Data Locality: Computer Hardware and the Memory Hierarchy

Christoph Koch

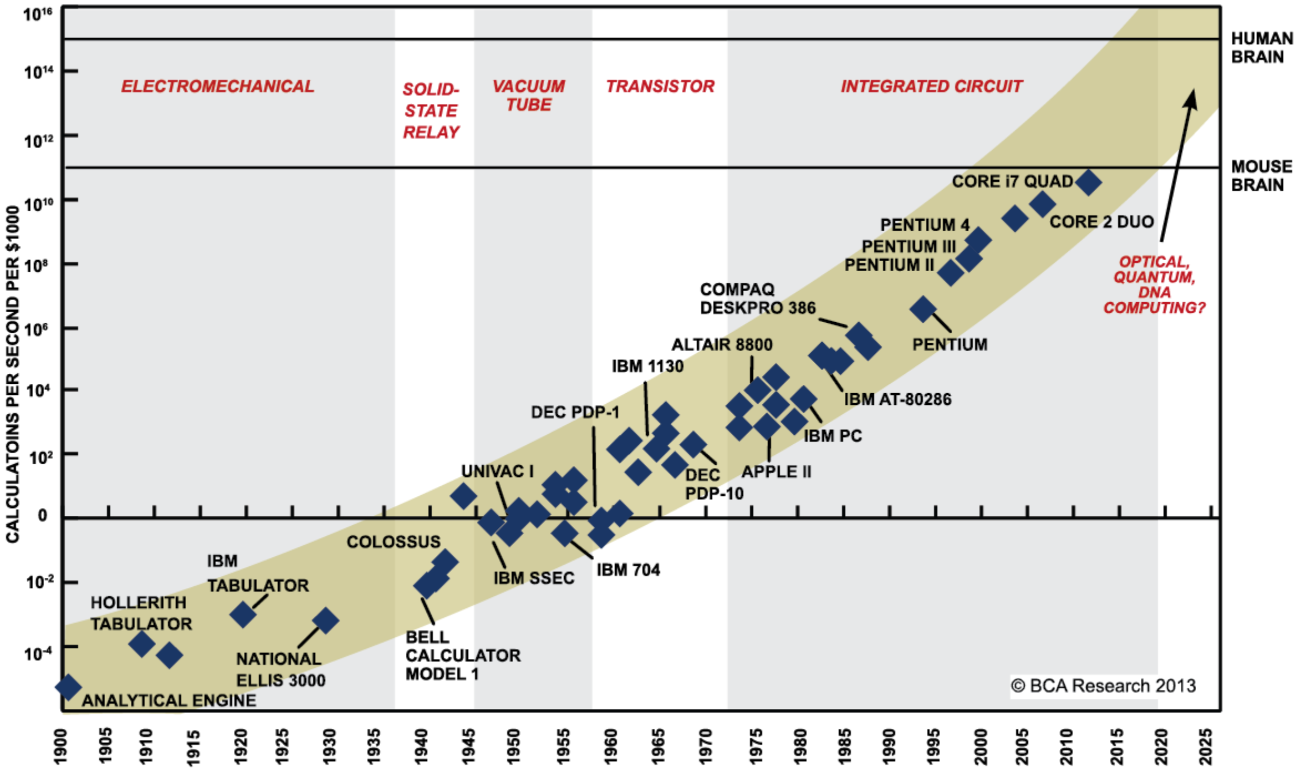
School of Computer & Communication Sciences, EPFL

Goals of this lecture

- Moore's law is failing:
  - *Computers don't get faster anymore.*
  - *We need to understand parallel computing, and its limitations.*
  - *We need to optimize our computations for locality.*
- Understanding data locality; principles of leveraging and maximizing locality.
- The memory hierarchy; caches.

Moore's Law through the Classic Era

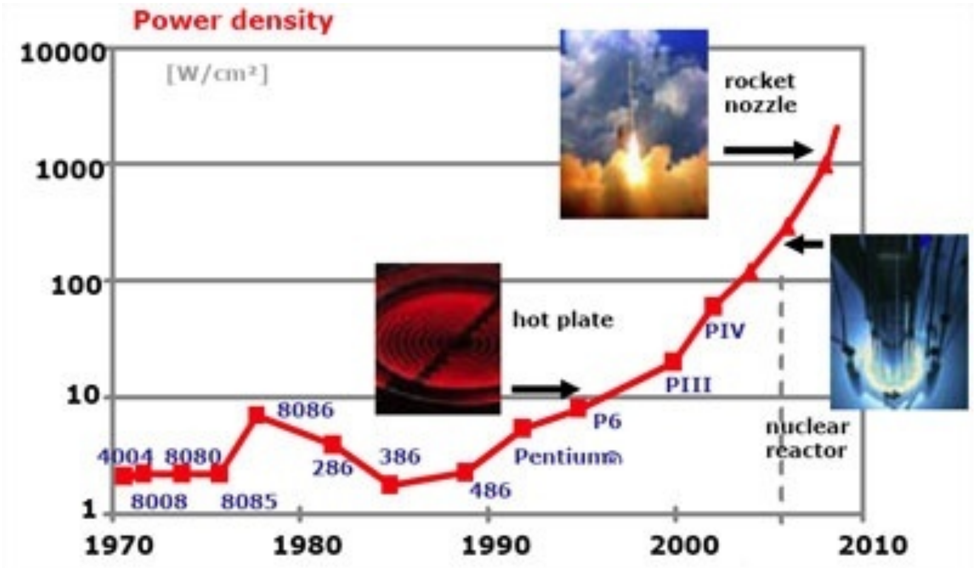
Moore's Law: #transistors achievable per mm^2 doubles every ~18 months.



SOURCE: RAY KURZWEIL, "THE SINGULARITY IS NEAR: WHEN HUMANS TRANSCEND BIOLOGY", P.67, *THE VIKING PRESS*, 2006. DATAPOINTS BETWEEN 2000 AND 2012 REPRESENT BCA ESTIMATES.

