ECE459: Programming for Performance Winter 2011

Lecture 20 — March 22, 2011

Patrick Lam *version 1*

Clusters and cloud computing

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.

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;
}
```

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 4; it's cheaper to commandeer machines and set them up in my office instead.

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).