ECE459: Programming for Performance Winter 2021

Lecture 29 — Clusters & Cloud Computing

Patrick Lam *2020-10-15*

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. If you're lucky, then the problem can be divided among 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.

Message Passing

For the majority of this course, we've talked about shared-memory systems. Last week's discussion of GPU programming moved away from that a bit: we had to explicitly manage copying of data. Message-passing is yet another paradigm. In this paradigm, often we run the same code on a number of nodes. These nodes may potentially run on different computers (a cluster), which communicate over a network.

MPI, the *Message Passing Interface*, is a de facto standard for programming message-passing systems. Communication is explicit in MPI: processes pass data to each other using `MPI_Send` and `MPI_Recv` calls.

Relevant piece about the relevance of MPI today: [Dur15]

Hello, World in MPI. As with OpenCL kernels, the first thing to do when writing an MPI program is to figure out what the current process is supposed to compute. Here's fairly standard skeleton code for that, from `http://www.dartmouth.edu/~rc/classes/intro_mpi/`:

```
#include <stdio.h>
#include <mpi.h>

int main (int argc, char * argv[])
{
  int rank, size;

  MPI_Init (&argc, &argv);      /* starts MPI */
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);        /* get current process id */
  MPI_Comm_size (MPI_COMM_WORLD, &size);        /* get number of processes */
  printf( "Hello world from process %d of %d\n", rank, size );
  MPI_Finalize();
  return 0;
}
```

Simple communication example. The slides and live coding example contain a second MPI example which demonstrates `MPI_Send` and `MPI_Recv` usage, also found at `http://en.wikipedia.org/wiki/Message_Passing_Interface`.

Matrix multiplication example. We'll next discuss the code from another MPI example. You can find the code at `http://www.nccs.gov/wp-content/training/mpi-examples/C/matmul.c`. I'll discuss the structure of the code and include relevant excerpts. Here are the steps that the program uses to compute the matrix product AB:

1. Initialize MPI, as in the Hello, World example.

2. If the current process is the master task (task id 0):
   (a) Initialize the matrices.
   (b) Send work to each worker task: row number (offset); number of rows; row contents from A; complete contents of matrix B. For example,

```
MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
```

   (c) Wait for results from all worker tasks (`MPI_Recv`).
   (d) Print results.
3. For all other tasks:
   (a) Receive offset, number of rows, partial matrix A, and complete matrix B, using `MPI_Recv`, e.g.

```
MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
```

   (b) Do the computation.
   (c) Send the results back to the sender.

On communication complexity. To write fast MPI programs, keeping communication complexity down is key. Each step from multicore machines to GPU programming to MPI brings with it an order-of-magnitude decrease in communication bandwidth and a similar increase in latency.

Cloud Computing

Historically, 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.
- 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.

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/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`.

[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`.