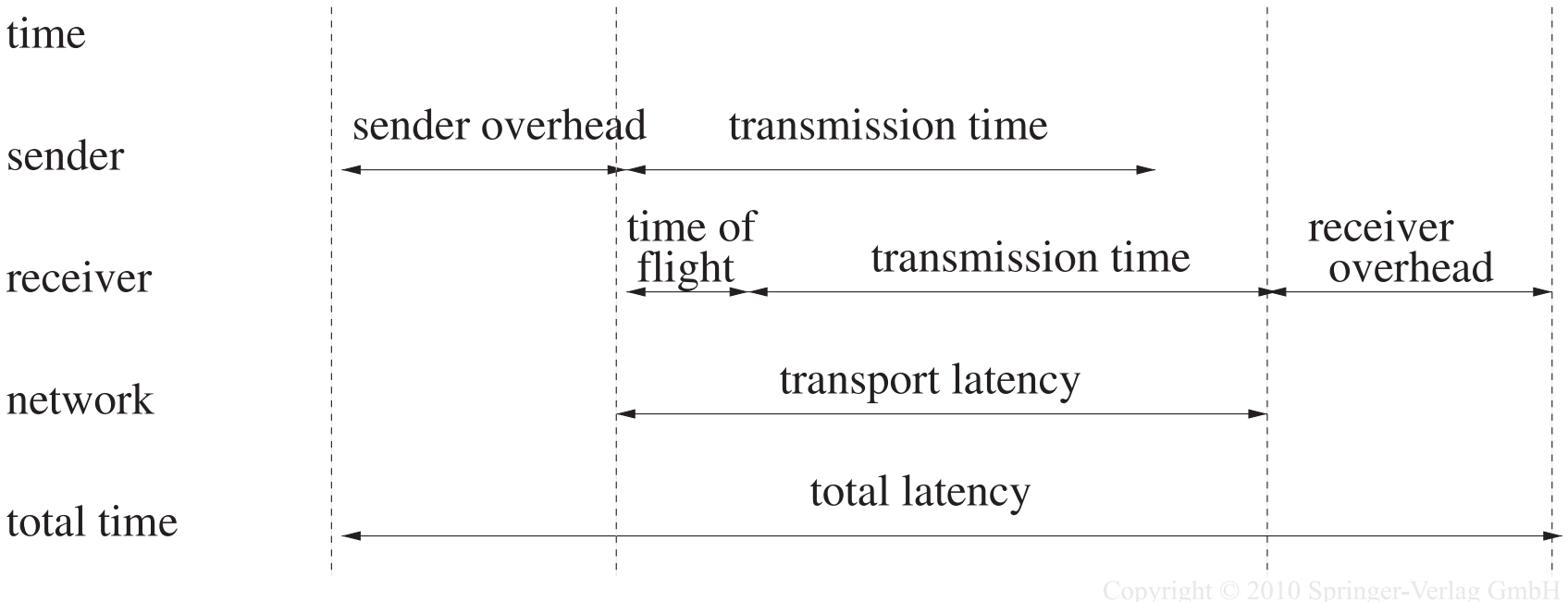
Elemento receptor:

1. Copia a mensagem da rede para o *buffer* de sistema;
2. o *checksum* é calculado. Se conferir com o do cabeçalho, envia um reconhecimento ao emissor. Caso contrário, descarta a mensagem;
3. ainda se os *checksuns* baterem, copiar a mensagem do *buffer* para as próximas etapas.

Métricas para a Transmissão de Mensagens

- Largura de banda;
- tempo de transferência de um *byte*;
- tempo de vôo;
- tempo de transmissão;
- latência de transporte;
- sobrecarga do emissor;
- sobrecarga do receptor;
- vazão.

