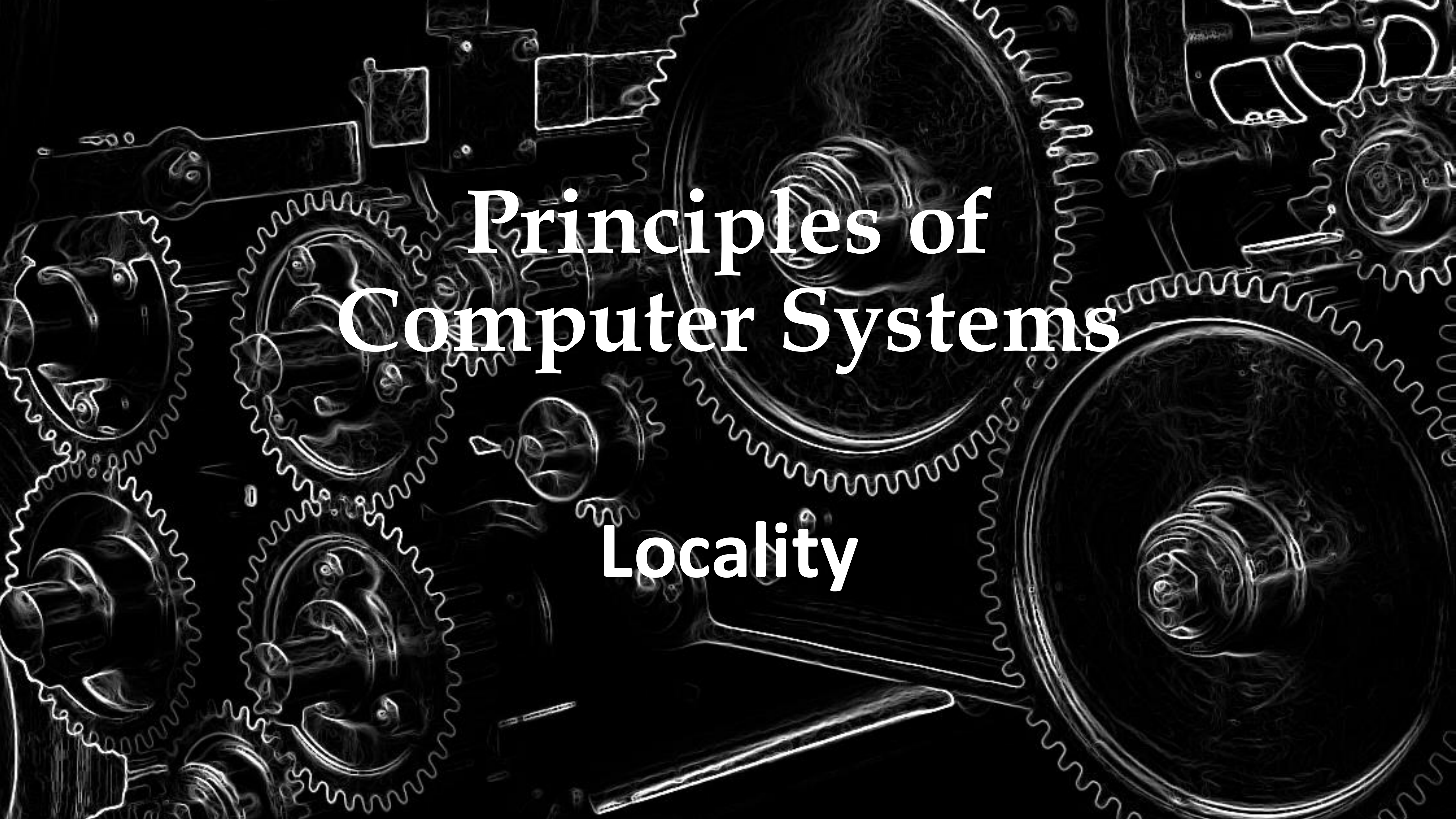


Principles of Computer Systems

Locality

EPFL

The "Free computation" fallacy

You buy a 4GHz CPU. What percentage of the time is it actually doing useful work?

