CMSC 691
High Performance Distributed Systems

Introduction

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu

Course global idea:

- Comprehend the power and limitation of parallel and distributed computing under temporal, spatial, and memory locality constraints

Making programs faster:

- Increase bandwidth/throughput (tasks per unit time)
- Decrease latency (time per task)

Similar to:

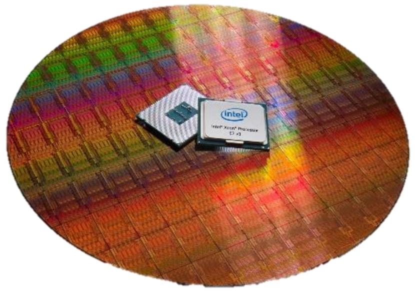
- Internet: connection speed/bandwidth vs latency
- Traffic: lanes vs speed

Making programs faster:

- Decreasing time per task is usually harder, with fewer gains
    - Profile and rewrite the code
    - Improve the hardware
        - Reducing # cycles for an instruction
        - Specialized hardware. FPGAs.
- CPUs have been going towards more cores rather than raw speed
    - Desktop: 4, 6, 12 cores evolution, frequency < 4 GHz
    - Mobile: 10-cores cell phone, what for?

The race of increasing the number of cores:

- Why not 32 core desktop CPU today?
- Economic / Business motivation $$$:
  - Intel / AMD / ARM "free market"
  - Marketing and vaporware
- Manufacturing motivation:
  - Energy and thermal design power (TDP)
  - Lithography and wafers
  - End of Moore's law



Barriers to parallelization:

- Embarrassingly parallel are trivial but dependencies cause problems
- Unable to start task until previous task finishes
- Synchronization and combination of results
- Difficult to reason about, since execution may happen in any order
- Sequential tasks will always dominate maximum performance
- Some sequential problems may be parallelizable by recoding
- However, no matter how many processors you have, you won't be able to speed up the program as a whole (Amdahl's Law)

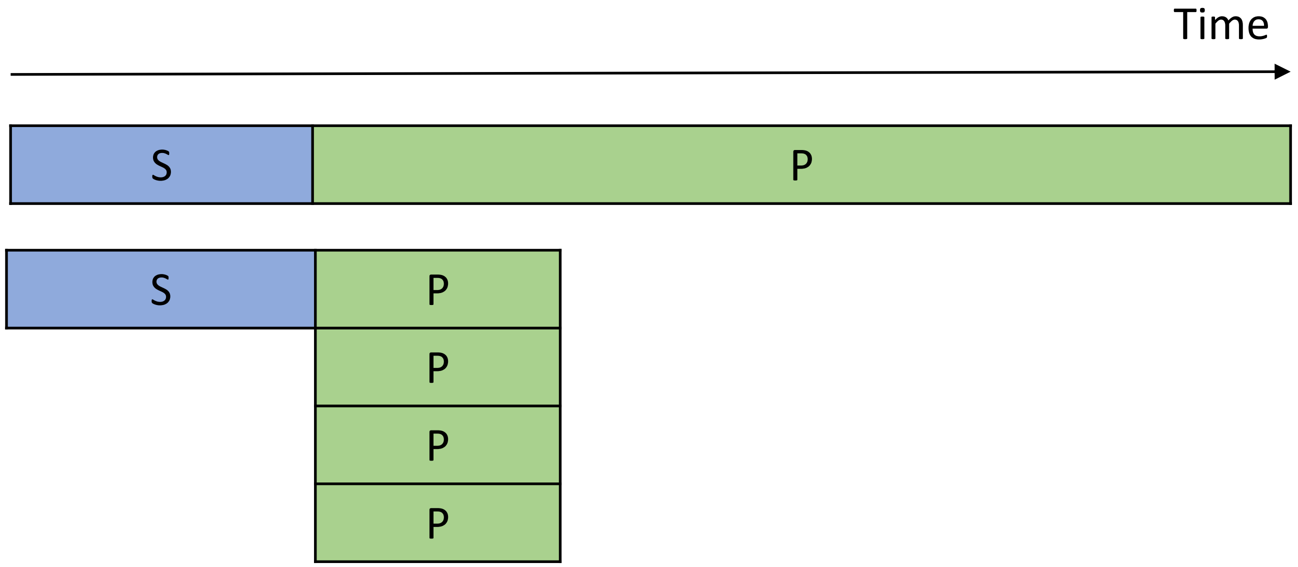
Limitations of speedup:

- Programs have a sequential part and a parallel part (S + P = 1)

S: fraction of serial runtime in a serial execution

P: fraction of parallel runtime in a serial execution

- With N processors, best case, how much can we speed up?



