

# **Thread Level Parallelism**

Overtaking the law of diminishing returns

# Source material

- J. L. Hennessy, D. A. Patterson, and K. Asanović, *Computer Architecture: A Quantitative Approach*, 5th ed. Waltham, MA: Morgan Kaufmann/Elsevier, 2012.
  - Chapter 1: Fundamentals of Quantitative Design Analysis
  - Chapter 5: Thread-Level Parallelism
- D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition*. Cambridge, MA: Morgan Kaufmann, 2018.
  - Chapter 6: Parallel Processors from Client to Cloud
- D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture A Hardware / Software Approach*
  - Chapter 1: Introduction (suggested read)
  - Chapter 2: Parallel Programs (suggested read)

# Elapsed time

$$T_{\text{exe}} = N * \text{CPI} * T_c$$

- $T_{\text{exe}}$  := Execution time
- $N$  := Number of instructions
- $\text{CPI}$  := Average Cycles Per Instructions
- $T_c$  := Cycle time (inverse of CPU frequency)
  
- Goal: Minimize  $T_{\text{exe}}$
- Key: For a given problem, use cheaper instructions ( $\text{CPI}$ ) or faster clocks ( $T_c$ )

# Moore's Law

*The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. [...] Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years.*

Gordon Moore, 1965

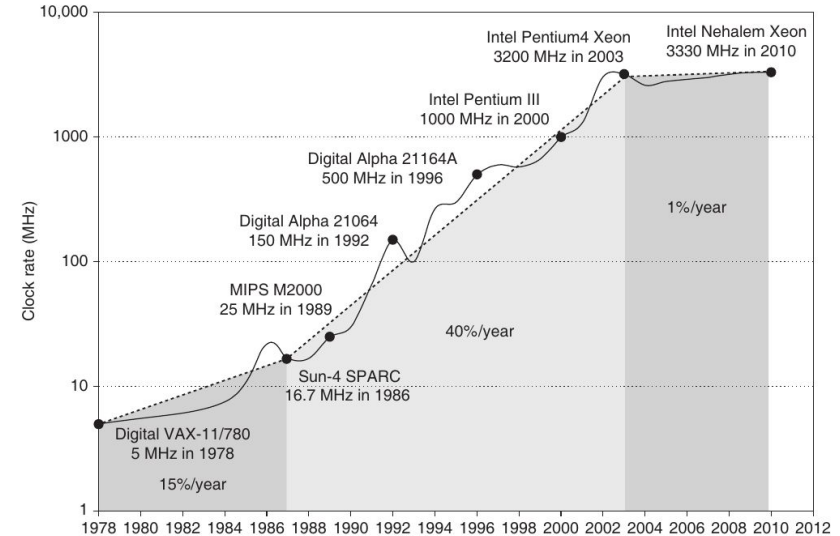
- Original message: Component density will improve exponentially over time
- Distorted message: Performance will improve exponentially over time “for free”
- Key:
  - Moore's law was a prediction based on an observation, not a law
  - Density is still improving, but performance “for free” stopped around the 90s

# A powerful problem

- Static power consumption
  - *“Increases linearly with leakage current and supply voltage”*
  - 😊 Smaller components allow reducing supply voltage
  - 😬 As we get smaller and smaller components, leakage current becomes dominant
- Dynamic power consumption
  - *“Increases linearly with the switching frequency and number of components”*
  - 😬 More complex circuits and faster clock rates were giving performance “for free”
- Huge oversimplification (by me)... Just to get the point across
- [Short read on power consumption of CMOS circuits](#)

# Limits of single processors

- Up until the 90s and early 2000s, performance gains were achieved by
  - micro-arch improvements
  - increasing clock rates (frequency)
- Eventually, the law of diminishing returns stroke
- Cost of higher clock rates outweighs performance benefits
- Time for a different approach



**Figure 1.11** Growth in clock rate of microprocessors in Figure 1.1. Between 1978 and 1986, the clock rate improved less than 15% per year while performance improved by 25% per year. During the “renaissance period” of 52% performance improvement per year between 1986 and 2003, clock rates shot up almost 40% per year. Since then, the clock rate has been nearly flat, growing at less than 1% per year, while single processor performance improved at less than 22% per year.

# “Running in parallel”

- Concurrency

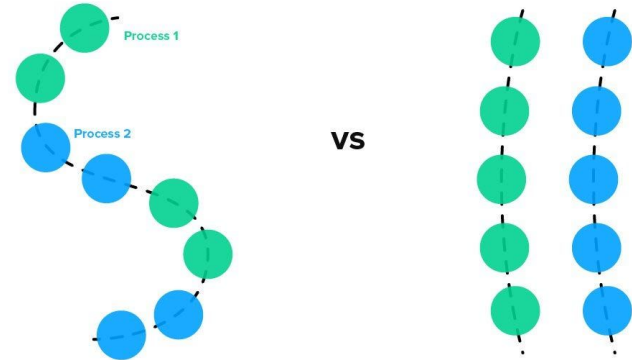
- “The ability of a system to execute multiple tasks by sharing resources and managing interactions”
- Execute two or more programs at once