Reality check:

- Theoretical peak: 4 billion operations/second
- Actual **useful** work: Often < 10%
- Rest: Waiting for data

Obsess over O(n) algorithmic complexity

But in systems, the constants (latency) dominate

Efficient data movement is all that matters

- Fundamental cost associated with data movement
  - Time/energy ➔ Moving data between compute←→storage
  - Bandwidth ➔ Communication links have limited capacity
  - Queueing ➔ Contention induces delay
- Some reported numbers wrt data movement:
  - Google datacenter tax: 50—60% CPU cycles
  - Google consumer device workloads: ~62.7% of total system energy

Why can't we just make things faster?

https://www.brendangregg.com/blog/2017-05-09/cpu-utilization-is-wrong.html
Kanev et al., "Profiling a warehouse-scale computer," ISCA 2015
Ghose et al., "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," ASPLOS 2018

The hardware wall

- Fundamental limitations exist:
  - Dennard scaling has failed
    - Power density limits clock speeds
    - Dark silicon exists
  - Cooling constraints
    - Even 3D chips can’t pack more compute
  - Speed of light
    - Signals take time to cross a chip

The latency hierarchy

Access type	Latency	Relative to L1	
L1 cache	~1 ns	1x	1 second
L2 cache	~4 ns	4x	
L3 cache (local)	~12-20 ns	12-20x	
L3 cache (remote socket)	~30-90 ns	30-90x	
Local DRAM	~80 ns	80x	1.5 minutes
Remote DRAM (NUMA)	~120-200ns	130-200x	
CXL memory (new)	~150-300ns	150-300x	
NVMe SSD	~2-40 us	2,000-40,000x	6—11 hours
Network (remote machine)	~2+ us	2,000+x	
HDD	~10ms	10,000,000x	4 months

These gaps have existed since the beginning of computing!

How did we learn to deal with them?

EPFL

What is locality?

The general principle:

Locality refers to the idea that interactions or effects are limited to immediate, adjacent areas

In computing:

Locality refers to the efficiency of data access and processing—the tendency of a process to access a relatively small subset of its total address space over a short period

Why locality matters

Modern computers are designed using the principle of locality:

- Caches (keep recently used data close)
- Predictive loading (prefetch what's likely needed)
- Faster storage transfer (batch nearby data)

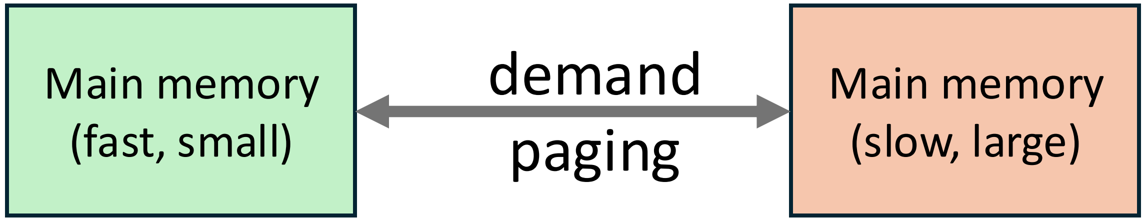
Locality isn't just an optimization—it's a *design assumption* baked into hardware

Goal: Minimize data movement or have data ready before it's needed

Historical context: The birth of virtual memory

Atlas Computer (University of Manchester, 1962)

- First implementation of virtual memory
- Problem: Main memory was expensive and small
- Insight: Programs don’t need ALL data ALL the time



