ECE459: Programming for Performance Winter 2024

Lecture 30 — Clusters & Cloud Computing

Jeff Zarnett & Patrick Lam *2024-09-17*

Clusters and Cloud Computing

Almost everything we've seen so far has improved performance on a single computer. Sometimes, you need more performance than you can get on a single computer, or you have reached a point where it's a lot better value for your money to buy two cheap computers than one very expensive computer. For the most part, if you could split your problem into multiple threads or multiple processes, you can do the same with multiple computers. We'll survey techniques for programming for performance using multiple computers; although there's overlap with distributed systems, we're looking more at calculations here rather than coordination mechanisms.

Message Passing. Rust encourages message-passing, but a lot of your previous experience when working with C may have centred around shared memory systems and you may have done that in Rust too (`Mutex`, anyone?). In some circumstances that we've seen, you don't have a choice! The model of GPU programming meant we had to explicitly manage copying of data. When we have multiple computers, shared memory is not an option, so we have to use message passing. Fortunately, we know how to do that. The only thing to note now is that communication over the network is much more expensive than communicating within the same system, so we will need to think carefully about how much to communicate.

There was a time when we would talk about MPI, the *Message Passing Interface*, a de facto standard for programming message-passing multi- computer systems. This is, unfortunately, no longer the way. MPI sounds good, but in practice people tend to use other things. Here's a detailed piece about the relevance of MPI as of 10 years ago: [Dur15], if you are curious.

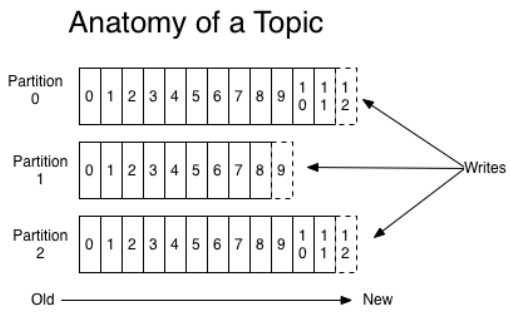
REST We've already seen asynchronous I/O using HTTP (curl) which we could use to interact with a REST API as one mechanism for multi-computer communication. You may have also learned about sockets and know how to use those, which would underlie a lot of the mechanisms we're discussing. The socket approach is too low-level for what we want to discuss, while the REST API approach is at a reasonable level of abstraction.

REST APIs are often completely synchronous, but don't have to be: you can set up callbacks to be notified when a computation is done, or the caller can check back later to see if the request is finished. But these aren't truly asynchronous, because the remote machine has to be available at the time of each call. Perhaps we'd like to decouple services even more than that and use something else...

Kafka. Let's talk about Apache Kafka, a self-described "distributed streaming platform" that's got significant adoption in industry as a mechanism of communication between different services running over a network. We're using [Kur20] as a guide and it provides a breakdown on how Kafka works.

Communication is based around the idea of producers writing a record (some data element, like an invoice) into a topic (categorizing messages) and consumers taking the item from the topic and doing something useful with it. A message remains available for a fixed period of time and can be replayed if needed. I think at this point you have enough familiarity with the concept of the producer-consumer problem and channels/topics/subscriptions that we don't need to spend a lot of time on it.

Kafka's basic strategy is to write things into an immutable log. The log is split into different partitions; you choose how many when creating the topic, where more partitions equals higher parallelism. The producer writes something and it goes into one of the partitions. Consumers read from each one of the partitions and writes down its progress ("commits its offset") to keep track of how much of the topic it has consumed. See this image from `kafka.apache.org`:



The nice part about such an architecture is that we can provision the parallelism that we want, and the logic for the broker (the system between the producer and the consumer, that is, Kafka) is simple. Also, consumers can take items and deal with them at their own speed and there's no need for consumers to coordinate; they manage their own offsets. Messages are removed from the topic based on their expiry, so it's not important for consumers to get them out of the queue as quickly as possible.

You can see a visualization of Kafka at https://softwaremill.com/kafka-visualisation/.

Quick aside on that: under normal circumstances, in something like a standard queue, there's a little bit of pressure to get items out of the queue quickly. If the queue is growing, it might mean the queue is full and new items can't be produced or are thrown away... or perhaps you get charged money for the storage space the queue is using.

You might think that it's a solution to take the item out of the queue in one transaction and then process it later. That's okay only if you've successfully saved it to a database or other persistent storage. Otherwise, you could take the item out and a crash or called shutdown means that the item doesn't actually get processed. Oops!

Alternatives. There's also some popular AWS (Amazon Web Services)-based solutions for this kind of multiple computer communication: SNS (Simple Notification Service) and SQS (Simple Queueing Service). They are, broadly speaking, just other ways to decouple the communication of your programs.

SNS is good for sending lots of messages to multiple receivers, maybe push notifications or making sure all systems update their records or that you send pager alerts to people about things that have gone wrong. SNS messages are not persistent, so if you miss it, you miss it.

SQS is more for batches of work where it's not particularly time-sensitive and the item will be consumed by a worker. SQS data is deleted after being taken out of the queue and may or may not have ordering guarantees. SQS does have limits on how long a message can be stored, though.

