Software and the Concurrency Revolution

Presentation for TDT01 2018
Presented by Vegard Seim Karstang

Architecture changes

- Sequential programs ran faster and faster
- Architecture changes to multicore
- Sequential programs may run slower
- Possible solution: concurrency
- Concurrency is hard
- Requires different mindset

Consequences for Software

- Software must be changed
- Semaphores and coroutines are hard
- OOP-like revolution
- Higher level abstractions

Client and server differences

- Concurrency used routinely server-side
- Independent requests
- Client-side differs
- Computation must be divided
- Smaller and more complex pieces

Programming models

- Models differ in two dimensions
- Granularity
- Coupling
- Independent parallelism
- Regular parallelism
- Unstructured parallelism

Shared state and lock problems

- Shared state is a big challenge
- Data races causing non-deterministic execution
- Locks
- Comes with its own set of problems
- Lock-free programming
- Transactional memory

What we need in languages

- Higher levels of abstraction
- Programming model must change
- Explicit approach
- Implicit approach
- Automatic approach
- Functional languages
- Still some mutable state
- Purely functional languages
- Slow
- Abstractions:
- Map and Map-Reduce
- Async calls and futures
- Active objects

What we need in tools

- Tools must find data races, deadlocks etc.
- Conventional tools struggle with these
- Systematic defect detection
- State space too big
- Abstractions that are easier to reason with
- Execution debugging needs more data
- Reverse execution
- Too costly and complex
- Testing must change
- Code coverage doesn’t provide a good picture
- Stress testing
- Gives probabilistic confidence

Conclusion

- The concurrency revolution is more of a software revolution
- Programming multicore processors is hard
- Parallelism through the whole software stack
- Higher level abstractions
- Easily understood and verifiable

Change from uniprocessors to multiprocessors

For many years improvements in fabrication and implementation of processors steadily increased the speed at which already existing sequential programs were executed. This changed when processor architecture changed from unicore processors til multicore processors. For sequential programs this shift means that they may not run faster in the future. They may actually run slower as the individual cores in a multiprocessor may run slower to reduce power consumption. This change is not the only reason to want concurrency in a program. For example, applications must move work off the UI thread to allow work to be done without making the application unresponsive. However concurrency is hard because it forces programmers to think in a way humans usually find difficult.

Consequences for software

This change in hardware means that we have to change the way we write software. Low-level concurrency tools such as semaphores and coroutines are difficult to work with and get right. What is really needed is a kind of revolution to concurrency where higher-level abstractions help programmers with writing concurrent and correct code. This higher level of abstraction is needed because concurrent programming is much harder than sequential programming.

Differences between client and server

Concurrency is a challenge for many client-side applications, but on the server-side concurrency is routinely being used. The reason is that many server-side applications receive requests which can be handled in parallel because they are independent of each other. The situation is quite different on the client-side because client-side applications are usually not as well structured and regular. To parallelize a client-side program one usually has to divide a computation into finer pieces which interact in more complex pattern compared to a server-side application. These finer pieces are often harder to parallelize because they contain heterogenous code; fine-grained, complicated interactions; and pointer-based data structures.

Programming models

Parallelism can be expressed in a number of ways, and they differ significantly in at least two dimensions: the granularity of parallel operations and the degree of coupling between tasks. The granularity can range from single instructions to whole programs that may take a long time to complete. The degree of coupling ranges from no coupling at all to high degree of coupling in irregular programs where the data sharing patterns are complex and difficult to comprehend. The simplest programming model is independent parallelism or so called embarrassingly parallel tasks. In this model one or more operations a independently applied to a data collection, and the operations share no input data or results therefore needing no communication. A second model is regular parallelism where the same operation is applied to a collection of data,

and the computations are mutually dependent. Regular parallelism may require special execution strategies, communication or synchronization between executing operations. The third model is unstructured parallelism where data accesses are not predictable and require explicit synchronization. This is the form of parallelism found in most programs.

The problem of shared state and why locks aren't the answer

The main challenge of unstructured parallelism is shared state. Multiple concurrent accesses to shared data may cause data races that make the program run non-deterministically and may cause the program to return the wrong answer. The simplest answer to this problem is to use locks. However using locks brings with it its own set of problems. Deadlocks and livelocks may cause code to not run to completion, and the programmer needs to be careful when using frameworks that do their own locking to prevent these from happening. Deadlocks and livelocks are hard to debug and techniques to prevent them are difficult to implement. Deadlock prevention through explicitly ordering the way locks should be gained does not compose with other modules or frameworks.

There are some alternatives to locks. One is lock-free programming which relies on knowledge about a processors memory model to create data structures that can be shared without locking. Another is transactional memory which brings in the idea of transactions from databases. This allows the programmer to write a series of atomic blocks, and the concurrently executing operations only see the state before and after these blocks.

What we need in programming languages

Current and new languages need to add higher levels of abstraction to allow existing applications to incrementally become concurrent. The programming model must adapt to make concurrency easy to understand and reason about both during development and maintenance. There are many different ways a programming language can approach concurrency. The explicit way is to provide programmers with abstractions so that they can state exactly where concurrency can occur. This gives the programmer complete control, but it also requires a higher level of programmer proficiency. Another way is the implicit way where the programmer has a sequential view and all the concurrency is baked into libraries or APIs. This approach lets more naive programmers safely use concurrency, but this lack of fine-grained control means some concurrency performance is lost. The third way is to allow the compiler to search for areas in the code that it can automatically parallelize. While this is possible in simple cases, the analysis required for more advanced cases is challenging and the code may not be written with automatic parallelization in mind.

A change from imperative languages to functional languages may be beneficial because they are more naturally suited to concurrency. However functional languages typically exposes function level parallelism which is may be too fine grained. The main reason functional languages are said to be easier to parallelize is that they don't have mutable state, but many

languages still allow mutable state because some problems may be much easier to solve with mutable state. A large drawback of purely functional languages is that they need to copy the whole data structure before changing it which may cause significant performance degradation. The main contributions of functional languages are high level abstractions such as map and map-reduce which are rich sources of concurrency.

Other abstractions that may help are asynchronous calls and futures. An asynchronous call is a non blocking call that is processed in the background, and the return value of this function can be accessed with a future. Another abstraction that may help is an active objects. Active objects run their own thread an acts as a monitor that received asynchronous messages, queues them and executes them on the object.

What we need in tools

Tools must also evolve to help developers adapt to the more challenging task of writing parallel programs. Tools must be able to help a developer find defects such as data races, deadlocks and livelocks. Conventional tools struggle with these errors because they often cause programs to behave in a nondeterministic fashion. Systematic defect detection tools are invaluable when working with concurrency. They use static program analysis to explore all possible executions of a program. The problem with this is that the state space of a concurrent program is too large to analyze in a reasonable amount of time. One way to help with this is to introduce abstractions that make it easier to reason about the concurrent execution of a program. However even with this programmers will need debuggers that can help them diagnose problems with their concurrent code. To facilitate this more data about the execution of the program must be collected. Another way to improve debugging is to allow for reverse execution. While this is a nice idea it struggles with both cost and complexity issues. Testing also needs to change because simple code coverage metrics no longer provides a good picture of how completely a program has been tested. Stress testing may be an alternative, but it only provides probabilistic confidence.

Conclusion

The concurrency revolution is more of a software revolution than a hardware revolution. Changing to multicore processors in hardware is not difficult, it is programming them to keep the exponential growth in performance that is difficult. Doing so requires everyone at every level of the software stack to start thinking about parallelism and the higher level constructs needed to program a parallel program that is easily understood and verifiable by tools.