Limitations of speedup:

T_S: time for the program to run in serial

T_P: time for the program to run in parallel

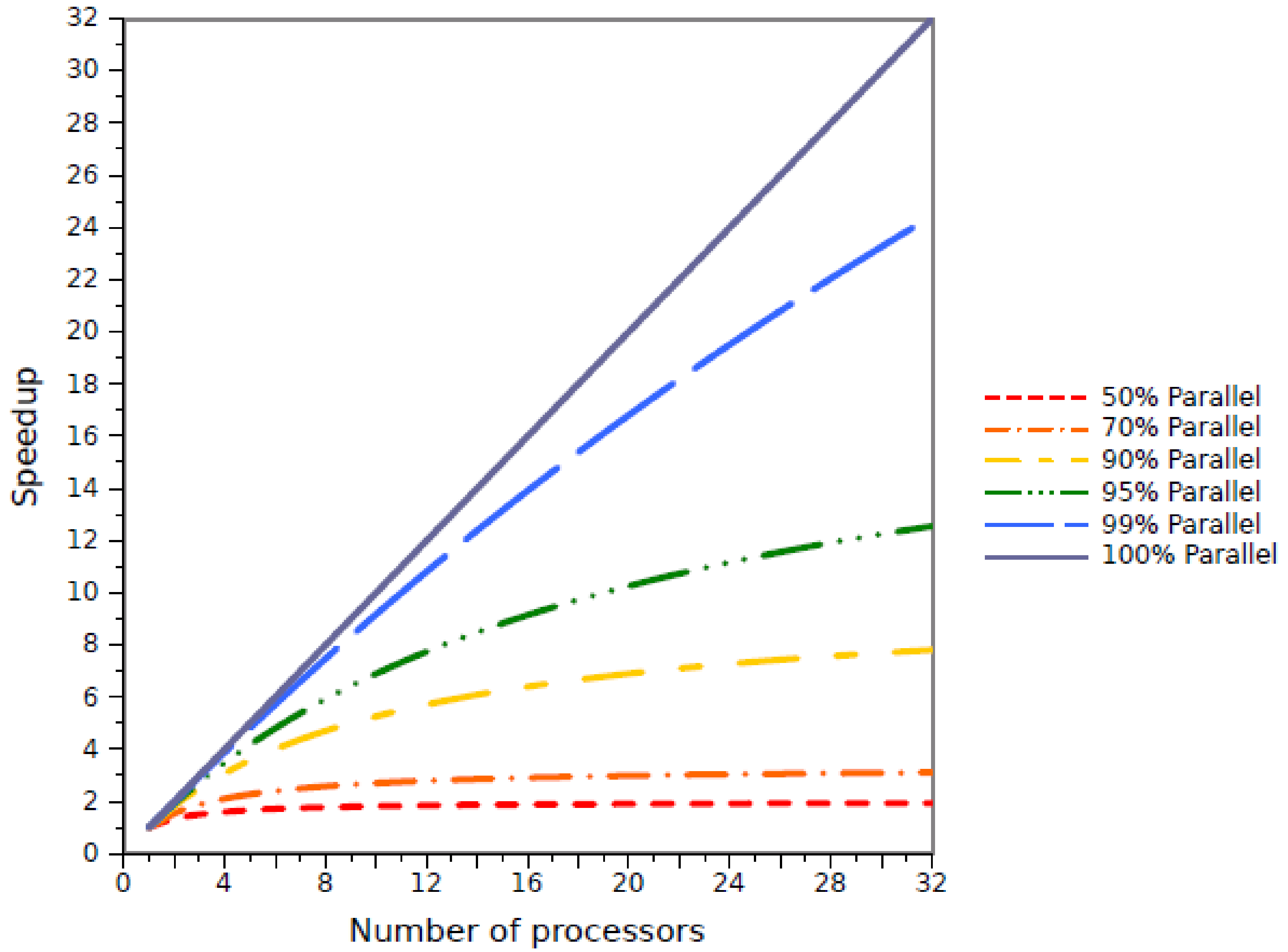
N: number of processors/parallel executions

