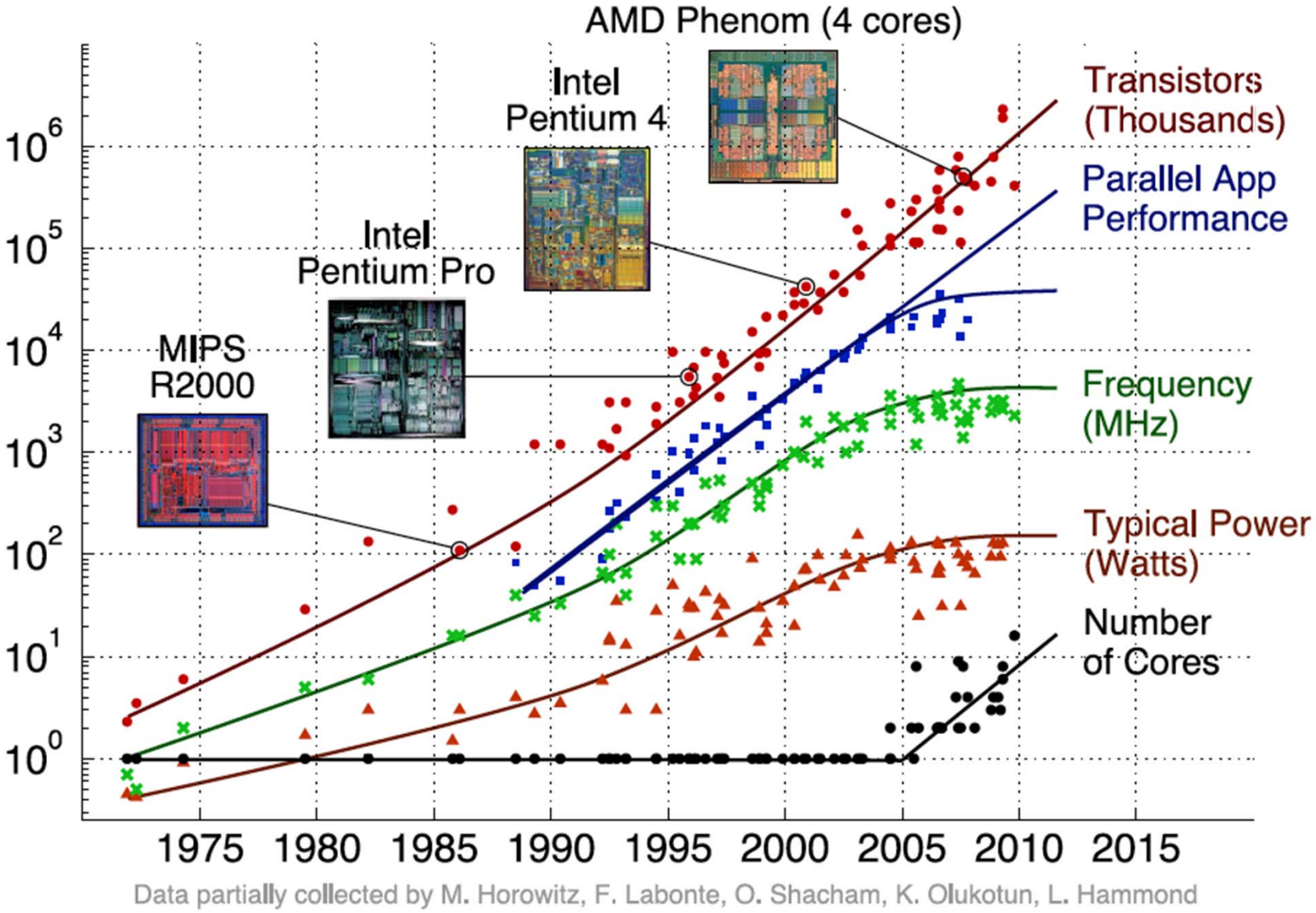
Lecture 27
Parallel Processors I

School of Computer Science and Engineering
Soongsil University

6. Parallel Processors from Client to Cloud

6.1 Introduction



Classes of Parallelism

1. **Bit-Level Parallelism (BLP)**
   - Parallel adder, parallel bus
2. **Instruction-Level Parallelism (ILP)**
   - Pipelining, superscalar, speculation, out-of-order execution
   - Single processor performance improvement ended in 2003
3. **Thread-Level Parallelism (TLP)**
   - Multithreading
4. **Task-Level Parallelism or Process-Level Parallelism**
   - MIMD
5. **Data-Level Parallelism (DLP)**
   - Vector processors, SIMD, Graphic Processor Units
6. **Request-Level Parallelism (RLP)**
   - Warehouse-scale computers (WSC), clusters

