

18-447

Computer Architecture  
Lecture 27: Multiprocessors

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2015, 4/6/2015

# Assignments

---

- Lab 7 out
  - Due April 17
  
- HW 6
  - Due Friday (April 10)
  
- Midterm II
  - April 24

# Where We Are in Lecture Schedule

---

- The memory hierarchy
- Caches, caches, more caches
- Virtualizing the memory hierarchy: Virtual Memory
- Main memory: DRAM
- Main memory control, scheduling
- Memory latency tolerance techniques
- Non-volatile memory
  
- **Multiprocessors**
- Coherence and **consistency**
- Interconnection networks
- Multi-core issues (e.g., heterogeneous multi-core)

# Multiprocessors and Issues in Multiprocessing

# Readings: Multiprocessing

---

## ■ Required

- Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” AFIPS 1967.

## ■ Recommended

- Mike Flynn, “Very High-Speed Computing Systems,” Proc. of IEEE, 1966
- Hill, Jouppi, Sohi, “Multiprocessors and Multicomputers,” pp. 551-560 in Readings in Computer Architecture.
- Hill, Jouppi, Sohi, “Dataflow and Multithreading,” pp. 309-314 in Readings in Computer Architecture.

# Memory Consistency

---

- **Required**

- Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, 1979

# Readings: Cache Coherence

---

## ■ Required

- Culler and Singh, *Parallel Computer Architecture*
  - Chapter 5.1 (pp 269 – 283), Chapter 5.3 (pp 291 – 305)
- P&H, *Computer Organization and Design*
  - Chapter 5.8 (pp 534 – 538 in 4<sup>th</sup> and 4<sup>th</sup> revised eds.)

## ■ Recommended:

- Papamarcos and Patel, “A low-overhead coherence solution for multiprocessors with private cache memories,” ISCA 1984.

# Remember: Flynn's Taxonomy of Computers

---

- Mike Flynn, “**Very High-Speed Computing Systems,**” Proc. of IEEE, 1966
- **SISD**: Single instruction operates on single data element
- **SIMD**: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor
- **MISD**: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor
- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor



# Why Parallel Computers?

---

- **Parallelism: Doing multiple things at a time**
- Things: instructions, operations, tasks
  
- Main (or Original) Goal
  - **Improve performance (Execution time or task throughput)**
    - Execution time of a program governed by Amdahl's Law
  
- Other Goals
  - **Reduce power consumption**
    - (4N units at freq F/4) consume less power than (N units at freq F)
    - Why?
  - **Improve cost efficiency and scalability, reduce complexity**
    - Harder to design a single unit that performs as well as N simpler units
  - **Improve dependability: Redundant execution in space**

# Types of Parallelism and How to Exploit Them

---

## Them

- Instruction Level Parallelism
  - Different instructions within a stream can be executed in parallel
  - Pipelining, out-of-order execution, speculative execution, VLIW
  - Dataflow
- Data Parallelism
  - Different pieces of data can be operated on in parallel
  - SIMD: Vector processing, array processing
  - Systolic arrays, streaming processors
- Task Level Parallelism
  - Different “tasks/threads” can be executed in parallel
  - Multithreading
  - Multiprocessing (multi-core)

# Task-Level Parallelism: Creating Tasks

---

- Partition a single problem into multiple related tasks (threads)
  - Explicitly: Parallel programming
    - Easy when tasks are natural in the problem
      - Web/database queries
    - Difficult when natural task boundaries are unclear
  - Transparently/implicitly: Thread level speculation
    - Partition a single thread speculatively
- Run many independent tasks (processes) together
  - Easy when there are many processes
    - Batch simulations, different users, cloud computing workloads
  - Does not improve the performance of a single task

# Multiprocessing Fundamentals

# Multiprocessor Types

---

- Loosely coupled multiprocessors
  - No shared global memory address space
  - Multicomputer network
    - Network-based multiprocessors
  - Usually programmed via message passing
    - Explicit calls (send, receive) for communication
- Tightly coupled multiprocessors
  - Shared global memory address space
  - Traditional multiprocessing: symmetric multiprocessing (SMP)
    - Existing multi-core processors, multithreaded processors
  - Programming model similar to uniprocessors (i.e., multitasking uniprocessor) except
    - Operations on shared data require synchronization

# Main Design Issues in Tightly-Coupled MP

---

- Shared memory synchronization
  - How to handle locks, atomic operations
- Cache coherence
  - How to ensure correct operation in the presence of private caches
- Memory consistency: Ordering of memory operations
  - What should the programmer expect the hardware to provide?
- Shared resource management
- Communication: Interconnects

# Main Programming Issues in Tightly-Coupled MP

---

- **Load imbalance**
  - How to partition a single task into multiple tasks
- **Synchronization**
  - How to synchronize (efficiently) between tasks
  - How to communicate between tasks
  - Locks, barriers, pipeline stages, condition variables, semaphores, atomic operations, ...
- **Ensuring correct operation while optimizing for performance**

# Aside: Hardware-based Multithreading

---

- Coarse grained
  - Quantum based
  - Event based (switch-on-event multithreading), e.g., switch on L3 miss
- Fine grained
  - Cycle by cycle
  - Thornton, “[CDC 6600: Design of a Computer](#),” 1970.
  - Burton Smith, “[A pipelined, shared resource MIMD computer](#),” ICPP 1978.
- Simultaneous
  - Can dispatch instructions from multiple threads at the same time
  - Good for improving execution unit utilization



