Lecture 36—Massive Scalability; Course Summary

ECE 459: Programming for Performance

April 6, 2015

Part I

Clusters vs Laptops

References

Frank McSherry, Michael Isard, Derek G. Murray.
"Scalability! But at what COST?"
To appear in Proceedings of HotOS XV.

The blog post is more digestible:

`http://www.frankmcsherry.org/graph/scalability/cost/2015/01/15/COST.html`

Problem: Overhead

Scaling to “big data” systems
introduces substantial overhead.

How much?
Let’s do a head-to-head comparison!

The Experiment

citation: "Macbook Pro Retina 13 2013" by TechGizmo - Own work. Licensed under CC BY-SA 3.0 via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:Macbook_Pro_Retina_13_2013.jpg

vs

(GraphX)

Conclusion

Big data systems haven't yet been shown to be obviously good;
current evaluation is lacking.

Takeaway: The important metric is not just scalability;
absolute performance matters a lot too.

Methodology

Laptop vs. Big Data systems (GraphX).
Domain: Graph Processing algorithms.

- PageRank (sparse matrix/vector multiplication)
- graph connectivity (label propagation)

Subject: graphs with billions of edges (few GBs of data).

Results

Twenty pagerank iterations:

System	cores	twitter_rv	uk_2007_05
Spark	128	857s	1759s
Giraph	128	596s	1235s
GraphLab	128	249s	833s
GraphX	128	419s	462s
Single thread	1	300s	651s

Label propagation to fixed-point (graph connectivity)

System	cores	twitter_rv	uk_2007_05
Spark	128	1784s	8000s+
Giraph	128	200s	8000s+
GraphLab	128	242s	714s
GraphX	128	251s	800s
Single thread	1	153s	417s

Algorithmic Improvements

Algorithmic improvements can win big.
(Hard to generalize, though.)

Examples:

- Hilbert curves for data layout improve memory locality (helps a lot for PageRank); and
- using a union-find algorithm (also parallelizable). "10× faster, 100× less embarrassing".

Results: overall, 2× speedup for PageRank and
10× speedup for label propagation.

Takeaways

- “If you are going to use a big data system for yourself, see if it is faster than your laptop.”
- “If you are going to build a big data system for others, see that it is faster than my laptop.”

Part II

Massive Scalability

People worth following

- Fran Allen (IBM)
- Jeff Dean (Google)
- Keith Packard (Intel)
- Herb Sutter (Microsoft)

Some Thoughts from Jeff Dean

URL: `research.google.com/pubs/jeff.html`

Selections from:
"Software Engineering Advice for Building
Large-Scale Distributed Systems".

On scaling:

- design for $\sim$ 10x, rewrite before 100x

Key concept for scaling:

- sharding, also known as partitioning.

Why Distribute?

Let's say that we want a copy of the Web.

20+ billion web pages × 20KB = 400+ TB.
$\sim$ 3 months to read the web.
$\sim$ 1000 HDs (in 2010) to store the web.

And that's without even processing the data!

Magic solution: distribute the problem!

1000 machines ⇒ < 3 hours.

No Free Lunch.
Problems: need to deal with . . .

- communication & coordination;
- recovering from machine failure;
- status reporting;
- debugging;
- optimization;
- locality

. . . and that, from scratch, for each problem!

Designing Systems as Services/Platforms

Steve Yegge on Google and Platforms:

`https://plus.google.com/112678702228711889851/posts/eVeouesvaVX`

[Bezos's] Big Mandate went something along these lines:
1) All teams will henceforth expose their data and functionality through service interfaces.
2) Teams must communicate with each other through these interfaces.
3) There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
4) It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols – doesn't matter. Bezos doesn't care.
5) All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
6) Anyone who doesn't do this will be fired.
7) Thank you; have a nice day! [j/k]

Why Services?

Decouple different parts of a system.

“Fewer dependencies, clearly specified”.

“Easy to test new versions.”

“Small teams can work independently.”

How to Design Stuff

Talk to people!

Write down a [some] rough sketch[es],
chat at a whiteboard.

Design good interfaces. (This is hard.)

Using Back-of-the-Envelope Calculations I

"How long to generate image results page (30 thumbnails)?"

1. read serially, thumbnail 256K images on-the-fly:
   30 seeks $\times$ 10 ms/seek $+$ 30 $\times$ 256K / 30MB/s $=$ 560ms
2. issue reads in parallel:
   10 ms/seek $+$ 256KB read / 30 MB/s $=$ 18ms
   (plus variance: 30-60ms)

Using Back-of-the-Envelope Calculations II

“How long to quicksort 1GB of 2-byte numbers?”

Comparisons: lots of branch mispredicts.
$\log(2^{28})$ passes over 2^{28} numbers $= \sim 2^{33}$ compares
$\sim 50\%$ mispredicts $= 2^{32}$ mispredicts $\times$ 5 ns/mispredict $=$ 21s.

Memory bandwidth: mostly sequential streaming.
2^{30} bytes $\times$ 28 passes $=$ 28GB;
memory bandwidth $\sim 4GB/s$, so $\sim$ 7 seconds.

Total: about 30 seconds to sort 1GB on 1 CPU.

Also, write microbenchmarks. Understand your building blocks.

Numbers Everyone Should Know

`http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html`

Part III

Course Summary

Key Concepts I: goals

- Bandwidth versus Latency.
- Concurrency versus Parallelism.
- More bandwidth through parallelism.
- Amdahl's Law and Gustafson's Law.

Key Concepts II: leveraging parallelism

- Features of modern hardware.
- Parallelism implementations: pthreads.
  - definition of a thread;
  - spawning threads.
- Problems with parallelism: race conditions.
  - manual solutions: mutexes, spinlocks, RW locks, semaphores, barriers.
  - lock granuarlity.
- Parallelization patterns; also SIMD.

Key Concepts III: inherently-sequential problems

- Barriers to parallelization: dependencies.
  - loop-carried, memory-carried;
  - RAW/WAR/WAW/RAR.
- Breaking dependencies with speculation.

Key Concepts IV: higher-level parallelization

- Automatic parallelization; when does it work?
- Language/library support through OpenMP.

Key Concepts V: hardware considerations

- Unwelcome surprises: memory models & reordering.
  - fences and barriers; atomic instructions.
  - cache coherency implementations.

Key Concepts VI: help from the compiler

- Three-address code.
- Compiler constructs: volatile, restrict.
- Inlining and other static optimizations.
- Profile-guided optimizations.

Key Concepts VII: profiling

- Profiling tools and techniques.
- Call graphs, performance counters from profilers.
- When your profiler lies to you!
- Query-based DTrace approach.

Key Concepts VIII: assorted topics

- Reduced-resource computing.
- Software transactions.
- DevOps for programming for performance.

Key Concepts IX: beyond single-core CPU programming

- Languages for high-performance computing.
- GPU Programming (e.g. with OpenCL).
- Clusters: MapReduce, MPI.
- Clouds.
- Big Data.

Final Words

Good luck on the final!