Issues in Parallel Processing

Lecture for CPSC 5155
Edward Bosworth, Ph.D.
Computer Science Department
Columbus State University

Introduction

- Goal: connecting multiple computers to get higher performance
  - Multiprocessors
  - Scalability, availability, power efficiency
- Job-level (process-level) parallelism
  - High throughput for independent jobs
- Parallel processing program
  - Single program run on multiple processors
- Multicore microprocessors
  - Chips with multiple processors (cores)

Questions to Address

1. How do the parallel processors share data?
2. How do the parallel processors coordinate their computing schedules?
3. How many processors should be used?
4. What is the minimum speedup S(N) acceptable for N processors?
   What are the factors that drive this decision?

Question: How to Get Great Computing Power?

- There are two obvious options.

1. Build a single large very powerful CPU.
2. Construct a computer from multiple cooperating processing units.

- The early choice was for a computing system with only a few (1 to 16) processing units.
- This choice was based on what appeared to be very solid theoretical grounds.

Linear Speed-Up

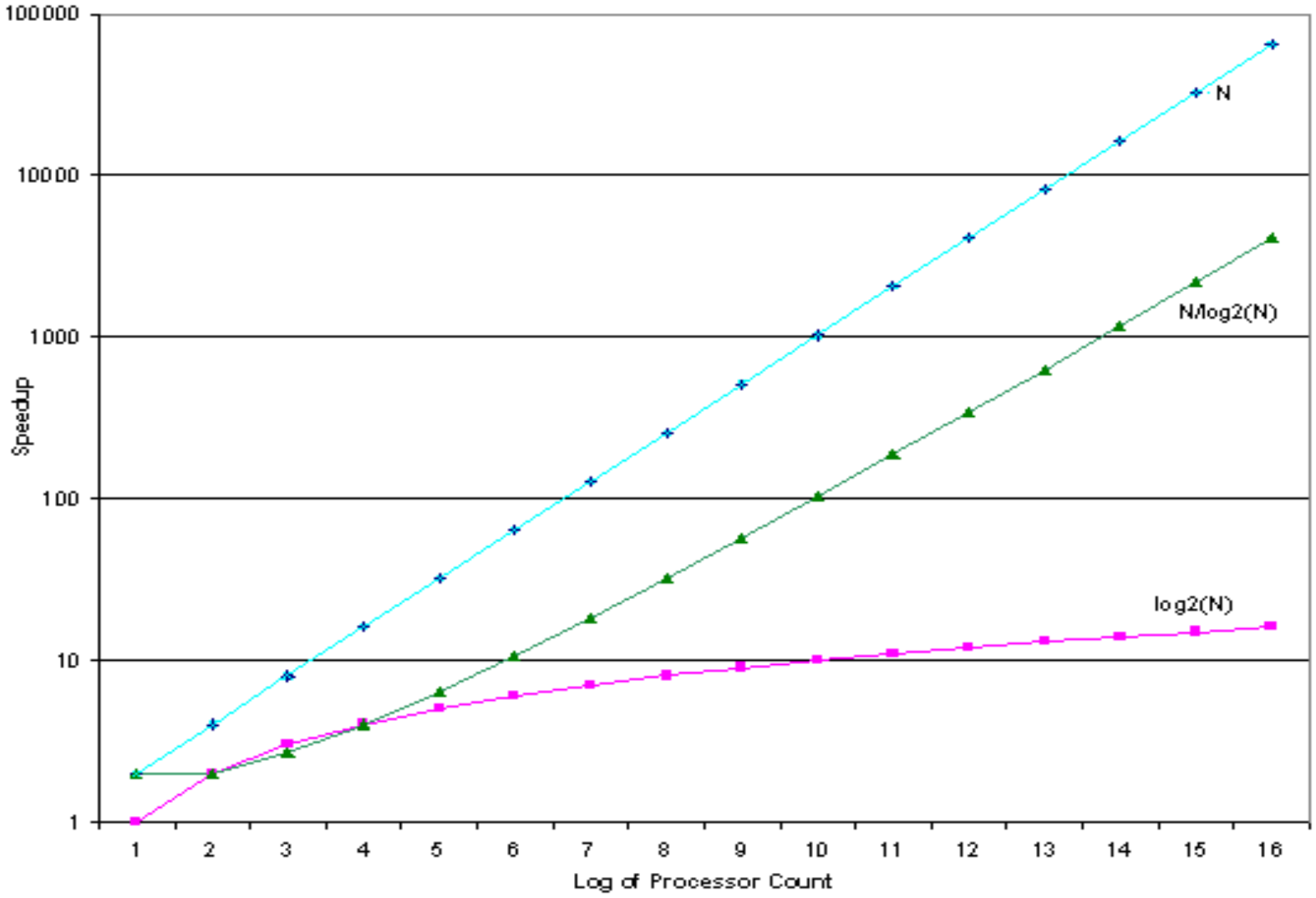
- The cost of a parallel processing system with N processors is about N times the cost of a single processor; the cost scales linearly.
- The goal is to get N times the performance of a single processor system for an N-processor system. This is **linear speedup**.
- For linear speedup, the cost per unit of computing power is approximately constant.

