Data Processing on Modern Hardware

Jana Giceva

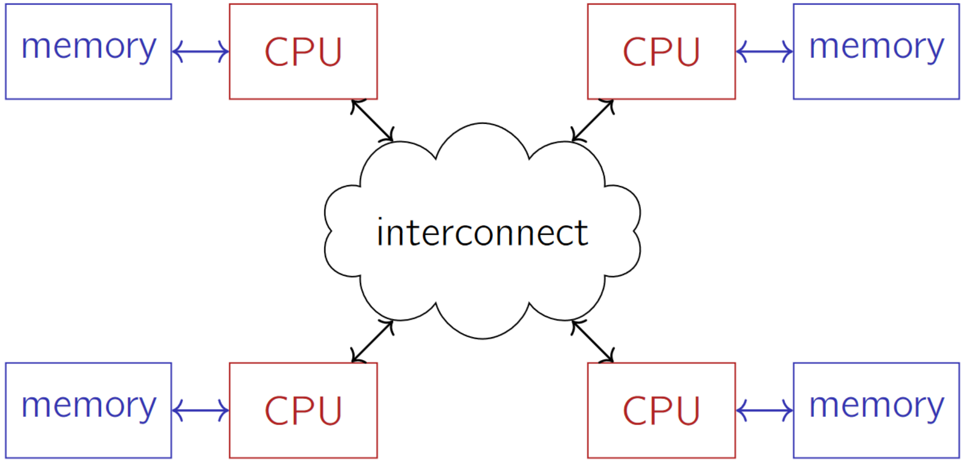
Lecture 8: Multicore CPUs

NUMA, chiplet-based processors

Non-uniform memory access (NUMA)

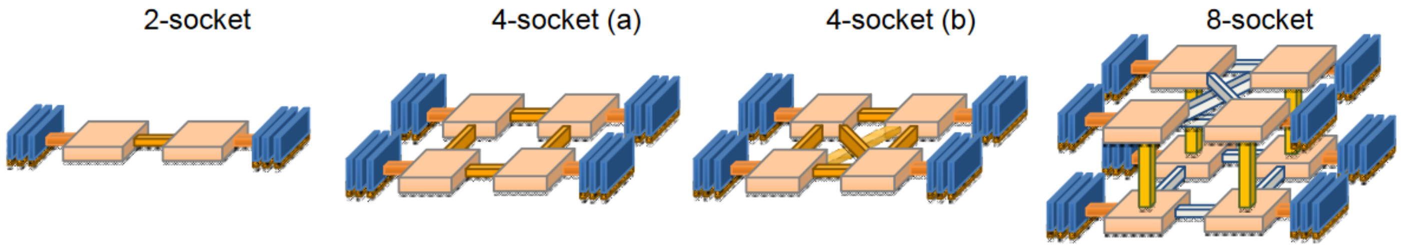
Distributed Shared Memory

- Single processor scalability (with shared memory) has limitations
- Idea: distribute memory



NUMA hardware

- Almost all mid-range and enterprise servers today are multi-socket



- Each server typically contains between 2-8 sockets

- Each socket contains:
  - between 4 and 52 cores (up to 96 cores on AMD Zen4)
  - a few memory DIMM modules attached through memory channels

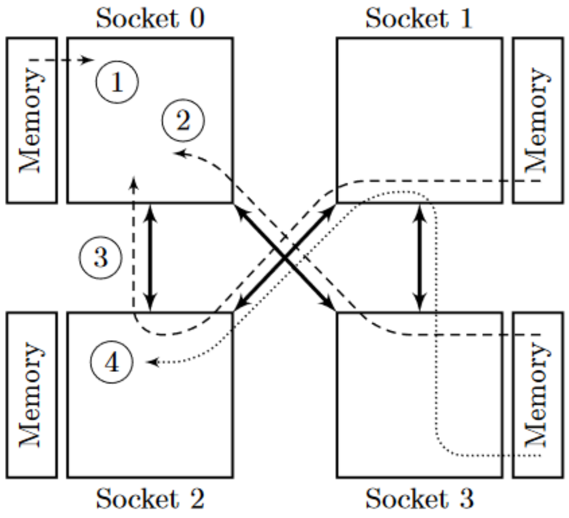
- An interconnect network among the sockets allows each core to access non-local memory

src. Li et al. “NUMA-aware algorithsm: the case of data shuffling” *CIDR 2013*

Effects of NUMA Hardware

- **Example multi-socket server:**
  - four 8-core Nehalem-EX processors, fully connected with 4 bi-directional 3.2 GHz QuickPath Interconnect (QPI)

- Measure performance for reading local vs. remote memory
  - **Flow 1: read locally** (from socket 0)
    - max. aggregate bandwidth (12 threads) is 24.7 GB/s
    - latency is 340 CPU cycles (~ 150ns)
  - **Flow 2: read remote** (over **1 QPI link**, from socket 3)
    - max. aggregate bandwidth is 10.9 GB/s
    - latency is 420 CPU cycles (~185ns)
  - **Flow 3: read remote** (over **2 QPI links**, from socket 1)
    - max. aggregate bandwidth is 10.9 GB/s
    - latency is 520 CPU cycles (~230ns)
  - **Flow 4: read remote** (over **2 QPI links**) **with cross traffic**
    - max. aggregate bandwidth is 5.3 GB/s
    - latency is 530 CPU cycles (~235ns)



src. Li et al. "NUMA-aware algorithsm: the case of data shuffling" *CIDR 2013*

What does that mean for data processing?

- **Designing algorithms and data structures**
  - Need to differentiate between local and remote memory
  - Local memory is faster and has higher bandwidth

- **Concurrency**
  - Synchronization within a socket / NUMA node is significantly faster
  - Concurrent data structures needs to scale across NUMA nodes

- **NUMA effects in systems and databases**

System‘s support for NUMA

- Modern **operating systems** are aware of NUMA architectures.
  - Linux partitions memory into NUMA zones, one for each socket.
  - For each NUMA zone, the kernel maintains separate management data structures.

- By **default** the Linux kernel **allocates memory** on the local NUMA node
  - The socket of the core on which the current thread is scheduled on.

- **Unless explicitly bound** to a specified socket (NUMA zone) through the **mbind() system call**

Memory allocation

- **Watch out for default OS**
  - **First touch** allocation **policy** (static and in place **until kernel version 2.6**)
  - **Today** there are two options:
    - **Transparent NUMA awareness**:
      - Allocate locally
      - Migrate thread or data to achieve good NUMA performance and balancing
    - **Explicit memory allocation policy**
      - Allocate memory (and do not migrate) based on the selected policy

- **NUMA memory policy**
  - System default policy
    - local (general), interleaved (during boot-up)
  - Task/Process policy – controls all page allocations made by or on behalf of the task
  - VMA policy – to a range of a task’s virtual address space

Memory allocation

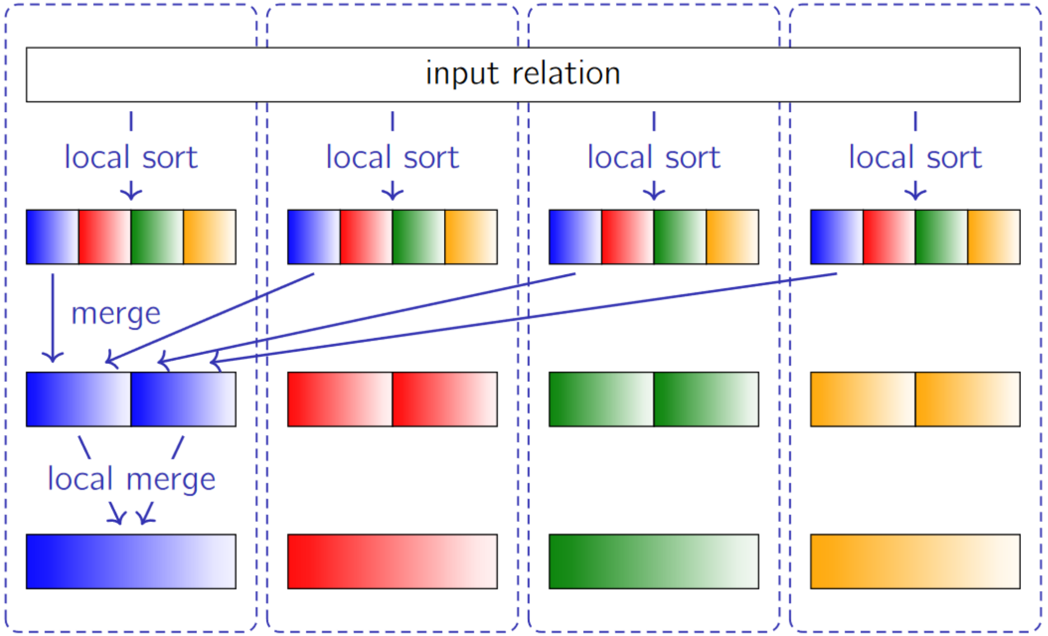
- **Allocation modes**
  - **Local** – memory from the local NUMA node
  - **Bind** – memory from the set of nodes specified by the policy
  - **Preferred –** memory from the set of nodes specified by the policy, if available
  - **Interleaved** – memory interleaved across all the NUMA nodes in the set provided by the policy

- Invoked from the process / thread itself or via the `numactl` library

- Memory policy APIs
  - `long set_mempolicy(int mode, const unsigned long *nmask, unsigned long maxnode)`
  - `long get_mempolicy(int *mode, const unsigned long *nmask, unsigned long maxnode, void *addr, int flags);`
  - `long mbind(void *start, unsigned long len, int mode, const unsigned long *nmask, unsigned long maxnode, unsigned flags)`

Where does it matter?

