

Computer Architecture

Lecture 19a: Multiprocessors

Prof. Onur Mutlu

ETH Zürich

Fall 2021

2 December 2021

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

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

■ Recommended:

- 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 4th and 4th revised eds.)

Multiprocessors and Issues in Multiprocessing

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

■ 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 synchronization: locks, atomic operations, barriers
- Cache coherence
 - How to ensure correct operation in the presence of private caches keeping the same memory address cached
- Memory consistency: Ordering of all 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, ...
- Contention (avoidance & management)
- Maximizing parallelism
- 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.

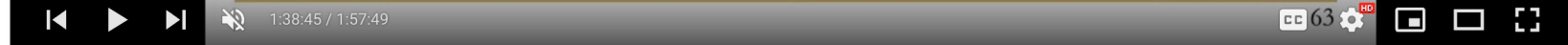
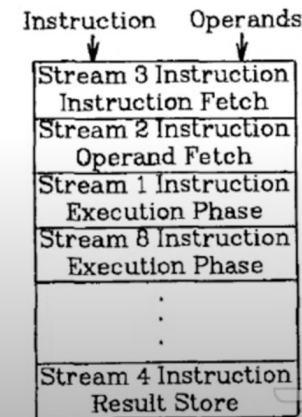
- Simultaneous
 - Can dispatch instructions from multiple threads at the same time
 - Good for improving execution unit utilization

Lecture on Fine-Grained Multithreading

Fine-Grained Multithreading

- Idea: Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread.
 - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
 - Branch/instruction resolution latency overlapped with execution of other threads' instructions

- + No logic needed for handling control and data dependences within a thread
- Single thread performance suffers
- Extra logic for keeping thread contexts
- Does not overlap latency if not enough threads to cover the whole pipeline



Onur Mutlu - Digital Design & Comp Arch - Lecture 14: Pipelined Processor Design (Spring 2021)

3,058 views • Streamed live on Apr 22, 2021

63 DISLIKE SHARE SAVE ...

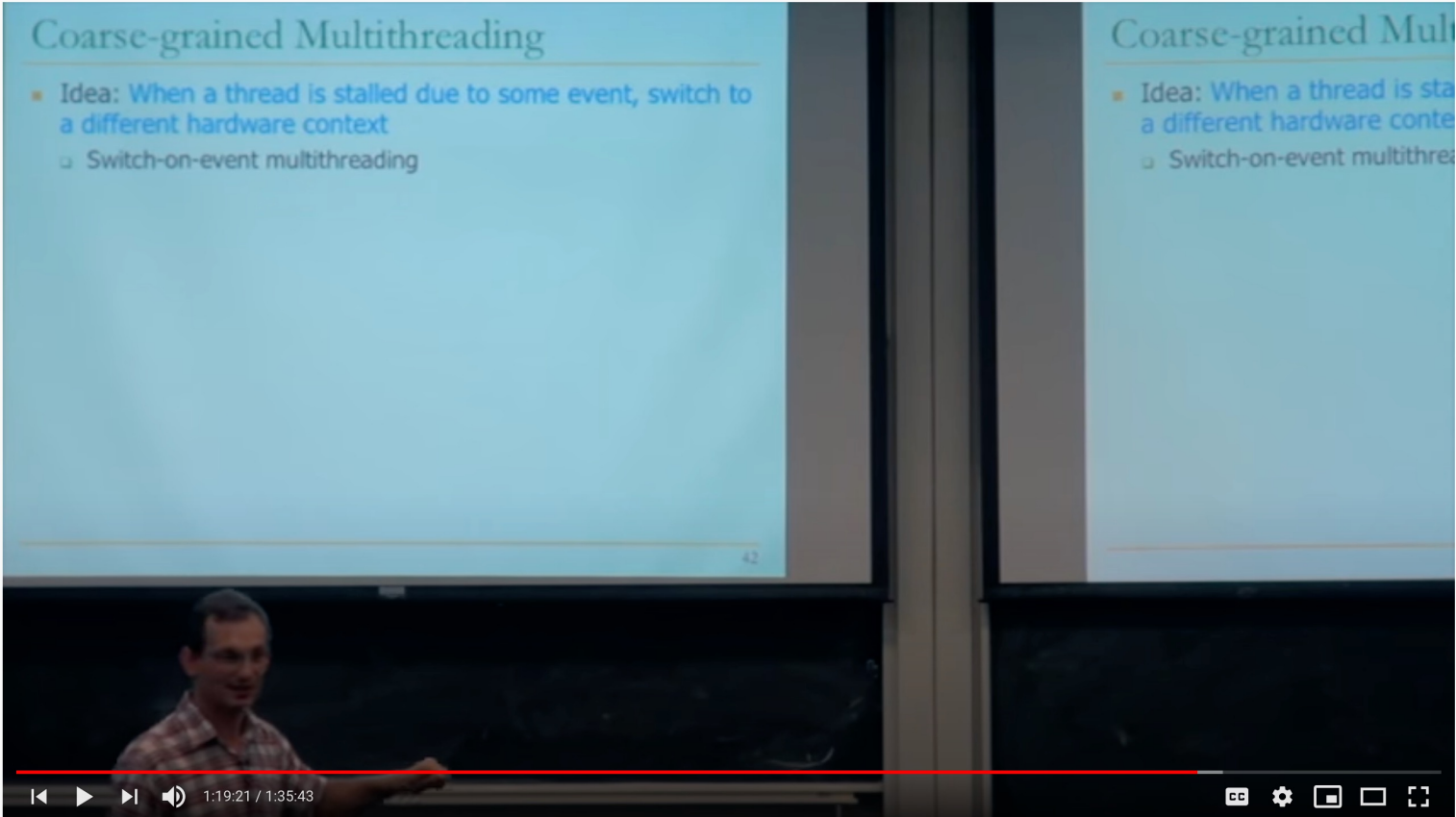


Onur Mutlu Lectures
20.4K subscribers

ANALYTICS

EDIT VIDEO

More on Multithreading (I)



The video player shows a lecture slide titled "Coarse-grained Multithreading". The slide content is as follows:


- Idea: When a thread is stalled due to some event, switch to a different hardware context
 - Switch-on-event multithreading

The video player interface includes a progress bar at 1:19:21 / 1:35:43, a like count of 10, and a comment count of 0. The video is titled "Carnegie Mellon - Parallel Computer Architecture 2013 - Onur Mutlu - Lec 9 - Multithreading" and has 1,252 views as of Nov 19, 2013. The channel is "Carnegie Mellon Computer Architecture" with 23K subscribers. The video description includes the lecture title, lecturer (Prof. Onur Mutlu), and date (September 26, 2013).

Carnegie Mellon - Parallel Computer Architecture 2013 - Onur Mutlu - Lec 9 - Multithreading

1,252 views • Nov 19, 2013

10 0 SHARE SAVE ...

 Carnegie Mellon Computer Architecture
23K subscribers

Lecture 9: Multithreading
Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)
Date: September 26, 2013.

ANALYTICS EDIT VIDEO

More on Multithreading (II)

Intel Pentium 4 Hyperthreading

- Long latency load handling
 - Multi-level scheduling window
- More partitioned structures
 - I-TLB
 - Instruction Queues
 - Store buffer
 - Reorder buffer
- 5% area overhead due to SMT

Marr et al., "Hyper-Threading Technology Architecture and Microarchitecture," Intel Technology Journal 2002.

replay loop
4 inst
soft line fetch
Instruction Buffer
I-TLB (CPU)
Accession et al.
Assigning Confidence to Conditional Branches
MICRO HT6.

Carnegie Mellon -Parallel Computer Architecture 2012 - Onur Mutlu - Lecture 10 - Multithreading II

1,594 views • Sep 21, 2013

11 0 SHARE SAVE ...



Carnegie Mellon Computer Architecture
1.81K subscribers

Lecture 10: Multithreading II

Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)

Date: September 28, 2012.