The Cray-1

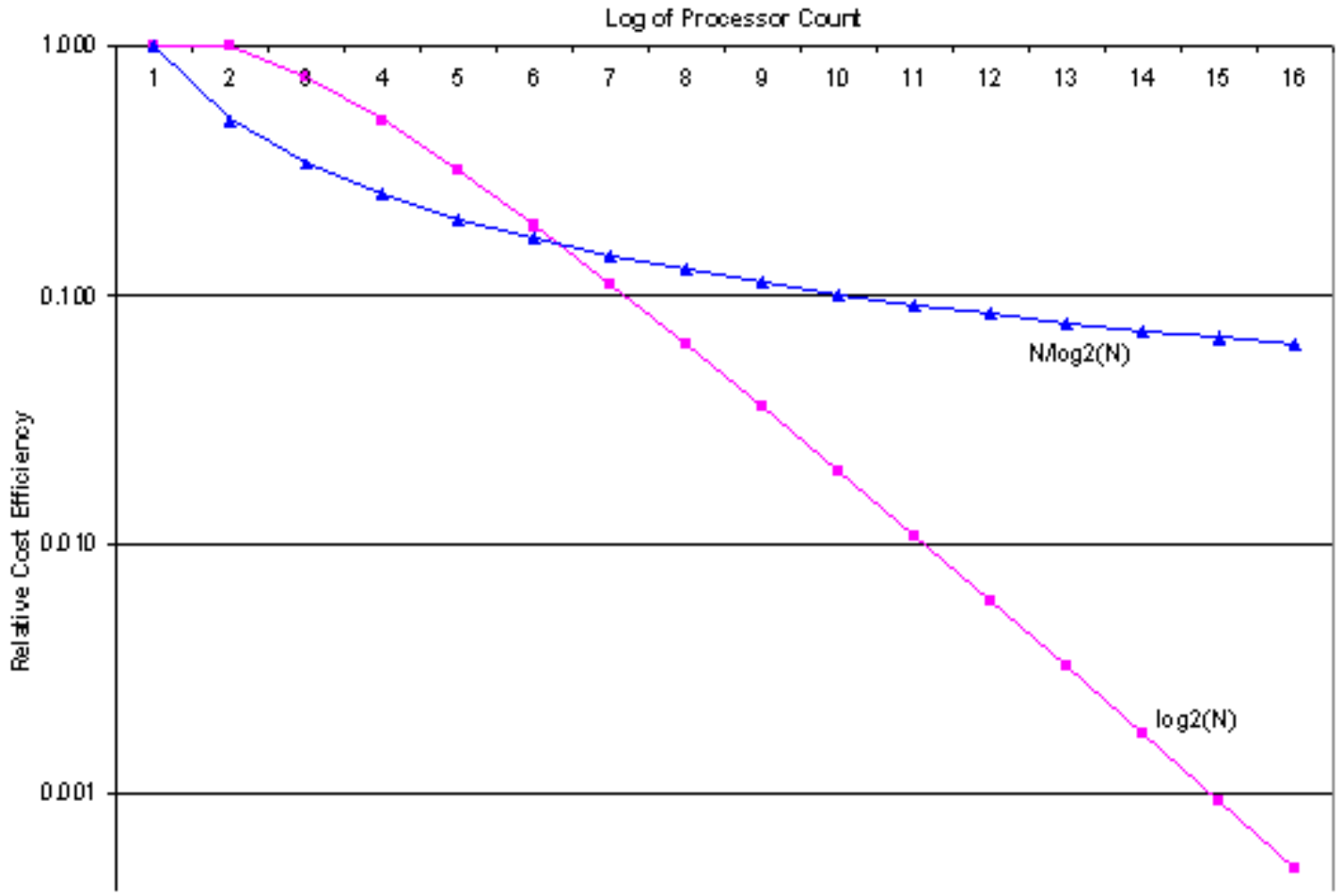
Supercomputers vs. Multiprocessor Clusters

- “If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens”. Seymour Cray
- Here are two opinions from a 1984 article.
  “The speedup factor of using an *n*–processor system over a uniprocessor system has been theoretically estimated to be within the range ($\log_2 n$, $n/\log_2 n$).”
- “By the late 1980s, we may expect systems of 8–16 processors. Unless the technology changes drastically, we will not anticipate massive multiprocessor systems until the 90s.”
- The drastic technology change is called “VLSI”.

