



Universidade Federal  
de Ouro Preto

Universidade Federal de Ouro Preto – UFOP  
Departamento de Computação e Sistemas - DECSI

# Paralelismo em Nível de Processadores

CSI509 – Organização e Arquitetura de Computadores II  
Prof. Samira Santos da Silva

# Sumário

- Introdução
- Conceitos Básicos
- Desafios da Programação Paralela.
- Sistemas de Memória (compartilhada / distribuída).
- Taxonomia de Flynn: SSID, SIMD, MISD, MIMD, etc.
- Programação Paralela em Memória Compartilhada/Distribuída.
- Benchmarks

# Introdução

# Introdução

- Multiprocessamento
  - Multiprocessadores: CPUs **mais poderosas** são construídas à partir da junção de **várias CPUs** mais simples.
  - Clientes/usuários podem enviar instruções a serem executadas a **todos os processadores** disponíveis.
  - Quanto **mais processadores ocupados**, “tende-se” a obter um **melhor desempenho** do sistema como um todo.
  - **Importante:** Softwares para ambientes multiprocessados funcionam com um **número variável de processadores**.

# Introdução

- Multiprocessamento
  - Como visto em OACI, consumo energético dos chips tornou-se o **principal problema** na evolução de desempenho.
  - Substituição de **um processador ineficiente por vários processadores eficientes** faz aumentar o desempenho por *watt/joule*.
  - *Software* para multiprocessadores deve ser **escalável**.
  - **Sistemas multiprocessados** tendem a ser **mais confiáveis**: em caso de falha de uma CPU, outras  $n-1$  CPUs podem continuar sendo utilizadas normalmente.

# Introdução

- Multiprocessamento:
  - Deve-se compreender a diferença entre os seguintes conceitos:
    - Paralelismo em nível de processo (*process-level parallelism*): Cada um dos múltiplos processadores executa um job/processo diferente.
    - Programa de processamento paralelo (*parallel processing program*): Um único programa/job/processo que executa simultaneamente em diferentes processadores

# Introdução

- Multiprocessamento:
  - Motivação para uso de múltiplos processadores é a **necessidade** cada vez **maior de processamento**.
  - Alguns dos problemas podem ser resolvidos utilizando-se uma **solução simples de multiprocessamento**: *clusters*.
  - *Clusters*: Interligação de **diversos microprocessadores** presentes em máquinas físicas diferentes (PCs ou servidores).
  - Evolução atual: **microprocessadores multicore**. Vários núcleos dentro de um único chip.

# Introdução

- Multiprocessamento
  - Espera-se que o número de cores **dobre a cada dois anos** (não necessariamente uma regra).
  - **Consequência dos multicores:** Programadores agora devem expandir seus conhecimentos ao “mundo paralelo” como forma de aproveitar melhor todo o poder dos novos hardwares.
  - Clarificando:

Hardware	Software		
	Sequential	Concurrent	
Serial	Matrix Multiply written in MatLab running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4	
Parallel	Matrix Multiply written in MATLAB running on an Intel Xeon e5345 (Clovertown)	Windows Vista Operating System running on an Intel Xeon e5345 (Clovertown)	

# Introdução

- Multiprocessamento
  - **Desafios** da paralelização:
    - Como fazer um *software* sequencial executar em *hardware* paralelo?
    - Como obter uma **melhoria contínua do desempenho de softwares concorrentes** executando em *hardware* paralelo à medida que o número de CPUs é aumentada?
    - Antes de se compreender os pormenores é preciso entender alguns conceitos básicos.

# Conceitos Básicos Paralelismo e Instruções: Sincronização

# Paralelismo e Instruções: Sincronização

- Introdução
  - Comumente, nossos programas **não executam em um único fluxo**.
  - Divisão do programa em **vários fluxos** visa **aumentar a eficiência e melhor utilização de recursos**.
  - Execução em paralelo é relativamente **fácil** quando as **tarefas são independentes**.
  - **Problema:** Normalmente, tarefas precisam cooperar umas com as outras para executarem determinado trabalho.

# Paralelismo e Instruções: Sincronização

- Introdução
  - **Cooperação**: Algumas tarefas escrevem novos valores que outras precisam ler.
  - Como determinada tarefa sabe quando outra terminou de escrever e já pode ler?
  - **Sincronização!**
  - **Problema**: A falta de sincronização pode levar a ocorrência do problema chamado de *data race*.

# Paralelismo e Instruções: Sincronização

- Introdução
  - **Data race:** “*Dois acessos a uma mesma região de memória vindos de diferentes threads onde pelo menos um deles é de escrita (write) e ocorrem um após o outro.*”
  - **Exemplo:** 8 jornalistas escrevendo uma mesma matéria.  
Qual o problema de quem escreve a conclusão?
    - Quem escreve a conclusão precisa ler todas as outras partes.
    - Quando escrevendo a conclusão, nada anteriormente escrito pode ser alterado.

# Paralelismo e Instruções: Sincronização

- Introdução
  - Na computação, comumente, **mecanismos de sincronização** estão em **nível de usuário**.
  - Tais mecanismos são **rotinas em software** que por sua vez são **amparadas por instruções em hardware**.
  - O foco é basicamente em instruções do tipo *lock* e *unlock*.
    - Criação de regiões onde somente um único processador pode operar, chamadas de regiões de **exclusão mútua** (*mutual exclusion*).

# Paralelismo e Instruções: Sincronização

- Exemplo (exclusão mútua)

```
#include <pthread.h>
#include <stdio.h>
#include <iostream>

using namespace std;

static int X = 0;
static int Y = 0;
static int Z = 0;

void *incrementa(void *pMsg)
{
    cout << (const char*)pMsg << endl;
    X = X + 1;
    usleep(10000);
    Y += X + 1;
    usleep(10000);
    Z += Y + 1;
}

void *imprime(void *pMsg)
{
    cout << (const char*)pMsg << endl;
    cout << "X = " << X << endl;
    cout << "Y = " << Y << endl;
    cout << "Z = " << Z << endl;
}

int main()
{
    pthread_t thread01, thread02, thread03, thread04;

    const char *msg1 = "Thread 1";
    const char *msg2 = "Thread 2";
    const char *msg3 = "Thread 3";
    const char *msg4 = "Thread 4";
    int retThread1, retThread2, retThread3, retThread4;

    retThread1 = pthread_create(&thread01, NULL, incrementa,
                               (void*)msg1);
    retThread3 = pthread_create(&thread03, NULL, incrementa,
                               (void*)msg3);
    retThread4 = pthread_create(&thread04, NULL, incrementa,
                               (void*)msg4);

    retThread2 = pthread_create(&thread02, NULL, imprime,
                               (void*)msg2);

    return 0;
}
```

Problema?

# Paralelismo e Instruções: Sincronização

- Exemplo (exclusão mútua)

```
#include <pthread.h>
#include <stdio.h>
#include <iostream>

using namespace std;

static int X = 0;
static int Y = 0;
static int Z = 0;
pthread_mutex_t mutexX;

void *incrementa(void *pMsg) {
    cout << (const char*)pMsg << endl;
    pthread_mutex_lock(&mutexX);
    X = X + 1;
    usleep(10000);
    Y += X + 1;
    usleep(10000);
    Z += Y + 1;
    pthread_mutex_unlock(&mutexX);
}

void *imprime(void *pMsg) {
    cout << (const char*)pMsg << endl;
    pthread_mutex_lock(&mutexX);
    cout << "X = " << X << endl;
    cout << "Y = " << Y << endl;
    cout << "Z = " << Z << endl;
    pthread_mutex_unlock(&mutexX);
}

int main()
{
    pthread_t thread01, thread02, thread03, thread04;

    pthread_mutex_init(&mutexX, NULL);

    const char *msg1 = "Thread 1";
    const char *msg2 = "Thread 2";
    const char *msg3 = "Thread 3";
    const char *msg4 = "Thread 4";
    int retThread1, retThread2, retThread3, retThread4;

    retThread1 = pthread_create(&thread01, NULL, incrementa, (void*)msg1);
    retThread3 = pthread_create(&thread03, NULL, incrementa, (void*)msg3);
    retThread4 = pthread_create(&thread04, NULL, incrementa, (void*)msg4);

    pthread_join(thread01, NULL);
    pthread_join(thread03, NULL);
    pthread_join(thread04, NULL);

    retThread2 = pthread_create(&thread02, NULL, imprime, (void*)msg2);
    pthread_join(thread02, NULL);

    pthread_mutex_destroy(&mutexX);

    return 0;
}
```

Solução!

