Locality

Sanidhya Kashyap

EPFL, Fall 2022

Outline

- Locality principle
- Types of locality
- Principles to achieve locality
    - Caching
    - Prefetching
    - Partitioning
- Locality examples
    - Data structure layout
    - Locality in locking primitives
    - Locality in NUMA machines

We have been building things for efficiency

Figure 1: Devices are complicated

Building things for efficiency, but . . .

- Fundamental limitation exists in packing computation and memory in a limited space
- Similar analogies exists in real world
  - Cities vs towns vs villages
- Computing: Fundamental limits exists in shrinking distances:
  - Quantum effects are already taking over
  - 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: ~2ns
  - L2 access: ~8ns
  - 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: ~2ns
  - L2 access: ~8ns
  - 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?

It is about efficient data access

- Almost all (time/energy) cost is due to moving data over a distance
  - Moving data to the CPU
  - Moving data through the CPU
- Communication links also need spaces: Buses, networks are bottlenecks
- In several applications, CPUs mostly wait for data to arrive
  - Solutions exists: prefetching, cache hierarchy etc. . .

It is about efficient data access

- Almost all (time/energy) cost is due to moving data over a distance
  - Moving data to the CPU
  - Moving data through the CPU
- Communication links also need spaces: Buses, networks are bottlenecks
- In several applications, CPUs mostly wait for data to arrive
  - Solutions exists: prefetching, cache hierarchy etc...
- *At the end, we want to minimize data movement or have data ready when we want to work with it*

Locality is a notion that relates systems (software) with the physical world. In computing, "it is about the patterns of programs referencing their data."

- A program accesses a **predictable** sequence of memory addresses.

Working set for locality

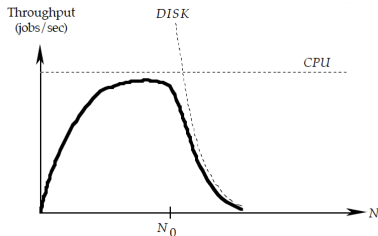
Locality is about the patterns of programs referencing their pages. Working set is about detecting those patterns in real time and using them to make memory management decisions.

- Working set measures the intrinsic memory demands of individual programs

The rise of virtual memory

- In the 50s ~60s
- System: ATLAS computer
- Two-level memory hierarchies:
  - Main memory + auxiliary storage
- Demand paging
- Backbone of multi-programming

Background: 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

- A unified approach to tackle the resource allocation problem: process scheduling and memory management
- Intended to model the behavior of programs, specifically the program's memory demand
- 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

- A unified approach to tackle the resource allocation problem: process scheduling and memory management
- Intended to model the behavior of programs, specifically the program's memory demand
- 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 is applicable to programs at every level!**

Types of locality (from parallel programming)

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

1. Temporal locality

Access same memory location several times within a small time window

- Given a memory trace, a memory location occurs multiples of times closely
- Constants and memory locations in loops; *hot functions*

- Every iteration of the innermost loop for given out loop indices (i,j) accesses the same memory C[i][j]
- Access to matrix B has minimal temporal locality, as success accesses to B are separated by $\mathcal{O}(N^2)$

```
for (i = 0;  i < N;  i++)
 for (j = 0; j < N; j++)
  for (k = 0; k < N; k++)
    C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

Figure 2: N*N multiplication

2. Spatial locality

Access nearby memory locations within a small time frame

- Reading contents of a file
- Accessing memory in a row-by-row fashion

- Both A and C are accessing memory in a row order and have spatial locality
- B does it in at the level of columns and does not have spatial locality

```
for (i = 0;  i  <  N ;  i++)
 for  (j = 0 ;  j < N ;  j++)
  for  (k = 0 ;  k < N ;  k++)
     C[i][j] = C[i][j] + A[i][k] * B[k][j] ;
```

Figure 3: N*N multiplication

3. Network locality

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

- CPU has faster access to memory locations mapped locally than to memory locations mapped to other CPUs
- Different access time exists among multiple CPUs
  - Computers are a set of distributed nodes
  - Each node communicate their own communication approaches
  - CPU to CPU can have varying latency in accessing memory location
- Also called **non-uniform memory access (NUMA)**
- Example: NUMA-aware system design, Remote memory, etc.

Locality becomes important for today's machines

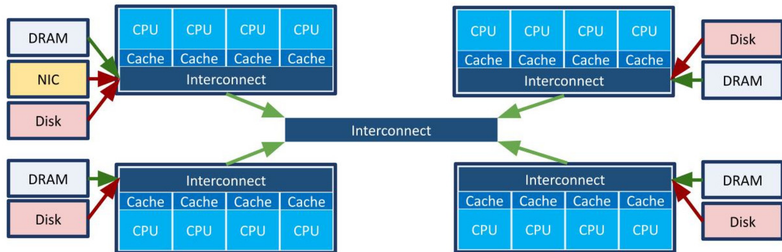


Figure 4: Simplified view of a 4-socket machine

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, out-of-core graph processing
  - Ex: Paging
- From NIC to memory:
  - Ex: Remote memory, paging

Locality principles

- Caching
  - Keep a working set of data close to the CPU that is used frequently
- Prefer sequential access over random access
  - Based on device characteristic, sequential read/write is faster than by random access
  - Eg: Prefetching (branch prediction)
- Partitioning of data or computation
  - Splitting up the parts of resources and using divide and conquer

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

Caching

- Ubiquitous in systems
  - CPU caches
  - MMUs: TLB
  - Networks (edge caches)
  - OS/DB buffers; storage device controller, DRAMs in storage
- Prefetching: Fill caches in a speculative manner, assume sequential access
- CPU branch prediction, buffer/page cache in the OS, DRAM on storage controller

One form: Sequential access

- Sequential access is faster than random access
- Comes from the physical notion
  - 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, only very ones erased
  - DRAM
    - Block addressing and transfer via the bus
    - TLBs (again)
- Examples: write-ahead logging, block nested loop joins

Partitioning

- Decomposing and embarrassingly parallel tasks
  - Embarrassing parallel jobs are the ones that do not require any synchronization, i.e., can work independently.
  - Decompose a large piece of the job, and process them in parallel.
  - Ex. Map/reduce

Partitioning

- Decomposing and embarrassingly parallel tasks
  - Embarrassing parallel jobs are the ones that do not require any synchronization, i.e., 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, the way CPU accesses data 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, nesting, co-clustering

Array: Row vs column representation

- Row representation: `struct { int a, int b }[]`
- Column representation: `struct { int a[], int b[] }`
- Column representation is usually much more efficient than row
  - Fewer objects are created ($\mathcal{O}(1)$ vs. $\mathcal{O}(array)$)

B-tree

- Balanced binary tree: twice as many nodes as in the above level
  - Tree grows exponentially fast as we go down
- Impossible to store data in linear memory to maintain the proximity of parent-child pairs
  - There is always be a non-local pairs going in particular direction

Q. What we can do in this case?

B-tree

- Balanced binary tree: twice as many nodes as in the above level
  - Tree grows exponentially fast as we go down
- Impossible to store data in linear memory to maintain the proximity of parent-child pairs
  - There is always be a non-local pairs going in particular direction

Q. What we can do in this case?

- We can keep siblings local: bread-first enumeration or even depth-first enumeration
  - There will be locality since the size of leaf levels dominates
    - Basis of index

Locality wrt 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

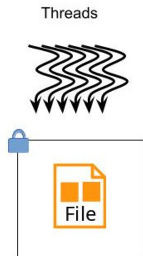


Figure 5: Threads going to access a file protected by a lock

Spin locks basic behavior

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

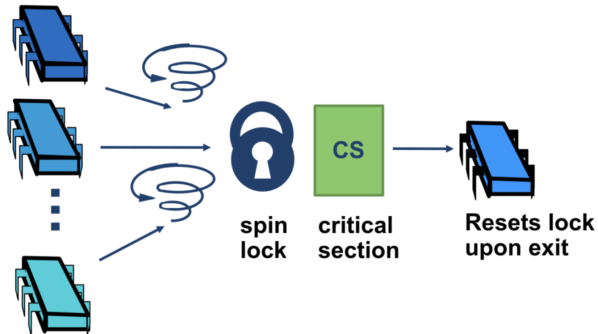


Figure 6: Basic spinlock (taken from Art of Multiprocessor Programming)

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

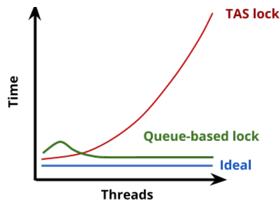


Figure 7: Lock latency

- 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)

Locality in locks

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

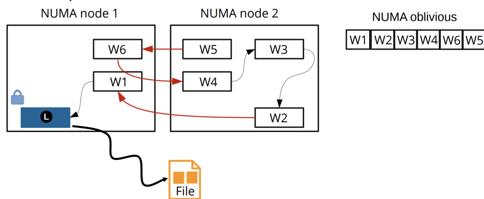


Figure 8: Accessing in non-NUMA fashion

Locality in locks . . .

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

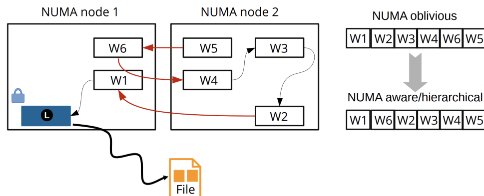


Figure 9: Accessing in NUMA fashion

- 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

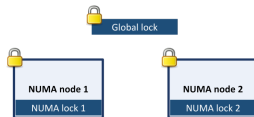
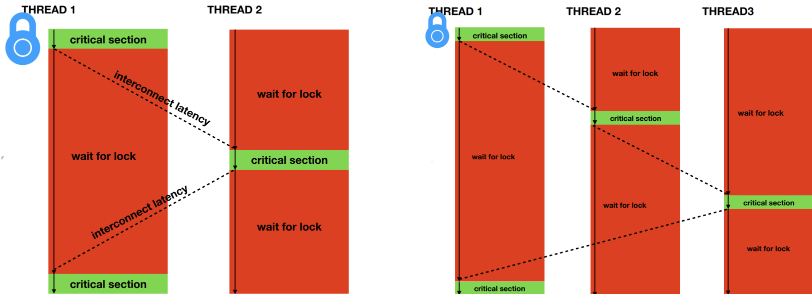


Figure 10: Cohort lock

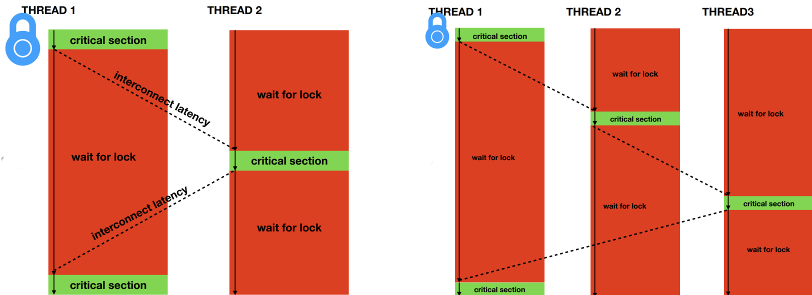
- Acquire: First acquire the local node lock, then acquire the global lock
- Release: First release the global lock, then release the local lock
- Maintain locality of data: minimize cache-line bouncing
  - Passes the lock within the same socket multiple times before releasing the global lock

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

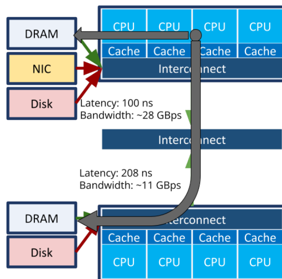
Q. How can we localize shared data?

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)

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