Ask For Forgiveness Programming - Or How We'll Program 1000 Cores

Tuesday, March 6, 2012 at 9:15AM

Todd Hoff in Paper, Strategy

The argument for a massively multicore future is now familiar: while clock speeds have leveled off, device density is increasing, so the future is cheap chips with hundreds and thousands of cores. That's the inexorable logic behind our multicore future.

The unsolved question that lurks deep in the dark part of a programmer's mind is: how on earth are we to program these things? For problems that aren't embarrassingly parallel, we really have no idea. IBM Research's David Ungar has an idea. And it's radical in the extreme...

Grace Hopper once advised "It's easier to ask for forgiveness than it is to get permission." I wonder if she had any idea that her strategy for dealing with human bureaucracy would the same strategy David Ungar thinks will help us tame the technological bureaucracy of 1000+ core systems?

You may recognize David as the co-creator of the Self programming language, inspiration for the HotSpot technology in the JVM and the prototype model used by Javascript. He’s also the innovator behind using cartoon animation techniques to build user interfaces. Now he’s applying that same creative zeal to solving the multicore problem.

During a talk on his research, Everything You Know (about Parallel Programming) Is Wrong! A Wild Screed about the Future, he called his approach “anti-lock or “race and repair” because the core idea is that the only way we’re going to be able to program the new multicore chips of the future is to sidestep Amdhal’s Law and program without serialization, without locks, embracing non-determinism. Without locks calculations will obviously be wrong, but correct answers can be approached over time using techniques like fresheners:

> *A thread that, instead of responding to user requests, repeatedly selects a cached value according to some strategy, and recomputes that value from its inputs, in case the value had been inconsistent. Experimentation with a prototype showed that on a 16-core system with a 50/50 split between workers and fresheners, fewer than 2% of the queries would return an answer that had been stale for at least eight mean query times. These results suggest that tolerance of inconsistency can be an effective strategy in circumventing Amdahl's law.*

During his talk David mentioned that he’s trying to find a better name than “anti-lock or “race and repair” for this line of thinking. Throwing my hat into the name game, I want to call it *Ask For Forgiveness Programming* (AFFP), based on the idea that using locks is “asking for permission” programming, so not using locks along with fresheners is really "asking for forgiveness." I think it works, but it's just a thought.

No Shared Lock Goes Unpunished

Amdahl's Law is used to understand why simply having more cores won't save us for a large class of problems. The idea is that any program is made up of a serial fraction and a parallel fraction. More cores only helps you with the parallel portion. If an operation takes 10 seconds, for example, and one second of it is serial, then having infinitely many cores will only help you make the parallelizable part faster, the serial code will always take one second. Amdahl says you can never go faster than that 10%. As long as your code has a serial portion it's impossible to go faster.

Jakob Engblom recounts a similar line of thought in his blog:

> *They also had the opportunity to test their solution [for parallel Erlang] on a Tilera 64-core machines. This mercilessly exposed any scalability limitations in their system, and proved the conventional wisdom that going beyond 10+ cores is quite different from scaling from 1 to 8… The two key lessons they learned was that no shared lock goes unpunished, and data has to be distributed as well as code.*
>
> *It seems that for all big system parallelization efforts turn into a hunt for locks and the splitting up of code and data into units that can run in parallel without having to synchronize. The "upper limit" of this process is clearly a system with no synchronization points at all.*

Sherlock Holmes says that when you have eliminated the impossible, whatever remains, however improbable, must be the truth, so the truth is: removing serialization is the only way to use all these cores. Since synchronization is the core serial component to applications, we must get rid of synchronization.

A Transaction View Isn't as Strong as it Once Was

Getting rid of locks/synchronization may not be the radical notion it once was. Developments over the last few years have conditioned us to deal with more ambiguity, at least for distributed systems.

Through many discussions about CAP, we've come to accept a non-transactional view of databases as a way to preserve availability in the face of partitions. Even if a read doesn't return the last write, we know it will eventually and that some merge logic will make them consistent once again.

The idea of compensating transactions as a way around the performance problems due to distributed coordination has also become more familiar.

Another related idea comes from the realm of optimistic concurrency control that lets multiple transactions proceed in parallel without locking and then checks at commit time if a conflict requires a rollback. The application then gets to try again.

And for some time now memcache has supported a compare-and-set operation that allows multiple clients to avoid writing over each other by comparing time stamps.

As relaxed as these methods are, they all still require that old enemy: synchronization.

Your Answers are Already Wrong

The core difficulty with abandoning synchronization is coming to terms with the notion that results may not be correct at all times. It's a certainty we love certainty, so even considering we could get wrong answers for any window of time is heresy.

David says we need to change our way of thinking. The idea is to get results that are "good enough, soon enough." Get wrong answers quickly, but that are still right enough to be useful. Perfect is the enemy of the good and perfect answers simply take too long at scale.

David emphasises repeatedly that there's a fundamental trade-off between correctness and performance. Without locks operations happen out of order, which gives the wrong answer. Race conditions happen. To get correct answers we effectively add in delays, making one thing wait for another, which kills performance, and we don't want to kill performance, we want to use all those cores.

