



CSE335: Cloud Computing Concurrency

본 강의 영상의 저작권은 교수자에게 있으므로 무단배포 또는 판매하는 행위를 금합니다.

Jaehong Kim
(jaehong.kim@khu.ac.kr)

Guide for spec & mid project

- In 7 weeks, you need to make a spec & mid presentation based on the team that was created before. Also you need to submit relevant reports
 - 4.13(수) 자정 까지 -> output 제출 (ecampus 과제함 제출)
 - 1. [proj1] ppt, report, mp4 2.[proj1] 평가표
- Outputs : 1.[팀별] Presentations, 2. [팀별] Reports, 3. [팀별] 15분 이내 발표 동영상 (꼭, mp4 파일로 변환(변환 X 성적 페널티) - ecampus에서 지원하는 포맷) 4. 팀원상호평가표는 (spec, mid에서는 제출 X, final 에서만 제출), 발표동영상은 4.13(수) 취합된 것을 4.14(목) 오전 중에 7주차 강의컨텐츠에 조별 mp4로 변환된 파일을 업로드 예정. (강의 컨텐츠는 4.14(목) ~ 4.15(금) 낮12시 까지만 오픈) 해당 mp4는 실제 시청 여부가 ecampus에 출석으로 남음 - 이를 기반으로 출석체크. 시청하고 각 조별로 평가 진행 (본인 조 mp4는 안봐도 무방) - 7주차 수업은 각조별 발표 동영상 시청으로 대체
- Presentation Time Guidelines
 - 15 mins for presents (15분 이내 발표 동영상 제출)
 - (각 팀별로 발표 동영상 제출 - 단, 팀별 모임은 온라인(카톡등) 으로 비대면 진행 바람 (오프라인 미팅 금지))
- The report and presentation outline will be posted on ecampus.
- Evaluation is done through mutual evaluation. The evaluation sheet is distributed on the day of presentation.
 - Evaluate by team excluding your team
 - Total 9 teams (A, B, C, D, E, F, G, H, I)
 - Each team has 8 team evaluations = $9 \times 8 = 72$
 - 각팀은 4.15(금) 낮12시까지 올린 발표 동영상을 보시고, 채점하고 조장이 취합(조원 평균) 하여 채점표 저에게 제출 (4.15(금) 낮12시까지 제출) - mp4 규정을 못지키거나, 4.13(수)까지 발표동영상을 저에게 안보내주시면, 강의 컨텐츠에 해당팀 것은 안올라갑니다. 그에 따른 페널티가 있습니다.)
 - 팀원상호평가표는 final 제출 때 개인별 제출. (spec, mid 때는 제출X, 즉 spec, mid final 과제 하면서 겪은 팀원에 대한 상호평가)



CSE335: Cloud Computing SpecMid Outline

- Presentation Date :
- Team Name :

Contents

- Overview
- Goal/Problem & Requirement
- Approach
- Development Environment
- Architecture
- Implementation Spec
- Current Status
- Further Plan
- Demo Plan
- Division and Assignment of work
- Schedule

Overview

Goal/Problem & Requirement

- 본 프로젝트의 목표 및 요구사항

Approach

- 프로젝트 수행 목표 또는 문제 정의 및 접근 방법

Development Environment

- 개발 환경 정리
 - 사용 언어
 - 사용 플랫폼
 - 사용 IDE 등

Architecture

- Architecture Diagram
- Architecture Description
 - Infra Architecture
 - Software Architecture

Implementation Spec

- API spec
 - Input / Output Interface
 - Inter Module Communication Interface
 - Modules (API)

Current Status

Further Plan

- Includes Demo Plan

Demo Plan

- Demo plan to demonstrate in Final Announcement
- With Specific scenarios
- Should also be in the report

Division and Assignment of Work (very important)

Item	Assignee

Schedule (very important)

Contents	3월			4월				5월				6월	
	2w	3w	4w	1w	2w	3w	4w	1w	2w	3w	4w	1w	2w

Thanks

- Thanks
- Presenter Name (email address)

Where are we?

- Basics: Scalability, Concurrency, Consistency...
- Cloud Basics: EC2, EBS, S3, SimpleDB, ...
- Cloud Practice (Computing, networking, etc)
- Cloud NoSQL
- Cloud Programming (MapReduce, Spark, etc)
- Serverless Computing
- Cloud IoT

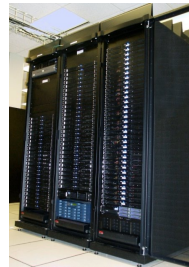
Scale increases complexity



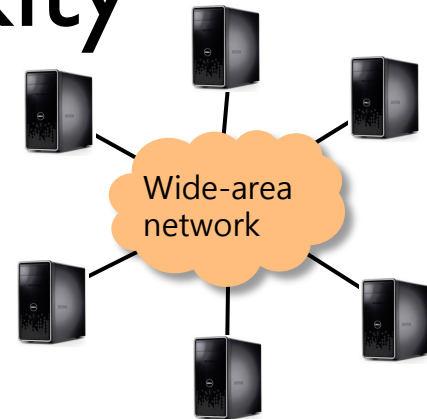
Single-core machine



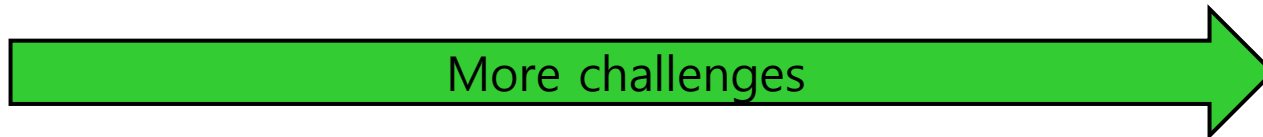
Multicore server



Cluster



Large-scale distributed system



Undergraduate
1~3



True
concurrency

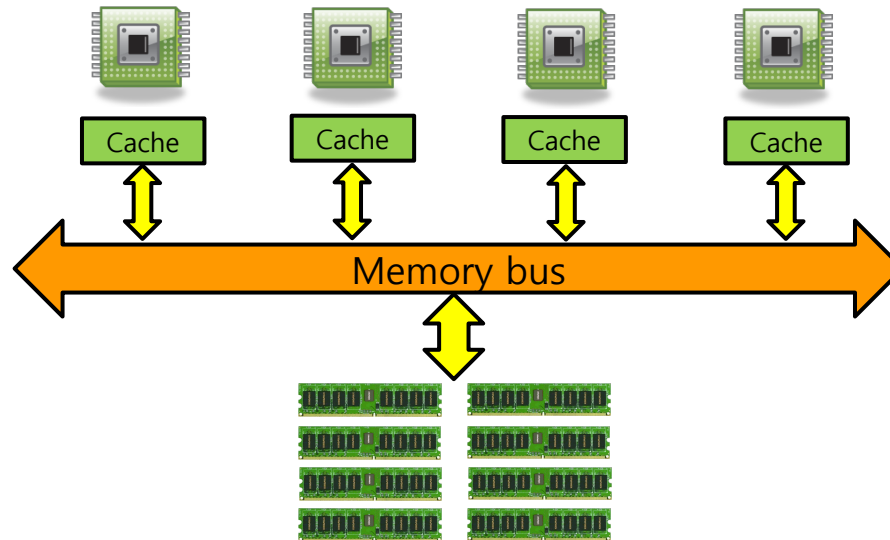
Network
Message passing
More failure nodes
(faulty nodes, ...)