- Under perfect conditions (assuming no overhead for parallelizing):

$$T_P = TS \cdot \left(S + \frac{P}{N}\right) \qquad Speedup = \frac{T_S}{T_P} = \frac{1}{\left(S + \frac{P}{N}\right)}$$

- Real-world: overheads may be a headache

Fixed-size problem scaling, varying fraction of parallel code:



Amdahl's Law (1967):

- Improvements in processor design for single processors would be more effective than designing multi-processor systems

Rewrite S as 1-P $$Speedup = \frac{1}{\left((1-P)+\frac{P}{N}\right)}$$

Maximum asymptotic speedup $= \frac{1}{(1-P)}$ since $\frac{P}{N} \rightarrow 0$

- Consequence: for over 30 years, most performance gains did indeed come from increasing single-core performance, while today we live the core count race

Amdahl's Law generalization:

- The program may have many parts, each of which we can tune to a different degree

  $f_1, f_2, \ldots, f_n$: fraction of time in part n

  $Sf_1, Sf_2, \ldots, Sf_n$: speedup for part n

$$Speedup = \frac{1}{\frac{f_1}{Sf_1} + \frac{f_2}{Sf_2} + \cdots + \frac{f_n}{Sf_n}}$$

- Consider a program with 4 parts. Which option is better?

Part	Fraction of Runtime	Speedup Option 1	Speedup Option 2
1	0.55	1	2
2	0.25	5	1
3	0.15	3	1
4	0.05	10	1

Amdahl's Law generalization:

- Option 1

$$speedup = \frac{1}{0.55 + \frac{0.25}{5} + \frac{0.15}{3} + \frac{0.05}{5}} = 1.53$$

- Option 2

$$speedup = \frac{1}{\frac{0.55}{2} + 0.45} = 1.38$$

- Performance gain vs effort tradeoff
  - Coding time required to achieve a small speedup
  - Code optimization & parallelization takes time to implement
- Overhead consideration, especially latency in distributed systems

Empirically estimating the maximum parallel speedup:

- Quick way to guess the fraction of parallel code ($P_{estimated}$)
- Run the program forcing 1 core and N cores, compute speedup
- Predict the speedup for a different number of processors

$$P_{estimated} = \frac{\frac{1}{speedup} - 1}{\frac{1}{N} - 1}$$

- Estimated speedup using N_2 cores = $\frac{1}{\left((1 - Pestimated) + \frac{P_{estimated}}{N_2}\right)}$
- Real-world: additional overhead and memory bottleneck happen

Gustafson's Law:

- Speedup predictions according to Amdahl's law are pessimistic
- Amdahl's law assumed the fraction of parallelizable code is fixed
- Gustafson: parallelism increases when the problem size increases

n: problem size

S(n): fraction of serial runtime for a parallel execution

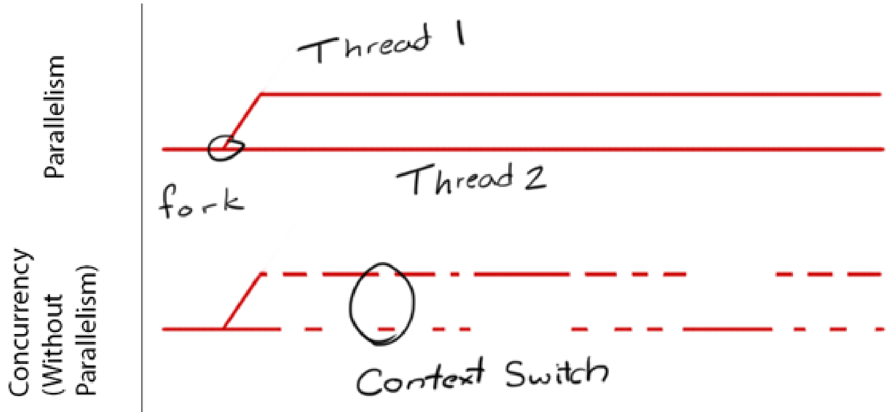
P(n): fraction of parallel runtime for a parallel execution

$$\text{Speedup} = S(n) + N \cdot P(n)$$

- 10x speedup in a 10 seconds vs 10 days runtime problem
- Serial and overhead times become irrelevant for very large problems

Concurrency vs parallelism:

- **Concurrency** is when two or more tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant. Does not require multiple processors. Keyword: *Interruptible*.
- **Parallelism** is when two or more tasks literally run simultaneously at a particular moment in time. Requires multiple processors. Keyword: *Independency*.