- Multiplexing / Context switching

- Interleaved execution on single processing unit

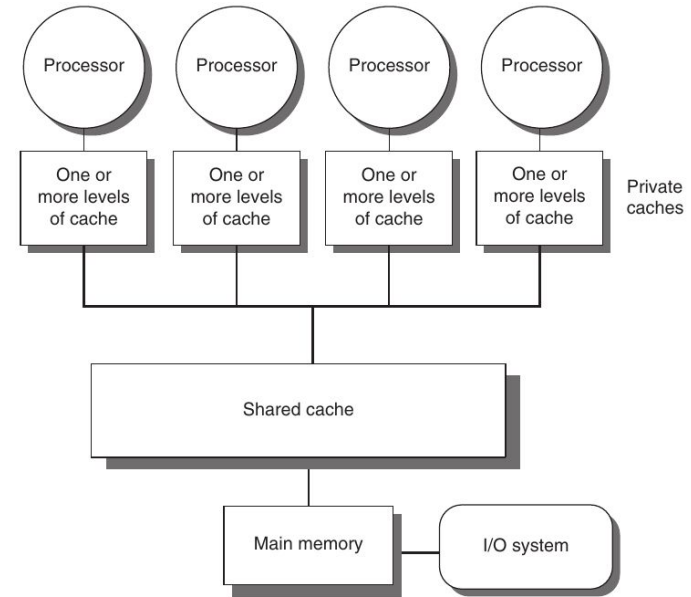
- Parallelism

- Simultaneous execution on multiple processing units



# Shared-Memory Processor (SMP)

- Replicate processor
  - Also may include first levels of cache
  - We call these resources private
- Centralized memory
  - Also may include higher levels of cache
  - We call these resources shared
- Cost of accessing shared resources is the same for all processors
  - We call this feature symmetric
  - Uniform Memory Access (UMA)

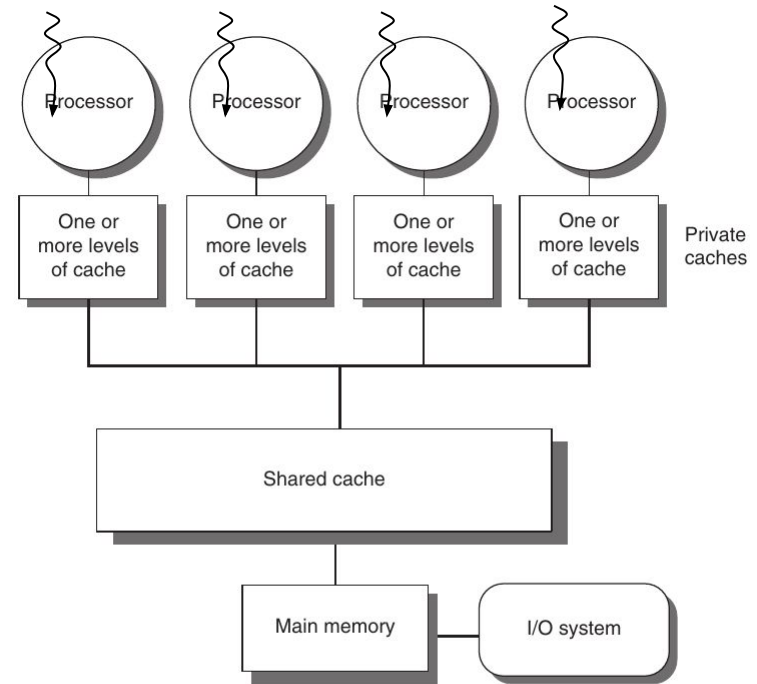


**Figure 5.1** Basic structure of a centralized shared-memory multiprocessor based on a multicore chip. Multiple processor-cache subsystems share the same physical memory, typically with one level of shared cache, and one or more levels of private per-core cache. The key architectural property is the uniform access time to all of the memory from all of the processors. In a multichip version the shared cache would be omitted and the bus or interconnection network connecting the processors to memory would run between chips as opposed to within a single chip.