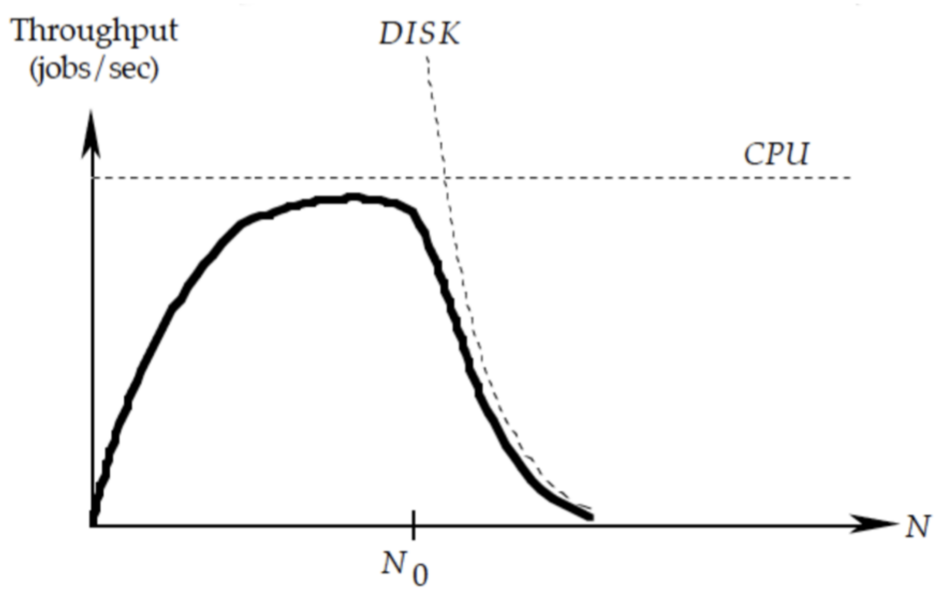
One-level store: Appeared as a single, contiguous, high-speed memory space

Background: “Paging to death” → thrashing

Q. A system is running N jobs. As N increases, throughput rises … then suddenly crashes. Why?

At N_0, more paging → CPU idle

→ scheduler adds jobs → collapse



“**thrashing** was unexpected, a sudden drop in throughput of a multiprogrammed system… I explained the phenomenon in 1968 and showed that a **working-set memory controller** would stabilize the system.”

- Peter Denning

EPFL

Working set model

The working set describes the set of information a process needs to access in a given period to carry out its computation

- Models program behavior over time
- Two perspectives:

Programmer's view	Smallest collection of data needed in memory for efficient execution
System's view	Set of pages referenced in recent time window

EPFL

Working Set: Visual Example

```
Time:      1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
Pages:     A   B   A   C   A   B   D   D   E   F   E   F   E   G   G

Window τ = 4:

At t=6:  W(6,4)  = {A, B, C}     (pages in t=3..6)
At t=10: W(10,4) = {D, E, F}     (pages in t=7..10)
At t=15: W(15,4) = {E, F, G}     (pages in t=12..15)
```

Key property:

- If physical memory ≥ working set → few page faults
- If physical memory < working set → thrashing

Question …

Imagine you are playing an open-world game (like GTA or Zelda). Describe how the working set changes in these three phases:

- The loading screen: You are loading 'Level 1'
- Gameplay: You are walking around a specific town square
- Fast travel: You teleport to a completely different city on the map

- **Loading screen**: Data being streamed from disk to memory
- **Gameplay**: Stable working set (textures and geometry for nearby buildings)
- **Fast travel**: Phase change → old working set (town A) to new working set (town B)

Relationship: Working set and locality

- The working set is a reflection of the current active locality of reference for a process

Concept	Role
Locality	Dictates which resources are critical
Working set	Leverages locality to maintain useful resources

- The working set fluctuates based on locality pattern changes throughout execution
- **Without locality**: cannot predict future resource requirements → inefficient system

Working set in modern systems

System	Working set	"Paging" equivalent
Virtual memory	Recently-used pages	Page faults
CPU cache	Hot cache lines	Cache misses
TLB	Active translations	TLB misses
Database buffer	Hot pages	Storage I/O
Web cache	Popular objects	Origin fetch
CDN	Regional content	Cross-region fetch

EPFL

Three types of locality

1. **Temporal**
   - Recently accessed → likely accessed again
2. **Spatial**
   - Nearby addresses → likely accessed together
3. **Network**
   - Physically close → faster access

Temporal locality: Repeated access over time

- Repeatedly accessing the same location over a short time

```
int sum = 0;
int array[10000];
for (int i = 0; i < 10000; i++) {
  sum += array[i]; // 'sum' accessed 10,000 times!
}
```

Question: Which variable exhibits temporal locality?

