CIEL Tutorial

This page provides instructions for setting up your own CIEL installation on Amazon's Elastic Compute Cluster (EC2), and running some basic Skywriting jobs.

Contents

Connecting to your personal cluster

You should have received some instructions containing a link to a tarball that holds the necessary files for this tutorial. Inside the archive you will find:

- The CIEL/Skywriting distribution
- EC2 API command line tools
- Security certificates for connecting to AWS
- Scripts for setting up the environment

You will first need to extract the contents of the archive to a folder on your local machine. This will create a folder named `ec2-bin`. The `ec2-config.sh` script defines a set of environment variables that are needed in order to utilise the EC2 command line tools.

```
tar zxf dcn-ciel-2012.tgz
cd ec2-bin
. ec2-config.sh
```

The environment variables have now been loaded into the current shell session. The next step is to create a personal private key for your own cluster. The following command will request and download a private key with the filename `pk-[userid]` into the current folder.

```
./create-key.sh pk-$USER
```

The machines in your cluster (referred to as "instances") will be distinguished from other students' machines using this private key. The next step is to actually provision the cluster.

```
./cluster-setup.sh pk-$USER
```

The above command will request five "on-demand" instances and provision these with a basic Ubuntu system image. For this tutorial, only "micro" instances with minimal resources will be used. The script creates three files in the local directory:

- `hosts-pub` - contains the public DNS hostnames of each instance
- `hosts-prv` - contains the private (EC2 internal) hostnames of each instance
- `hosts-ins` - contains the instance IDs utilised by the EC2 API for each instance

After the command has completed, the request will take some further time to process (usually a few seconds). The following command can be used to check if all of the instances have been created.

```
for i in `cat hosts-pub`; do
 ssh ubuntu@$i -i pk-$USER \
               -o "ConnectTimeout 1"\
               -o "StrictHostKeyChecking no"\
               "echo \`hostname\` up";
done
```

The command should produce five `[hostname] up` messages. Once all of the instances have been provisioned, we can begin deploying the CIEL software to the cluster.

```
./deploy-sw.sh pk-$USER
```

The CIEL dependencies will take some time to install and configure (usually several minutes). During this period there may be a great deal of logging output on your console. Most of this can be safely ignored, however once the messages stop appearing, this is an indication that the process has completed. In the meantime, you should familiarise yourself with the rest of the tutorial.

Following the successful completion of the software deployment process, we can connect to the cluster. We will use the first instance as our master node.

```
master=`head -n 1 hosts-pub`
scp -i pk-$USER hosts-prv pk-$USER root@$master:/opt/skywriting/
ssh -i pk-$USER root@$master
```

You are now connected to an EC2 instance, running Ubuntu 10.04. Since this is a virtual machine, you have full root access, so you may want to install some additional packages using `sudo apt-get install`. (For example, I typically install emacs and lynx on the master, but your taste may vary.)

Launching a CIEL cluster

Given a list of hostnames, the CIEL cluster management scripts (`sw-launch-cluster`, `sw-update-cluster` and `sw-kill-cluster`) are designed to launch the master on the first host, and workers on the remainder of the hosts. You can run the cluster management scripts from any host that has a private key for accessing your cluster hosts, but in this tutorial we assume that you are running them from the master (i.e. the host named on the first line of your hostnames files).

To launch your cluster, type the following (assuming that your current directory is `/opt/skywriting`, substituting your userid for [username]:

```
scripts/sw-launch-cluster -f hosts-prv -i pk-[username]
```

You should see some initial logging output on your terminal. To confirm that your cluster has started successfully, wait a few seconds then type the following:

```
curl -s -S http://localhost:8000/control/worker/ | grep netloc | wc -l
```

The result should be the number of workers in your cluster (i.e. the number of lines in `hosts-pub`, minus one for the master). At this point, if the result is zero, there is a problem with your configuration, so let the supervisor know. If the result is fewer than you were expecting, carry on for the moment, because you should still be able to run tasks on some of the machines, and the supervisor will take a look at your setup later on.

