Morsel-Drive Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age

Presented by Dennis Grishin
What is the problem?

Efficient computation requires distribution of processing between many cores and associated memory.
Why is it important?

Rise of the multi-core CPU architecture

The rise of the multi-core CPU architecture is significant because it addresses the limitations of single-core processors. With the increase in processor cores, we can harness parallel processing capabilities, leading to higher performance and efficiency. This is particularly true in fields such as scientific computing, artificial intelligence, and data processing, where tasks can be divided into smaller, concurrently executable components.

Rise of the NUMA architecture

Non-Uniform Memory Access (NUMA) architecture is important because it allows for efficient memory access in multi-processor systems. In a NUMA system, each processor has its own dedicated memory, and the inter-processor communication is transparent to the application. This is in contrast to Uniform Memory Access (UMA) systems, where all processors share a single memory space.

NUMA architectures are designed to maximize the scalability and performance of large-scale systems. By assigning memory to local processors, NUMA systems can reduce the latency of accessing data, which is crucial for applications that require fast access to data, such as databases and high-performance computing.

There are two main types of NUMA architectures:

1. Distributed Memory Architecture: In this type of architecture, each processor has its own local memory, and the processors are connected over a network. Examples of this type of architecture include clusters of servers, where each server node has its own memory and the processors communicate over the network.

2. Shared Memory Architecture: This type of architecture is prevalent in industry-standard multiprocessor systems, such as Intel microprocessors. Here, all processors access a single, shared memory space. This architecture is advantageous when algorithms can be expressed in terms of shared memory, because it allows for faster access to data compared to the distributed approach.

Both architectures have their own set of benefits and challenges. Understanding the differences between UMA and NUMA is crucial for optimizing the performance of applications on multi-processor systems.
Why is it hard?

How to distribute work evenly between many out-of-order cores?

How to maximize NUMA-local execution?
Why existing solutions do not work?

*Plan-driven parallelism:* query fragmentation at compile time into big fragments and initiation of static number of threads

Insufficient load-balancing due hard-to-predict performance of out-of-order CPUs.
What is the core intuition of the solution?

*Morsel-driven parallelism:* query fragmentation at runtime into small fragments and dynamic scheduling of threads.

Runtime scheduling is elastic and achieves perfect load-balancing.
Solution I – three-way join

\[
\text{select * from R, S, T where R.A = S.A and S.B = T.B}
\]
Table partitioning on the join key -> matching tuples usually on the same socket -> less cross-socket communication for joins

Tagging of hash bucket lists reduces -> list traversal skipped -> number of cache misses reduced to 1
Solution III – probe-phase

Morsel-wise probing

Storage area of blue core
Storage area of green core
Storage area of red core

HT(T)  HT(S)

... (R) v ... (R) v ... (R) v

Figure 4: Morsel-wise processing of the probe phase

The result of the probe pipeline is again stored in NUMA local storage areas in order to preserve NUMA locality for further processing (not present in our sample query plan).

In all, morsel-driven parallelism executes multiple pipelines in parallel, which is similar to typical implementations of the Volcano model. Different from Volcano, however, is the fact that the pipelines are not independent. That is, they share data structures and the operators are aware of parallel execution and must perform synchronization (through efficient lock-free mechanisms – see later). A further difference is that the number of threads executing the plan is fully elastic. That is, the number may differ not only between different pipeline segments, as shown in Figure 2, but also inside the same pipeline segment during query execution – as described in the following.

3. DISPATCHER: SCHEDULING PARALLEL PIPELINE TASKS

The dispatcher is controlling and assigning the compute resources to the parallel pipelines. This is done by assigning tasks to worker threads. We (pre-)create one worker thread for each hardware thread that the machine provides and permanently bind each worker to it. Thus, the level of parallelism of a particular query is not controlled by creating or terminating threads, but rather by assigning them particular tasks of possibly different queries. A task that is assigned to such a worker thread consists of a pipeline job and a particular morsel on which the pipeline has to be executed. Preemption of a task occurs at morsel boundaries – thereby eliminating potentially costly interrupt mechanisms. We experimentally determined that a morsel size of about 100,000 tuples yields good tradeoff between instant elasticity adjustment, load balancing and low maintenance overhead.

There are three main goals for assigning tasks to threads that run on particular cores:

1. Preserving (NUMA-)locality by assigning data morsels to cores on which the morsels are allocated
2. Full elasticity concerning the level of parallelism of a particular query
3. Load balancing requires that all cores participating in a query pipeline finish their work at the same time in order to prevent (fast) cores from waiting for other (slow) cores.

In Figure 5 the architecture of the dispatcher is sketched. It maintains a list of pending pipeline jobs. This list only contains pipeline jobs whose prerequisites have already been processed. E.g., for our running example query the build input pipelines are first inserted into the list of pending jobs. The probe pipeline is only inserted after these two build pipelines have been finished. As described before, each of the active queries is controlled by a QEPobject which is responsible for transferring executable pipelines to the dispatcher. Thus, the dispatcher maintains only lists of pipeline jobs for which all dependent pipelines were already processed. In general, the dispatcher queue will contain pending pipeline jobs of different queries that are executed in parallel to accommodate inter-query parallelism.

3.1 Elasticity

The fully elastic parallelism, which is achieved by dispatching jobs “a morsel at a time”, allows for intelligent scheduling of these inter-query parallel pipeline jobs depending on a quality of service model. It enables to gracefully decrease the degree of parallelism of, say a long-running query at any stage of processing in order to prioritize a possibly more important interactive query. Once the higher prioritized query is finished, the pendulum swings back to the long running query by dispatching all or most cores to tasks of the long running query. In Section 5.4 we demonstrate this dynamic elasticity experimentally. In our current implementation all queries have the same priority, so threads are distributed...
Solution III - dispatcher

- Dispatcher implemented as a lock free data structure and executed by work requesting threads
- Maintains a list of pending pipeline jobs whose prerequisites have already been processed
- Segmentation of queries upon request by processing thread
- NUMA-locality awareness
- "Work stealing" if necessary
Solution IV – morsel size

- Morsels should be large enough to amortize scheduling overhead while providing a good response time.

![Execution time dependency on morsel size](image)
Experiment results I - speedup

Processing of 22 TPC-H queries
Experiment results II - elasticity

**Intra- and inter-query parallelism**

 Threads per query stream

![Graph showing throughput vs. number of query streams]

**Morsel-wise processing**

Worker 0, Worker 1, Worker 2, Worker 3

q13 start, q14 start, q14 finish, time
Does the paper prove its claims?

Yes.
Are there any gaps in the logic?

- Can the compilation at runtime cause a significant overhead?

- What is the overhead caused by the dispatcher in *morsel*-driven parallelism?

- Can all types of queries be easily broken into morsels?
Possible next steps?

• Priority based scheduling.

• Hardware specific optimization.

• Real world testing.