

Chapter 6

Parallel Processors from Client to Cloud

Introduction

- Multiprocessor
 - Goal: connecting multiple computers to get higher performance
 - Scalability, availability, power efficiency
- Multicore processors
 - Chips with multiple processors (cores)
- Task-level (process-level) parallelism
 - High throughput for independent jobs
- Parallel program (parallel software)
 - A single program run on multiple processors
 - Challenges: partitioning, coordination, communications overhead

Amdahl's Law

- Sequential part can limit speedup
- Example: 100 processors, 90x speedup?
 - $T_{\text{new}} = T_{\text{parallelizable}}/100 + T_{\text{sequential}}$
 - $$\text{Speedup} = \frac{1}{F_{\text{parallelizable}}/100 + (1 - F_{\text{parallelizable}})} = 90$$
 - Solving: $F_{\text{parallelizable}} = 0.999$
- Need sequential part to be 0.1% of original time

Scaling Example


- Workload
 - sum of 10 scalars (sequential)
 - sum of a pair of 10×10 matrix (parallel)
 - What's the speed up from single to 10 and 100 processors?
- Single processor:
 - $\text{Time} = (10 + 100) \times t_{\text{add}}$
- 10 processors
 - $\text{Time} = 10 \times t_{\text{add}} + 100/10 \times t_{\text{add}} = 20 \times t_{\text{add}}$
 - $\text{Speedup} = 110/20 = 5.5$ (55% of potential)
- 100 processors
 - $\text{Time} = 10 \times t_{\text{add}} + 100/100 \times t_{\text{add}} = 11 \times t_{\text{add}}$
 - $\text{Speedup} = 110/11 = 10$ (10% of potential)



Assumes load balanced
across processors

Scaling Example (cont)

- What if matrix size is 100×100 ?
- Single processor:
 - $\text{Time} = (10 + 10000) \times t_{\text{add}}$
- 10 processors
 - $\text{Time} = 10 \times t_{\text{add}} + 10000/10 \times t_{\text{add}} = 1010 \times t_{\text{add}}$
 - $\text{Speedup} = 10010/1010 = 9.9$ (99% of potential)
- 100 processors
 - $\text{Time} = 10 \times t_{\text{add}} + 10000/100 \times t_{\text{add}} = 110 \times t_{\text{add}}$
 - $\text{Speedup} = 10010/110 = 91$ (91% of potential)



Assumes load balanced
across processors

Instruction and Data Streams

- An alternate classification on parallel hardware

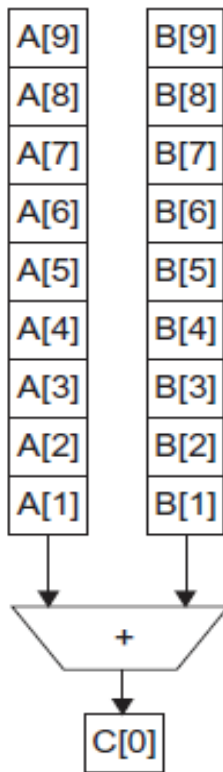
		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD (vector processor): SSE instructions of x86
	Multiple	MISD: No example yet	MIMD : Intel Xeon e5345

Vector architecture

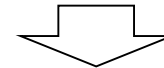
- Highly pipelined function units
- Stream data from/to **vector registers** to units
 - Data collected from memory into vector registers
 - Results stored from vector registers to memory
- Example: Vector extension to MIPS
 - 32 vector registers, each has 64 64-bit elements
 - Vector instructions
 - `lv, sv`: load/store vector
 - `addv.d`: add vectors of double
 - `addvs.d`: add scalar to each element of vector of double