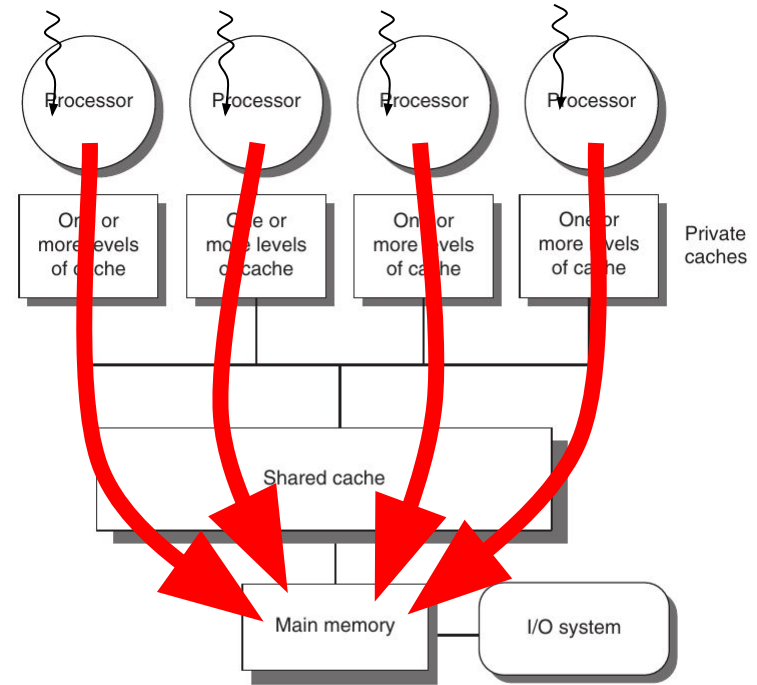
# Software running on an SMP

- Each processor has its own register file, including PC and other control state
- We call thread the sequence of instructions running on a processor
- Threads of instructions may belong to the same program or to different programs
- The Operating System (OS) manages programs (processes) and its threads



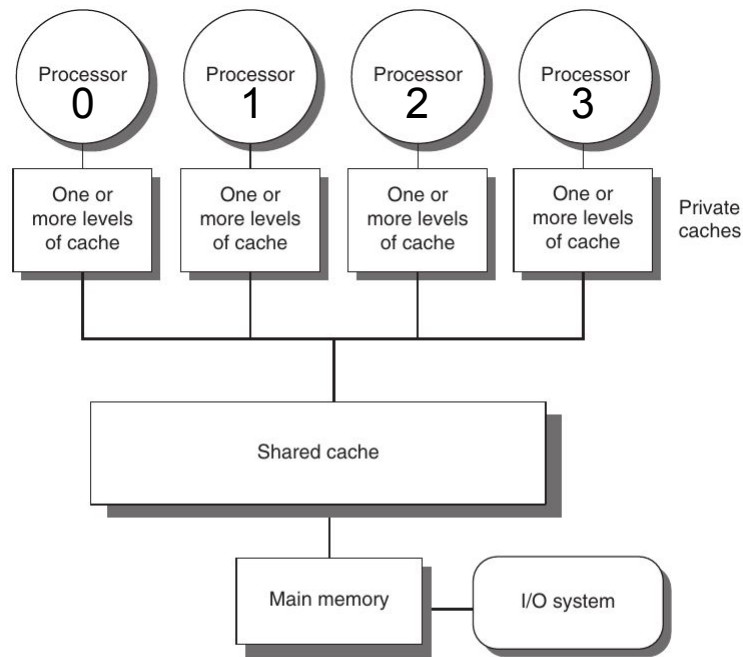
# Software running on an SMP (cont.)

- Threads that belong to the same process share the same address space
- Writes to `@x[i]` from thread 1 will be visible from thread 2 (and vice versa)
- High-level languages that support this memory model are called shared-memory programming models



# Implications of shared memory with private resources

1. Processor 0 loads  $x[i]$ 
    - Data is loaded into a register
    - Data is also kept in the private cache
  2. Processor 0 modifies  $x[i]$ 
    - Modified data is in a register
  3. Processor 0 stores  $x[i]$ 
    - If cache is copy back
      - Only private cache is updated
    - If cache is write through
      - Main memory is also updated
- What happens if any other thread tries to access  $x[i]$  while a copy is stored by processor 0?



# Sharing data in SMPs

## ■ Coherence

- *“What values can be returned by a read”*
- *“Defines the behavior of reads and writes to the same memory location”*
- If two or more copies of the same value exists, which one will be served by a read (load)?

## ■ Consistency

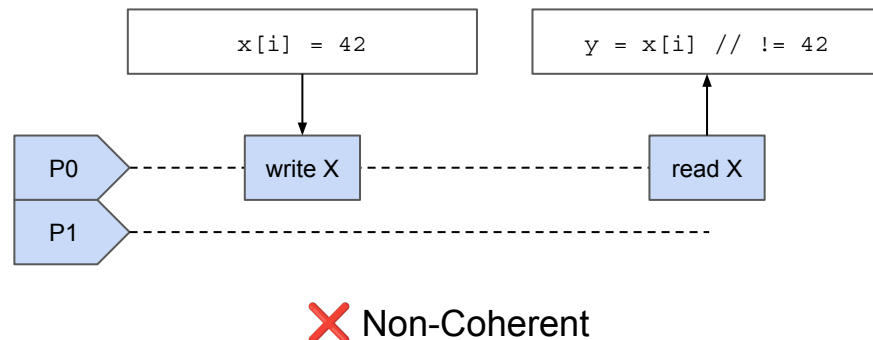
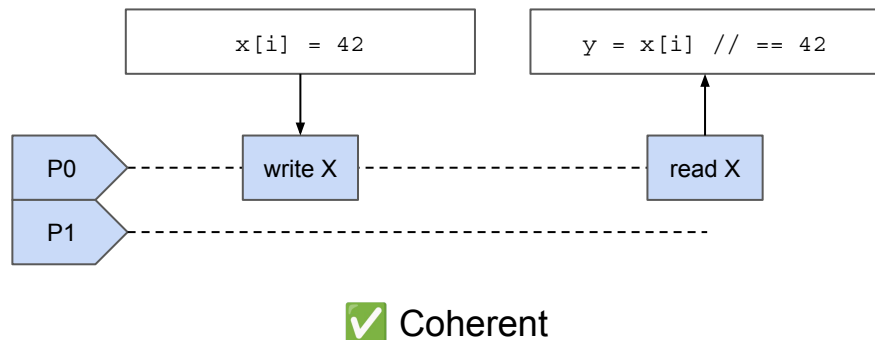
- *“When a written value will be returned by a read”*
- *“Defines the behavior of reads and writes with respect to accesses to other memory locations”*
- Once a value has been written (store), when will the modification be visible from a read (load)?