SUBSCRIBED



More on Multithreading (III)

g (Tandem, Compaq Himalaya)

Microprocessor R1 ← (R2) and R2 ← (R1)

Input Register and Output Register

Memory covered by ECC
RAID array covered by parity
Servernet covered by CRC

the processor, compare the results of two
re committing an instruction

Lockstepping (Tandem, Compaq Himalaya)

Microprocessor R1 ← (R2) and R2 ← (R1)

Input Register and Output Register

Memory covered by ECC
RAID array covered by parity
Servernet covered by CRC

Idea: Replicate the processor, compare the results of two
processors before committing an instruction

Carnegie Mellon - Parallel Computer Architecture 2013 - Onur Mutlu - Lec 13-Multi-threading II

1,132 views • Sep 21, 2013

8 0 SHARE SAVE ...



Carnegie Mellon Computer Architecture
1.81K subscribers

Lecture 13: Multi-threading III

Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)

Date: October 5, 2012.

SUBSCRIBED



<https://www.youtube.com/onurmutlulectures>

More on Multithreading (IV)



Carnegie Mellon - Parallel Computer Architecture 2013 - Onur Mutlu - Lec 15 - Speculation 1

915 views • Sep 21, 2013

👍 9 💬 0 ➦ SHARE ⚙️ SAVE ...



Carnegie Mellon Computer Architecture
1.81K subscribers

SUBSCRIBED



Lecture 15: Speculation I

Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)

Date: October 10, 2012.

<https://www.youtube.com/onurmutlulectures>

Lectures on Multithreading

■ Parallel Computer Architecture, Fall 2012, Lecture 9

- ❑ Multithreading I (CMU, Fall 2012)
- ❑ https://www.youtube.com/watch?v=iqi9wFqFiNU&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnnyV6D&index=51

■ Parallel Computer Architecture, Fall 2012, Lecture 10

- ❑ Multithreading II (CMU, Fall 2012)
- ❑ https://www.youtube.com/watch?v=e8lfl6MbILg&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnnyV6D&index=52

■ Parallel Computer Architecture, Fall 2012, Lecture 13

- ❑ Multithreading III (CMU, Fall 2012)
- ❑ https://www.youtube.com/watch?v=7vkDpZ1-hHM&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnnyV6D&index=53

■ Parallel Computer Architecture, Fall 2012, Lecture 15

- ❑ Speculation I (CMU, Fall 2012)
- ❑ https://www.youtube.com/watch?v=-hbmzIDe0sA&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnnyV6D&index=54

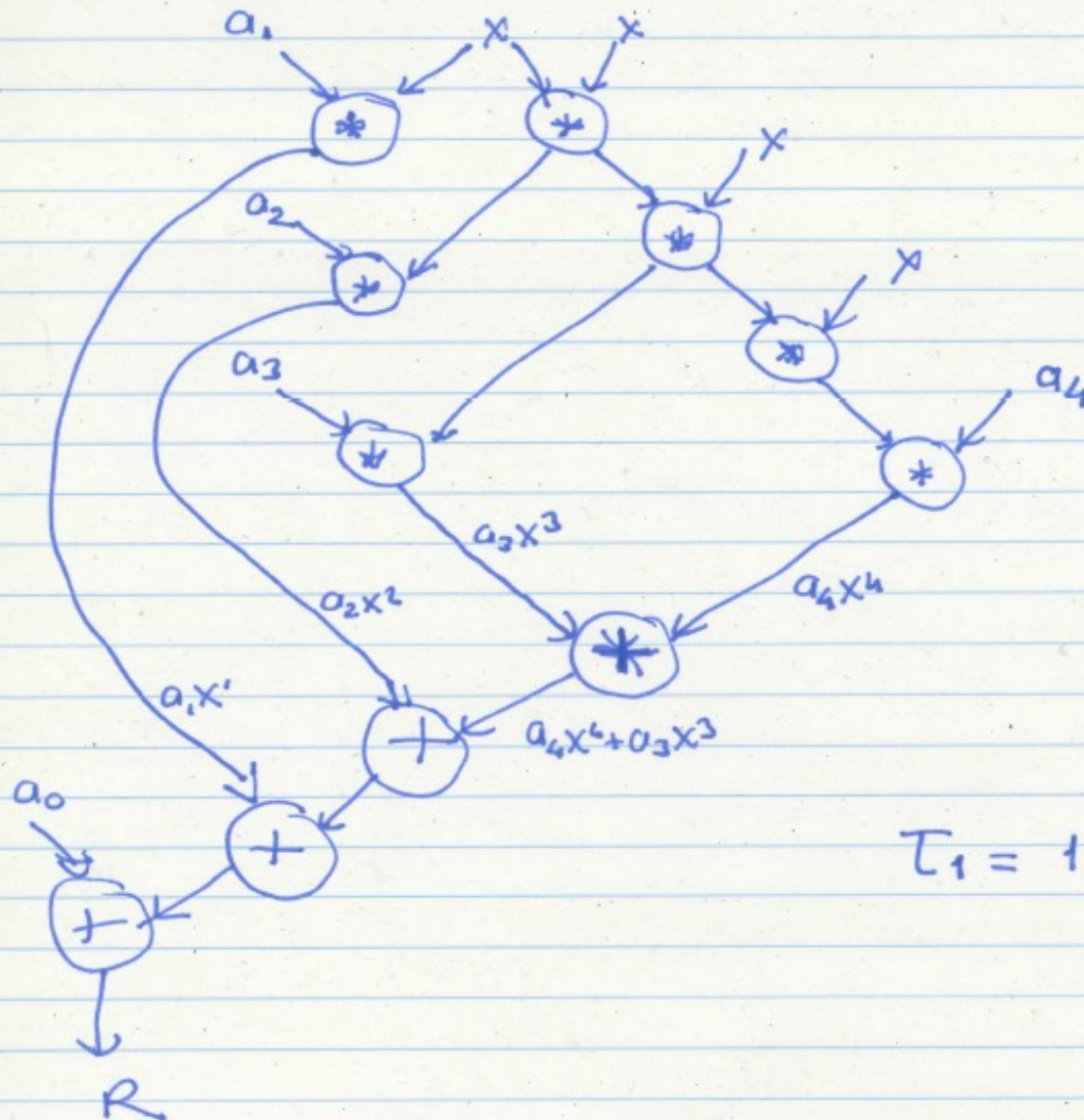
Limits of Parallel Speedup

Parallel Speedup Example

- $a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$
- Assume given inputs: x and each a_i
- 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?