# Paralelismo e Instruções: Sincronização

- Introdução
  - **Suporte a sincronização** em *hardware* de sistemas multiprocessados necessita de **primitivas especiais**.
  - Habilidade de **ler e modificar** uma dada **região de memória**.
  - Sem tal capacidade, **custo** para se construir uma estrutura de sincronização no *hardware* é **muito alto**.
  - **Importante:** Arquitetos de *hardware* projetam **primitivas de sincronização** para serem utilizadas por **programadores de sistemas operacionais** (não em nível de usuário).

# Paralelismo e Instruções: Sincronização

- Introdução
  - Operação básica (*atomic exchange/atomic swap*): Troca um valor em um registrador por um valor em memória.
  - Exemplo: Criação de um *lock* simples.
    - Valor 0 (*lock* disponível) / valor 1 (*lock* indisponível).
    - Processador tenta pegar o *lock* colocando o valor 1 em uma determinada posição de memória.
    - O valor retornado será 1 se algum outro processador já setou tal posição de memória. Caso contrário, 0 (zero).

# Paralelismo e Instruções: Sincronização

- Introdução:
  - Operação chave para conseguir sincronismo: **Atomicidade!**
  - **Impossível** que dois processadores atuem **simultaneamente sobre a mesma porção de memória** (escrita).
  - Implementar operações atômicas simples pode introduzir **novos desafios**:
    - Requer suporte a leitura e escrita em uma única instrução.
    - Instrução não pode ser interrompida.

# Paralelismo e Instruções: Sincronização

- Introdução
  - **Alternativa:** Possuir um **par de instruções** no qual a **segunda retorne** um valor que descreva se o **par de instruções foi executado atomicamente**.
  - Par de instruções será efetivamente atômico se as **duas forem executadas sem interrupção**.
  - Nenhum outro processador/processo pode alterar o valor entre o par de instruções.

# Paralelismo e Instruções: Sincronização

- Introdução
  - Em MIPS:
    - Suporte de um par de instruções especiais;
    - Operações de *load* e *store* podem ser executadas de forma diferente.
    - *load linked / store conditional*.
    - Instruções são utilizadas em sequência: Se o **conteúdo de um endereço de memória** especificado pelo *load linked* é alterado antes do *store conditional* executar no mesmo endereço de memória, o resultado final pode diferir.

# Paralelismo e Instruções: Sincronização

- Introdução
  - Em MIPS:
    - O *store conditional* é definido para ao mesmo tempo **armazenar o valor de um registrador em memória e alterar o valor de tal registrador** para:
      - 1 se a operação obteve sucesso; e
      - 0 se a operação falhou.

# Paralelismo e Instruções: Sincronização

- Introdução
  - Em MIPS:
    - Exemplo:

```
try: add $t0,$zero,$s4      ;copy exchange value
    li   $t1,0($s1)          ;load linked
    sc   $t0,0($s1)          ;store conditional
    beq  $t0,$zero,try       ;branch store fails
    add  $s4,$zero,$t1        ;put load value in $s4
```

- Copia valor que será armazenado em \$s4 para dentro de \$t0.

# Paralelismo e Instruções: Sincronização

- Introdução

- Em **MIPS**:

- Exemplo:

```
try: add $t0,$zero,$s4      ;copy exchange value
    li   $t1,0($s1)          ;load linked
    sc   $t0,0($s1)          ;store conditional
    beq  $t0,$zero,try       ;branch store fails
    add  $s4,$zero,$t1       ;put load value in $s4
```

- load linked especifica o endereço de memória em (\$s1).

# Paralelismo e Instruções: Sincronização

- Introdução
  - Em MIPS:
    - Exemplo:

```
try: add $t0,$zero,$s4      ;copy exchange value
    li   $t1,0($s1)          ;load linked
    sc   $t0,0($s1)          ;store conditional
    beq  $t0,$zero,try       ;branch store fails
    add  $s4,$zero,$t1        ;put load value in $s4
```

- *store condition* salvará o conteúdo de \$t0 em \$s1+0:
  - ✓ Se ao final, \$t0 contém 1, então operação foi bem sucedida;
  - ✓ Se ao final, \$t0 contém 0, então operação falhou.

# Paralelismo e Instruções: Sincronização

- Introdução

- Em **MIPS**:

- Exemplo:

```
try: add $t0,$zero,$s4      ;copy exchange value
    li   $t1,0($s1)          ;load linked
    sc   $t0,0($s1)          ;store conditional
    beq  $t0,$zero,try       ;branch store fails
    add  $s4,$zero,$t1        ;put load value in $s4
```

- Verificação se operação falhou ou teve sucesso.

# Paralelismo e Instruções: Sincronização

- Introdução

- Em **MIPS**:

- Exemplo:

```
try: add $t0,$zero,$s4      ;copy exchange value
    l1   $t1,0($s1)          ;load linked
    sc   $t0,0($s1)          ;store conditional
    beq  $t0,$zero,try       ;branch store fails
    add  $s4,$zero,$t1        ;put load value in $s4
```

- Retorno do valor que estava em memória para dentro de \$s4.

# Paralelismo e Instruções: Sincronização

- Introdução
  - Noções apresentadas servem tanto para **múltiplos processadores** quanto para **diferentes threads/processos em um mesmo SO**.
  - Em um único processador: *store conditional* também falhará quando da ocorrência de uma **troca de contexto**.
  - De forma segura, somente instruções **registrador-registrador** podem ser inseridas entre as duas instruções `ll` e `sc`.

# Paralelismo e Instruções: Sincronização

- Introdução
  - Número de instruções entre ll e sc devem ser **minimizadas**.
  - Possibilidade do problema conhecido como **deadlock**.
    - Processador nunca completa o sc devido ao grande número de faltas.
  - Quando as instruções ll e sc precisam ser utilizadas?
    - Em um **programa paralelo** onde **múltiplas threads** precisam ser **sincronizadas**; e
    - Em um **único processador** quando necessita-se **sincronizar a escrita/leitura dos dados**.

# Conceitos Básicos Paralelismo e Aritmética Computacional: Associatividade

# Paralelismo e Aritmética Computacional: Associatividade

- Introdução
  - Um mesmo *software* em suas **versões sequencial e paralela** deve prover **resultados iguais**.
  - Se **resultados diferentes** forem obtidos, deve-se depurar o *software* para se descobrir o **problema/bug**.
  - A **aritmética computacional não pode ser alterada** na transição do mundo sequencial para o mundo paralelo.
  - **Operações com inteiros** são ditas **associativas**, diferentemente de operações com números em ponto flutuante.

