18-447: Computer Architecture
Lecture 30B: Multiprocessors

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2013, 4/22/2013

Readings: Multiprocessing

- Required
  - Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.
  - Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, 1979

- 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.

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 4th and 4th revised eds.)

- Recommended:
  - Papamarcos and Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," ISCA 1984.

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 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 Issues in Tightly-Coupled MP

- Shared memory synchronization
  - Locks, atomic operations
- Cache consistency
  - More commonly called cache coherence
- Ordering of memory operations
  - What should the programmer expect the hardware to provide?
- Resource sharing, contention, partitioning
- Communication: Interconnection networks
- Load imbalance

Aside: Hardware-based Multithreading

- Coarse grained
  - Quantum based
  - Event based (switch-on-event multithreading)

- 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

Parallel Speedup Example

- $a4x^4 + a3x^3 + a2x^2 + a1x + a0$

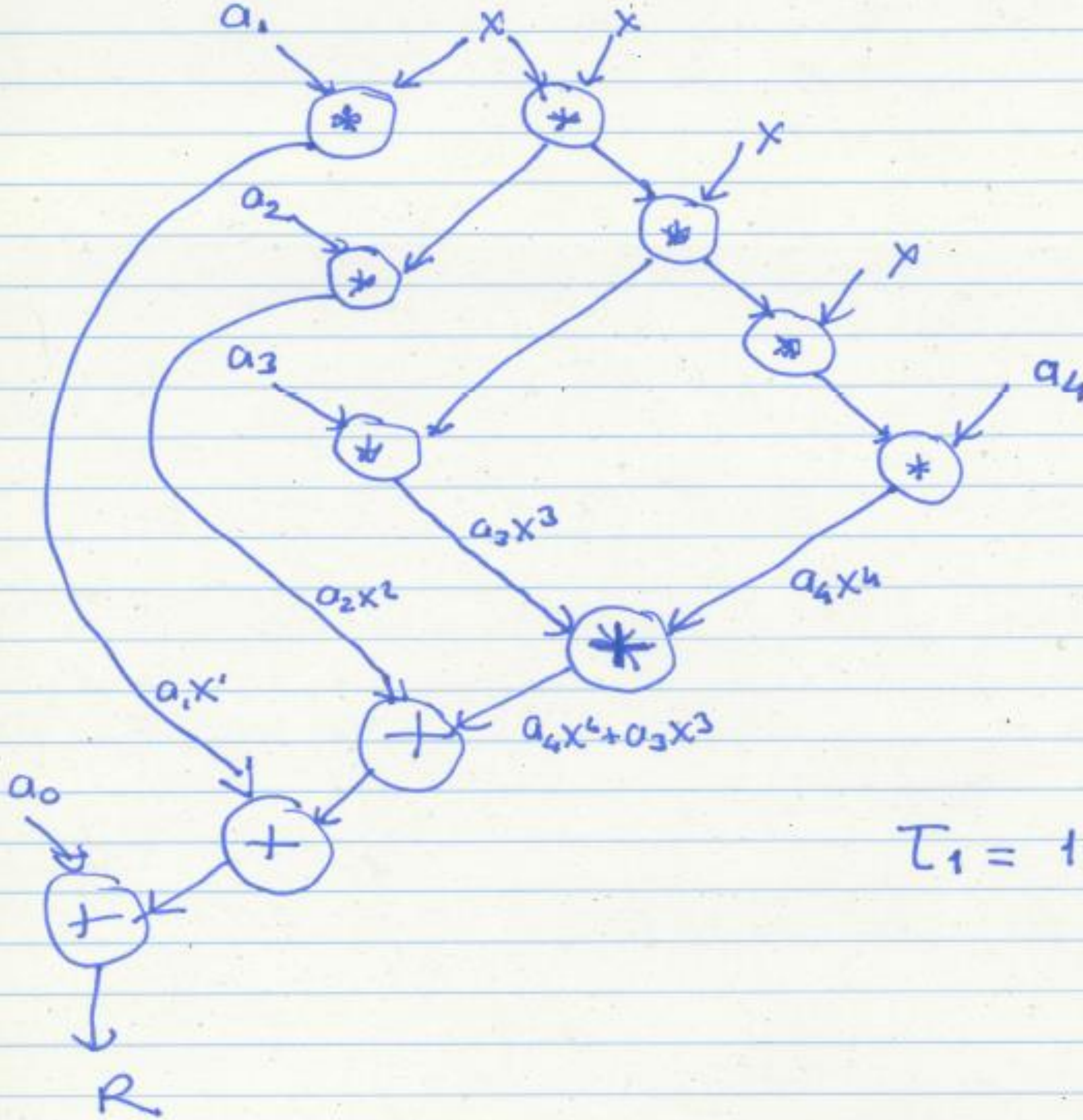
- Assume each operation 1 cycle, no communication cost, each op can be executed in a different processor