- Example 1: Sorting and NUMA



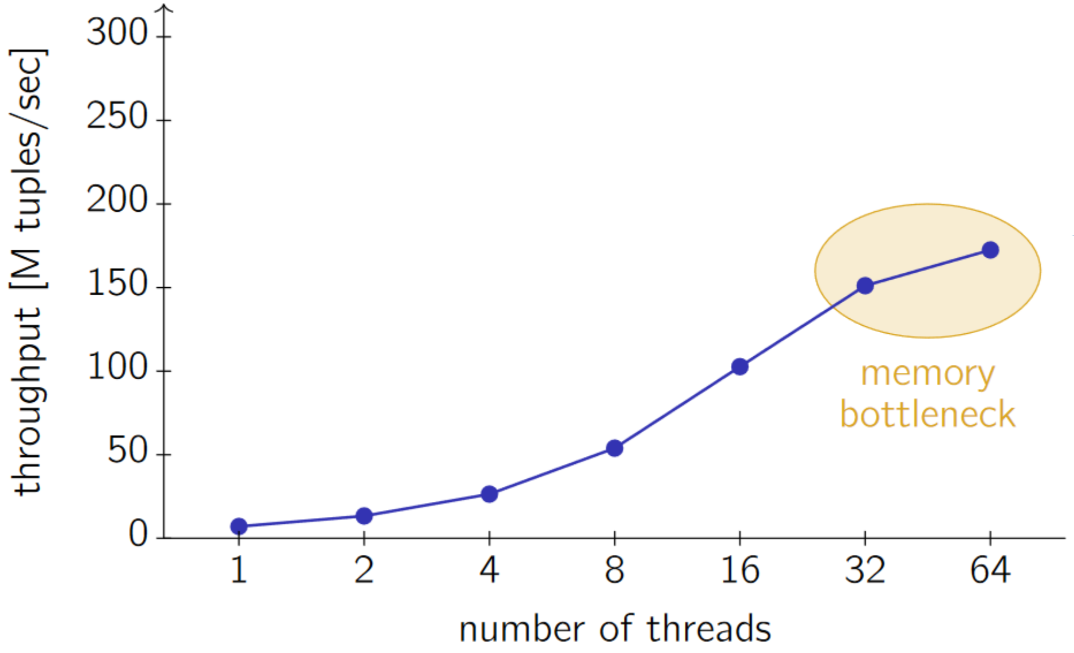
Step 1: Operate on local data as much as possible.

Step 2: Exchange data to gather local partitions on a local place.

Step 3: Merge the results locally.

src. Balkesen et al. “Multi-core, Main-Memory Joins: Sort vs Hash Revisited” *VLDB 2014*

Sorting and NUMA

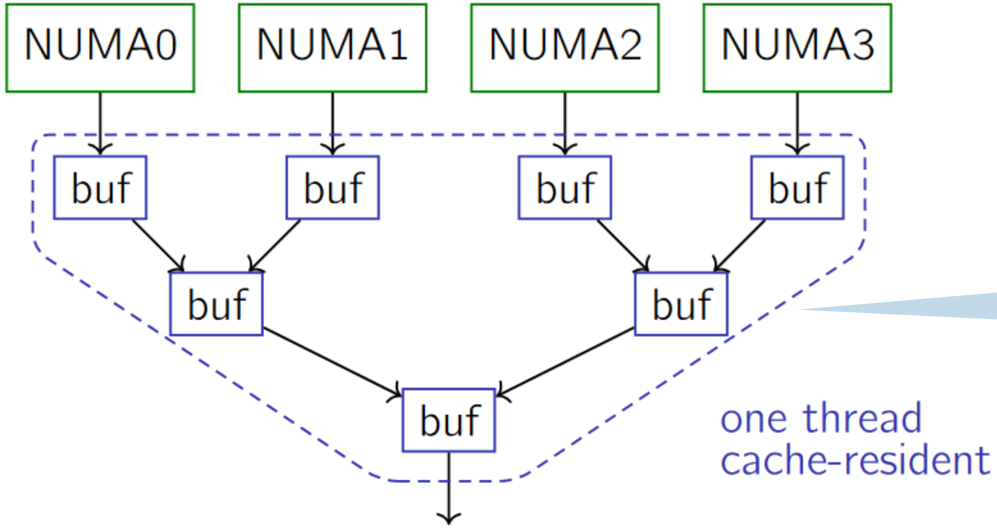


Even though NUMA-aware, the algorithm does not scale well.

Step 2 is very memory-bandwidth intensive and saturates the system‘s resources.

src. Balkesen et al. “Multi-core, Main-Memory Joins: Sort vs Hash Revisited” *VLDB 2014*

Multi-way merging as an alternative

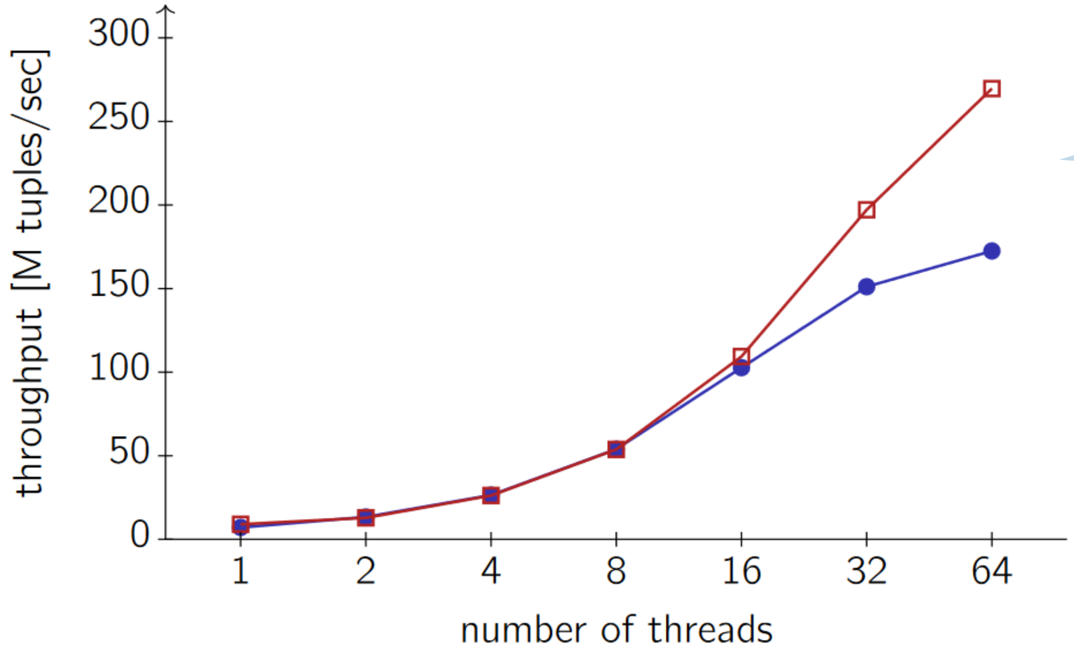


Step 1: Operate on local data as much as possible. (similar to before)

Step 2: Gather data in a cache- and NUMA-conscous way. Recall the multi-way sort when we did SIMD?

src. Balkesen et al. “Multi-core, Main-Memory Joins: Sort vs Hash Revisited” *VLDB 2014*