`sum`: accessed every iteration → keep it in a register

- Other examples:
  - Loop counters
  - Function return addresses on the stack
  - Hot objects in web caches (popular videos)

Spatial locality: Nearby access in space

- Access nearby memory locations within a small time frame

```
int array[1000];
int sum = 0;
for (int i = 0; i < 1000; i++) {
  sum += array[i]; // Consecutive memory locations
}
```

Q. Why is this efficient even though one variable is being accessed?

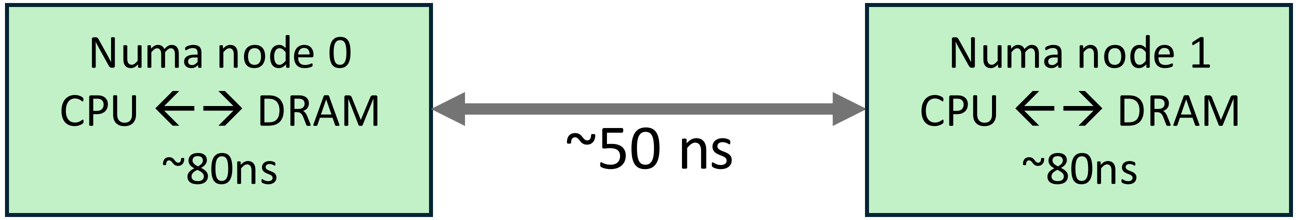
```
Memory: [array[0]][array[1]][array[2]]...[array[15]]
        └───────────── one cache line ─────────────┘
```

  - Access array[0] once, rest are already cached
- Other examples:
  - Sequential access
  - Instruction fetching (code is sequential)
  - Database table scans

Network locality: Distance matters

- Accessing data that is physically "**near**" in the system topology is faster

Multi-socket machine



- Local access: ~80 ns
- Remote access: ~130—200 ns
  - CPU from node 0 accesses memory on the remote socket

Network locality examples

- CPU cache hierarchy (L1 → L2 → L3)
- NUMA memory placement
- CDN edge servers
- Database read replicas
- Distributed cache sharding

Engineering goal: Shorten the wire!

EPFL

Exercise: Identify the locality type

#	Scenario	Temporal?	Spatial?	Network?
1	LRU keeping hot pages in RAM			
2	Matrix multiply with loop tiling			
3	CDN caching popular videos at edge			
4	`jemalloc` per-thread memory caches			
5	RSS steering packets to CPU cores			

Exercise: Identify the locality type

#	Scenario	Temporal?	Spatial?	Network?
1	LRU keeping hot pages in RAM	Recency predicts future access	✗	✗
2	Matrix multiply with loop tiling	Reuse blocks	Access contiguous submatrices	✗
3	CDN caching popular videos at edge	Cache popular content	Prefetch video segments	Nearby users
4	`jemalloc` per-thread memory caches	Thread reuses its cache	✗	Cache is core-local
5	RSS steering packets to CPU cores	Connection state reused	✗	Pinned to core

Approaches using locality principle

- Caching
- Prefer sequential access
- Partitioning
- Batching

Caching: The most basic optimization

- *Keep a working set of data close to the CPU that is used frequently*
- Ubiquitous in systems
  - CPU caches: L1, L2, L3
  - MMUs: TLB (translation lookaside buffer)
  - Networks: edge caches, CDNs
  - OS/DB: page cache, buffer pool
  - Storage device: DRAM in SSDs

Sequential acecss: Physical properties

- *Sequential access is faster than random access*
- Comes from the physical properties of devices
  - Hard drives
    - Mechanically moving parts: seek time >> transfer time
    - Reading a byte is not cheaper than reading a page
  - Flash/solid state devices
    - Write unit is pages/blocks, not bytes
  - DRAM
    - Row buffer hits are fast; activations are slow
- Example: write-ahead log converts random writes to sequential

Partitioning: Divide and conquer

- *Split resources and process independently*
- Embarrassingly parallel jobs
  - No synchronization required
  - Can work independently
  - Decompose large jobs, process in parallel
  - Example: MapReduce
- Limitations
  - Non-uniform distribution (hot keys in KV store)
  - Tasks requiring synchronization
  - Not always applicable

Batching: Amortize data movement

- *Collect multiple operations and process them together*
- Pay the movement cost once, use the data for many operations
- Data/code stays hot in cache during batch processing
- Examples:
  - Storage IO: io_uring batches syscalls
  - Databases: group commits, batched writes
  - Locks: Cohort locks batch by NUMA node