Parallel Processing

- **Multiprocessors**
  - Goal: connecting multiple computers to get higher performance
  - Future increases in performance will primarily come from explicit hardware parallelism rather than much higher clock rates or vastly improved CPI
  - Scalability, availability, power efficiency
- **Use of multiple processors**
  - Task-level (process-level) parallelism
    - Utilizing multiple processors by running independent programs simultaneously
    - High throughput for independent jobs
  - Parallel processing program
    - A single program run on multiple processors simultaneously
- **Multicore microprocessors**
  - Chips with multiple processors (cores)

Hardware and Software

- **Hardware**
  - Serial: e.g., Intel entium 4
  - Parallel: e.g., quad-core Intel Core i7-920
- **Software**
  - Sequential: e.g., matrix multiplication
  - Concurrent: e.g., operating system
- **Sequential/concurrent software can run on serial/parallel hardware**
  - Challenge: making effective use of parallel hardware

		Software	
		Sequential	**Concurrent**
Hardware	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 Core i7	Windows Vista Operating System running on an Intel Core i7

Figure 6.1

6.2 The Difficulty of Creating Parallel Processing Programs

- **Parallel software is the problem**
- **Need to get significant performance improvement**
  - Otherwise, just use a faster uniprocessor, since it's easier!
- **Difficulties**
  - Scheduling
  - Partitioning the work into parallel pieces
  - Balancing the load evenly between the processors
  - Synchronization
  - Coordination
  - Communications overhead

Amdahl's Law

- Sequential part can limit speedup
- **Example**
  - **90 times speedup with 100 processors?**

[Answer]

- $T_{new} = T_{parallelizable}/100 + T_{sequential}$
- $$\text{Speedup} = \frac{1}{(1 - F_{\text{paralleliz able}}) + F_{\text{paralleliz able}}/100} = 90$$
- Solving: $F_{parallelizable} = 0.999$
- Need sequential part to be 0.1% of original time

Example: Speed-up Challenge (1/2)

- Workload: add 10 scalars and add two 10×10 matrices
- Assumes load can be balanced across processors
- **What speedup do you get with 10 and 100 processors?**

[Answer]

- Single processor
  - Time = $(10 + 100) \times t_{add} = 110 \times t_{add}$
- 10 processors
  - Time = $10 \times t_{add} + 100/10 \times t_{add} = 20 \times t_{add}$
  - Speedup = 110/20 = 5.5 (55% of potential)
- 100 processors
  - Time = $10 \times t_{add} + 100/100 \times t_{add} = 11 \times t_{add}$
  - Speedup = 110/11 = 10 (10% of potential)

Example: Speed-up Challenge (2/2)

- **What if matrix size is 100 × 100?**

[Answer]

- Single processor
  - Time = (10 + 10000) × t_{add} = 10010 × t_{add}
- 10 processors
  - Time = 10 × t_{add} + 10000/10 × t_{add} = 1010 × t_{add}
  - Speedup = 10010/1010 = 9.9 (99% of potential)
- 100 processors
  - Time = 10 × t_{add} + 10000/100 × t_{add} = 110 × t_{add}
  - Speedup = 10010/110 = 91 (91% of potential)

Strong vs Weak Scaling

- **Strong scaling**
  - Speed-up achieved on a multiprocessor while keeping the problem size fixed
- **Weak scaling**
  - Speed-up achieved on a multiprocessor while increasing the size of the problem proportionally to the increase in the number of processors
  - 10 processors, sum of 10 scalars and 10 × 10 matrix
    - Time = 20 × t_{add}
  - 100 processors, sum of 10 scalars and 32 × 32 matrix
    - Time = 10 × t_{add} + 1024/100 × t_{add} = 20.24 × t_{add} ≈ 20 × t_{add}
  - Constant performance in this example

6.3 SISD, MIMD, SIMD, SPMD, and Vector

- **Flynn's taxonomy (1966)**

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Core i7

Figure 6.2

- **SPMD: Single Program Multiple Data**
  - Writing a single program that runs on all processors of an MIMD
  - Relying on conditional statements when different processors should execute different sections of code

