


Super Escalaridade (ILP - Instruções level parallelism)

Pipelining: Executar várias instruções em simultâneo em estágios diferentes

→ Permite: aumentar a frequência

→ CPI ideal = 1 → difícil de manter

↳ dependências de dados

↳ " " " controlo

↳ atrasos nos acessos à memória

Multiple Issue: múltiplos pipelines (múltiplas unidades funcionais → exec. simultânea de múltiplas instruções)

↳ exemplo: ALU1MUL e ALU e LS

↳ potencialmente podemos ter CPI < 1

Limitações:

↳ dependências de dados e/o controlo

↳ disponibilidade de recursos (nº de unidades funcionais, latências das instruções, atrasos no acesso à memória)

Static Multiple Issue

↳ compilador agrupa empacota as inst. que serão lançadas em cada ciclo do relógio

↳ pode ser visto como uma longa instrução que comporta tantas inst. como pipelines

Compilador:

VLIW

↳ garantir que não há dependências dentro de uma VLIW e entre as outras VLIW

↳ que as inst. de uma VLIW possam ser executadas pelas unidades funcionais

- Considere uma versão com *static multiple issue* do MIPS, com 2 pipelines:

- Pipe 0: ALU ou branch
- Pipe 1: load / store

Loop:
lw \$t0, 0(\$s1)
addu \$t0, \$t0, \$s2
sw \$t0, 0(\$s1)
addi \$s1, \$s1, -4
bne \$s1, \$zero, Loop

CC	Pipe 0 (ALU/Branch)	Pipe 1 (LS)
1	nop	lw \$t0, 0(\$s1)
2	addu \$t0, \$t0, \$s2	nop
3	addi \$s1, \$s1, -4	sw \$t0, 0(\$s1)
4	bne \$s1, \$zero, Loop	addi \$s1, \$s1, -8
CPI	4/5 = 0.8	

Loop Unrolling - Exemplo: MIPS

Loop:
lw \$t0, 0(\$s1)
addu \$t0, \$t0, \$s2
sw \$t0, 0(\$s1)
lw \$t1, -4(\$s1)
addu \$t1, \$t1, \$s2
sw \$t1, -4(\$s1)
lw \$t2, -4(\$s1)
addu \$t2, \$t2, \$s2
sw \$t2, -4(\$s1)
bne \$s1, \$zero, Loop

anti-dependência -> register renaming

CC	Pipe 0 (ALU/Branch)	Pipe 1 (LS)
1	nop	lw \$t0, 0(\$s1)
2	addu \$t0, \$t0, \$s2	lw \$t1, -4(\$s1)
3	addu \$t1, \$t1, \$s2	sw \$t0, 0(\$s1)
4	addi \$s1, \$s1, -8	sw \$t1, -4(\$s1)
5	bne \$s1, \$zero, Loop	nop
CPI	5/8 = 0.625	

Loop Unrolling:

↳ fazer múltiplas cópias do corpo de um ciclo

↳ nem sempre positivo para um desempenho

↳ contribui para:

↳ + inst. independentes, ILP ↑ e CPI ↓

Static Multiple Issue - Limitações:

↳ código gerado depende da organização do processador

↳ diferentes implementações → recompilação do código

Dynamic Multiple Issue

↳ processador seleciona dinamicamente quais inst. a executar em cada pipeline (unidade funcional)

↳ dependência de compilação

↳ só o hardware é responsável

Super Escalaridade: in-order scheduling

↳ inst. selecionadas na ordem com que aparecem no programa, baseado na disponibilidade dos operandos e da unidade funcional apropriada

Super Escalaridade: out-of-order scheduling

↳ inst. são reordenadas pelo processador para minimizar o CPI, garantindo a correta exec. do programa

ILP: limitações

↳ Difícil lances mais do que 2 inst. por ciclo → 3 dependências de controle e dados

↳ penalização de acesso à memória → limita capacidade de manter os pipelines ocupados