## ■ Memory Model

- Set of rules defining how coherence and consistency are implemented
- Direct impact on how load/store operations behave
- Mostly defined by the architecture (ISA)

# Memory system is coherent if...

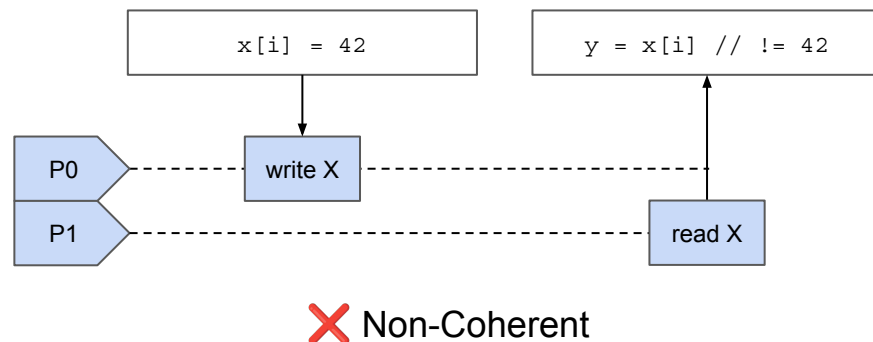
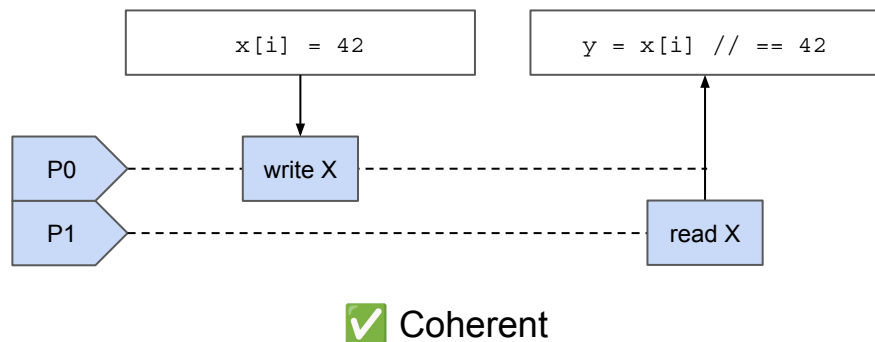
1. “A read by processor *P* to location *X* that follows a write by *P* to *X*, with no writes of *X* by another processor occurring between the write and the read by *P*, always returns the value written by *P*.”



This rule defines Program Order

# Memory system is coherent if...

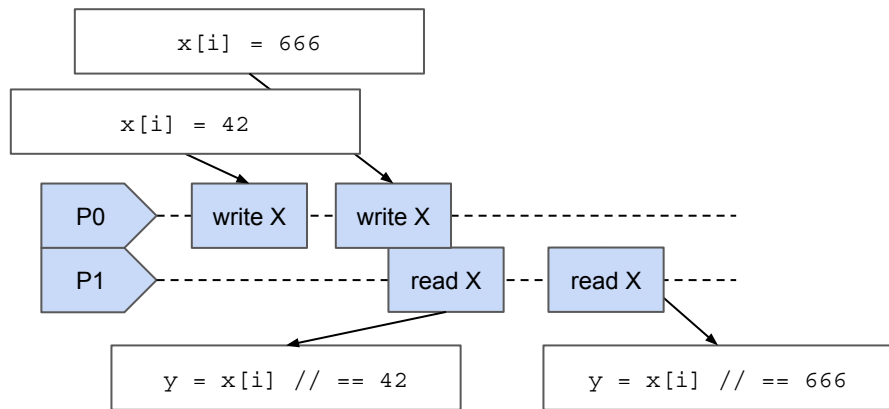
2. “A read by a processor to location  $X$  that follows a write by another processor to  $X$  returns the written value if the read and write are sufficiently separated in time and no other writes to  $X$  occur between the two accesses.”