Wide-area network
Even more failure
modes

Fear not!

- You will hear a lot of challenges
 - Packet loss, faulty machine, network partition, inconsistency, variable memory latencies, deadlocks...
- But there are frameworks that will help you
 - This course is **NOT** about low-level programming
- Nevertheless, it is important to know about these challenges
 - Why?

Symmetric Multiprocessing (SMP)



- For now, assume we have multiple cores that can access the same shared memory
 - Any core can access any byte; speed is uniform (no byte takes longer to read or write than any other)
 - Not all machines are like that -- other models discussed later

Plan for the next lectures

- **Parallel programming and its challenges**
 - Parallelization and scalability, Amdahl's law
 - Synchronization, consistency
 - Mutual exclusion, locking, issues related to locking
 - Architectures: SMP, NUMA, Shared-nothing
- **All about the Internet in 30 minutes**
 - Structure; packet switching; some important protocols
 - Latency, packet loss, bottlenecks, and why they matter
- **Distributed programming and its challenges**
 - Network partitions and the CAP theorem
 - Faults, failures, and what we can do about them



What is scalability?

- A system is **scalable** if it can easily adapt to increased (or reduced) demand
 - Example: A storage system might start with a capacity of just 10TB but can grow to many PB by adding more nodes
 - Scalability is usually limited by some sort of **bottleneck**
- Often, scalability also means...
 - The ability to operate at a very large scale
 - The ability to grow efficiently
 - Example: 4x as many nodes -> ~4x capacity (not just 2x!)

Scalability, efficiency, and speed

- Suppose you need to process 1TB of data
- You have money for up to 100 machines
- You can choose between three systems:
 - System A : Runs on a single machine
 - System B : Throughput grows with $O(N)$, where N is the number of machines, up to $N = 50$
 - System C : Throughput grows with $O(\sqrt{N})$, where N is the number of machines
- Which system should you use?
- Page rank?

How to build systems that scale

```
void bubblesort(int nums[]) {
    boolean done = false;
    while (!done) {
        done = true;
        for (int i=1; i<nums.length; i++) {
            if (nums[i-1] > nums[i]) {
                swap(nums[i-1], nums[i]);
                done = false;
            }
        }
    }
}
```

```
int[] mergesort(int nums[]) {
    int numPieces = 10;
    int pieces[][] = split(nums, numPieces);
    for (int i=0; i<numPieces; i++)
        sort(pieces[i]);
    return merge(pieces);
}
```

Can be done in parallel!

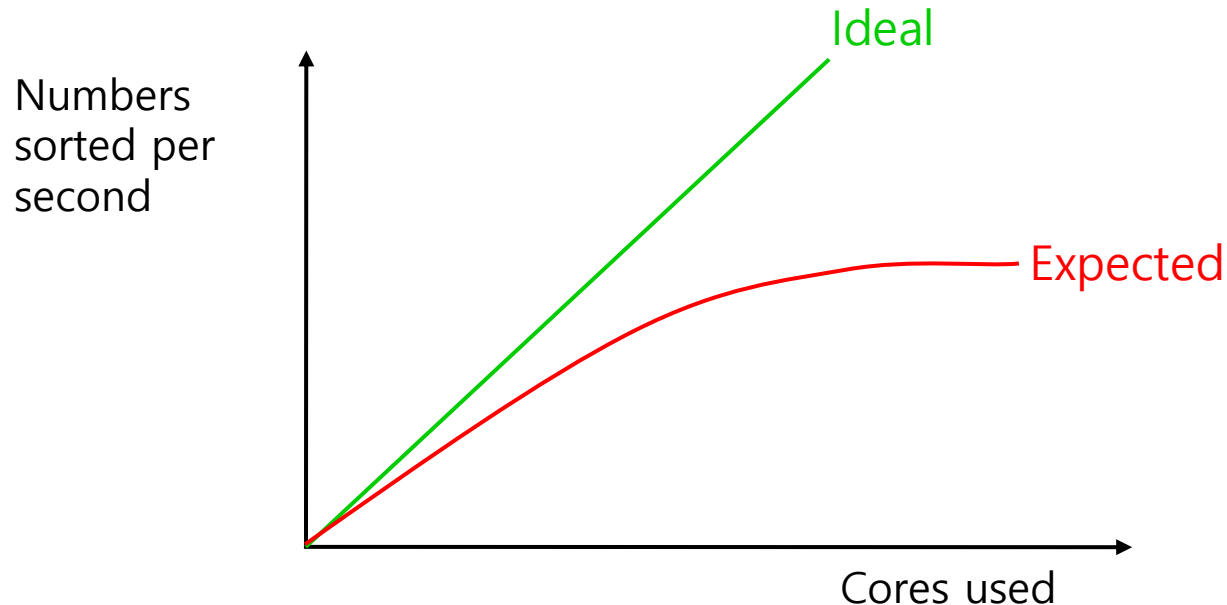
- The left algorithm works fine on one core
- Can we make it faster on multiple cores?
 - Difficult - need to find something for the other cores to do
 - There are other sorting algorithms where this is much easier
 - Not all algorithms are equally parallelizable
 - Can you have scalability without parallelism?

Scalability in practice

Speedup: $S_N = \frac{T_1}{T_n}$

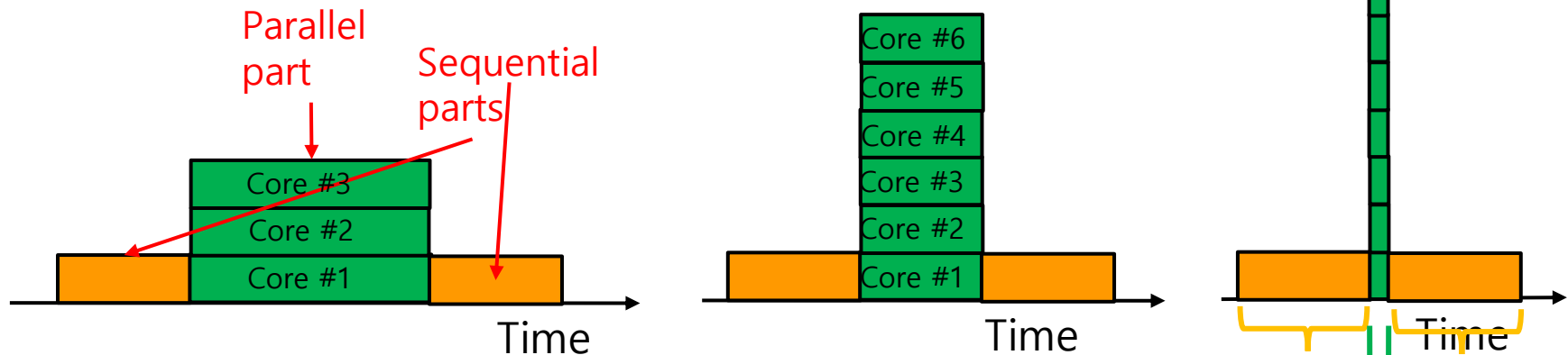
Completion time with one core $\rightarrow T_1$

Completion time with n cores $\rightarrow T_n$



- If we increase the number of processors, will the speed also increase?
 - Yes, but (in almost all cases) only up to a point
 - Why?