# Limits of Parallel Speedup

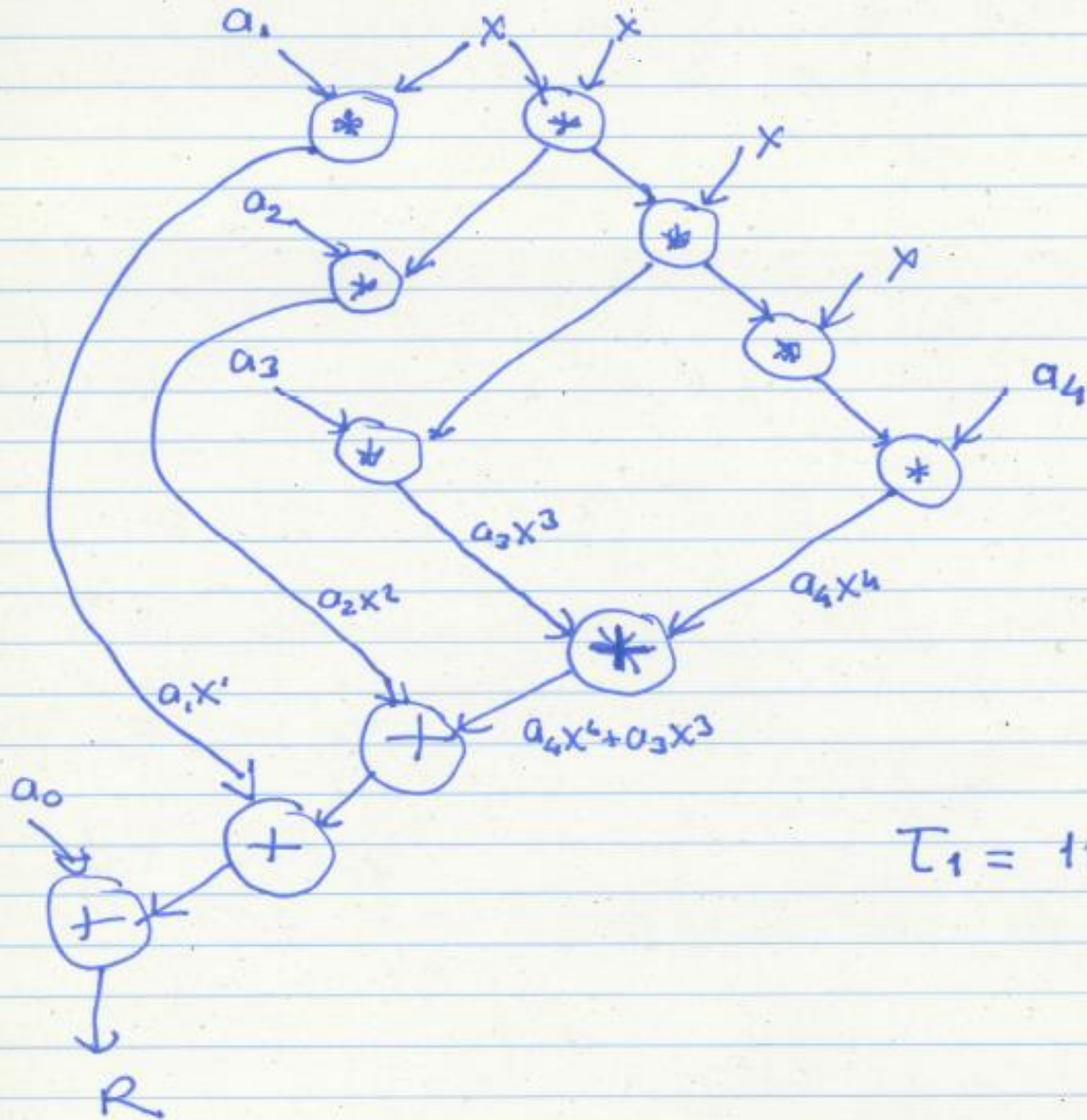
# Parallel Speedup Example

---

- $a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$
- Assume each operation 1 cycle, no communication cost, each op can be executed in a different processor
- How fast is this with a single processor?
  - Assume no pipelining or concurrent execution of instructions
- How fast is this with 3 processors?

$$R = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

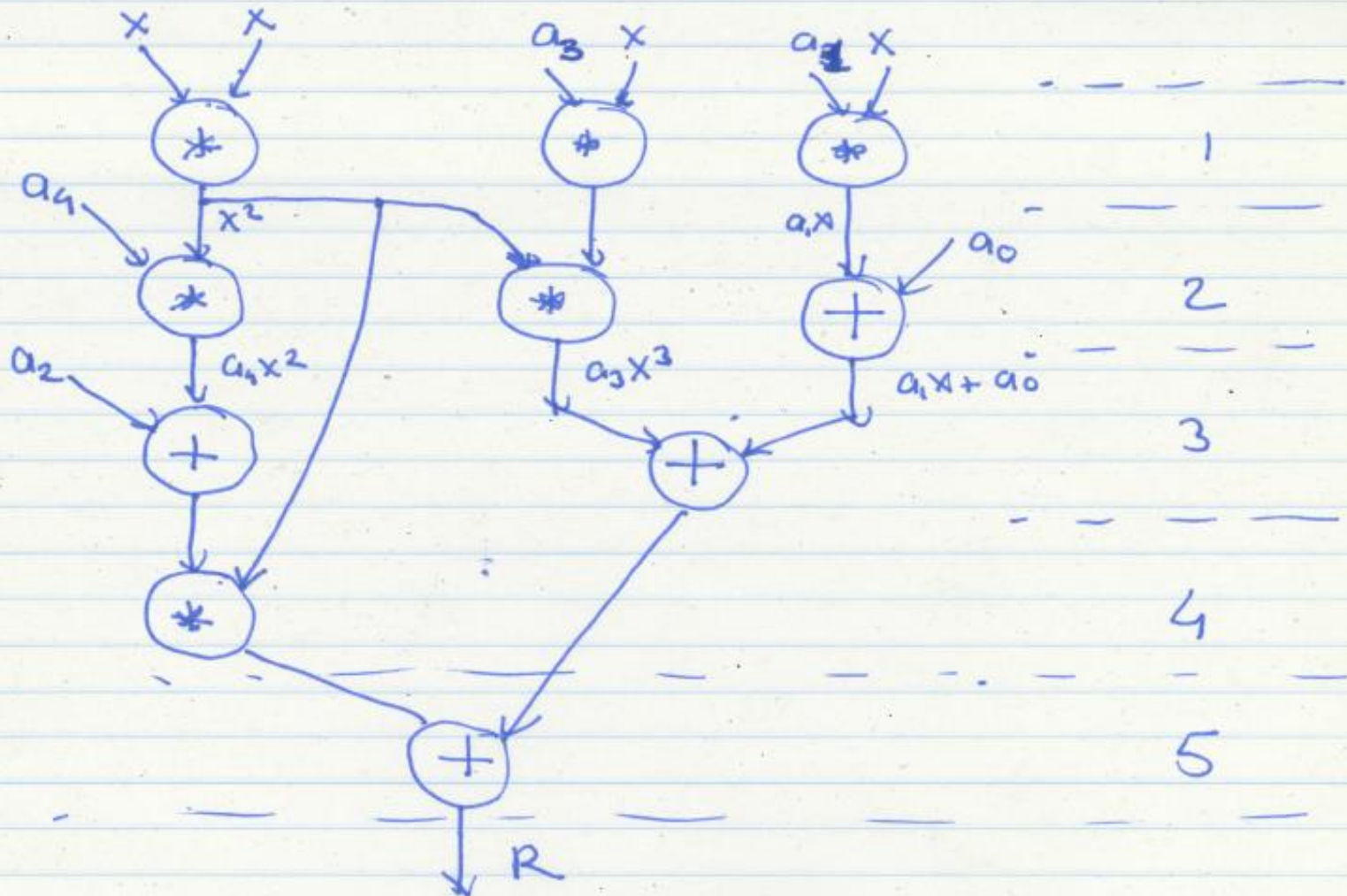
Single processor : 11 operations (data flow graph) DRAW the



