Sewing Together the 3 Layers of Shared-Memory Programming

Lefteris Sidirourgos
Systems Group, Dept. of Computer Science
ETH Zurich, Switzerland
lsidir@inf.ethz.ch

The three layers of shared-memory programming can be identified as *i)* the message passing interface (MPI), *ii)* the fork–join model, and *iii)* the single (or multiple) instruction – multiple data extensions (SIMD/MIMD). There are fundamental differences between these three layers, but each layer has a special place in the design of a data management system. MPI is used to support multiple users/clients and for responding to external events (e.g., I/O). The fork–join extends the control flow of the query execution engine and allows the parallel execution of a query after the data has been horizontal or hash partitioned. The SIMD/MIMD instructions are used internally within each individual operator to accelerate scalar computation to vector or array computation.

But the differences between the three layers of shared-memory programming are not only their places in a database engine, but also on our ability to reason about their correctness. For example, the SIMD/MIMD instructions do not change the flow of the query execution, in fact the flow remains sequential. Thus, it is easy to reason about the correctness of the implementation as the equivalent sequential operator is obvious. Therefore, database designers do not need to introduce locks or other structures to prevent data races or out of order updates when using SIMD extensions.

On the other hand, the fork–join model changes the flow of the query execution plan. It splits the execution into many independent and parallel sequential plans, which are later joined back together. Fork–join is harder to program but it is still easy to reason about, i.e., a depth-first execution of the forked code provides the equivalent sequential plan. Because a fork–join program is deterministic, in many cases we can avoid using locks and thus greatly reduce the query execution overheads associated with them. Avoiding locks for this layer is always an interesting research subject and many solutions do exist, such as latch-free B+trees.

MPI is used to compose together multiple (sequential) executions, but contrary to the previous two layers, MPI is hard to reason about. That is because even though MPI programs can be restricted to be inherently deterministic, in practice this is rare and deadlocks do appear. Here is where concurrency control solutions come into play, such as optimistic concurrency control (occ) or multi-version concurrency control (mvcc) to name a few.

Traditionally in software design, the three layers of shared-memory programming remain independent of each other. For example, MPI design is unaware if a function uses SIMD instructions or not, and fork–join control flow is unaware of the fact that after a join a lock is released or a message is passed. The question that this text is asking is *whether the watertight separation between the three layers is always the best way to go when designing a data management system? Or a seamless fusion of these layers could lead to better query execution planning, increased performance, and stronger reasoning to avoid locks and concurrency control?*

At first glance, the answer seems to be no. There is a good reason why these layers remain independent and oblivious of each other. Watertight separation allows for easy coding and reasoning. But, before dismissing this idea, there are a few design steps we could take in order to shed some more light to the answer.

Index structures and relational operators are designed first as sequential algorithms and only later are extended to support SIMD instructions. But there are cases where the design choices made for scalar CPU execution are not the best for the equivalent SIMD ones. For example, compression is used to increase data transfer rates between memory and CPU cache. Compression introduces `if` control-flow statements and decompression computation, but that is ok since data management is by definition data and not computation intensive, thus we have enough cycles to spare. However, SIMD instructions do not do well with control flow, nor do with memory `gather` instructions (which are needed to load scattered qualifying compressed values). SIMD instructions perform better when a `stream_load` instruction is used. This observation implies that a higher level decision has to be made, i.e., at query planning, whether or not to use compressed data, and this depends on the underlying operator implementation. In other words, the control flow of the fork–join plan execution may depend on the operator implementation (scalar or SIMD) in order to achieve absolute maximum performance. The same observation holds for other database accelerators, such as GPUs or FPGAs, where instead of trying to stick them on the side of existing data management system designs, we should try to seamless integrate them in the design. However, this subject is out of our scope for now.

A much more important and interesting design step is to provide lock-free execution (or zero-cost concurrency control) in the layers of fork–join and MPI. For that we should take advantage of modern programming languages that can provide guarantees for data-race free programs, as early as during compile time. For example, the RUST language of the Mozilla foundation, through a combination of (relaxed) linear typing (where each variable can be used only once), and an ownership/borrow system can detect possible data-races at compile time and prevent the user from doing such mistakes. A combination of such strong data access/typing system and just-in-time query compilation may lead to high performance data management systems that only perform static concurrency control (scc) at query compile time.