# Paralelismo e Aritmética Computacional: Associatividade

- Introdução

- Exemplo:

- Qual o resultado das seguintes operações?

$$x = -1.5 \times 10^{38}$$

$$y = 1.5 \times 10^{38}$$

$$z = 1.0$$

$$\begin{aligned}sum &= x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0) \\&= -1.5 \times 10^{38} + (1.5 \times 10^{38}) \\&= 0.0\end{aligned}$$

a)  $sum = x + (y + z)$

b)  $sum = (x + y) + z$



$$\begin{aligned}sum &= (x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0 \\&= (0.0) + 1.0 \\&= 1.0\end{aligned}$$

# Paralelismo e Aritmética Computacional: Associatividade

- Introdução
  - Problema pode ser agravado em ambientes computacionais distribuídos/paralelos.
  - Programador/desenvolvedor pode ser pego de surpresa ao notar que para uma mesma entrada seu programa provê resultados diferentes.
  - Execução do *software* em um sistema multiprocessado pode fazer com que as somas de ponto-flutuante sejam feitas em ordem diferente a cada execução.

# Conceitos Básicos Paralelismo e Paralelismo Avançado em Nível de Instruções

# Paralelismo e Paralelismo Avançado em Nível de Instruções

- Introdução
  - A técnica de *pipeline* explora um potencial paralelismo entre as instruções através do ILP (*Instruction Level Parallelism*).
  - Maneiras de se melhorar o desempenho de um *pipeline*:
    - Aumentar o número de estágio do *pipeline*; e
    - Replicar o número de componentes internos que perfazem determinadas tarefas (*multiple issue*).
  - **Problema:** Mais **esforço computacional** para manter todos os **estágios ocupados** e **transferindo dados** para os próximos estágios

# Paralelismo e Paralelismo Avançado em Nível de Instruções

- Introdução
  - *Multiple issue* permite que o CPI seja menor que um (<1).
    - Nesse caso, melhor métrica é utilizar IPC (instruções por ciclo).
  - Dois modos principais de se implementar *multiple issue*:
    - *Multiple issue estático*: A maioria das decisões sobre divisão do trabalho são feitas pelo **compilador**.
    - *Multiple issue dinâmico*: A maioria das decisões sobre divisão do trabalho são feitas pelo **hardware** em tempo de execução.

# Paralelismo e Paralelismo Avançado em Nível de Instruções

- Introdução
  - Pontos que precisam ser tratados em um processador com *multiple issue*:
    - Quantas e quais instruções devem ser lançadas em um dado ciclo de *clock*?
      - **Estaticamente**: Responsabilidade em maior parte a cargo dos compiladores;
      - **Dinamicamente**: Responsabilidade em maior parte a cargo do *hardware*, mas pode-se ter ajuda do compilador.

# Paralelismo e Paralelismo Avançado em Nível de Instruções

- Introdução
  - Pontos que precisam ser tratados em um processador com *multiple issue*:
    - Gerenciamento e tratamento dos dados em hazards de controle:
      - **Estaticamente**: Compilador tenta resolver o máximo possível dos problemas em tempo de compilação;
      - **Dinamicamente**: *Hardware* tenta reduzir o impacto dos *hazards* através de técnicas específicas em tempo de execução.

# Conceitos Básicos Paralelismo e Hierarquia de Memória: Coerência de Cache

# Paralelismo e Hierarquia de Memória: Coerência de Cache

- Introdução
  - Um processador **multicore** - como o próprio nome diz - possui internamente **vários núcleos** de processamento.
  - Esquemas de *cache* de memória precisam ser cuidadosamente implementados.
  - **Problema:** Dados compartilhados em *cache* podem ser vistos sob diferentes óticas por cada um dos processadores.
  - **Mesmos dados** podem ser enxergados com **diferentes valores** nas **caches de cada processador**.

# Paralelismo e Hierarquia de Memória: Coerência de Cache

- Introdução
  - Problema de coerência de cache: Dois processadores/núcleos podem ter diferentes valores associados a um mesmo local na memória.

Time step	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A stores 1 into X	1	0	1

- Um sistema de memória é **coerente** se uma leitura retorna sempre o valor mais recentemente escrito naquele segmento.

# Paralelismo e Hierarquia de Memória: Coerência de Cache

- Introdução
  - Dois conceitos críticos para uso de memória compartilhada:
    - **Consistência** - Determina quando um valor escrito será retornado por uma leitura.
    - **Coerência** - Um sistema de memória é coerente se:
      - Leitura executada por processo (P) em região (X) retorna sempre o valor previamente escrito por P (se entre escrita e leitura o valor de X não foi alterado por nenhum outro processo);
      - Uma leitura executada por P para X, logo após uma escrita de X, retorna novo valor de X se leitura/escrita estão suficientemente separados por um determinado período de tempo; e
      - Escritas executadas em um mesmo local (X) devem ser serializadas; i.e., devem ser enxergadas na mesma ordem por todos os outros processos.

# Paralelismo e Hierarquia de Memória: Coerência de Cache

- Introdução
  - Coerência
    - Premissas básicas:
      - Migração: Um dado pode ser **movimentado** para uma **cache local** e utilizado de forma **transparente**. Reduz latência de acesso a um item compartilhado e largura de banda para acessar um dado remoto.
      - Replicação: Dado pode ser replicado de forma que duas leituras sejam executadas mais rapidamente.
    - Protocolos para coerência de *cache*
      - **Requisito básico** para sua implementação é **manter o controle** e **acompanhar mudanças** em qualquer bloco de dados compartilhados.

# Paralelismo e Hierarquia de Memória: Coerência de Cache

- Introdução
  - Coerência
    - Protocolos *Snooping*:
      - **Forçar** com que cada **processador** tenha **acesso exclusivo** a um dado (região de memória) antes de uma **escrita**.
      - *Write invalidate protocol*: Invalideção de todas as outras cópias dos dados em *caches* de outros processos.
      - Acesso exclusivo garante que nenhuma outra cópia (leitura/escrita) exista quando o processo estiver executando a escrita).

# Paralelismo e Hierarquia de Memória: Coerência de Cache

- Introdução
  - Coerência
    - Protocolos *Snooping*:

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

# Conceitos Básicos

## Paralelismo e E/S: *Redundant Arrays of Inexpensive Disks*

# Paralelismo e E/S: *Redundant Arrays of Inexpensive Disks*

- Introdução
  - E/S deve ser considerada no mundo paralelo.
  - Grande impacto da E/S no desempenho geral do sistema.
  - Exemplo:
    - Um programa leva **100 segundos** para executar, sendo que **90 segundos** são gastos em **tempo de CPU** e o **restante em E/S**.
    - Considerando que o **número de processadores** dobrou a cada **dois anos** e que o **tempo de E/S não seja melhorado...**
    - Qual o tempo de execução deste mesmo programa ao final de seis anos?

# Paralelismo e E/S: *Redundant Arrays of Inexpensive Disks*

- Introdução
  - Exemplo: (cont.)
    - $\text{Tempo\_total} = \text{tempo\_cpu} + \text{tempo\_es}$   
 $100 = 90 + \text{tempo\_es}$   
**tempo\_es = 10s**

After n years	CPU time	I/O time	Elapsed time	% I/O time
0 years	90 seconds	10 seconds	100 seconds	10%
2 years	$\frac{90}{2} = 45$ seconds	10 seconds	55 seconds	18%
4 years	$\frac{45}{2} = 23$ seconds	10 seconds	33 seconds	31%
6 years	$\frac{23}{2} = 11$ seconds	10 seconds	21 seconds	47%