$T_1 = 11$  cycles

$$R = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

Three processors :  $T_3$  (exec. time with 3 proc.)



$$T_3 = \underline{5 \text{ cycles}}$$

# Speedup with 3 Processors

---

$$T_3 = \underline{5 \text{ cycles}}$$

$$\text{Speedup with 3 processors} = \frac{11}{5} = 2.2$$

$$\left( \frac{T_1}{T_3} \right)$$

Is this a fair comparison?

# Revisiting the Single-Processor Algorithm

---

Revisit  $\tau_1$

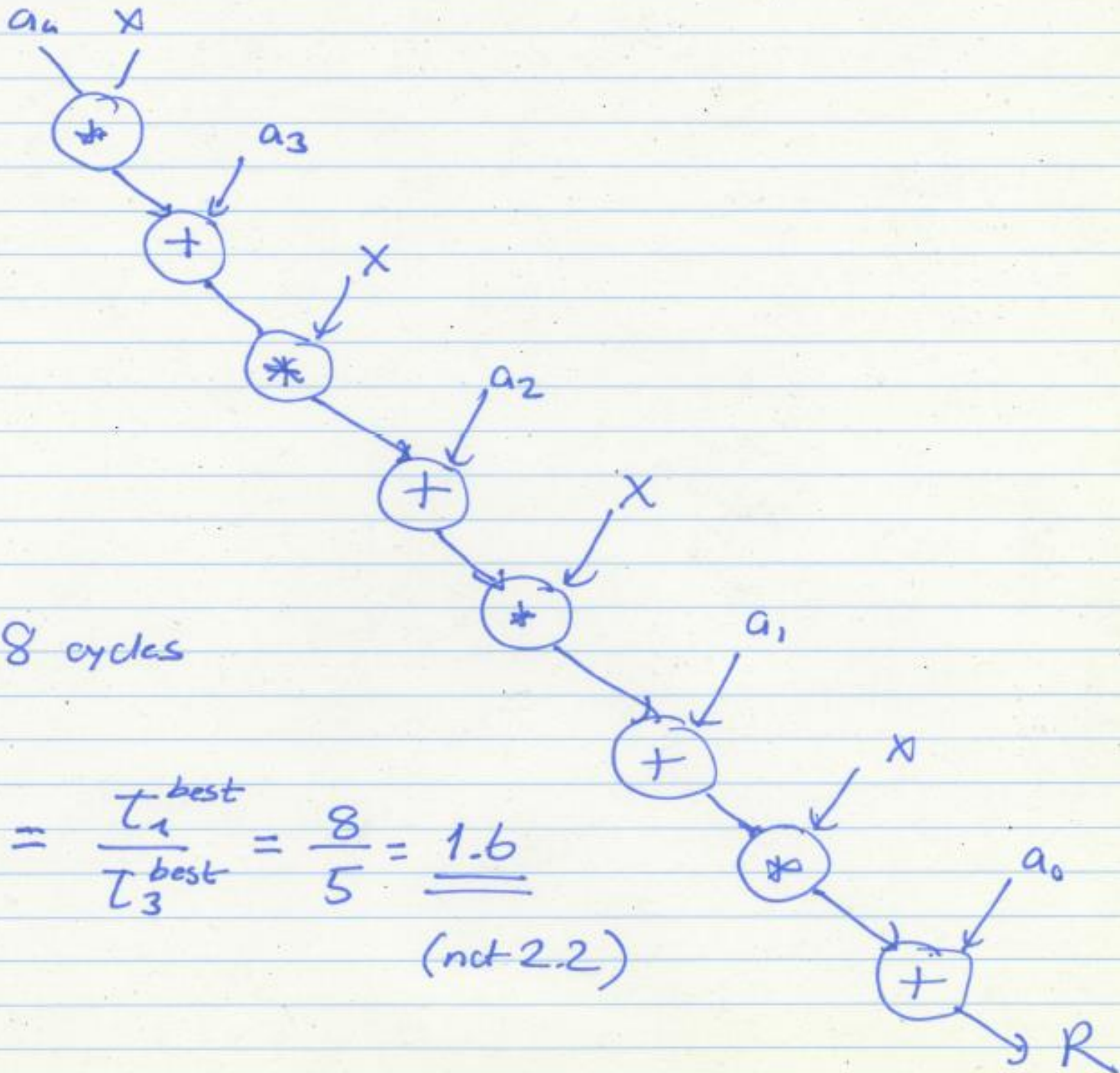
Better single-processor algorithm:

$$R = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

$$R = \left( \left( \left( a_4 x + a_3 \right) x + a_2 \right) x + a_1 \right) x + a_0$$

(Horner's method)

Horner, "A new method of solving numerical equations of all orders, by continuous approximation," Philosophical Transactions of the Royal Society, 1819.