Vector architecture

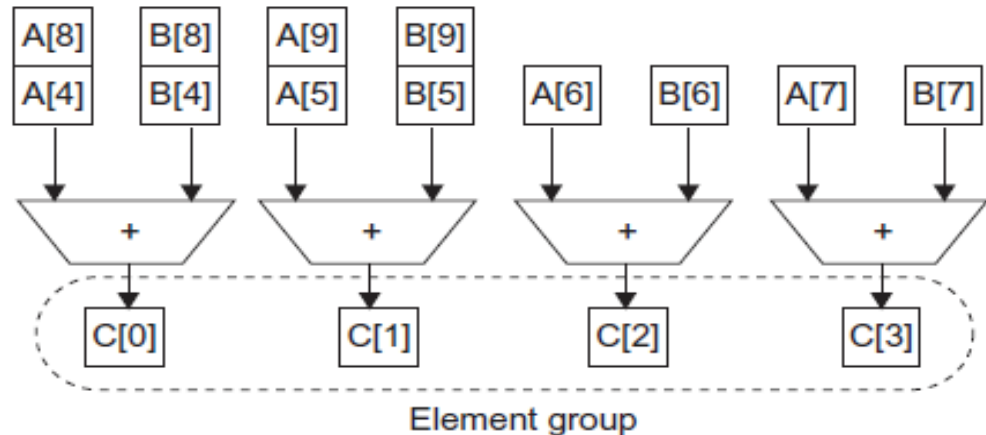
- **Single add pipeline**
 - Complete one addition per cycle



An array of parallel functional units



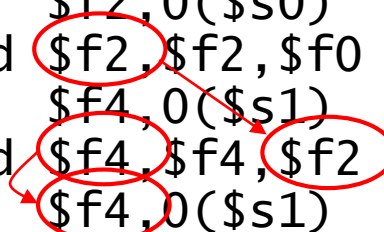
- **Four add pipeline**
 - Complete four additions per cycle



Example: DAXPY ($Y = a \times X + Y$)

■ Conventional MIPS code

```
l.d    $f0,a($sp)      ;load scalar a
addiu  r4,$s0,#512      ;bound of what to load
loop:  l.d    $f2,0($s0) ;load x(i)
mul.d  $f2,$f2,$f0      ;a x x(i)
l.d    $f4,0($s1)      ;load y(i)
add.d  $f4,$f4,$f2      ;a x x(i) + y(i)
s.d    $f4,0($s1)      ;store into y(i)
addiu  $s0,$s0,#8       ;increment index to x
addiu  $s1,$s1,#8       ;increment index to y
subu   $t0,r4,$s0       ;compute bound
bne    $t0,$zero,loop   ;check if done
```



■ Vector MIPS code

```
l.d    $f0,a($sp)      ;load scalar a
lv     $v1,0($s0)      ;load vector x
mulvs.d $v2,$v1,$f0     ;vector-scalar multiply
lv     $v3,0($s1)      ;load vector y
addv.d $v4,$v2,$v3     ;add y to product
sv     $v4,0($s1)      ;store the result
```

Vector vs. Scalar architecture

- Reduce Instruction fetch and decode
 - A single vector instruction is equivalent to executing an entire loop.
- Avoid data hazard checking
 - Only check data hazard between vectors, not for every element within the vectors (computation of every element within the same vector is independent).
- Avoid control hazard
 - An entire loop is replaced by a vector instruction → loop branch (leading to control hazard) is non-existent.
- Efficient memory access
 - If the vector's elements are all adjacent in memory, fetching them is efficient using interleaved memory banks.