The Speed–Up Factor: S(N)



Cost Efficiency: S(N) / N



Harold Stone on Linear Speedup

- Harold Stone wrote in 1990 on what he called "**peak performance**".
- "When a multiprocessor is operating at peak performance,

1. All processors are engaged in useful work.
2. No processor is idle, and no processor is executing an instruction that would not be executed if the same algorithm were executing on a single processor.
3. In this state of peak performance, all N processors are contributing to effective performance, and the processing rate is increased by a factor of N.
4. Peak performance is a very special state that is rarely achievable."

The Problem with the Early Theory

- The early work focused on the problem of general computation.
- Not all problems can be solved by an algorithm that can be mapped onto a set of parallel processors.
- However, many very important problems can be solved by parallel algorithms.

Hardware and Software

- Hardware
  - Serial: e.g., Pentium 4
  - Parallel: e.g., quad-core Xeon e5345
- 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

Cooperation Among Processes

- Parallel execution on a multi-core CPU is not inherently a difficult problem. The problems arise when the processes need to cooperate.
- Example: A quad-core running 4 independent programs that do not communicate.
- One measure of the complexity of parallel execution is the amount of communication required among the processes.
- More communication means more complex.

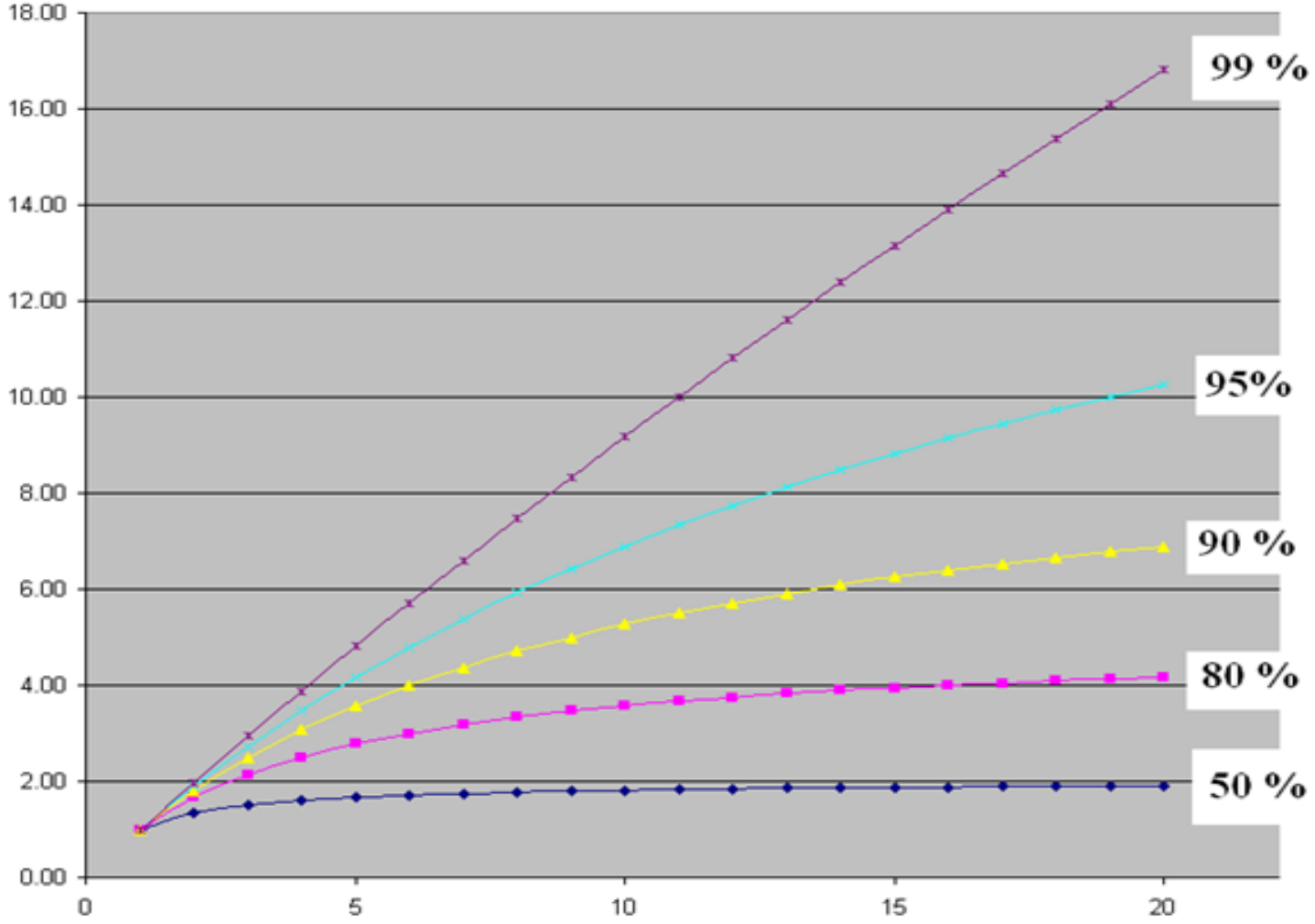
Parallel Programming

- Parallel software is the problem
- Need to get significant performance improvement
  - Otherwise, just use a faster uniprocessor, since it's easier!
- Difficulties
  - Partitioning
  - Coordination
  - Communications overhead

Amdahl's Law

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

Some Results Due to Amdahl's Law



Characterizing Problems

- One result of Amdahl's Law is that only problems with very small necessarily sequential parts can benefit from massive parallel processing.
- Fortunately, there are many such problems

1. Weather forecasting.
2. Nuclear weapons simulation.
3. Protein folding and issues in drug design.

Scaling Example

- Workload: sum of 10 scalars, and 10 × 10 matrix sum
  - Speed up from 10 to 100 processors
- Single processor: Time = (10 + 100) × t_{add}
- 10 processors
  - Time = 10 × t_{add} + 100/10 × t_{add} = 20 × t_{add}
  - Speedup = 110/20 = 5.5 (55% of potential)
- 100 processors
  - Time = 10 × t_{add} + 100/100 × t_{add} = 11 × t_{add}
  - Speedup = 110/11 = 10 (10% of potential)
- Assumes load can be balanced across processors

Scaling Example (cont)

- What if matrix size is 100 × 100?
- Single processor: Time = (10 + 10000) × 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)
- Assuming load balanced