- How fast is this with a single processor?
  - Assume no pipelining or concurrent execution of instructions

- How fast is this with 3 processors?

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

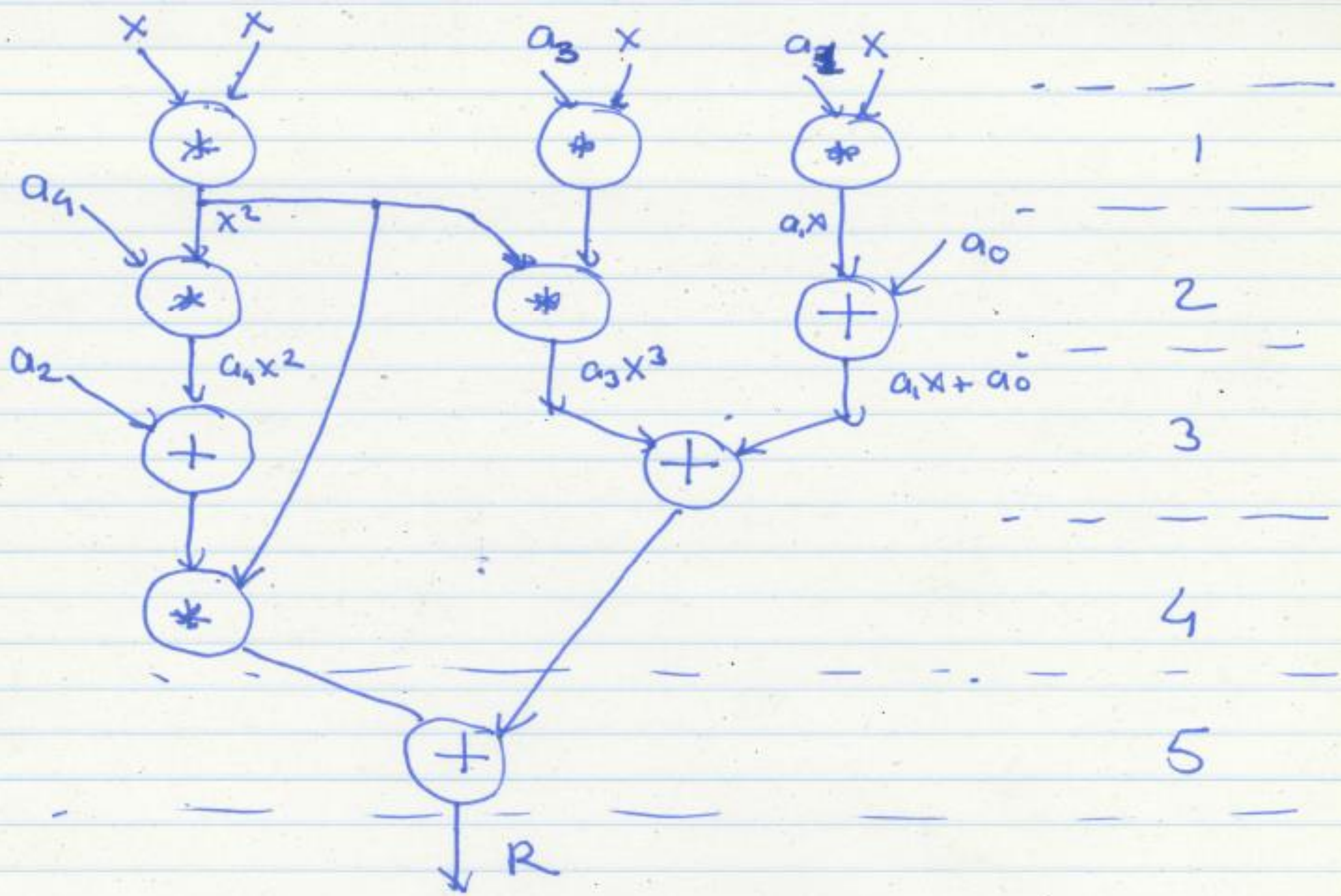
Single processor: 11 operations (DRAW the data flow graph)