Métricas para a Transmissão de Mensagens



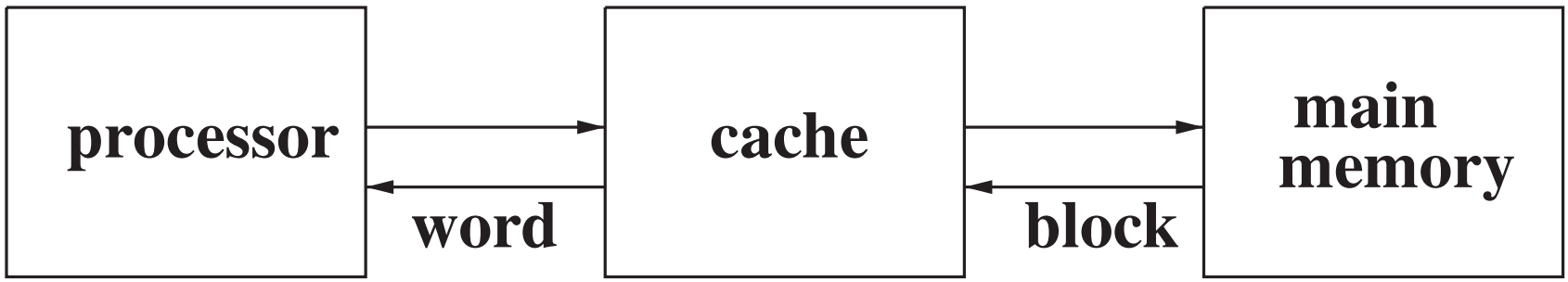
Latência total: $T(m) = O_{send} + T_{delay} + m/B + O_{recv}$

Caches e Hierarquia de Memória

- Diferença de velocida entre ciclo da CPU e tempo de acesso à memória.
- memória principal (DRAM);
- memórias *caches* (SDRAM);
- objetivo é diminuir o tempo médio de acesso à memória;
- estratégias de substituição são responsáveis pela manutenção da *cache*.

Em sistemas muliprocessados, nos quais cada processador ou núcleo tem uma *cache* local, existe o problema adicional de manter a consistência do espaço de endereçamento compartilhado.

Características das *Caches*



- Blocos ou linhas de *cache*;
- o processador não se importa com a organização da *cache*, apenas gera endereços;
- *cache hit*;
- *cache miss*;

Características das *Caches*

O comportamento do controlador de *cache* é oculto das outras unidades do processador:

- O programador não pode detalhar diretamente qual dado deve ficar na *cache*;
- mas é possível reordenar os acessos de memória indiretamente:
  - localidade espacial;
  - localidade temporal.
- o compilador também pode ajudar.

Tamanho da *Cache*

Tamanho total da *cache*:

- Devido ao número de conexões internas, aumentar o tamanho da *cache* pode aumentar o tempo de acesso;
- mas *caches* maiores levam a menos substituições, melhorando o desempenho.

Tamanho de um bloco da *cache*:

- Blocos grandes tendem a ser substituídos com maior frequência;
- blocos muitos pequenos causam muitas transferências entre a memória principal e a *cache*.

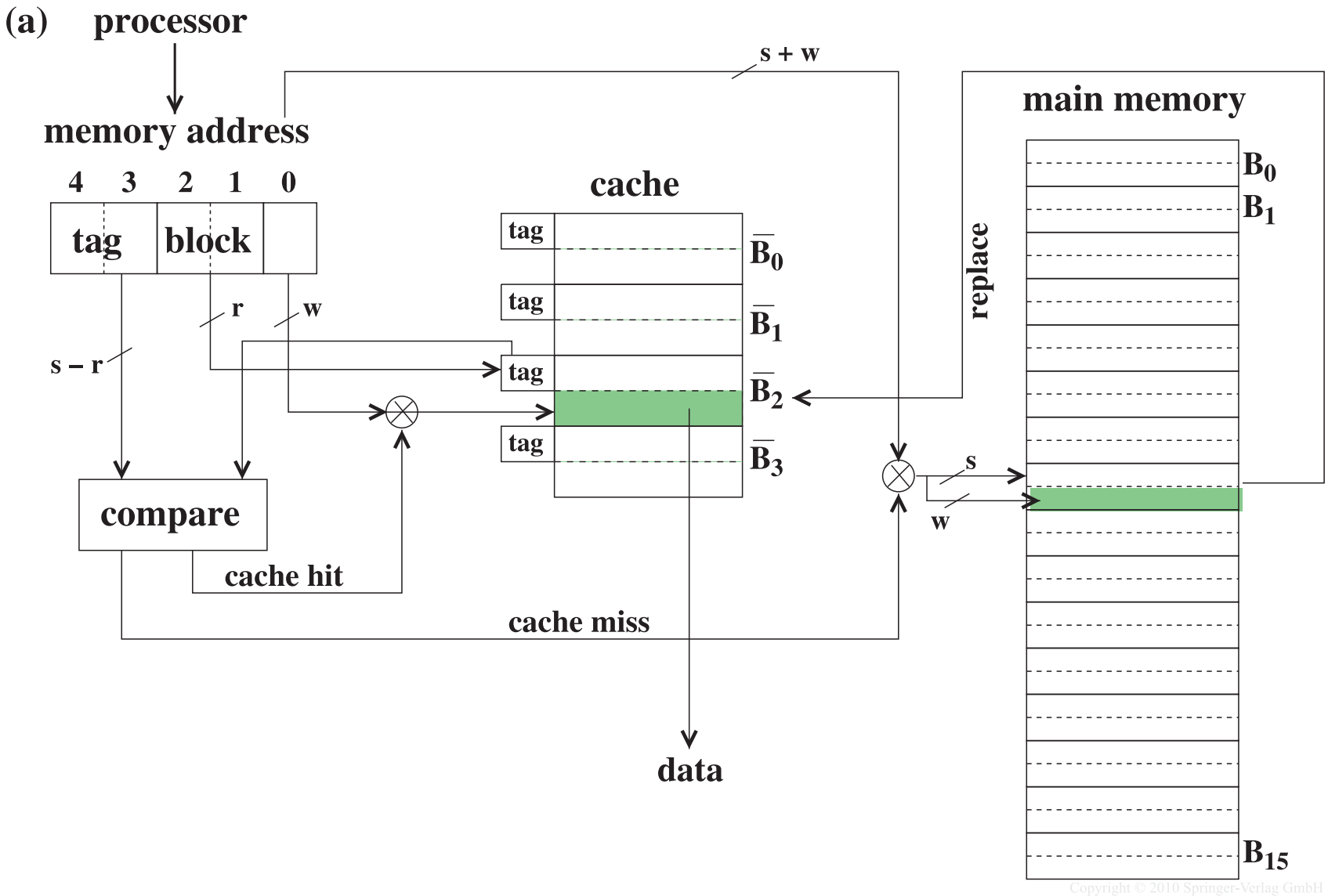
No meio-termo, os tamanhos dos blocos costumam ficar entre 4 ou 8 palavras.