Strong vs Weak Scaling

- Strong scaling: problem size fixed
  - As in example
- Weak scaling: problem size proportional to number of processors
  - 10 processors, 10 × 10 matrix
    - Time = $20 \times t_{add}$
  - 100 processors, 32 × 32 matrix
    - Time = $10 \times t_{add} + 1000/100 \times t_{add} = 20 \times t_{add}$
  - Constant performance in this example

Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don’t synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

The Necessity for Synchronization

- “In a multiprocessing system, it is essential to have a way in which two or more processors working on a common task can each execute programs without corrupting the other’s sub-tasks”.
- “Synchronization, an operation that guarantees an orderly access to shared memory, must be implemented for a properly functioning multiprocessing system”.
- Chun & Latif, MIPS Technologies Inc.

Synchronization in Uniprocessors

- The synchronization issue posits 2 processes sharing an area of memory.
- The processes can be on different processors, or on a single shared processor.
- Most issues in operating system design are best imagined within the context of multiple processors, even if there is only one that is being time shared.

The Lost Update Problem

- Here is a synchronization problem straight out of database theory. Two travel agents book a flight with one seat remaining.
- A1 reads seat count. One remaining.
- A2 reads seat count. One remaining.
- A1 books the seat. Now there are no more seats.
- A2, working with old data, also books the seat. Now we have at least one unhappy customer.

I've Got It;
You Can't Have It

- What is needed is a way to put a "lock" on the seat count until one of the travel agents completes the booking. Then the other agent must begin with the new seat count.
- Database engines use "**record locking**" as one way to prevent lost updates.
- Another database technique is the idea of an **atomic transaction**, here a 2-step transaction.

Atomic Transactions

- We do not mean the type of transaction at left.
- An **atomic read and modify** must proceed without any interruption.
- No other process can access the shared memory between the read and write back to the memory location.

Synchronization in MIPS

- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt
- Example: atomic swap (to test/set lock variable)

```
try: add $t0,$zero,$s4 ;copy exchange value
     ll  $t1,0($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
```

Details on LL and SC

- These commands work with the cache memory system at a cache line level.
- Each cache line has a LL bit, which is set by the Load Linked command.
- The LL bit will be cleared if another process writes to that specific cache line.
- The SC command works only if the LL bit remains set; otherwise it fails.