Amdahl's law



- Usually, not all parts of the algorithm can be parallelized
- Let f be the fraction of the algorithm that can be parallelized, and let S_{part} be the corresponding speedup
- Then

$$S_{overall} = \frac{1}{(1-f) + \frac{f}{S_{part}}}$$

The diagram shows yellow and green arrows pointing from the terms in the equation to the corresponding parts in the Gantt charts. A yellow arrow points from the $(1-f)$ term to the sequential part (orange bars) in Chart 3. A green arrow points from the $\frac{f}{S_{part}}$ term to the parallel part (green bars) in Chart 3.

Example

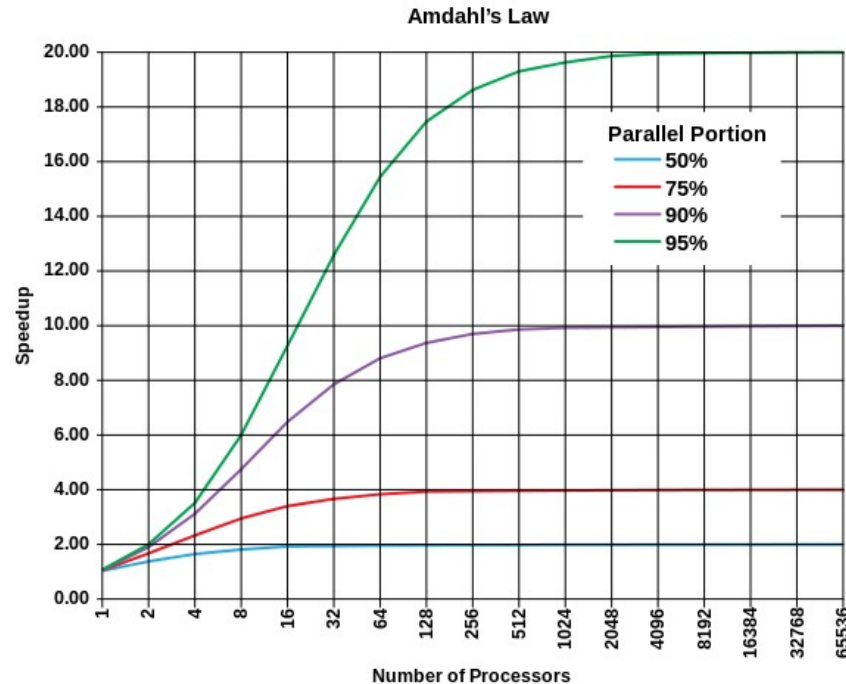
- What is the overall speedups when there is a performance improvement of S_{part} in a system by f ?

$$S_{overall} = \frac{1}{(1-f) + \frac{f}{S_{part}}}$$

- If you can double the speed of 40% of a job, what is the overall speedups?

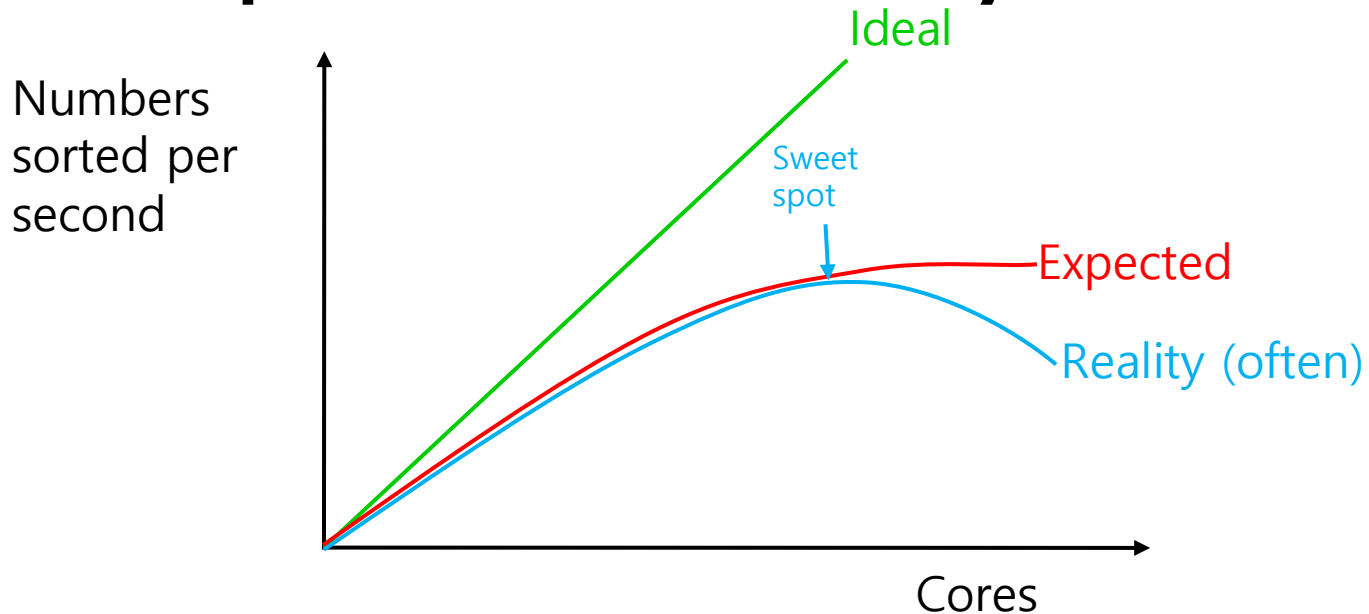
$$\frac{1}{(1-0.4) + \frac{0.4}{2}} = 1.25$$

Amdahl's law



- When using multiprocessors in parallel computing, the performance improvement of the program is limited by the sequential part of the program.
 - For example, even if 95% of the programs are parallelized, the theoretical maximum performance improvement is limited to a maximum of 20 times, no matter how many processors are used.

Is more parallelism always better?

