

Data Processing on Modern Hardware

Jens Teubner, TU Dortmund, DBIS Group
`jens.teubner@cs.tu-dortmund.de`

Summer 2015

Part IV

Execution on Multiple Cores

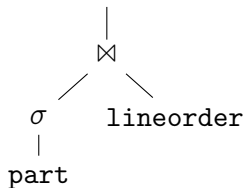
Example: Star Joins

Task: run parallel instances of the query (\nearrow introduction)

dimension

```
SELECT SUM(lo_revenue)
FROM part, lineorder
WHERE p_partkey = lo_partkey
AND p_category <= 5
```

fact table

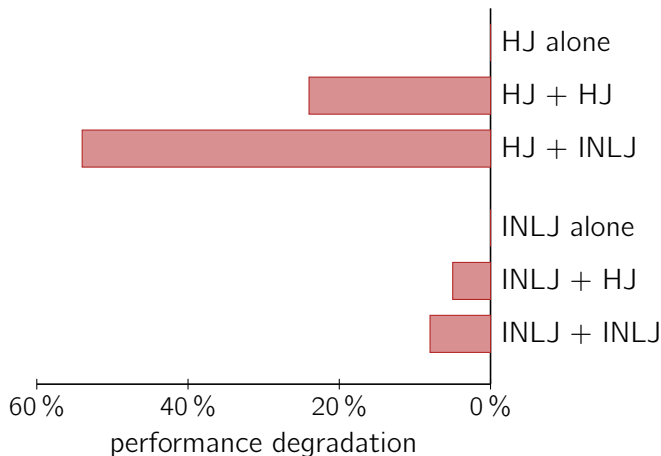


To implement \bowtie use either

- a **hash join** or
- an **index nested loops join**.

Execution on “Independent” CPU Cores

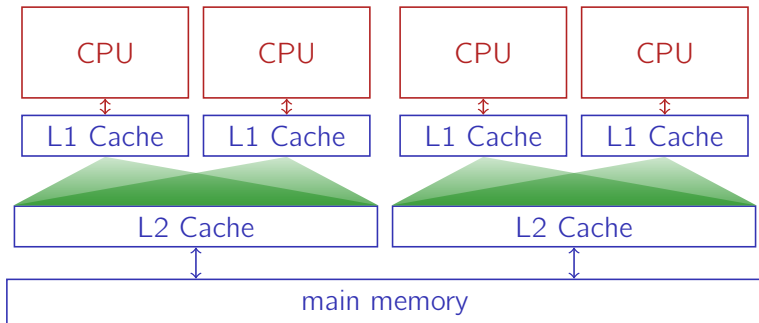
Co-run independent instances on different CPU cores.



Concurrent queries may seriously affect each other's performance.

Shared Caches

In Intel Core 2 Quad systems, two cores **share** an L2 Cache:

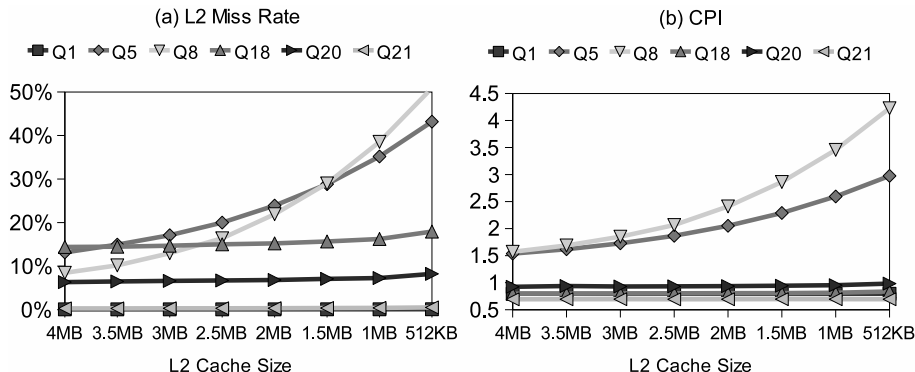


What we saw was **cache pollution**.

→ How can we avoid this cache pollution?

Cache Sensitivity

Dependence on cache sizes for some TPC-H queries:



Some queries are more sensitive to cache sizes than others.

- **cache sensitive:** hash joins
- **cache insensitive:** index nested loops joins; hash joins with very small or very large hash table

This behavior is related to the **locality strength** of execution plans:

Strong Locality

small data structure; reused very frequently

- e.g., small hash table

Moderate Locality

frequently reused data structure; data structure \approx cache size

- e.g., moderate-sized hash table

Weak Locality

data not reused frequently or data structure \gg cache size

- e.g., large hash table; index lookups

Execution Plan Characteristics

Locality effects how caches are used:

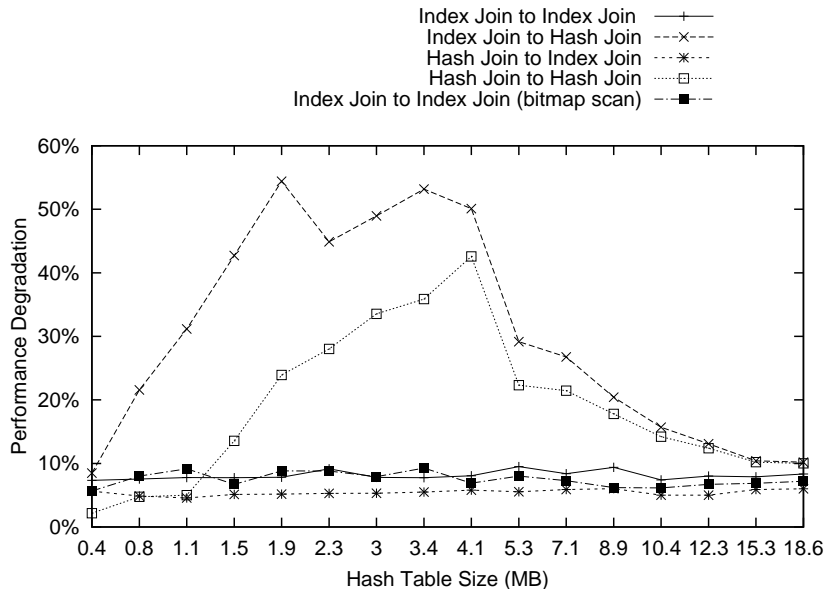
cache pollution	strong	moderate	weak
amount of cache used	small	large	large
amount of cache needed	small	large	small

Plans with **weak locality** have most severe impact on co-running queries.

Impact of co-runner on query:

	strong	moderate	weak
strong	low	moderate	high
moderate	moderate	high	high
weak	low	low	low

Experiments: Locality Strength



Source: Lee et al. MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases. VLDB 2009

Locality-Aware Scheduling

An optimizer could use knowledge about localities to **schedule** queries.

- **Estimate** locality during query analysis.
 - Index nested loops join \rightarrow weak locality
 - Hash join:

hash table \ll cache size \rightarrow strong locality

hash table \approx cache size \rightarrow moderate locality

hash table \gg cache size \rightarrow weak locality

- **Co-schedule** queries to minimize (the impact of) cache pollution.

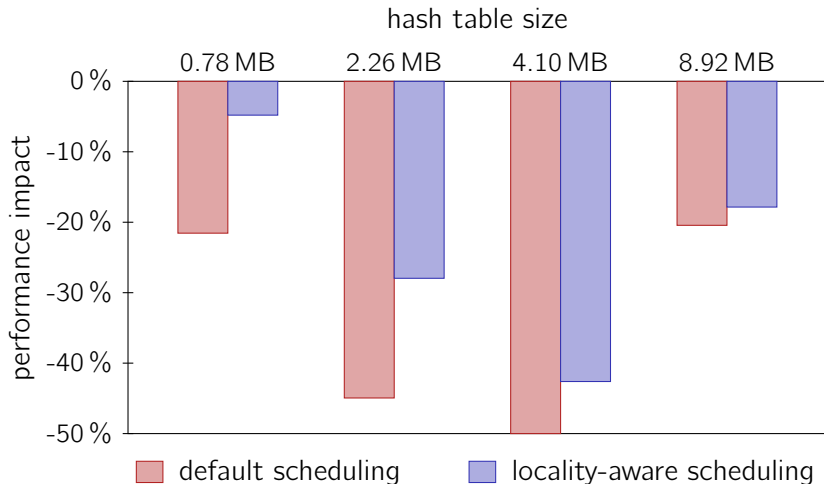


Which queries should be co-scheduled, which ones not?

- Only run weak-locality queries next to weak-locality queries.
 - \rightarrow They cause high pollution, but are not affected by pollution.
- Try to co-schedule queries with small hash tables.

Experiments: Locality-Aware Scheduling

PostgreSQL; 4 queries (different `p_category`s); for each query: $2 \times$ hash join plan, $2 \times$ INLJ plan; impact reported for hash joins:

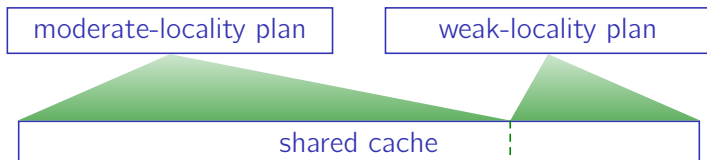


Source: Lee et al. VLDB 2009.

Cache Pollution

Weak-locality plans cause cache pollution, because they **use** much cache space even though they do not strictly **need** it.

By **partitioning** the cache we could reduce pollution with little impact on the weak-locality plan.



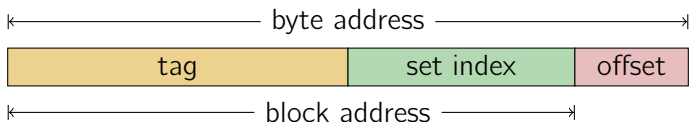
But:

- Cache allocation controlled by **hardware**.

Cache Organization

Remember how caches are organized:

- The **physical address** of a memory block determines the **cache set** into which it could be loaded.



Thus,


- We can **influence hardware behavior** by the **choice of physical memory allocation**.

Page Coloring

The address \leftrightarrow cache set relationship inspired the idea of **page colors**.

- Each memory page is assigned a **color**.⁵
- Pages that map to the **same cache sets** get the **same color**.



 **How many colors are there in a typical system?**


⁵Memory is organized in **pages**. A typical **page size** is **4 kB**.

- By using memory only of certain colors, we can effectively restrict the cache region that a query plan uses.

Note that

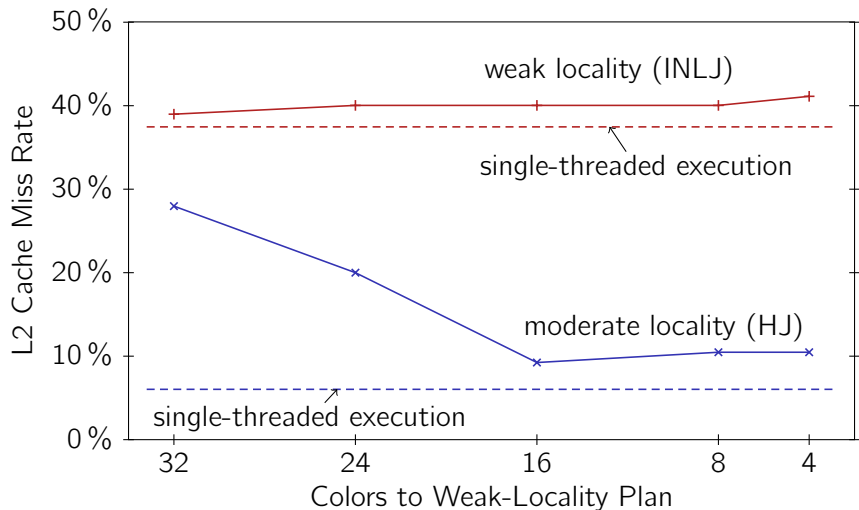
- Applications (usually) have **no control** over physical memory.
- Memory allocation and virtual \leftrightarrow physical mapping are handled by the **operating system**.
- We need **OS support** to achieve our desired **cache partitioning**.

MCC-DB (“Minimizing Cache Conflicts”):

- Modified Linux 2.6.20 kernel
 - Support for **32 page colors** (4 MB L2 Cache: 128 kB per color)
 - **Color specification** file for each process (may be modified by application at any time)
- Modified instance of PostgreSQL
 - **Four colors** for regular buffer pool
 -  **Implications on buffer pool size (16 GB main memory)?**
 - For **strong- and moderate-locality** queries, allocate colors as needed (*i.e.*, as estimated by query optimizer)

Experiments

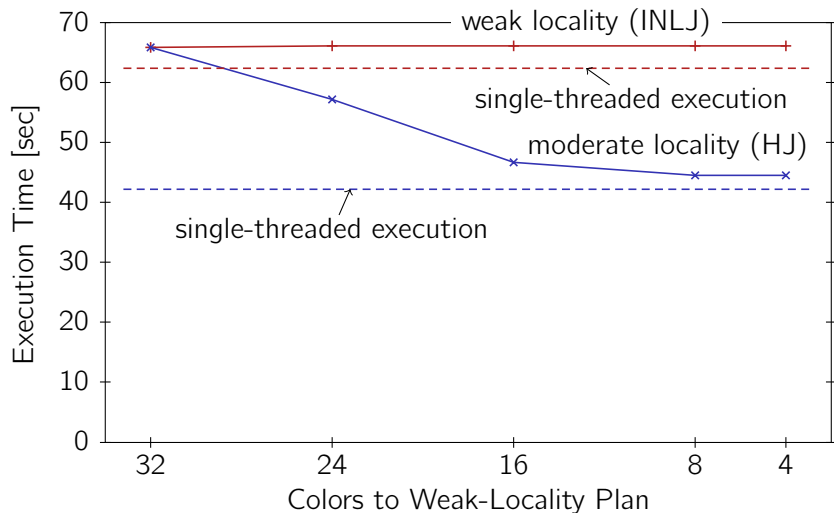
Moderate-locality hash join and weak-locality co-runner (INLJ):



Source: Lee et al. VLDB 2009.

Experiments

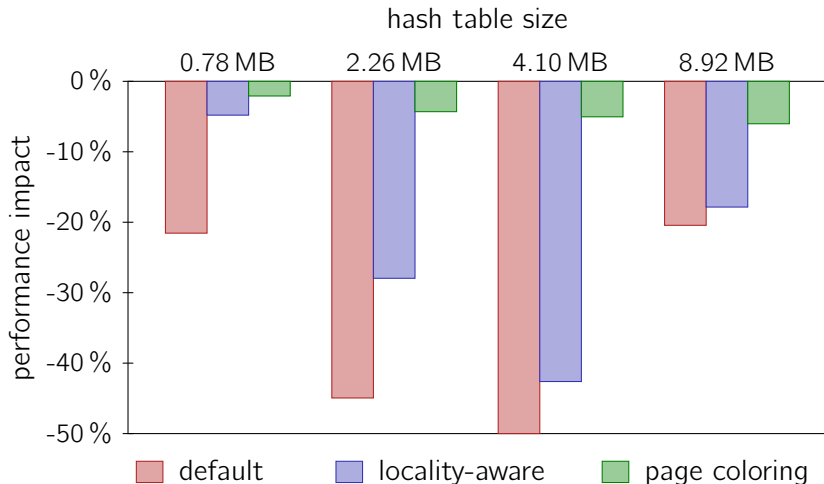
Moderate-locality hash join and weak-locality co-runner (INLJ):



Source: Lee et al. VLDB 2009.

Experiments: MCC-DB

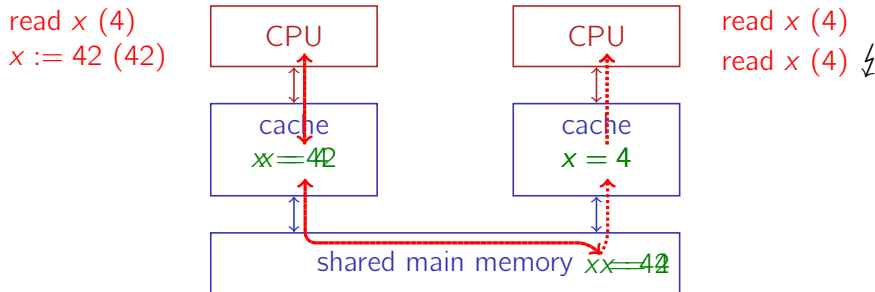
PostgreSQL; 4 queries (different `p_category`s); for each query: $2 \times$ hash join plan, $2 \times$ INLJ plan; impact reported for hash joins:



Source: Lee et al. VLDB 2009.

Caches and Shared Memory

We saw before **local caches** on top of a **shared memory**:

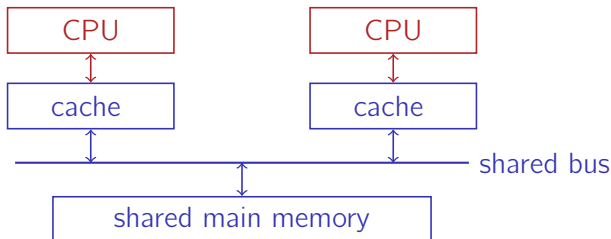


Challenge: Ensure **cache coherency** during parallel data access.

→ Cache coherence protocol

Cache Coherence Protocol—Snooping-Based

Machine Model:



Idea:

- Caches can **see** each others' interactions with main memory.
- **Keep track** of shared data items in caches.

Cache Coherence Protocol—Snooping-Based

Writes **invalidate** all copies of a data item:

Activity	Bus	Cache A	Cache B	Memory
				$x = 4$
A reads x	cache miss for x	$x = 4$		$x = 4$
B reads x	cache miss for x	$x = 4$	$x = 4$	$x = 4$
A reads x	– (cache hit)	$x = 4$	$x = 4$	$x = 4$
B writes x	invalidate x	$x = 4$	$x = 42$	$x = 4^6$
A reads x	cache miss for x	$x = 42$	$x = 42$	$x = 42$

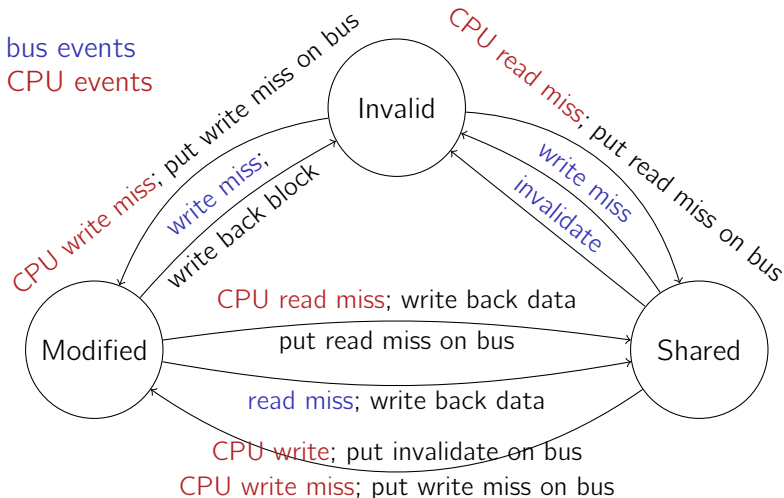
- After invalidation, a cache will re-fetch data item upon a read.
- Since the bus is shared, other caches may answer “cache miss” messages (\leadsto necessary for write-back caches).

⁶With write-through caches, memory will be updated immediately.

Cache Coherence Protocol—Snooping-Based

Remember the 'status' field in the CPU cache (↗ slide 31)?

→ Track cache line status according to bus messages.



Cache Coherence Protocol—MESI

Actual implementations use extensions of this “MSI” protocol:

- M Modified**—cache (exclusively) holds a modified copy
- E Exclusive**—no other cache has a copy of this item
 - can do writes without asking
- S Shared**—cache holds a valid copy of the item, but others might have one, too
- I Invalid**—contents of cache line are invalid

AMD Opteron implements the “MOESI” protocol:

- O Owned**—shared item, but out of date in memory
 - In MESI, CPU must do a write to memory during E → S.

E.g., Intel Xeon E7-8880 v3:

- 2.3 GHz clock rate
- 18 cores per chip (36 threads)
- Up to 8 processors per system

Back-of-the-envelope calculation:

- 1 byte per cycle per core → 331 GB/s
- Data-intensive applications might demand much more!

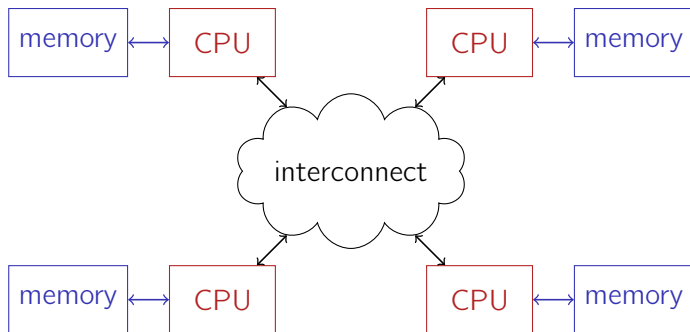
Shared memory bus?

- Modern bus standards can deliver at most a few ten GB/s.
- Switching very high bandwidths is a challenge.

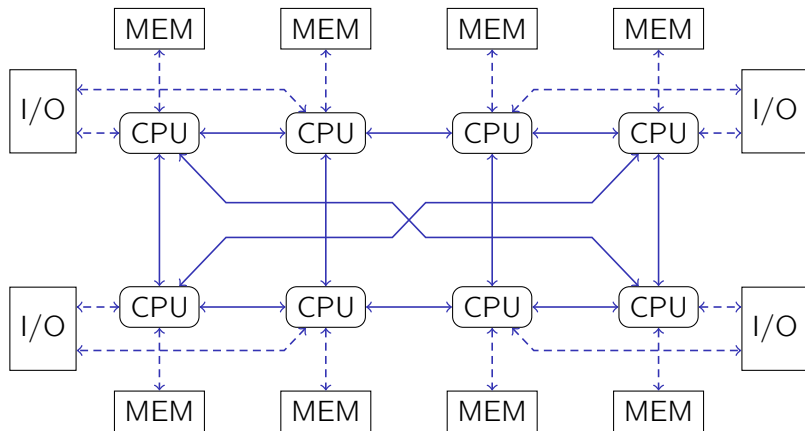
Distributed Shared Memory

Idea: Distribute memory

→ Attach to individual compute nodes



Example: 8-Way Intel Nehalem-EX



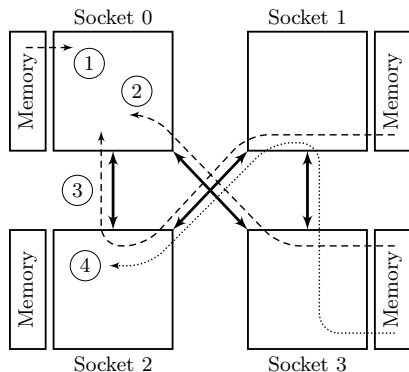
Intel QuickPath Interconnect: Point-to-point links between chips

NUMA—Non-Uniform Memory Access



Distribution makes memory access **locality-sensitive**.

→ **Non-Uniform Memory Access (NUMA)**

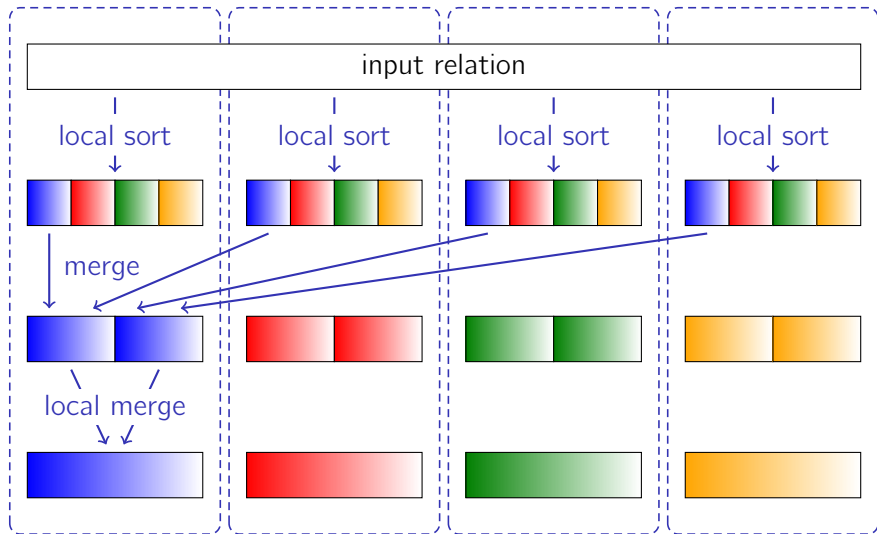


	bandwidth	latency
①	24.7 GB/s	150 ns
②	10.9 GB/s	185 ns
③	10.9 GB/s	230 ns
③/④ ⁷	5.3 GB/s	235 ns

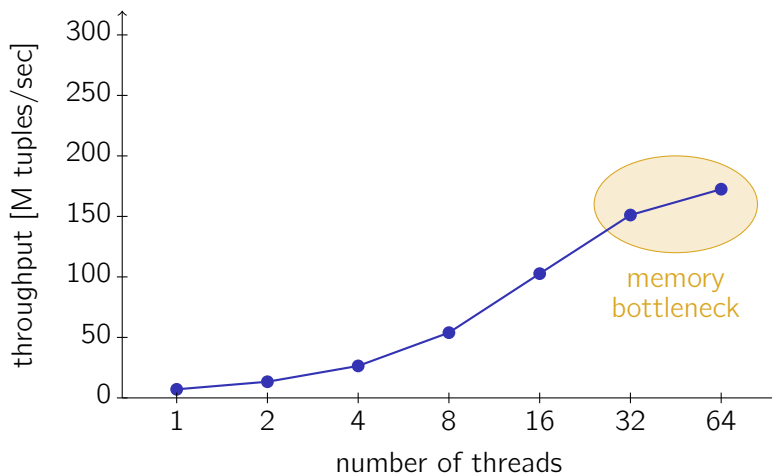
↗ Li *et al.* NUMA-Aware Algorithms: The Case of Data Shuffling. *CIDR 2013*

⁷ ③ with cross traffic along ④.

Sorting and NUMA



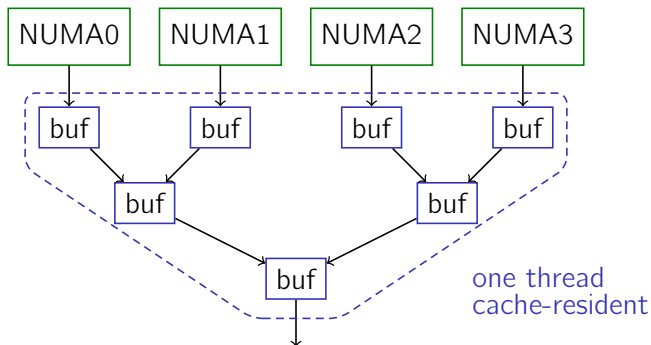
Resulting Throughput



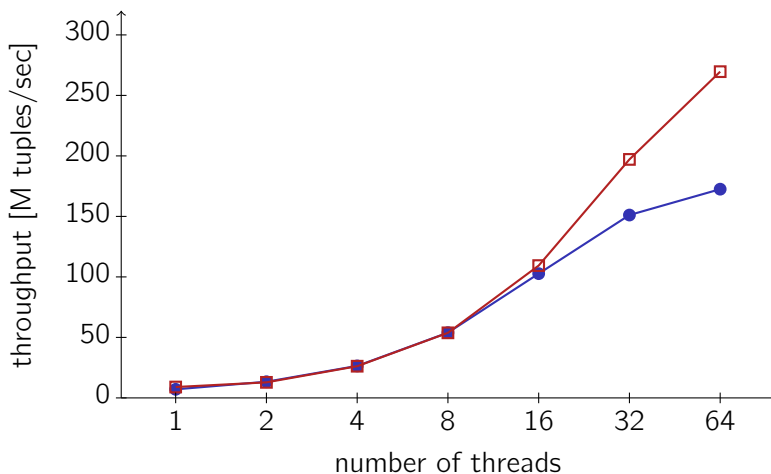
NUMA and Bandwidth

Problem: Merging is **bandwidth-bound**.

- Merge multiple runs (from NUMA regions) at once
(Two-way merging would be more CPU-efficient because of SIMD.)
- Might need **more instructions**, but brings bandwidth and compute **into balance**.



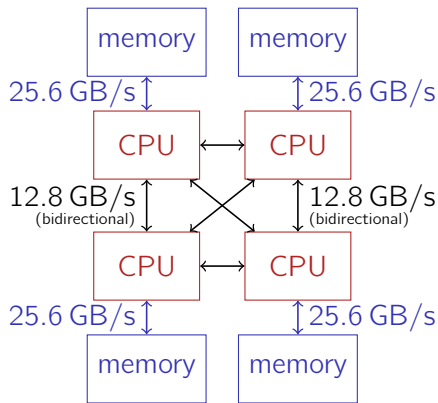
Throughput With Multi-Way Merging



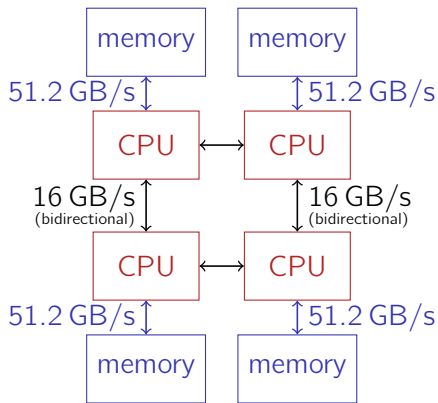
NUMA Effects in Detail

Bandwidth:

- Single links have **lower bandwidth** than memory controllers.



Intel Nehalem EX



Intel Sandy Bridge EP

Bandwidth:

- Improve **locality** code \leftrightarrow data.
- Use bandwidth **evenly**.
- **Balance** bandwidth and compute load (see previous example).

Latency:

- Observation: Data might be 0, 1, or more “hops” away.
- **However:**
 - **Cache coherency** might make conclusions difficult.

Snooping-based cache coherency:

- **No shared bus** to synchronize (and snoop).
- Must
 - (a) **broadcast** coherency messages across bus and
 - (b) **wait** for responses.(E.g., a far-away cache might hold a modified copy of a data item.)

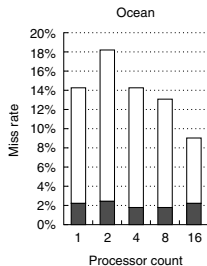
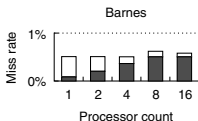
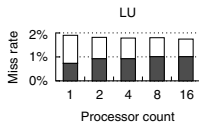
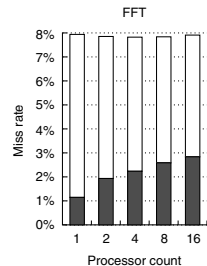
Effect:

- latency locally \approx latency remote (!)

Problem:

- Lots of broadcast traffic \rightarrow poor **scalability**

Snooping-Based Cache Coherency: Scalability



■ Coherence miss rate □ Capacity miss rate

Example:

- Scientific Applications
- ↗ Hennessy
Patterson, Section I.5

To improve scalability:

- Keep a **directory** that knows whether and where data copies exist in any cache.
- Broadcasts → directed messages

Typically:

- **Collocate** directory with memories/caches.
- Intuitively, keep **status information** (↗ slide 31) for every cache line and every cache in the system.

Strategies for NUMA

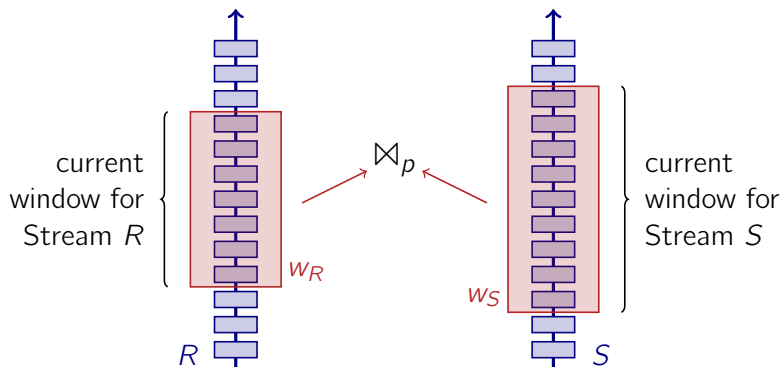
Idea:

- Make algorithms **communication-aware**.
- Specifically, respect **NUMA topology**.

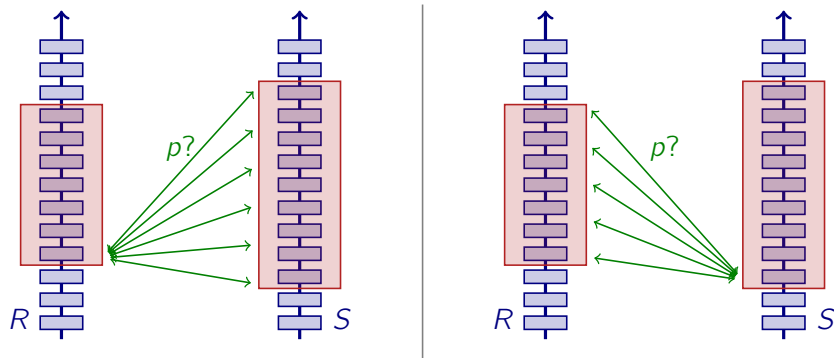
Example:

- Joins over data streams.

Joins Over Data Streams:



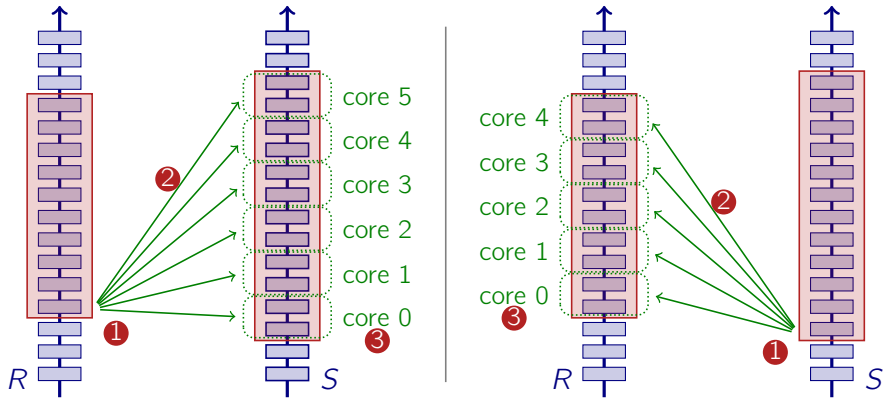
Task: Find all $\langle r, s \rangle$ in w_R, w_S that satisfy $p(r, s)$.



1. **scan** window, 2. **insert** new tuple, 3. **invalidate** old

NUMA-Aware Execution?

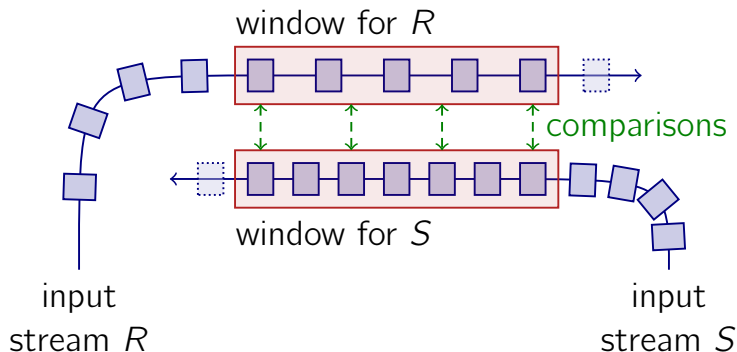
CellJoin [Gedik *et al.*, VLDBJ 2009]



- replicate partition
- ① **bandwidth** bottlenecks
 - ② **long-distance** communication
 - ③ **centralized** coordination and memory

→ Parallel, but not NUMA-aware.

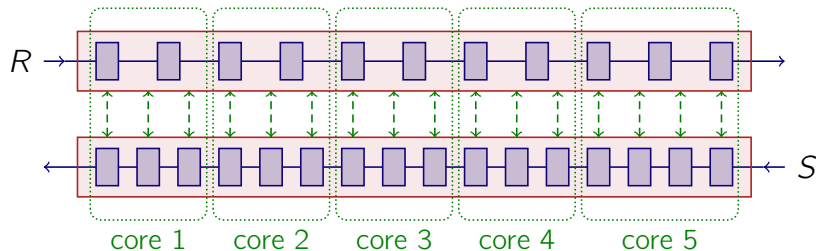
Handshake Join:



Streams flow by in **opposite directions**
Compare tuples when they **meet**

Handshake Join on Many Cores

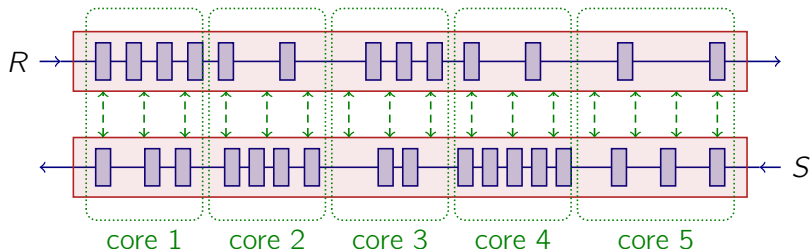
Data flow representation → **parallelization**:



■ **No bandwidth bottleneck** ① ✓

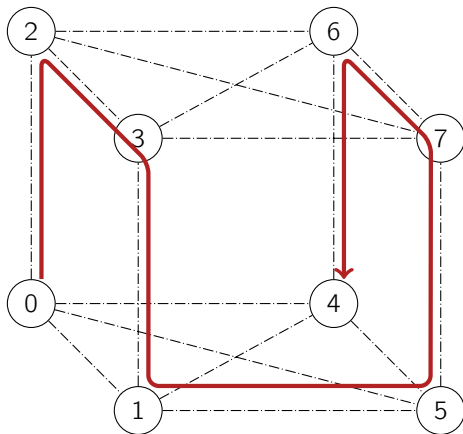
■ **Communication/synchronization stays local** ② ✓

Coordination can now be done **autonomously**

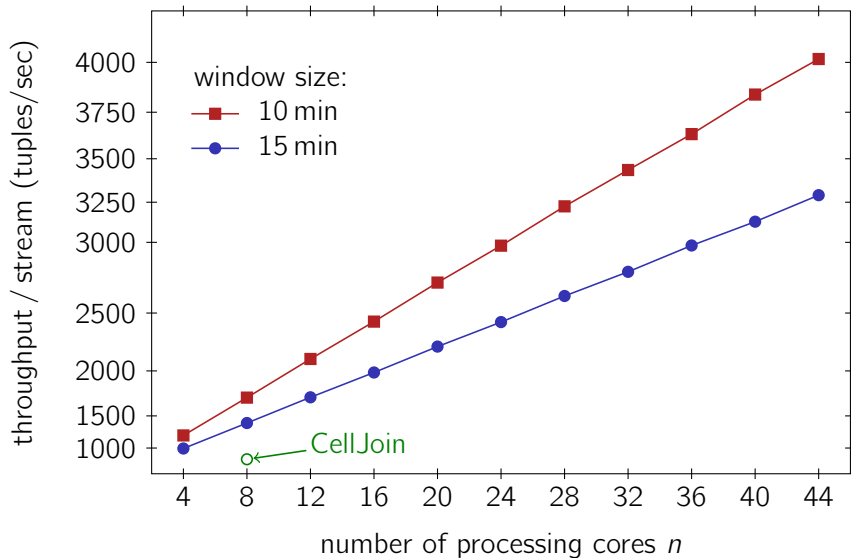


- no more centralized coordination ③ ✓
- Autonomous **load balancing**
- **Lock-free message queues** between neighbors

Example: AMD “Magny Cours” (48 cores)



Experiments (AMD Magny Cours, 2.2 GHz)



Beyond 48 Cores. . . (FPGA-based simulation)