Examples in detail

1. Data layout
2. Locality in locking protocols

3. False sharing
4. Evolving memory hierarchy

The matrix access problem

Scenario: MxN matrix stored in row-major order

```
Memory: A₁₁ A₁₂ A₁₃ ... A₁ₙ A₂₁ A₂₂ ... Aₘₙ
        └────── row 1 ──────┘└── row 2 ──...
```

Two traversal patterns

```c
// Loop A: Row-major traversal
for (int i = 0; i < M; i++)
  for (int j = 0; j < N; j++)
    process(A[i][j]);
```

```c
// Loop B: Column-major traversal
for (int j = 0; j < N; j++)
  for (int i = 0; i < M; i++)
    process(A[i][j]);
```

Setup: 4-byte integers, 64-byte cache lines, 1000×1000 matrix

1. Which loop is faster?
2. Predict the cache miss rate for each
3. Estimate the performance difference

Solution: Loop A (row-major)

Matches storage layout:

Access: `A[0][0], A[0][1], A[0][2], ...` (sequential)

Cache line for `A[0][0]` contains `A[0][0..15]`

→ `Miss on A[0][0]`

→ `Hit on A[0][1] through A[0][15]`

Miss rate: 1 miss per 16 accesses = 6.25%

EPFL

Solution: Loop B (column-major)

Mismatches storage layout:

`Access: A[0][0], A[1][0], A[2][0], ...` (stride = 4000 bytes)

Each cache access is on a different cache line

→ `Miss on A[0][0]`

→ `Miss on A[1][0]`

→ `Miss on A[2][0]`

Miss rate: 100%

Performance difference: Typically **10-20×** on modern CPUs!

Q. What if two matrices (10Kx10K) are multiplied? Does row-major approach work?

EPFL

Beyond matrices: Row vs. column stores

- **Row store (traditional OLTP)**

```
Storage: [ID:1, Name:Alice, Age:30, City:LA]
         [ID:2, Name:Bob, Age:40, City:NY]
```

  - Query: `SELECT AVG(Age) FROM Users`
  - Issue: must read `Name` and `City` → leads to cache pollution

- **Column store (analytics/OLAP)**

```
Storage: ID: [1, 2, 3, ...]
         Name: [Alice, Bob, ...]
         Age: [30, 40, 50, ...] ← contiguous!
```

  - Query: `SELECT AVG(Age) FROM Users`
  - Benefit: Only Age column is loaded → spatial locality

Takeaway: Row vs. column stores

- Optimal layout depends on access pattern
  - Analytics → columns
  - Transactions → rows

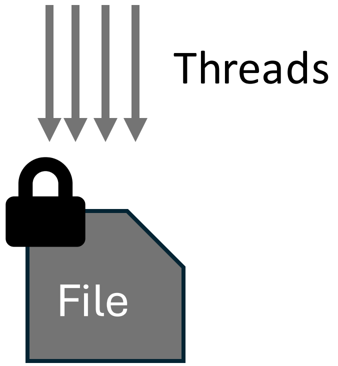
Q. What if you need BOTH? How do systems like SAP HANA handle this?

Hybrid layouts, materialized views, or maintaining both formats

EPFL

Why do locks care about locality?

- Locks serialize access – that's the obvious cost
  - Provide mutually exclusive access to shared data
  - Orders waiters accessing the critical section
- **Hidden cost:** Locks induce massive data movement
- Example: Threads accessing a file protected by a lock



- Every lock handoff = cache line transfer

Lock algorithms try to minimize the movement of shared data!