Processamento Vectorial

escalar → processa 1 elemento do conjunto de dados

vectorial → cada inst. processa N elementos do conjunto de dados

↳ data level parallelism

$$T_{exec} = CPI * \#I/f$$

diminui → cada Inst. processa N elementos

→ tende a aumentar → principalmente por causa da quantidade de dados a transferir de e para a memória

loop unrolling → geralmente: #instruções = 3 e CPI = 1 (e Texec = 3)

auto-vectorização (pelo compilador): #instruções = 16 e CPI = 1 → Texec = 16 (op. vectoriais = 16)

vectorização + loop unrolling: #instruções = 3 e CPI = 1 e Texec = 3 (op. vectoriais = 3)

vectorização (pelo programador): #instruções = 16 e CPI = 16 → Texec = 16 (op. vectoriais = 16) → mov vectorial → int passado

vectorização (pelo compilador) + loop unrolling: #instruções = 16 e CPI = 16 → Texec = 16 (op. vectoriais = 16)

CPI ↑ → por causa de sobrecarregar a memória

128 bits		Tipos de Dados
xmm0		2 Packed Double-Precision Floating-Point Values
xmm1		16 Packed Byte Integers
xmm2		8 Packed Word Integers
xmm3		4 Packed Doubleword Integers
xmm4		2 Quadword Integers
xmm5		Double Quadword
xmm6		4 Packed Single-Precision Floating-Point Values
xmm7		

instruções AVX:

• [V] MOV [A | U] P [S | D]

Mover quantidades de 128 ou 256 bits (prefixo V), representando valores SPFP ou DPFP (sufixo S ou D), de endereços alinhados ou não (modificador A ou U)

→ 128 bits
- VMOVUPD m256, %ymm? → move 4 DPFP de memória (endereço não alinhado) para %ymm?

- VMOVAPD %xmm?, m128 → move 2 DPFP de %xmm? para memória (endereço alinhado)

• **Alinhamento:** um bloco de dados com B bytes, diz-se alinhado, se o endereço inicial desse bloco em memória é múltiplo de B.

• Acessos alinhados são **significativamente** mais eficazes do que acessos não alinhados.

Instruções	Operandos
[V]ADD[S P][S D]	Sem V? : xmm/m128, xmm
[V]SUB[S P][S D]	Com V? : ymm, ymm/m256, ymm
[V]MUL[S P][S D]	→ todos os tipos mixed significativos
[V]DIV[S P][S D]	[S P] : escalar ou vectorial ?
[V]SQRT[S P][S D]	
[V]MAX[S P][S D]	[S D] : SPFP ou DPFP ?
[V]MIN[S P][S D]	→ 16 bytes
[V]AND[S P][S D]	Endereços em memória alinhados
[V]OR[S P][S D]	
...	O resultado não pode ser em memória

float * - restrict - a

↳ para evitar aliasing → regiões de memória apontadas pelas diferentes apontadores podem - se sobrepor
↳ não há qualquer outra referência para essa zona de memória

Blockadores Auto-vectorização

→ dados não contíguos → Array of structures



↳ elementos do mesmo campo, não são armazenados consecutivamente em memória

↳ structures of array → vetorizável → armazenados consecutivamente

→ stride != 1 → se o incremento for ≠ 1 → acessos não contíguos

↳ ordenados
↳ mesmo que se consiga vetorizar
↳ não é eficiente
↳ localidade espacial

→ ch countable loops → nº de iterações do ch loop não pode ser computado

→ condições → estruturas condicionais (if else)

↳ máscaras → ($i < 0 \ ? \ s : 0$) → vetorizável

→ funções → não vetorizável

↳ funções intrínsecas → vetorizável

↳ não vetorizável

→ dependências read after write (RAW) ($a[i] = a[i-1] + 1$ → $i - i+1 = 1 \Rightarrow$ RAW X)