But consider an aspect of working on distributed systems that people don't like to think about: your answers are already always wrong.

Unless you are querying a read-only corpus or use a global lock, in distributed systems any answer to any query is potentially wrong, always. This is the hardest idea to get into your brain when working in distributed systems with many cores that experience simultaneous updates. The world never stops. It's always in flux. You can never assume for a single moment there's a stable frame of reference. The answer from a query you just made could be wrong in the next instant. A query to see if a host is up, for example, can't ever be assumed right. That host maybe up or down in the next instant and your code won't know about it until it finds a conflict.

So are we really that far away from accepting that all answers to queries could be wrong?

Lessons from Nature

One strategy for dealing with many cores is to move towards biological models instead of mathematical models, where complicated behaviours emerge without global determinism. Bird flocks, for example, emerge from three simple rules: avoid crowding, steer towards average heading of neighbors, steer towards average position of neighbors. No pi-calculus required, it works without synchronization or coordination. Each bird is essentially its own thread, it simply looks around and makes local decisions. This is a more cellular automaton view of the world.

Race and Repair - Mitigating Wrong Results

The idea is that errors created by data races won’t be prevented, they will be repaired. A calculation made without locks will be wrong under concurrent updates. So why not use some of our many cores to calculate the right answers in the background, in parallel, and update the values? This approach:

- Uses no synchronization.
- Tolerates some wrong answers.
- Probabilistically fixes the answers over time.

Some obvious questions: how many background threads do you need? What order should values be recalculated? And how wrong will your answers be?

To figure this out an experiment was run and described in Inconsistency Robustness for Scalability in Interactive Concurrent-Update In-

Memory MOLAP Cubes, which test updates on a complicated spreadsheet.

With locking the results were correct, but scalability was limited. Without locking, results were usually wrong. Both results are as might be expected. And when they added freshener threads they found:

> *Combining the frequency with the duration data suggests that in a 16-core system with a 50/50 split between workers and fresheners, fewer than 2% of the queries would return an answer that had been stale for at least eight mean query times.*

I found this result quite surprising. I would have expected the answers to be wrong more of the time. Could this actually work?

Breadcrumbs

The simple idea of Race and Repair is open to a lot of clever innovations. Breadcrumbs are one such innovation that attempts to be smarter about which values need recalculating. Meta-data is attached on a value indicating that a entity is recalculating the value or has changed a dependent value such that this value is now out of date. Any entity that might want to use this data can wait until a “valid” value is calculated and/or not to insert a calculated value if it is out of data. This narrows the window of time in which errors are introduced. It’s a mitigation.

There are endless variations of this. I can imagine remembering calculations that used a value and then publishing updated values so those calculations can be rerun. The result is a roiling event driven sea that is constantly bubbling with updates that are trying to bring values towards correctness, but probably never quite getting there.

Probabilistic Data Structures

Another area David has researched are hashtables that can be inserted into without synchronization. This would allow entries to be added to a hashtable from any number of cores without slowing down the entire system with a lock. Naive insertion into a hashtable in parallel will mess up pointers, which can either result in the loss of values or the insertion of values, but it’s possible to work around these issues and create a lock free hashtable.

His presentation goes into a lot of detail on how this might work. He rejects light-weight locking schmes like CAS because these are still a choke point and there’s a penalty for atomic instructions under contention. They won’t scale.

He thinks there’s a big research opportunity in probabilistic data structures that work without synchronization and that work with mitigation.

Is this the Future?

This is just a light weight introduction. For more details please read all the papers and watch all the videos. But I think it’s important to talk about and think about how we might make use of all these cores for traditional programming tasks, though the result may be anything but traditional.

A bad experience on one project makes me somewhat skeptical that human nature will ever be comfortable in accepting wrong answers as the norm. My experience report was on an event driven system with a large number of nodes that could generate events so fast that events had to be dropped. Imagine a city undergoing an earthquake and a huge sensor net spewing out change events to tell the world what’s going on in real-time.

The original requirement was events could never be dropped, ever, which made a certain amount of sense when the number of sensors was small. As the number of sensors expands it's simply not possible. My proposal was to drop events and have background processes query the actual sensors in the background so that the database would be synced over time. Very much like the proposals here.

It was a no go. A vehement no go. Titanic arguments ensued. Management and everyone on down simply could not accept the idea that their models would be out of sync with the sensors (which of course they were anyway). My eventual solution took a year to implement and radically changed everything, but that was simpler than trying to convince people to deal with uncertainty.

So there might be some convincing to do.

Related Articles

- Everything You Know (About Parallel Programming) Is Wrong!: A Wild Screed About the Future (slides, video)
- Renaissance Project
- On Hacker News
- On Reddit
- David Ungar: It is Good to be Wrong
- We Really Don't Know How To Compute!
- Harnessing emergence for manycore programming: early experience integrating ensembles, adverbs, and object-based inheritance
- Inconsistency Robustness for Scalability in Interactive Concurrent-Update In-Memory MOLAP Cubes

Article originally appeared on High Scalability (http://highscalability.com/).

See website for complete article licensing information.