Flynn-Johnson Classification

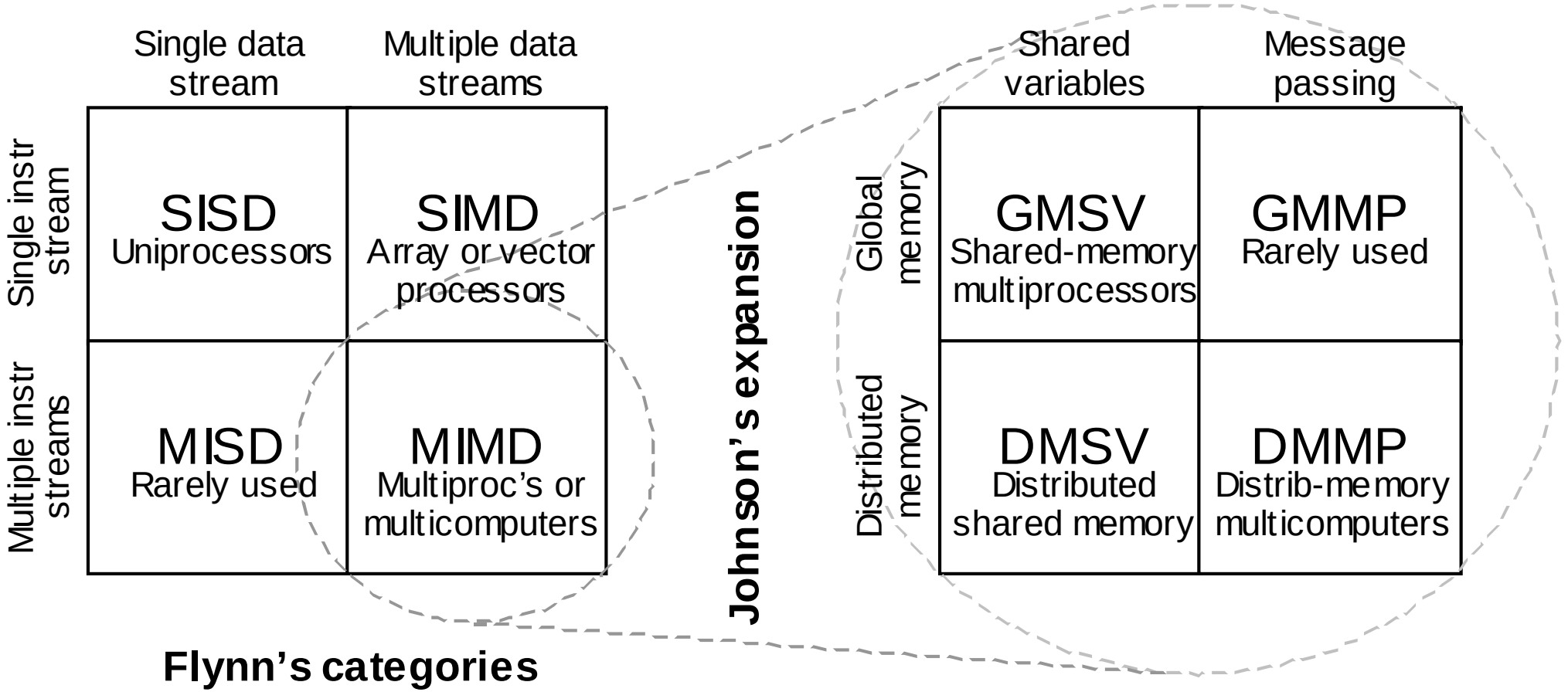
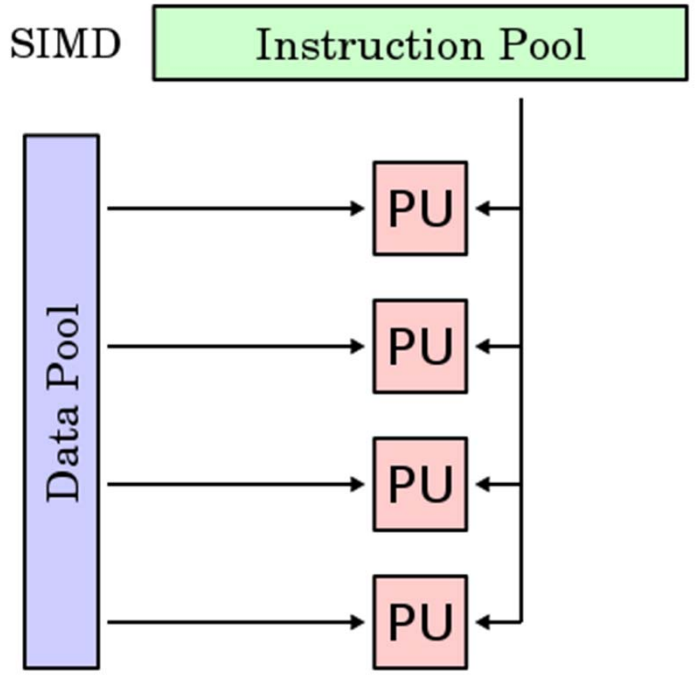


