



NATIONAL RESEARCH
UNIVERSITY



Computer Architecture and Operating Systems

Lecture 14: Thread-Level Parallelism

Andrei Tatarnikov

atatarnikov@hse.ru

[@andrewt0301](https://github.com/andrewt0301)

Why We Need Thread-Level Parallelism

Goals

- Task-level (process-level) parallelism
 - High throughput for independent jobs
- Parallel processing program
 - Single program run on multiple processors

Implementations

- Hardware multithreading
- Multicore microprocessors
 - Chips with multiple processors (cores)
- Multiprocessors
 - Connecting multiple computers to get higher performance
 - Scalability, availability, power efficiency

Challenge: Parallel Programming

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

Threading: Definitions

- Process: program running on a computer
 - Multiple processes can run at once: e.g., surfing Web, playing music, writing a paper
 - Separate virtual memory, stack, registers
- Thread: part of a program
 - Each process has multiple threads: e.g., a word processor may have threads for typing, spell checking, printing
 - Shared virtual memory, separate stack and registers

Threads in Conventional Uniprocessor (SISD)

- One thread runs at once
- When one thread stalls (for example, waiting for memory):
 - Architectural state of that thread stored
 - Architectural state of waiting thread loaded into processor and it runs
 - Called **context switching** (can take thousands of cycles)
- Appears to user like all threads running simultaneously
- Does not improve performance

Parallel Processing Challenge: Amdahl's Law

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

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

Scaling Example 1

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

Scaling Example 2

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

Strong vs Weak Scaling

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

Hardware Multithreading

- Multiple copies of architectural state
- Multiple threads active at once:
 - When one thread stalls, another runs immediately
 - If one thread can't keep all execution units busy, another thread can use them
- Does not increase instruction-level parallelism (ILP) of single thread, but increases throughput

Intel calls this “hyperthreading”

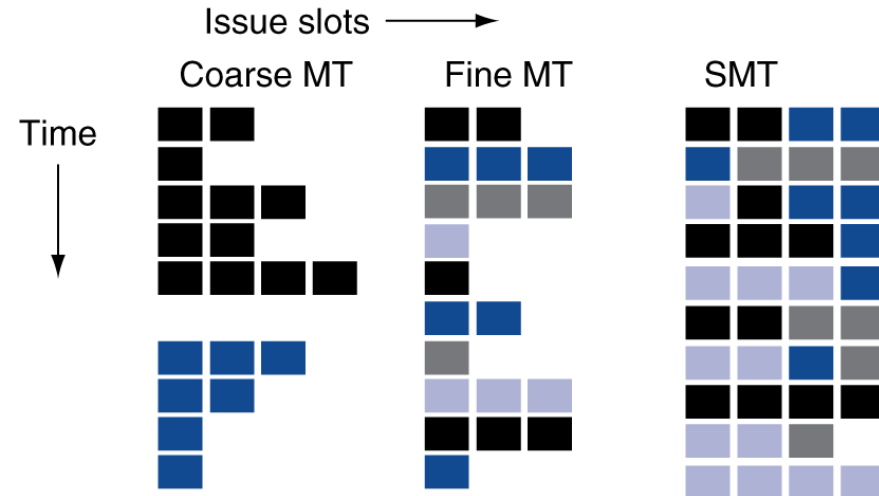
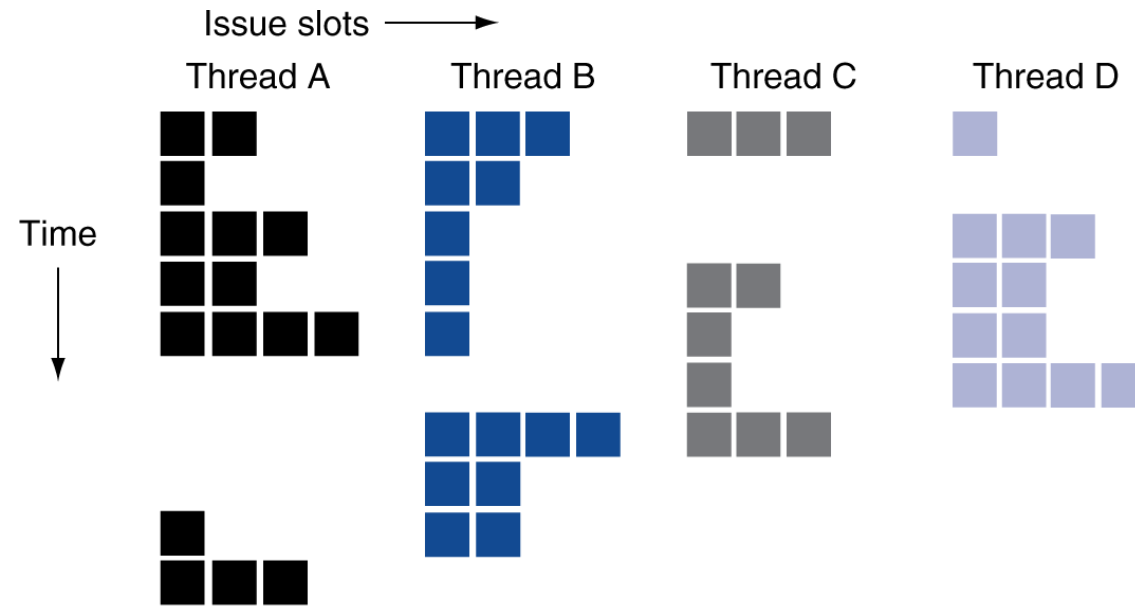
Hardware Multithreading