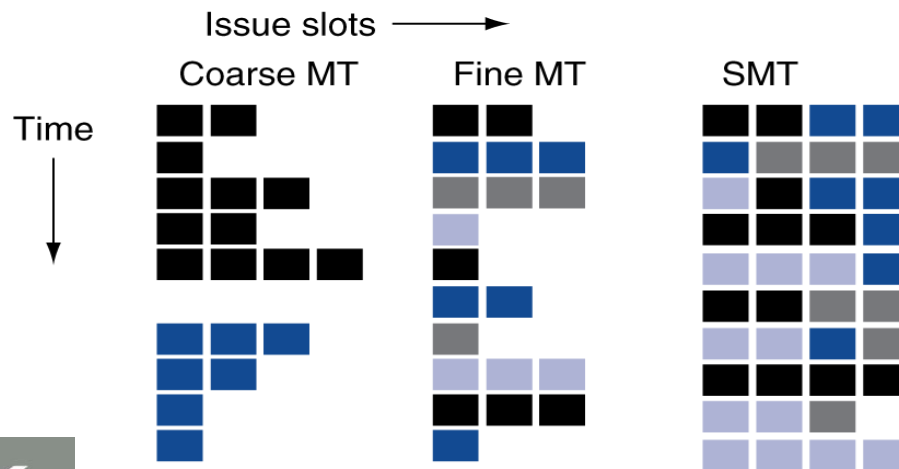
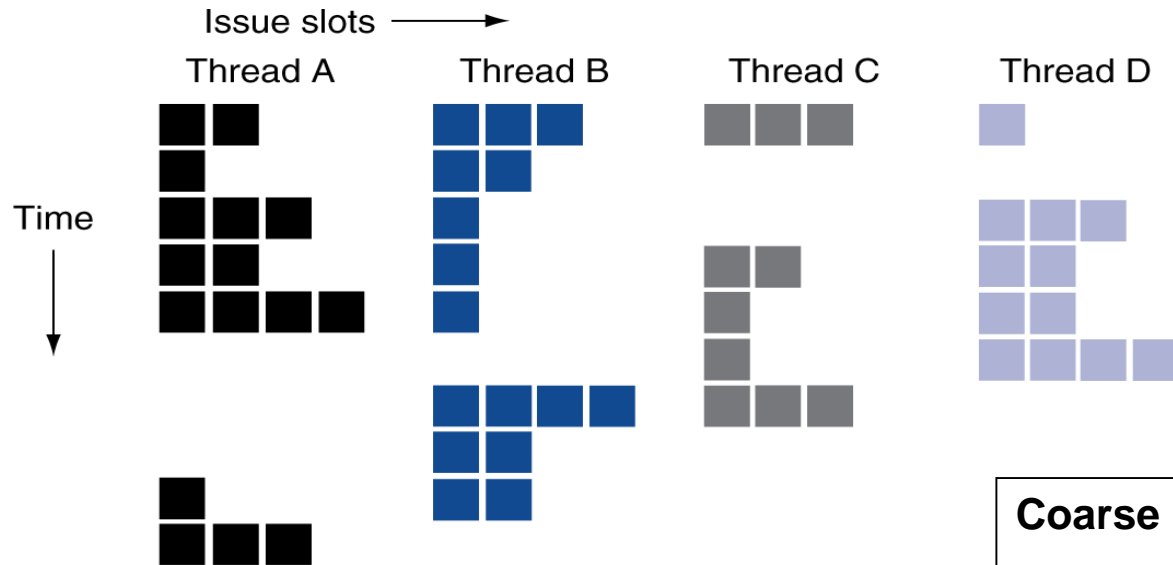
Hardware Multithreading

- Hardware Multithreading
 - Multiple **hardware threads** (replicate registers, PC, etc.)
 - **fast context switching** between threads
- **1. Fine-grain multithreading**
 - Switch threads after execution of each instruction
 - If one thread stalls (long and short), others are executed
 - Slow down the execution of individual threads, especially those without stalls
- **2. Coarse-grain multithreading**
 - Only switch on long stall (e.g., L2-cache miss)
 - Simplifies hardware, but doesn't hide short stalls (eg, data hazards)

Simultaneous Multithreading (SMT)

- **3. Simultaneous multithreading (SMT)**
 - in *multiple-issue dynamically scheduled pipelined CPU*
 - Motivation
 - Multiple-issue processors often have more functional units available than single thread needs to use.
 - Schedule instructions from multiple threads
 - Instructions from independent threads execute when function units are available
 - hide the throughput loss from both short and long stall