Figure 25.11 of Parhami

SIMD

- **Exploiting data-level parallelism (DLP)**
  - All the execution units execute the same instruction with different data
  - Each execution unit has its own address registers
- **Advantages**
  - Simplified synchronization
  - Reduced instruction control hardware
  - Reduced program memory
- **Weakness**
  - Conditional statements ... case or switch
- **SIMD in x86: Multimedia Extensions**
  - Subword parallelism for narrow integer data
  - Multimedia Extension (MMX), Streaming SIMD Extensions (SSE), Advanced Vector Extensions (AVX)



Vector Processors

- CPU design that is able to run mathematical operations on multiple data elements simultaneously
  - Vector registers, vector length registers, vector functional units, vector instructions
  - Significantly reduces instruction bandwidth
- Highly pipelined functional units
  - Pipelined SIMD vs. Parallel SIMD
- **Example**: Vector extension to MIPS
  - 32 × 64-element registers (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

Example: DAXPY (Y = a × X + Y)

- **Conventional MIPS code**

```
      l.d   $f0,a($sp)      ;load scalar a
      addiu r4,$s0,#512     ;upper bound of what to load
loop: l.d   $f2,0($s0)      ;load x(i)
      mul.d $f2,$f2,$f0     ;a × x(i)
      l.d   $f4,0($s1)      ;load y(i)