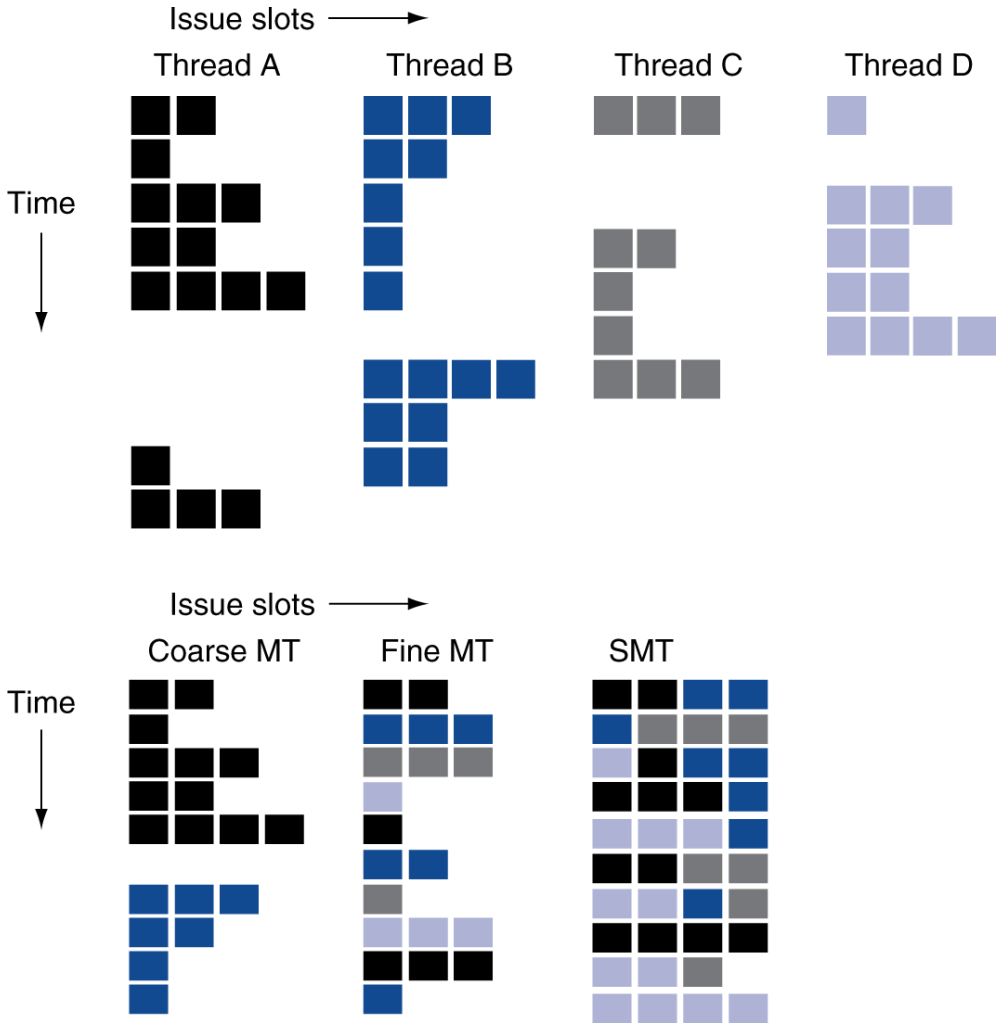
Multithreading

- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast switching between threads
- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- 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

- 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
- Example: Intel Pentium-4 HT
  - Two threads: duplicated registers, shared function units and caches

Multithreading Example



Future of Multithreading

- Will it survive? In what form?
- Power considerations ⇒ simplified microarchitectures
  - Simpler forms of multithreading
- Tolerating cache-miss latency
  - Thread switch may be most effective
- Multiple simple cores might share resources more effectively

Instruction and Data Streams

- An alternate classification

		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

- SPMD: Single Program Multiple Data
  - A parallel program on a MIMD computer
  - Conditional code for different processors

SIMD

- Operate elementwise on vectors of data
  - E.g., MMX and SSE instructions in x86
    - Multiple data elements in 128-bit wide registers
- All processors execute the same instruction at the same time
  - Each with different data address, etc.
- Simplifies synchronization
- Reduced instruction control hardware
- Works best for highly data-parallel applications

Vector Processors

- Highly pipelined function units
- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Results stored from registers to memory
- 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
- Significantly reduces instruction-fetch bandwidth

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
```

Vector vs. Scalar

- Vector architectures and compilers
  - Simplify data-parallel programming
  - Explicit statement of absence of loop-carried dependences
    - Reduced checking in hardware
  - Regular access patterns benefit from interleaved and burst memory
  - Avoid control hazards by avoiding loops
- More general than ad-hoc media extensions (such as MMX, SSE)
  - Better match with compiler technology