Paper review: Amdahl's Law in the Multicore Era

September 11, 2011

This is a paper review of “Amdahl’s Law in the Multicore Era”, by Hill and Marty.

Main ideas

There are two important equations in this paper that lay the foundation for the rest of the paper: Amdahl’s law, and the processor performance law.

```
speedup = 1 / ((1-f + f/s)
perf(r) = sqrt(r)
```

These two equations have some deep implications. Starting with the latter, there are diminishing returns from investing more chip resources to single core performance. It’s the reason why per-core performance has basically peaked in recent years, and everything is moving toward multicore. Adding more cache and beefing up adders can only do so much; since clock rates have peaked because of power limitations, we’re basically stuck with the per-core performance we’ve got.

Moving on to Amdahl’s law, `f` is the fraction of a program that is parallelizable, and `s` is the speedup of the parallelizable part. Interpreting this, we see there are essentially three ways of making your program run faster:

1. Run the serial part faster (`1-f` component)
2. Make `s` bigger (increase the granularity of parallelism)
3. Make `f` bigger (parallelize serial parts of the code)

Combining both equations, what are the limitations we see?

- Running the serial part faster gets expensive quickly in terms of chip resources.
- Increasing the granularity of parallelism is a great approach, but only works really well for data-parallel tasks like serving webpages. Going beyond this natural parallelism into true fine-grained parallelism is expensive and limited.
- Parallelizing serial parts of the code is also a great approach, but similarly limited. Some serial code simply can’t be parallelized, and there’s the same diminishing returns effect in terms of programmer time.

The article goes on to talk about different ways of allocating chip resources in light of these two laws, covering the symmetric, asymmetric, and dynamic multicore chips. Basically, favoring bigger cores makes the serial part run faster, but reduces the number of ways you can split the parallel part, with smaller cores having the opposite effect. The net result of the paper is that it all depends on your workload. Asymmetric and dynamic multicore offer better performance characteristics than symmetric since they can better handle both the serial and parallel parts of a program well.

Applying this to cloud computing, we see that there are a lot of parallels between

multicore and cloud computing. Instead of getting speedup by parallelizing at the level of instructions in a program, web services are typically scaled through request-level parallelism: distributing requests among a cluster of machines. In this case, the `1-f` factor in Amdahl's law can be effectively zero, since requests can be handled independently by each machine.

The ideas of asymmetric cores is also already seen in request-level parallelism since bigger tasks can be allocated to bigger machines.

Future trends

Servers processors are going to be manycore in the future, whether we want it or not. Peak per-core performance is probably not going to increase, and I don't hold out much hope for the effectiveness of dynamic multicore, but I think there's a strong argument for asymmetric multicore since there are demonstrated benefits over symmetric. Big cores will be used for serial computation, with data-parallel parts off-loaded to small cores as possible. This is also better for the coming scarcity of memory bandwidth; having big cores that can use it more effectively makes sense.

This also lends flexibility into request handling, since high-priority tasks can get the big core while latency-tolerant operations can run on a small core. The big problem here is going to be performance isolation for shared resources like mem, disk, and network; with more threads, there's more contention, and potentially more variability in performance (which is the killer). Scheduling also becomes more difficult, since "slots" on the same machine are now unequal in size.

For batch processing (like Hadoop), throughput matters more than latency. Here, SMP is the name of the game: we want to optimize `perf(r)` and have a bunch of wimpy cores. I still worry about I/O demands on durable storage, since HDD throughput doesn't seem to be increasing at the same rate that cores are being added.

Software stack predictions:

- We're going to see a focus on better asymmetric-aware, load-based cluster scheduling algorithms. This needs to balance out I/O load too.
- We're going to see moderate fine-grained parallelism added to existing request-level parallelism, since it's the real only way of reducing latency.
- As a corollary, I think there's going to be a big focus on performance isolation even at the cost of throughput, since variation in latency is the real killer.
- MapReduce as a programming model and framework probably will not go away, but we're going to see mappers making better use of multi-core. Whether this is going to be MapReduce-in-MapReduce or OpenMP-in-MapReduce or Pthreads-in-MapReduce is unclear, but it makes sense to further break down an already data-parallel task into core-sized chunks.

umbrant

My name is Andrew Wang. I'm a CS PhD student at UC Berkeley working in distributed systems.

Subscribe to my feed