Cloud Computing on Amazon's EC2

1. Introduction to Amazon’s EC2

As computer components have become cheaper and more widely available, there is an increasing trend for corporations and scientific research / academic institutions to off-load their data-processing to a high performance computing (HPC) center. These centers, specifically Amazon's EC2, offer a truly virtual computing environment for use of processing power at a premium. Instead of these corporations and institutions purchasing equipment for peak traffic volume or a short-lived scientific experiment, they can instead purchase as much processing power on EC2 as they need and only pay for what they use.

This virtual environment is created by the user in the form of an Amazon Machine Image (AMI). An AMI contains the application(s) the user wishes to run along with any libraries, data and/or configuration settings. It is essentially an all-inclusive operating system. This AMI is bundled up and the user uploads or otherwise transfers the AMI to another of Amazon's products, Amazon Simple Storage Solution (S3). This is the location from which Amazon loads the user's AMI to run on a processor or series of processors, depending on the type and quality of service the user requires.

After an instance is instantiated using a command-line tool or web interface provided by Amazon, this instance has its own machine name, ip address, firewall properties, ethernet interfaces, etc., just like it is its own standalone machine. The instance can then be accessed remotely just as any other machine can be accessed, through the use of ssh and scp protocols.

A user can choose among several different instance types to fit her specific needs. Following is a snippet from Amazon's website:

- **Standard Instances**
  - Instances of this family are well-suited for most applications
  - **Small Instance (Default)**: 1.7GB of memory, 1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit), 160GB of instance storage, 32-bit platform
  - **Large Instance**: 7.5 GB of memory, 4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each), 850 GB of instance storage, 64-bit platform
  - **Extra Large Instance**: 15 GB of memory, 8 EC2 Compute Units (4 virtual cores with 2 EC2 Compute Units each), 1690 GB of instance storage, 64-bit platform

- **High-CPU Instances**
  - Instances of this family have proportionally more CPU resources than memory (RAM) and are well-suited for compute-intensive applications
  - **High-CPU Medium Instance**: 1.7 GB of memory, 5 EC2 Compute Units (2 virtual cores with 2.5 EC2 Compute Units each), 350 GB of instance storage, 32-bit platform
**High-CPU Extra Large Instance** 7 GB of memory, 20 EC2 Compute Units (8 virtual cores with 2.5 EC2 Compute Units each), 1690 GB of instance storage, 64-bit platform

Note: EC2 Compute Unit (ECU) – One ECU provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

EC2 together with S3 provide a high-capacity computing resource worthy of use by virtual parallel computing clusters. Here, it is important to note that when an instance is destroyed, any data generated by that instance is lost. This is where S3 comes into play. It is used as a persistent storage device for instances. S3 holds users’ files in what Amazon calls *buckets*. Amazon provides an API to access user files in buckets and the necessary security requirements of these buckets (i.e. bucket permissions, access, etc.) so that another user cannot purposefully or unwittingly destroy, overwrite, etc. files in another user’s bucket. Since my experiments on Amazon’s EC2 do not use this persistent storage provided by Amazon except to store the AMIs I created for experimentation purposes, I will skip further content for brevity and urge the curious reader to pursue this further on Amazon’s website.

2. **Experiment 1**

The first experiment I chose to help evaluate the performance of Amazon’s EC2 computing environment is a distributed prime search. The intensive processing nature of finding all prime numbers in a certain range is well-suited for parallel processing. First, I started with 1 instance (compute node) and increased this number, one-by-one, to 15, recording the length of time it took each instance to finish its calculations. I repeated the same experiment three times for each number of instances. I then repeated the same experiment with two different instance types (m1.small and c1.medium) and the Linux cluster in Shop 304. (see the accompanying Microsoft Excel document)

One important factor is how the range is divided and given to each instance. On each instance, I ran a java program as follows: `java prime <id> <number of instances> <beginning number> <ending number>`. One way to partition the range is as follows (`<number of instances> = 2, <beginning number> = 3, <ending number> = 10`):

<table>
<thead>
<tr>
<th>Number to Test</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assigned to Instance</td>
<td>&lt;id&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

However, partitioning the range in this way ensures that instances with lower `<id>` always finish before instances with higher `<id>` (i.e. it takes longer to determine if a number is prime if it is large). Instead, the way I chose to partition the range of numbers is as follows (`<number of instances> = 2, <beginning number> = 3, <ending number> = 10`):

<table>
<thead>
<tr>
<th>Number to Test</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assigned to Instance</td>
<td>&lt;id&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

First, notice that even numbers are not checked at all. This speeds up the processing a little, so the program can focus on “real” primes. This spreads out the “larger possible primes” over the instances, giving each instance a more roughly similar number of possible primes to test.

3. **Experiment 2**
In my second experiment, I wanted to find some way to test networking between these instances. In this experiment, I decided to use my implementation of a distributed version of Conway’s Game of Life implemented in libsynk. I tested the performance of an m1.small instance and c1.medium instance against the Shop 304 Linux machines. The data I gathered show the performance of the m1.small instance is eclipsed by both the c1.medium instance and the Linux cluster in Shop 304. The c1.medium instance performed best.