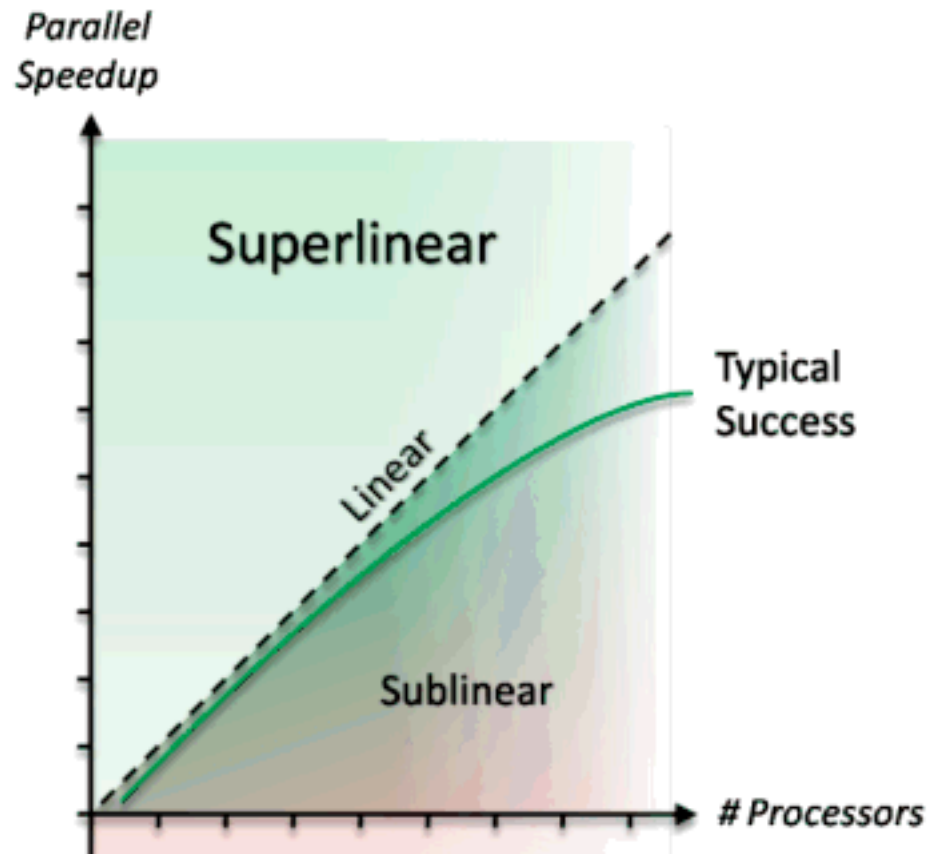


$T_1 = 8 \text{ cycles}$

Speedup with 3 procs. =  $\frac{T_1^{\text{best}}}{T_3^{\text{best}}} = \frac{8}{5} = \underline{\underline{1.6}}$   
 (not 2.2)

# Superlinear Speedup

- Can speedup be greater than  $P$  with  $P$  processing elements?
- **Unfair comparisons**  
Compare best parallel algorithm to wimpy serial algorithm  $\rightarrow$  unfair
- **Cache/memory effects**  
More processors  $\rightarrow$   
more cache or memory  $\rightarrow$   
fewer misses in cache/mem





# Utilization, Redundancy, Efficiency

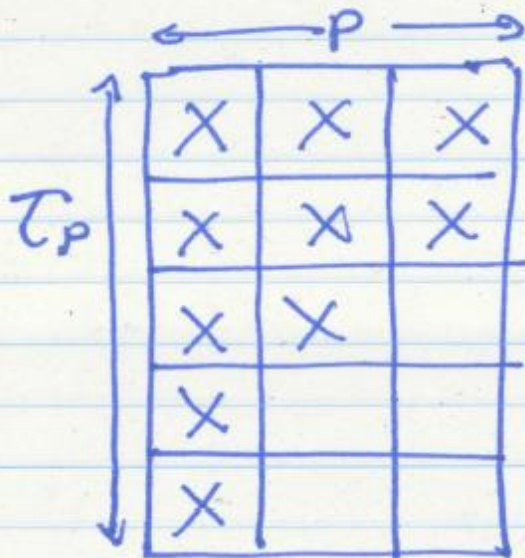
---

- Traditional metrics
  - Assume all P processors are tied up for parallel computation
- Utilization: How much processing capability is used
  - $U = (\# \text{ Operations in parallel version}) / (\text{processors} \times \text{Time})$
- Redundancy: how much extra work is done with parallel processing
  - $R = (\# \text{ of operations in parallel version}) / (\# \text{ operations in best single processor algorithm version})$
- Efficiency
  - $E = (\text{Time with 1 processor}) / (\text{processors} \times \text{Time with P processors})$
  - $E = U/R$

# Utilization of a Multiprocessor

## Multiprocessor metrics

Utilization: How much processing capability we use



$$U = \frac{10 \text{ operations (in parallel version)}}{3 \text{ processors} \times 5 \text{ time units}}$$
$$= \frac{10}{15}$$

$$U = \frac{\text{Ops with } p \text{ proc.}}{p \times T_p}$$

Redundancy: How much extra work due to multiprocessing

$$R = \frac{\text{Ops with } p \text{ proc.}^{\text{best}}}{\text{Ops with 1 proc.}^{\text{best}}} = \frac{10}{8}$$

R is always  $\geq 1$

Efficiency: How much resource we use compared to how much resource we can get away with