- Performing multiple threads of execution in parallel
 - Replicate registers, PC, etc.
 - Fast switching between threads
- Fine-grained multithreading
 - Switch threads after each cycle
 - Interleave instruction execution
 - If one thread stalls, others are executed
- Coarse-grained multithreading
 - Only switch on long stall (e.g., L2-cache miss)
 - Simplifies hardware, but doesn't hide short stalls (eg, data hazards)
- Simultaneous multithreading

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

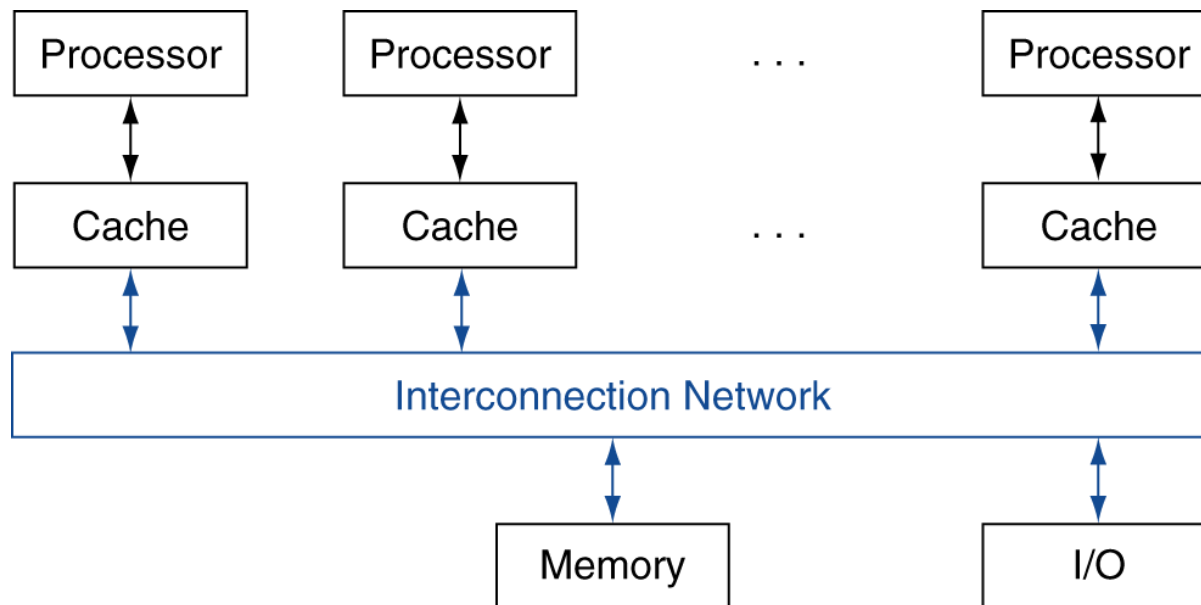


Multiprocessors (MIMD)

- Multiple processors (cores) with a method of communication between them
- Types:
 - Homogeneous: multiple cores with shared memory
 - Heterogeneous: separate cores for different tasks (for example, DSP and CPU in cell phone)
 - Clusters: each core has own memory system