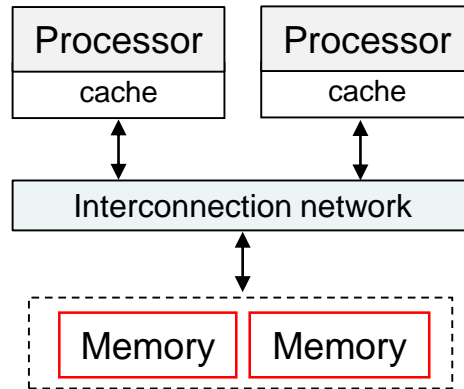
Multithreading Example



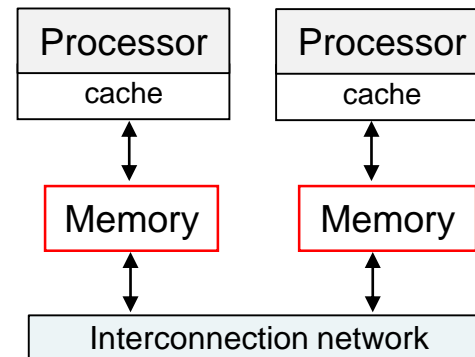
Coarse MT: Coarse-grained multithreading
Fine MT: Fine-grained multithreading
SMT: Simultaneous multithreading

Shared Memory Multiprocessor

- SMP: Symmetric Multi-Processing
 - Hardware provides single physical address space for all processors
 - Synchronize shared variables using locks
 - Memory access time
 - UMA (uniform) vs. NUMA (nonuniform)



UMA



NUMA

Example: Sum Reduction

- Sum 100,000 numbers on 100 processor UMA
 - Each processor has ID: $0 \leq P_n \leq 99$
 - Partition 1000 numbers per processor
 - Initial summation on each processor

```
sum[Pn] = 0;  
for ( i=1000*Pn; i<1000*(Pn+1); i=i+1)  
    sum[Pn] = sum[Pn] + A[i];
```

- Now need to add these partial sums
 - Reduction: divide and conquer
 - Half the processors add pairs, then quarter, ...
 - Need to synchronize between reduction steps

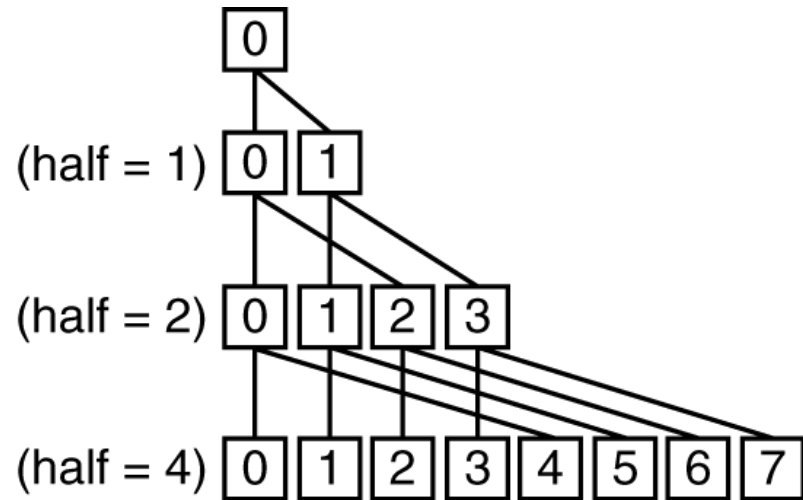
Example: Sum Reduction

- Sum 100,000 numbers on 100 processor UMA
 - Each processor has ID: $0 \leq P_n \leq 99$
 - Partition 1000 numbers per processor
 - Initial summation on each processor

```
sum[Pn] = 0;  
for ( i=1000*Pn; i<1000*(Pn+1); i=i+1)  
    sum[Pn] = sum[Pn] + A[i];
```

- Now need to add these partial sums
 - Reduction: divide and conquer
 - Half the processors add pairs, then quarter, ...
 - Need to synchronize between reduction steps

Example: Sum Reduction



```
half = 100;
```

```
repeat
```

```
    synch();
```

```
    if (half%2 != 0 && Pn == 0)
```

```
        sum[0] = sum[0] + sum[half-1];
```

// The condition
// when half is odd

```
    half = half/2;
```

```
    if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
```