Hello world in Skywriting

The simplest Skywriting script returns a value directly. Create a file called `helloworld.sw` with the following contents:

```
return "Hello, world!";
```

To run the script, type the following command:

```
scripts/sw-job -m http://`hostname -f`:8000/ helloworld.sw
```

The `sw-job` script is a utility for launching simple Skywriting scripts that do not have any external dependencies. We will use it to launch the Skywriting scripts in the remainder of this section. Typing the `-m` flag each time rapidly becomes tedious, so you can avoid it by setting the `CIEL_MASTER` environment variable:

```
export CIEL_MASTER=http://`hostname -f`:8000/
scripts/sw-job helloworld.sw
```

A more-realistic Skywriting script will `spawn()` one or more tasks. Create a file called `helloworld2.sw` with the following contents:

```
function hello(who) {
 return "Hello, " + who + "!";
}
salutation = spawn(hello, ["world"]);
return *salutation;
```

Using non-Skywriting executors

The next example shows how a script can make use of non-Skywriting executors. Create a file called `linecount.sw` with the following contents (but don't attempt to run it just yet):

```
include "stdinout";
lines = stdinout([package("input1")], ["wc", "-l"]);
return *lines;
```

The `stdinout` function invokes the `stdinout` executor to integrate legacy applications in a CIEL job. The function takes a list of *references*, and a command line (as a list of strings). In the above example, the single input reference is the value of `package("input1")`, and the

task executes `wc -l` on that reference. (When more than one reference is specified, the contents of those references are concatenated together, as if they were multiple inputs to `cat`. The implementation of the `stdinout` function can be found in `src/sw/stdlib/stdinout`).

Since the result of `wc -l` is an integer (and hence valid JSON), we can dereference it using the * operator. How does Skywriting resolve `package("input1")`? The answer is that it must be supplied as an *external reference*. In general, the job package mechanism is used to provide a key-value dictionary of files, URLs, lists of files and lists of URLs. However, in this case, we can use the `sw-job -p` option to define the package manually:

```
apt-get install wcanadian-insane wbritish-insane wamerican
scripts/sw-job linecount.sw -p input1 /usr/share/dict/words
```

Let's now try a script that performs some analysis in parallel. Create a file called `linecount-parallel.sw` with the following contents:

```
include "stdinout";
inputs = [package("input1"), package("input2"), package("input3")];
results = [];
for (input in inputs) {
      results += stdinout([input], ["wc", "-l"]);
}
total = 0;
for (count in results) {
      total += *count;
}
return total;
```

Execute the job using the following command.

```
scripts/sw-job linecount-parallel.sw -p input1 /usr/share/dict/canadian-
english-insane \
    -p input2 /usr/share/dict/british-english-insane \
    -p input3 /usr/share/dict/american-english
```

The above script is a simple example of a MapReduce coordination pattern: the lines are counted in parallel, then reduced (added together). In the following section, we will see how to build a more general form of MapReduce from scratch.

Implementing MapReduce

We have just seen how to implement a simple form of MapReduce using a combination of command-line utilities and Skywriting scripts. However, in general, this combination does not offer a sufficiently comprehensive collection of libraries etc. for developing realistic applications. In this section, you are going to implement a fuller version of the MapReduce, using Python. (If you aren't happy using Python, you can use any programming language of your choice for this part of the practical. However, you will have to implement the simple interface for accessing task inputs and outputs).

A simple test harness that creates the appropriate task graph exists in the `examples/MapReduce` directory. The `mapreduce.sw` script is as follows (note that you should not have to edit this, at least at first):

```
include "mapreduce";
include "environ";
include "stdinout";
include "sync";
```

These include statements allow functions from the "standard library" to be used in the script. The corresponding implementations are insrc/sw/stdlib/*.

```
function make_environ_map_task(num_reducers) {
    return function(map_input) {
        return environ([map_input], [package("map-bin")], num_reducers);
    };
}
function make_environ_reduce_task() {
    return function(reduce_inputs) {
        return environ(reduce_inputs, [package("reduce-bin")], 1)[0];
    };
}
```

These higher-order functions are constructors for the map and reduce tasks. They define the template for each kind of task: a map task takes a single input and produces `num_reducers` outputs; a reduce task takes many inputs (one per map task) and produces a single output. The `environ` function (and `env` executor) are similar to the `stdinout` version, except that they provide individual access to multiple inputs and outputs, by using the environment variables as an indirection.

```
inputs = *package("input-files");
num_reducers = int(env["NUM_REDUCERS"]);
```

These are the input parameters. Notably, `package("input-files")` is a reference to a *list of references*, which makes it simpler to include variable-length inputs.

```
results = mapreduce(inputs, make_environ_map_task(num_reducers),
make_environ_reduce_task(), num_reducers);
catted_results = stdinout(results, ["/bin/cat"]);
return sync([catted_results]);
```

These statements perform the execution: first a MapReduce graph is built using the task constructors, then the results of the reduce tasks are `cat`ted together, and finally a `sync` task is used to force execution of the whole graph.