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

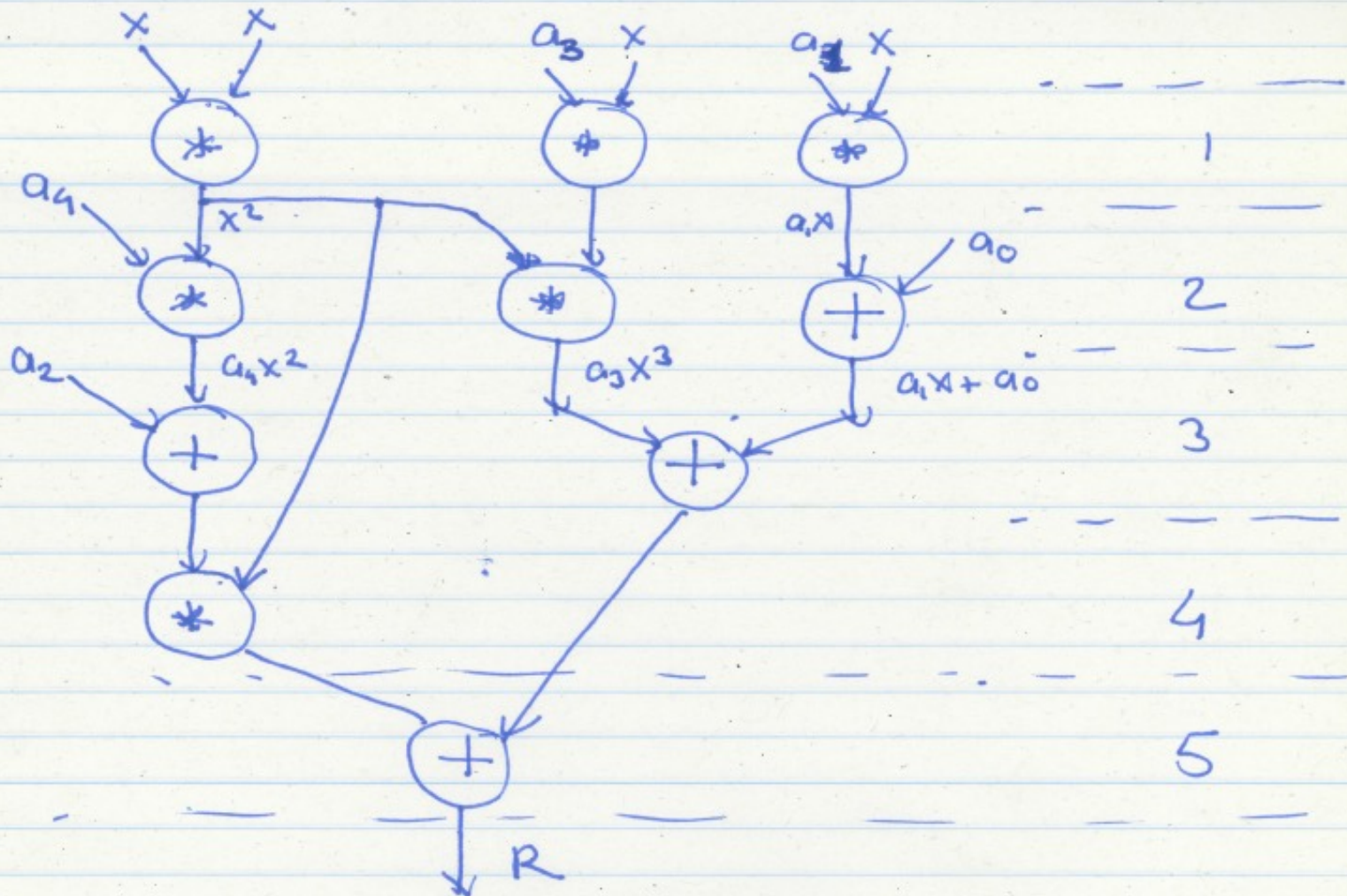
Single processor : 11 operations (data flow graph) ^{draw the}



$T_1 = 11$ 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 T_1

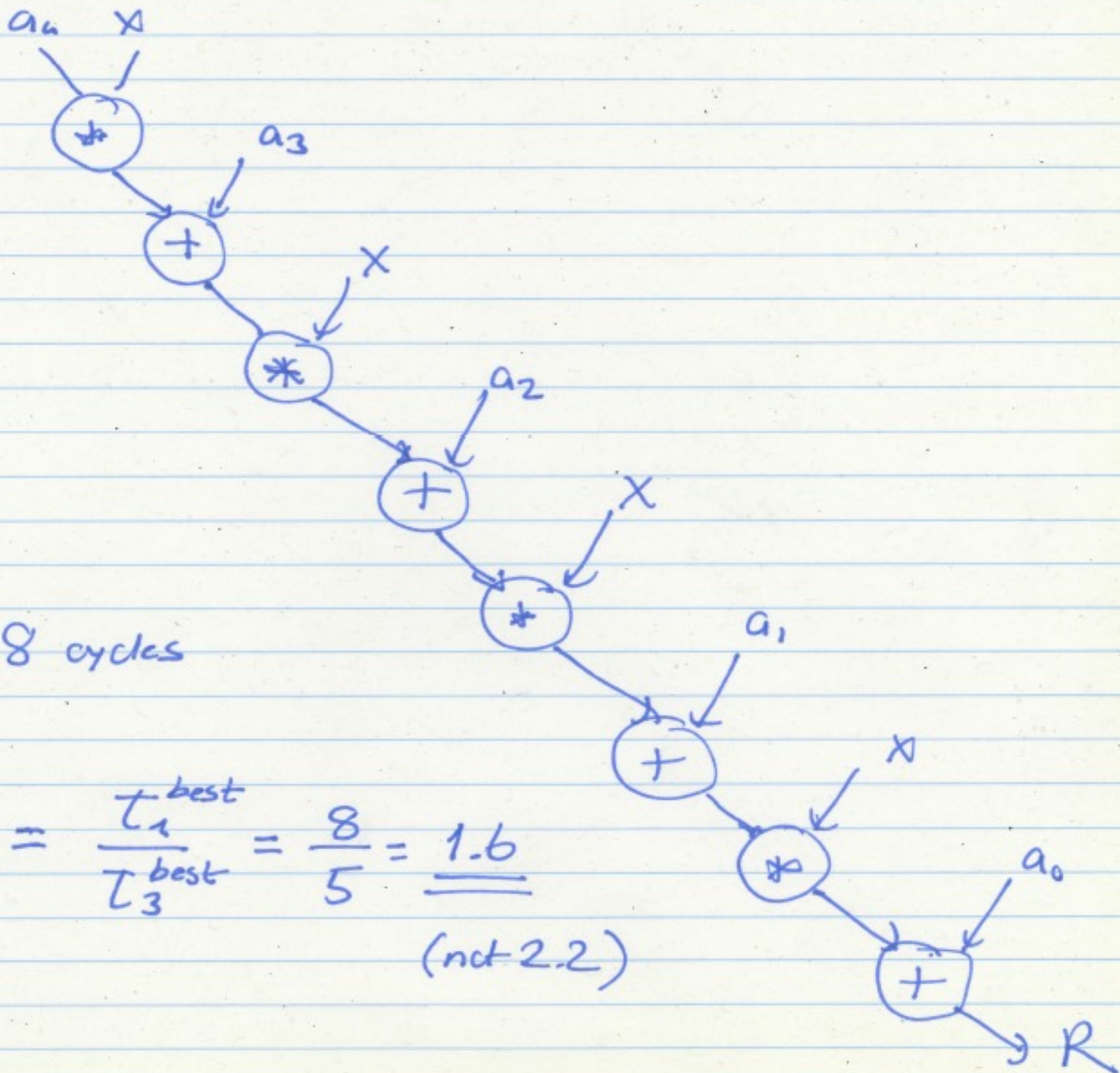
Better single-processor algorithm:

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

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

(Horner's method)

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

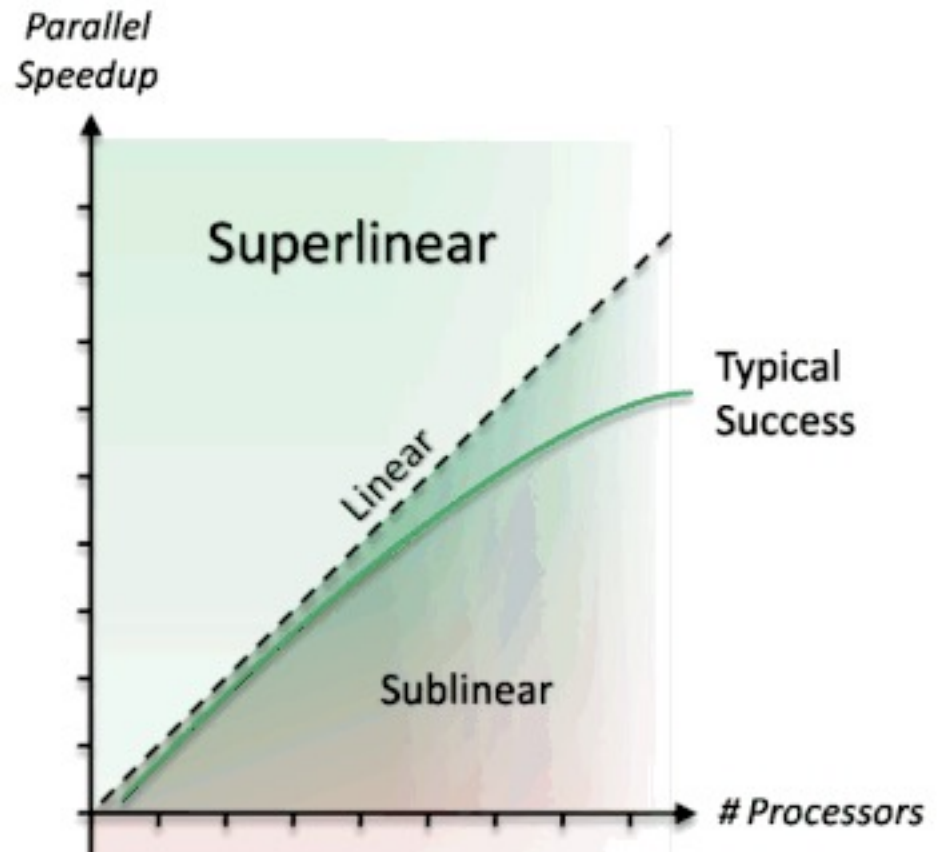


$T_1 = 8$ cycles

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 → unfair
- **Cache/memory effects**
More processors →
more cache or memory →
fewer misses in cache/mem