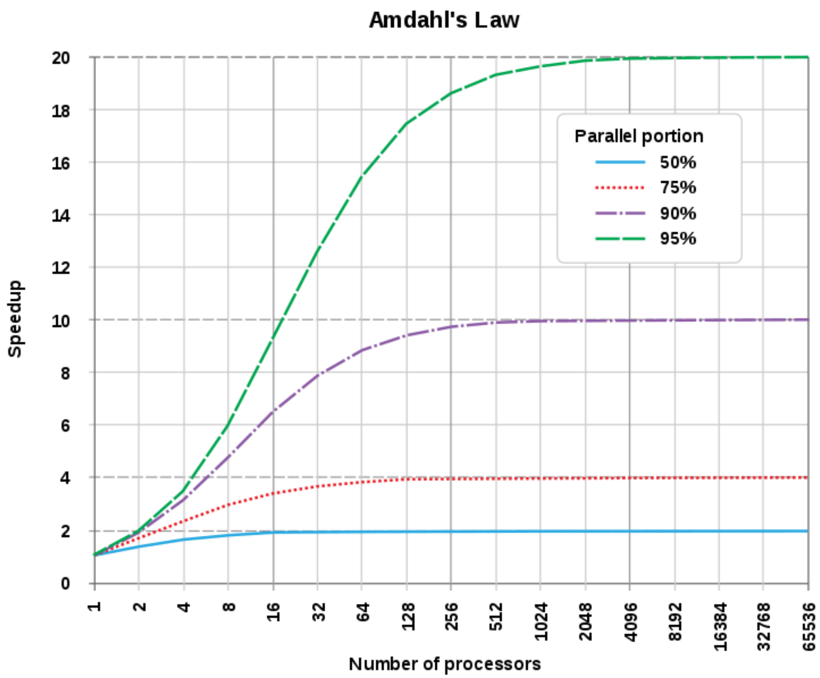
Dennard Scaling has failed

- Dennard scaling: As integration increases (transistor size decreases) by 2x, required voltage decreases by 4x, so power density remains constant.
- Dennard scaling has failed
  - *Due to quantum effects*
- Consequence: we can't shrink logic/ increase clock rates anymore and still cool the chips.



Amdahl’s law

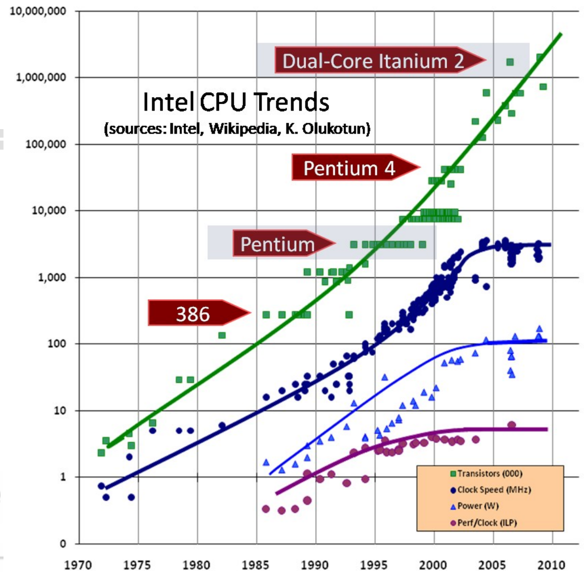
The benefits of parallelization are limited due to the latency caused by the inherently sequential part of the computation.

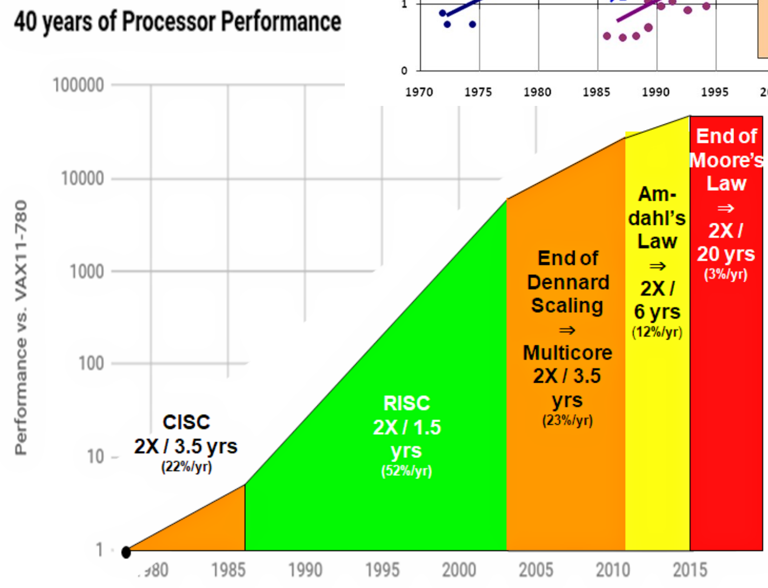


Moore’s Law is Failing

- Failure of Dennard Scaling
- Amdahl’s law – parallelization has its limits
- Can’t communicate a bit/signal with less than one electron (or anything approachi it) – can’t shrink transistors further after ~2030.

- Consequence: computation becomes less local.





Locality

Locality relates (software) systems with the physical world.

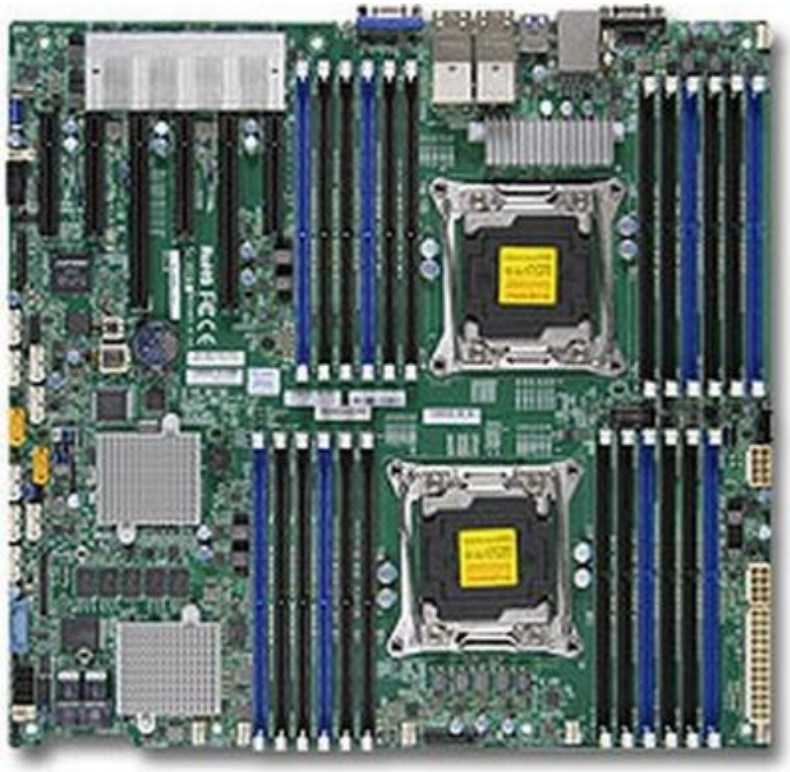
- We can fundamentally only pack so much memory and computing power into a limited space.
- As data and our computing challenges grow, we get into locality problems.

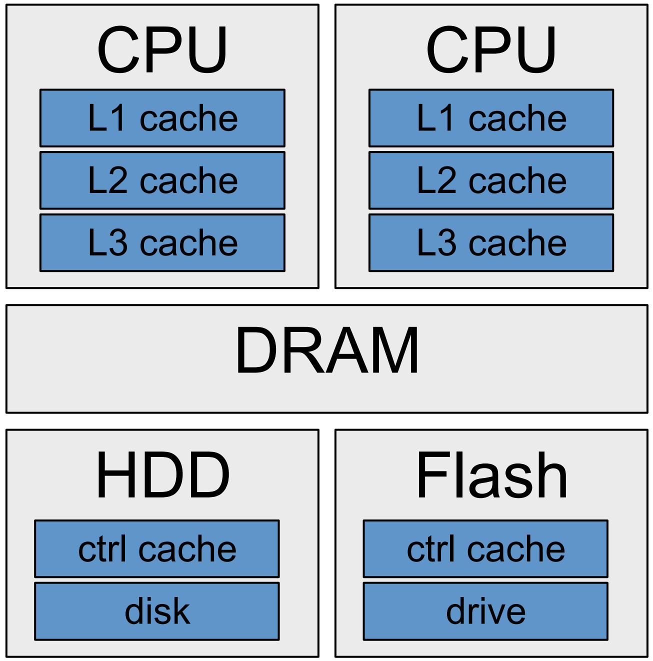
- We can reduce pretty much any performance concern to a locality issue.

The role of locality

- In computing, essentially all (time/energy) cost is due to moving data over a distance.
  - *Moving data into the CPU*
  - *Moving data through the CPU*
- Fundamental limits to shrinking distances:
  - *Quantum effects soon to take over (one cannot work with less than an electron; also noise)*
  - *Failure of Dennard scaling.*
  - *We have only 2/3 dimensions to pack stuff into space. (3d chips, but also consider cooling!)*
- Communication links need space: Buses, networks are bottlenecks
- In many (most?) applications, CPUs are mainly waiting for data to arrive.
  - *Not just explicit data management applications.*
  - *Cache hierarchies, prefetching, …*

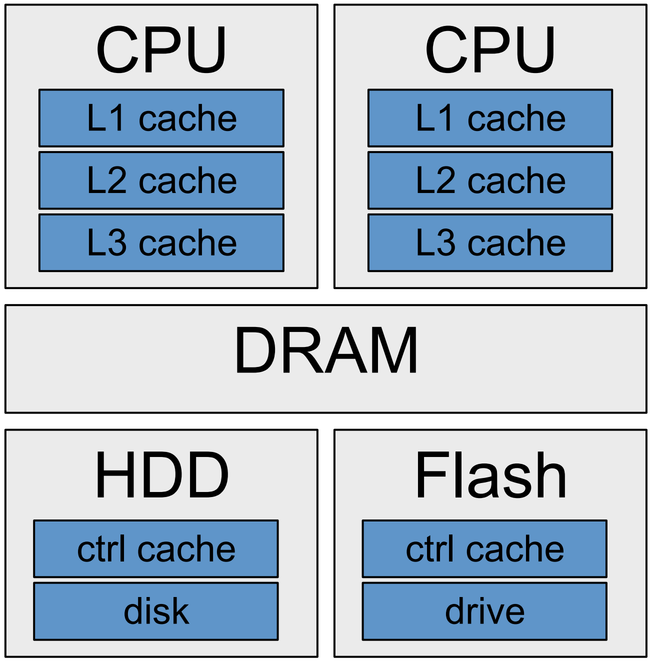
Nodes, sockets, CPUs, RAM, I/O etc.





Memory hierarchies; time scales

- Core i7 Xeon 5500 Series Data Access Latency (approximate)
  - *L1 CACHE hit, ~4 cycles*
  - *L2 CACHE hit, ~10 cycles*
  - *L3 CACHE hit, line unshared ~40 cycles*
  - *L3 CACHE hit, shared line in another core ~65 cycles*
  - *L3 CACHE hit, modified in another core ~75 cycles*
  - *remote L3 CACHE ~100-300 cycles (@~ 0.3ns/cycle)*
- Local DRAM ~60 ns
- Remote DRAM ~100 ns
- Accessing a hard drive – 10ms(seek)
  - *0.1ms/page (transfer)*
- Accessing a tape – minutes (seek)
- L1 to DRAM: 10^2x; DRAM to HD: 10^5x slowdown



Caches

- The further away a cache is from the core, the larger, slower, and cheaper (due to different technologies used) it is.
- While Moore's law was active, the speed of computation (and register access) was diverging from the speed of RAM – more and more cache layers were inserted.
  - *When does it make sense to introduce another layer of cache?*

Units of computation

- Nodes, sockets (CPUs), and cores. Why so many abstractions?
- Data locality considerations!
- Peers of these three types communicate using very different technologies, which have vastly different latencies. The argument for having all of these is the same as for memory hierarchies!

Data center servers vs. supercomputers

- Supercomputers have higher compute/network density, not just by volume/watt but also per $. Network fabrics are quite different from those in data centers; there is much higher connectedness.
- Data center servers support better data locality for large data volumes; they have more RAM per node and usually have their own secondary storage devices.
- Use case: Matrix computations.
  - *Supercomputers better for dense linear algebra*
  - *Data center servers often more cost-effective for certain sparse linear algebra computations and tasks (such as data cleaning and transformation) where the amount of data moved is high relative to the intensity of computation.*

Locality Principles

- Caching
  - *Keep a working set of data that is used frequently close.*

Prefetching

- Prefer sequential over random access
  - *Physics governs that some data is better read/written sequentially than by random access.*
- Partitioning
  - *Some tasks allow to consider data in parts: either use all resources to process a part at a time, or work on the partitions in parallel.*
- Use cases: Out-of-core algorithms: joins, sorting (of data on hard drives)

Caching

- Ubiquituous in systems
  - *CPU caches*
  - *MMUs: TLB*
  - *Networks (edge caches)*
  - *OS/DBMS buffers; storage device controller caches (hard drives, solid state, …)*
  - *DBMS: materialized views*
- Caching is frequently (erroneously) equated with locality. [Salzer and Kaashoek]
- Prefetching: Speculative filling of cache – usually assumes sequential access
  - *CPU branch prediction, storage device controllers – what else could be done?*

Sequential access

- Sequential access faster than random access
  - *Hard drives*
    - *Mechanically moving parts: seek time >> transfer time*
    - *Reading a byte is not cheaper than reading a page*
  - *Flash/solid state: only large blocks can be written, only very large ones erased.*
  - *DRAM*
    - *Block addressing and transfer via the bus*
    - *TLBs (again)*
- Examples: Block nested loops join, external-memory sort

Partitioning

- Decomposability and embarrassing parallelism.
  - *When can different parts of the data be processed completely independently? (No communication needed)*
  - *Map/reduce*
- But: not applicable everywhere.
  - *Frequent: the graph of our data (dependencies, relationships) has small diameter even though sparse ("small world phenomenon")*
  - *Such graphs have no small cuts (see next video)*
- Out-of-core algorithm examples: External memory sort, GRACE hash join

Locality and data structure(s)

Christoph Koch

School of Computer & Communication Sciences, EPFL

Data Structures: Arrays

- Does the storage layout match the looping order of the algorithms that access the array? – sequential access vs random access.
- Example: Matrix

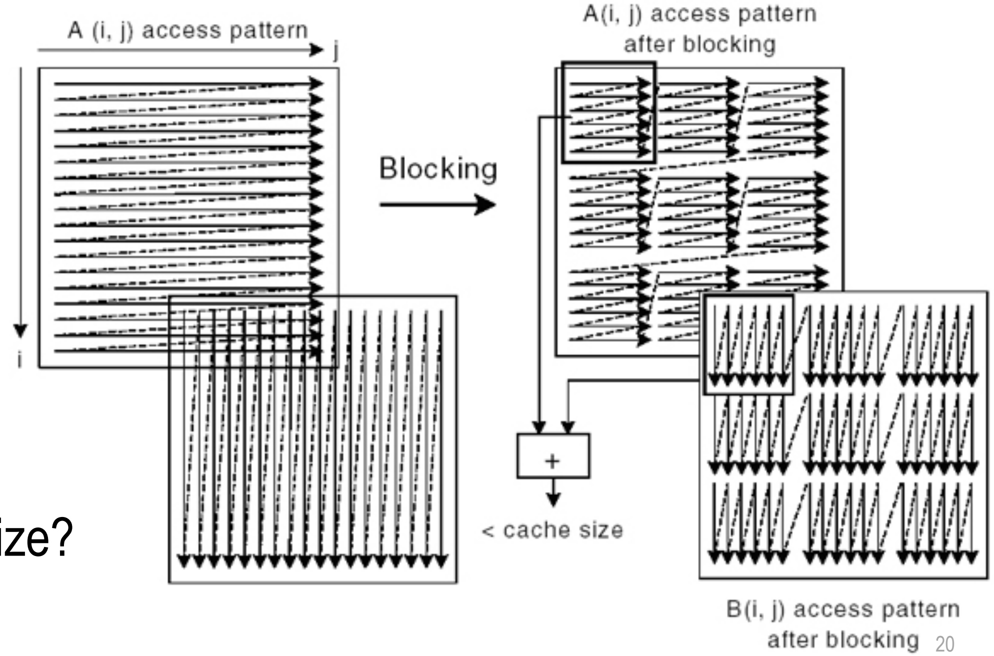
A11	A21	…	Am1
A12	A22	…	Am2
…	…		…
A1n	A2n	…	Amn

- 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 (due to block addressing, prefetching, cache misses)
- Align storage layout with use cases (pattern of access) if possible, or vice versa.
  - *Loop reordering in compilers.*
  - *Sorting, nesting, co-clustering in DBMS.*

Loop tiling

- Assume (2d-)array to be processed is larger than cache:
- Reorder loops to process large array by small array regions (tiles).

```
for (i=0; i< MAX; i++) {
  for (j=0; j< MAX; j++) {
    A[i,j] = A[i,j] + B[j, i];
  }
}
```



New loop structure? Tile size vs. cache size?

(source: intel)

Arrays/Relations: Row vs. columnar representation

- (OLAP) databases: Row vs column-stores
  - *Many queries use only some columns of a relational table. Only fetch the data from disk that you need.*
  - *Better on-disk compression of columns*
    - *E.g. many consecutively stored phone numbers compress better than random data.*
  - *Lots of hype about this!*
- OO PLs, e.g. Java (VM)
  - *row representation: array of struct { int a, int b }*
  - *column representation: struct {int a[], int b[] }*
  - *column representation much more efficient:*
  - *Much fewer objects created (O(1) vs O(array size)). Boxing/unboxing overhead!*

Sales

Product	Customer	Date	Sale
Beer	Thomas	2011-11-25	2 GBP
Beer	Thomas	2011-11-25	2 GBP
Vodka	Thomas	2011-11-25	10 GBP
Whiskey	Christian	2011-11-25	5 GBP
Whiskey	Christian	2011-11-25	5 GBP
Vodka	Alexei	2011-11-25	10 GBP
Vodka	Alexei	2011-11-25	10 GBP

Product

ID	Value
1	Beer
2	Beer
3	Vodka
4	Whiskey
5	Whiskey
6	Vodka
7	Vodka

Customer

ID	Customer
1	Thomas
2	Thomas
3	Thomas
4	Christian
5	Christian
6	Alexei
7	Alexei

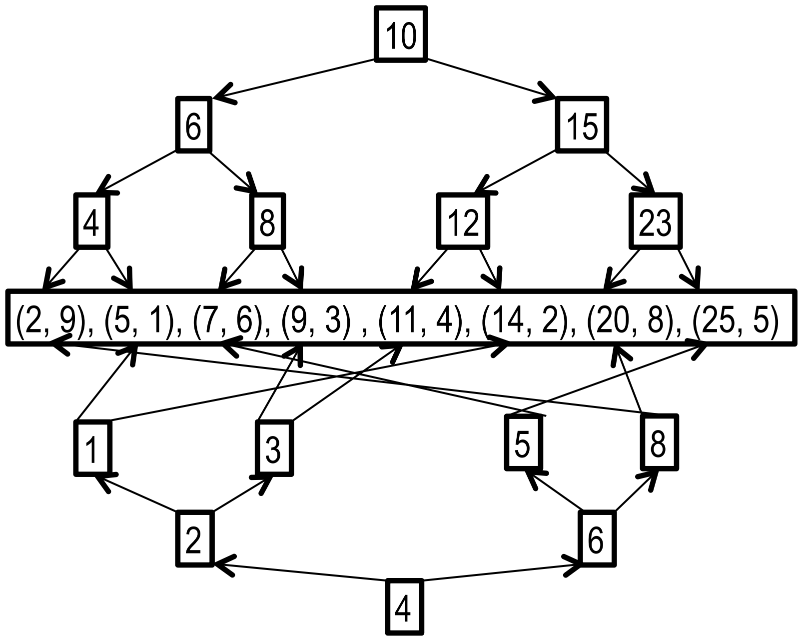
Data Structures: Trees

- Example: Balanced binary tree.
- On each level, twice as many nodes as in the level above.
  - *Exponential growth.*
- There is NO WAY to store data in linear (or finite-dimensional) memory so that parent-child pairs remain close!

- But: can keep siblings local: breadth-first enumeration.
- Or: leaf level size dominates: leaf level of depth-first enumeration is essentially local.
  - *Basis of indexing!*

Tree indexes

- The leaf level of a balanced tree is (essentially) local in reasonable representations.
- Idea of primary (B-)tree indexes in DBMS: leaf level is aligned with sort order in data file.
- Range lookup.
  - *Find first matching element using index, then scan data file sequential until range condition becomes false.*
  - *But: no two different tree indexes can index locally into shared data! (Secondary indexes)*
  - *Thus indexes are not as effective as often naively assumed.*
- Even prim. index is not very useful: same log-many random accesses as in binary search on data file.



Types of graphs

- Graphs with edge relations that have a local representation
  - *Simple, special deterministic constructions: chains, trees – not interesting here.*
- Graphs with (relatively) small cuts (partitioning/parallelization!)
  - *Resource-constrained graphs:*
    - *(Almost) planar embeddings into a 2d surface (map)*
    - *Constraints on density of edges crossing any line on the map.*
    - *Not trees, but relatively low degree of cyclicity. Few (redundant) nonlocal links*
  - *Road networks – roads take space, too many is pointless.*
  - *Physical internetworks – deal with line cost, routing complexity.*
  - *The brain (in theory)*
- All other graphs: locality nightmare.
  - *Internet communication patterns, social networks, the brain (in practice), …*

Types of graphs

- Random graphs: edges added randomly
  - *Above edge-to-node ratio 1, essentially all of the graph is connected (monster connected component)*
  - *Have low degrees of separation (small world phenomenon) and NO SMALL CUTS already when sparse (linear edge-to-node ratio)*
- Real-world graphs and networks
  - *Exclude resource-constrained graphs, e.g. road networks, physical internetworks*
  - *Essentially all other graphs/networks: internet communication patterns, social networks, the brain, …*

Real-world graphs

These notions are essentially equivalent/interchangeable:

- Power law graphs: random graphs in which #neighbors follows a power law.
- Social networks: “rich get richer” phenomenon: popular nodes are more likely to get further connections.
- Small world graphs: “k degrees of separation”
  - *Example construction (Kleinberg): Take grid (fishing net) and add a certain ratio of non-local shortcuts.*

- Already for sparse graphs
- Deterministic construction: expander graphs

Data Structures: Graphs and Networks

- (Non-resource bounded) real graphs have no small cuts:
  - *Take a graph. There is no partition of the nodes into two about equally sized sets such that the #edges crossing the partition is less than linear.*
  - *CANNOT be partitioned effectively to handle regions independently without requiring lots of "communication" between regions.*
  - *Essentially impossible to parallelize graph analytics effectively.*
  - *Everyone still does it. Horrible performance (Pregel, Giraffe) – every node has to talk to every node in every step.*
    - *A worst-case scenario from the locality perspective.*
- But: small-world phenomenon
  - *There are short paths between any two nodes (routing!)*
  - *Does not mean communication is spatially local, but is local if you only count hops!*
  - *Theory of weak ties (sociology) vs. routing heuristics!*

Graphs with small cuts [not relevant to the final exam]

Definition: A graph G=(V,E) has tree-width (<=)k if there is a set H (the hypergraph/tree decomposition of G) of subsets of V such that,

- for each S in H, |S| <= k+1 and
- for each edge {v1, v2} in E, there exists at least one S in H such that v1, v2 in S (H covers E).

- Test if graph has tree-width k: Bodlaender's algorithm (linear in |E| if k is fixed)
- Compute tree-width of graph: NP-complete.
- NP-complete problems: fix tree-width of combinatorial structure => in P.
- Traverse tree decomposition of graph is often the best-known technique for processing hard combinatorial problems.

Classroom exercise: Distributed All-Pairs Shortest Paths

- Floyd-Warshall:
  sp(i, j, 0) := w(i, j)
  sp(i, j, k) := min(sp(i, j, k-1), sp(i, k, k-1) + sp(k, j, k-1))

- On a road network

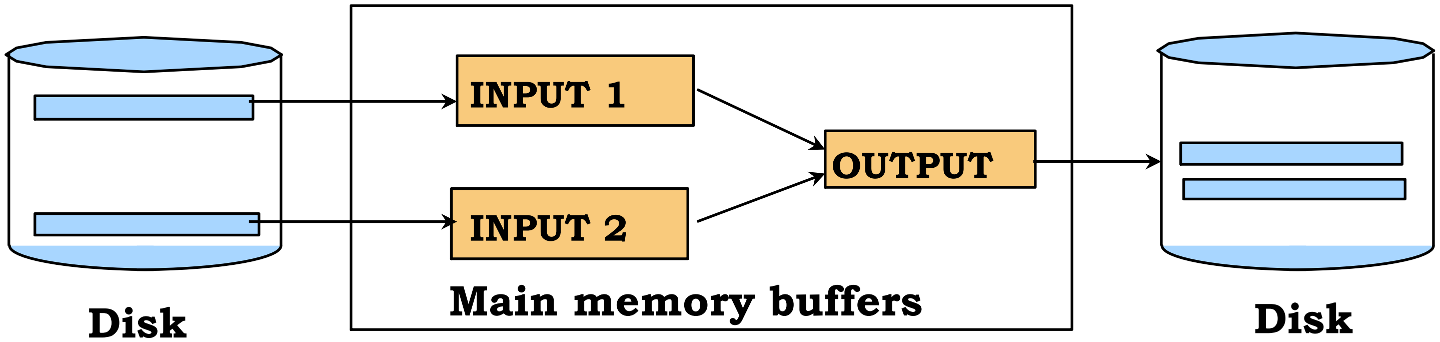
Out-of-core algorithms: external sorting

Christoph Koch

School of Computer & Communication Sciences, EPFL

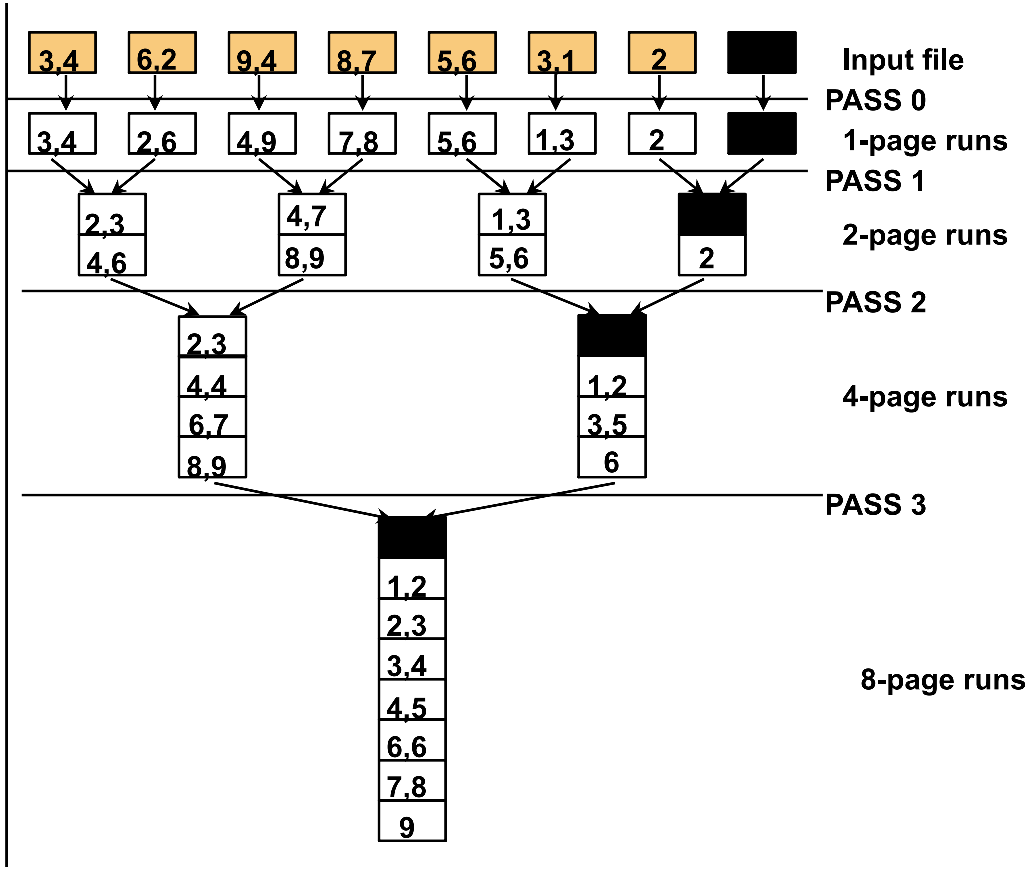
2-Way Sort: Requires 3 Buffers

- Pass 1: Read a page, sort it, write it.
  - *only one buffer page is used*
- Pass 2, 3, …, etc.:
  - *three buffer pages used.*



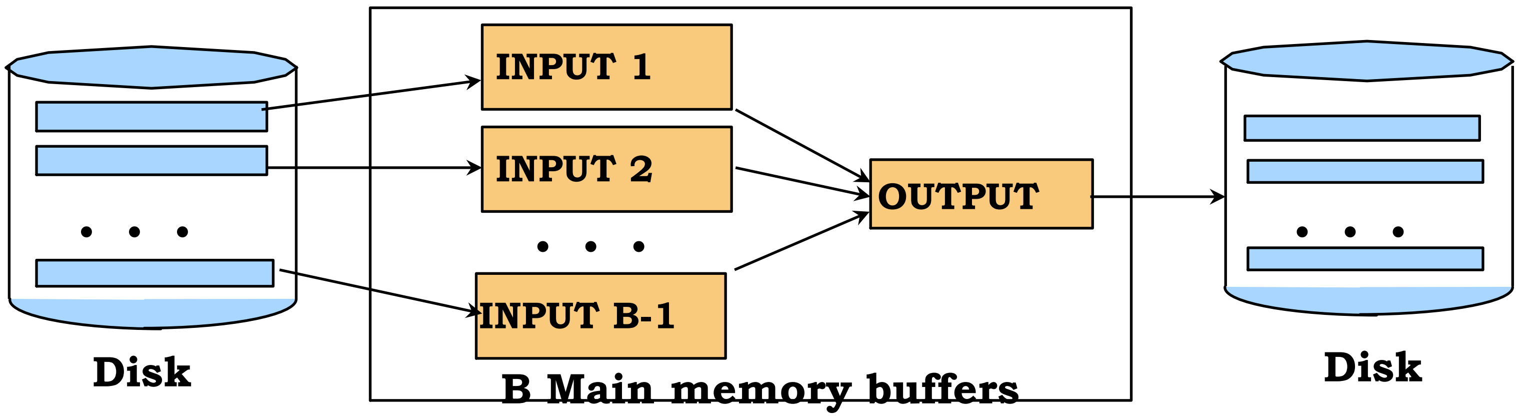
Two-Way External Merge Sort

- In each pass we read + write each page in file.
- N pages in the file => the number of passes $= \lceil \log_2 N \rceil + 1$
- So the total cost is $2N\left(\lceil \log_2 N \rceil + 1\right)$
- <u>*Idea:*</u> ***Divide and conquer:*** sort subfiles and merge



General External Merge Sort

- To sort a file with N pages using B buffer pages:
  - *Pass 0: use B buffer pages. Produce* $\lceil N / B \rceil$ *sorted runs of B pages each.*
  - *Pass 2, …, etc.: merge B-1 runs.*



Cost of External Merge Sort

- Number of passes: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- Cost = 2N * (# of passes)
- E.g., with 5 buffer pages, to sort 108 page file:
  - *Pass 0:* $\lceil 108 / 5 \rceil$ *= 22 sorted runs of 5 pages each (last run is only 3 pages)*
  - *Pass 1:* $\lceil 22 / 4 \rceil$ *= 6 sorted runs of 20 pages each (last run is only 8 pages)*
  - *Pass 2: 2 sorted runs, 80 pages and 28 pages*
  - *Pass 3: Sorted file of 108 pages*

Number of Passes of External Sort

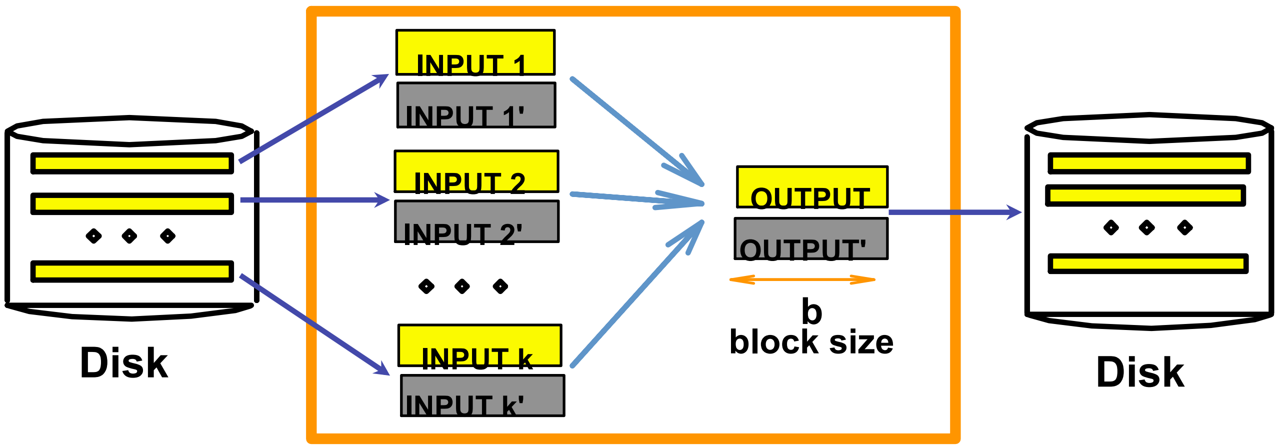
N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

I/O for External Merge Sort

- … longer runs often means fewer passes!
- Actually, do I/O a page at a time
- In fact, read a *block* of pages sequentially!
- Suggests we should make each buffer (input/output) be a *block* of pages.
  - *But this will reduce fan-out during merge passes!*
  - *In practice, most files still sorted in 2-3 passes.*

Double Buffering

- To reduce wait time for I/O request to complete, can *prefetch* into `shadow block'.
  - *Potentially, more passes; in practice, most files still sorted in 2-3 passes.*



B main memory buffers, k-way merge

Out-of-core algorithms: joins

Christoph Koch

School of Computer & Communication Sciences, EPFL

Simple Nested Loops Join

foreach tuple r in R do

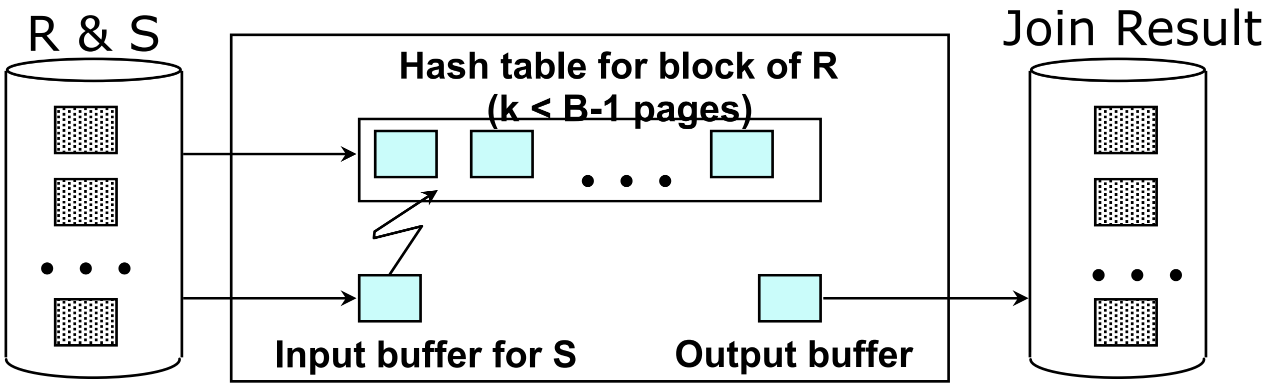
foreach tuple s in S do

if $r_i == s_j$ then add <r, s> to result

- For each tuple in the *outer* relation R, we scan the entire *inner* relation S.
  - *Cost: $M + p_R * M * N$ = 1000 + 100*1000*500 I/Os.*
- Page-oriented Nested Loops join: For each *page* of R, get each *page* of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.
  - *Cost: M + M*N = 1000 + 1000*500*
  - *If smaller relation (S) is outer, cost = 500 + 500*1000*

Block Nested Loops Join

- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ``block'' of outer R.
  - *For each matching tuple r in R-block, s in S-page, add <r, s> to result. Then read next R-block, scan S, etc.*

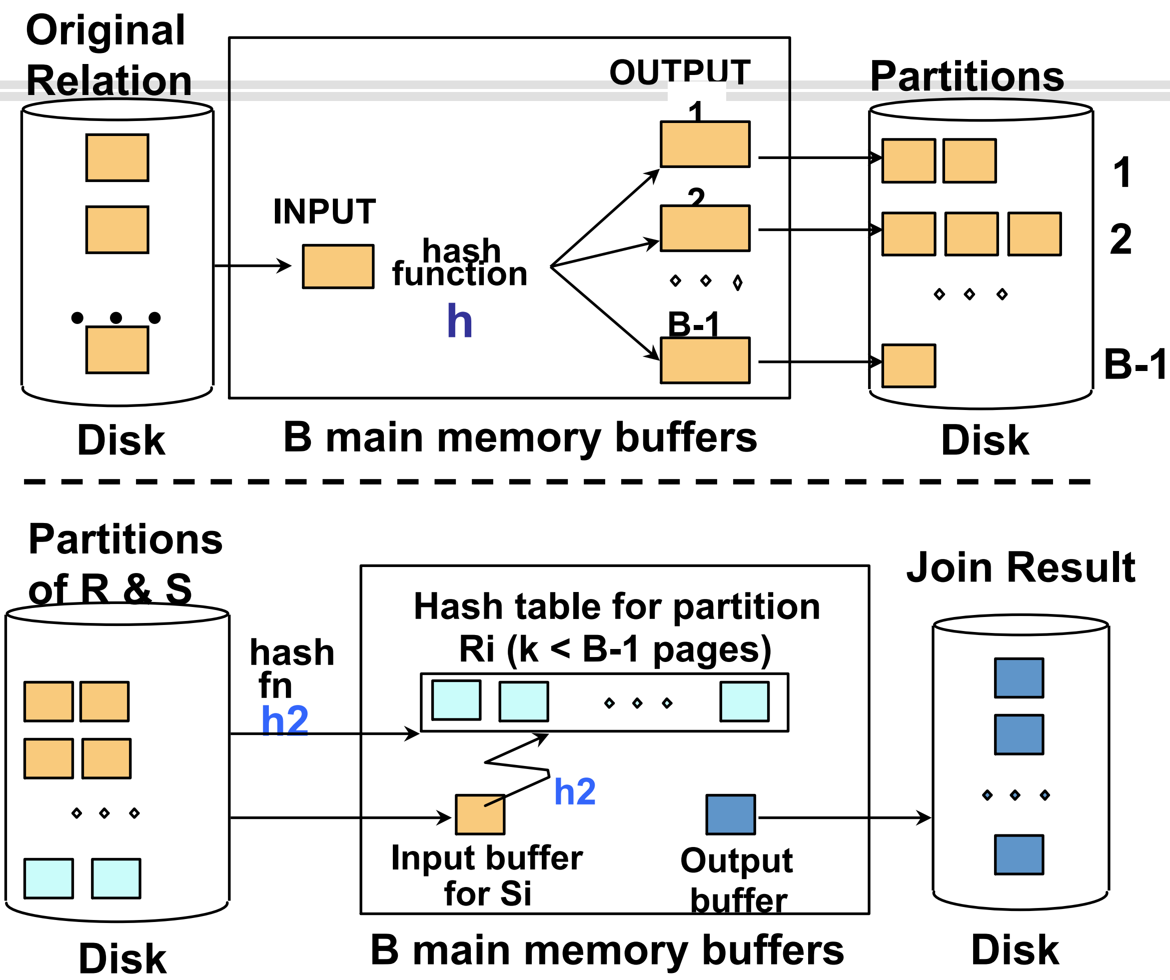


Block Nested Loops Join

- Cost: Scan of outer + #outer blocks * scan of inner
  - *#outer blocks =* $\lceil \#\ of\ pages\ of\ outer\ /\ blocksize \rceil$
- With *sequential reads* considered, analysis changes: may be best to divide buffers evenly between R and S.
  - *Depends on whether join processing can keep up with the scan of the inner relation.*

Hash-Join

- Partition both relations using hash fn **h**: R tuples in partition i will only match S tuples in partition i.
- Read in a partition of R, hash it using **h2 (<> h!)**. Scan matching partition of S, search for matches.



Observations on Hash-Join

- #partitions k < B-1, and B-2 > size of largest partition to be held in memory. Assuming uniformly sized partitions, and maximizing k, we get:
  - *k= B-1, and M/(B-1) < B-2, i.e., B must be >* $\sqrt{M}$
- If we build an in-memory hash table to speed up the matching of tuples, a little more memory is needed.
- If the hash function does not partition uniformly, one or more R partitions may not fit in memory. Can apply hash-join technique recursively to do the join of this R-partition with corresponding S-partition.

Cost of Hash-Join

- In partitioning phase, read+write both relns; 2(M+N).
- In matching phase, read both relns; M+N I/Os.

- Sort-Merge Join vs. Hash Join:
  - *Given a minimum amount of memory, both have a cost of 3(M+N) I/Os.*
  - *Hash Join superior on this count if relation sizes differ greatly.*
  - *Also, Hash Join shown to be highly parallelizable.*
  - *Sort-Merge less sensitive to data skew; result is sorted.*