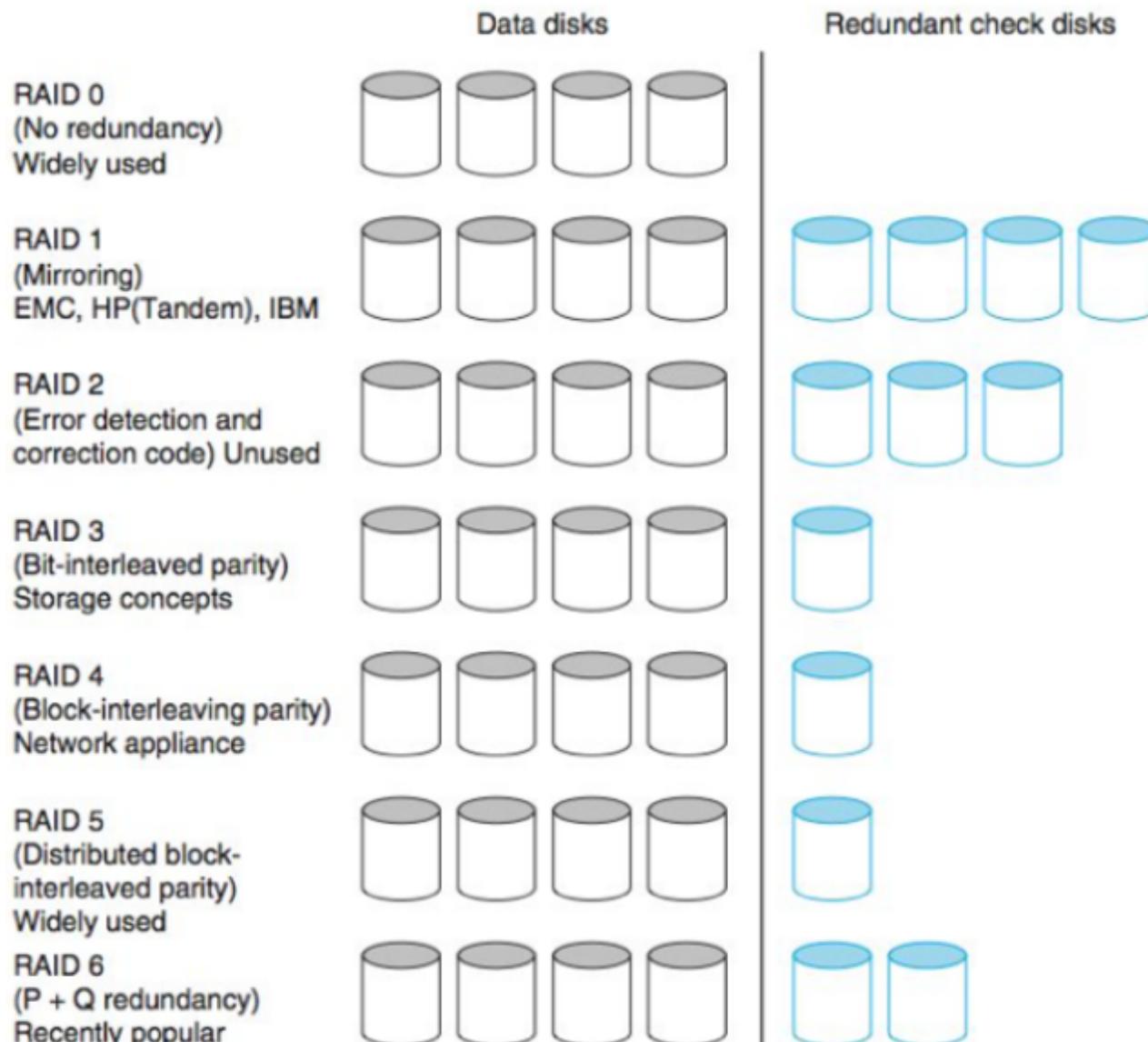
# Paralelismo e E/S: *Redundant Arrays of Inexpensive Disks*

- Introdução
  - Evolução no paralelismo dos processadores precisa ser acompanhada também pelo aumento de desempenho da E/S.
  - Array de discos surgiram na década de 80. Justificativa: Substituição de um grande disco por vários menores poderia paralelizar as leituras.
  - A mesma ideia se aplica a múltiplos processadores, uma vez que mais leituras poderão ser executadas em paralelo.

# Paralelismo e E/S: *Redundant Arrays of Inexpensive Disks*

- Introdução
  - **Confiabilidade:** Ao se aumentar o número de discos, **aumenta-se** também a **prob. de ocorrência de erro** em algum deles.
  - **Redundância:** Um aumento na confiabilidade passa pela redundância. Mesmo em caso de falhas, **informações não são perdidas**.
  - Criação da redundância tem um **custo muito baixo** nesse contexto. Daí o nome RAID (*redundant array of inexpensive disks*).
  - Quando considerando RAID:
    - O quanto de redundância é necessária?
    - Informações adicionais para detecção de falhas?
    - Modo de se organizar a informação importa?

# Paralelismo e E/S: *Redundant Arrays of Inexpensive Disks*



# Paralelismo e E/S: *Redundant Arrays of Inexpensive Disks*

- Tipos de RAID
  - RAID 0 (sem redundância)
    - Baseia-se fortemente no **espalhamento dos dados** pelos “n” discos.
    - Espalhamento dos dados **aumenta o desempenho** pois pode-se **buscar** de forma **concorrente** as informações.
    - Softwares/processos enxergam os dados como sendo uma única e grande coleção.

# Paralelismo e E/S: *Redundant Arrays of Inexpensive Disks*

- Tipos de RAID
  - RAID 1 (espelhamento)
    - Utiliza o **dobro de discos** se comparado ao RAID 0.
    - Sempre que um **dado é escrito** em algum **disco**, o mesmo dado é também **escrito no disco redundante**.
    - Se um **disco falha**, o sistema simplesmente **recorre** aos dados armazenados no **disco espelhado**.
    - Devido a duplicação de discos, é a **solução mais cara** de RAID existente.

# Paralelismo e E/S: *Redundant Arrays of Inexpensive Disks*

- Tipos de RAID
  - RAID 2 (detecção e código de correção)
    - Utiliza um esquema de **detecção e correção** muito utilizado em memórias.
    - Entretanto atualmente é muito **pouco utilizado**.
  - RAID 3 (bits intercalados)
    - O **custo** de se ter “alta disponibilidade” pode ser **reduzido** utilizando-se o conceito de grupo de proteção.
    - Ao contrário de se replicar todos os dados, é possível **armazenar somente dados específicos** que permitam **restaurar os dados originais**.

# Paralelismo e E/S: *Redundant Arrays of Inexpensive Disks*

- Tipos de RAID
  - RAID 3 (bits intercalados) (cont.)
    - Popular em aplicações com **grandes conjuntos de dados**.
    - **Paridade** é um dos possíveis esquemas a serem implementados para **recuperação de dados**.
  - RAID 4 (blocos intercalados)
    - Utiliza o **mesmo esquema de controle** que o RAID 3, mas **acesso aos dados** é feito de forma **diferente**.
    - A **paridade** é **armazenada** como um **bloco de dados** associados a um conjunto de informações.