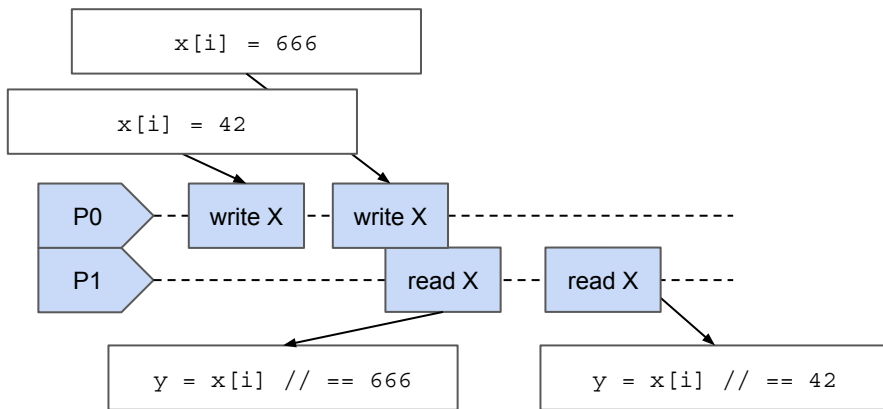
This rule defines Coherent View

# Memory system is coherent if...

3. “Writes to the same location are serialized; that is, two writes to the same location by any two processors are seen in the same order by all processors.



✓ Coherent



✗ Coherent

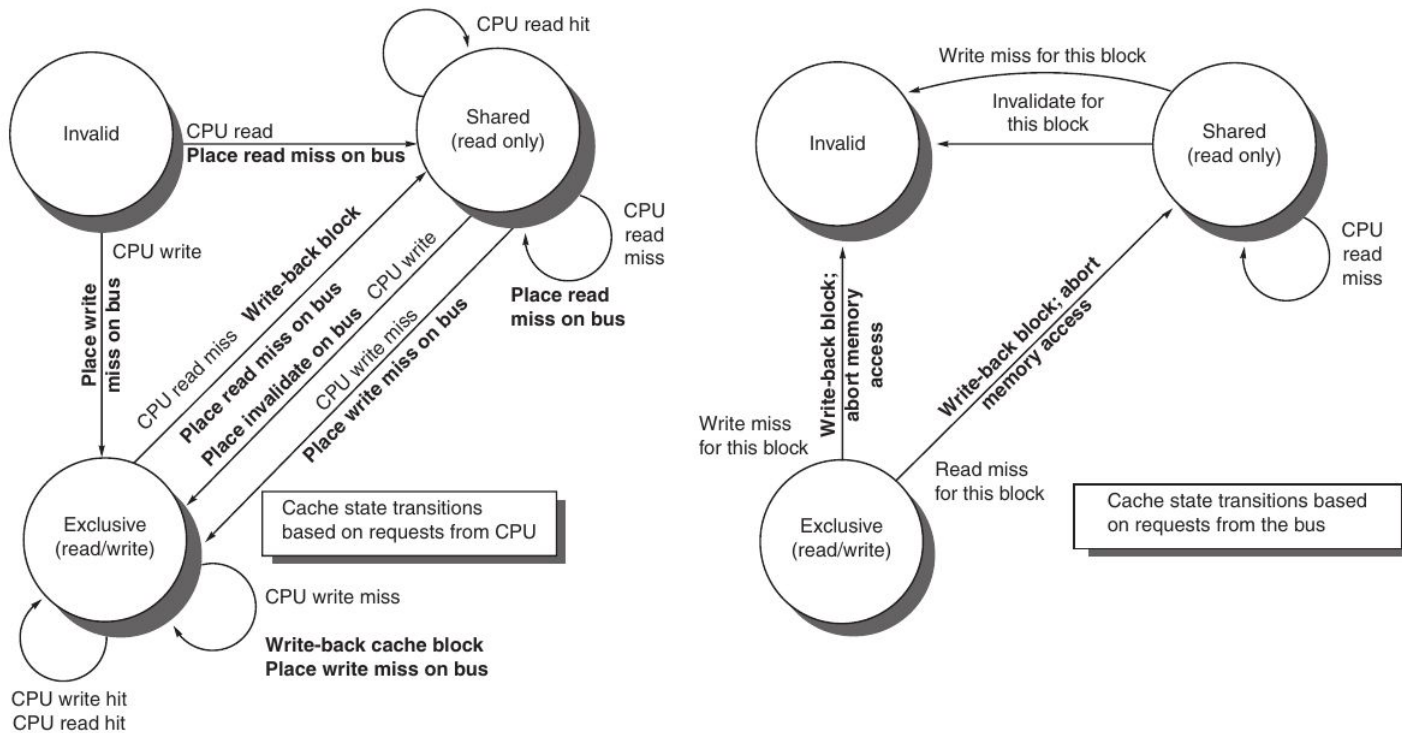
This rule defines Serialization

# Cache coherence protocols


- Hardware logic to enforce coherence between shared resources
  - Mostly transparent to the software
  - ISA might define special instructions to alter the state of shared data
  - Examples: MSI, MESI, MOESI
- What is the granularity of a block of data?
  - Typically, cache line
- Who tracks the status of a shared block of data?
  - Directory-based protocols → Single location that keeps track
  - Snooping-based protocols → Every cache keeps track and “snoops” traffic to/from other caches
- Coherence protocols are implemented as Finite-State-Machines (FSMs)
  - Blocks of data can be in one of a set of states
  - Requests from the CPU and/or the memory bus cause transitions between states