Mapeamento de Blocos de Memória para Blocos na *Cache*

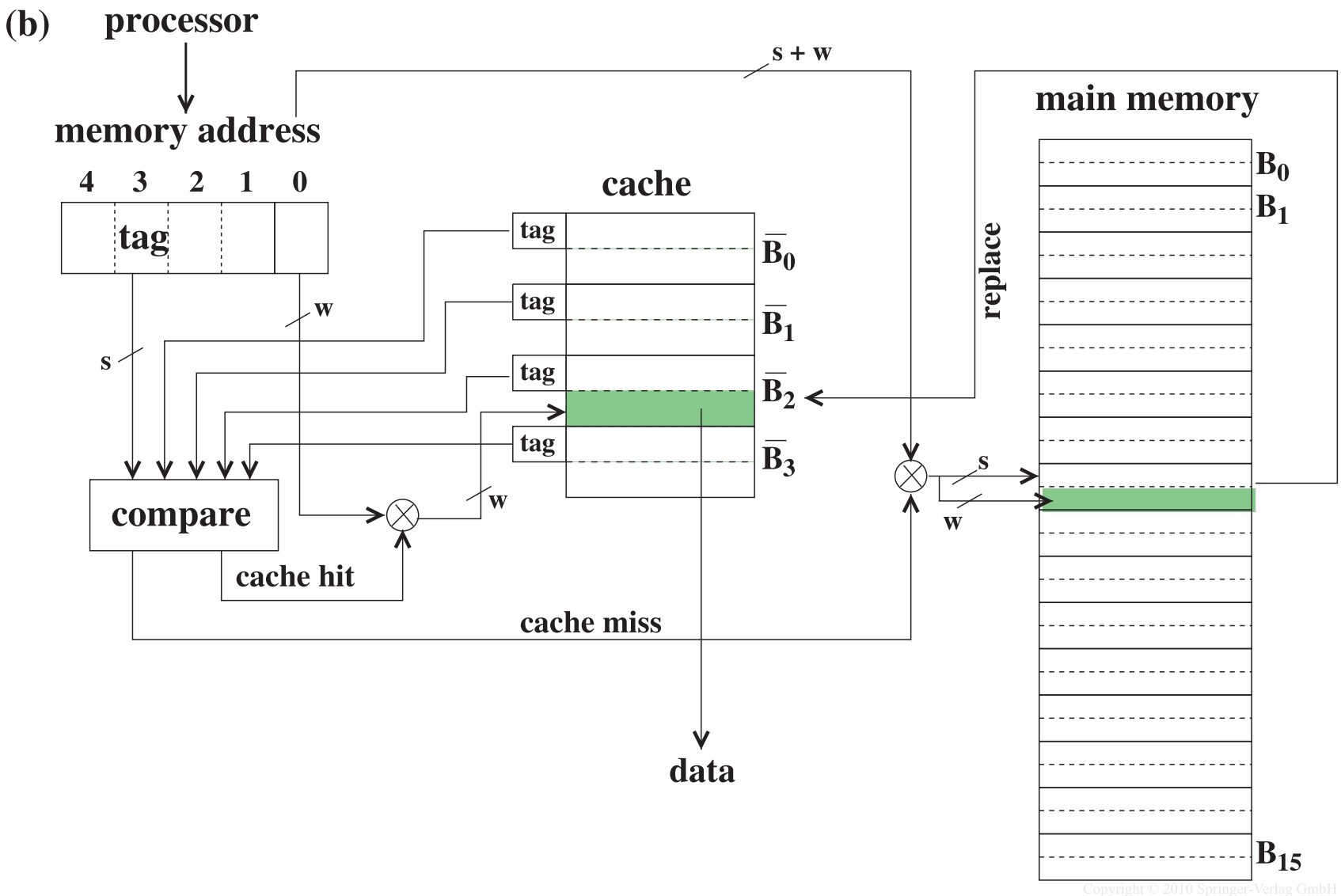
- Mapeamento Direto;
- mapeamento associativo completo;
- mapeamento associativo por conjunto;