      add.d $f4,$f4,$f2     ;a × 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**

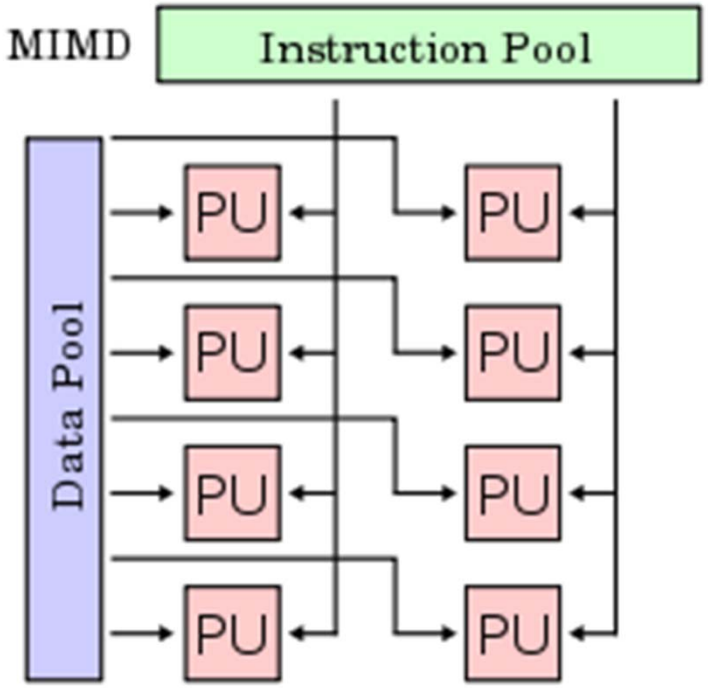
```
      l.d     $f0,a($sp)   ;load scalar a
      lv      $v1,0($s0)   ;load vector x
      mulvs.d $v2,$v1,$f0  ;vector-scalar multiply
      lv      $v3,0($s1)   ;load vector y
      addv.d  $v4,$v2,$v3  ;add y to product
      sv      $v4,0($s1)   ;store the result
```

Pros and Cons

- **Pros**
  - Improved code density
  - Reduced instruction bandwidth
    - Fewer page faults and instruction cache misses
  - No loop-carried dependences
    - Data hazard free within a vector instruction
  - Easier data-parallel programming
  - Regular access patterns benefit from interleaved and burst memory
  - Avoid control hazards by avoiding loops
- **Cons**
  - Extensions/modifications to the instruction set architecture
  - Extensions to the functional units and register files
  - Extensions to the memory system

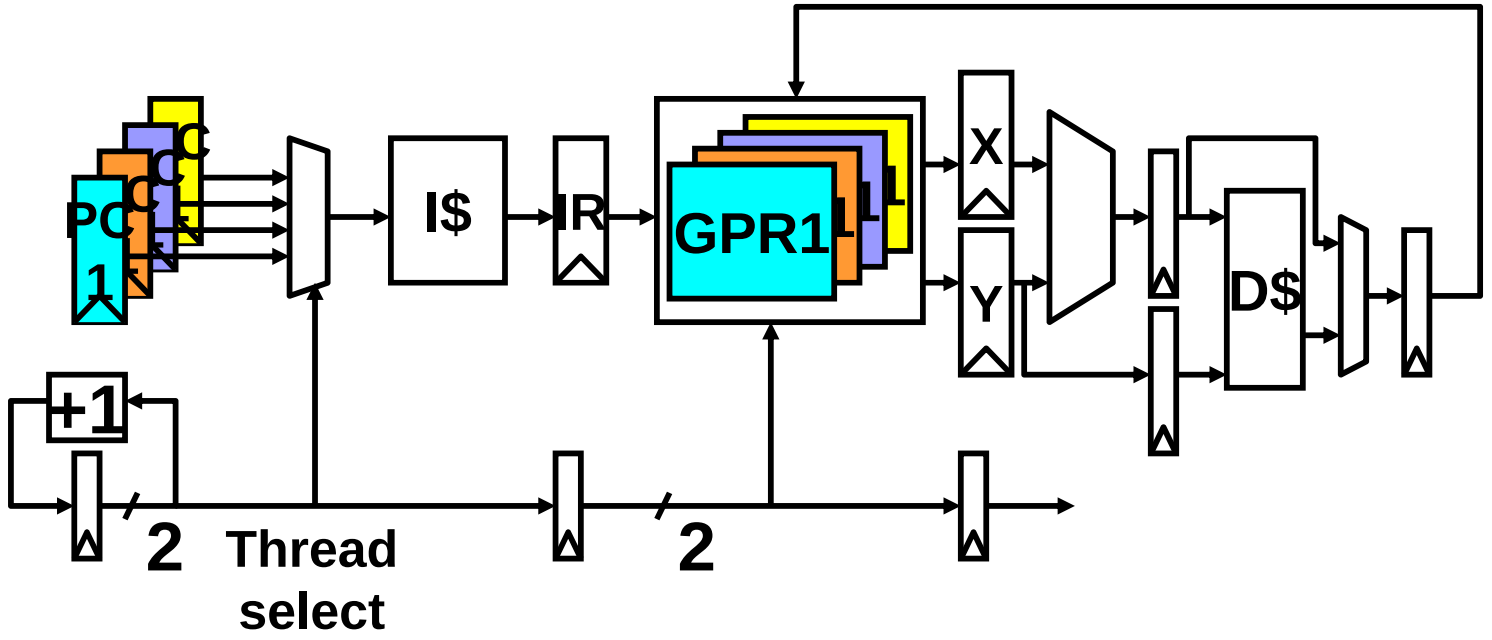
MIMD

- A number of processors that function asynchronously and independently
- Different processors may be executing different instructions on different pieces of data
- The processors can be different or identical
- Classification
  - UMA vs. NUMA
  - Centralized memory vs. distributed memory
  - Shared memory vs. private memory