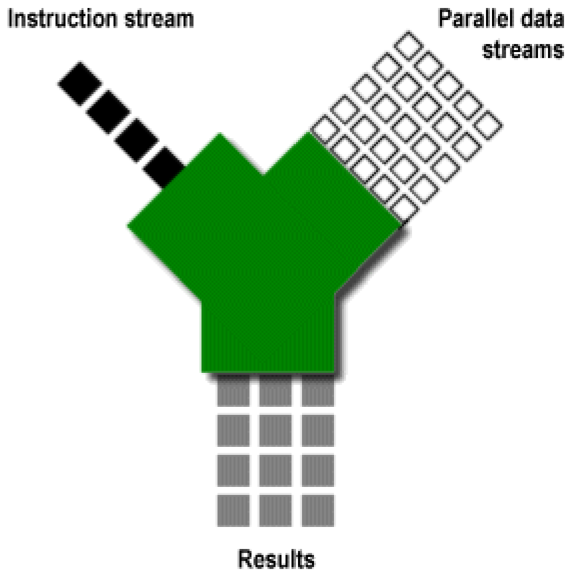


Levels of parallelism:

- Data-level parallelism
- Instruction-level parallelism
- Thread-level parallelism
- Process-level parallelism

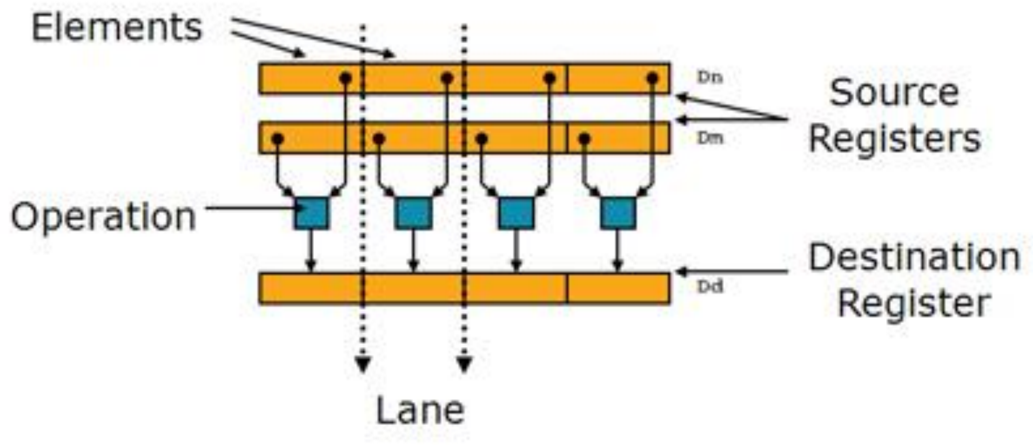
Flynn’s Classification Scheme:

- SISD (Single Instruction Single Data)
- SIMD (Single Instruction Multiple Data)
- MIMD (Multiple Instruction Multiple Data)



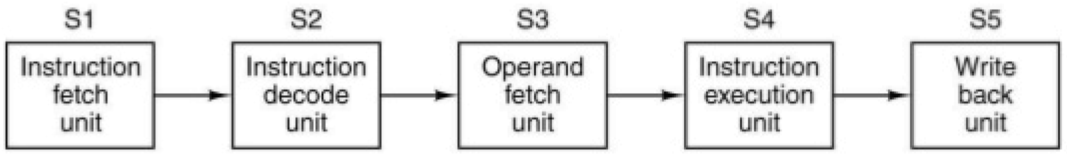
Data-level parallelism:

- Perform identical operations on (possibly) lots of data
- Vector processor arrays, Streaming SIMD Extensions (SSE)
- Drawbacks: portability and compatibility