To execute a MapReduce-style job, you need to invoke a package (rather than the script directly). The is done as follows:

```
scripts/sw-start-job examples/MapReduce/mapreduce.pack \
    PATH_TO_MAPPER_EXECUTABLE \
    PATH_TO_REDUCER_EXECUTABLE \
    PATH_TO_INPUT_INDEX \
    [NUMBER_OF_REDUCERS=1]
```

The `PATH_TO_INPUT_INDEX` is the name of a file containing a list of filenames (one per line), for each of which a mapper will be created. The mapper and reducer executables can be simple Python scripts, for example:

```
echo /usr/share/dict/american-english > dicts.txt
echo /usr/share/dict/british-english-insane >> dicts.txt
echo /usr/share/dict/canadian-english-insane >> dicts.txt
scripts/sw-start-job examples/MapReduce/mapreduce.pack \
    examples/MapReduce/src/python/count_lines.py \
    examples/MapReduce/src/python/total.py \
    dicts.txt
```

This performs the same calculation as the example using stdinout and Skywriting. However, you have a lot more flexibility as to what you can do in the Python scripts. At this point, try writing Python mapper and reducer scripts to perform a more interesting calculation.

Extensions

Using `examples/MapReduce/src/python/count_lines.py` and `examples/MapReduce/src/python/total.py` as starting points, try to write a program that calculates the probability of different *k*-grams in a text corpus, where *k* is a constant number of characters. In the mapper, you will have to do the following:

- Loop through the input file, identifying all of the subsequences of length *k*, and update a data structure holding the counts.
- Serialise the data structure (perhaps as plain text, to simplify matters) to one or more of the mapper's outputs.

In the reducer, you will have to do the following:

- Read and deserialise the data structures from each mapper, and update a global data structure.
- Serialise the data structure (as plain text) to the single reducer output.

`examples/MapReduce/src/python/count_lines.py` illustrates how to read from inputs and write to outputs: `fcpy.inputs` is a list of file-like objects that are open for reading, and `fcpy.outputs` is a list of file-like objects that are open for writing. Since you are dealing with text, the `fcpy.inputs[i].readlines()` function will be useful for reading input, and the `fcpy.outputs[i].write()` function will be useful for writing output.

By now, you should know enough about Skywriting and CIEL to implement your own applications. If any time remains, you might want to try the following exercises:

- A real MapReduce provides an `emit()` function that takes an arbitrary key-value pair and writes it out to the appropriate reducer (one of many). Can you implement the data structures that are necessary to do this?
- It is common to pre-sort the output from the mappers in order to lower the memory requirements at the reducers (allowing them to perform a streaming mergesort). Try to implement this in your `emit()` function and reducer.
- Often, the reducer performs a commutative and associative aggregation, which means that the reduce function can be applied early to mapper outputs. This is usually called a *combiner*. Can you implement a combiner for your mapper tasks?
- CIEL supports a greater diversity of patterns than simple MapReduce. For example, it is possible for MapReduce tasks to have more than one input. By modifying the

Skywriting script `mapreduce.sw`, can you add a secondary data set to the mappers? This would allow you to perform a Cartesian product, which underlies many data analysis algorithms, such as arbitrary joins. Can you implement other join algorithms, such as hash-joins and broadcast joins?

- The key feature of CIEL is its ability to perform data-dependent control flow. Therefore, it is often used for algorithms that iterate until a convergence criterion is reached. Can you implement an iterative algorithm using your MapReduce primitives? For example, try generating a random graph, then implementing the PageRank or Single-Source Shortest Paths algorithms on it.

The fcpy interface

If you want to implement this part of the practical in another language, you will need to implement the equivalent of the fcpy interface:

```
import os
import sys
# Task inputs are accessed through fcpy.inputs; similarly for outputs.
inputs = None
outputs = None
def init():
    """Must be called at the start of a task to configure the
environment."""
    global inputs, outputs

    try:
        # $INPUT_FILES contains a single filename; similarly for
$OUTPUT_FILES.
        input_files = os.getenv("INPUT_FILES")
        output_files = os.getenv("OUTPUT_FILES")

        if input_files is None or output_files is None:
            raise KeyError()

        # Each line of `cat $INPUT_FILES` is a filename corresponding to
one of the task inputs.
        with open(input_files) as f:
            inputs = [open(x.strip(), 'r') for x in f.readlines()]

        # Each line of `cat $OUTPUT_FILES` is a filename corresponding to
one of the task outputs.
        with open(output_files) as f:
            outputs = [open(x.strip(), 'w') for x in f.readlines()]
```