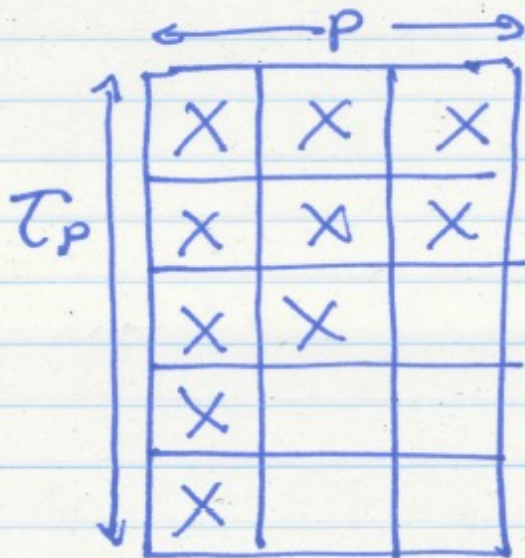
Utilization, Redundancy, Efficiency

- 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 ≥ 1

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

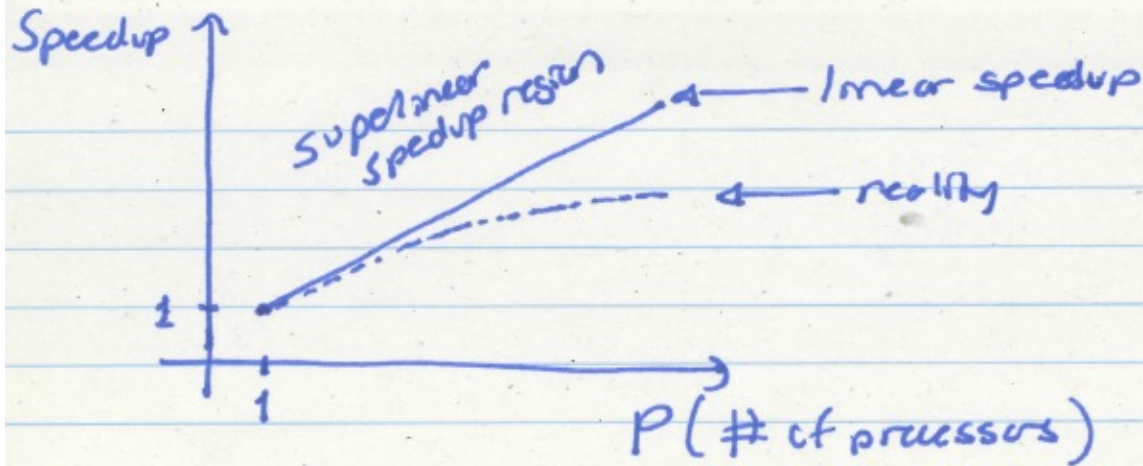
$$E = \frac{1 \cdot T_1^{\text{best}}}{p \cdot T_p^{\text{best}}} \quad \begin{array}{l} \text{(tying up 1 proc for } T_p \text{ time units)} \\ \text{(tying up } p \text{ proc. for } T_p \text{ time units)} \end{array}$$

$$= \frac{8}{15}$$

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

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

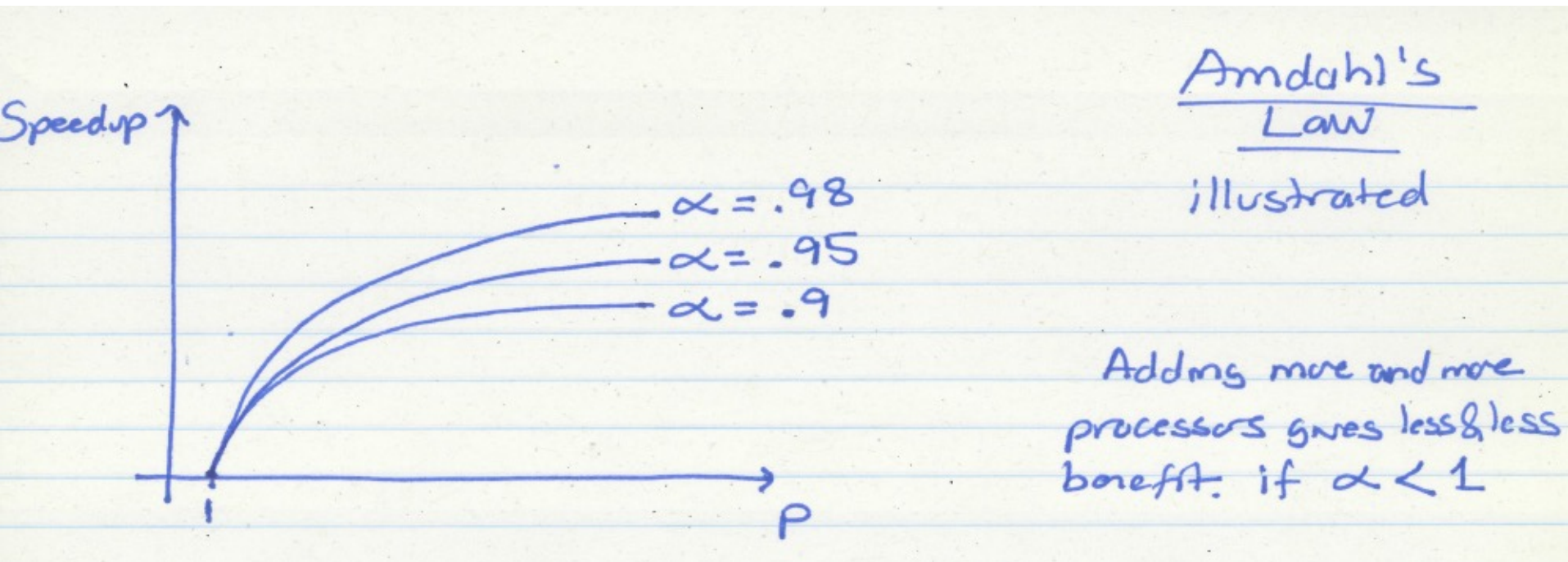
$$\text{Speedup}_{\text{with } p \text{ proc.}} = \frac{T_1}{T_p} = \frac{1}{\frac{\alpha}{p} + (1-\alpha)}$$