# Paralelismo e E/S: *Redundant Arrays of Inexpensive Disks*

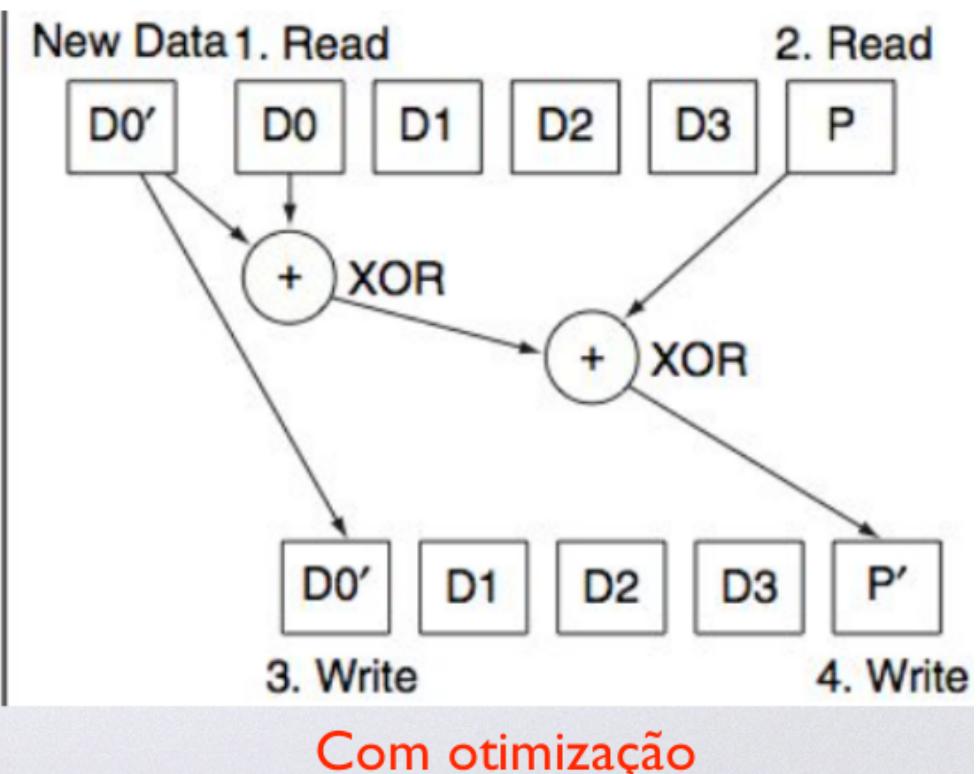
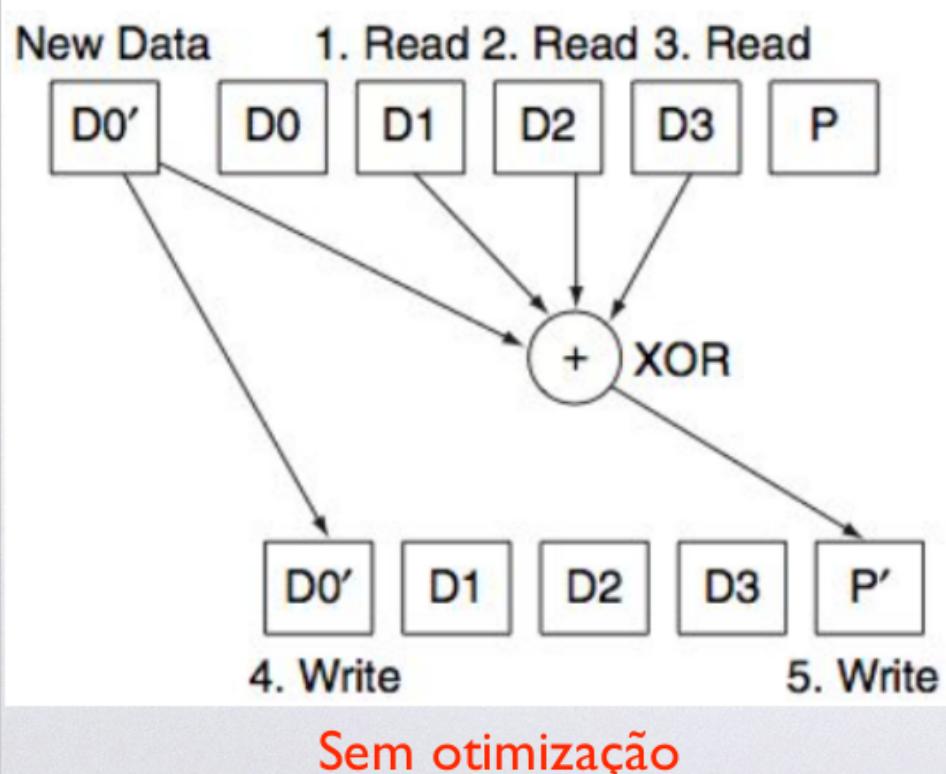
- Tipos de RAID
  - RAID 4 (blocos intercalados) (cont.)
    - Assim como no RAID 3, **pequenos acessos** são feitos para se **checar a consistência dos dados**.
    - Tais “pequenos acessos” são feitos **paralelamente** a acessos de **leitura normais**.
    - Uma “pequena leitura” vai diretamente para um disco de um grupo protegido - uma vez que só é necessário ler um setor.
    - Acessos a **dados úteis** podem ser feitos de **forma paralela** acessando dois ou mais discos de uma só vez.

# Paralelismo e E/S: *Redundant Arrays of Inexpensive Disks*

- Tipos de RAID
  - RAID 4 (blocos intercalados) (cont.)
    - “Pequena escrita” requer mais trabalho: **Ler o valor antigo do dado** e a respectiva **paridade antiga**. **Adicionar a nova informação**, e então **escrever a nova paridade** no disco de paridade e os **novos dados no disco de dados**.
    - Operação acima pode ser reduzida: Observa-se quais **bits são alterados** quando nova informação é escrita; somente **bits de paridade correspondentes** precisam ser **alterados no disco de paridade**.

# Paralelismo e E/S: Redundant Arrays of Inexpensive Disks

- Tipos de RAID
  - RAID 4 (blocos intercalados) (cont.)