Vamos mostrar exemplos para $m = 4$ blocos na *cache* ($r = 2$) e memória principal de $n = 16$ blocos ($s = 4$). Considere cada bloco com duas palavras ($w = 1$).

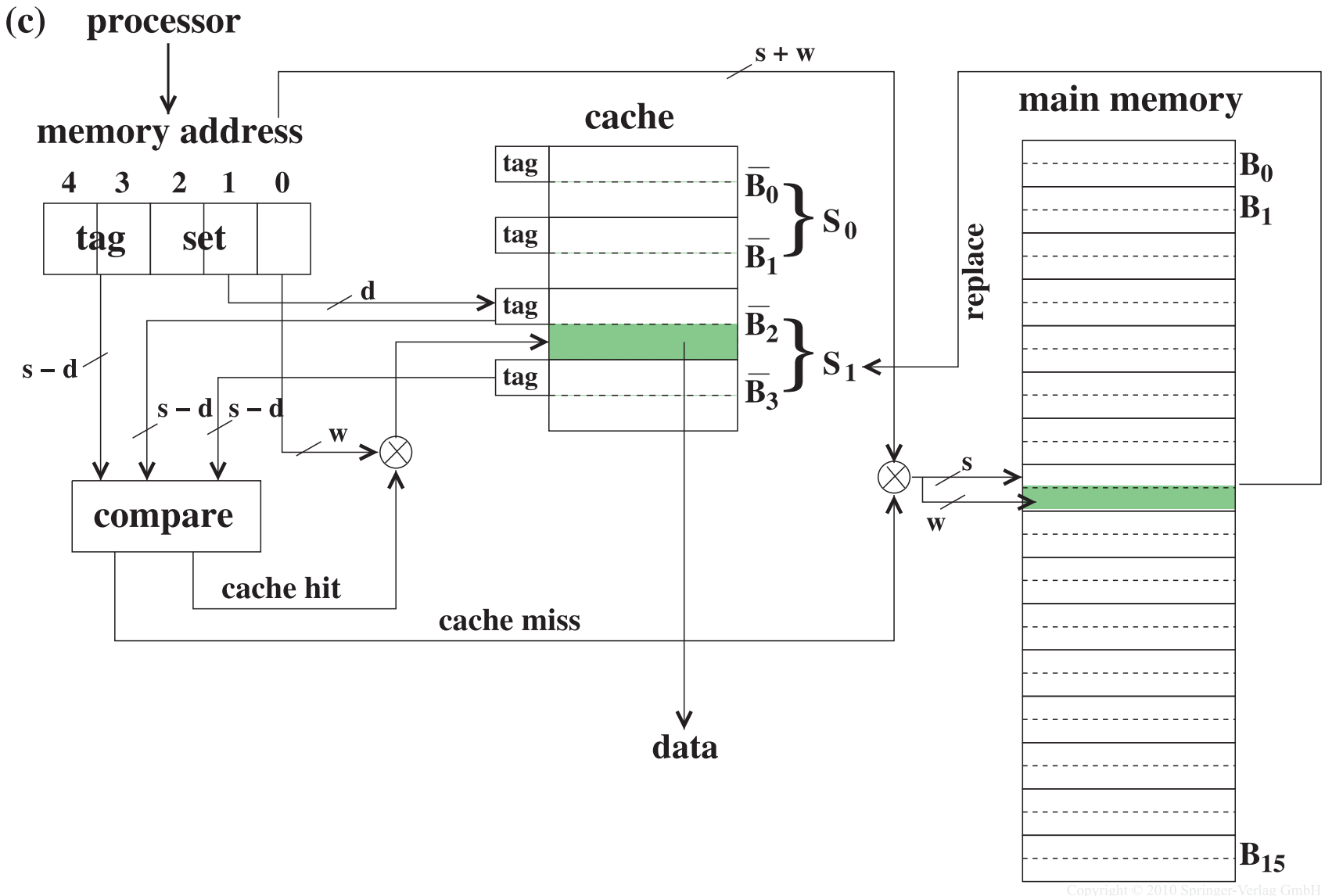
Mapeamento Direto



Mapeamento Associativo Completo



Mapeamento Associativo por Conjunto



Metódos de Substituição de Blocos

- LRU;
- LFU;
- aleatório.

Política de Escrita

Qual o bloco na memória principal é atualizado?

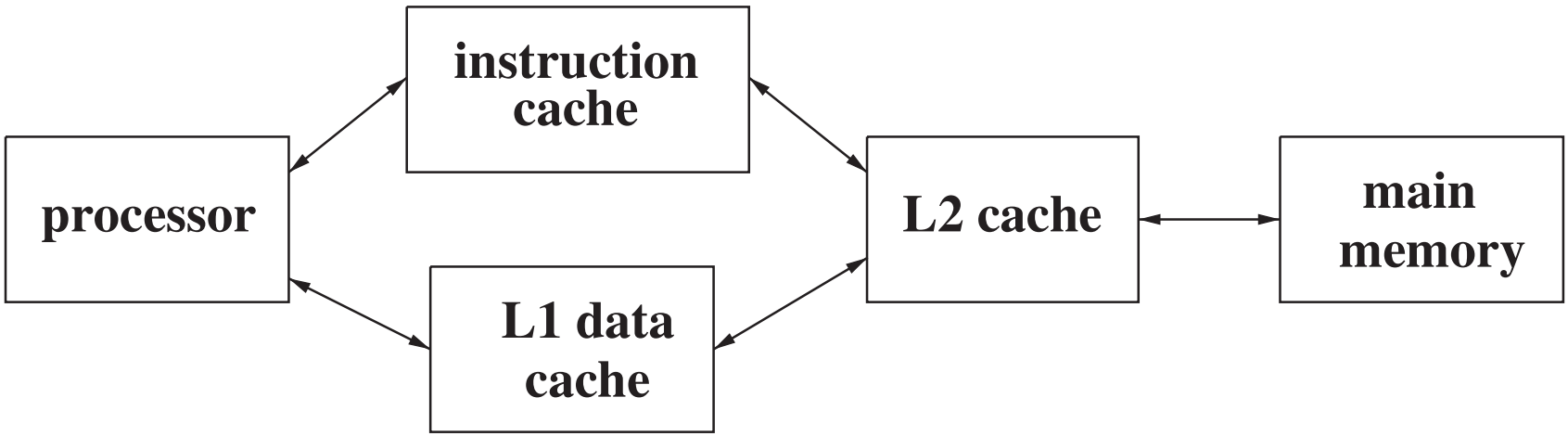
Write-Through

Imediatamente após a atualização da versão na *cache*.

Write-Back

Apenas escrever quando o bloco da *cache* for substituído.

Quantidade de *Caches*



Coerência de *Cache*

O problema de manter diferentes cópias do conteúdo de um endereço de memória consistentes.

Processadores P_1, P_2, P_3, *caches* C_1, C_2, C_3 e memória M por barramento.

1. P_1 lê variável u. Bloco contendo u, $M \rightarrow C_1$;
2. P_3 lê variável u. Bloco contendo u, $M \rightarrow C_3$;
3. P_3 escreve 7 em u. Através do *write-through*, valor atualizado na memória;
4. P_1 lê u acessando C_1;

Seja *write-through* ou *write-back*, o problema ocorre. Um sistema de memória é coerente se para cada posição de memória a leitura retorna o valor mais recente escrito.