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

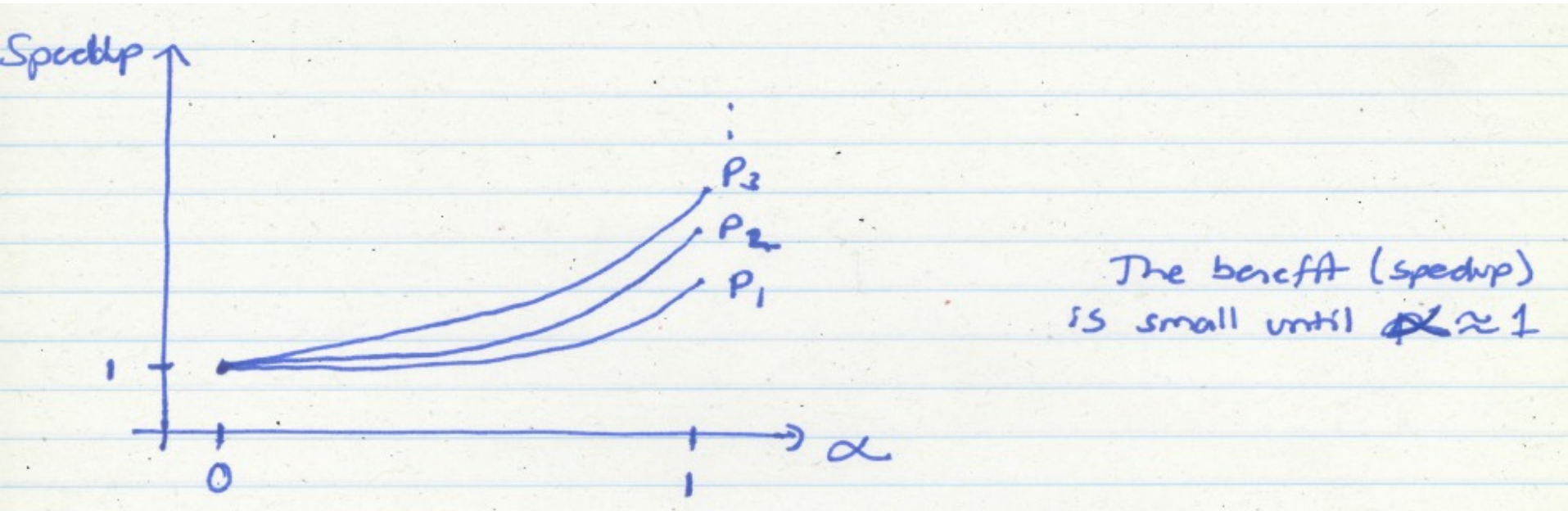
α → bottleneck for parallel Speedup

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

Amdahl's Law Implication 1



Amdahl's Law Implication 2



Caveats of Parallelism (II)

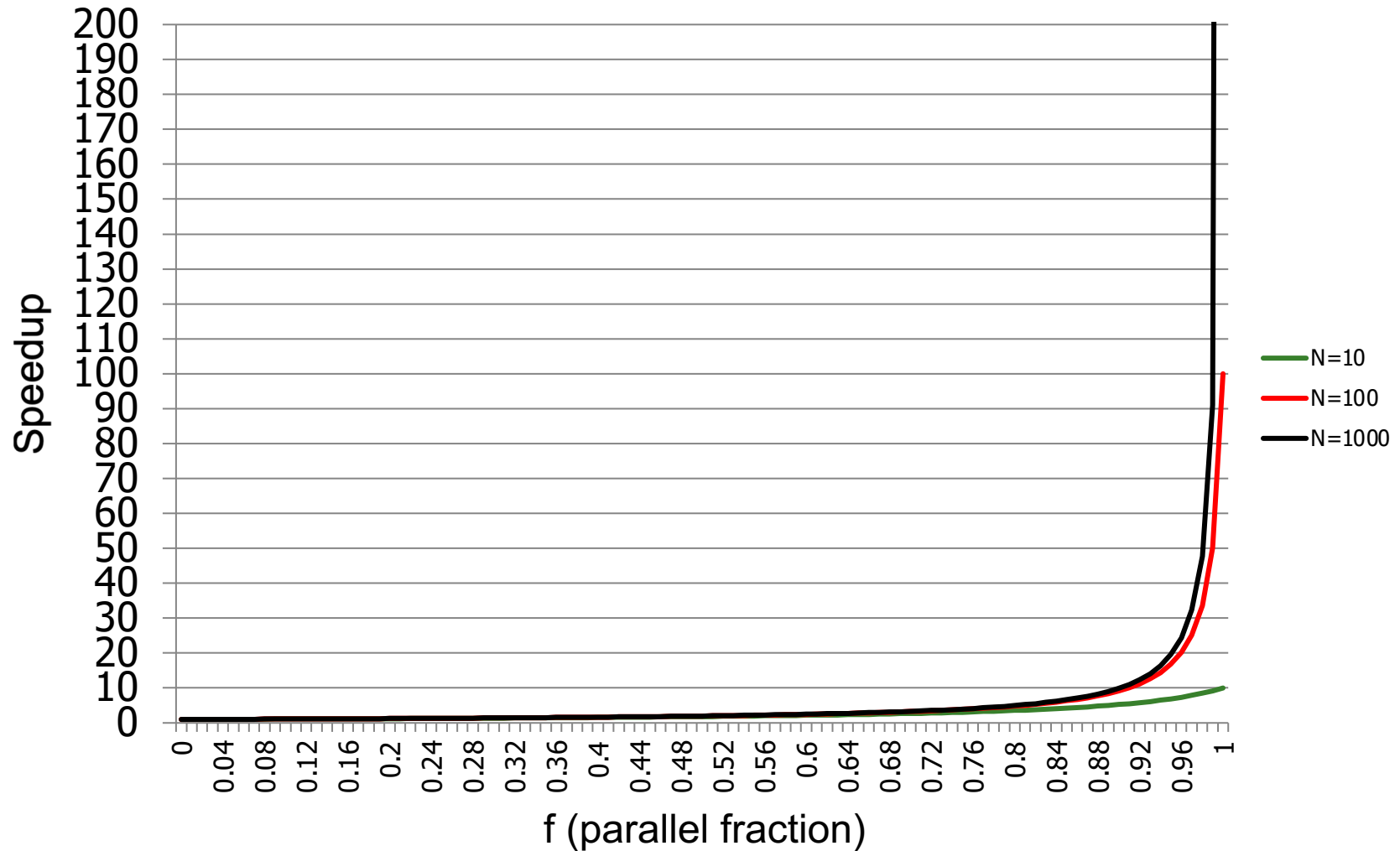
■ Amdahl's Law

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

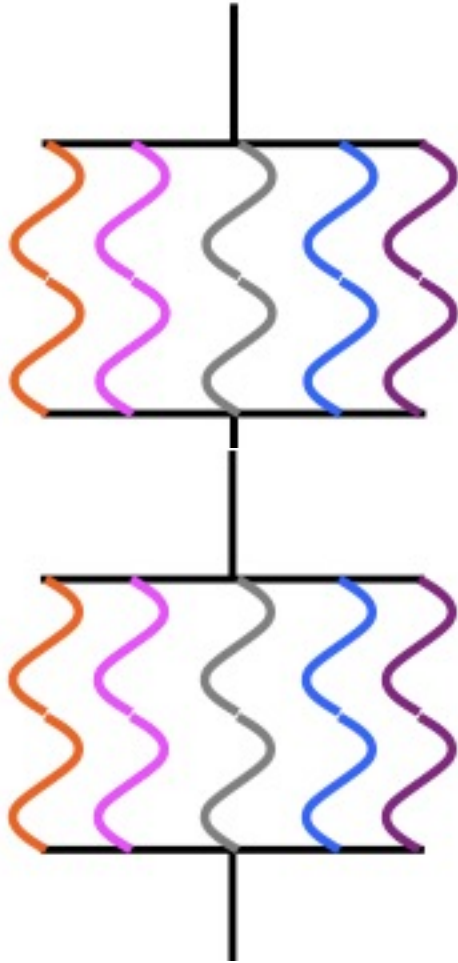
$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