EPFL

Test-and-set: Locality disaster

```
void lock(atomic_t *L) {
  while (test_and_set(L) != 0) ; // spin
}
void unlock(atomic_t *L) { *L = 0; }
```

Q. What happens with 4 threads on 2 NUMA nodes?

```
T₀ (Node 0) acquires lock → cache line moves to Node 0
T₁ (Node 1) spins, writes → cache line moves to Node 1
T₂ (Node 2) spins, writes → cache line moves to Node 0
T₃ (Node 3) spins, writes → cache line moves to Node 1
T₀ unlocks → cache line moves to Node 0
```

Cache-line bouncing: ~200 ns per transfer x many transfers per acquire

- This saturates the memory interconnect

EPFL

Queue locks: Reduce contention

- MCS lock idea: Each thread spins on its **own** cache line

```
Global: tail ──────────────────────────────► [Node D]
                                                ↑
Queue: [Node A] → [Node B] → [Node C] → [Node D]
        (done)     (done)   (spinning)    (new)
                                └─ spins on own 'locked' field, not global
```

- Improvement: No cache line bouncing during spinning
- Issue: Lock handoff crosses NUMA boundaries in arrival order

NUMA-oblivious vs. NUMA-aware ordering

- FIFO order (NUMA-oblivious)

```
W1 → W2 → W3 → W4 → W5 → W6
N0   N1   N0   N1   N0   N1
 ↑    ↑    ↑    ↑    ↑
 5 cross-node transfers!
```

- Batched by NUMA node (NUMA-aware)

```
W1 → W3 → W5 → W2 → W4 → W6
N0   N0   N0   N1   N1   N1
                ↑
                1 cross-node transfer!
```

Design exercise

- We need to design a lock that batches waiters by NUMA node

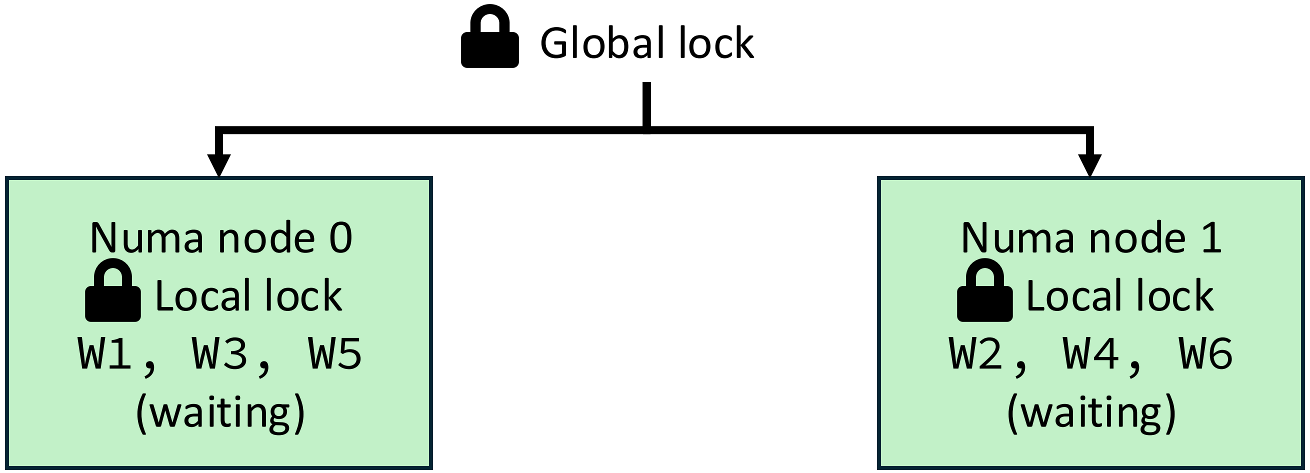
- Constraints:
  - Must eventually serve all waiters (no starvation)
  - Should minimize cross-node transfers
  - Can use multiple lock objects

- Hint: Think hierarchically – what if each node has its own lock?

EPFL

Solution: Cohort locks

- Structure: One global lock + one local lock per NUMA node



- Protocol:
  1. **Acquire:** Get local lock first, then (if first in node) get global lock
  2. **Execute:** Run critical section
  3. **Release:** Pass to the next waiter on **same** node if any exist
  4. **Handoff:** Only release global lock when local queue is empty

A step further with lock design

Even with NUMA-aware locks, critical section data still moves

- Traditional: move data to computation

```
Thread A → Lock → Data (data bounces!)
Thread B → Lock → Data
```

- **Idea:** *Delegation (server-client model)* → move computation to data

```
Thread A ──request──►┐
Thread B ──request──►├──► Server ──► Data
Thread C ──request──►┘   (Node 0)   (never moves!)
```