$T_1 = 11$ cycles

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

Three processors: τ_3 (exec. time with 3 proc.)



$\tau_3 = \underline{5 \text{ cycles}}$

Speedup with 3 Processors

$$\tau_3 = \underline{5 \text{ cycles}}$$

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

$$\left(\frac{\tau_1}{\tau_3}\right)$$

<u>Is this a fair comparison?</u>

Revisiting the Single-Processor Algorithm

Revisit τ_1

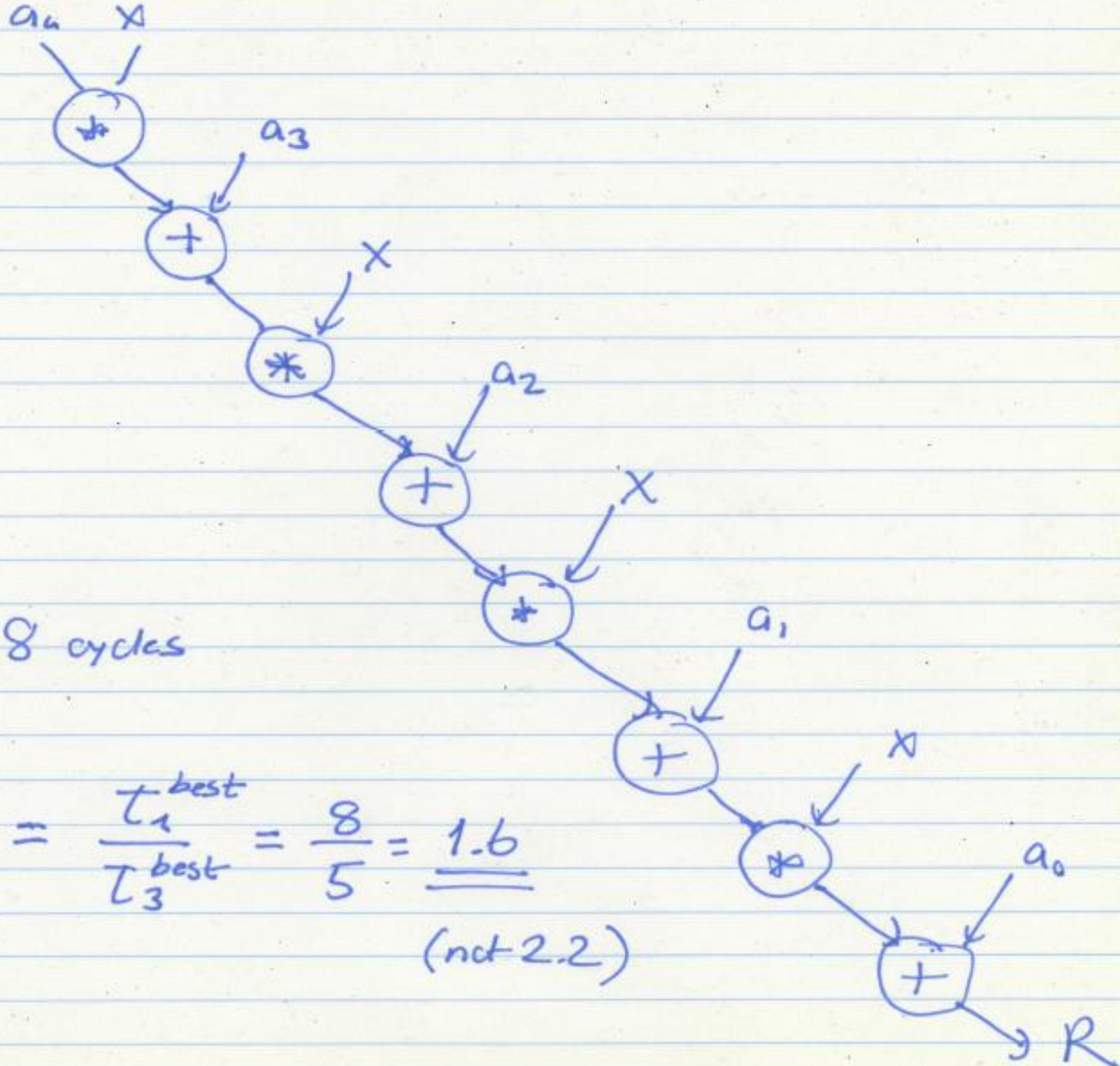
Better single-processor algorithm:

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

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

(Horner's method)

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



$\tau_1 = 8$ cycles

Speedup with 3 procs. $= \dfrac{\tau_1^{best}}{\tau_3^{best}} = \dfrac{8}{5} = \underline{\underline{1.6}}$

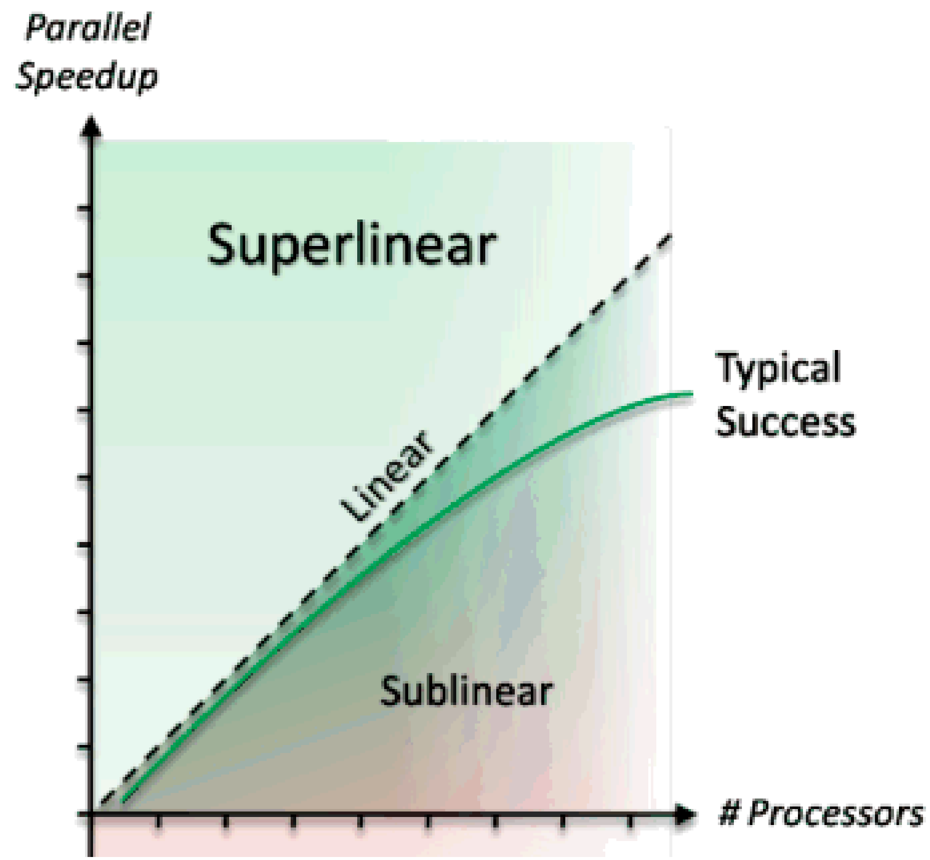
(not 2.2)

Superlinear Speedup

- Can speedup be greater than P with P processing elements?