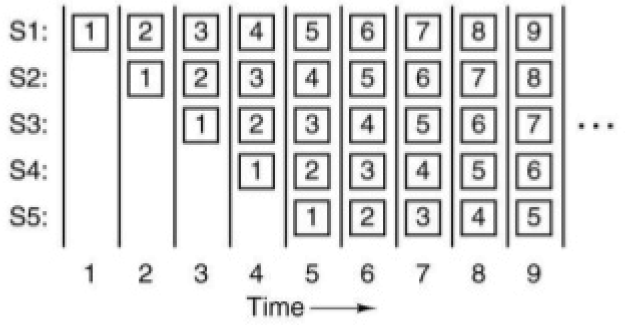


Instruction-level parallelism:

- Overlapping among instructions
- Pipelining, e.g. simultaneous ALU execution / data transfer
- Superscalar CPUs

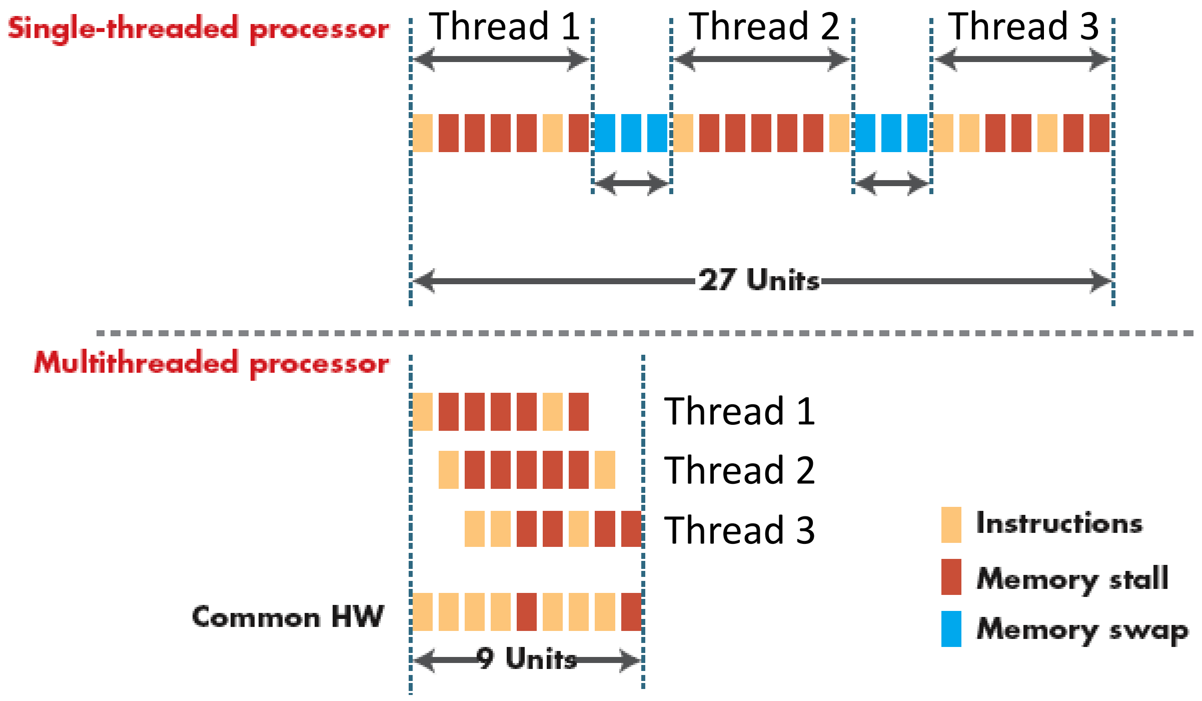


(a)



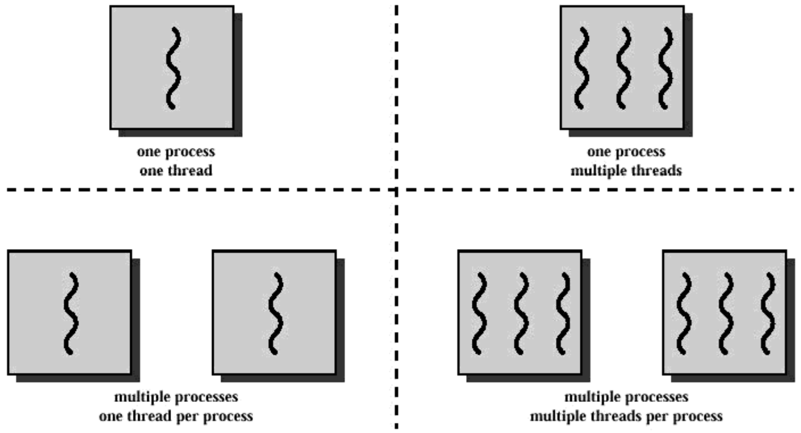
Thread-level parallelism:

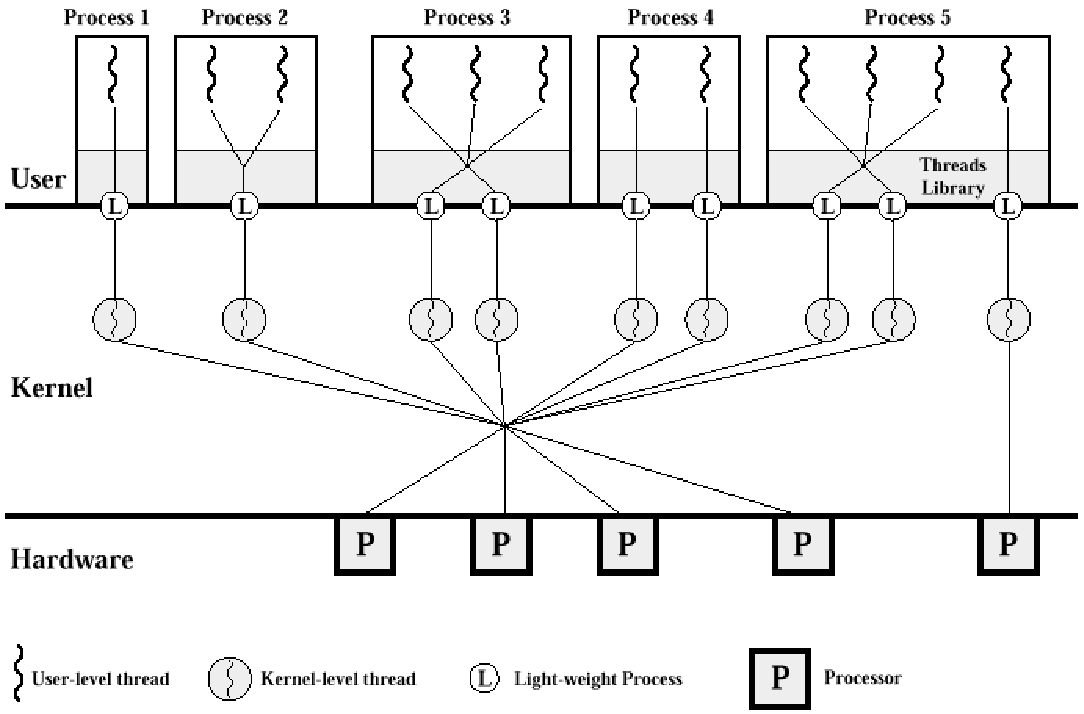
- Multiple threads (tasks) in a process
- Shared-memory space in multiprocessors



Process-level parallelism:

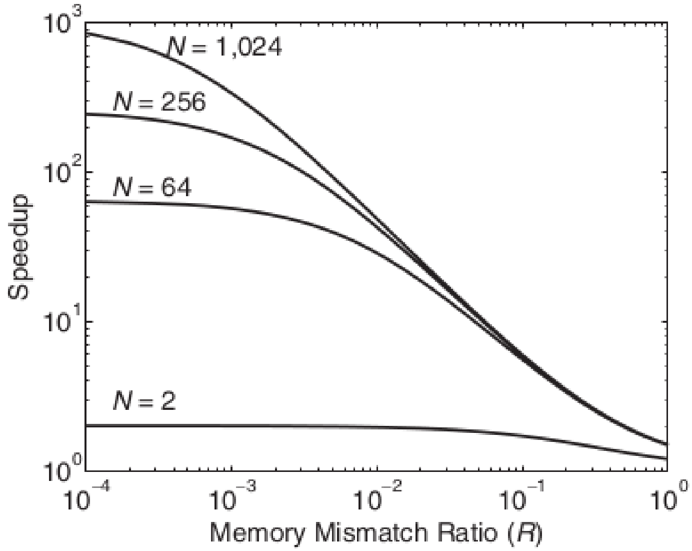
- Multiple processes in a system
- Separate memory spaces