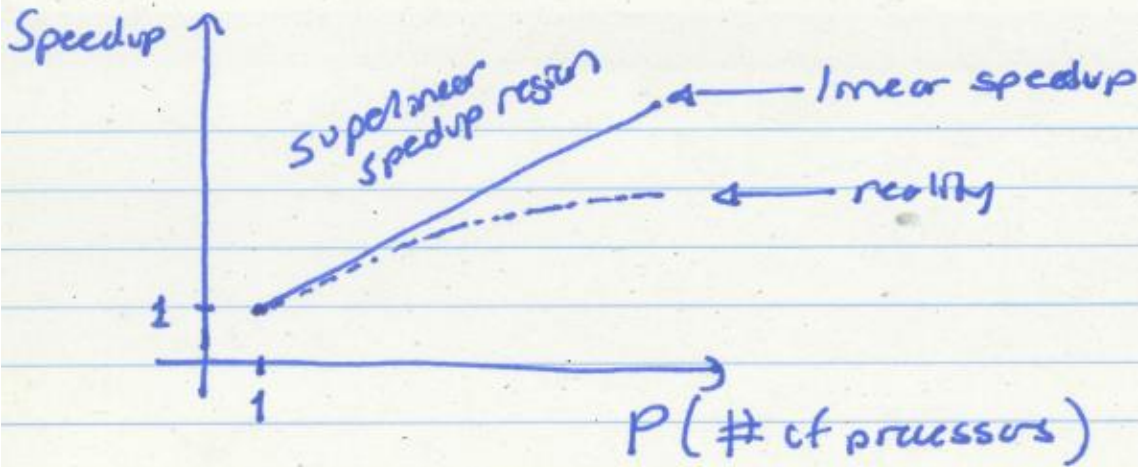
$$E = \frac{1 \cdot T_1^{\text{best}}}{p \cdot T_p^{\text{best}}}$$

(tying up 1 proc for  $T_p$  time units)  
(tying up  $p$  proc. for  $T_p$  time units)

$$= \frac{8}{15} \quad \left( E = \frac{U}{R} \right)$$

# Amdahl's Law and Caveats of Parallelism

# Caveats of Parallelism (I)



Why the reality? (diminishing returns)

$$T_p = \alpha \cdot \frac{T_1}{p} + (1 - \alpha) \cdot T_1$$

┌───┐  
└───┘  
↓  
parallelizable part/fraction  
of the single-processor  
program

┌───┐  
└───┘  
→ non-parallelizable part

# Amdahl's Law

---

$$\text{Speedup}_{\text{with } p \text{ proc.}} = \frac{T_1}{T_p} = \frac{1}{\frac{\alpha}{p} + (1-\alpha)}$$

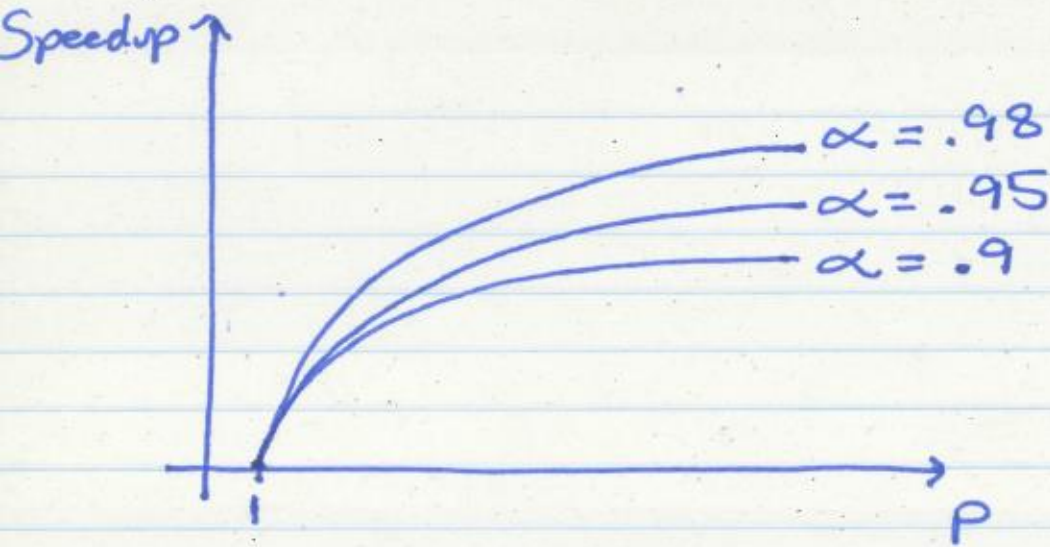
$$\text{Speedup}_{\text{as } p \rightarrow \infty} = \frac{1}{1 - \alpha}$$