Condições para um Sistema de Memória Coerente

1. Se P escreve em x em t_1 e em seguida lê o mesmo x em $t_2 > t_1$, sem nenhum outro processador acessar x entre t_1 e t_2, então o valor obtido em t_2 é o mesmo de t_1;
2. se P_1 escreve em x em t_1 e outro processo P_2 lê x em $t_2 > t_1$,então P_2 deve obter o valor escrito por P_1 se nenhum outro processador tiver escrito em x no intervalo t_1, t_2 e o intervalo $t_2 - t_1$ seja suficientemente grande;
3. se dois processadores escrevem em x ao mesmo tempo, essas operações devem ser serializadas e todos os processadores devem percebê-las na mesma ordem.

Protocolos *Snooping*

- Usam o fato de barramentos funcionarem como *broadcast*;
- baseados na atualização: quando um controlador de *cache* detecta a escrita em uma das posições de memória mapeadas na sua *cache*, ele atualiza o seu valor;
- baseados na invalidação: ao detectar uma escrita, o controlador apenas invalida o bloco na sua *cache*. Uma próxima leitura deve trazê-lo da memória;
- só funciona para a write-through e interconexão barramento.

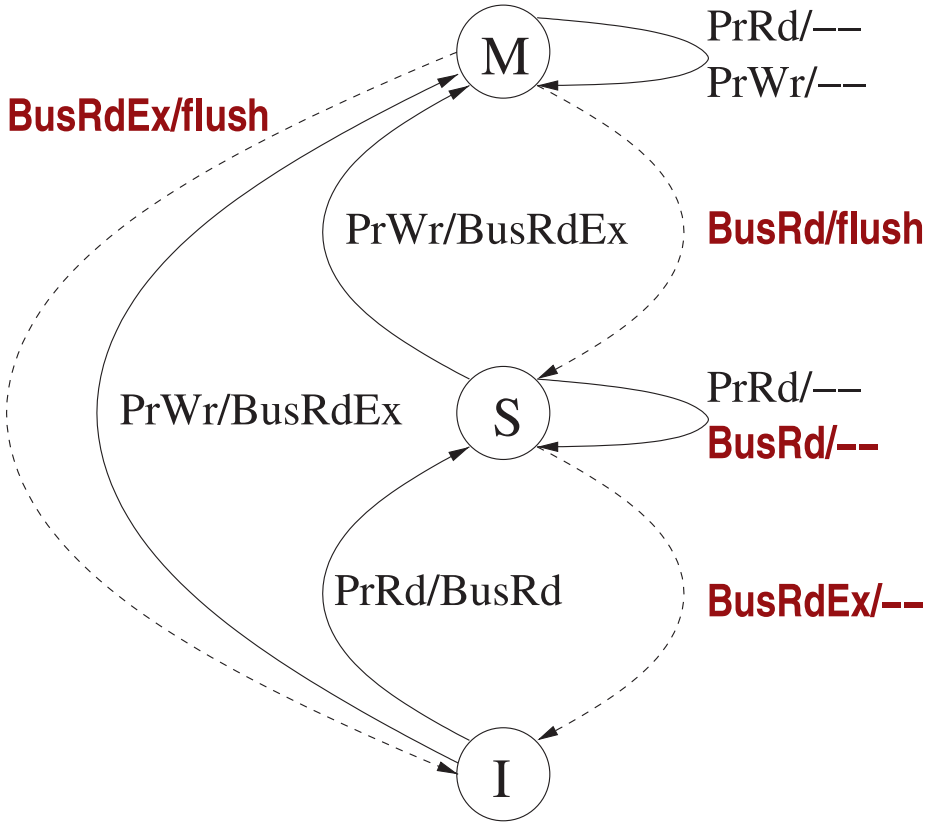
Protocolo de Invalidação *Write-Back*

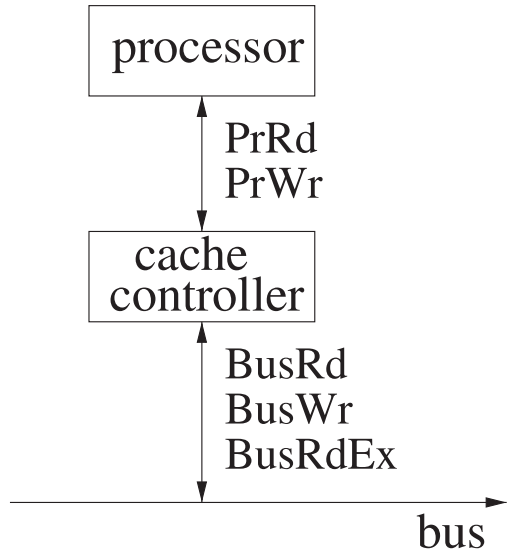
Estados de um bloco na *cache*:

- **M** (modificado): bloco foi atualizado na *cache*;
- **S** (compartilhado): bloco que não foi atualizado e contém o mesmo valor da memória e das outras *caches* do sistema;
- **I** (inválido): não contém o último valor escrito.

Operações no barramento:

- **Bus Read** (*BusRd*): originada pela leitura de bloco de memória por um processador (*PrRd*);
- **Bus Read Exclusive** (*BusRdEx*): originada pela escrita de um bloco de memória por um processador (*PrWr*);
- **Write-Back** (*BusWr*): escrever de volta um bloco modificado na memória, quando se torna necessário substituí-lo.





operation of the processor/issued operation of the cache controller

observed bus operation/issued operation of the cache controller

A operação *flush* significa que o controlador de *cache* coloca o valor do bloco no barramento.

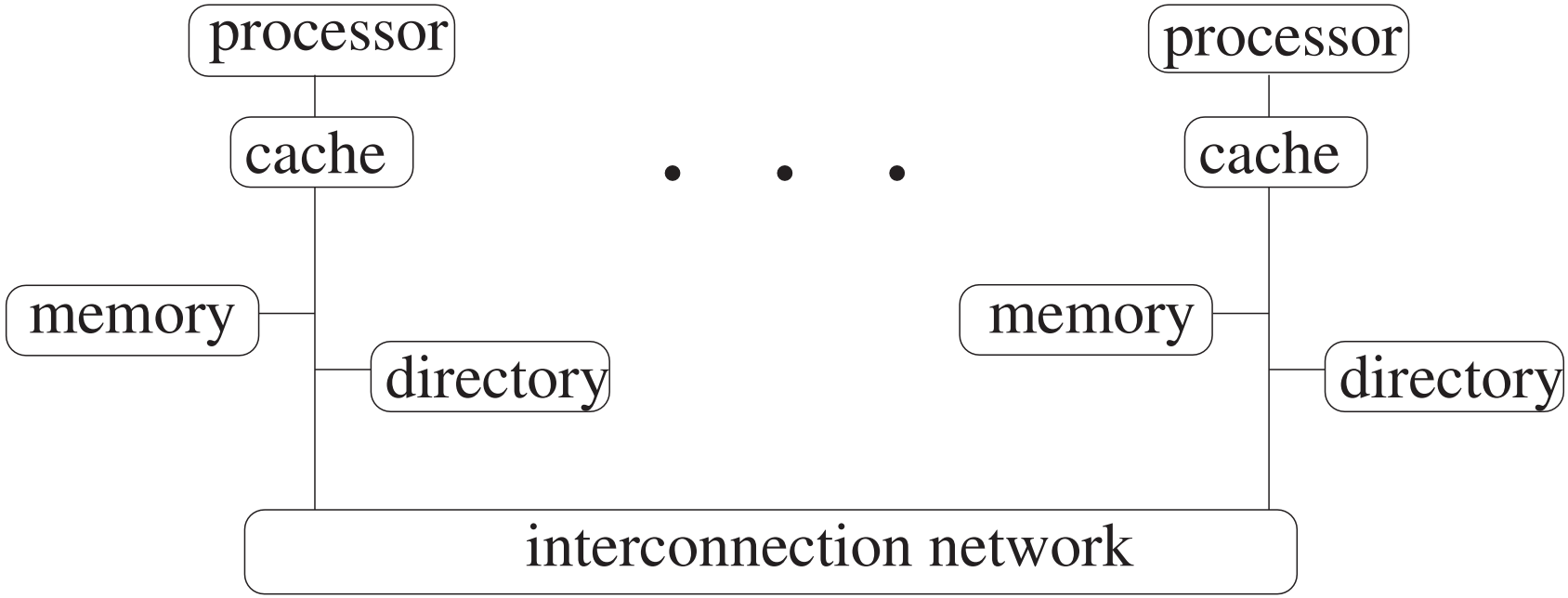
Protocolo de Invalidação *Write-Back*

- Desvantagem: leitura seguida de escrita acarretam duas operações no barramento (*BusRd, BusRdEx*);
- solução: adicionar um novo estado E (exclusivo): a *cache* contém o bloco, inalterado, mas somente a memória tem uma cópia, nenhum outro processador;
- se um processador emite *PrRd* e nenhum outro processo tem o bloco, o bloco é carregado e marcado como E;
- se entre a leitura e escrita local outro processador carregar o bloco, ele é alterado para S.

