Locality

Sanidhya Kashyap

POCS, Fall 2023

Efficiency matters

Figure 1: The Line, Saudi Arabia

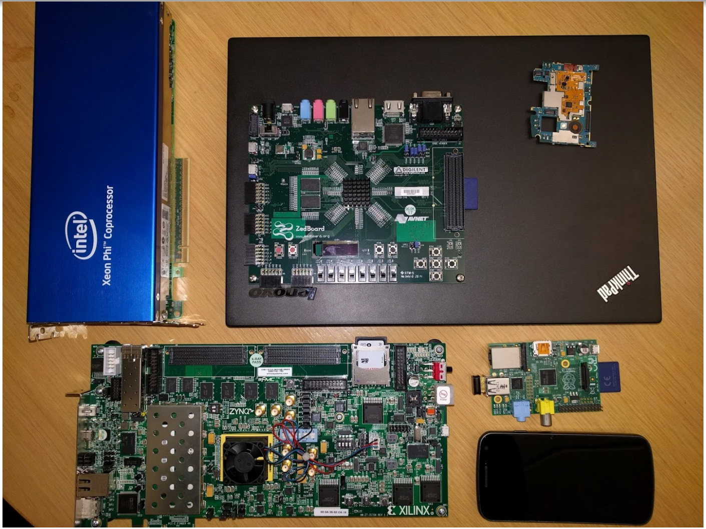


Figure 2: Devices are complicated

Outline

- Principle of locality
- Types of locality
- Approaches that exploit locality
  - Caching
  - Prefetching
  - Partitioning
- Locality examples
  - Data structure layout
  - Locality in locking primitives
  - Locality in NUMA machines

Locality

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
- Modern computers are designed using the principle of locality
  - Caches, predictive loading, faster storage transfer

Efficient data movement is all that matters

- Time/energy cost: moving data
  - One compute unit to a storage unit (CPU $\longleftrightarrow$ memory)
  - One storage unit to another (disk $\longleftrightarrow$ memory)
- Communication links also need spaces
  - Buses, networks are bottlenecks

Efficient data movement is all that matters

- Time/energy cost: moving data
  - One compute unit to a storage unit (CPU $\longleftrightarrow$ memory)
  - One storage unit to another (disk $\longleftrightarrow$ memory)
- Communication links also need spaces
  - Buses, networks are bottlenecks

At the end, we want to minimize data movement or have data ready when we want to work with it

Achieving efficiency becomes complicated

Fundamental limitations exists:

- Packing computation and memory in a limited space
- Shrinking distance among units:
  - Failure of Dennard scaling
  - Cooling is becoming an issue even with 3D chips

An example of complexity: the memory hierarchy

- Time scale for CPU to access data (or data movement latency):
  - L1 access: ~1ns
  - L2 access: ~4ns
  - L3 access (local): ~12-20ns
  - L3 access (remote): ~30-90ns
  - Local DRAM: ~80ns
  - Remote DRAM: ~130-200ns
  - Byte addressable non-volatile memory: ~300ns
  - SSD: ~2-40us
  - Remote machine: ~2us+
  - HDD: ~10ms

An example of complexity: the memory hierarchy

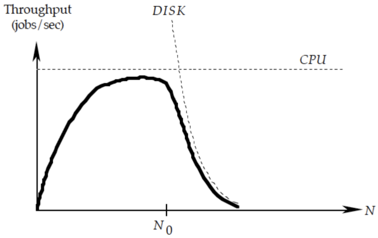
- Time scale for CPU to access data (or data movement latency):
  - L1 access: ~1ns
  - L2 access: ~4ns
  - L3 access (local): ~12-20ns
  - L3 access (remote): ~30-90ns
  - Local DRAM: ~80ns
  - Remote DRAM: ~130-200ns
  - Byte addressable non-volatile memory: ~300ns
  - SSD: ~2-40us
  - Remote machine: ~2us+
  - HDD: ~10ms

How do we ensure that we can keep up with this complexity?