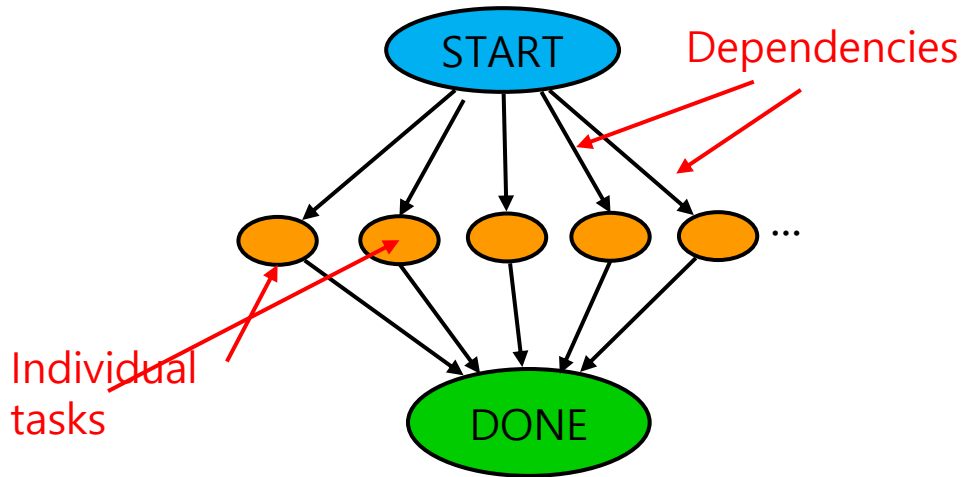


- Increasing parallelism beyond a certain point can cause performance to decrease! Why?
- Time for serial parts can depend on #cores
 - Example: Need to send a message to each core to tell it what to do

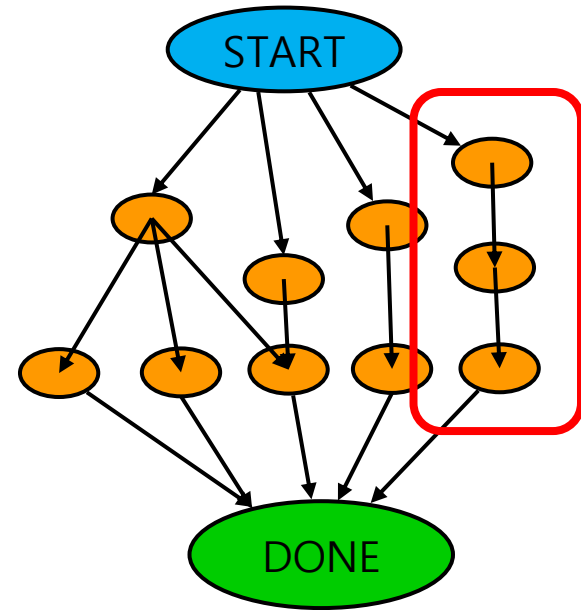
Granularity

- How big a task should we assign to each core?
 - Coarse-grain vs. fine-grain parallelism
- Frequent coordination creates overhead
 - Need to send messages back and forth, wait for other cores...
 - Result: Cores spend most of their time communicating
- Coarse-grain parallelism is usually more efficient
 - **Bad** : Ask each core to sort three numbers
 - **Good** : Ask each core to sort a million numbers

Dependencies



"Embarrassingly parallel"



With dependencies

- What if tasks depend on other tasks?
 - Example: Need to sort lists before merging them
 - Limits the degree of parallelism
 - Minimum completion time (and thus maximum speedup) is determined by the longest path from start to finish
 - Assumes resources are plentiful; actual speedup may be lower

Heterogeneity

- What if...
 - Some tasks are larger than others?
 - Some tasks are harder than others?
 - Some tasks are more urgent than others?
 - Not all cores are equally fast, or have different resources?
- Results: Scheduling problem
 - Can be very difficult

Recap: Parallelization

- Parallelization is hard
 - Not all algorithms are equally parallelizable – need to pick very carefully
- Scalability is limited by many things
 - Amdahl's law
 - Dependencies between tasks
 - Communication overhead
 - Heterogeneity
 - ...

Plan for the next two lectures

- Parallel programming and its challenges
 - Parallelization and scalability, Amdahl's law ✓
 - Synchronization, consistency ← NEXT
 - Mutual exclusion, locking, issues related to locking
 - Architectures: SMP, NUMA, Shared-nothing
- All about the Internet in 30 minutes
 - Structure; packet switching; some important protocols
 - Latency, packet loss, bottlenecks, and why they matter
- Distributed programming and its challenges
 - Network partitions and the CAP theorem
 - Faults, failures, and what we can do about them

Why do we need synchronization?



```
void transferMoney(customer A, customer B, int amount)
{
    showMessage("Transferring "+amount+" to "+B);
    int balanceA = getBalance(A);
    int balanceB = getBalance(B);
    setBalance(B, balanceB + amount);
    setBalance(A, balanceA - amount);
    showMessage("Your new balance: "+(balanceA-amount));
}
```

- Simple example: Accounting system in a bank
 - Maintains the current balance of each customer's account
 - Customers can transfer money to other customers

Why do we need synchronization?

Starting value :

Alice : \$200

Bob : \$800



Alice

\$100



\$500



Bob

Expected value :

Alice : $200 + 400 = 600$

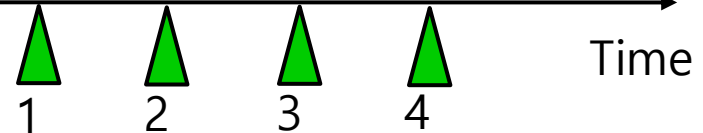
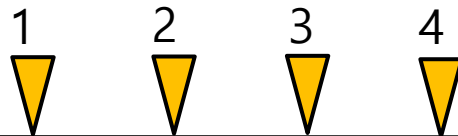
Bob : $800 - 400 = 400$

- 1) `B=Balance (Bob)`
- 2) `A=Balance (Alice)`
- 3) `SetBalance (Bob, B+100)`
- 4) `SetBalance (Alice, A-100)`

- 1) `A=Balance (Alice)`
- 2) `B=Balance (Bob)`
- 3) `SetBalance (Alice, A+500)`
- 4) `SetBalance (Bob, B-500)`

Assumption : A, B, Alice, and Bob are all global variables.

- What can happen if this code runs concurrently?



Alice's balance:	\$200	\$200	\$200	\$100
Bob's balance:	\$800	\$800	\$900	\$900

\$100	\$100	\$600	\$600
\$900	\$900	\$900	\$400



Why do we need synchronization?

