Technical perspective: Unexpected connections

Marc Shapiro

Sorbonne-Universités-UPMC-LIP6 & Inria Paris
http://lip6.fr/Marc.Shapiro/

Revision -2 (0000-00-00)

Scalability is the capability of a parallel program to speed up its execution as we provide it with more CPUs. Back in 1967, Gene Amdahl noticed that the sequential part of a parallel program has a disproportionate influence on scalability [**?**]. Suppose that some program takes 100 s to run on a sequential processor. Now let's run it on a parallel computer. If we are able to parallelise, say, 80% of the code, then with enough CPUs those 80% would take essentially zero time. However, the remaining sequential portion will not run any faster; this means that the parallel program will always take at least 20 s to run, a maximum speed-up of only 5×. If we are able to parallelise 95% of the code, speed-up is still limited to 20×, even with an infinite number of CPUs! This back-of-the-envelope calculation, known as Amdahl's Law, does not take into account other factors, such as increased memory size, but remains an important guideline.

In 1967, parallelism was a niche topic, but not any more. To improve program performance on today's clusters, clouds and multicore computers requires the developer to pay serious attention to scalability. The inherent scalability of an interface is the focus of our Research Highlight, "The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors."

When a thread updates some shared datum, and another thread wants to read or write the most recent version of that datum (they *conflict*), they must

synchronise, which constitutes a sequential bottleneck. This is a general result that does not depend on any particular implementation, even with efficient hardware support for cache coherence, as explained in the paper.

Here comes the paper's main insight. If two concurrent procedure calls *commute* with each other (i.e., executing them in either order is equivalent), this means that neither one depends on the result of the other. Therefore, *there is no inherent reason why these calls should conflict*; and, hence, *it is possible to implement them in a way that scales well.*

The advantages of commutativity in software have been known for a long time; see the paper for relevant references. It is only recently however that focus has shifted, from simply leveraging existing commutativity, towards designing software to achieve commutativity [**?** **?**]. The current paper goes well beyond previous work. First, instead of simple abstract data types, it considers the more complex case of software with an intricate interface and massive amount of shared state — a whole operating system (OS). Second, instead of just a black-and-white characterisation "commute/don't-commute," it considers calls that may commute in some states and not in others. This is especially important when commuting is the common case, as in many OS calls. Finally, it leverages static program verification techniques, providing a tool that will prove if and when a given interface is commutative, and will generate test cases exercising the scalability of its implementation.

The authors designed a whole OS based on these ideas. It's similar to Linux, but its APIs are designed for commutativity. The implementation mostly scalable, but not always: even when a scalable implementation of an API exists in theory, it's not necessarily be the most obvious or even the most efficient; sometimes, it's simply not worthwhile. They also learned that many advanced data structures do not scale well; for instance, rebalancing a tree may modify a portion of the tree that is semantically unrelated to the update that triggered the rebalancing.

This paper presents a simple and powerful idea. It is not just about OSes, but applies to any piece of parallel software, whether running on a multicore computer or in the cloud. Commutativity enables us to reason about scalability in a principled way, independently of a particular implementation, benchmark or workload. We can now *design* our APIs to be scalable, by ensuring that calls commute in the common case, and we can use verification tools to automate and exercise this reasoning. An

unexpected connection between high-school math theory and hardcore computer science.