Hardware paralelo

Processadores multicore

- Um processador com múltiplas unidades de processamento

Multiprocessadores

- Máquinas com vários processadores
- *Symmetric multiprocessing*
  - \+ Todos os processadores têm igual estatuto

Multiprocessadores heterogéneos

- Multiprocessadores com processadores de diferentes tipos
  - \+ CPU(s)
  - \+ GPU(s)
  - \+ *Many-core*(s) (dezenas ou centenas de cores)

Clusters

- Máquinas independentes
- Ligação por redes de baixa latência

Sistemas de memória partilhada

Shared memory processing (SMP)

É o normal nos sistemas multicore

Comum nos sistemas multiprocessador

Comunicação através de posições de memória (variáveis)

Comunicação é implícita

Coordenação/sincronização através da memória

Organização da memória em sistemas SMP

Uniform memory access (UMA)

- Todos os processadores acedem igualmente a toda a memória
- Competição no acesso à memória limita número de processadores

Non-uniform memory access (NUMA)

- Cada processador acede a uma zona da memória com menor latência que às outras
- Diminui o tráfego no *bus* de acesso à memória
- Permite mais processadores
- Programas devem ter em conta diferenças nos acessos à memória

Sistemas de memória distribuída

É o caso dos *clusters*

E de alguns multiprocessadores (raros e caros, como o Blue Gene da IBM, que pode ter até 2^{20} cores)

O processador só tem acesso à memória local

Comunicação por troca de mensagens

- Explícita
- Coordenação/sincronização por mensagens
  - + *Message-passing interface* (MPI): *standard* de uso comum

Memória partilhada distribuída (DSM)

- Permite o acesso directo à memória de outros nós do sistema
- Acesso não transparente
  - + Acesso a memória não local distinto do acesso à memória local

Compreender o funcionamento da máquina (3)

Como se explica?

```
#define VALS 500000000
long A[16];
void sum(long p)
{
  for (int i = 1; i <= VALS; ++i)
    A[p] = A[p] + i;
}
```

Execução	Tempo	Resultado
`sum(0)`	2.122 s	`A[0]` = 12500000250000000
`sum(0), sum(0)`	4.239 s	`A[0]` = 25000000500000000
`sum(0) | sum(8)`	2.125 s	`A[0]` = `A[8]` = = 12500000250000000
`sum(0) | sum(0)`	7.859 s	`A[0]` = 18612361288548600 0 ou 14236514494395715 6 ou ...
`sum(0) | sum(1)`	8.067 s	`A[0]` = `A[1]` = = 12500000250000000

Coordenação/sincronização entre processos

Sistemas SMP

CPU A	CPU B
	`x = 0;`
`x = x + 1;`	`x = x + 2;`

Qual o valor de `x` depois da execução deste código? 3? 2? 1? ...

Código MIPS

CPU A	CPU B
	`sw $0, x($0)`
`lw $t0, x($0)`	`lw $t0, x($0)`
`addiu $t0, $t0, 1`	`addiu $t0, $t0, 2`
`sw $t0, x($0)`	`sw $t0, x($0)`

Secção crítica

Zona do código em que é feita uma operação que não pode ser feita por mais do que um processador em simultâneo (como a modificação de uma estrutura de dados partilhada)

É necessário um mecanismo que permita garantir a exclusão mútua no acesso às secções críticas, *i.e.*, que só permita o acesso de um processo de cada vez a uma secção crítica

Exclusão mútua

Primeira tentativa

CPU A	CPU B
	`allow = 1;`
	`x = 0;`
`lock();`	`lock();`
`x = x + 1;`	`x = x + 2;`
`unlock();`	`unlock();`

```
void lock()
{
  while (allow == 0)
    ;
  allow = 0;
}

void unlock() { allow = 1; }
```

Está errada

Sofre do problema da versão inicial

Coordenação/sincronização em sistemas SMP

Feita através da memória partilhada

Requer a possibilidade de um processador realizar uma sequência de operações sobre a memória sem interferência por parte de outro processador, *i.e.*, de forma atómica

Primitivas para acessos atómicos à memória

- *Test-and-set* (instrução M680x0 `tas`)
  - \+ Escreve 1 na posição de memória
  - \+ Indica se o valor anterior era 0
- *Compare-and-swap* (instrução x86 `cmpxchg`)
  - \+ Compara conteúdo da posição de memória com um valor (*e.g.*, conteúdo antigo ou 0)
  - \+ Se são iguais, escreve 2º valor, senão lê o que está na memória
- *Load-linked/store-conditional* (instruções MIPS `ll` e `sc`)

Acessos atómicos à memória no MIPS

`ll rt, offset(rs)` (instrução *load linked*)

- Funciona como `lw`
- Associa uma marca ao endereço acedido

`sc rt, offset(rs)` (instrução *store conditional*)

Se for executada no mesmo processador em que foi executado o `ll`, sobre o mesmo endereço, e a marca ainda lhe está associada

- Escreve o conteúdo de `rt` no endereço pretendido
- Remove a marca
- Põe o valor `1` em `rt`

Caso contrário

- Põe o valor `0` em `rt`

Qualquer escrita no endereço usado remove a marca, mesmo se efectuada noutro processador

Exclusão mútua

Segunda tentativa

Recorrendo às instruções MIPS que permitem acessos atómicos à memória

```
lock: ll   $3, allow($0)     # lê valor de allow
      beq  $3, $0, lock      # se for 0, tenta outra vez
      addi $3, $3, -1        # decrementa o valor
      sc   $3, allow($0)     # escreve o novo valor
      beq  $3, $0, lock      # se falhou, volta a tentar
      jr   $ra
```

A instrução `sc` (*store conditional*) falha se o bloco que contém a posição de memória lida por `ll` (*load linked*) foi escrito depois da execução de `ll`

Quando sucede, `sc` põe o valor `1` no registo `rt`; se falha põe lá o valor `0`

Cache em sistemas de memória partilhada

O problema

CPU A	CPU B
	`X = 0;`
`y = X;`	`z = X;`
`X = y + 1;`	

Qual a visão com que A e B ficam sobre o conteúdo da 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

Os conteúdos das caches de A e B representam uma visão não coerente do conteúdo da memória

Cache em sistemas de memória partilhada

Uma solução

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

Protocolo com *snooping*

- Todos os processadores se mantêm atentos ao *bus* de acesso à memória (*snooping*)
- Quando um quer escrever num bloco partilhado, envia uma mensagem de *write invalidate* para o *bus*
- Todas as cópias nas outras caches são marcadas como inválidas
- Se outro processador tentar ler um bloco modificado localmente, envia-lhe esse bloco, que passa a estar partilhado

Cache em sistemas de memória partilhada

Outra solução

Protocolos baseados num directório

- Existe um directório (uma lista) que contém informação sobre o estado de todos os blocos nas caches
- Todos os acessos à memória são feitos através deste directório

Comparação

O uso do directório torna os acessos mais lentos

Gera menos tráfego entre processadores que o uso de *snooping*

Permite maior número de processadores

Cache em sistemas de memória partilhada

False sharing

Há *false sharing* (ou falsa partilha) quando dois processadores acedem a posições de memória diferentes do mesmo bloco

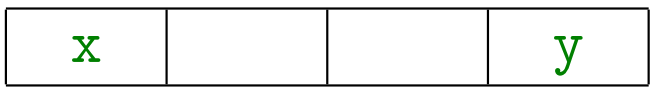
CPU A

```
for (...)
  {
    ...
    x = ...;
    ...
  }
```

CPU B

```
for (...)
  {
    ...
    y = ...;
    ...
  }
```

Bloco com 4 palavras



Sempre que um processador altera o valor da sua variável, a cópia do bloco na cache do outro é invalidada

Cabe ao programador evitar a ocorrência de *false sharing* no programa

Hardware multithreading

Um processador (core) alterna entre a execução de vários processos

O custo da mudança de contexto efectuada para executar um novo processo é da ordem das centenas ou milhares de ciclos (têm de ser guardados os valores nos registos e repostos os do processo cuja execução é retomada)

Se o processador tiver várias cópias dos registos, a mudança de contexto pode ser instantânea

Num processador com *hardware multithreading*, vários processos são executados em simultâneo

- Em cada ciclo, pode haver instruções de mais do que um processo em execução (possivelmente, em diferentes andares do *pipeline*)
- A partilha do processador é feita sem a intervenção do SO

Fine-grained multithreading

O processador lança instruções de um processo diferente em cada ciclo, seguindo uma estratégia *round-robin*

A mudança de contexto tem de ser *muito* rápida

Todos os atrasos que ocorrem na execução de um processo são aproveitados na execução de outro