- Cache effects
- Working set effects

- Happens in two ways:
  - Unfair comparisons
  - Memory effects



Utilization, Redundancy, Efficiency

- Traditional metrics
  - Assume all P processors are tied up for parallel computation

- Utilization: How much processing capability is used
  - U = (# Operations in parallel version) / (processors x Time)

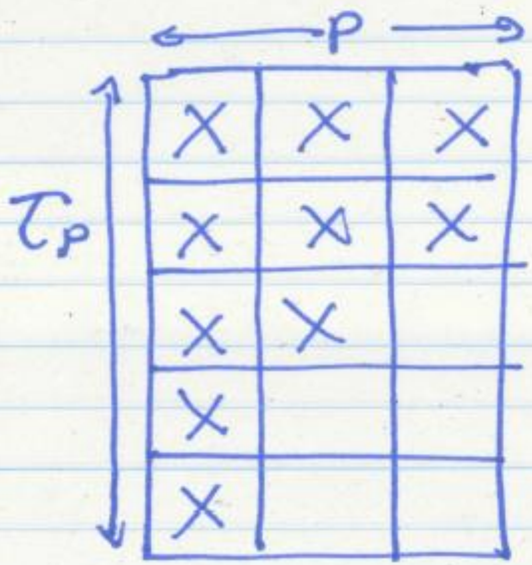
- Redundancy: how much extra work is done with parallel processing
  - R = (# of operations in parallel version) / (# operations in best single processor algorithm version)

- Efficiency
  - E = (Time with 1 processor) / (processors x Time with P processors)
  - E = U/R

Utilization of a Multiprocessor

Multiprocessor metrics

Utilization: How much processing capability we use



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

$$= \frac{10}{15}$$

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

Redundancy: How much extra work due to multiprocessing

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

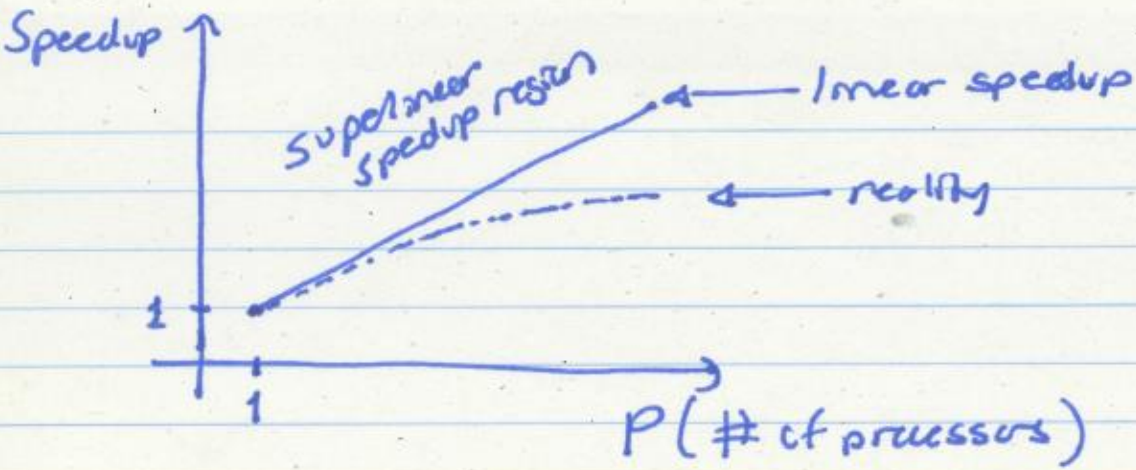
R is always ≥ 1

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

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

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

Caveats of Parallelism (I)



<u>Why the reality? (diminishing returns)</u>

$$\tau_p = \underbrace{\alpha \cdot \frac{\tau_1}{p}}_{\text{parallelizable part/fraction of the single-processor program}} + \underbrace{(1-\alpha)\cdot \tau_1}_{\text{non-parallelizable part}}$$