- 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



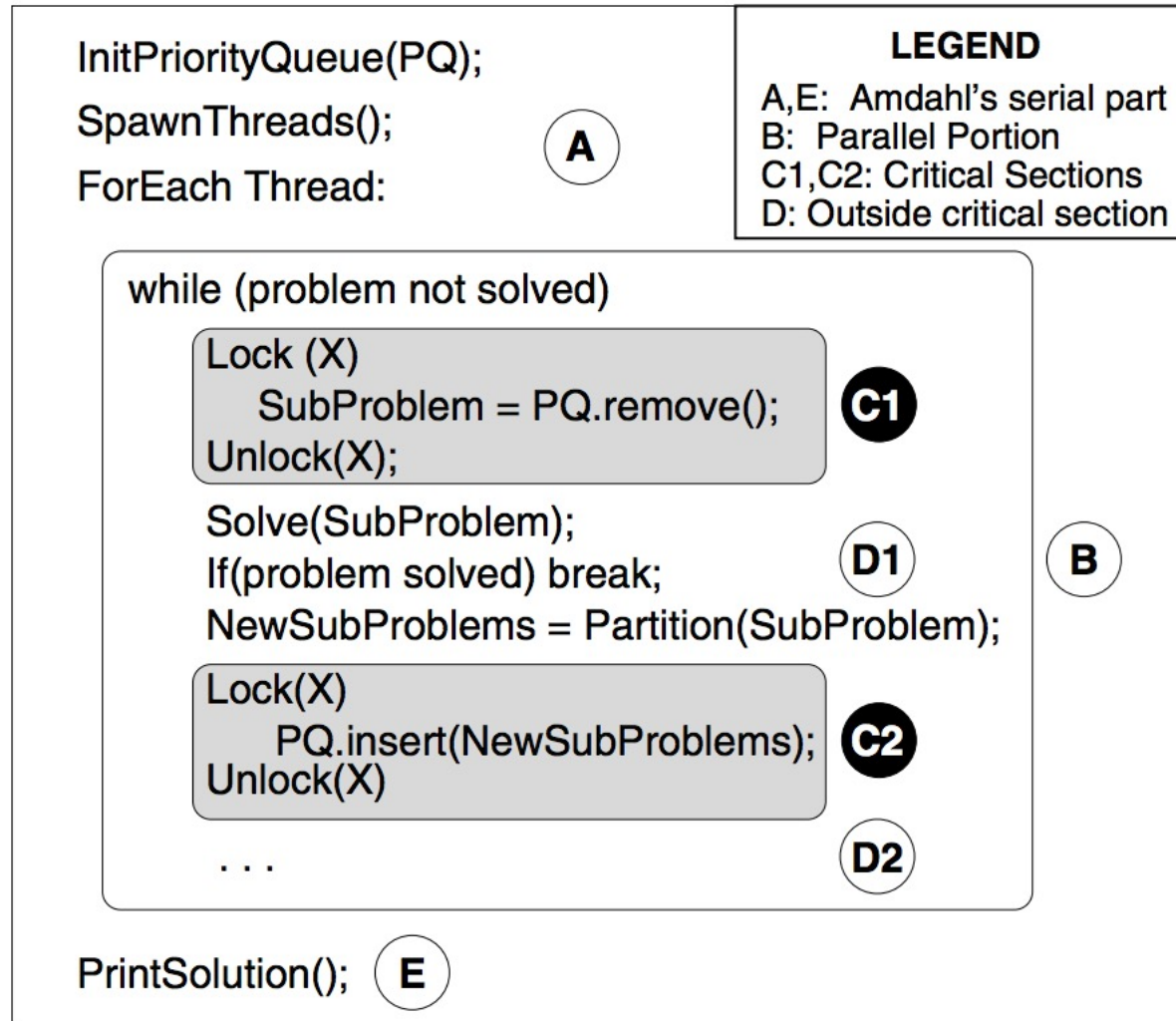
Why the Sequential Bottleneck?



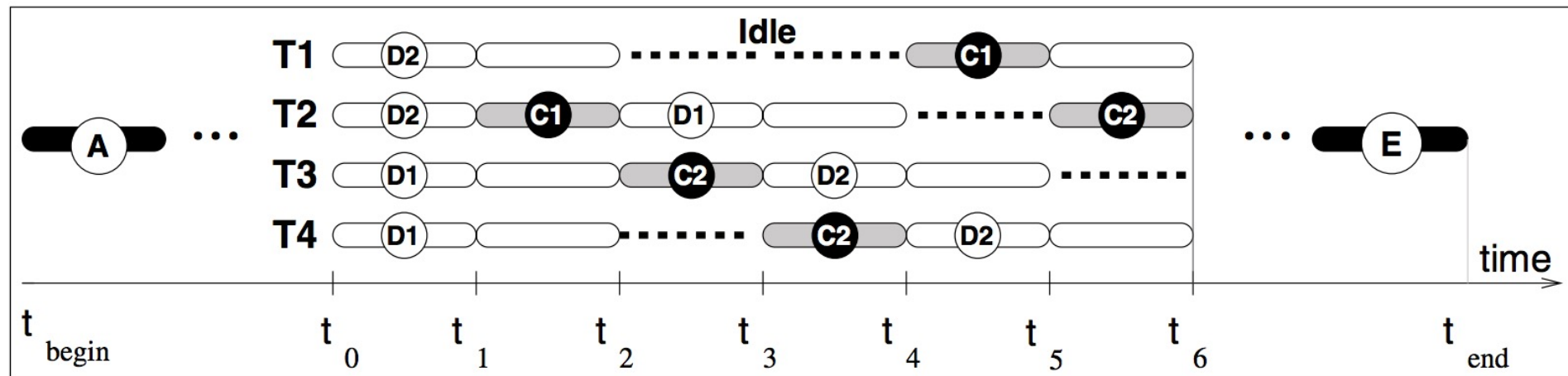
- Parallel machines have the sequential bottleneck
- Main cause: **Non-parallelizable operations on data** (e.g. non-parallelizable loops)

```
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 (I)



Another Example of Sequential Bottleneck (II)



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

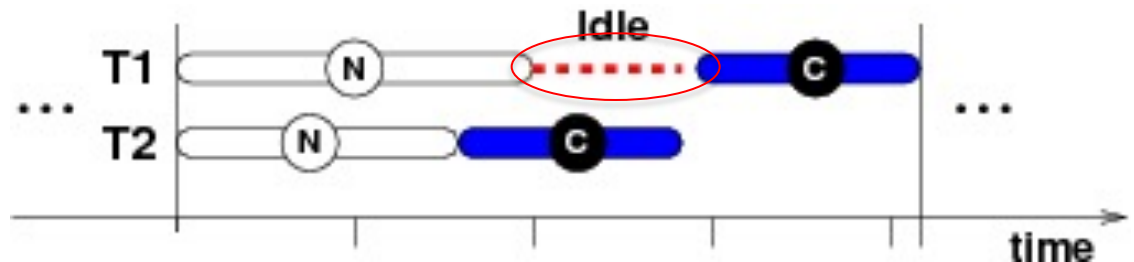
```
  lock(A)
```