↳ dependências write after read (WAR) ($a[i] = a[i+1] + 1$ → $i - i-1 = -1 \Rightarrow$ WAR ✓)

↳ indica a escrita → vetorizável

↳ $d = C^W - C^R$ → índice de escrita - índice de leitura

or $d = -(C^W - C^R)$ → se círculo for decrementado

↳ se $d \leq 0$ → não há depênd RAW → círculo vetorizado

→ máquinas AVX → largura das unidades funcionais $W = 8$

$a[i] = a[i-5] + 1$

↳ $d = i - (i-5) = 5 \Rightarrow d \geq W \Rightarrow$ não vetorizável

Multicore

$$P = C \cdot V \cdot f$$

↳ tensão
↳ frequência
↳ capacidade
↳ potência dinâmica

Thread Level Parallelism

- Paralelismo a um nível mais grosso do que as instruções de um programa:

- explorar o paralelismo entre diferentes fios de execução (threads) de um mesmo programa ou de programas diferentes
- as threads podem ser partes de um programa paralelo ou mesmo programas diferentes (processos)
- cada thread tem o seu estado (instruções, dados, contexto (conteúdo de registos, instruction pointer, etc.)) para que possa executar independentemente

thread 0

```
for (i=0; i < S/2; i++)  
{ processa a[i]; }
```

thread 1

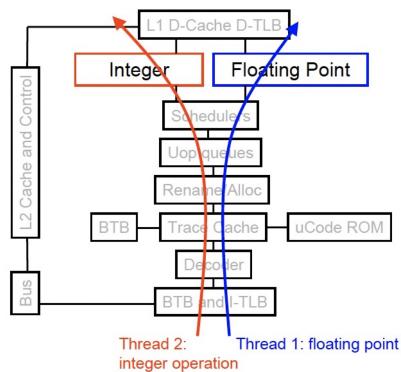
```
for (i=S/2; i < S; i++)  
{ processa a[i]; }
```

Thread Level Parallelism

- ILP explora paralelismo implícito numa única sequência de instruções
 - TLP explora paralelismo explícito entre múltiplas sequências de instruções
- “O código deve ser escrito explicitamente para expor TLP”
“the free lunch is over”
- Objectivo:
 - aumentar o débito em computadores que executam vários programas
 - diminuir o tempo de execução de programas paralelos (multithreaded)

Simultaneous Multi Threading

- Com SMT ambas as threads podem executar simultaneamente

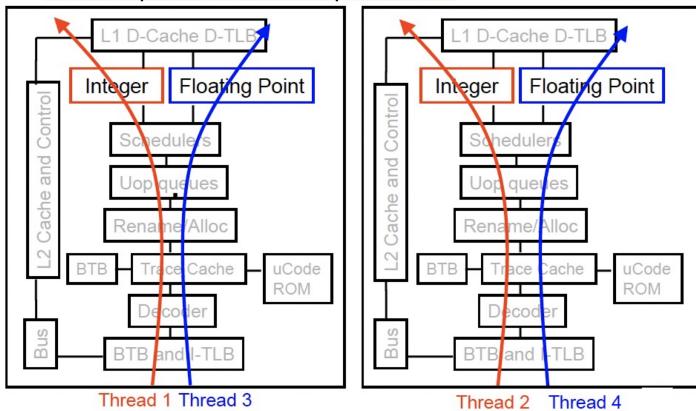


AC – Arquitecturas MultiCore

14

Multi Core Chip e SMT

- Os cores podem ou não suportar SMT



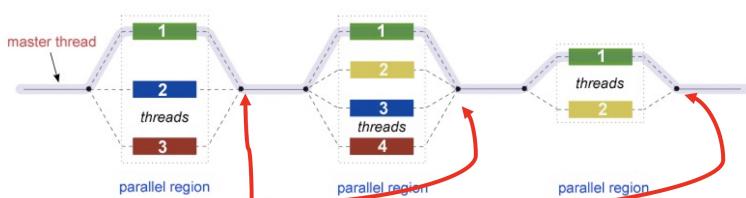
AC – Arquitecturas MultiCore

21

Open MP

- API para expressar paralelismo *multi-threaded* e de memória partilhada
- Criação explícita de blocos paralelos de código executados por um grupo (*team*) de threads

