Data Processing on Modern Hardware

Jana Giceva

Lecture 8: Multicore CPUs
NUMA, interference and isolation
Non-uniform memory access (NUMA)
Distributed Shared Memory

- Single processor scalability (with shared memory) has limitations
- Idea: distribute memory
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 24 cores (up to 64 cores on AMD EPYC)
- 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 algorithm: 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 algorithm: 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
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.

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

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

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-conscious way. Recall the multi-way sort when we did SIMD?

Sorting and NUMA

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

Radix-join and NUMA
Radix-join and NUMA

NUMA-awareness brings additional 20-30% improvement.

- 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.

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

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 each
- **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
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 (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]

Latest generation hardware

- Intel scalable with UltraPath interconnect
- Succeeds Intel QuickPath Interconnect (QPI)
- Can connect each processor with up to 3 UPI links for connecting to other Intel Xeon processor.
- UPI uses a directory-based home snoop coherency protocol, operational speed of up to 10.4 GT/s
- Between 2- and 8-socket configurations
Performance isolation
Concurrency does not only affect correctness and hence the need for efficient synchronization.

The impact of resource sharing must not be overlooked:
- *e.g.*, OLAP + OLAP *or* OLAP + OLTP

Challenges of multi-programming (concurrency) due to interference:
1. Restructuring the algorithm (less sensitive to noisy environment)
2. Careful co-scheduling (victim and noisy neighbors)
3. Isolation through pinning and running on a separate NUMA node
4. Isolation with cache partitioning
Execution on Multiple cores

- Recall the example that we presented in the introductory lecture

- **Task:** run parallel instances of the query

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

- To implement the join use either
  - a **hash join** or
  - an **index nested loops join**

- Co-execute the independent instances on different CPUs and compare the performance to baseline when they are run in isolation.
Concurrent queries may seriously affect each other's performance.

Some algorithms are more sensitive to noisy environments (victims) and their performance can be significantly affected if collocated with a bad neighbor.

More cores share the last-level cache (LLC)

The problem we saw in the previous slide is cache pollution
  - How can we avoid it?
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 loop joins, hash joins with very small or very large hash tables

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

- **Strong locality**
  - Small data structure; reused very frequently
  - *e.g.*, a small hash table

- **Moderate locality**
  - Frequently reused data structure; data structure $\sim$ 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
Locality effects how caches are used:

<table>
<thead>
<tr>
<th>Cache pollution</th>
<th>strong</th>
<th>moderate</th>
<th>weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount of cache used</td>
<td>small</td>
<td>large</td>
<td>large</td>
</tr>
<tr>
<td>amount of cache needed</td>
<td>small</td>
<td>large</td>
<td>small</td>
</tr>
</tbody>
</table>

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

Impact of co-runner on query:

<table>
<thead>
<tr>
<th>query / co-runner</th>
<th>strong</th>
<th>moderate</th>
<th>weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>strong</td>
<td>low</td>
<td>moderate</td>
<td>high</td>
</tr>
<tr>
<td>moderate</td>
<td>moderate</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>weak</td>
<td>low</td>
<td>low</td>
<td>low</td>
</tr>
</tbody>
</table>
Hash join is the only algorithm sensitive to sharing the caches.

The index join is not affected, regardless of the co-runner query.
Locality-aware scheduling

An optimizer could use knowledge about localities to schedule queries:

- **Estimate** locality during query analysis
  - Index nested loop join → weak locality
  - Hash join:
    - Hash table \( \ll \) cache size → strong locality
    - Hash table \( \approx \) cache size → moderate locality
    - Hash table \( \gg \) cache size → 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 alongside other weak-locality queries.
  - They cause high pollution, but are not affected by pollution.
- Try to co-schedule queries with small hash tables.
Locality-aware scheduling

- PostgreSQL
- 4 queries (different p_categories); for each query:
  - 2 x hash join,
  - 2 x INLJ;
- Performance impact reported for the hash joins

Cache pollution

- Weak-locality plans cause cache pollution, because they use much cache space even though they do not strictly need it or benefit from it.
- By partitioning the cache we could reduce the pollution with little impact on the weak locality plan.
In the past, people had to rely on page coloring to achieve cache partitioning from the software side.

- The address <-> cache set relationship inspired the idea of page colors.

Today, Intel provides the Resource Directory Technology (RDT)
- Cache Monitoring and Allocation Technology (CMT and CAT)
- CAT is a software programmable control over the space that can be consumed by a given thread, application, virtual machine (VM), or a container.
Class of service (CLOS) or an application priority class
- resource control tag that allows us to group threads or applications.

Associate the CLOS with resource capacity bitmasks (CBMs) indicating how much of the cache can be used by a given CLOS.
- The CBMs indicate the relative amount of cache available, the degree of overlap or isolation.

Can be further refined with code and data-prioritization (CDP) technology.

CLOS[1] has less cache available than CLOS[3], even though it has higher priority.
CLOS[2] and CLOS[3] have overlapping bitmasks, can achieve higher throughput than in isolation, but relative priorities will be preserved.

src: Intel
Experiments: MCC-DB with page coloring

- PostgreSQL
- 4 queries (different p_categories); for each query:
  - 2 x hash join,
  - 2 x INLJ;

- Performance impact reported for the hash joins

![Graph showing performance impact for different hash table sizes.]

But, it is not only the cache is shared

Impact of interference on transactions

- **Goal:** Measure the performance impact that a local NUMA scan can have on an OLTP workload.

- **No explicit sharing of resources**
  - the scan runs on a separate dataset and in a separate process from the OLTP workload.

- **Set-up:** 5/10 cores allocated to the OLTP process and measure its performance when:
  - runs alone (in isolation, no interference)
  - runs co-located with the bandwidth intensive scan running on the other 5 cores (i.e., local NUMA scan)
  - the scan runs on 5 cores on another CPU and reads data locally (i.e., remote NUMA scan)

![Graph showing throughput comparison](image)

*src: Makreshanski et al. “BatchDB: Efficient Isolated Execution of Hybrid OLTP and OLAP workloads for Interactive Applications” SIGMOD’17*
Bandwidth allocation and partitioning

- In concurrent data processing workloads (and complex data center and enterprise deployments), we can easily get memory-bound (e.g., bottlenecked on the memory bandwidth).

- Need to ensure that the performance critical tasks (e.g., OLTP transactions) still meet their SLAs.

- New addition to Intel’s RDT is Memory Bandwidth Allocation (in Intel Xeon Scalable processors), which extends the CAT
  - Also groups threads and applications into CLOS
  - Throttles them based on priorities

References

- Various papers cross-referenced in the slides
  - Li et al. “NUMA-aware algorithm: 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
  - 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
  - Volume 3b: chapters 17.16 and 17.16 (for Intel RDT)