Sorting and NUMA

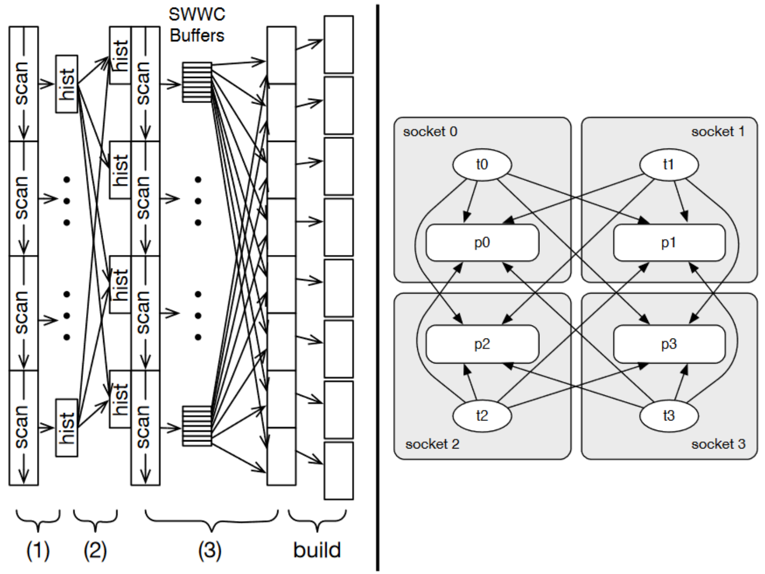


Performance speed-up much better than before because of NUMA-awareness.

src. Balkesen et al. “Multi-core, Main-Memory Joins: Sort vs Hash Revisited” *VLDB 2014*

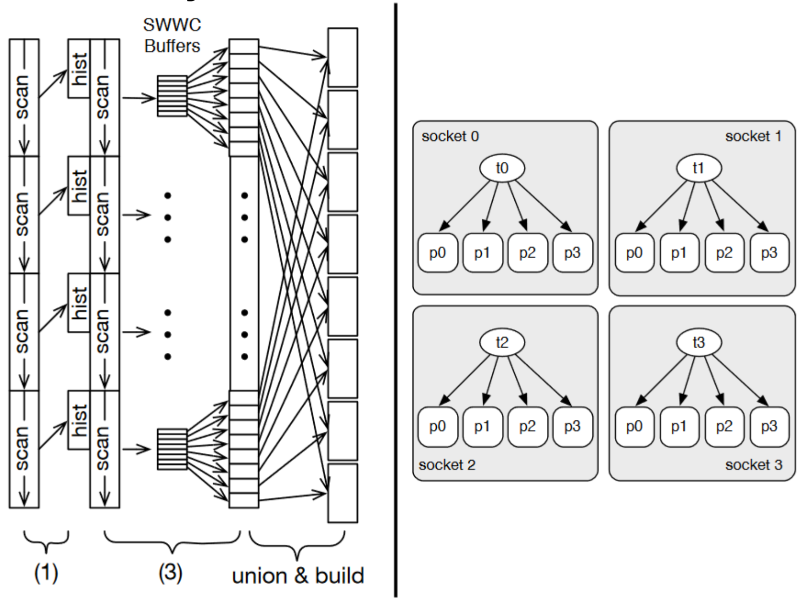
Radix-join and NUMA

v1: radix-join PRO



- Step (3) introduces a lot of small random writes to remote memory (different NUMA regions).
- Can we somehow avoid it?

v2: radix-join CPRL



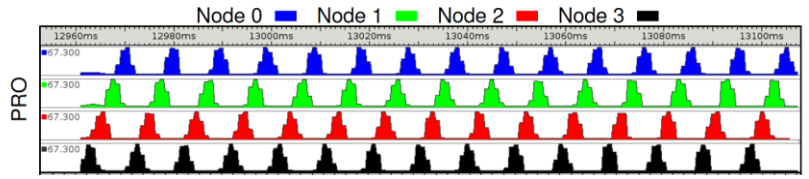
- Skip step (2) – computing the global histogram
- Proceed with a node-local step (3)
- Trade-off small remote-writes for large remote-reads

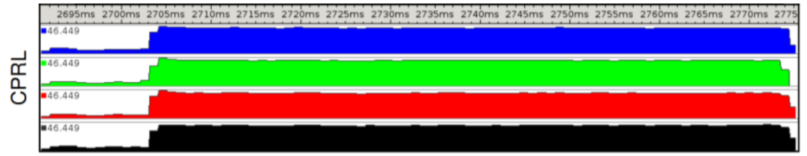
Src: Schul et al. "An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory." *SIGMOD 2016*

Radix-join and NUMA

- **How about the join phase?**
  - All radix-join algorithms build co-partitions that should be joined independently.
  - They are scheduled using a LIFO task-queue, to be processed by different threads.
  - With a naive schedule, all the first threads taking a task from the queue will have to read their input data from the same NUMA region.
  - For ¾ of the threads, this is a remote node.

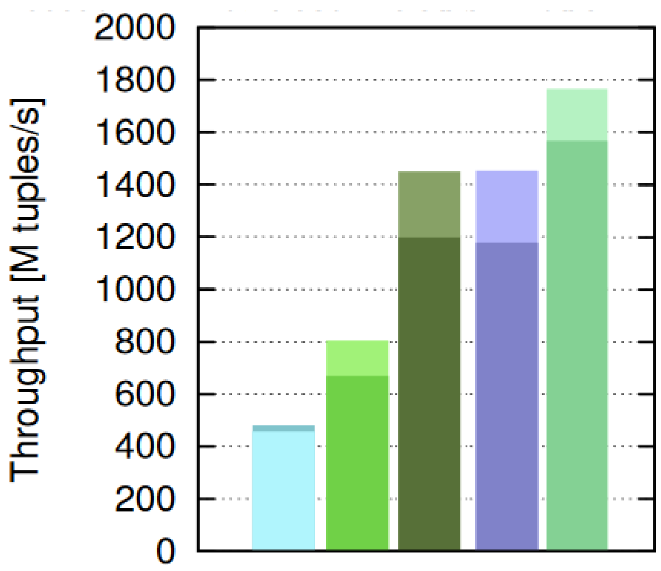
- **NUMA-aware scheduling**
  - Can improve this by reordering the join tasks in an interleaved fashion or by using separate task queues for each NUMA region.





Src: Schul et al. "An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory." *SIGMOD 2016*

Radix-join and NUMA



- 🟦 radix join (basic)
- 🟩 no partitioned hash join with prefetching
- 🟩 radix join (with chain hashing) with optimizations
- 🟪 radix join (with linear hashing) with optimizations
- 🟩 radix join (with LH) with optimizations and NUMA awareness

NUMA-awareness brings additional 20-30% improvement.

Src: Schul et al. "An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory." *SIGMOD 2016*

NUMA-awareness engine-wide

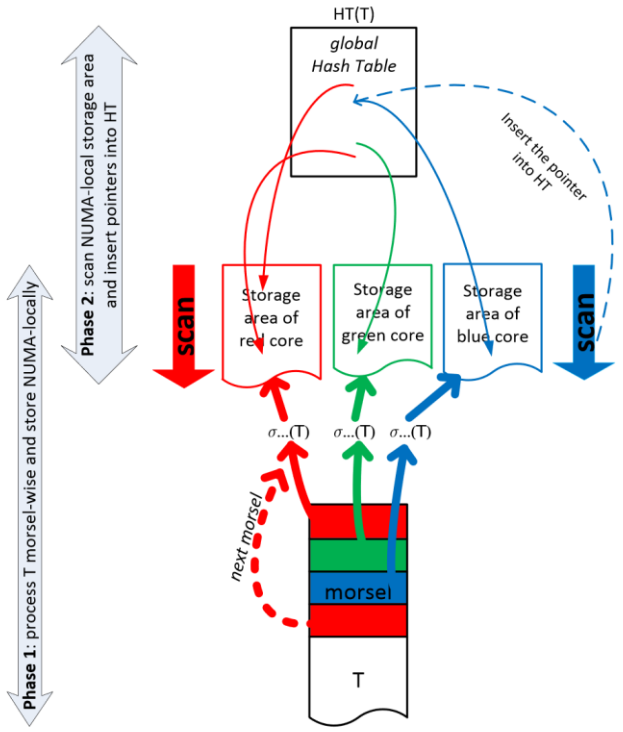
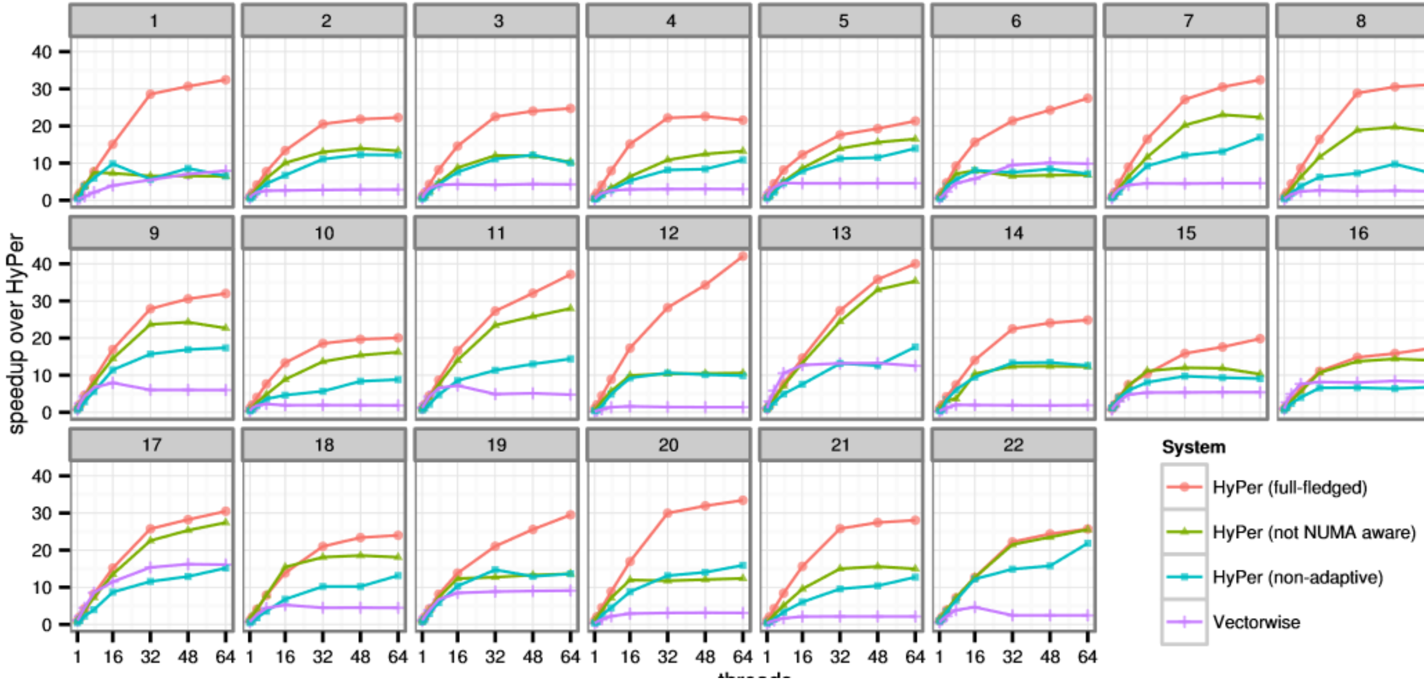