Communication overhead:

- Interconnection network delay (HFT)
- Memory bandwidth and latency
- Memory collisions and RAW/WAW conflicts
- Memory hierarchy
  - Register <-> cache <-> RAM <-> disk
  - Memory-bound code is limited by bandwidth and cache misses



Enhancing performance (architecture and compiler):

- Cache is useful if code shows temporal and spatial locality
- Data load/store operations take place using blocks of data
- Pipelining and segmentation
- Asynchronous execution & data transfer
- Branch prediction & speculation, out of order execution
- Loop unrolling
- Memory layout of multi-dimensional arrays

Enhancing performance (what we can do in our code):

- Understand your algorithm’s computational complexity
- Optimize memory access patterns, data types, and structures
- Optimize arrays size and alignment
- Minimize I/O interruptions
- Optimize conditional statements
- Use functions calls properly
- Binary search or sequential search, sorting, min/max
- Profile your code to identify performance bottlenecks

Distributed systems

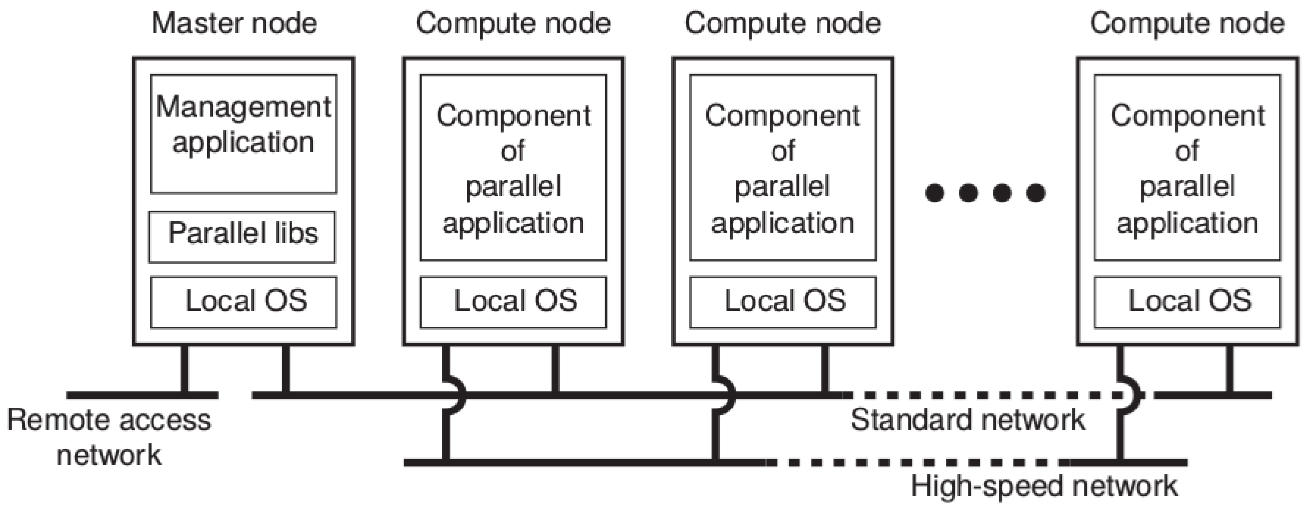
- A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system
- Transparency: access, location, migration, concurrency, failure
- Flexibility: easier to change and update
- Reliability: availability, maintenance, fault tolerance
- Performance: communication delays, transparency cost
- Scalability: plug & play computing elements, problem size
- Security: well …

Scaling distributed systems

- Hide communication latencies. Asynchronous communication.
- Distribution. Partition data and computation across machines.
- Replication. Mirroring servers. Inconsistencies and synchronization.
- False assumptions:
  - Network is reliable, secure, homogeneous
  - Latency is zero, bandwidth is infinite
  - Migration cost is zero