Aumenta o *throughput* do processador

Atrasa a execução de processos que não teriam atrasos

Coarse-grained multithreading

O processador lança instruções de um processo diferente sempre que é necessário atrasar o processo em execução um número significativo de ciclos (se há um *miss* no nível mais baixo da cache, por exemplo)

Esconde a ocorrência de atrasos longos

O *pipeline* é limpo quando outro processo é executado

A mudança de contexto pode ser mais lenta

Aumenta menos o *throughput* do processador (não esconde atrasos curtos)

Não atrasa a execução de processos que não teriam atrasos

Simultaneous multithreading (SMT)

Em cada ciclo, o processador lança instruções de diferentes processos

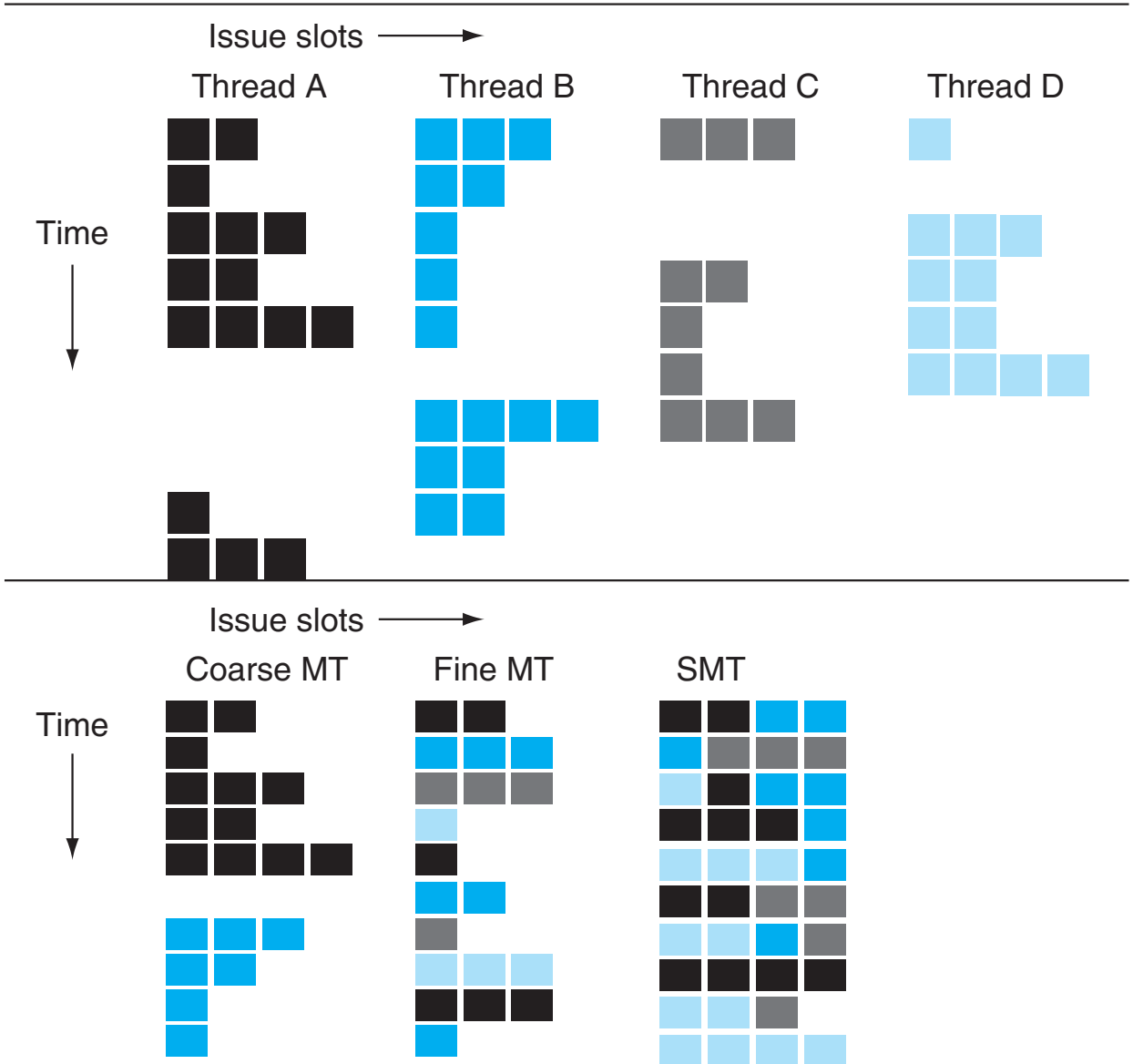
Processadores superescalares

Permite maior utilização das unidades funcionais do processador

Conhecido como hyperthreading nos processadores Intel

Hardware multithreading

Processador superescalar



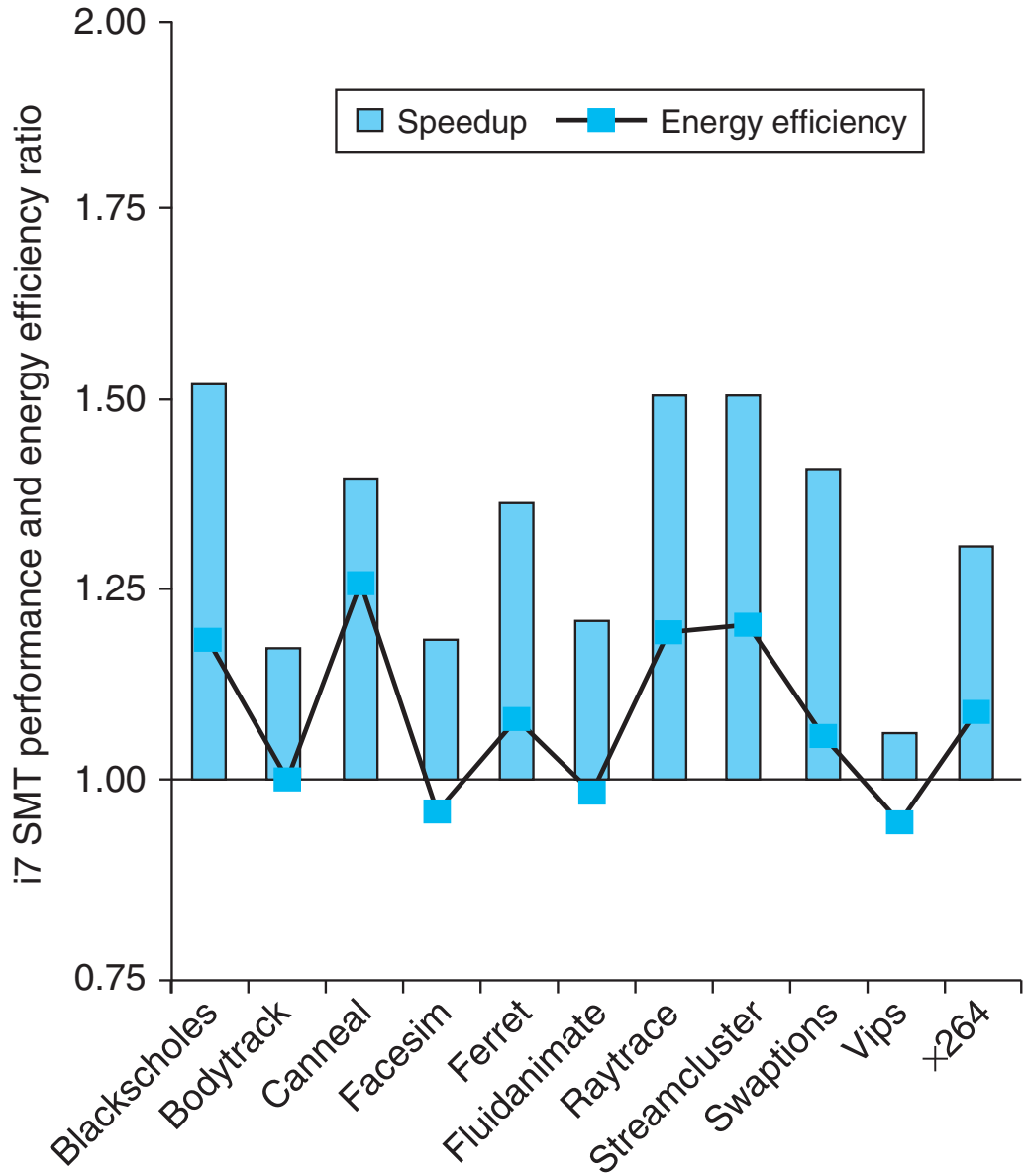
(Cada ■ representa uma instrução)

Eficácia do SMT no Intel Core i7 960

Suporta 2 *threads* por core

Speedup médio de 1.31

Eficiência energética aumenta 1.07, em média



Benchmarks PARSEC, para programas *multithreaded*