6.4 Hardware Multithreading

- Difficult to continue to extract **instruction-level parallelism (ILP)** from a single sequential thread of control
- **Thread-level parallelism (TLP)**
  - TLP from multiprogramming (run independent sequential jobs)
  - TLP from multithreaded applications (run one job faster using parallel threads)



Multithreading Categories

- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast switching between threads

1. **Fine-grain multithreading**
   - Switch threads after each cycle
   - Interleave instruction execution
   - If one thread stalls, others are executed
2. **Coarse-grain multithreading**
   - Only switch on long stall (e.g., L3-cache miss)
   - Simplifies hardware, but doesn't hide short stalls (e.g., data hazards)
3. **Simultaneous multithreading (SMT)**
   - Thread-level parallelism + instruction-level parallelism
   - Dynamic scheduling + multiple issue

Simultaneous Multithreading (SMT)

- **In multiple-issue dynamically scheduled processor**
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- **Intel Hyperthreading(HT)**
  - First introduced in the Foster MP-based Xeon in March 2002
  - 2-thread SMT
  - Duplicated registers and shared function units and caches
  - Logical processors share nearly all resources of the physical processor
    - Caches, execution units, branch predictors
  - Die area overhead of hyperthreading ~ 5%
  - When one logical processor is stalled, the other can make progress

Four Different Approaches

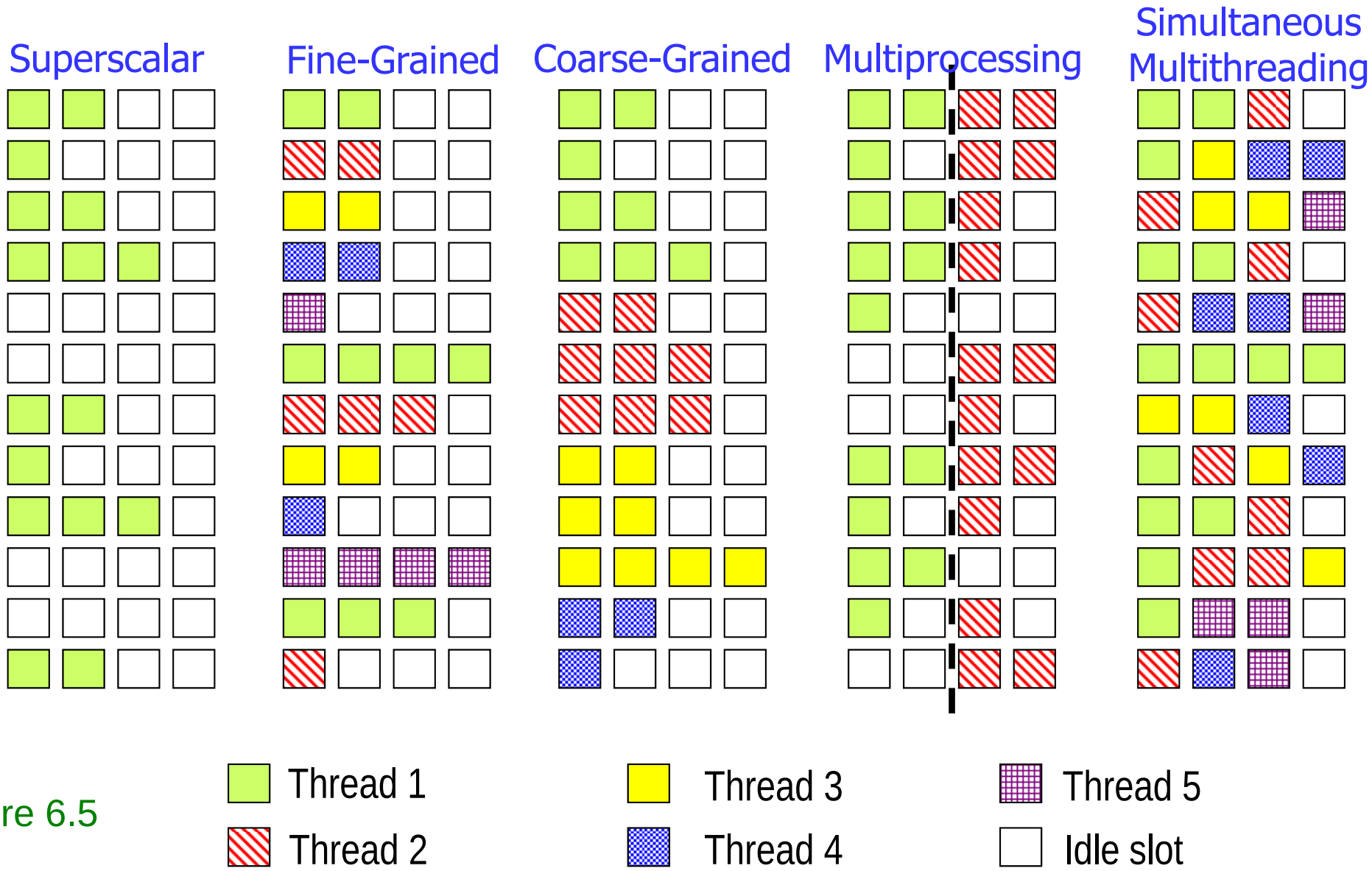


Figure 6.5

6.5 Multicore and Other Shared Memory Multiprocessors

- **SMP: shared memory multiprocessor**
  - Hardware provides single physical address space for all processors
  - Processors communicate through shared variables in memory
  - Synchronize shared variables using locks
  - Memory access time
    - UMA (uniform) vs. NUMA (nonuniform)

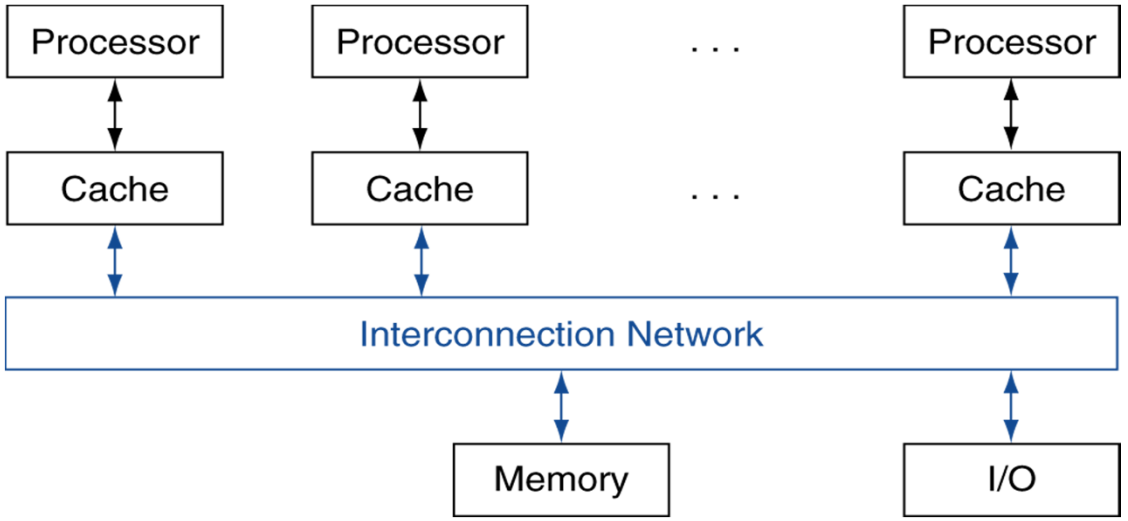


Figure 6.7

Example: A Simple Parallel Processing Program for a Shared Address Space (Sum Reduction)

- **Sum 64,000 numbers on 64 processor UMA**
  - Each processor has ID: 0 ≤ Pn ≤ 63
  - Partition 1000 numbers per processor
  - Initial summation on each processor

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

- **Now need to add these partial sums**
  - Reduction: divide and conquer
    - A function that processes a data structure and returns a single value.
  - Half the processors add pairs, then quarter, …
  - Need to synchronize between reduction steps

Example: (continued)

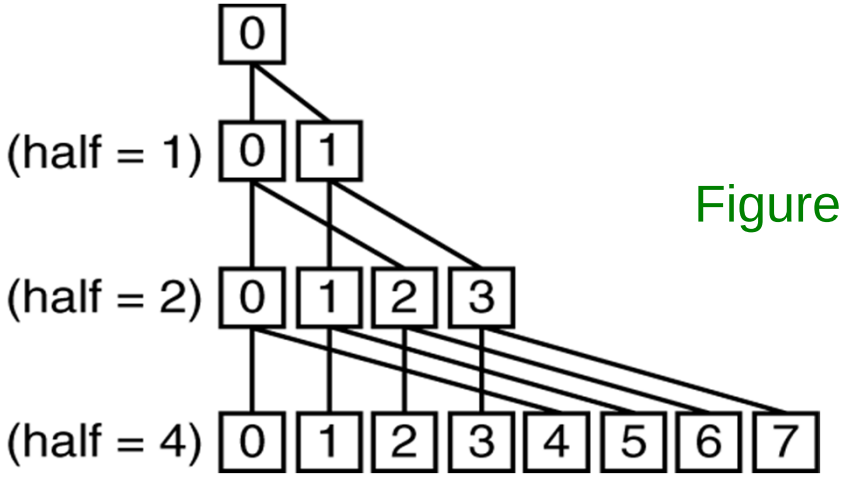
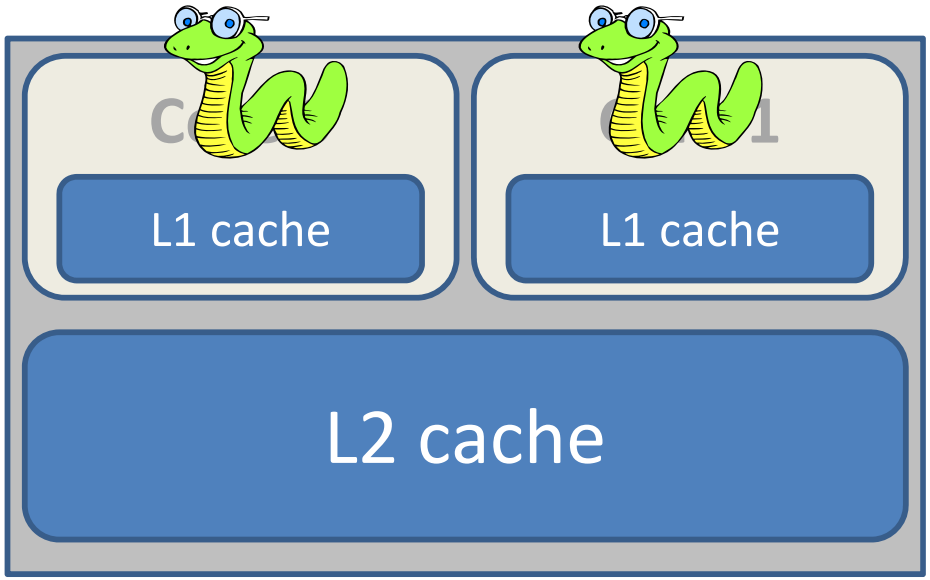


Figure 6.8