Starting value :

Alice : \$200

Bob : \$800



Alice

\$100



\$500



Bob

Expected value :

Alice : $200 + 400 = 600$

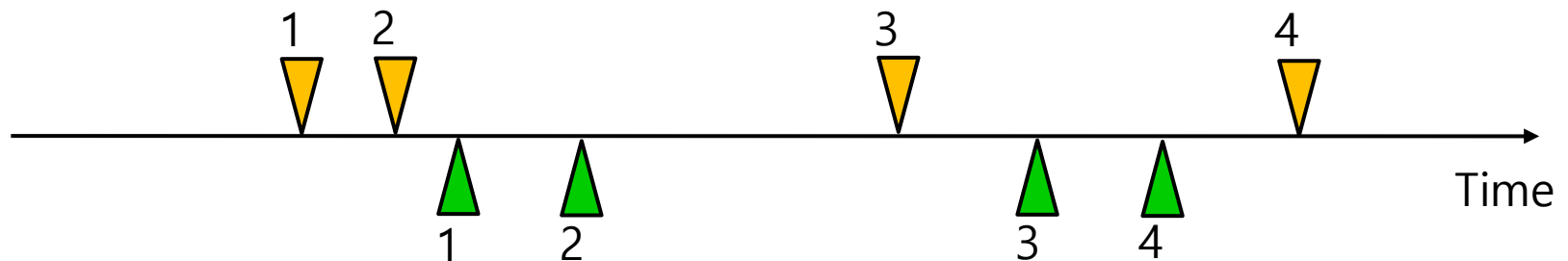
Bob : $800 - 400 = 400$

- 1) `B=Balance (Bob)`
- 2) `A=Balance (Alice)`
- 3) `SetBalance (Bob, B+100)`
- 4) `SetBalance (Alice, A-100)`

- 1) `A=Balance (Alice)`
- 2) `B=Balance (Bob)`
- 3) `SetBalance (Alice, A+500)`
- 4) `SetBalance (Bob, B-500)`

Assumption : A, B, Alice, and Bob are all global variables.

- What can happen if this code runs concurrently?



Alice's balance: \$200

Bob's balance: \$800

\$200

\$900

\$700

\$900

\$700

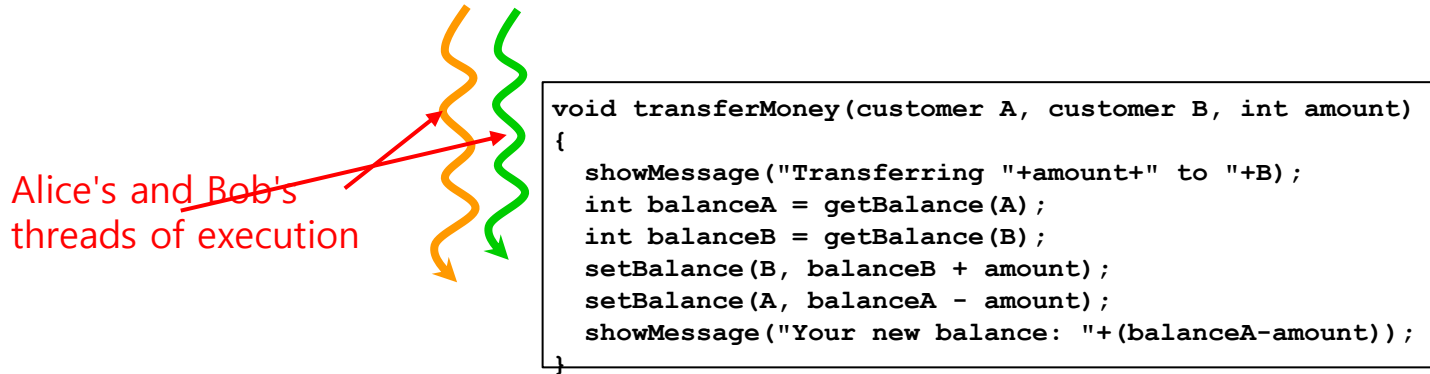
\$300

\$100

\$300



Problem : Race condition



- What happened?
 - **Race condition:** Result of the computation depends on the exact timing of the two threads of execution, i.e., the order in which the instructions are executed
 - Reason: Concurrent updates to the same state

Goal : Consistency

- What should have happened?
 - Intuition : It shouldn't make a difference whether the requests are executed concurrently or not
- How can we formalize this?
 - Need a **consistency model** that specifies how the system should behave in the presence of concurrency

Other consistency models

- Strong consistency
 - After update completes, all subsequent accesses will return the updated value
- Weak consistency
 - After update completes, accesses do not necessarily return the updated value; some condition must be satisfied first
 - Aws s3 (simple storage service)
 - Example: Update needs to reach all the replicas of the object
- Eventual consistency
 - Specific form of weak consistency: If no more updates are made to an object, then eventually all reads will return the latest value
- Why would we want any of these?
- How do we build systems that achieve them?

Plan for the next two lectures

- Parallel programming and its challenges
 - Parallelization and scalability, Amdahl's law ✓
 - Synchronization, consistency ✓
 - Mutual exclusion, locking, issues related to locking ← NEXT
 - Architectures: SMP, NUMA, Shared-nothing
- All about the Internet in 30 minutes
 - Structure; packet switching; some important protocols
 - Latency, packet loss, bottlenecks, and why they matter
- Distributed programming and its challenges
 - Network partitions and the CAP theorem
 - Faults, failures, and what we can do about them

Mutual exclusion

Critical section

```
void transferMoney(customer A, customer B, int amount)
{
    showMessage("Transferring "+amount+" to "+B);
    int balanceA = getBalance(A);
    int balanceB = getBalance(B);
    setBalance(B, balanceB + amount);
    setBalance(A, balanceA - amount);
    showMessage("Your new balance: "+(balanceA-amount));
}
```

- How can we achieve better consistency?
 - Key insight: Code has a **critical section** where accesses from other cores to the same resources will cause problems
- Approach: Mutual exclusion
 - Enforce restriction that only one core (or machine) can execute the critical section at any given time
 - What does this mean for scalability?

Locking

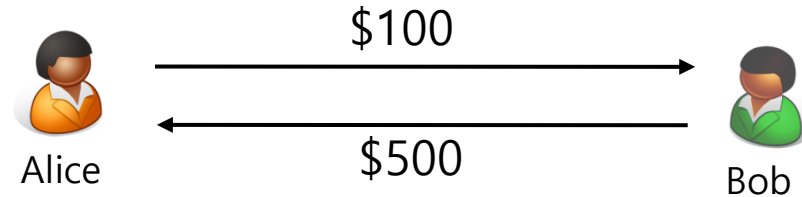
Critical section

```
void transferMoney(customer A, customer B, int amount)
{
    showMessage("Transferring "+amount+" to "+B);
    int balanceA = getBalance(A);
    int balanceB = getBalance(B);
    setBalance(B, balanceB + amount);
    setBalance(A, balanceA - amount);
    showMessage("Your new balance: "+(balanceA-amount));
}
```

- Idea : Implement locks

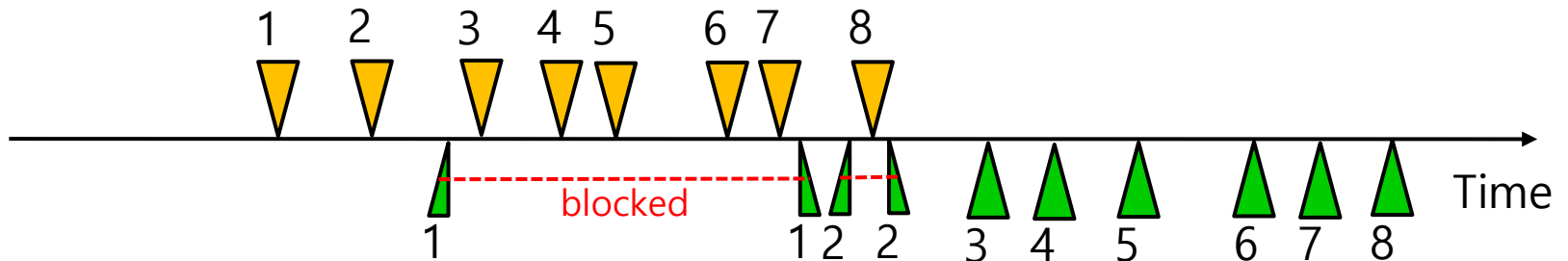
- If LOCK(X) is called and X is not locked, lock X and continue
- If LOCK(X) is called and X is locked, wait until X is unlocked
- If UNLOCK(X) is called and X is locked, unlock X

Locking helps!



- 1) **LOCK (Bob)**
- 2) **LOCK (Alice)**
- 3) `B=Balance (Bob)`
- 4) `A=Balance (Alice)`
- 5) `SetBalance (Bob, B+100)`
- 6) `SetBalance (Alice, A-100)`
- 7) **UNLOCK (Alice)**
- 8) **UNLOCK (Bob)**

- 1) **LOCK (Alice)**
- 2) **LOCK (Bob)**
- 3) `A=Balance (Alice)`
- 4) `B=Balance (Bob)`
- 5) `SetBalance (Alice, A+500)`
- 6) `SetBalance (Bob, B-500)`
- 7) **UNLOCK (Bob)**
- 8) **UNLOCK (Alice)**



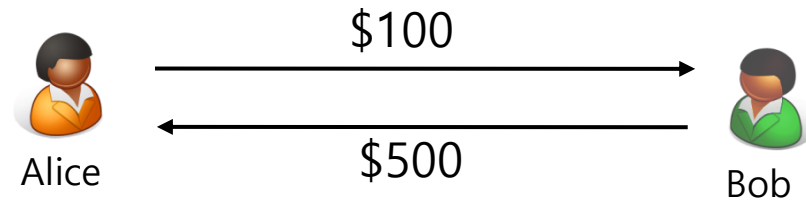
Alice's balance: \$200
Bob's balance: \$800

\$200 \$100
\$900 \$900

\$600 \$600
\$900 \$400

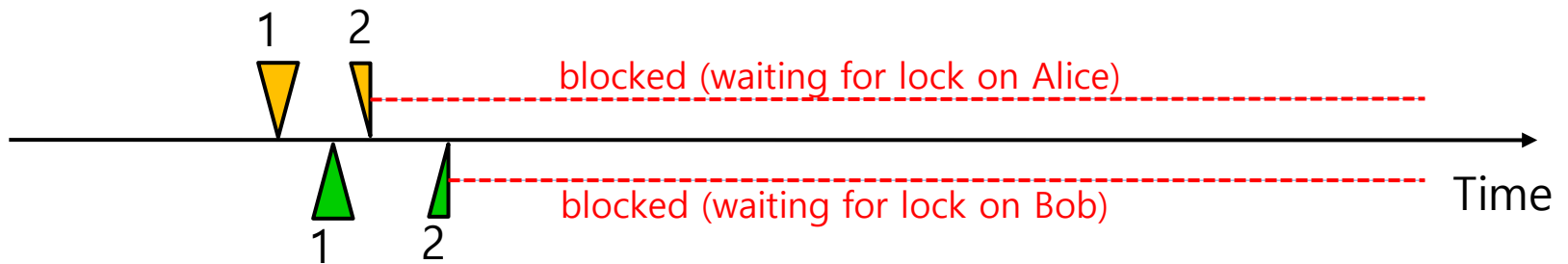


Problem: Deadlock



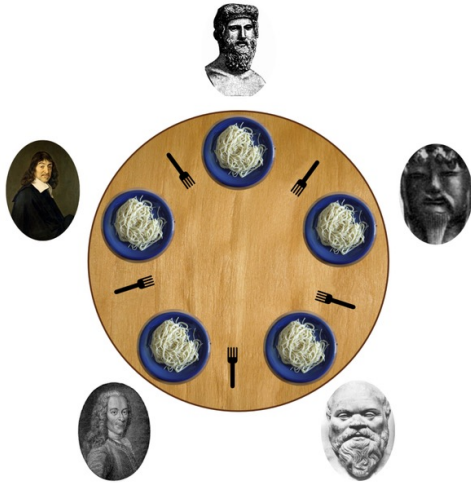
- 1) LOCK (Bob)
- 2) LOCK (Alice)
- 3) B=Balance (Bob)
- 4) A=Balance (Alice)
- 5) SetBalance (Bob, B+100)
- 6) SetBalance (Alice, A-100)
- 7) UNLOCK (Alice)
- 8) UNLOCK (Bob)

- 1) LOCK (Alice)
- 2) LOCK (Bob)
- 3) A=Balance (Alice)
- 4) B=Balance (Bob)
- 5) SetBalance (Alice, A+500)
- 6) SetBalance (Bob, B-500)
- 7) UNLOCK (Bob)
- 8) UNLOCK (Alice)



- Neither processor can make progress!

The dining philosophers problem

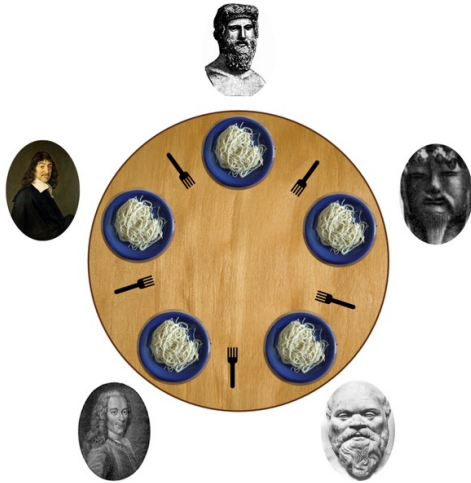


```
Philosopher:
repeat
  think
  pick up left fork
  pick up right fork
  eat
  put down forks
forever
```

- Rules

- The philosophers are sitting on the table and thinking individually. When a philosopher feels hungry, he eats spaghetti on both sides of the fork. If the philosopher is full, he put down fork, and the philosopher thinks again.
Thinking -> Hungry -> Eating -> Thinking -> ...
- Philosophers are blunt and have no say. And Philosophers do not care what others do.
- Philosophers are man of principle. They eat only when there are two forks. In other words, if they have one fork, they wait. For example, if Aristotle is eating, Cartesian and Plato can not eat next to it.

The dining philosophers problem



```
Philosopher:
repeat
  think
  pick up left fork
  pick up right fork
  eat
  put down forks
forever
```

- Problems

- A philosopher who has begun to eat can eat forever. Philosophers on both sides can not eat.
- If every philosopher holds his right fork, all philosophers are deadlocked.
- Let's say that philosophers A, B, and C are in order. B is waiting for A to finish the meal, and suddenly C starts eating. Then A finishes the meal and B waits for C's meal, and A starts eating again. In this way, if A and C repeat the meal, B starves to death (livelock)


What to do about deadlocks

- Many possible solutions, including:
 - Lock manager : Hire a waiter and require that philosophers must ask the waiter before picking up any forks
 - Consequences for scalability?
 - Others : impact for scalability?

Alternatives to locks

- Locks aren't the only solution!
 - There are many ways to handle concurrency...
 - ... But you Do have to handle it properly!
- Example : “Optimistic” concurrency control
 - Allow accounts to be updated concurrently
 - Try to “merge” the effects
 - If not possible, do things in sequence
- Example : Abort and restart
 - If not all locks are available, release all and retry
- What does this mean for consistency?

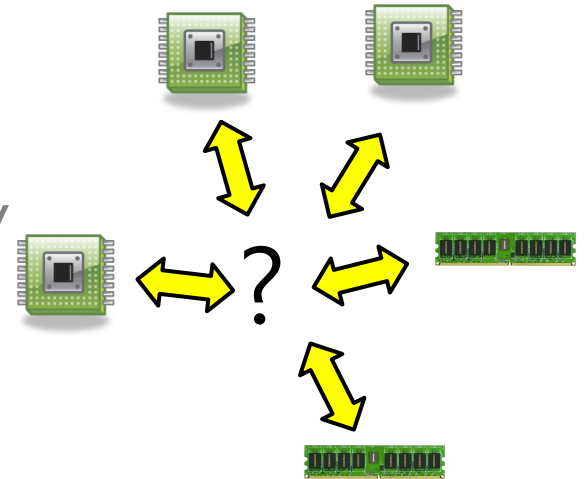
Plan for the next two lectures

- Parallel programming and its challenges
 - Parallelization and scalability, Amdahl's law ✓
 - Synchronization, consistency ✓
 - Mutual exclusion, locking, issues related to locking ✓
 - Architectures: SMP, NUMA, Shared-nothing 
- All about the Internet in 30 minutes
 - Structure; packet switching; some important protocols
 - Latency, packet loss, bottlenecks, and why they matter
- Distributed programming and its challenges
 - Network partitions and the CAP theorem
 - Faults, failures, and what we can do about them

Other architectures

- Earlier assumptions:

- All cores can access the same memory
- Access latencies are uniform



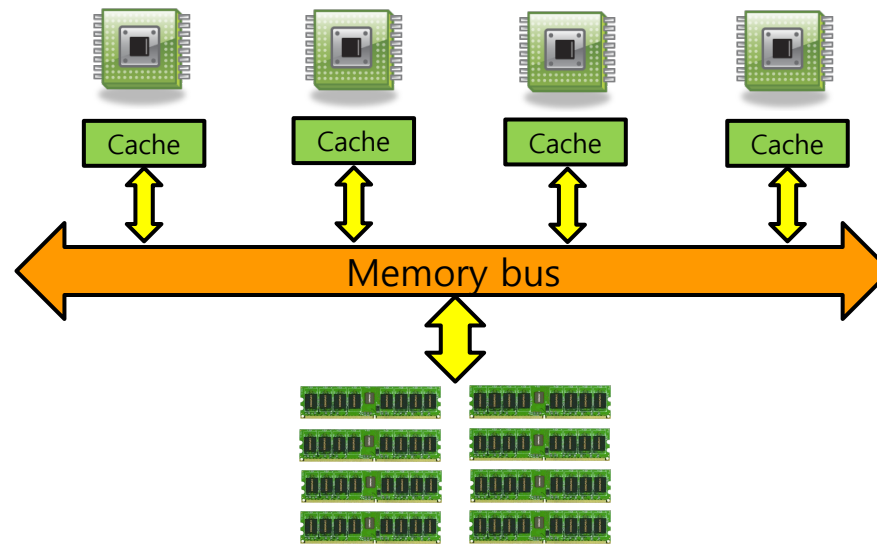
- Why is this problematic?

- Processor speeds in GHz, speed of light is 299 792 458 m/s
 - Processor can do 1 addition while signal travels 30cm
 - Putting memory very far away is not a good idea

- Let's talk about other ways to organize cores and memory!

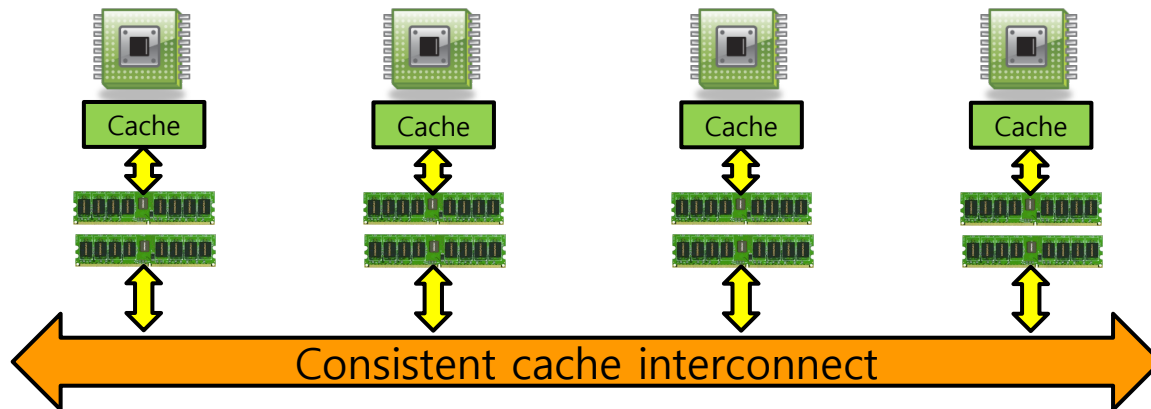
- SMP, NUMA, Shared-nothing

Symmetric Multiprocessing (SMP)



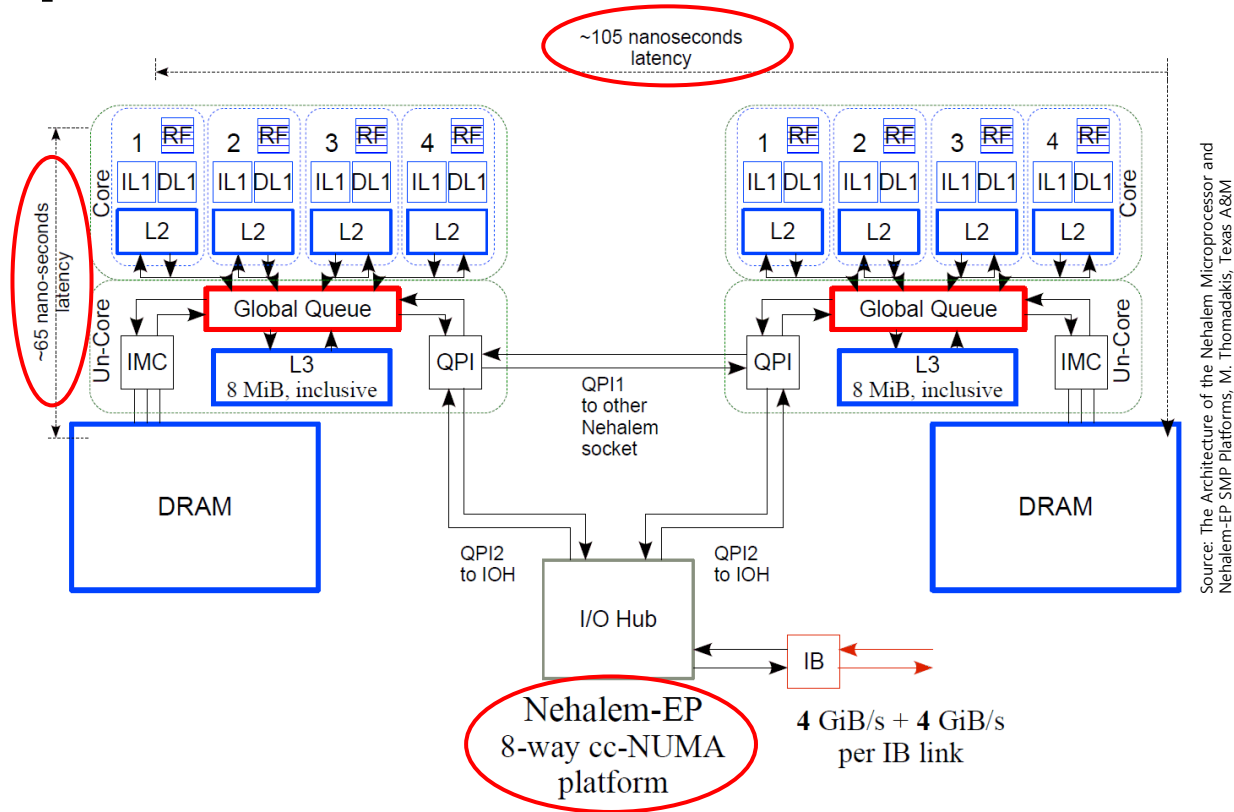
- All processors share the same memory
 - Any CPU can access any byte; latency is always the same
 - Pros: Simplicity, easy load balancing
 - Cons: Limited scalability (~10 processors), expensive

Non-Uniform Memory Architecture (NUMA)



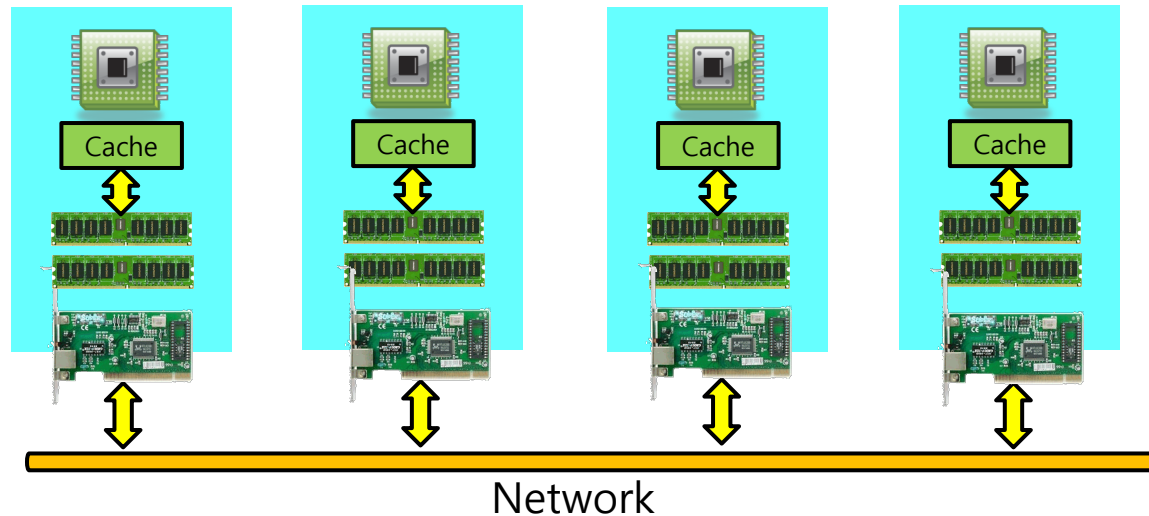
- Memory is local to a specific processor
 - Each CPU can still access any byte, but accesses to 'local' memory are considerably faster (2-3x)
 - Pros: Better scalability
 - Cons: Complicates programming a bit, scalability still limited

Example: Intel Nehalem




- Access to remote memory is slower
 - In this case, 105ns vs 65ns

Shared-Nothing



- Independent machines connected by network
 - Each CPU can only access its local memory; if it needs data from a remote machine, it must send a message there
 - Pros: Much better scalability
 - Cons: Nontrivial programming model

Plan for the next two lectures

- Parallel programming and its challenges ✓
 - Parallelization and scalability, Amdahl's law ✓
 - Synchronization, consistency ✓
 - Mutual exclusion, locking, issues related to locking ✓
 - Architectures: SMP, NUMA, Shared-nothing ✓
- All about the Internet in 30 minutes 
 - Structure; packet switching; some important protocols
 - Latency, packet loss, bottlenecks, and why they matter
- Distributed programming and its challenges
 - Network partitions and the CAP theorem
 - Faults, failures, and what we can do about them

For next time

- Homework submission due policy
 - Submit it by the day before the next class.
 - For example, if the next class is 12.25, submit it by 12.24 11:59 PM.
- Homework # 2 (due : 3.23(수) 자정)
 - Read the Vogels, paper “Eventually consistent”
 - <http://bit.ly/2vzQAzU>
 - Submit a review of the article into 1 A4-sized paper to me

For next time

- Homework submission due policy
 - Submit it by the day before the next class.
 - For example, if the next class is 12.25, submit it by 12.24 11:59 PM.
- Homework # 3 (due : 3.30(수) 자정)
 - Learn python & Django (backend)
 - <https://tutorial.djangogirls.org/ko/>
 - Output : your own blog site
 - Please read the above site and accomplish the project. And you create your own blog and submit it. You will have to complete this homework successfully before you are ready to start the term project. If you do not do your homework, the term project will be difficult to perform.

Next Time



- Next time you will learn about :
 - Internet basics; faults and failures

Any Questions?



Credits & References

- Credits : A. Haeberlen, Z. Ives (University of Pennsylvania)