Modelo Fork & Join



- No final de cada bloco:
 - todas as threads sincronizam (barreira implícita)
 - todas as threads excepto a principal deixam de existir

- Paralelismo especificado usando directivas embutidas no código

```
#pragma omp <nome directiva> [cláusula, ...]
```

- Cada directiva aplica-se ao bloco de instruções que se lhe segue

```
#pragma omp parallel
{
    ... // bloco paralelo
}
... // bloco sequencial
```

directiva parallel

```
#pragma omp parallel
{
    ... // bloco paralelo
}
```

- cria um grupo (*team*) de N threads
- cada uma destas threads executa independentemente o bloco paralelo
- **no fim do bloco existe uma barreira (sincronização) implícita:**
 - a *thread* principal só continua depois de todas as outras
 - também terão chegado ao fim do bloco

#include <omp.h>	
Função	Descrição
int omp_get_thread_num (void)	Devolve ID da thread
int omp_get_num_threads (void)	Devolve número de threads actualmente existentes num bloco paralelo
void omp_set_num_threads (int)	Estabelece número de threads a ser criadas no próximo bloco paralelo
int omp_get_num_procs (void)	Devolve número de processadores disponíveis para o programa
double omp_get_wtime (void)	Devolve um <i>time stamp</i> em segundos
... e muitas mais ...	

directiva single

Apenas a primeira *thread* a atingir o bloco *single* o executa

Todas as *threads* sincronizam no fim do bloco (barreira implícita)

parallel – ordem de execução

```
#include <omp.h>
#pragma omp parallel num_threads(2) {
    printf("Thread %d...\n",omp_get_thread_num());
    #pragma omp single nowait
        printf("Há %d threads!\n",omp_get_num_threads ());
    printf("...thread %d\n",omp_get_thread_num());
}
printf("program done\n");
```

nowait elimina a barreira no fim do bloco *single*

>./prog	Thread 1...	>./prog
Há 2 threads!	Há 2 threads!	Thread 0...
Thread 1...	...thread 1	Há 2 threads!
Thread 0...	Thread 0...	Thread 1...
...thread 0	...thread 0	...thread 0
...thread 1	...thread 1	...thread 1
program done	program done	program done
>_	>_	>_

Loop construct : directiva `for`

```
#pragma omp parallel num_threads(2)
{
#pragma omp for
    for (i=0; i<N; i++) A[i] = B[i] + C[i]; }

• deve estar dentro de um bloco parallel
• distribui as iterações do ciclo pelas threads activas no grupo (team) actual:
    - o espaço de iterações (i=0 .. N-1, neste exemplo) é decomposto em sub-intervalos (chunks) consecutivos;
    - os chunks são distribuídos pelas threads
    - sem informação adicional nada se pode assumir sobre o número ou tamanho dos chunks, nem sobre a sua distribuição pelas threads, excepto de que cada thread será responsável pela execução de pelo menos 1 chunk
• for e parallel podem ser combinadas
```

```
#pragma omp parallel for num_threads(2)
for (i=0; i<N; i++) A[i] = B[i] + C[i];
```

$$T_{exec} = \#I * CPI / f$$

- Com a programação *multithreaded* o número de instruções executadas aumenta, devido à gestão do paralelismo
- Como medir o CPI?