$\alpha$  → bottleneck for parallel Speedup

Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

# Amdahl's Law Implication 1

---



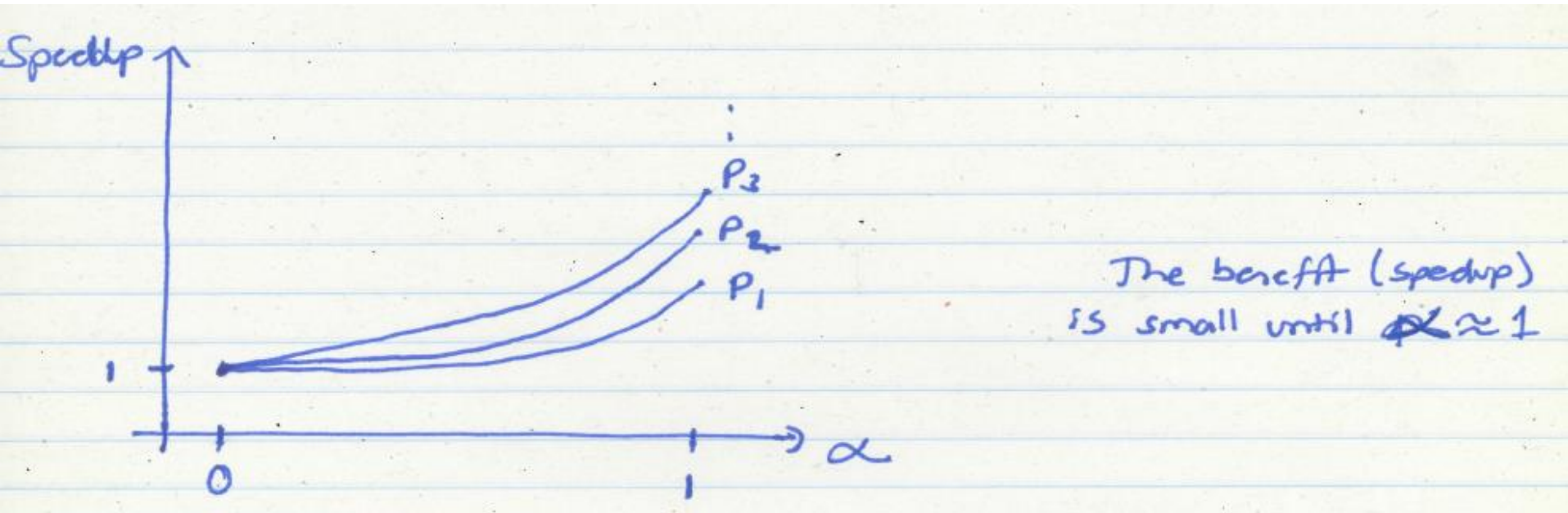
Amdahl's  
Law

illustrated

Adding more and more  
processors gives less & less  
benefit. if  $\alpha < 1$

# Amdahl's Law Implication 2

---





# Caveats of Parallelism (II)

---

## ■ Amdahl's Law

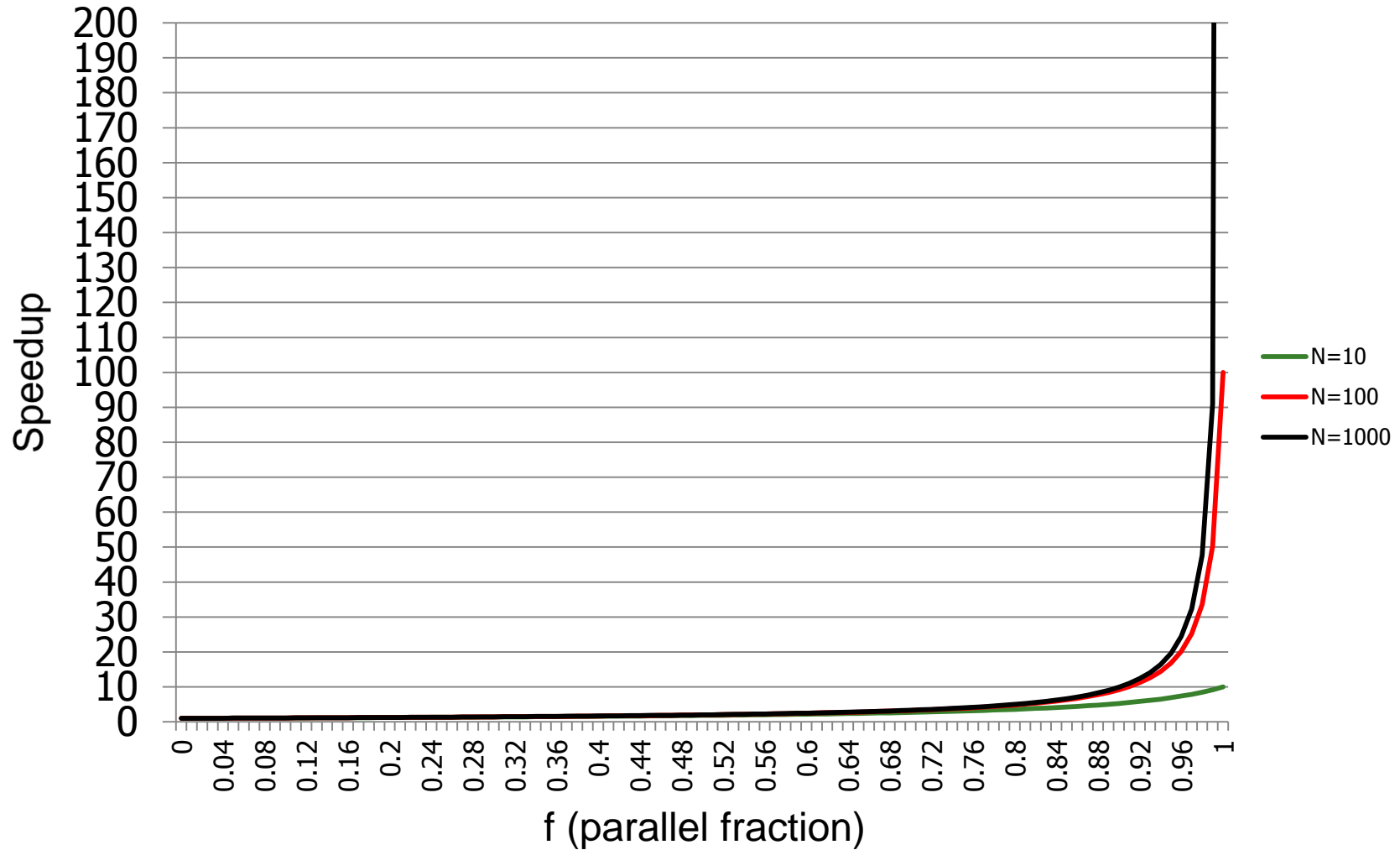
- f: Parallelizable fraction of a program
- N: Number of processors

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

- Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” AFIPS 1967.
- **Maximum speedup limited by serial portion: Serial bottleneck**
- **Parallel portion is usually not perfectly parallel**
  - **Synchronization** overhead (e.g., updates to shared data)
  - **Load imbalance** overhead (imperfect parallelization)
  - **Resource sharing** overhead (contention among N processors)