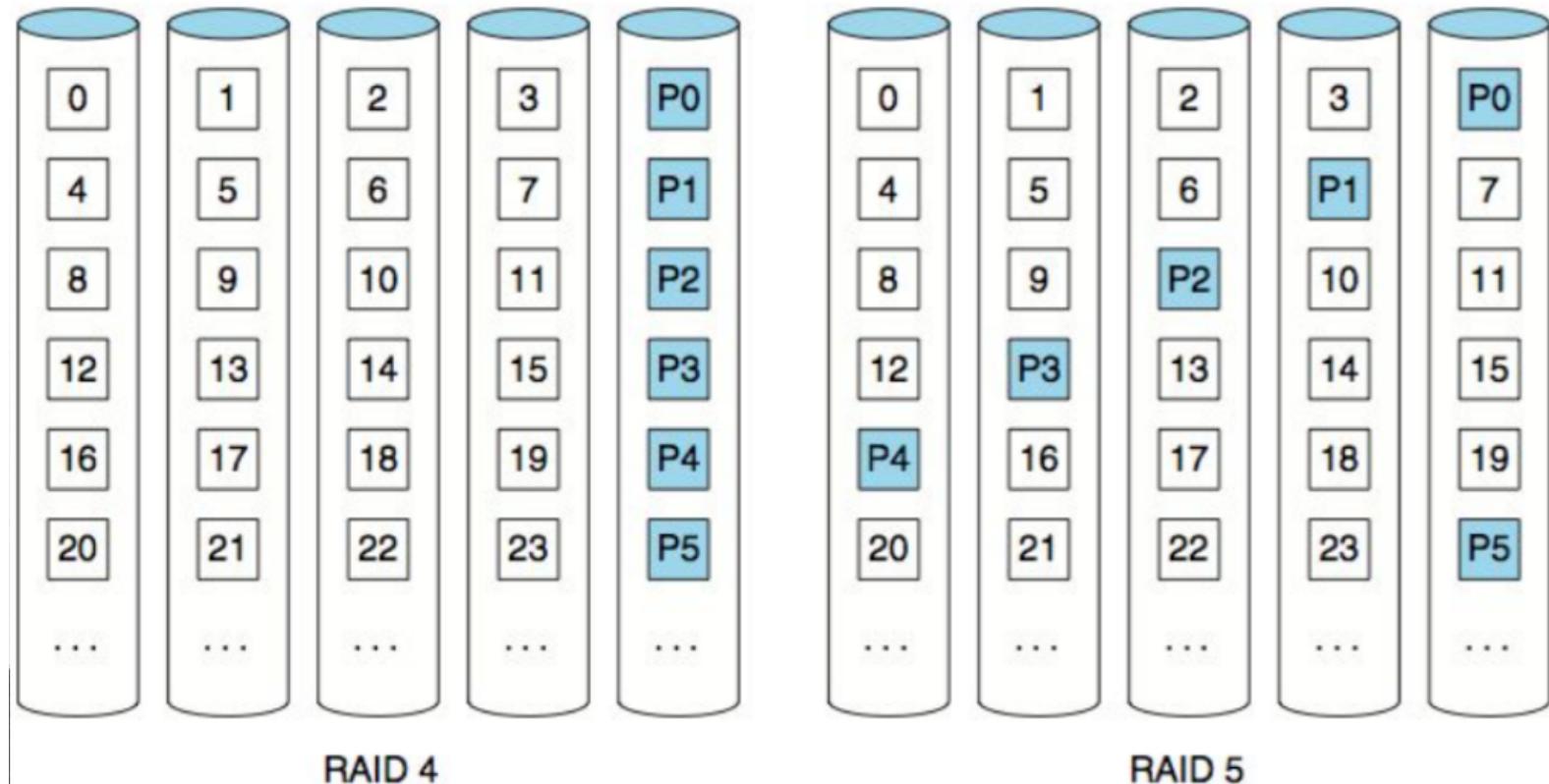


# Paralelismo e E/S: *Redundant Arrays of Inexpensive Disks*

- Tipos de RAID
  - RAID 5 (blocos intercalados distribuídos)
    - **Problema:** Desvantagem do RAID 4 é que o **disco de paridade** precisa ser **atualizado a cada escrita**, se tornando o **gargalo** do sistema.
    - **Solução:** Informação sobre **paridade** pode ser **espalhada** por todos os **discos do sistema**. Dessa forma não existiria um gargalo central para as escritas.

# Paralelismo e E/S: *Redundant Arrays of Inexpensive Disks*

- Tipos de RAID
  - RAID 5 (blocos intercalados distribuídos)



# Paralelismo e E/S: *Redundant Arrays of Inexpensive Disks*

- Tipos de RAID
  - RAID 6 (Redundância P + Q)
    - Esquemas de proteção via **paridade protegem contra uma falha simples.**
    - Mesmo esquema pode ser generalizado para um **nível adicional de proteção**.
    - Um segundo conjunto de **blocos de checagem (Q)** possibilita a **restauração** em caso de um **segundo erro**.
    - Consequentemente, o *overhead* de armazenamento é duas vezes maior que no RAID 5.

# Criação de Programas de Processamento Paralelo

# Criação de Programas de Processamento Paralelo

- Introdução
  - A **dificuldade** com o paralelismo em sua grande maioria **não está relacionada ao hardware**.
  - Quanto mais processadores/núcleos existem, mais **complicado** fica **escrever programas** que os utilizem e tirem vantagem dessa arquitetura.
  - Por que escrever programas paralelos é tão mais complicado que programas sequenciais?
    - **Obrigatoriamente** precisa-se obter **melhor desempenho e eficiência**.

# Criação de Programas de Processamento Paralelo

- Introdução
  - Algumas técnicas em *hardware* tiram vantagem do ILP (*instruction level parallelism*) **sem necessidade de reescrita dos programas.**
  - Outras razões para a complexidade de programas paralelos:
    - **Divisão e balanceamento** dos **trabalhos** de cada uma das tarefas do programa;
    - **Sincronização** das tarefas envolvidas no trabalho.
    - De acordo com a lei de Amdahl, pouquíssimas partes **sequenciais** afetam de forma significativa a execução do todo.

# Criação de Programas de Processamento Paralelo

- Introdução

- **Exemplo:** Suponha que você possua um programa deseja obter uma **melhoria** de desempenho de **90 vezes** utilizando **100 processadores**. Qual o porcentagem do programa inicial pode ser mantida como “sequencial”?
    - Lei de Amdahl:

$$\text{ExecutionTimeAfterImprovement} = \frac{\text{ExecutionTimeAffectedByImprovement}}{\text{AmountOfImprovement}} + \text{ExecutionTimeUnaffected}$$

- Reescrevendo para considerar a melhoria (*speedup*) em relação ao tempo de execução original:

$$\text{SpeedUp} = \frac{\text{ExecutionTimeBefore}}{(\text{ExecutionTimeBefore} - \text{ExecutionTimeAffected}) + \frac{\text{ExecutionTimeAffected}}{100}}$$

# Criação de Programas de Processamento Paralelo

- Introdução
  - Exemplo: (cont.)
    - Considerando o tempo de execução anterior como “1”:

$$SpeedUp = \frac{1}{(1 - FractionTimeAffected) + \frac{FractionTimeAffected}{100}}$$

- Substituindo os valores/objetivos do problema:

$$90 = \frac{1}{(1 - FractionTimeAffected) + \frac{FractionTimeAffected}{100}}$$

- Resolvendo:

$$90 - 1 = 90 \times 0.9 \times FractionTimeAffected$$

$$FractionTimeAffected = \frac{89}{89.1} = 0.99$$

- Resultado: Código sequencial pode ser no máximo 0.1% do total.

# Criação de Programas de Processamento Paralelo

- Introdução
  - Análise das características de possível paralelização leva a dois conceitos diferentes:
    - *Strong scaling:*
      - Melhoria no speedup em multiprocessadores enquanto complexidade do software/problema se mantém.
    - *Weak scaling:*
      - Melhoria no speedup em multiprocessadores aumentando a complexidade do software/problema.

# Criação de Programas de Processamento Paralelo

- Introdução
  - Exemplo: Balanceamento de carga
    - No exemplo anterior (100 processadores), assumiu-se que a **carga dos processadores** estava **perfeitamente balanceada**.
    - **Cada** um dos 100 processadores possui exatamente **1% da carga total** do sistema.
    - O que aconteceria **se um dos processadores tivesse uma carga maior que todos os outros (2% ou 5%)?**
    - Considerando um programa que faz 10000 somas, o processador teria:  $2\% \times 10000 = 200$  operações para executar.
    - Outros 99 procs. teriam 9800 somas a executar

# Criação de Programas de Processamento Paralelo

- Introdução
  - Exemplo: Balanceamento de carga (cont.)
    - Uma vez que os processadores estão executando em paralelo, é possível se calcular o **tempo de execução** como sendo:

$$\text{ExecutionTime} = \text{Max}\left(\frac{9800t}{99}, \frac{200t}{1}\right) + 10t = 210t$$

- Speedup cai para  $10010t/210t = 48$ .
  - Considerando que agora um só processador é responsável por 5% da carga:

$$\text{ExecutionTime} = \text{Max}\left(\frac{9500t}{99}, \frac{500t}{1}\right) + 10t = 510t$$

- Speedup cai para  $10010t/510t = 20$ .

# Multiprocessadores de Memória Compartilhada

# Multiprocessadores de Memória Compartilhada

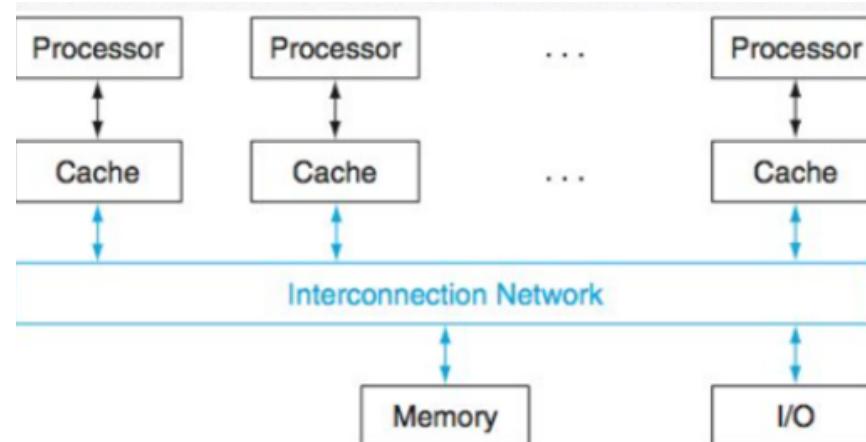
- Introdução
  - Como visto anteriormente, rescrever programas antigos para executar em paralelo pode ser uma tarefa dispendiosa.
  - Uma opção a isso é a utilização de processadores que compartilham um mesmo espaço de endereçamento de memória.
  - Nesse cenário, uma variável de um programa em execução poderia ser acessada por qualquer um dos processadores.
  - **Importante:** Quando o espaço de endereçamento é compartilhado, normalmente o *hardware* já fornece coerência de cache.