An example from the past: The rise of virtual memory

- Two-level memory hierarchies in the ATLAS computer
  - Main memory + auxiliary storage
- Demand paging
- Backbone of multi-programming

Background: “Paging to death” $\rightarrow$ thrashing



“When it was first observed in the 1960s, thrashing was an unexpected, 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 D. Denning

Working set model

Describes the set of information that a process needs to access in a given period of time to carry out its information.

- Model program's memory behavior over time
- Working set of a program:
  - *Programmer's view*: Smallest collection of information present in main memory to assure efficient execution of a program
  - *System's view*: The set of most recently referenced pages

Working set model

Describes the set of information that a process needs to access in a given period of time to carry out its information.

- Model program's memory behavior over time
- Working set of a program:
  - *Programmer's view*: Smallest collection of information present in main memory to assure efficient execution of a program
  - *System's view*: The set of most recently referenced pages
- The working set is a reflection of the current active locality of reference for a process

Relationship between **working set** and **locality**

- Locality allows the concept of working set to be effective
- Locality determines which resources are required with some degree of accuracy
- Without locality, unable to predict future resource requirements
    - Leads to inefficient systems

Types of locality (from parallel programming)

1. Temporal locality
2. Spatial locality
3. Network locality

Repeatedly access same memory locations over time period

- Frequent access to `sum`'s memory location illustrates *temporal locality*
- Other examples:
  - Function call and recursion
  - Caching data

```c
int sum = 0;
int array[10000];

// Assume array is already filled with values.

for (int i = 0; i < 10000; i++) {
    sum += array[i];
}
```