```
    Update shared data
```

```
  unlock(A)
```

C

```
}
```

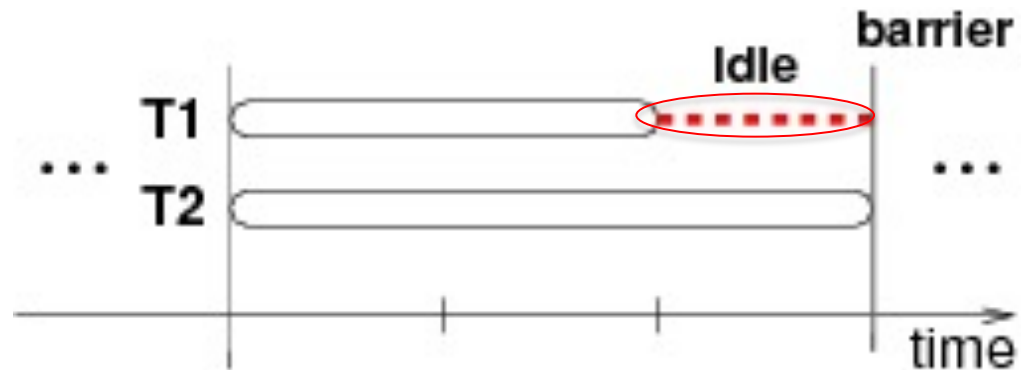


Remember: Barriers

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

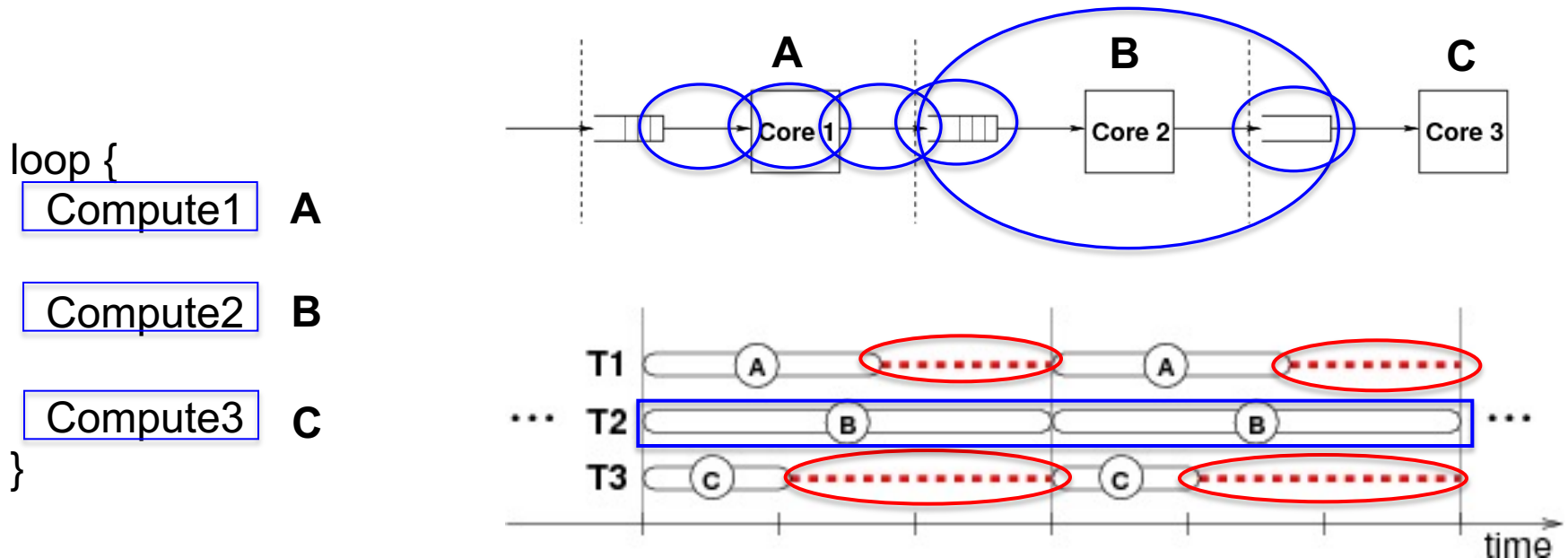
Each thread:

```
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

We Have Already Seen Examples

In Previous Two Lectures

- Lecture 17b: Parallelism and Heterogeneity
 - https://www.youtube.com/watch?v=GLzG_rEDn9A&list=PL5Q2soXY2Zi-Mnk1PxjEIG32HAGILkTOF&index=18
- Lecture 18a: Bottleneck Acceleration
 - <https://www.youtube.com/watch?v=P8l3SMAbyYw&list=PL5Q2soXY2Zi-Mnk1PxjEIG32HAGILkTOF&index=19>

More on Accelerated Critical Sections

- M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt, **"Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures"**

Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 253-264, Washington, DC, March 2009. Slides (ppt)

One of the 13 computer architecture papers of 2009 selected as Top Picks by IEEE Micro.

Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures

M. Aater Suleman

University of Texas at Austin
suleman@hps.utexas.edu

Onur Mutlu

Carnegie Mellon University
onur@cmu.edu

Moinuddin K. Qureshi

IBM Research
mkquresh@us.ibm.com

Yale N. Patt

University of Texas at Austin
patt@ece.utexas.edu

More on Bottleneck Identification & Scheduling

- Jose A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt,
"Bottleneck Identification and Scheduling in Multithreaded Applications"

Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), London, UK, March 2012. Slides (ppt) (pdf)

Bottleneck Identification and Scheduling in Multithreaded Applications

José A. Joao

ECE Department

The University of Texas at Austin
joao@ece.utexas.edu

M. Aater Suleman

Calxeda Inc.

aater.suleman@calxeda.com

Onur Mutlu

Computer Architecture Lab.

Carnegie Mellon University
onur@cmu.edu

Yale N. Patt

ECE Department

The University of Texas at Austin
patt@ece.utexas.edu

More on Utility-Based Acceleration

- Jose A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt, **"Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs"**

Proceedings of the 40th International Symposium on Computer Architecture (ISCA), Tel-Aviv, Israel, June 2013. [Slides \(ppt\)](#)
[Slides \(pdf\)](#)

Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs

José A. Joao [†] M. Aater Suleman ^{‡†} Onur Mutlu [§] Yale N. Patt [†]

[†] ECE Department
The University of Texas at Austin
Austin, TX, USA
{joao, patt}@ece.utexas.edu

[‡] Flux7 Consulting
Austin, TX, USA
suleman@hps.utexas.edu

[§] Computer Architecture Laboratory
Carnegie Mellon University
Pittsburgh, PA, USA
onur@cmu.edu

More on Data Marshaling

- M. Aater Suleman, Onur Mutlu, Jose A. Joao, Khubaib, and Yale N. Patt, **"Data Marshaling for Multi-core Architectures"**
Proceedings of the 37th International Symposium on Computer Architecture (ISCA), pages 441-450, Saint-Malo, France, June 2010. Slides (ppt)
One of the 11 computer architecture papers of 2010 selected as Top Picks by IEEE Micro.

Data Marshaling for Multi-core Architectures

M. Aater Suleman[†] Onur Mutlu[§] José A. Joao[†] Khubaib[†] Yale N. Patt[†]

[†]The University of Texas at Austin
{suleman, joao, khubaib, patt}@hps.utexas.edu

[§]Carnegie Mellon University
onur@cmu.edu

Computer Architecture

Lecture 19a: Multiprocessors

Prof. Onur Mutlu

ETH Zürich

Fall 2021

2 December 2021

An Example Parallel Problem: Task Assignment to Processors

Static versus Dynamic Scheduling

- Static: Done at compile time or parallel task creation time
 - Schedule does not change based on runtime information
- Dynamic: Done at run time (e.g., after tasks are created)
 - Schedule changes based on runtime information
- Example: Instruction scheduling
 - Why would you like to do dynamic scheduling?
 - What pieces of information are not available to the static scheduler?

Parallel Task Assignment: Tradeoffs

- Problem: N tasks, P processors, $N > P$. Do we assign tasks to processors statically (fixed) or dynamically (adaptive)?
- Static assignment
 - + Simpler: No movement of tasks.
 - Inefficient: Underutilizes resources when load is not balanced
When can load not be balanced?
- Dynamic assignment
 - + Efficient: Better utilizes processors when load is not balanced
 - More complex: Need to move tasks to balance processor load
 - Higher overhead: Task movement takes time, can disrupt locality

Parallel Task Assignment: Example

- Compute histogram of a large set of values
- Parallelization:
 - Divide the values across T tasks
 - Each task computes a local histogram for its value set
 - Local histograms merged with global histograms in the end

```
GetPageHistogram(Page *P)
```

```
  For each thread: {
```

```
    /* Parallel part of the function */  
    UpdateLocalHistogram(Fraction of Page)
```

```
    /* Serial part of the function */  
    Critical Section:  
    Add local histogram to global histogram
```

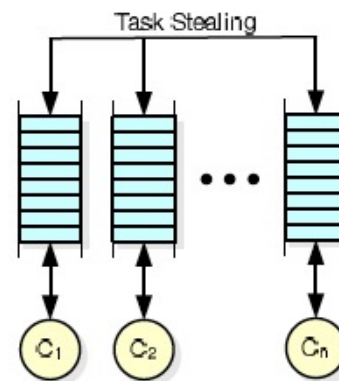
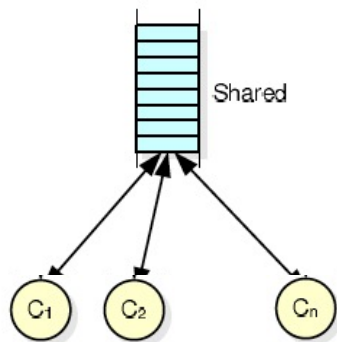
```
  Barrier
```

```
}
```

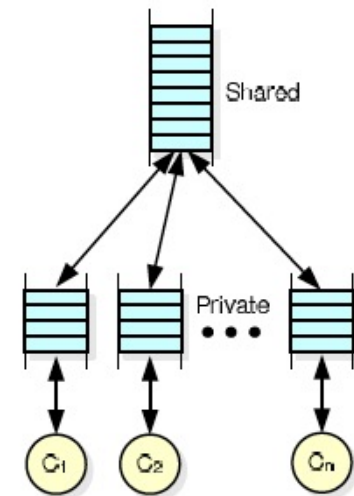
```
  Return global histogram
```

Parallel Task Assignment: Example (II)