# Multiprocessadores de Memória Compartilhada

- Introdução

- *Shared Memory Multiprocessor (SMP)*

- Oferece ao programa em execução um único espaço de endereçamento de memória para todos os processadores.
    - Somente espaço de endereçamento físico é compartilhado. Endereçamento virtual continua único para cada processador.
    - Processadores se comunicam através de variáveis compartilhadas (via *load/store*).



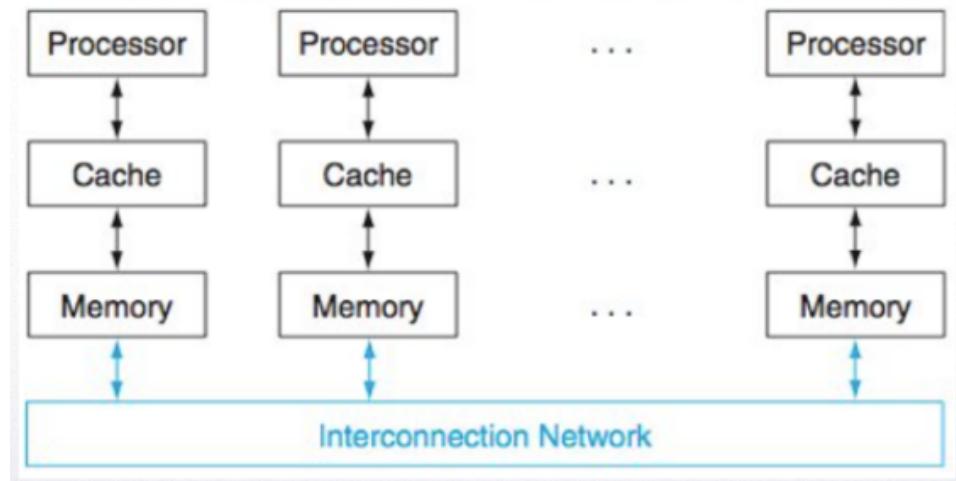
# Multiprocessadores de Memória Compartilhada

- Introdução
  - *Shared Memory Multiprocessor (SMP)*
    - Compartilhamento de espaço de memória podem ser feitos de duas maneiras diferentes:
      - *Uniform Memory Access (UMA)*: Acesso à memória principal leva o mesmo tempo, não importando qual processador está fazendo o acesso.
      - *NonUniform Memory Access (NUMA)*: Determinados acessos à memória podem ser executados muito mais rapidamente, dependendo do processador que a requisita.
    - Regiões compartilhadas de memória geram necessidade de “*sincronização*” de acesso que pode ser implementada via “*locks*”.

# Clusters e Multiprocessadores de Trocas de Mensagens

# Clusters e Multiprocessadores de Trocas de Mensagens

- Introdução
  - Uma alternativa ao modelo de memória compartilhada é a arquitetura na qual cada processador possui sua memória.



- Comunicação entre processadores ocorre via trocas de mensagens.

# Clusters e Multiprocessadores de Trocas de Mensagens

- Introdução
  - Sistema possui rotinas para envio e recebimento de mensagens.
  - Coordenação e sincronização está implícito no envio/recebimento de mensagens:
    - Processador que envia sabe quando a mensagem foi enviada;
    - Processador que recebe sabe quando a mensagem foi recebida;
    - Se houver necessidade, confirmação de recebimento pode ser entregue ao remetente.

# Clusters e Multiprocessadores de Trocas de Mensagens

- Introdução
  - *Clusters*
    - Coleção de computadores conectados via rede cooperando através de mecanismos de trocas de mensagens.
    - Cada elemento do *cluster* executa uma cópia distinta do sistema operacional.
    - São justificados pois dificilmente existem aplicações que justifiquem a utilização de um esquema de comunicação muito rápido.
    - Entretanto, custo de administração de um *cluster* de  $n$  máquinas é quase o mesmo de se administrar tais  $n$  máquinas separadamente.

# Clusters e Multiprocessadores de Trocas de Mensagens

- Introdução
  - *Clusters*
    - Custo para se administrar um sistema de memória compartilhada de  $n$  núcleos é o mesmo que se administrar uma única máquina.
    - Recentemente, tais custos têm levado ao uso de máquinas virtuais:
      - Facilidade na administração;
      - Migração de programas de uma VM para outra sem parar o sistema;
      - Em caso de erro de *hardware*, programa e execução pode ser aproveitado.

# Clusters e Multiprocessadores de Trocas de Mensagens

- Introdução
  - Clusters
    - Barramentos de comunicação dos *clusters* são exponencialmente mais lentos que barramentos de comunicação em multiprocessadores.
    - Memórias dos componentes de um *cluster* são utilizadas por  $n$  cópias de SOs, enquanto que em uma máquina multiprocessada, somente uma cópia do SO é executada.

# Clusters e Multiprocessadores de Trocas de Mensagens

- Introdução
  - Exemplo: Eficiência de memória
    - Considerando as seguintes informações:
      - Sistema 1: 1 multiprocessador com 20 GB de memória RAM;
      - Sistema 2: 1 *cluster* com 5 máquinas, cada uma contendo 4GB;
      - Em ambos os casos o SO ocupada 1GB;
      - Quanto de memória principal a mais terá o sistema 1 em comparação com o 2?
    - Sistema 1 terá disponível 25% mais memória que o sistema 2.

$$AvailableMemRate = \frac{MemUseSis1}{MemUseSis2} = \frac{20-1}{5 \times (4-1)} = \frac{19}{15} = 1.25$$

# *Hardware Multithreading*

# *Hardware Multithreading*

- Introdução
  - Permite que múltiplas *threads* compartilhem as unidades funcionais de um único processador.
  - Para que isso seja possível, algumas informações precisam ser armazenadas em cada uma das *threads* (p.ex.: PC).
  - *Hardware* precisa estar apto a escalar rapidamente entre as *threads*:
    - Enquanto a troca de processos leva centenas a milhares de ciclos de clock, a troca de *threads* deve ser feita de modo quase instantâneo.

# *Hardware Multithreading*

- Introdução
  - Três modelos principais de *hardware multithreading*:
    - Granulação fina (*fine-grained*)
      - Troca das *threads* acontece entre cada uma das instruções;
      - Execução das *threads* é feita de forma intercalada utilizando-se algoritmo de escalonamento *round robin*;
      - *Threads* que não estão prontas a executar são desprezadas na rodada;
      - Para tornar tal esquema possível, processador necessita conseguir mudar de *thread* a cada ciclo de *clock*.

# *Hardware Multithreading*

- Introdução
  - Três modelos principais de *hardware multithreading*:
    - Granulação fina (*fine-grained*)
      - **Vantagem:** Esquema esconde as perdas de *stalls* nas *threads*, uma vez que outras instruções (de outras *threads*) são executadas no lugar.
      - **Desvantagem:** Diminui a velocidade de execução de cada *thread* uma vez que todas as *threads* compartilham o mesmo espaço de tempo e a troca ocorre a cada ciclo de *clock*.