Figure 3: `sum` access

2. Spatial locality

Access nearby memory locations within a small time frame

- Consecutive memory access of `array`
- Other examples:
    - Sequential vs random access of storage media
    - Accessing memory in a row-by-row fashion

```
int array[1000]; // assume this array is already populated with data
int sum = 0;

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

Figure 4: `array` access

3. Network locality

Access to a memory location nearby is faster than access to a memory location that is farther

- Examples:
  - Caches in CPUs: L1, L2, LLC
  - Multi-socket machines
  - Content delivery networks (CDNs)
- Minimize the latency and bandwidth requirements by minimizing the distance for data access

Locality becomes important for today's machines

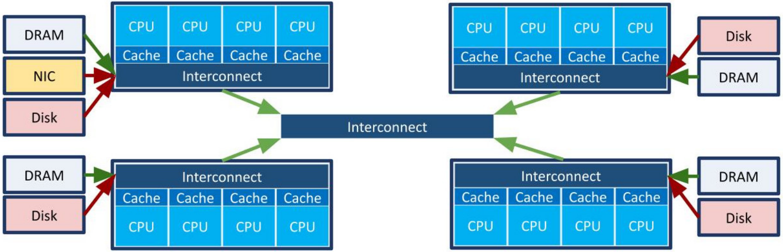


Figure 5: Simplified view of a 4-socket machine

- Accessing data from the local socket is faster than accessing from the remote socket
  - Described as non-uniform memory access (NUMA)

Approaches using locality principle

- Caching
- Prefer sequential access over random access
- Partitioning of data or computation

Use cases: working set, lock algorithms, out-of-core graph algorithms, distributed kv stores

Caching

Keep a working set of data close to the CPU that is used frequently

- Ubiquitous in systems
  - CPU caches
  - MMUs: TLB
  - Networks (edge caches)
  - OS/DB buffers; storage device controller, DRAMs in storage

One form: Sequential access

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: only large blocks can be written
  - DRAM
    - Block addressing and transfer via the bus
    - TLBs (again)
- Examples: write-ahead logging, block nested loop joins

Partitioning

Splitting up the parts of resources and using divide and conquer

- Decomposing an embarrassingly parallel tasks
    - Embarrassing parallel jobs: Do not require any synchronization
        - Can work independently
    - Decompose a large piece of the job, and process them in parallel
    - Ex. Map/reduce

Partitioning

Splitting up the parts of resources and using divide and conquer

- Decomposing an embarrassingly parallel tasks
  - Embarrassing parallel jobs: Do not require any synchronization
    - Can work independently
  - Decompose a large piece of the job, and process them in parallel
  - Ex. Map/reduce
- But they are not applicable everywhere
  - Non-uniform distribution of access in a key-value store
  - Synchronizing tasks

Why locality matters so much?

- Locality starts impacting when the cost to access/modify/move data changes by a huge factor.
- Several scenarios to keep in mind with respect to locality:
  - Minimizing data movement
    - Caching, partitioning for parallel computation and movement
    - Involves either moving computation to data or moving data to the computation unit
  - Data layout for efficient fetching of data
    - Sequential vs random
  - Overlapping computation and data movement
    - Prefetching

Examples in detail

1. Data structure layout
2. Locking primitives minimizing data movement
3. NUMA: Data structure replication and partitioning

Data layout

- When accessing memory, CPU accesses data in a way that impacts application's performance
- Two data structures as an example:
  - Arrays
  - Tree data structure

Arrays

- Matching storage layout with the looping order of algorithms
  - Sequential vs random access
  - Example: Matrix

- Stored as A11, A12, . . . , A1n, A21, A22, A2n, . . . , Amn
- Loop: for i in 1 . . . n { for j in 1 .. m { Aij . . . }} efficient
- Loop: for j in 1 . . . m { for i in 1 .. n { Aij . . . }} inefficient

A11	A21	...	Am1
A12	A22	...	Am2
...	...	...	...
A1n	A2n	...	Amn

- Align storage layout with use cases if possible
  - Loop reordering in compilers
  - Sorting

Locality with respect to locks

- Locks are the basic building blocks for concurrent systems
- Locks:
  - Provide mutually exclusive access to shared data
  - Order waiters accessing the critical section >
- Lock algorithms try to minimize the movement of shared data

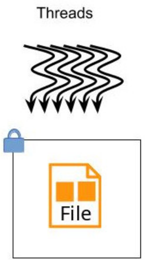


Figure 6: Threads going to access a file protected by a lock

Spin locks basic behavior

- Waiters wait for their turn
- Locks serialize the access: *Introduce sequential bottleneck*

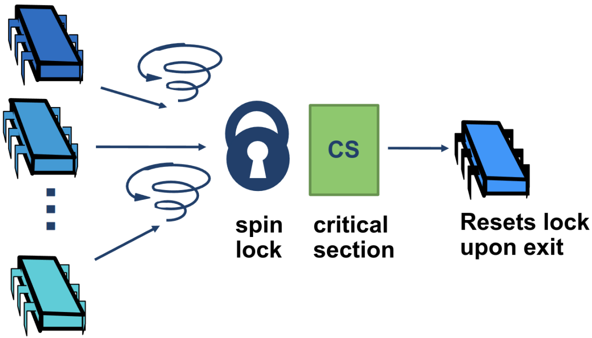


Figure 7: Basic spinlock (taken from Art of Multiprocessor Programming)

Locks first try to minimize contention

- Contention: Threads writing to the same cache line (shared data)
- Hardware maintains a consistent state of the shared data using the coherence protocol

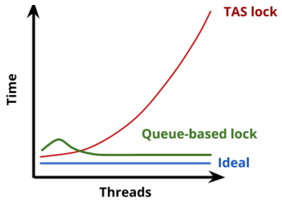


Figure 8: Lock latency

- TAS broadcasts to everyone of the lock situation
  - Saturates memory bandwidth (different from locality)
- Queue lock: Maintains a queue of waiters and notify next in line without bothering others
  - Minimizes shared data contention (cache line)

- Let's consider a NUMA machine
  - Accessing the local socket is faster than remote socket

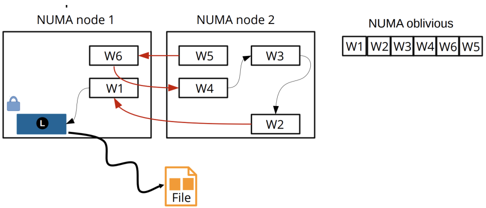


Figure 9: Accessing in non-NUMA fashion

- Let's consider a NUMA machine
  - Accessing the local socket is faster than remote socket

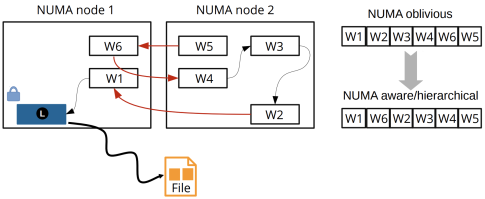


Figure 10: Accessing in NUMA fashion

- Group lock waiters from one socket, process them, and then pass to another socket

NUMA-aware lock

- Comprises of multiple locks (n+1)
  - A global lock
  - NUMA node lock on each node

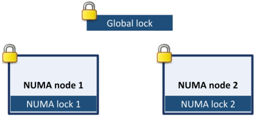
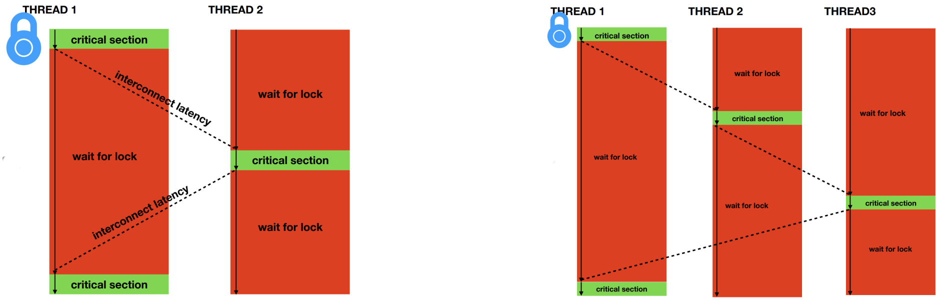


Figure 11: Cohort lock

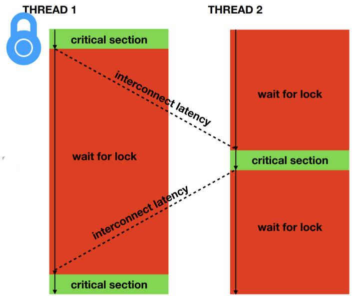
- Acquire: First acquire the local node lock, then acquire the global lock
- Release: First release the global lock, then release the local lock
- Maintain locality of data: minimize cache-line bouncing
  - Passes the lock within the same socket multiple times before releasing the global lock

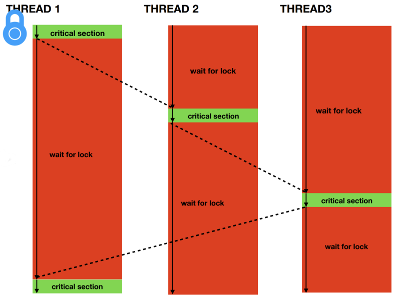
Need to localize shared data



- Critical section data is transferred for each lock acquire
- The wait for lock increases with increasing thread count

Need to localize shared data





- Critical section data is transferred for each lock acquire
- The wait for lock increases with increasing thread count

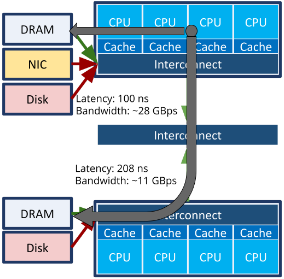
Q. How can we localize shared data?

Put the shared data on one core

- Locality: Keep all shared data on one core
- Use a server client model
  - Clients send request to server (encode their critical section function)
  - Server processes request on client's behalf
- Shared data is ALWAYS accessed by one core!

Data placement in NUMA machines

- Goal: Keep application's data close to the computation
  - Latency is problematic for memory sensitive applications
  - Bandwidth is an issue for memory intensive applications



- Allocate memory using first touch or interleaved policy
  - First touch: allocating from the local node first
  - Interleaved: Allocate memory using round robin
- Use page migration during application execution (AutoNUMA)

Realizing locality at various levels

- From caches to CPU
  - Ex: data structure layout: arrays vs linked list
- From one CPU to another
  - HPC algorithms, synchronization primitives
- From memory to LLC
  - Ex: graph algorithms, packet processing
- From one NUMA domain to another NUMA domain
  - Ex: data structures, synchronization primitives (locks)
- From SSD to memory
  - Ex: Paging, out-of-core graph processing applications
- From NIC to memory:
  - Ex: Remote memory, paging

Summary

- Locality is one of the most important principles
  - Started from virtual memory; now applicable everywhere
- Three types of locality: temporal, spatial, network
- Locality is applicable across the whole stack