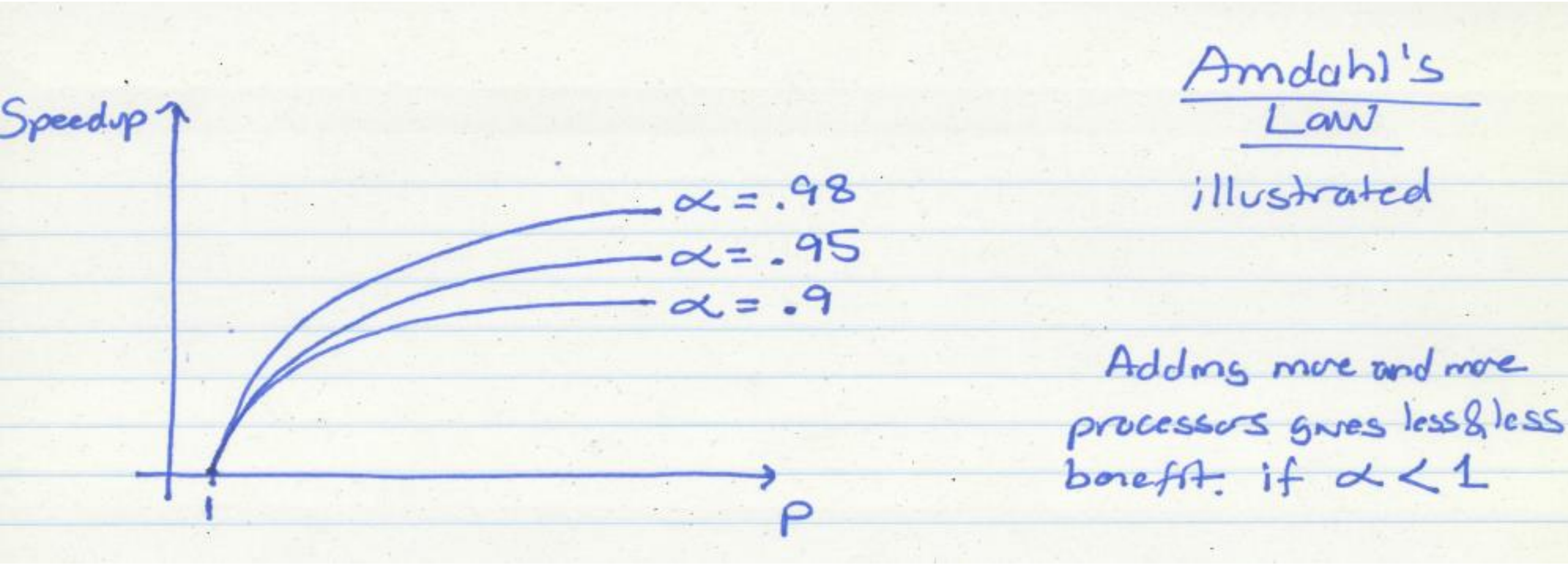
Amdahl's Law

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

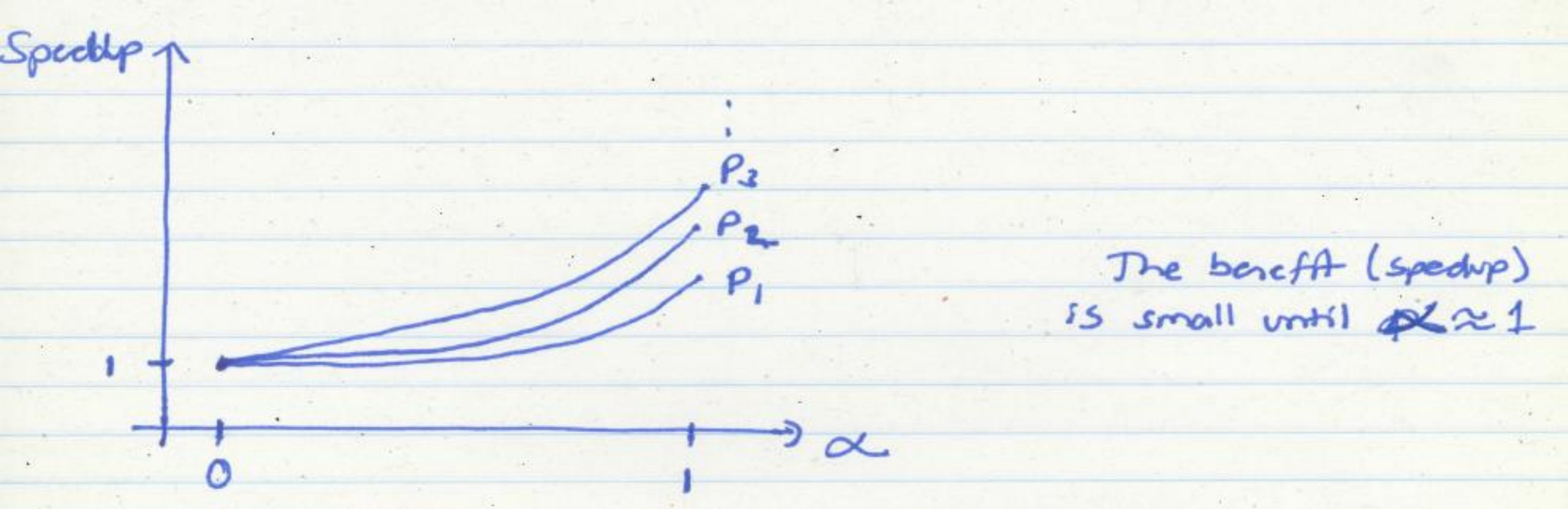
Speedup as $p \to \infty$ $= \dfrac{1}{1-\alpha}$ → bottleneck for parallel speedup

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

Amdahl's Law Implication 1



Amdahl's Law Implication 2



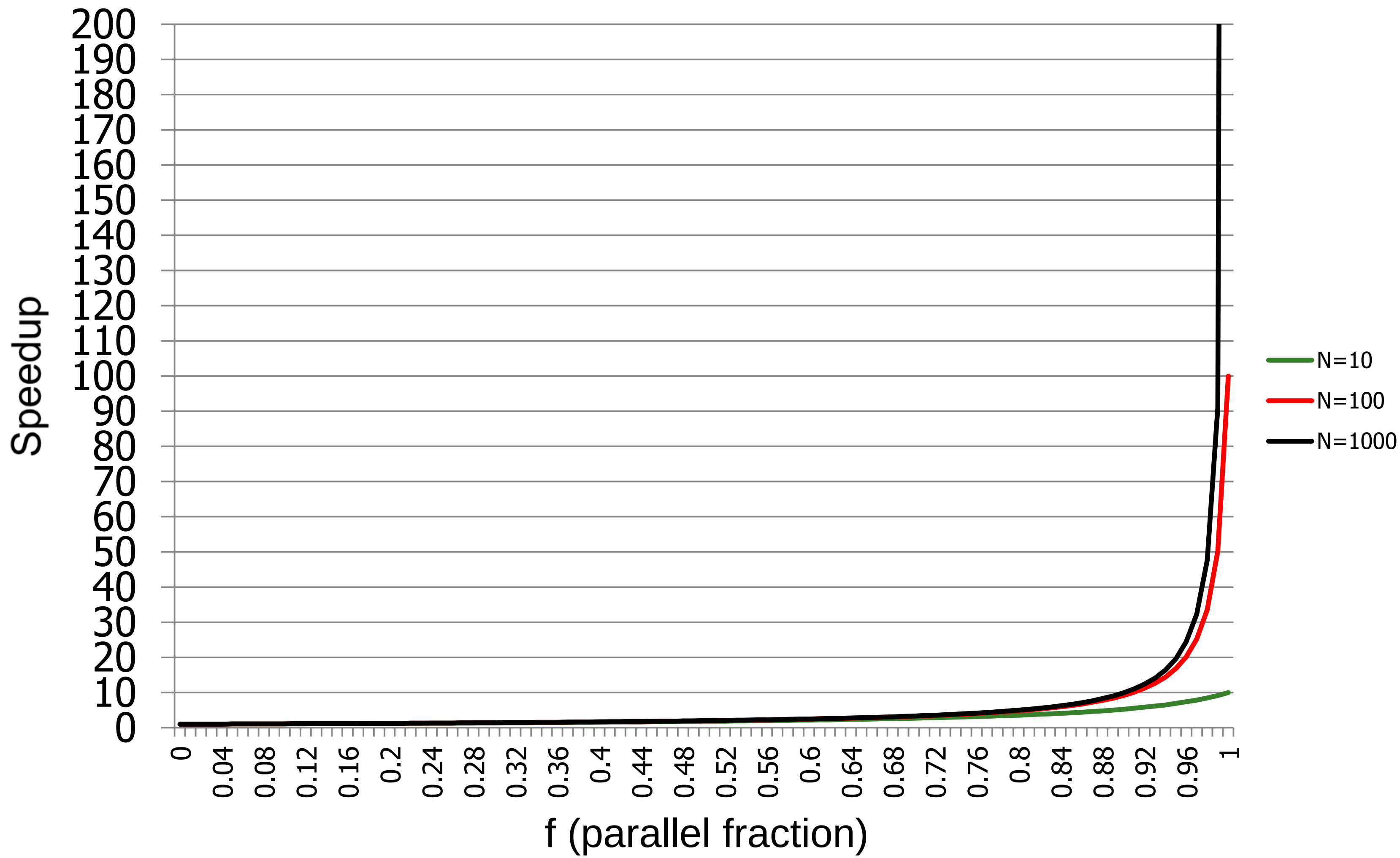
Caveats of Parallelism (II)

- Amdahl's Law
  - f: Parallelizable fraction of a program
  - N: Number of processors

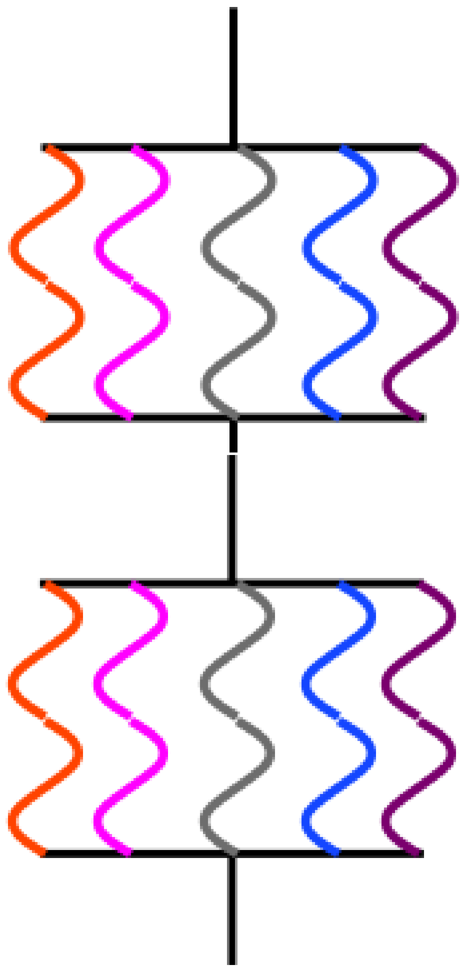
$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

  - 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
  ```

- Single thread prepares data and spawns parallel tasks (usually sequential)

Another Example of Sequential Bottleneck

LEGEND

A,E: Amdahl's serial part
B: Parallel Portion
C1,C2: Critical Sections
D: Outside critical section

```
InitPriorityQueue(PQ);
SpawnThreads();                                   A

ForEach Thread:

  while (problem not solved)
      Lock (X)
          SubProblem = PQ.remove();               C1
      Unlock(X);
      Solve(SubProblem);                          D1        B
      If(problem solved) break;
      NewSubProblems = Partition(SubProblem);
      Lock(X)
          PQ.insert(NewSubProblems
      Unlock(X)
      . . .                                       D2

PrintSolution();                                  E
```

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