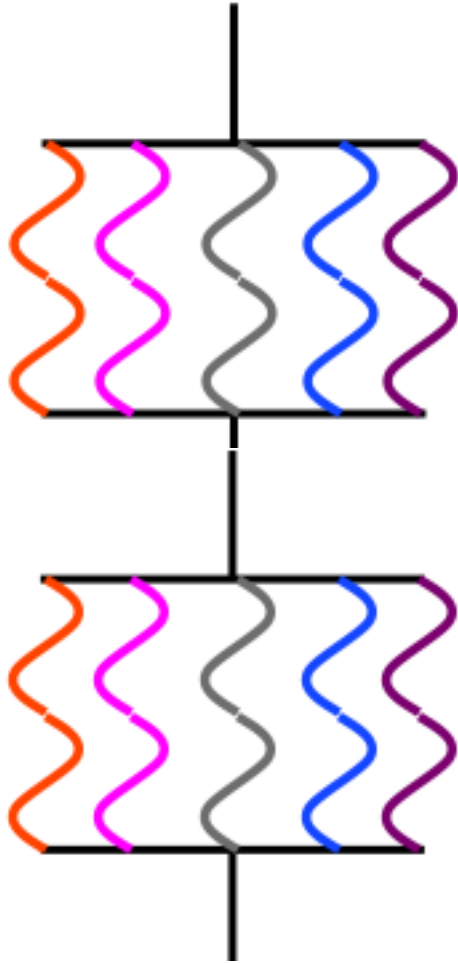
# Sequential Bottleneck

---



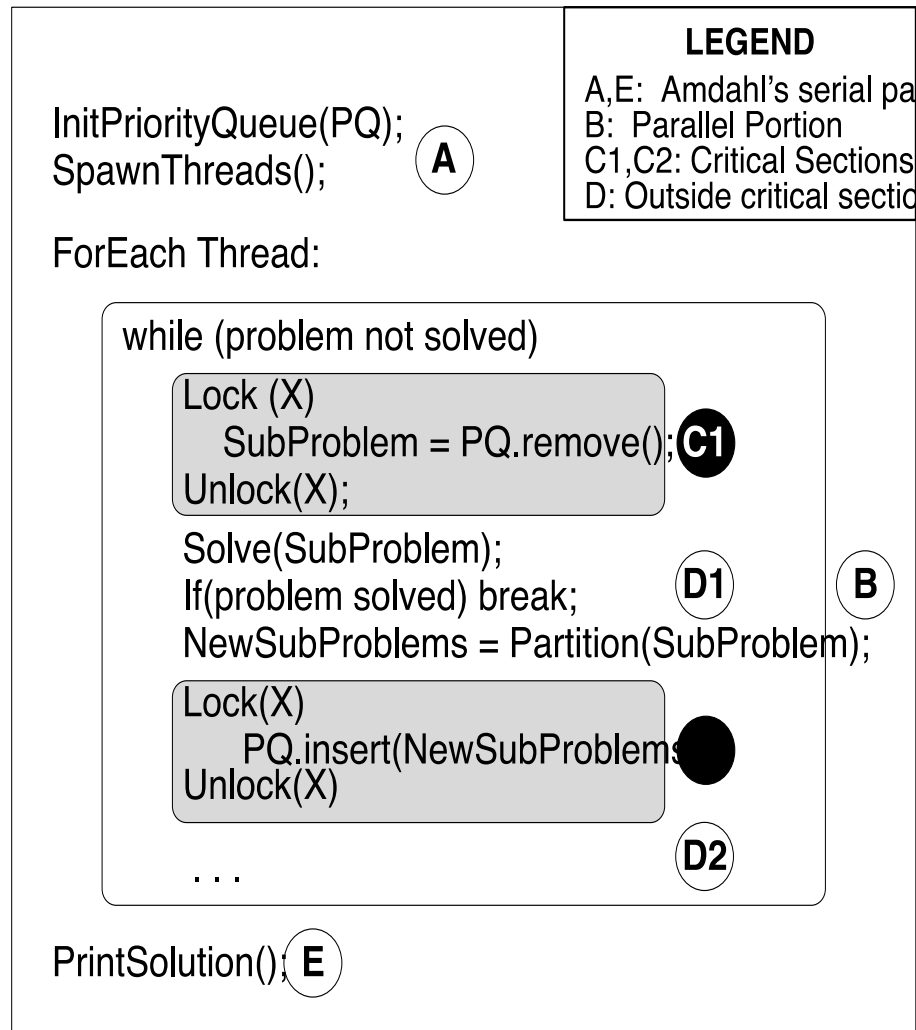
# Why the Sequential Bottleneck?

---



- Parallel machines have the sequential bottleneck
- Main cause: **Non-parallelizable operations on data** (e.g. non-parallelizable loops)  
for ( i = 0 ; i < N; i++)  
     $A[i] = (A[i] + A[i-1]) / 2$
- There are other causes as well:
  - Single thread prepares data and spawns parallel tasks (usually sequential)

# Another Example of Sequential Bottleneck



# Bottlenecks in Parallel Portion

---

- **Synchronization:** Operations manipulating shared data cannot be parallelized
  - Locks, mutual exclusion, barrier synchronization
  - **Communication:** Tasks may need values from each other
    - Causes thread serialization when shared data is contended
- **Load Imbalance:** Parallel tasks may have different lengths
  - Due to imperfect parallelization or microarchitectural effects
    - Reduces speedup in parallel portion
- **Resource Contention:** Parallel tasks can share hardware resources, delaying each other
  - Replicating all resources (e.g., memory) expensive
    - Additional latency not present when each task runs alone

# Bottlenecks in Parallel Portion: Another View

---

- Threads in a multi-threaded application can be inter-dependent
  - As opposed to threads from different applications
- Such threads can synchronize with each other
  - Locks, barriers, pipeline stages, condition variables, semaphores, ...
- Some threads can be on the critical path of execution due to synchronization; some threads are not
- Even within a thread, some “code segments” may be on the critical path of execution; some are not