- Shared data stays in server’s (node 0) L1/L2 cache

False sharing: The anti-pattern

```c
// The bug (looks innocent!)
struct counters {
  long thread_a_count; // 8 bytes
  long thread_b_count; // 8 bytes – SAME cache line!
};
```

- Two threads, two different variables, no locks, performance crashes

```
Cache Line (64 bytes):
┌──────────────────────────────────────────────────────┐
│ thread_a_count │ thread_b_count │ ... unused ...     │
└──────────────────────────────────────────────────────┘
       ↑                  ↑
 Thread A writes    Thread B writes
```

- Each write invalidates the other thread's cache → ping-pong effect

EPFL

False sharing fix: Padding

```
// Bad
struct counters {
  long thread_a_count; // 8 bytes
  long thread_b_count; // 8 bytes – SAME cache line!
};

// Good: padding
struct counters_fixed {
  alignas(64) long thread_a_count; // Own cache line
  alignas(64) long thread_b_count; // Own cache line
};
```

- Spatial locality is a double-edged sword
  - Optimizes single-threaded access, but can kill multi-threaded performance

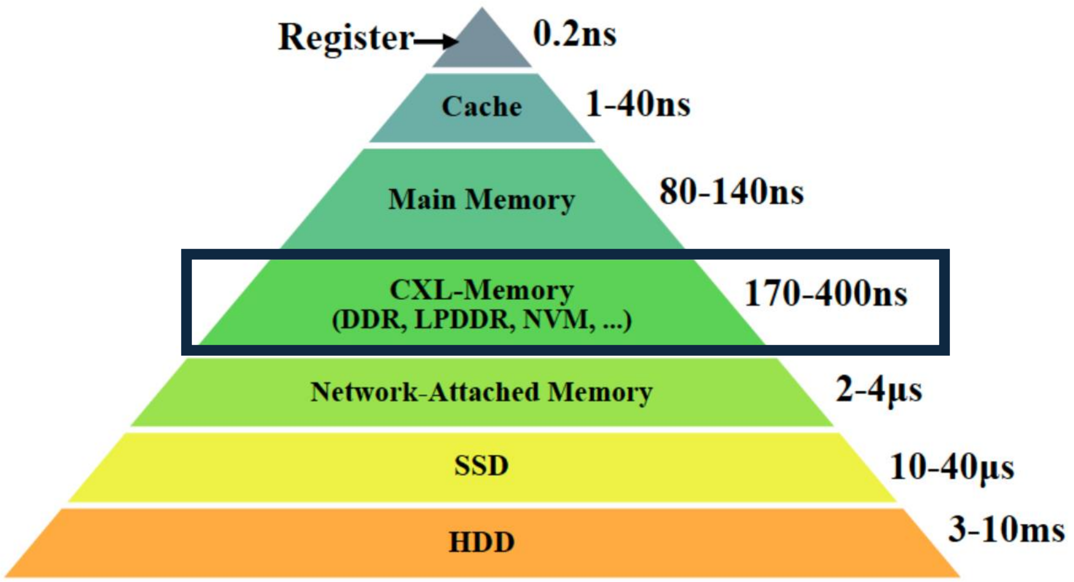
The evolving memory hierarchy

- CXL-memory adds a new tier

Q. How do we manage a memory space that has non-uniform access time?

Software defined-memory tiering

- OS does page promotion/demotion
  - **Scanning:** Figures out hot pages using accessed bits in the page table
  - **Migration:** CXL page is hot → promote to DRAM and demote a cold DRAM page to CXL



Realizing locality at various levels

- From caches to CPU
  - Data structure layout
- From one CPU to another
  - HPC algorithms, synchronization primitives
- From memory to LLC (L3)
  - Graph algorithms, packet processing
- From one NUMA node to another NUMA node
  - Data structures, synchronization primitives (locks)
- From SSD to memory
  - Paging, out-of-core graph processing
- From NIC to memory
  - Far memory, prefetching

Design exercise

- **Locality is everywhere**
- Three types:
  - Temporal
  - Spatial
  - Network
- Locality is applicable across the stack
- **Design for locality:** Before optimizing algorithms, ask: *Where is the data? How often does it move?*