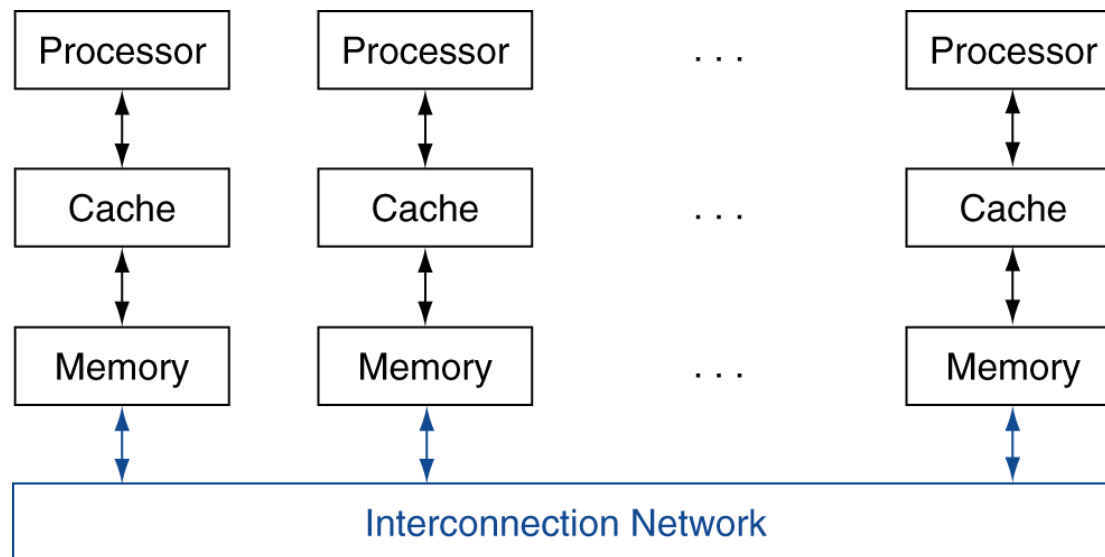
```
until (half == 1);
```

Loosely Coupled Clusters

- Network of independent computers
 - Each has private memory and OS
 - Connected using I/O system
 - E.g., Ethernet/switch, Internet
- Suitable for applications with independent tasks
 - Web servers, databases, simulations, ...
- High availability, scalable, affordable
- Problems
 - Administration cost
 - Low interconnect bandwidth
 - c.f. processor/memory bandwidth on an SMP

Message Passing

- Each processor has private physical address space
- Hardware **sends/receives** messages between processors

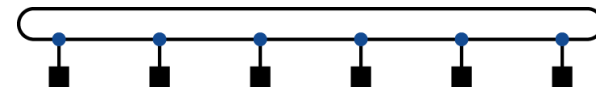


Interconnection Networks

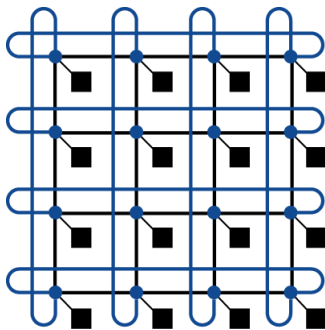
- Network topologies
 - Arrangements of processors, switches, and links



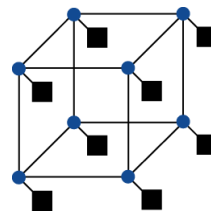
Bus



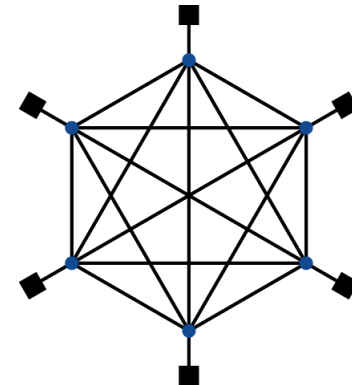
Ring



2D Mesh



N-cube ($N = 3$)



Fully connected

Concluding Remarks

- Higher performance by using multiple processors
 - Difficulties
 - Developing parallel software
 - Devising appropriate architectures
- SIMD and vector operations match multimedia applications and are easy to program
- Higher disk performance by using RAID