```
half = 64;
do
  synch();
  if (half%2 != 0 && Pn == 0)
    sum[0] += sum[half-1];
    /* Conditional sum needed when half is odd;
       Processor0 gets missing element */
  half = half/2; /* dividing line on two sums */
  if (Pn < half) sum[Pn] += sum[Pn+half];
while (half > 1); /* exit with final sum in Sum[0] */
```

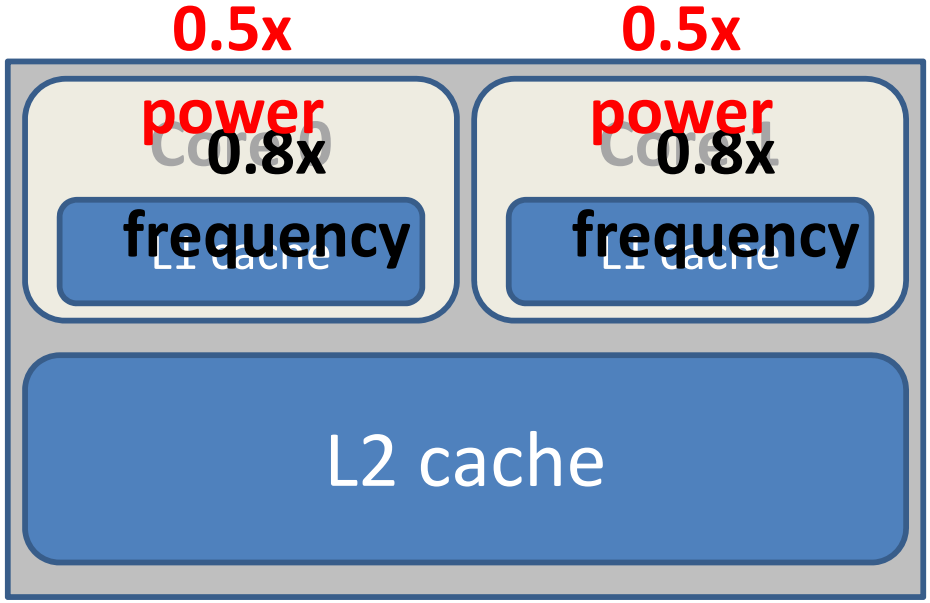
Multicore Processor



- **Conventional processor**
  - Single core
  - Dedicated caches
- **Multicore processor**
  - At least two cores
  - Shared caches

Why Multicore ?

- Power consumption is a huge problem
- Multicore chips potentially produce a lot more computation per unit of power
- **Example**
  - Reduce CPU clock frequency by 20%
  - Power consumption reduces by 50%!
- Put two 0.8 frequency cores on the same chip
  - Get 1.6 times the computation at the same power consumption



Multicore Processor

- Single computing component with two or more independent actual processors
- **Dual-core processors**
  - AMD Phenom II X2, Intel Core 2 Duo, IBM POWER6
- **Quad-core processors**
  - AMD Phenom II X4, Intel Core 2 Quad, Core i5, Core i7, IBM POWER7
- **Hex-core processors**
  - AMD Phenom II X6, Intel Core i7 Extreme Edition 980X
- **Octo-core processors**
  - Intel Core i7, IBM POWER7, Nvidia GeForce 9, SUN UltraSPARC T1, T2, T3
- **Many-core processors**
- **Chip multiprocessor = CMP**
  - Multicore processors integrated onto a single chip