HT(T)
global Hash Table
Insert the pointer into HT
Phase 2: scan NUMA-local storage area and insert pointers into HT
Phase 1: process T morsel-wise and store NUMA-locally
scan
Storage area of red core
Storage area of green core
Storage area of blue core
scan
σ...(T)
σ...(T)
σ...(T)
next morsel
morsel
T

Figure 3: NUMA-aware processing of the build-phase

- Relation T is interleaved “morsel-wise” across the NUMA nodes
- The scheduler assigns a morsel located on the same NUMA node where the thread is executed.
- In the first phase the filtered tuples are inserted into NUMA-local storage areas, i.e., for each core there is a separate storage area in order to avoid synchronization.
- The global HT is probed by threads located on various sockets of a NUMA system.
  - To avoid contention, it is interleaved across all sockets.

src. Leis et al. “Morse-Driven Parallelism: A NUMA-aware query evaluation framework for the many-core age” *SIGMOD 2014*

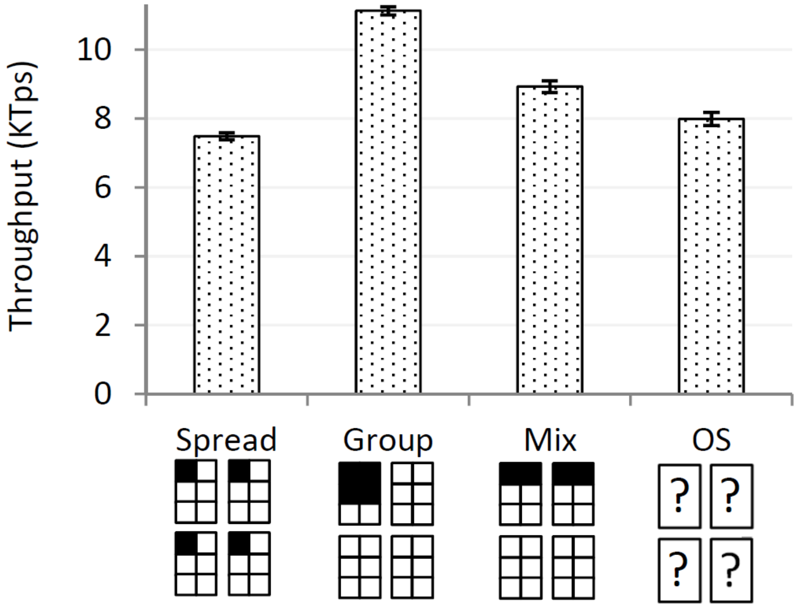
Engine-wide NUMA awareness



src. Leis et al. “Morse-Driven Parallelism: A NUMA-aware query evaluation framework for the many-core age” *SIGMOD 2014*

Impact of NUMA on DB synchronization

- Performance implications for synchronization and concurrency



- **Goal:** check the impact of NUMA latencies on OLTP transactions and the overall throughput.
- **OLTP workload**: TPC-C payment transaction
- **Machine:** 4 CPUs with 6 cores earch
- **Test:** Run the database with 4 worker threads, either using the default OS scheduling or pinning them to different cores.

- **Insights:**
  – DB threads collocated on the same NUMA node exhibit much better performance than alternatives.
  – Communication over the interconnect is expensive.
  – OS-scheduling can be unpredictable.

src. Porobic et al. “Analyzing the impact of system architecture on the scalability of OLTP engines for high-contention workloads” *VLDB 2017*

Locks and NUMA

- **Synchronization within the processor is cheaper than to synchronization over the interconnect**
  – due to **latency** concerns, but also for **increased memory traffic.**

- Two main approaches to make locks NUMA-aware locks (and concurrent data structures):
  – **Cohort locks** (hierarchical locks) [1]
  – **Combining** + remote core execution (select a leader, etc.) [2]

- Recent **black box approach** allows any **linear data structure** to be made NUMA-aware [3]

- Parking lock (e.g., optimized futex from last week) can be made a scalable and NUMA-aware blocking synchronization primitive: CST [4]

[1] Chabbi et al. *High Performance Locks for Multi-Level NUMA Systems.* PPoPP 2015
[2] Lozi et al. *Fast and Portable Locking for Multicore Architectures.* ACM Trans. Computing Systems 2016
[3] Calciu et al. *Black-box Concurrent Data Structures for NUMA Architectures.* ASPLOS 2017
[4] Kashyap et al. *Scalable NUMA-aware Blocking Synchronization Primitives.* USENIX ATC 2017
https://taesoo.kim/pubs/2017/kashyap:cst-slides.pdf

Chiplet-based processors

Modern chiplet-based Processor Architectures

- A *chiplet* is a small, specialized die (e.g., CPU core cluster, GPU, I/O) designed to work as part of a larger system.
  - Lego-like approach that let’s designers mix and match dies (even from different vendors) to build more complex SoCs cost-effectively and chiplet-independent innovation.

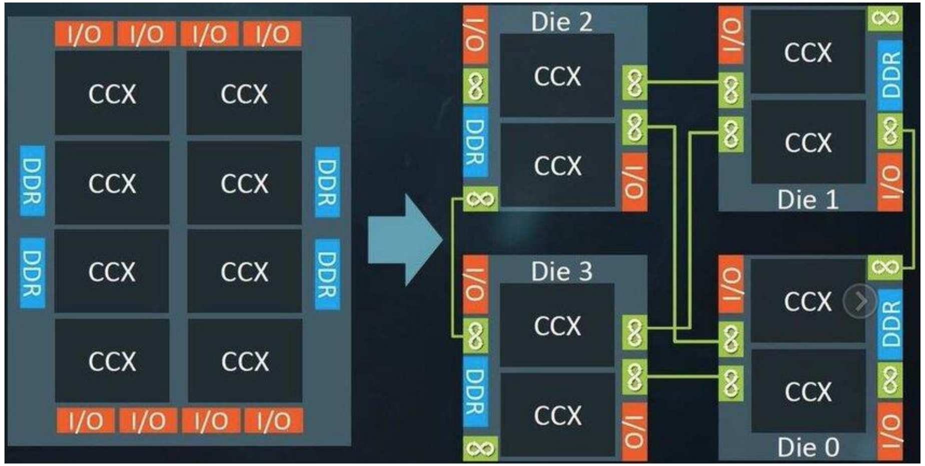
- Technology tailoring: each chiplet can use the optimal process node for its function
  - Cutting-edge 3-5nm for CPU logic
  - Mature 6-28nm for I/O
  - Higher yield → chance of defect killing a die drops with its area

- Die-to-die links:
  - High-speed serial links connect the chiplets. Often packetized (unlike on-chip buses) and are thus designed for bandwidth.
    - Examples: AMD’s Infinity Fabric, Intel’s Foveros interconnect, NVLink

Chiplets vs. Monolithic Processors

- A ***monolithic CPU/GPU*** places ***all*** logic on one large die
- A ***chiplet*** splits functions into ***separate*** dies.

https://www.researchgate.net/figure/Monolithic-Dieleft-vs-Chipletsright_fig4_376192917

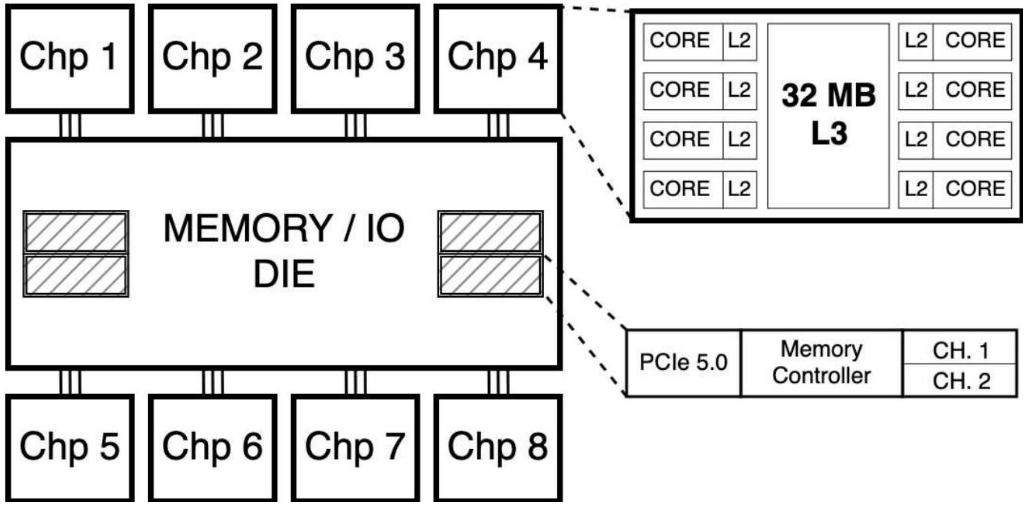


- Pros chiplets:
  - Higher yield, lower per-chip wafer cost
  - Better scalability to many cores/dies
    - E.g., AMD scaled EPYC from 8-core to 192-cores by adding chiplets (otherwise impractical)
- Cons chiplets:
  - Inter-chiplet communication is slower

- **Performance/power trade-off**
  - Monolithic chips achieve low latency for tight coupling
  - Chiplets introduce latency/bandwidth penalty across the interconnect

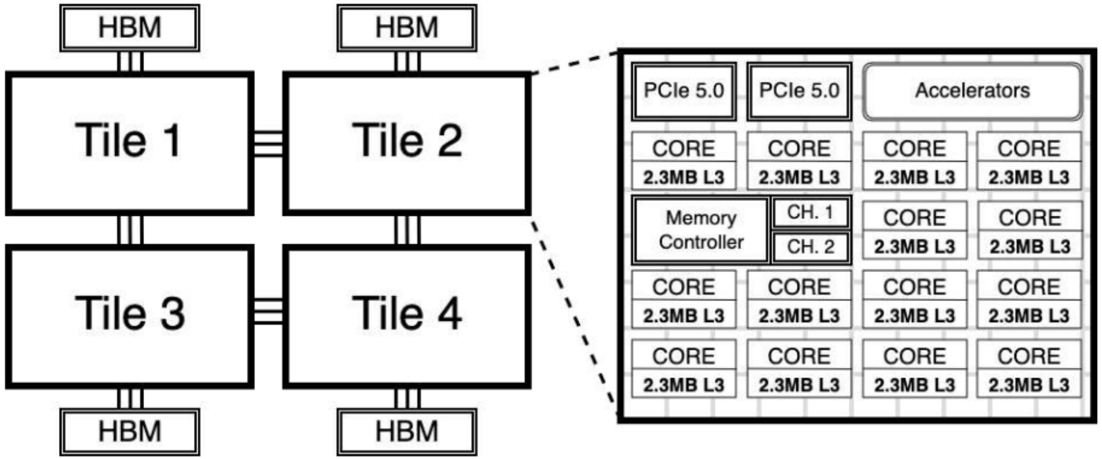
Examples

- **AMD EPYC (Zen Architecture)**
  - Use multi-chip modules chiplet architectures
  - Each Core Complex Die (CCD) is an 8-core Zen chiplet
  - A central memory I/O die (IOD) has the memory controllers, PCIe, etc.
  - CCDs are placed around the IOD and connected with the Infinity Fabric.



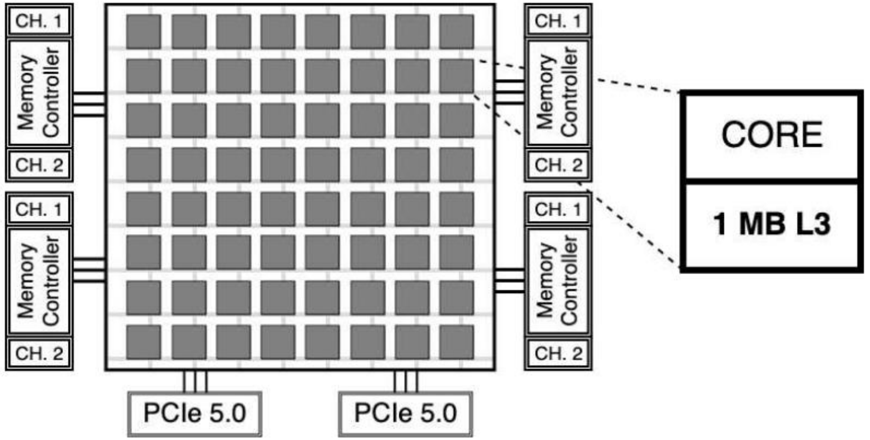
Examples

- **Intel Sapphire Rapids XCC (tile design)**
  - Four tiles per package
  - Each tile with up to 15 cores, and a single UPI link
    - Two channels for DDR5
  - Accelerators:
    - Data Streaming Accelerator
    - Quick Assist Technology
    - Dynamic Load Balancing
    - In-Memory Analytics Accelerator
  - Compute Express Link (CXL 1.1)



Other examples

- **ARM Graviton 3**
  - 3 sockets connected and managed with one Nitro card
  - Each socket CPU follows a chiplet design with 7 silicon dies
  - All 64 cores placed on a single chiplet
  - Remaining 6 chiplets handle IO, DDR5/PCIe 5.0



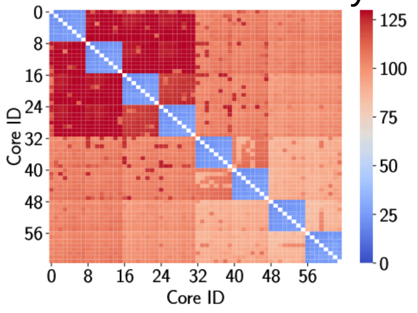
- **Apple M1 Ultra**
  - Glues two M1 Max dies side-by-side
  - Each M1 includes CPU cores, GPU cores, an NPU and a unified physical pool of DDR memory
  - The result acts like one 20-core CPU, 64-core GPU and 128 GB unified RAM.

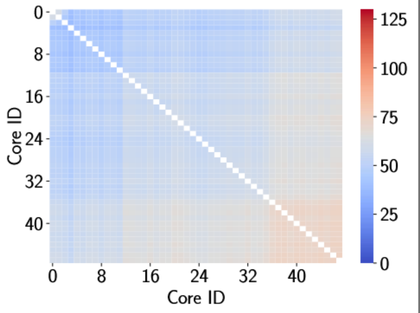
- **NVIDIA Grace Hopper (Het. Superchip)**
  - CPU+GPU Superchip
    - Grace CPU die (ARM Neoverse cores, up to 512 GB LPDDR5X 546 Gb/s), Hopper GPU die (96 GB HBM, 3000 GB/s)
    - Connected by NVLink C2C interconnect fully cache-coherent (900 GB/s).

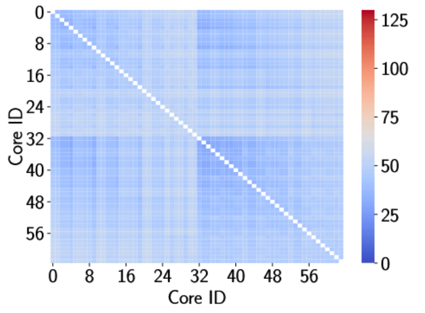
System Design effect on properties

- **NUMA-like effects:**
  - multi-chiplet designs often behave like NUMA systems even on a single socket
  - Each chiplet is a “node” with local memory (except ARM’s Graviton)
  - Accessing another chiplet’s memory adds latency and reduces bandwidth

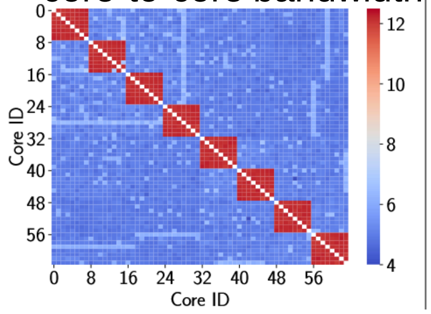
core-to-core latency

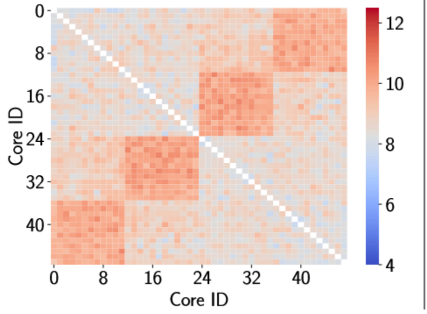


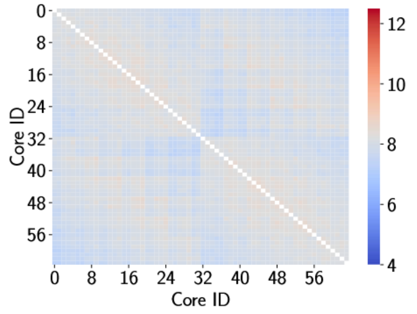




core-to-core bandwidth



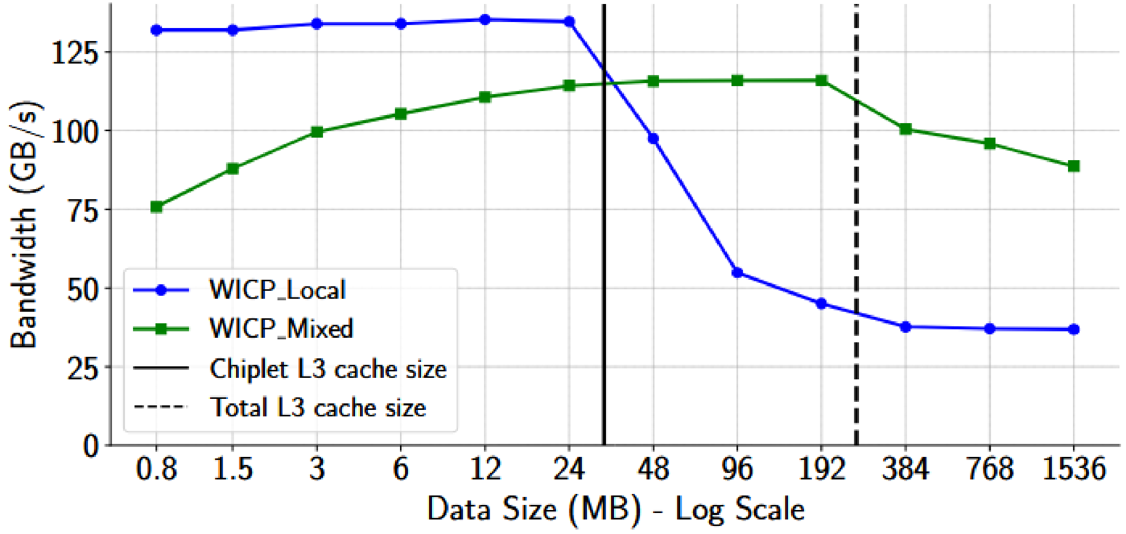




  - Q: on the right we see the behavior of three architectures: AMD, ARM and Intel. Which is which?

The effect of partitioned LLC cache

- The visible LLC does not expose that it is partitioned among the chiplets
- Yet, it can have a significant effect on our program’s performance



As long as the working set size fits the L3 chiplet portion, keep the threads working together on that chiplet.

Beware of the cliff – locality brings performance but overoptimizing may hurt us once the working set exceeds the threshold. Then spreading is better.

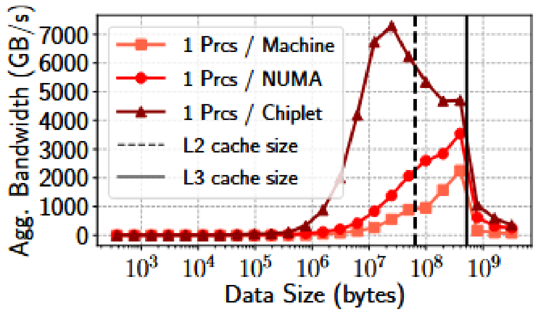
Is it the same across architectures?

- Our experience shows that there is a performance difference between architectures, but the principles apply.

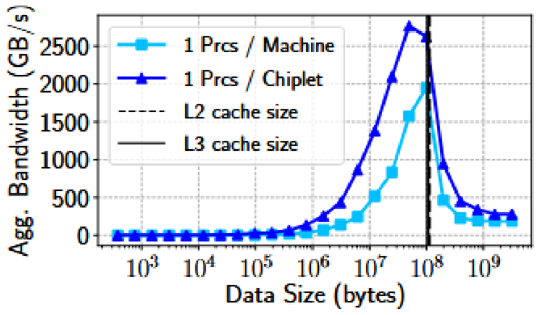
AMD has more chiplets and achieves higher aggregate bandwidth.

For Intel, the difference is not as obvious – expect less speed-up, but also the aggregate bandwidth is considerably lower.

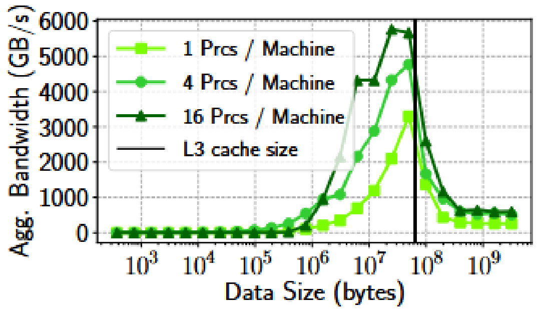
Chiplet-awareness has no effect on monolithic systems.



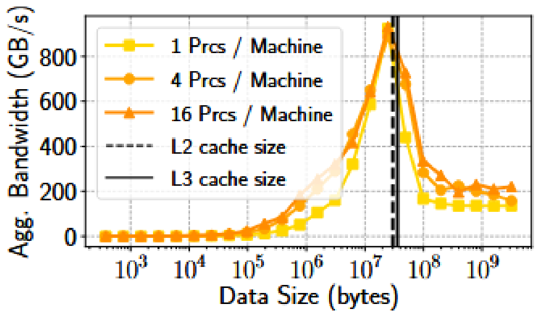
(a) AMD EPYC Milan



(b) Intel Sapphire Rapids



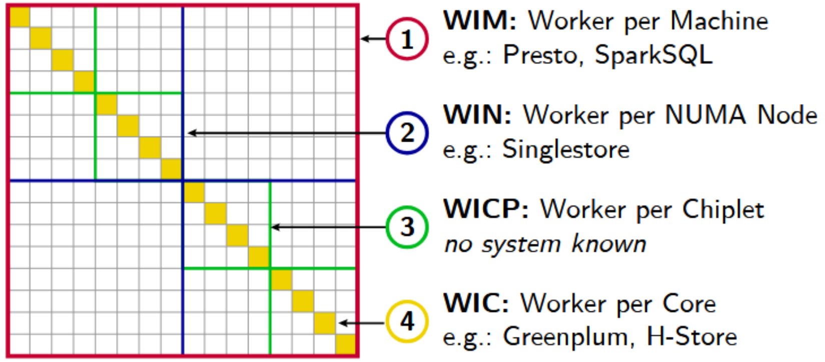
(c) ARM Graviton 3



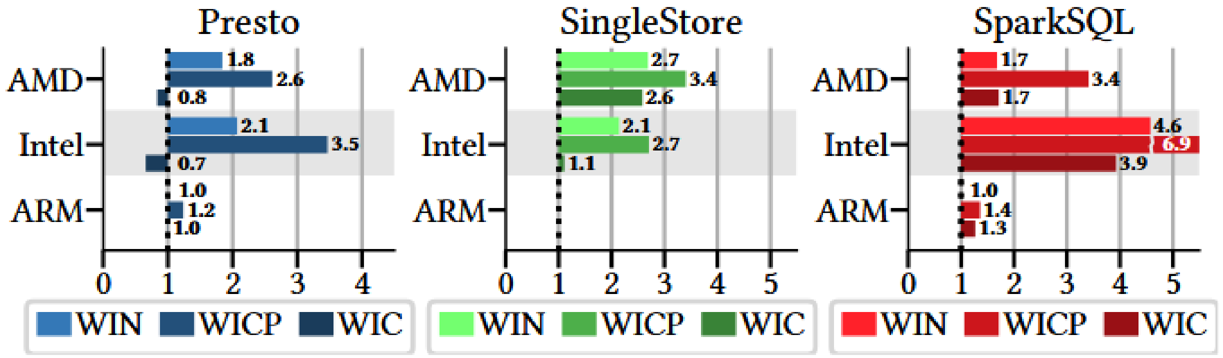
(d) 3rd Gen. Intel Xeon Gold

Chiplet-aware scheduling

- **OLAP on modern Chiplet-based Processors**
  - Default: one worker per machine
  - Optimized: one worker per NUMA
  - Some can't leverage parallel execution:
    - One worker per core
  - Not explored: one worker per chiplet

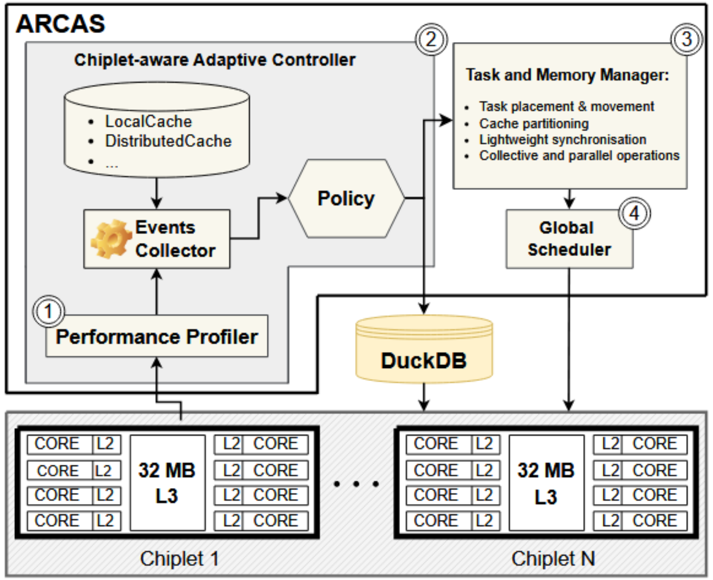


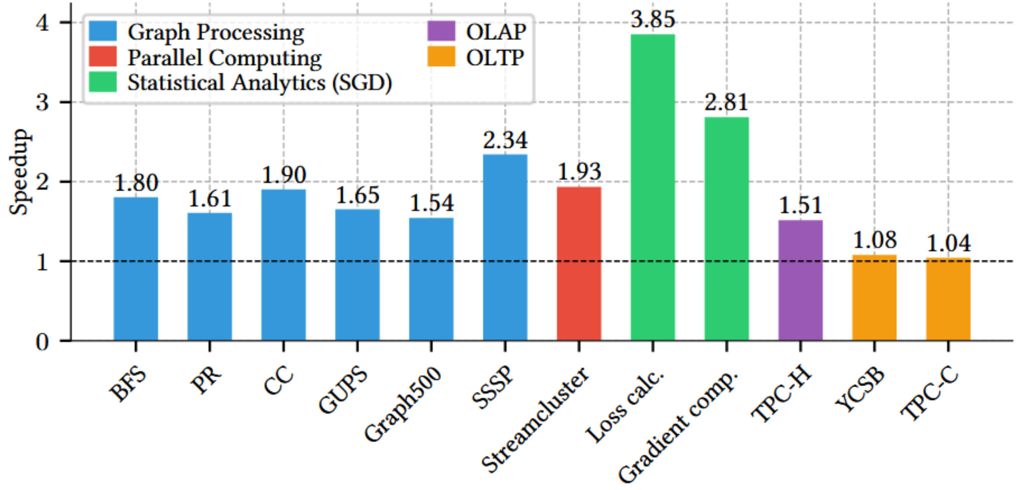
- **Results** are quite surprising



Chiplet-aware scheduling

- **Q: Does it generalize beyond analytics?**
- Yes, and we can even make it adaptive.





- Light-weight threads – co-routines
- Task-based scheduling
- Light-weight monitoring on local/remote chiplet accesses
- Runtime adaptive migration of tasks to workers, depending on the working set size

Src: "Fogli et al. ARCAS: Adaptive Runtime System for Chiplet-Aware Scheduling" *EuroSys revision 2025/2026*

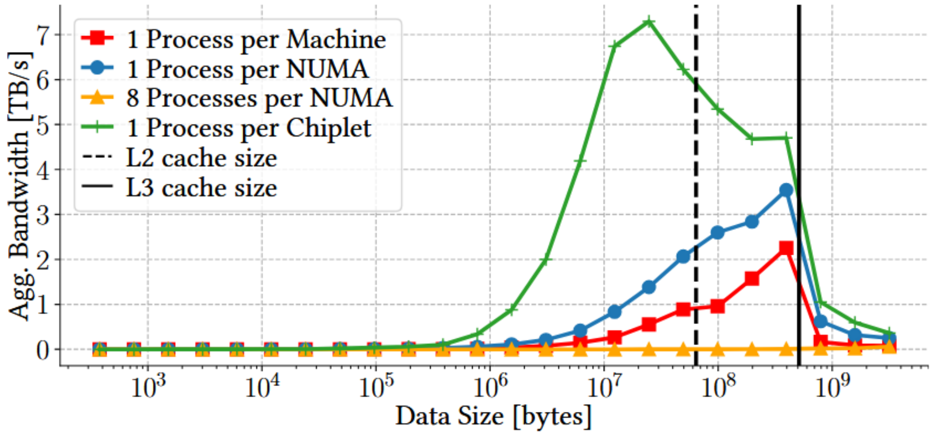
Does it affect how we design algorithms?

- Let’s look at a **sorting**

  – **Cache-conscious sort**
    – In-register handles runs that fit within the SIMD
    – In-cache includes runs that are confined within the CPU’s L3 cache
    – Out-of-cache is for runs that exceed the cache capacity.

  – **Chiplets now partition the L3 cache and thus require rethinking the in-cache sorting:**
    – In-local-chiplet-cache vs. in-remote-chiplet cache.

  – **What about the aggregate bandwidth?**



From NUMA to chiplet-aware sorting

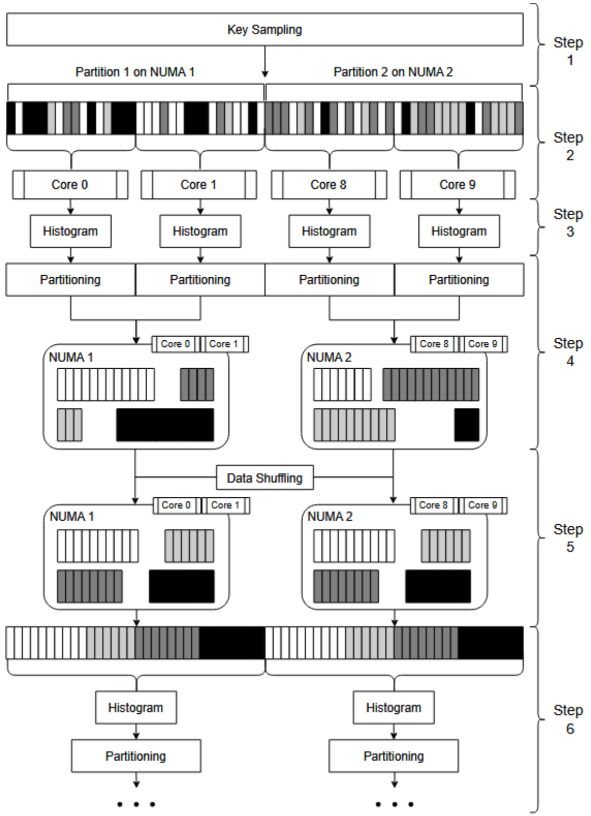


Fig. 5: NUMA-aware LSB Radix-Sort step by step execution.

Step 1: from NUMA-balanced partitions to degree of parallelism

Step 2: bind each part to a NUMA node. Now, bind each thread to a chiplet. Rely on chiplet-aware task-scheduling.

Step 3: unchanged

Step 4-5: avoid the NUMA data shuffling.

Step 6: the process is repeated based on the sorting bits.

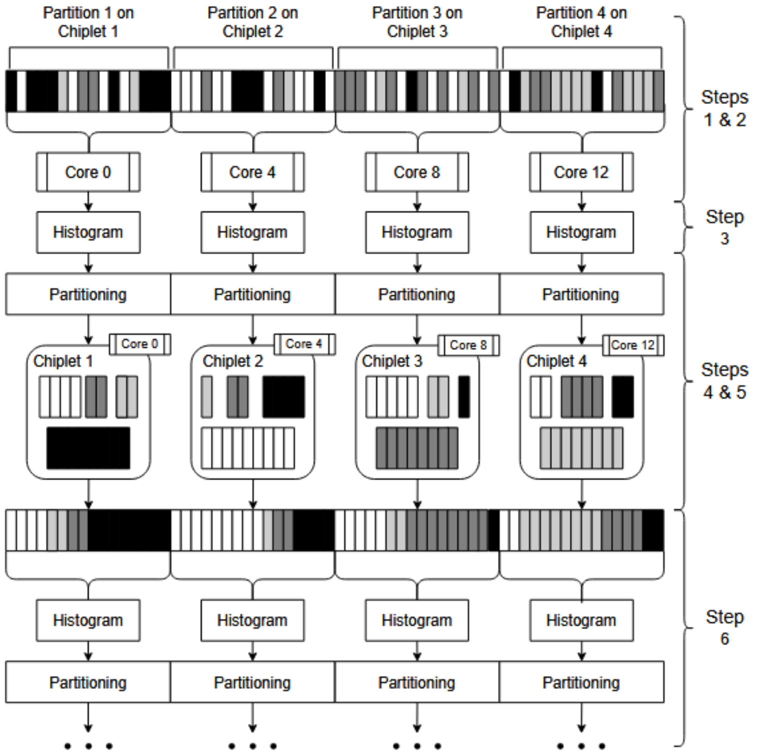
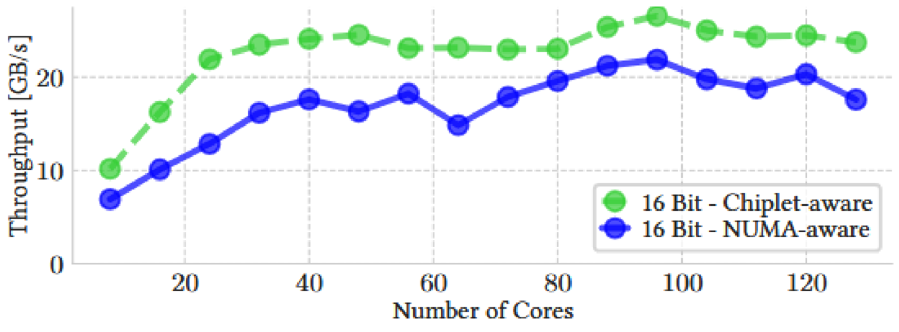


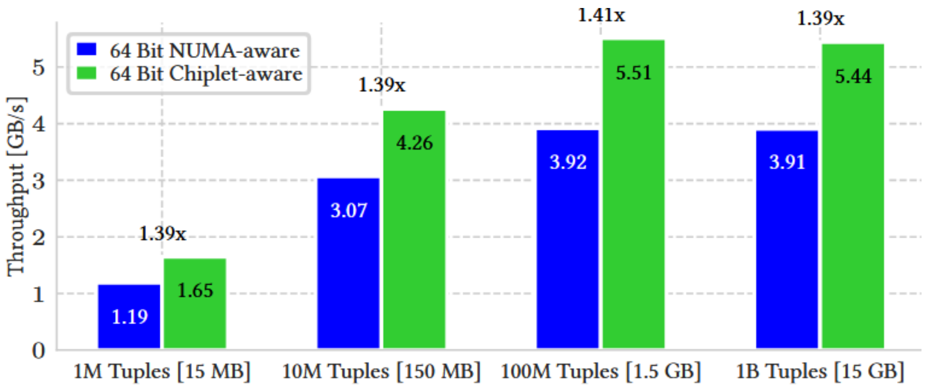
Fig. 6: Chiplet-aware LSB Radix-Sort step by step execution.

Src: "Fogli et al. Optimizing Sorting for Chiplet-Based CPUs" ADMS@*VLDB 2024*

Performance impact on Radix-sort



(a) 16 Bit Radix-Sort, 100M Tuples



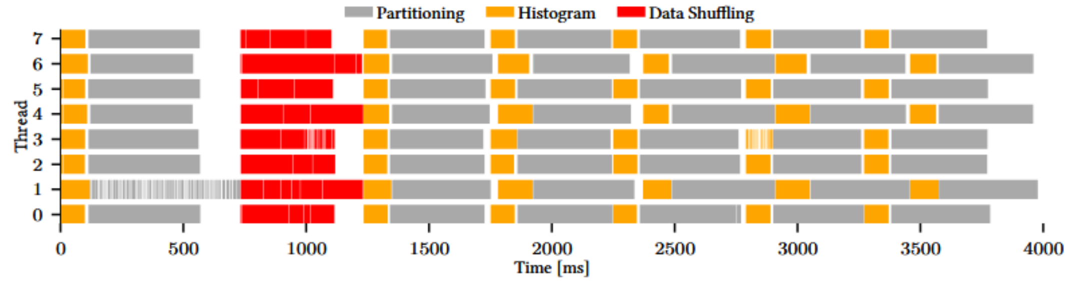
(a) Radix-Sort

- Key take-away:
  - Chiplet-aware sorting gives a significant boost to NUMA-optimized alternatives
  - One observes approximately 40% performance improvements.

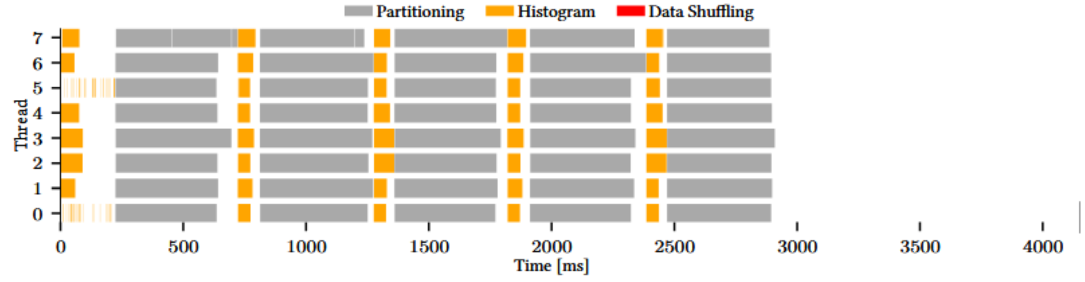
- Similar observations on various dataset sizes, 16-,32-,64-bit keys, and skew distributions.

Why the performance boost?

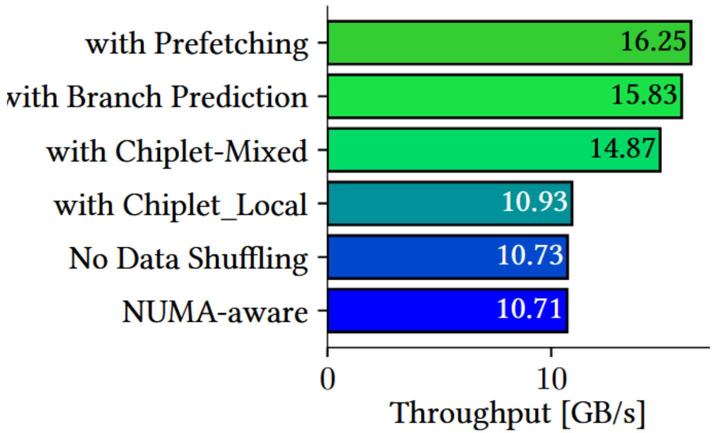
- Party by removing the data shuffling phase between the NUMA nodes.
- Partly by adjusting the number of partitioning phases by looking at the data size.
- And finally, by adjusting between chiplet-local vs. mixed on aggregate bandwidth.



(a) NUMA-aware LSB Radix-Sort



(b) Chiplet-aware LSB Radix-Sort



References

- Various papers cross-referenced in the slides
  - Li et al. “NUMA-aware algorithsm: the case of data shuffling” *CIDR 2013*
  - Balkesen et al. “Multi-core, Main-Memory Joins: Sort vs Hash Revisited” *VLDB 2014*
  - Schul et al. “An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory.” *SIGMOD 2016*
  - Leis et al. “Morse-Driven Parallelism: A NUMA-aware query evaluation framework for the many-core age” *SIGMOD 2014*
  - Porobic et al. “Analyzing the impact of system architecture on the scalability of OLTP engines for high-contention workloads” *VLDB 2017*
  - Lee et al. “MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases” *VLDB 2009*
  - Lo et al. “Heracles: Improving Resource Efficiency at Scale” *ISCA 2015*
  - Makreshanski et al. “BatchDB: Efficient Isolated Execution of Hybrid OLTP and OLAP workloads for Interactive Applications” *SIGMOD’17*

- Lecture: *Data Processing on Modern Hardware* by Prof. Jens Teubner (TU Dortmund, past ETH)

- Book: *What every programmer should know about memory?* by Ulrich Drepper
  - Chapters 5 and 6.5

- Intel Architectures Software Developer Manuals
  - Optimizing Applications for NUMA (https://software.intel.com/content/dam/develop/external/us/en/documents/3-5-memmgt-optimizing-applications-for-numa-184398.pdf)
  - Volume 3b: chapters 17.16 and 17.16 (for Intel RDT)