# *Hardware Multithreading*

- Introdução
  - Três modelos principais de *hardware multithreading*:
    - Granulação grossa (*coarse-grained*)
      - Troca das *threads* ocorre somente quando da execução de *stalls* maiores.
      - **Vantagem**: *Threads* podem executar por mais tempo - até que ocorra um *stall* grande.
      - **Desvantagem**: Se houver muitos *stalls* a vazão final pode ser prejudicada.

# *Hardware Multithreading*

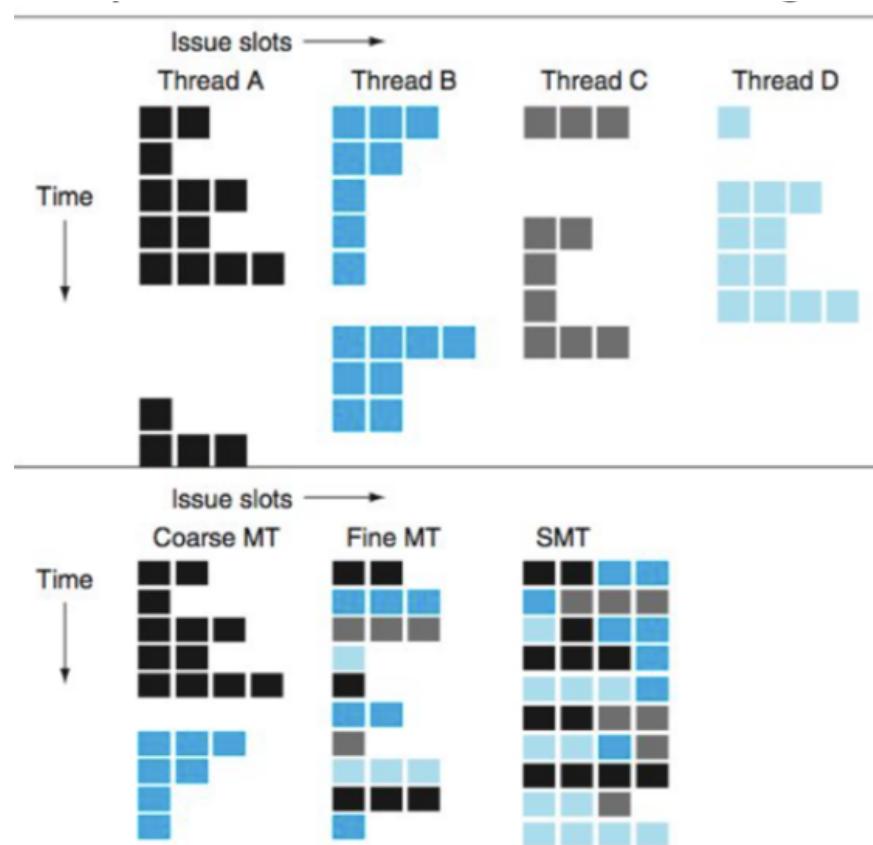
- Introdução
  - Três modelos principais de *hardware multithreading*:
    - Multithreading Simultâneo (*Simultaneous Multithreading SMT*)
      - É uma variação dos dois modelos apresentados anteriormente.
      - Utiliza os recursos de *multiple issue* e escalonamento dinâmico de processadores para conseguir ao mesmo paralelismo em nível de threads e instruções.
      - Conceito básico de SMT: Normalmente existem mais unidades funcionais disponíveis que uma única *thread* consegue utilizar.
        - Instruções de diferentes *threads* podem ser carregadas simultaneamente para execução.

# *Hardware Multithreading*

- Introdução
  - Três modelos principais de *hardware multithreading*:
    - Multithreading Simultâneo (*Simultaneous Multithreading SMT*)
      - Não existe a troca de *threads* em execução a cada ciclo: Ocorre execução de instruções de múltiplas *threads*. *Hardware* escolhe qual instrução associar a cada slot.

# *Hardware Multithreading*

- Introdução
  - Três modelos principais de *hardware multithreading*:
    - Exemplos



# **SISD, MIMD, SIMD, SPMD e Vetor**

# SISD, MIMD, SIMD, SPMD e Vetor

- Introdução
  - Desde 1960 existem outras caracterizações de *hardware paralelo*.
  - Caracterizações baseadas no número de fluxos de instruções e fluxos de dados.
    - Processadores mais simples possuem um único fluxo de instrução e outro de dados; (**SISD**)
    - Multiprocessadores comumente possuem vários fluxos de instruções e vários de dados. (**MIMD**)

# SISD, MIMD, SIMD, SPMD e Vetor

- Introdução
  - Principais caracterizações de acordo com nível de paralelismo:

		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Xeon e5345 (Clovertown)

\* SSE: Streaming SIMD Extension

- *Single Program Multiple Data (SPMD)*:
  - Único programa que executa em todos os processadores de um MIMD.

# SISD, MIMD, SIMD, SPMD e Vetor

- Introdução
  - Computadores **SIMD** operam em vetores de dados.
    - Ex.: Uma única instrução **SIMD** pode fazer 64 somas; envia-se os fluxos de dados a 64 ALUs diferentes para formar 64 somas em um único ciclo de *clock*.
    - Unidades de execução paralela estão sincronizadas e respondem a uma única instrução - de um único programa.
    - Do ponto de vista do programador, é como se fosse uma arquitetura **SISD**.
    - Em **SIMD** existe somente uma cópia do programa em execução; em **MIMD**, uma cópia para cada processador.

# SISD, MIMD, SIMD, SPMD e Vetor

- Introdução
  - Computadores SIMD operam em vetores de dados.
    - SIMDs funcionam melhor quando tratam dados de vetores em um *loop*.
    - Normalmente, para o paralelismo funcionar de maneira adequada, deve haver certo grau de similaridade na estrutura dos dados.
    - Pior caso para SIMD são estruturas de código do tipo *switch* e *if*, onde somente um dos fluxos é executado.
    - No caso do *switch/if*, somente uma unidade permanece ativa, enquanto todas as outras são desabilitadas.

# *Benchmarks*

# Benchmarks

- Introdução
  - *Benchmark* é um programa (ou conjunto de programas) que quando executados podem ser utilizados como referencial sobre o desempenho de um computador/sistema computacional.
    - Normalmente as medidas aferidas por um *benchmark* são baseadas no *hardware*.
    - Capacidade de efetuar um maior número de operações de ponto flutuante pode ser utilizada como base nas medições.
    - Resultados de um *benchmark* podem influenciar tanto tecnicamente quanto mercadologicamente.

# *Benchmarks*

- Introdução
  - *Benchmarks* para multiprocessadores
    - Quanto melhor os resultados de um *benchmark* mais sucesso o *hardware* tende a conseguir.
    - Algumas regras foram impostas para se assegurar que o código testado realmente faça aquilo que se propõe (e que não haja trapaça nas análises).
    - Muitos dos *benchmarks* possibilitam se aumentar o tamanho dos testes de acordo com o número de processadores na arquitetura avaliada.

# *Benchmarks*

- Introdução
  - *Benchmarks* para multiprocessadores
    - Exemplos:
      - *Linpack*: Coleção de rotinas de álgebra linear para cálculo de eliminação gaussiana.
      - *SPECrate*: Análise baseada na vazão. Além de programas individuais, avalia também a execução de várias cópias de um mesmo sistema paralelamente.
      - *SPLASH e SPLASH2*: Reúne uma suite de *kernel* + aplicativos para *benchmarking*.

# Benckmarks

- Introdução
  - *Benchmarks* para multiprocessadores
    - Exemplos:
      - **NAS**: Aplicado a ambientes paralelos, provê cinco *kernels* diferentes para avaliação. Ainda, possui suporte a weak scaling.
      - **PARSEC**: Aplicado a programas *multithread*, utilizam *pthread*s (*POSIX threads*) e *OpenMP* (*Open Multiprocessing*). Nove diferentes aplicações e três *kernels*.

# Dúvidas?