	Texec	CPI	# instruções	Speedup	Eficiência
OpenMP: #pragma omp parallel → divide o código em várias threads	com 1 thread: 344 msecs	0,49	640	1,0	100%
	com 2 threads: 288 msecs (1)	0,25	320	1,23	96,5%
	com 4 threads: 194 msecs (1)	0,20	160	2,0	85%
	com 8 threads: 92 msecs (1)	0,10	80	3,7	74%
	com 16 threads: 49 msecs (1)	0,05	40	6,0	60%

	Texec	CPI	# instruções	Speedup	Eficiência
OpenMP + vectorizado:	com 1 thread: 220 msecs	0,42	4620	1,0	100%
	com 2 threads: 160 msecs (1)	0,21	2300	1,30	86%
	com 4 threads: 96 msecs (1)	0,10	1150	2,20	78%
	com 8 threads: 48 msecs (1)	0,053	575	4,0	67%
	com 16 threads: 24 msecs (1)	0,027	287	8,0	58%

↓/ aumento de sobreengarantia sistemática na memória

AC – OpenMP

Desempenho: como medir o CPI

- CPI por processador *p*

Tende a manter-se constante com a introdução de múltiplos cores

$$CPI_p = \frac{\#cc_p}{\#I_p}$$

- CPI global

Tende a manter-se constante com a introdução de múltiplos cores

$$CPI = \frac{\sum_{p=0}^{P-1} \#cc_p}{\sum_{p=0}^{P-1} \#I_p}$$

- CPI percepcionado

(pelo utilizador)

Tende a diminuir com a adição de múltiplos cores, se o tempo de execução diminuir

$$CPI_{perceived} = \frac{\max(\#cc_p)}{\sum_{p=0}^{P-1} \#I_p}$$

desempenho

$$T_{exec} = \#I * CPI_{perceived} / f$$

- Com a programação *multithreaded* o número de instruções executadas aumenta, devido à gestão do paralelismo

- O $CPI_{perceived}$ diminui com o número de cores

– Esta diminuição compensa largamente o aumento de $\#I$ levando a diminuições muito significativas de T_{exec}

– A taxa de diminuição do $CPI_{perceived}$ reduz à medida que o número de cores aumenta

- Os dados podem ser **partilhados** ou **privados**
- Dados partilhados dentro de um grupo são acessíveis a todas as *threads* desse grupo
- Dados privados são acessíveis apenas à *thread* que os possui
- Por omissão os dados são **partilhados**, i.e.,

as **variáveis globais** a um bloco paralelo são **partilhadas**

- **Variáveis privadas:**
 - declaradas dentro de um bloco paralelo
 - explicitamente marcadas como privadas
 - índices dos ciclos associados a uma directiva `for`

directiva `parallel` – *data scope*

Variáveis globais a um bloco paralelo são partilhadas por omissão

```
main () {
    int tid;
    #pragma omp parallel
    {
        tid = omp_get_thread_num();
        printf("Thread %d\n", tid);
    }
    printf("program done\n");
}
```

variável partilhada:
as **várias threads** **escrevem e lêem em** **qualquer ordem, sem** **controlo de acesso**

Resultado indeterminado

directiva `parallel` – *data scope*

- São variáveis privadas:
 - locais ao bloco
 - explicitamente declaradas com a cláusula `private(...)`
 - os índices dos ciclos abrangidos pela directiva `for`
- ```
#pragma omp parallel
{ int i;
 ...
}

#pragma omp parallel private (x)
int x;
#pragma omp parallel private (x)
for (i=0; i<N; i++)
 A[i] = B[i] + C[i];
```

### Cláusula **private**: variável privada

```
main () {
 int tid;
```

cada *thread* tem a sua própria  
instância local de **tid**

```
#pragma omp parallel private (tid)
{
 tid = omp_get_thread_num();
 printf("Thread %d\n",tid);
}

printf("program done\n");
}
```