Cloud Computing

We'll start with a little bit of history. In the old days, if you wanted a cluster, you had to find a bunch of money to buy and maintain a pile of expensive machines. Not anymore. Cloud computing is perhaps way overhyped, but we can talk about one particular aspect of it, as exemplified by Amazon's Elastic Compute Cloud (EC2).

Consider the following evolution:

- Once upon a time, if you wanted a dedicated server on the Internet, you had to get a physical machine hosted, usually in a rack somewhere. Or you could live with inferior shared hosting.
- Virtualization meant that you could instead pay for part of a machine on that rack, e.g. as provided by `slicehost.com`. This is a win because you're usually not maxing out a computer, and you'd be perfectly happy to share it with others, as long as there are good security guarantees. All of the users can get root access to their virtual part computer. (Unfortunately, Spectre and Meltdown allow you to see parts of the machine that are not yours.)

- Clouds enable you to add more machines on-demand. Instead of having just one virtual server, you can spin up dozens (or thousands) of server images when you need more compute capacity. These servers typically share persistent storage, also in the cloud.

In cloud computing, you pay according to the number of machines, or instances, that you've started up. Providers offer different instance sizes, where the sizes vary according to the number of cores, local storage, and memory. Some instances even have GPUs, but it seemed uneconomic to use this for Assignment 3. Instead we have the `ecetesla` machines.

Launching Instances. When you need more compute power, you launch an instance. The input is a virtual machine image. You use a command-line or web-based tool to launch the instance. After you've launched the instance, it gets an IP address and is network-accessible. You have full root access to that instance.

Amazon provides public images which run a variety of operating systems, including different Linux distributions, Windows Server, and OpenSolaris. You can build an image which contains the software you want, including Hadoop and OpenMPI.

Terminating Instances. A key part of cloud computing is that, once you no longer need an instance, you can just shut it down and stop paying for it. All of the data on that instance goes away.

Continuous Deployment. The combination of launching and terminating instances can mean that updates don't require downtime: you start up the new instances and then shut down the old ones so there's no gap where the service is down.

Sometimes this causes slight headaches, like if you make a database change that's incompatible: the old nodes will have a problem, so you need to be a little careful with this.

Storing Data. You probably want to keep some persistent results from your instances. Basically, you can either mount a storage device, also on the cloud (e.g. Amazon Elastic Block Storage); or, you can connect to a database on a persistent server (e.g. Amazon SimpleDB or Relational Database Service); or, you can store files on the Web (e.g. Amazon S3).

Clusters versus Laptops

There is a paper about this: Frank McSherry, Michael Isard, Derek G. Murray. "Scalability! But at what COST?" HotOS XV. This part of the lecture is based on the companion blog post [McS15].

The key idea: scaling to big data systems introduces substantial overhead. Let's just see how, say, a laptop compares, in absolute times, to 128-core big data systems.

Summary. Big data systems haven't yet been shown to be obviously good; current evaluation is lacking. The important metric is not just scalability; absolute performance matters a lot too. We don't want a situation where we are just scaling up to n systems to deal with the complexity of scaling up to n systems. Or, as Oscar Wilde put it: "The bureaucracy is expanding to meet the needs of the expanding bureaucracy."

Methodology. We'll compare a competent single-threaded implementation to top big data systems, as described in an OSDI 2014 (top OS conference) paper on GraphX [GXD+14]. The domain: graph processing algorithms, namely PageRank and graph connectivity (for which the bottleneck is label propagation). The subjects: graphs with billions of edges, amounting to a few GB of data.

Results. 128 cores don't consistently beat a laptop at PageRank: e.g. 249–857s on the twitter_rv dataset for the big data system vs 300s for the laptop, and they are 2× slower for label propagation, at 251–1784s for the big data system vs 153s on twitter_rv. From the blogpost:

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

Wait, there's more. I keep on saying that we can improve algorithms for additional performance boosts too. But that doesn't generalize, so it's hard to teach. In this case, two improvements are: using Hilbert curves for data layout, improving memory locality, which helps a lot for PageRank; and using a union-find algorithm (which is also parallelizable). "10× faster, 100× less embarrassing". We observe an overall $2\times$ speedup for PageRank and $10\times$ speedup for label propagation.

Takeaways. Some thoughts to keep in mind, from the authors:

- "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."

Movie Hour

Let's take a humorous look at cloud computing: James Mickens' session from Monitorama PDX 2014.

`https://vimeo.com/95066828`

References

[Dur15] Jonathan Dursi. HPC is dying, and MPI is killing it, 2015. Online; accessed 6-January-2016. URL: `http://www.dursi.ca/post/hpc-is-dying-and-mpi-is-killing-it/`.

[GXD+14] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework, 2014. 11th USENIX Symposium on Operating Systems Design and Implementation. URL: `https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-gonzalez.pdf`.

[Kur20] Merrin Kurian. Why should anyone use Apache Kafka?, 2020. Online; accessed 2020-11-07. URL: `https://medium.com/swlh/why-should-anyone-use-apache-kafka-f2b632d0963c`.

[McS15] Frank McSherry. Scalability! but at what COST?, 2015. Online; accessed 11-January-2016. URL: `http://www.frankmcsherry.org/graph/scalability/cost/2015/01/15/COST.html`.