4. Conclusions

As the reader can see from the accompanying Microsoft Excel document and Figure 1 below, the truly parallel Linux cluster at first out-performs Amazon’s c1.medium instance, but soon they level off to provide about the same performance. No significant gain is achieved by adding more instances.

Figure 1

Figure 2 shows the relative performance of the Game of Life experiment using libsynk.

Figure 2

Figure 1 shows that having your own cluster is quite comparable to the c1.medium Amazon image. At 20¢ an hour, that’s not a bad deal, especially if the computing you’re trying to perform will complete in the short term. The m1.small instance, while not entirely a bad deal at 10¢ an hour, it doesn’t match in performance to the c1.medium instance nor the shop 304 Linux cluster.

Figure 2 surprised me a little. I would have expected the Linux cluster to out-perform even the c1.medium instance. My reasoning was thus: Since Amazon charges for bandwidth usage as well, surely
my 100Mbps Linux lab Ethernet connection will be faster than Amazon’s due to the fact that Amazon has some sort of in-house overhead in keeping up with my bandwidth usage. My reasoning is thus corrected. Amazon apparently has little to no overhead in keeping up with my bandwidth usage from instance-to-instance, or they have a much faster connection than 100Mbps. All-in-all, the c1.medium instance outperformed both the m1.small and shop 304 Linux cluster.

Computing surely has come a long way, and Amazon’s EC2 is doing a lot to help improve on the cloud-computing front (no pun intended). With the most powerful instance Amazon has to offer only costing 80¢ per hour, cloud computing will one day no longer be a vision for the future, but a realization that all business owners and researchers alike can use to help them with their short-term computing needs.

Eventually, this technology will come around and perhaps Amazon will one day, like Microsoft, Cisco, A+, etc. [insert technical certification programs here] provide certification programs for computer professionals to help qualify them for jobs, etc. A business looking for an outside hire could advertise they are looking for computer scientists with Amazon EC2 certifications to work for them. This would ease the burden on companies having to train their employees in Amazon’s cloud computing environment. Instead, the employee would already have the certification and be ready to go to work.

Dr Gu, you already have the source code for the distributed Game of Life simulation from homework 2. All I did to modify it to run on the instances was to take out all the code concerning the viewer and recompile it.

Following is my source code for the prime search program:
public static void main(String[] args) {
    id = 0; numInstances = 0;
    BigInteger begNum, endNum, numToCheck;
    if (args.length == 4) {
        try {
            id = Integer.parseInt(args[0]);
        } catch (NumberFormatException e) {
            System.err.println("argument 1 must be an integer
usage: java prime <id> <number of instances> <beginning number> <ending number>\n");
            System.exit(1);
        }
        try {
            numInstances = Integer.parseInt(args[1]);
        } catch (NumberFormatException e) {
            System.err.println("argument 2 must be an integer
usage: java prime <id> <number of instances> <beginning number> <ending number>\n");
            System.exit(1);
        }
    } else {
        System.out.println("usage: java prime <id> <number of instances> <beginning number> <ending number>\n");
        System.exit(-1);
    }
    begNum = new BigInteger(args[2]);
    endNum = new BigInteger(args[3]);
    if (begNum.compareTo(new BigInteger("3")) < 0) {
        System.out.println("error: <beginning number> must be greater than or equal to 3\n");
        System.exit(-1);
    }
    if (begNum.compareTo(endNum) >= 0) {
        System.out.println("error: <beginning number> must be strictly less than <ending number>\n");
        System.exit(-1);
    }
    if (id >= numInstances || id < 0) {
        System.out.println("error: <id> must be on [0, <number of instances> - 1]\n");
        System.exit(-1);
    }
    BigInteger two = new BigInteger("2");
    BigInteger zero = new BigInteger("0");
    BigInteger one = new BigInteger("1");
    BigInteger remainder = begNum.mod(two);
    int num;
    if (remainder.compareTo(zero) == 0) {
        num = (2 * id) + 1;
        numToCheck = begNum.add(new BigInteger(new Integer(num).toString()));
    } else {
        num = 2 * id;
        numToCheck = begNum.add(new BigInteger(new Integer(num).toString()));
    }
    int numPrimes = 0;
    num = numInstances + 2;
    BigInteger interval = new BigInteger(new Integer(num).toString());
    long startTime = System.currentTimeMillis();
    while (numToCheck.compareTo(endNum) <= 0) {
        if (numToCheck.isProbablePrime(100)) {
            numPrimes++;
        }
        numToCheck = numToCheck.add(interval);
    }
    long endTime = System.currentTimeMillis();
    long timeToCompute = (endTime - startTime);
    System.out.println("node id = \nprogram executed for "+ timeToCompute + " milliseconds\nfound "+ numPrimes + " primes\n");
}