- How to schedule tasks updating local histograms?
 - ❑ Static: Assign equal number of tasks to each processor
 - ❑ Dynamic: Assign tasks to a processor that is available
 - ❑ When does static work as well as dynamic?
- Implementation of Dynamic Assignment with Task Queues



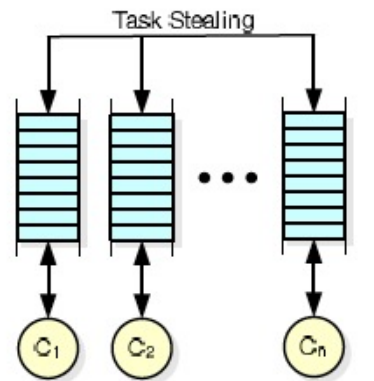
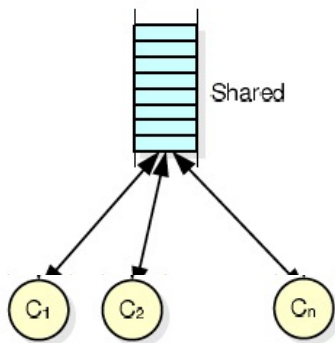
(a) Distributed Task Stealing



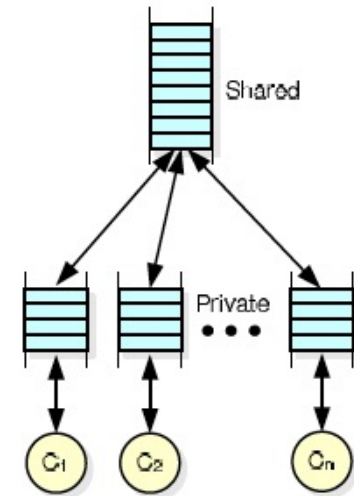
(b) Hierarchical Task Queuing

Software Task Queues

- What are the advantages and disadvantages of each?
 - ❑ Centralized
 - ❑ Distributed
 - ❑ Hierarchical



(a) Distributed Task Stealing



(b) Hierarchical Task Queuing

Task Stealing

- **Idea:** When a processor's task queue is empty it steals a task from another processor's task queue
 - Whom to steal from? (Randomized stealing works well)
 - How many tasks to steal?
- + Dynamic balancing of computation load
- Additional communication/synchronization overhead between processors
- Need to stop stealing if no tasks to steal

Parallel Task Assignment: Tradeoffs

- Who does the assignment? Hardware versus software?
- Software
 - + Better scope
 - More time overhead
 - Slow to adapt to dynamic events (e.g., a processor becoming idle)
- Hardware
 - + Low time overhead
 - + Can adjust to dynamic events faster
 - Requires hardware changes (area and possibly energy overhead)

How Can the Hardware Help?

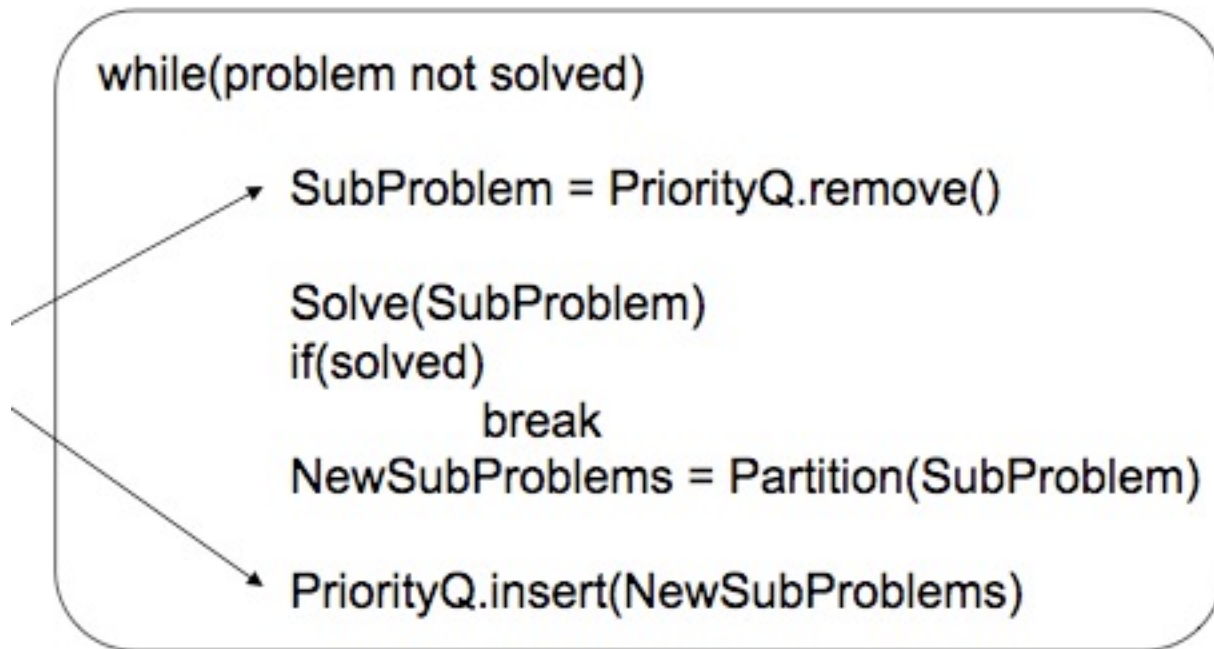
- Managing task queues in software has overhead
 - Especially high when task sizes are small

- An idea: Hardware Task Queues
 - Each processor has a dedicated task queue
 - Software fills the task queues (on demand)
 - Hardware manages movement of tasks from queue to queue
 - There can be a global task queue as well → hierarchical tasking in hardware

- Kumar et al., “Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors,” ISCA 2007.
 - Optional reading

Dynamic Task Generation

- Does static task assignment work in this case?
- Problem: Searching the exit of a maze



Programming Model vs. Hardware Execution Model

Programming Models vs. Architectures

- Five major models
 - (Sequential)
 - Shared memory
 - Message passing
 - Data parallel (SIMD)
 - Dataflow
 - Systolic
- Hybrid models?

Shared Memory vs. Message Passing

- Are these programming models or execution models supported by the hardware architecture?
- Does a multiprocessor that is programmed by “shared memory programming model” have to support a shared address space processors?
- Does a multiprocessor that is programmed by “message passing programming model” have to have no shared address space between processors?

Programming Models: Message Passing vs. Shared Memory

- Difference: how communication is achieved between tasks
- Message passing programming model
 - Explicit communication via messages
 - Loose coupling of program components
 - Analogy: telephone call or letter, no shared location accessible to all
- Shared memory programming model
 - Implicit communication via memory operations (load/store)
 - Tight coupling of program components
 - Analogy: bulletin board, post information at a shared space
- Suitability of the programming model depends on the problem to be solved. Issues affected by the model include:
 - Overhead, scalability, ease of programming, bugs, match to underlying hardware, ...

Message Passing vs. Shared Memory Hardware

- Difference: how task communication is supported in hardware
- Shared memory hardware (or machine model)
 - All processors see a global shared address space
 - Ability to access all memory from each processor
 - A write to a location is visible to the reads of other processors
- Message passing hardware (machine model)
 - No global shared address space
 - Send and receive variants are the only method of communication between processors (much like networks of workstations today, i.e. clusters)
- Suitability of the hardware depends on the problem to be solved as well as the programming model.

Programming Model vs. Hardware

- Most of parallel computing history, there was no separation between programming model and hardware
 - Message passing: Caltech Cosmic Cube, Intel Hypercube, Intel Paragon
 - Shared memory: CMU C.mmp, Sequent Balance, SGI Origin.
 - SIMD: ILLIAC IV, CM-1
- However, any hardware can really support any programming model
- Why?
 - Application → compiler/library → OS services → hardware