# Example of Snooping-based protocol



# Atomic transactions

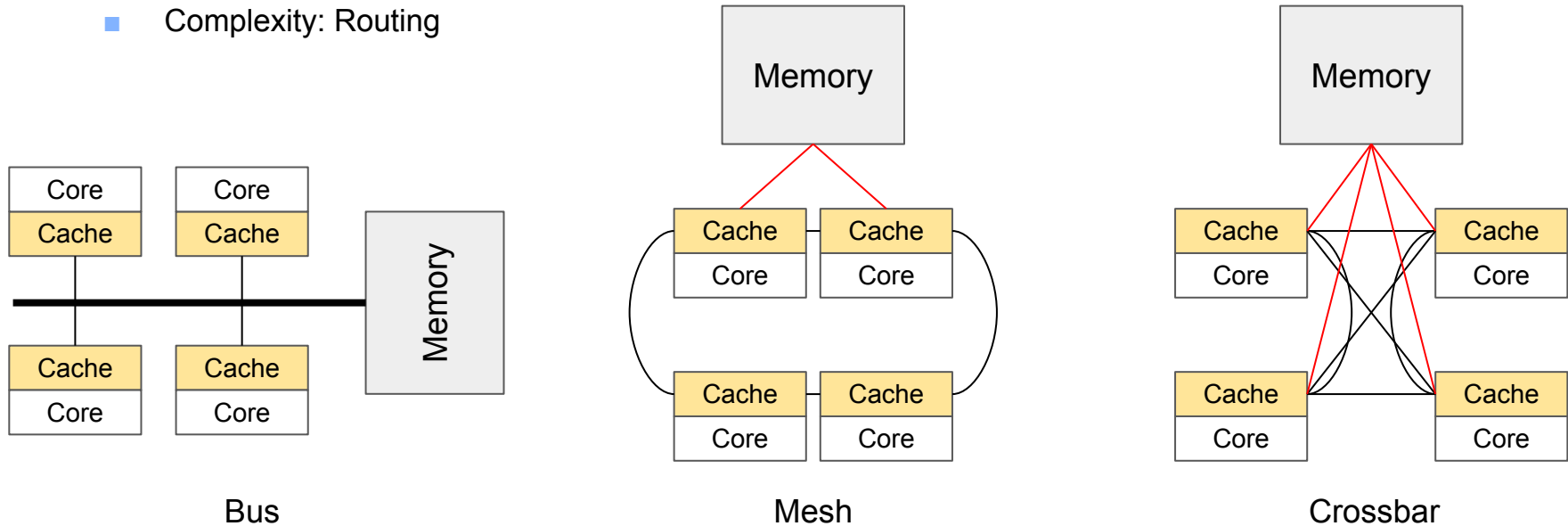
- In some circumstances, software needs to perform more than one operation on a certain memory location
  - Example: Load + conditional branch
- If other threads modify the value between the two operations... 
- ISAs provide a set of instructions that perform these operations ensuring no other thread can modify data in between
  - We call these atomic operations

# Relaxing the model

- Some ISAs define a more “relaxed” memory model
- Instead of enforcing memory coherence by hardware, Software has access to special instructions
- Examples:
  - Memory fence instruction (thread synchronization)
  - Cache line invalidation instruction

# Interconnection networks

- How caches are connected between them and to higher memory levels impacts
  - Performance: Contention, Latency, Bandwidth
  - Cost: Area
  - Complexity: Routing



# **A word on code optimization**

Focus on the important bits

# Speedup

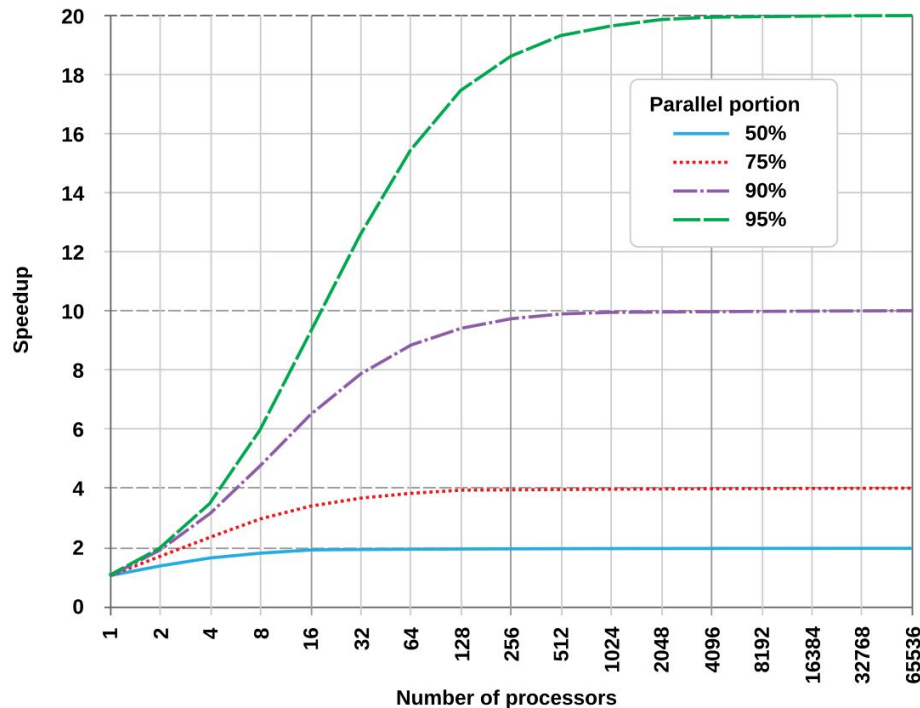
$$S = T_0 / T_1$$

- $S$  := Speedup of optimized version with respect to original version
- $T_0$  := Execution time of original version (version 0)
- $T_1$  := Execution time of optimized version (version 1)
  
- Goal: Maximize  $S$

# Amdahl's law

$$S = 1 / (1 - \alpha + \alpha/\beta)$$

- $S$  := Speedup
- $\alpha$  := Fraction of code that is optimized
- $\beta$  := Factor of optimization in region  $\alpha$
- Goal: Maximize  $S$
- Keys:  $\alpha$  is the main limiting factor to optimization



# OpenMP Programming Model

Use #pragma to win



# Source material

- [OpenMP 6.0 specification](#)
- [OpenMP 4.0 API C/C++ Syntax Quick Reference Card](#)

# OpenMP Programming Model

- Shared-memory programming model
- Thread level parallelism
- Implicit parallelism
- Programming via pragma annotations
- Some utility functions available
- Same code works for serial and parallel execution

```
#include <omp.h>
#include <stdio.h>

int main() {
    int nthreads;
    int mythread;

    #pragma omp parallel
    {
        nthreads = omp_get_num_threads();
        mythread = omp_get_thread_num();

        #pragma omp single
        {
            printf("Master thread prints\n");
        }
    }
}
```

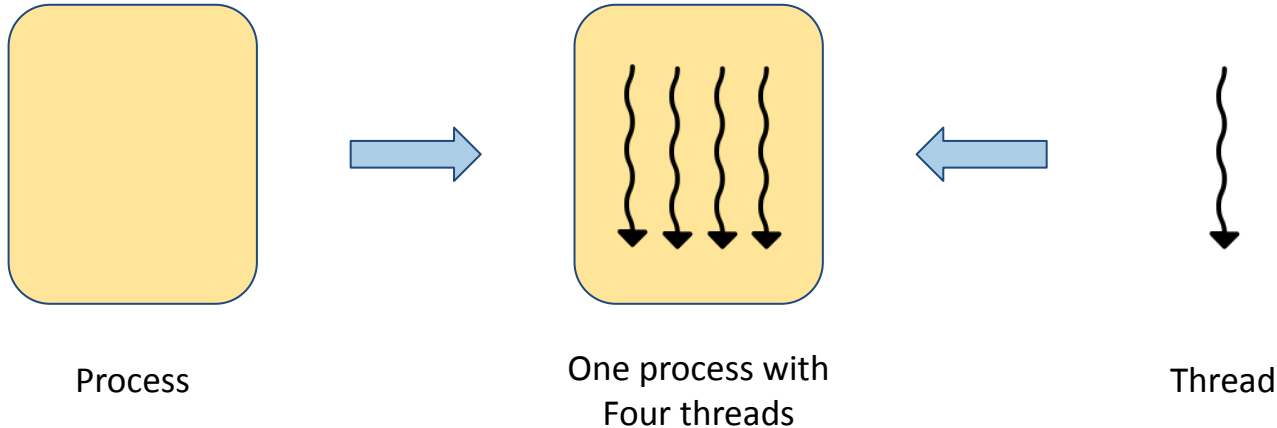
# Vocabulary

## Process

- Program in execution
- Has a Process Identifier (PID)
- Has its own memory space

## Thread

- Sequence of instructions
- Part of a thread pool
- Shares memory with other threads of same process



# OpenMP standards, compilers and libraries

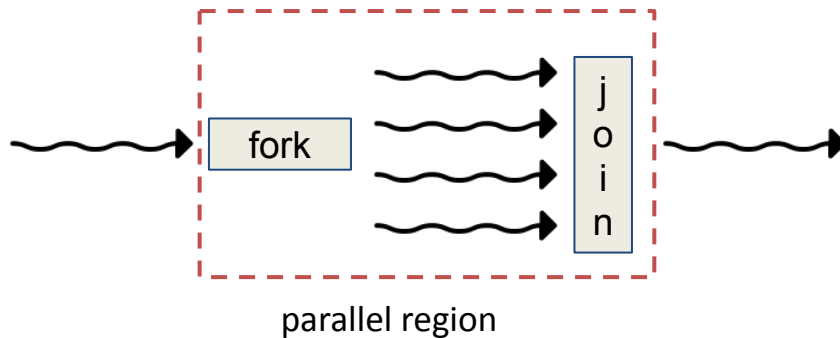
- OpenMP is a standard
  - `omp_get_num_threads`, `omp_get_thread_num`
  - `#pragma omp parallel`
- Compilers replace pragmas for calls to an OpenMP library (a.k.a runtime)
  - GNU compiler → `-fopenmp` → `libgomp.so`
  - Intel compiler → `-qopenmp` → `libiomp.so`
- Different runtimes implement the same standard with different strategies

# OpenMP environment variables and SLURM

- Environment variable to set the number of threads: `OMP_NUM_THREADS`
- Depending on SLURM configuration `OMP_NUM_THREADS`
  - Will not be set at all (user has to set it)
  - Will be set based on the allocation option `--cpus-per-task`
  - Will be set to a default value (usually the total number of cores)
- Best practices
  - If allocating via SLURM, always set `--cpus-per-task`
  - Always print/echo the value of `$OMP_NUM_THREADS`

# The Fork-Join model

- A thread team is created when a parallel region is started (fork), and when the parallel region ends the team is destroyed (join)
- Key pragma: `parallel`



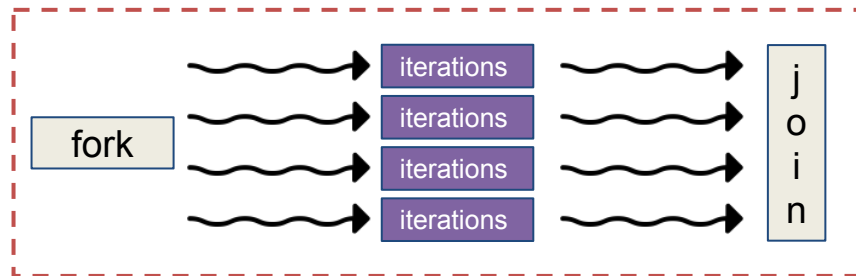
# Visibility of variables

- High-level definition of variable coherence
- Variables across threads can be...
  - Coherent (`shared`)
  - Non-coherent (`private`)
- Key pragma: `shared`, `private`

# Worksharing model

- Workload is distributed among the threads
  - The workload usually comes from a for loop
- Standard defines different scheduling options (`static`, `dynamic`, `guided`)
- Key pragmas: `for`, `scheduling()`

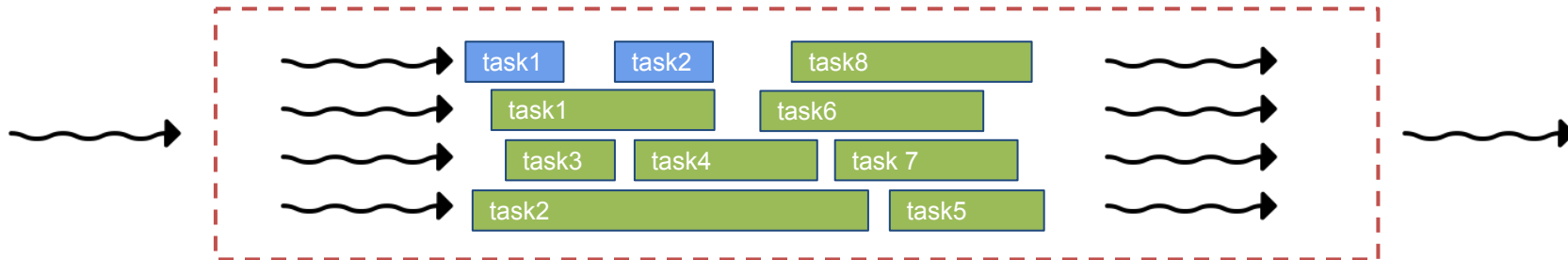
```
#pragma omp for private(i)
{
    for(i = 0; i < ITERATIONS; i++) {
        // Do stuff...
    }
}
```





# Tasking model

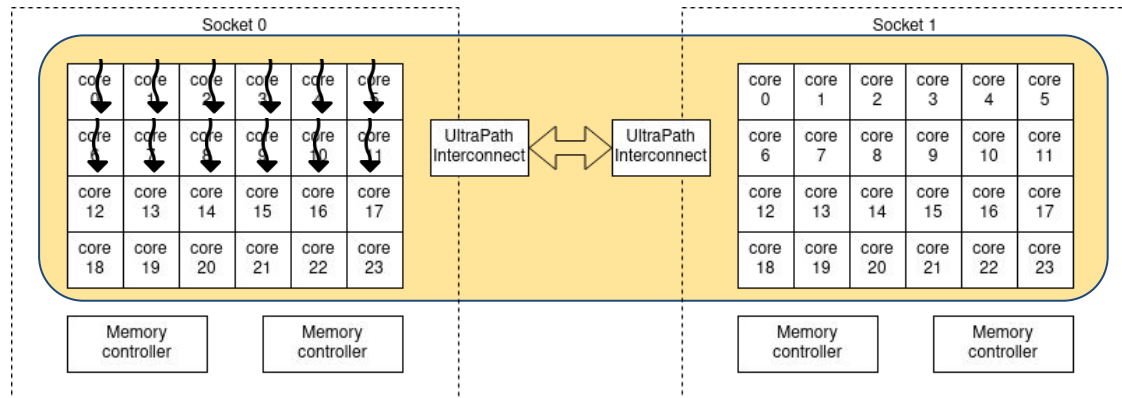
- Instead of manually distributing the workload, it is divided in pieces called tasks
- Tasks are instantiated, and then executed by any thread in the team that is available
- Key pragma: `task`



# Thread pinning

- Close

- Place threads side by side
- `OMP_PROC_BIND="close"`



- Spread

- Balance threads across sockets
- `OMP_PROC_BIND="spread"`