Multicores: Shared Memory

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



Multicores and Cache Coherence

- Suppose two CPU cores share a physical address space
 - Write-through caches

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

Coherence Defined

- Informally: Reads return most recently written value
- Formally:
 - P writes X; P reads X (no intervening writes)
⇒ read returns written value
 - P_1 writes X; P_2 reads X (sufficiently later)
⇒ read returns written value
 - CPU B reading X after step 3 in example
 - P_1 writes X, P_2 writes X
⇒ all processors see writes in the same order
 - End up with the same final value for X

Cache Coherence Protocols

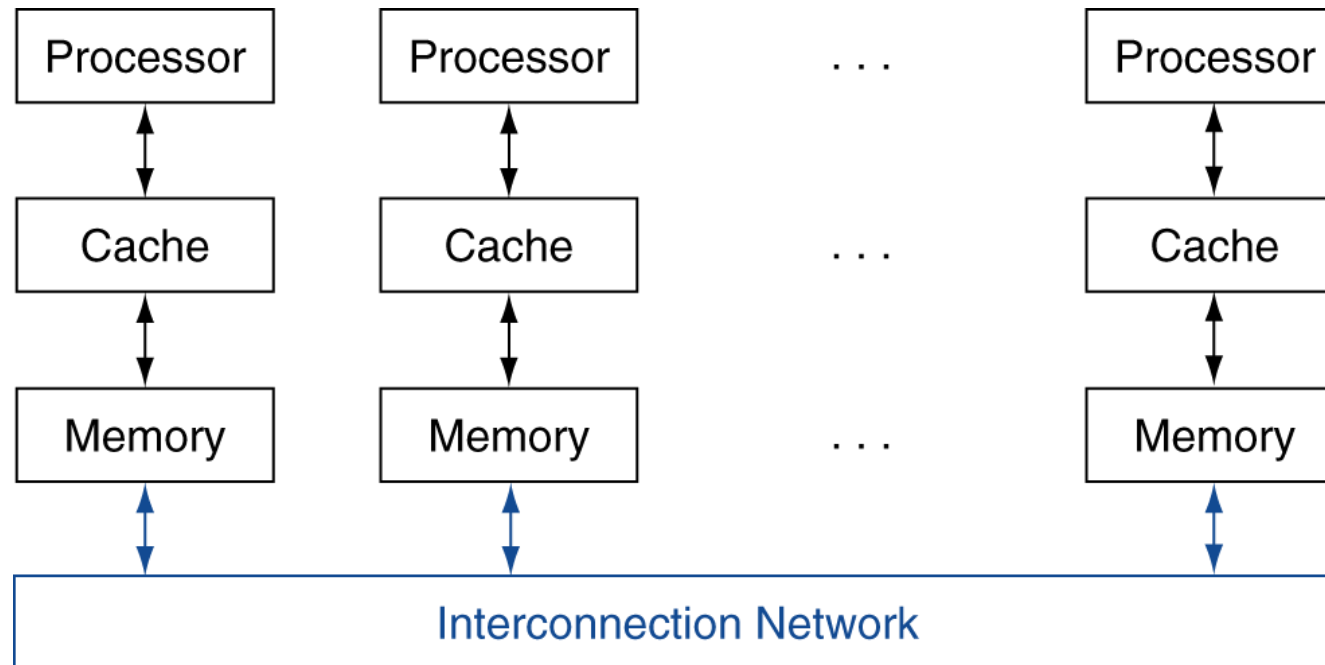
- Operations performed by caches in multiprocessors to ensure coherence
 - Migration of data to local caches
 - Reduces bandwidth for shared memory
 - Replication of read-shared data
 - Reduces contention for access
- Snooping protocols
 - Each cache monitors bus reads/writes
- Directory-based protocols
 - Caches and memory record sharing status of blocks in a directory

Synchronization: Basic Building Blocks

- Atomic exchange
 - Swaps register with memory location
- Test-and-set
 - Sets under condition
- Fetch-and-increment
 - Reads original value from memory and increments it in memory
- Requires read and write in uninterruptable instruction
- RISC-V: load reserved/store conditional
 - If the memory location specified by the load is changed before the store conditional to the same address, the store conditional fails

Multiprocessors: Message Passing

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



Any Questions?

```
                .text
__start:      addi t1, zero, 0x18
                addi t2, zero, 0x21
cycle:        beq t1, t2, done
                slt t0, t1, t2
                bne t0, zero, if_less
                nop
                sub t1, t1, t2
                j cycle
                nop
if_less:      sub t2, t2, t1
                j cycle
done:         add t3, t1, zero
```