# Remember: Critical Sections

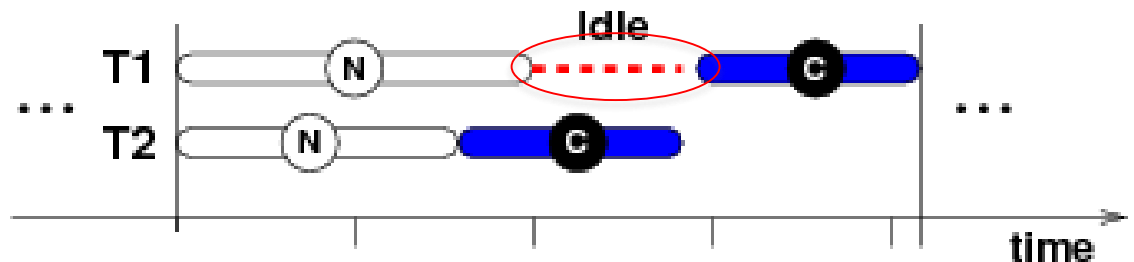
- Enforce mutually exclusive access to shared data
- Only one thread can be executing it at a time
- Contended critical sections make threads wait → threads causing serialization can be on the critical path

Each thread:

```
loop {
```

Compute	N
lock(A)	C
Update shared data	
unlock(A)	

```
}
```



# Remember: Barriers

---

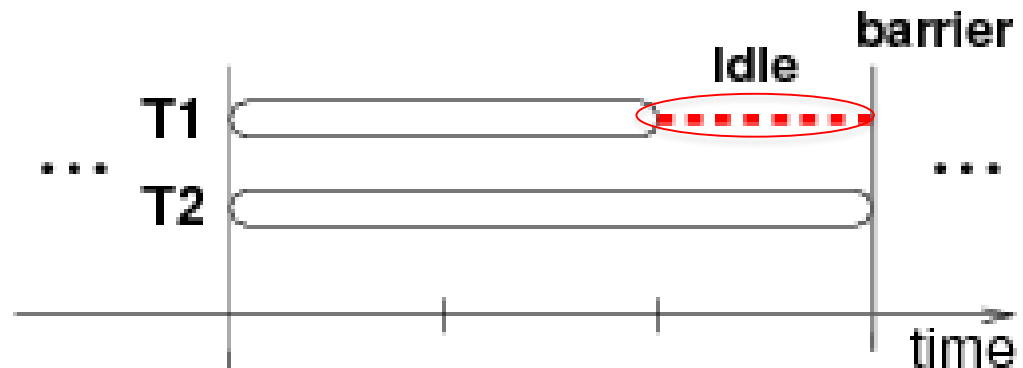
- Synchronization point
- Threads have to wait until all threads reach the barrier
- Last thread arriving to the barrier is on the critical path

Each thread:

```
loop1 {  
    Compute  
}
```

```
barrier
```

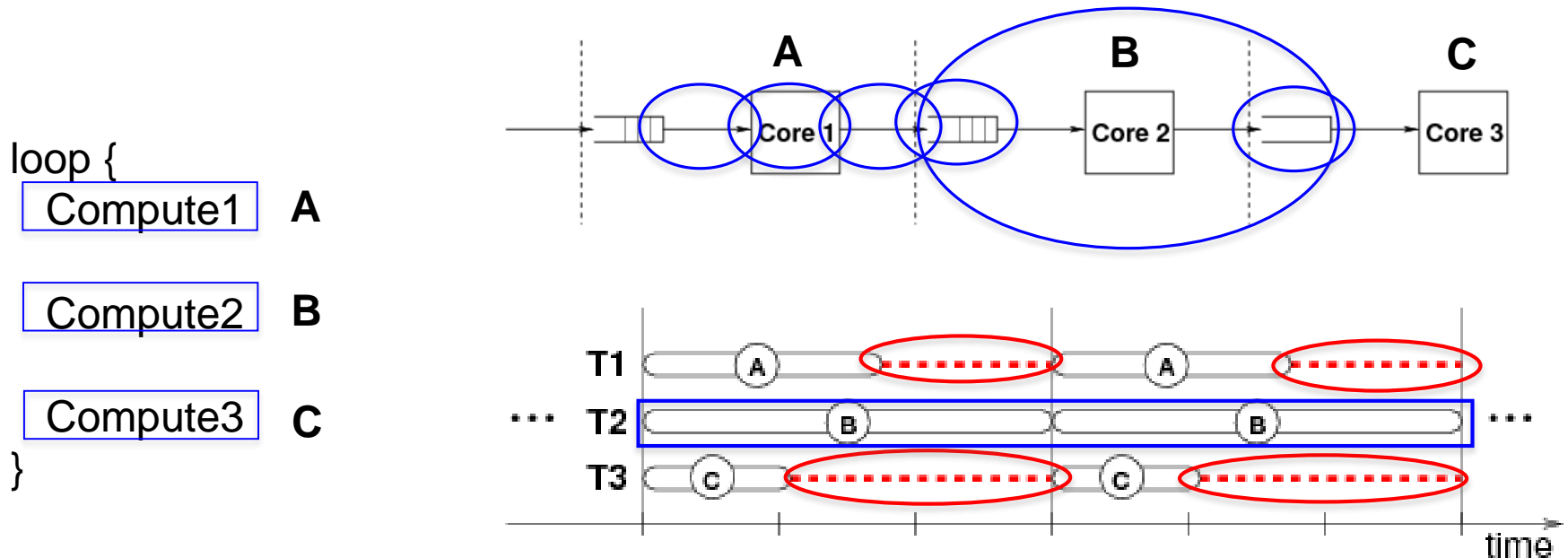
```
loop2 {  
    Compute  
}
```





# Remember: Stages of Pipelined Programs

- Loop iterations are statically divided into code segments called *stages*
- Threads execute stages on different cores
- Thread executing the slowest stage is on the critical path



# Difficulty in Parallel Programming

---

- Little difficulty if parallelism is natural
  - “Embarrassingly parallel” applications
  - Multimedia, physical simulation, graphics
  - Large web servers, databases?
- Difficulty is in
  - Getting parallel programs to work correctly
  - Optimizing performance in the presence of bottlenecks
- Much of **parallel computer architecture** is about
  - Designing machines that overcome the sequential and parallel bottlenecks to achieve higher performance and efficiency
  - Making programmer’s job easier in writing correct and high-performance parallel programs