Distributed computing systems

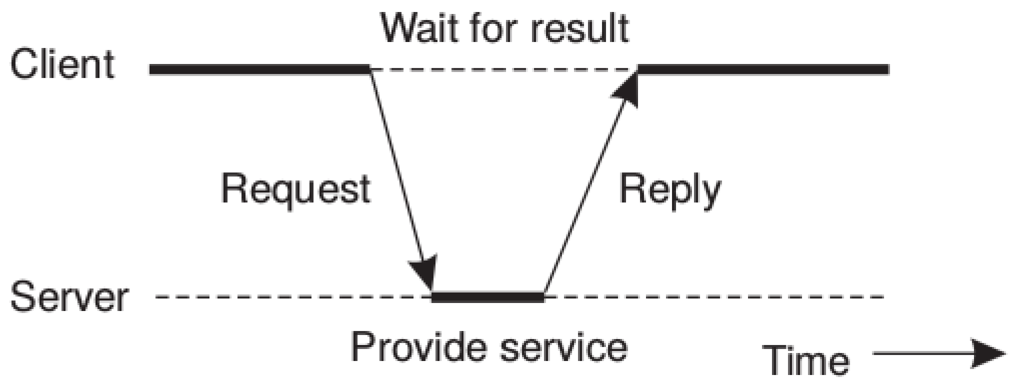
- Configured for high-performance computing
- Group of homogeneous high-end systems connected through high-speed LAN. One master node and multiple compute nodes

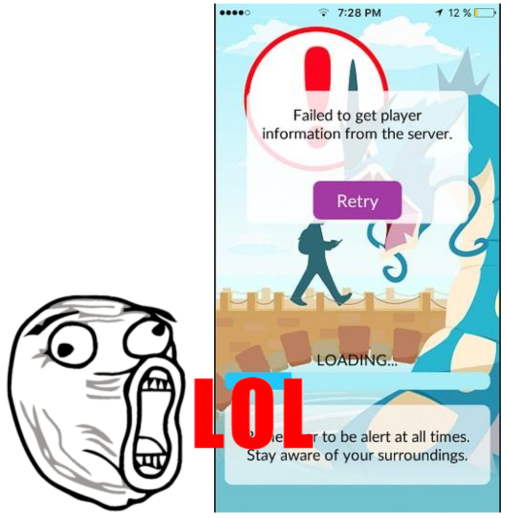


- Grid and cloud computing: dispersed across multiple locations

Centralized architectures: client-server model

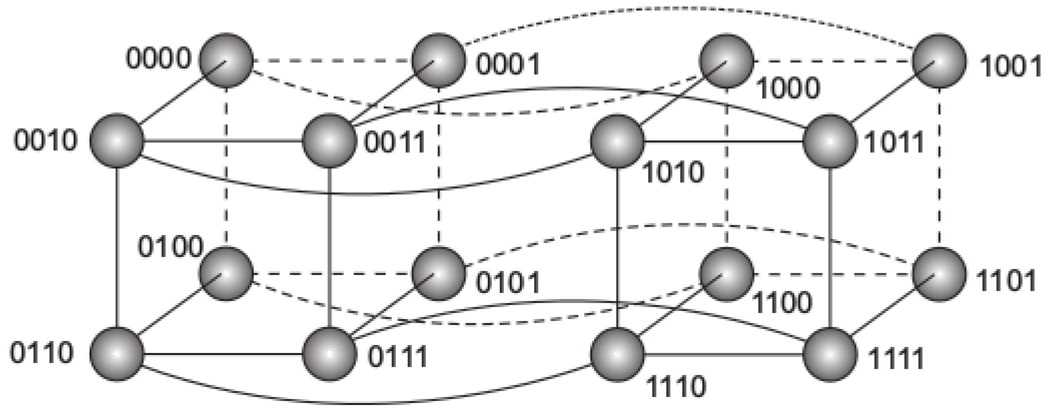
- There are processes offering services (high-end servers)
- There are processes that use services (thin clients)
- Clients and servers can be on different machines
- Clients follow request/reply model for using services




7:28 PM
12 %
Failed to get player information from the server.
Retry
LOADING...
LOL
r to be alert at all times.
Stay aware of your surroundings.

Decentralized architectures: peer-to-peer systems

- Structured / unstructured P2P topology
- Organize the nodes in a structured overlay network such as a logical ring, or a hypercube, and make specific nodes responsible for services based only on their ID.
- Requires efficient P2P routing. Flooding, random walk. Superpeer.



CMSC 691
High Performance Distributed Systems

Introduction

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu