Data Processing on Modern Hardware

Jana Giceva

Lecture 7: Multicore CPUs
Parallelization and Synchronization
To make the most out of multicore processors we can:

- Allow multiple different tasks to be running concurrently → **concurrency** *(multiprogramming)*
- Parallelize the implementation of a single task → **parallelism** *(parallel programming)*
Parallelism
Basic concepts

- **Work partitioning** (expressing parallelism)
  - Work must be split into **parallel tasks**
  - Also known as domain decomposition

- **Scheduling**
  - Tasks must be mapped into execution contexts

- **Task granularity**
  - How much work a task performs?
  - Too little $\rightarrow$ large overhead
  - Too much $\rightarrow$ difficult for efficient load balancing

- **Correctness**
  - Order of reads and writes is non-deterministic
  - **Synchronization** is required to enforce the order
An overloaded concept:
- *e.g.*, how well a system reacts to increased load, *e.g.*, clients in a server

**Speed-up** – how well does the RT reduces for the same problem size by adding resources (*e.g.*, cores).
- Speed up for problem size $X$ with $N$ resources: $SpeedUp(N) = RT(1, X)/RT(N, X)$
- *Ideal*: linear function

**Scale-up** – how well the system deals with larger load (problem size) by adding resources
- Scale up for $N \times$ larger problem by adding $N \times$ resources: $ScaleUp(N) = RT(1, X)/RT(N, NX)$
- *Ideal*: constant function

**Scale-out** – how well the system deals with larger load (problem size) by adding more servers / machines
- Scale out for $N \times$ larger problem by executing on $N \times$ machines: $ScaleOut(N) = TP(1, x)/TP(N, NX)$
- *Ideal*: constant function (should behave like Scale-up)
Our focus: speed-up

- Sequential execution time: $T_1$
- Execution time $T_p$ on $p$ CPUs
- (parallel) speed-up $S_p$ on $p$ CPUs: $S_p = \frac{T_1}{T_p}$
  - $S_p = p$: linear speed-up
  - $S_p < p$: sub-linear speed-up / performance loss
  - $S_p > p$: super-linear speed-up / usually poor baseline

- Why $S_p < p$?
  - Programs may not contain enough parallelism
    - Some parts may be inherently sequential
  - Overheads due to parallelization
    - Typically associated with synchronization
  - Architectural limitations
    - Memory contention (memory bound)
Suppose we parallelize an algorithm using $n$ cores and $p$ is the proportion of the task that can be parallelized ($1 - p$ cannot be parallelized)

- The speed up of the algorithm is $\frac{1}{(1-p)+\frac{p}{n}}$
- For infinite parallelism, the speed-up is $\frac{1}{1-p}$
- For example, if 90% of the work is parallelized, the maximum speed up is 10
- Ensure that every phase of one’s algorithm that depends on the input data size is parallelized.

Amdahl’s Law

![Amdahl's Law Graph](https://via.placeholder.com/150?text=Image+Source+Wikipedia)
Pitfalls in parallel code

- **Non-scalable algorithm**
  - Rethink the algorithm
  - e.g., searching a tree: which one is easier to parallelize BFS or DFS?

- **Load imbalance**
  - Break work into smaller tasks, dynamically schedule these between threads

- **Task overhead**
  - Set a minimum per-thread task size (not too small, not too large)
Parallelize database workloads

In database systems:

- **Inter-query parallelism** (Concurrency, Multi-programming)
  - Requires a sufficient number of **co-running queries**.
  - May work well for **OLTP workloads**
    - Characterized by many simple queries
  - Data analytics / OLAP are resource-heavy
  - Will not help an individual query

- **Intra-query parallelism**
  - Intra-query parallelism is a must
  - Should still allow a few **co-running queries**.
Processes, kernel- and user-level threads and fibers

**Process:** an instance of a program that is isolated from other processes on the machine.
- Has its own private section of the machine’s memory.
- A process abstraction is a virtual computer. Scheduled by the kernel.

**Thread:** a locus of control inside a running program.
- A thread abstraction is a virtual processor. Scheduled by the kernel.
- Threads share all the memory in the process.

**User-level threads:** act like threads, but implemented in user-space.
- Can be scheduled preemptively or cooperatively. Invisible to the kernel.

**Fibers:** light-weight thread of execution that uses co-operative multi-tasking.
- Fibers yield themselves to run another fiber while executing.
Process model in databases

- **OS Process per DBMS worker**
  - Used by early DBMS implementations
  - DBMS workers are mapped directly onto OS processes

- **OS Thread per DBMS worker**
  - Single multi-threaded processes hosts all DBMS worker activity
  - A dispatcher thread listens for new connections. Each connection is allocated a new thread.

- **DBMS Threads**
  - Lightweight user-space threading constructs (replacing the need for OS threads)
  - Fast task switching at the expense of replicating a good deal of the OS logic in the DBMS
    - Task-switching, thread state management, scheduling, etc.

- **Are co-routines (fibers) next?**
Parallelize database workloads

In database systems:

- **Inter-query parallelism** (Concurrency, Multi-programming)
  - Requires a sufficient number of **co-running queries**.
  - May work well for **OLTP workloads**
    - Characterized by many simple queries
  - Data analytics / OLAP are resource-heavy
  - Will not help an individual query

- **Intra-query parallelism**
  - Intra-query parallelism is a must
  - Should still allow a few **co-running queries**.
Parallelization strategies for **intra-query parallelism:**

- Pipeline parallelism?

- Data partitioning / parallel operator implementation?
Volcano-style parallelism

Goal: Parallelize the query engine in a clean, uniform way.

Volcano's Solution: encapsulate the parallelism in a query operator of its own, not in the QP infrastructure.

Overview: kinds of intra-query parallelism available:
- pipeline
- partition, with two subcases:
  - intra-operator parallelism (e.g. parallel hash join, or parallel sort)
  - inter-operator parallelism -- bushy trees

We want to enable all -- including setup, teardown, and runtime logic -- in a clean encapsulated way.

The exchange operator:
- an operator you pop into any single-site dataflow graph as desired -- anonymous to the other operators.

src: Graefe. Volcano – An Extensible and Parallel Query Evaluation System. IEEE Transactions on Knowledge and Data Engineering 1994
Volcano-style parallelism

- Plan-driven approach:
  - Optimizer determines at compile time the degree of parallelism
  - Instantiates one query operator plan for each thread
  - Connects these with exchange operators, which encapsulate parallelism and manage threads

- Elegant model which is used by many systems

src: Graefe. Volcano – An Extensible and Parallel Query Evaluation System. IEEE Transactions on Knowledge and Data Engineering 1994
Volcano-style parallelism

- **Positive** aspects:
  - Operators are largely oblivious to parallelism

- **Drawbacks**:
  - Static work partitioning can cause **load imbalance**
  - **Degree of parallelism** cannot be easily change mid-query
  - **Potential overhead**:
    - Thread over-subscription causes context switching
    - Exchange operators create additional copies of the tuples

---

*src: Graefe. Volcano – An Extensible and Parallel Query Evaluation System. IEEE Transactions on Knowledge and Data Engineering 1994*
Parallelism in Modern DBMSs today

- **Query coordinator** manages the parallel execution
  - Obtains the number of parallel servers
  - Determines granularity of partitioning and load-distribution

- **Parallelism within and between operators**
  - Pipeline with depth 2 (producer – consumer pair)
  - *e.g.*, parallel scan and group-by uses 8 servers in total.

- **DOP (degree of parallelism)** – the number of parallel execution servers associated with a single operator
  - Can be chosen **manually** or **automatic**
  - **Adaptive** means it can reduce the DOP as the load in the system increases

Source: https://docs.oracle.com/cd/E11882_01/server.112/e25523/parallel002.htm
Parallelizing the radix join

- Use the **task queuing model** that decomposes the execution into **parallel tasks**, each executing a fraction of the total work.
- The runtime system can then **dynamically** schedule the tasks on different **hardware threads** $T$.

**General guidelines:**
- Create more tasks than there are threads
- If a task’s input size exceeds a threshold (e.g., due to skew):
  - Further split it up or if not possible put it aside and handle it afterwards
  - Ensure to have good load-balancing among the hardware threads.

More details for the specific stages of the join in *Sort vs Hash Revisited: Fast Join Implementations on Modern Multicore CPUs* by Kim et al. (VLDB 2009)
Impact of task granularity on parallel operators

- Different stages in radix join:
  - 1 – 2: compute local histogram for R and S
  - 3 – 4: partitioning passes 1 and 2
  - 5: join phase (partition-wise build and probe)

- Evaluate the effect of task granularity and queuing on the performance of the radix join (zipf 1.5)
  - Left – simple task queuing
  - Right – task decomposition for large part/join tasks

- All threads do useful work in the beginning of each execution stage (busy time with different gray shades)

- Simple task queuing leads to poor load-balancing and threads need to wait on barriers → 25% perf. reduction

- With fine-grained task decomposition, we can identify the large tasks and break them down for good load balancing among all the working threads.

[Diagram showing time [billion cycles] vs. thread id with different tasks]

src: Balkesen et al.: Main Memory Hash Joins on Multi-core CPUs: Tuning to the Underlying Hardware. ICDE 2013
Lessons learned:

- Use **fine-grained partitioning**
  - Increased scheduling overhead seems bearable
- Assign partitions / tasks **dynamically** to processors
  - Make load balancing easier

- How to incorporate that at an engine level?
  - Morsel-driven parallelism (as implemented in HyPer)
Morsel-driven query execution

- Example of user-level task-based parallelism as framework in database systems.

- Break input data into **constant-sized work units** (“morsels”)

- **Dispatcher** assigns morsels and a pipeline (of operators) to worker threads (**scheduling**)

- Number of worker threads = number of hardware threads

- Operators are **designed for parallel execution**

---

**Figure 1:** Idea of morsel-driven parallelism: $R \bowtie_A S \bowtie_B T$

---

src: Leis et al. Morsel-driven Parallelism: A NUMA-aware query evaluation framework for the many-core age. SIGMOD 2014
Each pipeline is parallelized individually using all threads.

src: Leis et al. Morsel-driven Parallelism: A NUMA-aware query evaluation framework for the many-core age. SIGMOD 2014
Each pipeline is parallelized individually using all threads

src: Leis et al. Morsel-driven Parallelism: A NUMA-aware query evaluation framework for the many-core age. SIGMOD 2014
Concurrency and Synchronization
Concurrence in database workloads

Databases are often faced with highly concurrent workloads.

Good news:
- Exploit parallelism offered by the hardware (increasing number of cores)

Bad news:
- Increases relevance of synchronization mechanisms.
Synchronization in databases

Two levels of synchronization in databases:

- Synchronize on **user data** to guarantee **transactional semantics**:
  - database terminology: **locks**

- Synchronize on **database-internal data structures**
  - database terminology: **latches**

We will **focus** on the latter (**latches**), even when we refer to them as locks.
Cache coherence

- Cores have **private caches**
- CPU manages the shared memory and private caches using a cache coherency protocol

**Cache coherency protocol** ensures the **consistency** of **data in caches**

- Implements the two fundamental operations: **load** and **store** using:
  - **Snooping-based** coherence
    - All processors communicate to agree on the state
  - **Directory-based** coherence
    - A centralized directory holds information about state/whereabouts of data items
Most contemporary processors use the MESI cache coherency protocol (or a variant).

MESI protocol has the following states:
- **Modified**: cache line is only in current cache and has been modified
- **Exclusive**: cache line is only in current cache and has not been modified
- **Shared**: cache line is in multiple caches
- **Invalid**: cache line is unused

Intel uses the MESIF protocol, with an additional Forward state
- Special shared state indicating a designated “responder”
Atomics

- x86 provides a `lock` prefix that tells the hardware:
  - Do not let anyone read / write the value until I am done with it
  - Not the default case (because it is slow!)

- Compare-and-swap (CAS):
  - `lock cmpxchg`

- Exchange:
  - `xchg` (automatically locks the bus)

- Read-modify-write:
  - `lock add`

- If the compiler (or you) also emit code using non-temporal stores, it must also emit sufficient fencing to make the usage of non-temporal stores un-observable to callers/callees.
  - `_mm_mfence()`, `_mm_lfence()`, `_mm_sfence()`
There are different synchronization modes:

- **Pessimistic** locking
  - Always take an (exclusive) lock to access/modify data in the critical section

- **Optimistic** locking
  - Validate whether the data read in the critical section is still valid upon completion

- Lock-free
  - Threads never block for any reason when reading or writing
  - Leverage HW-support for synchronization (atomics)

- Speculative locking (hardware transactional memory (HTM))
There are many different types of locks (we only look at a subset)

**Pessimistic:**
- *Exclusive* lock
  - Only one thread may hold the lock at a time
- *Shared* (Reader-Writer RW) lock
  - Permit any number of readers to hold the lock concurrently
  - Only allow a single writer to hold the lock

**Optimistic:**
- Validate that the data read in the critical section has not changed
### Optimistic locking

- Validate that the data read in the critical section has not changed in the meantime
- Good for frequently read data
  - avoids the expensive atomic writes required by pessimistic lock
  - cache invalidation only needed on writes
- Challenges:
  - Use it when it is safe to fail and restart
  - All operations must be restart-able w/o side-effects
  - With too much write contention, could lead to starvation

```c
void readOptimistically(Lambda& readCallback){
    // Attempt to read optimistically
    for(i in [1 : MAX_ATTEMPTS]){\n        preVersion = getVersion();
        if(isLocked(preVersion()))
            continue;
        readCallback();
        postVersion = getVersion();
        if(preVersion == postVersion)
            return;
    }
    // Fallback to pessimistic locking
    lockPessimistic();
    readCallback();
    unlock();
}
```
Lock (latch) implementation

There are two strategies to implement (pessimistic) locking:

- **Spinning** (in user space) – *e.g.*, spinlock
  - Waiting thread repeatedly polls lock until it becomes free
  - But, the thread burns CPU cycles while sleeping
  - Cost two cache miss penalties (if implemented well) → 150nsec

- **Blocking** (OS service) – *e.g.*, mutex or user-space futex
  - De-schedule the waiting thread until the lock becomes free
  - Cost: two context switches (one to sleep, one to wake-up) → 12-20usec
Most database workloads mostly read data (even OLTP workloads)
- Reading should be fast and scalable

For tree-based data structures (e.g., indexes), we always need to traverse the top levels of the tree
- High contention on such hotspots – should be lockable with minimal overhead

Latency is critical
- Avoid context switching as much as possible → cannot solely rely on OS-based locks

Some fine-grain data like index nodes or hash buckets requires space efficient locks
- Standard mutex (std::mutex) can be as much as 40-80 bytes – double the size for an ART node

Efficient contention handling
- Handle contention gracefully, without sacrificing the uncontented path
### Qualitative overview of locking modes

- Which locking mode is best for a certain type of workload?
  - **Workloads**: read-only, read-mostly (big/small read-set), write-heavy, write-only
  - **Locking modes**: pessimistic (exclusive, shared), optimistic

<table>
<thead>
<tr>
<th>Workload Type</th>
<th>Exclusive</th>
<th>Shared</th>
<th>Optimistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-Only</td>
<td>Too restrictive</td>
<td>“Read-Read Contention”</td>
<td>No Overhead</td>
</tr>
<tr>
<td>Read-Mostly: cheap reads</td>
<td>Too restrictive</td>
<td>Still some contention</td>
<td>Restarts unlikely and cheap</td>
</tr>
<tr>
<td>Read-Mostly: big read set</td>
<td>Too restrictive</td>
<td>Lock overhead diminishes</td>
<td>Restarts can be expensive</td>
</tr>
<tr>
<td>Write-Heavy</td>
<td>Restrictive</td>
<td>Good</td>
<td>Many Aborts/Starvation</td>
</tr>
<tr>
<td>Write-Only</td>
<td>Equally good (all writes are locked exclusively)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*src: Bottcher et al. Scalable and Robust Latches for Database Systems. DaMoN 2020*
There are many different types of locks (we only look at a subset)

Pessimistic:
- Exclusive lock
  - Only one thread may hold the lock at a time
- Shared (Reader-Writer RW) lock
  - Permit any number of readers to hold the lock concurrently
  - Only allow a single writer to hold the lock

Optimistic:
- Validate that the data read in the critical section has not changed

Hybrid:
- Extend a shared lock with support for optimistic locking
Hybrid locking

Class HybridLock {
    RWMutex rwLock;
    std::atomic<uint64_t> version;

    public:
    // simply call rwLock
    void lockShared(); {rwLock.lockShared();}
    void unlockShared(); {rwLock.unlockShared();}
    void lockExclusive(); {rwLock.lockExclusive();}

    // always increment the version before
    // unlocking to avoid races!
    void unlockExclusive() {
        ++version; rwLock.unlockExclusive();
    }

    bool tryReadOptimistically(Lambda& readCallback) {
        if(rwLock.isLockedExclusive())
            return false;
        auto preVersion = version.load();
        // execute read callback
        readCallback();
        // was locked meanwhile?
        if(rwLock.isLockedExclusive())
            return false;
        // version still the same
        return preVersion == version.load();
    }

    void readOptimisticIfPossible(Lambda& readCallback) {
        if(!tryReadOptimistically(readCallback)) {
            // fallback to pessimistic locking
            lockShared();
            readCallback();
            unlockShared();
        }
    }
};

src: Bottcher et al. Scalable and Robust Latches for Database Systems. DaMoN 2020
Evaluating different locks on TPC-C

- Implemented a set of different locks in the HyPer database
- Evaluate their performance using the TPC-C benchmark

![Graph showing performance of different locks vs number of threads]

(a) TPC-C – Increasing the number of threads (100 warehouses)

src: Bottcher et al. Scalable and Robust Latches for Database Systems. DaMoN 2020
Granularity of locking

- The number of tuples protected by the lock can have a big impact on the system’s performance.

- For point accesses like updates and key look-ups, the granularity sets the number of concurrent accesses.
  - Fine granularity is good for write-heavy workloads
  - Coarse granularity is better for read-heavy workloads
  - e.g., no need to acquire a lock for every tuple during a scan

src: Bottcher et al. Scalable and Robust Latches for Database Systems. DaMoN 2020
Evaluate contention handling strategies

- How well do different contention handling strategies behave?

- Spinning
  - Naïve (test-and-set)
  - Test-test-and-set (with back-off)
  - Local spinning
  - Ticket-lock (with back-off)

- Blocking
  - std::mutex
  - ParkingLot
    - Each thread parks itself in a global hashtable (parking lot) until the callback condition is satisfied.

src: Bottcher et al. Scalable and Robust Latches for Database Systems. DaMoN 2020
Efficient implementation of concurrent data-structures
Concurrent list-based set

- Operations: insert(key), remove(key), contains(key)

- Keys are stored in a (single-)linked list, sorted by key
- head and tail are always there (“sentinel” elements)

Why atomics like CAS is sometimes not enough?
- Thread A: remove(7)
- Thread B: insert(9)
Coarse-grained locking

- Use a single lock to protect the entire data structure

- Positive:
  - Very easy to implement

- Negative:
  - Does not scale at all
Approaches to make it more scalable

- **Fine-grained locking**
  - Split object into independently synchronized components.
  - Conflict when they access the same component at the same time.

- **Optimistic synchronization**
  - Search without locking.
  - If you find it, lock and check. If OK, we are done. If not, start over (can be expensive).

- **Lazy synchronization**
  - Postpone the hard work
  - Removing components: logical removal (mark to be deleted), physical removal (do what’s needed).

- **Lock-free synchronization**
  - Don’t use locks at all. Disadvantages: complex and often with high overhead
Fine grained locking with lock coupling

- Also called **hand-over-hand locking** or **crabbing**
- Hold at most two locks at a time
- Interactive lock acquisitions / release pair-wise
- May use read/write locks to allow for concurrent readers
Lock coupling

- Also called “hand-over-hand locking” or “crabbing”
- Hold at most two locks at a time
- Interactive lock acquisitions / release pair-wise
- May use read/write locks to allow for concurrent readers
Lock coupling

- Also called “hand-over-hand locking” or “crabbing”
- Hold at most two locks at a time
- Interactive lock acquisitions / release pair-wise
- May use read/write locks to allow for concurrent readers
lock coupling

- Also called “hand-over-hand locking” or “crabbing”
- Hold at most two locks at a time
- Interactive lock acquisitions / release pair-wise
- May use read/write locks to allow for concurrent readers

- Positive:
  - Easy to implement
  - No restarts

- Negative:
  - Better than coarse-grained lock (e.g., threads can traverse in parallel), but inefficient.
- Trust, but verify
- Traverse the list optimistically without taking any locks
- Trust, but verify
- Traverse the list optimistically without taking any locks
- Lock 2 nodes (predecessor and current)
- Trust, but verify
- Traverse the list optimistically without taking any locks
- Lock 2 nodes (predecessor and current)
- Validate: traverse the list again and check that predecessor is still reachable and points to current
- If validation fails, unlock and restart
Trust, but verify
- Traverse the list optimistically without taking any locks
- Lock 2 nodes (predecessor and current)
- Validate: traverse the list again and check that **predecessor is still reachable and points to current**
- If validation fails, unlock and restart

**Positive:**
- Lock contention unlikely

**Negative:**
- Must traverse list twice, method contains acquires a lock
Optimistic lock coupling

- Associate lock with update counter

- Write:
  - Acquire lock (exclude other writes)
  - Increment counter when unlocking
  - Do not acquire locks for nodes that are not modified (traverse like a reader)

- Read:
  - Do not acquire locks, proceed optimistically
  - Detect concurrent modifications through counters (and restart if necessary)
Optimistic lock coupling

- Associate lock with update counter

- Write:
  - Acquire lock (exclude other writes)
  - Increment counter when unlocking
  - Do not acquire locks for nodes that are not modified (traverse like a reader)

- Read:
  - Do not acquire locks, proceed optimistically
  - Detect concurrent modifications through counters (and restart if necessary)

- Positive
  - Easy to implement
  - Scalable

- Negative
  - has restarts
Synchronization in ART tree

- Evaluate the different synchronization approaches (+ lazy (ROWEX), speculative (HTM) and Masstree) on the Adaptive Radix Tree

![Graph showing scalability for lookup, insert, and remove operations.](image)

**Figure 5: Scalability (50M 8 byte integers)**

src: Leis et al. The ART of Practical Synchronization. DaMoN 2016
References

- Various papers cross-referenced in the slides

- Lecture: *Data Processing on Modern Hardware* by Prof. Viktor Leis (Uni Jena, past TUM)
- Lecture: *Data Processing on Modern Hardware* by Prof. Jens Teubner (TU Dortmund, past ETH)
- Lecture: *Supporting Parallelism in OS and Programming Languages* by Dr. Kornilios Kourtis (IBM Research, past ETH)

- Book: *Architecture of a Database System* by Hellerstein, Stonebraker and Hamilton
  - Chapters 2 and 3
- Book: *The Art of Multiprocessor Programming* by Herlihy and Shavit
  - Chapters 7 and 8
- Book: *Is Parallel Programming Hard, And, If So, What Can You Do About It?* by McKenny
- Book: *Computer Architecture: A Quantitative Approach* by Hennessy and Patterson
  - Chapter 5