Temos então um protocolo MESI.

Protocolos Baseados em Diretórios

- Protocolos *snooping* são inadequados para redes que não são barramentos;
- em sistemas de memória distribuída, uma solução seria forçar a *cache* a só aceitar blocos de bancos de memórias locais;
- no caso de qualquer *cache* aceitar blocos de qualquer banco de memória, podemos usar protocolos baseados em diretórios;
- uma estrutura de diretórios (base de dados) é usada para armazenar o estado de cada bloco de memória do sistema.



Para cada memória local há um diretório que especifica para cada bloco local quais *caches* de outros processadores possuem uma cópia.

- vetor de p posições para cada bloco;
- *dirty bit*;
- na *cache* local, estados M, S, I;

Protocolos Baseados em Diretórios

P_i tenta acessar um bloco que não está na sua *cache*:

- Se o bloco for local e o *dirty bit* for 0, carrega o bloco;
- se o bloco for remoto ou o *dirty bit* for 1, é preciso requisitar o bloco na rede.

Quando o controlador da *cache* de P_j recebe a requisição de leitura de P_i:

- Se o *dirty bit* for 0, envia o bloco e atualiza seus *bits* no vetor de posições;
- se o *dirty bit* for 1, envia o bloco e atualiza o estado de M para S.

No caso da escrita, é preciso invalidar as cópias remotas. No momento da substituição dos blocos, eles devem ser retornados para sua memória de origem.

Consistência de Memória

- Coerência de *cache* garante que cada processador tenha a mesma visão da memória;
- não garante que a ordem das escritas seja a mesma para todos os processadores;
- modelo de consistência de memória:
  - Entrada: um conjunto de acessos a memória (leituras e escritas) parcialmente ordenados pelo código em execução nos processadores;
  - saida: a coleção dos valores retornados pelas operações de leituras.

Consistência de Memória - Exemplo

processador	P_1	P_2	P_3
programa	(1) $x_1 = 1;$	(3) $x_2 = 1;$	(5) $x_3 = 1;$
	(2) print $x_2, x_3;$	(4) print $x_1, x_3;$	(6) print $x_1, x_2;$

Quais são os possíveis valores da saída?

Consistência Sequencial

- Execução dos acessos na ordem do programa em cada processador;
- a ordem dos acessos é imprevísivel, mas uma vez definida, é vista como a mesma para todos os processadores do sistema;
- uma operação de memória tem que ser visível para todos os processadores antes que uma novo acesso seja feito.

As leituras não são tão sensíveis, o que complica a situação são as escritas.

Consistência Sequencial - Exemplo

Variáveis inicializadas em 0.

processador	P_1	P_2	P_3
programa	(1) $x_1 = 1;$	(2) while $x_1 == 0;$	(4) while $x_2 == 0;$
		(3) $x_2 = 1;$	(5) print $x_1;$

Se a atualização de P_1 não se propagar ao mesmo para para P_2 e P_3, que pode acontecer?

Condições para Consistência Sequêncial

- Cada processador obedece a ordem do programa ao emitir suas requisições de memória. O compilador não pode reoderná-las;
- após uma operação de escrita por um processador, se o bloco não estiver na *cache*, todos os blocos nas outras *caches* precisam ser invalidados antes do processador continuar;
- após uma operação de leitura por um processador; o valor só é atualizado na *cache* após a última escrita no mesmo endereço ter sido completada, ou seja, totalmente propagada.

Outros Modelos de Consistência

Permitem a reordenação de acessos para pelo desempenho;

- Consistência de Processador;
- Ordenação de armazenamento parcial;
- Modelos de ordenação fracos.

Conclusão

- Vimos vários detalhes sobre arquitetura de computadores paralelos;
- ainda é uma área de pesquisa intensa, principalmente devido a novas aplicações de IA e ML;
- mas agora vamos nos voltar para a programação.

Exercícios

Exercícios 2.1, 2.3, 2.13, 2.14, 2.15 e 2.16;