### Declaração dentro do bloco: variável privada

```
main () {
 int tid;
```

cada *thread* tem a sua própria instância local  
de **tid**

```
#pragma omp parallel
{
 int tid;

 tid = omp_get_thread_num();
 printf("Thread %d\n",tid);
}

printf("program done\n");
}
```

directiva **critical**: apenas uma *thread* executa esse bloco em cada instante.

```
int x=0;
#pragma omp parallel
 #pragma omp critical
 x = x+1;
```

Se uma *thread* está dentro de uma região crítica,  
então nenhuma outra *thread* entra nessa região até a *thread* anterior sair:  
-> a execução das regiões críticas não acontece em paralelo, é **sequencial**

```
int x=0, y=0;
#pragma omp parallel
{
 #pragma omp critical
 x = x+1;
 #pragma omp critical
 y = y+1; }
```

As regiões críticas sem nome são  
consideradas a mesma região!  
Se uma *thread* está em **x = x+1**,  
então nenhuma outra *thread*  
entra em **y = y+1**  
e vice-versa

Duas regiões críticas distintas.  
**x = x+1** e **y = y+1**  
podem acontecer em paralelo

```
int x=0, y=0;
#pragma omp parallel
{
 #pragma omp critical C1
 x = x+1;
 #pragma omp critical C2
 y = y+1; }
```

directiva `atomic`: garante que um endereço de memória é acedido de forma atómica. Pode ser vista como uma versão leve de `critical`.

Só se aplica a operações simples (atómicas).

Não garante que o lado direito da atribuição é avaliado de forma atómica.

- designa-se por **redução** uma operação que processa um conjunto de dados para a partir dele gerar um único valor, exemplo, a soma/máximo/produto de todos os elementos de um vector

```
int a[SIZE];
... inicializar a[]
int max=a[0];
#pragma omp parallel for
{
 for (i=0; i< SIZE ; i++)
 if (a[i]>max) max=a[i]; }
```

max é uma variável partilhada

- A redução é tão comum que o OpenMP inclui uma cláusula específica (mas não para o máximo)

```
int a[SIZE];
... inicializar a[]
int sum=0;
#pragma omp parallel for reduction (+:sum)
{
 for (i=0; i< SIZE ; i++)
 sum += a[i]; }
```

A cláusula `reduction` aplica-se apenas a operações associativas.

| C / C++                                                   |
|-----------------------------------------------------------|
| <code>x = x op expr</code>                                |
| <code>x = expr op x (excepto subtracção)</code>           |
| <code>x binop= expr</code>                                |
| <code>x++; ++x</code>                                     |
| <code>x--; --x</code>                                     |
| <code>x – variável escalar</code>                         |
| <code>expr – expressão escalar que não refere x</code>    |
| <code>op – +, *, -, /, &amp;, ^,  , &amp;&amp;,   </code> |
| <code>binop - +, *, -, /, &amp;, ^,  </code>              |

- As operações sobre operandos em vírgula flutuante (`float`, `double`) não são associativas: → por causa dos arredondamentos

```
float a[SIZE], sum=0.0;
... inicializar a[]
#pragma omp parallel for reduction (+:sum)
{
 for (i=0; i< SIZE ; i++)
 sum += a[i];
 printf ("sum= %.1f\n", sum)
```

```
>./prog
sum= 1233458.0
>./prog
sum= 1233463.0
>./prog
sum= 1233457.0
```

### desempenho – *speed up*

$$S_p = \frac{T_1}{T_p}$$

$p$  – número de processadores  
 $T_1$  – tempo de execução  $p=1$   
 $T_p$  – tempo de execução  
com  $p$  processadores

- indica quantas vezes mais rápida é a versão paralela com  $p$  processadores relativamente à versão sequencial

### desempenho – eficiência

$$E_p = \frac{S_p}{p}$$

$p$  – número de processadores  
 $S_p$  – *speed up* com  $p$  processadores

- indica em que medida estão os  $p$  processadores a ser bem utilizados
- Razão entre o *speed up* observado e o ideal ( $=p$ )
- A utilização total efectiva dos processadores resultaria numa eficiência de 100%