Parallelizing data processing on cluster nodes with the `Distributor`

Andreas Bernauer

February 20, 2004

1 Introduction

This article describes how to use the Perl module `Distributor.pm` with its counterpart `Collector.pm` to parallelize data processing on a cluster with several nodes which are accessible via `ssh`. `Distributor.pm` depends on the Perl module `Net::Server::Single`.

Basically, there are different approaches on how to parallelize calculations and thus reduce the time necessary to get results. `Distributor` uses the approach of splitting the data you want to process in smaller chunks that are handled by each node, independently from the other nodes. If you cannot split your data into smaller chunks, `Distributor` won't be helpful for you.

2 The big picture

`Distributor` works uses a client-server-architecture. That is, there is somewhere a server running that provides the data chunks, and there are somewhere several clients running that repeatedly connect the server, read the next data chunk(s), disconnect from the server and process the provided data chunk(s).

In most cases the server won't deliver the actual data chunks, but a hint for the client so it knows what the next data chunk is. Usually, this will be the name of an input file. That means, the server provides the client with an input file, which the client may find somewhere on its file system, because it has been copied there previously. This works well with client nodes that share the same file system.

The definition for the server are placed in the Perl module `Distributor.pm`. Another Perl module, `Collector.pm` provides some helpful routines to read data from the server. You are not forced to use `Collector.pm`.

3 Setup

The following subsection will tell you how to write the server and the clients.

3.1 Server setup

Here's an example server which provides the data items 3, 6, 9 and 12:

```
#!/usr/bin/perl -w

use Distributor;
@data = qw ( 3 6 9 12);
@nodes = qw ( c1n2.math.uconn.edu );

sub nextItem { return shift @data; }

runDistribution( "c1n1.math.uconn.edu", 49876, "neynex", 3,
                 \&nextItem, "gogarten", @nodes,
                 "./client.pl");
```

The perl program opens the module for the `Distributor`, defines the array of data items, the array of nodes to which connect and a function that delivers the next data item. The last command launches the server which is explained in detail here.

The arguments to `runDistribution` are the following:

Server hostname (here `"c1n1.math.uconn.edu"`) specifies on which machine the server is running.

Server port (here 49876) specifies the port on which the server will wait for connections. Later, the clients will use the server hostname and the server port to connect to the server.

Password (here `"neynex"`) specifies a password for the shutdown of the server. The server allows you to connect to it and to tell it to shutdown. To disallow anybody to shutdown your server, the server expects a password for a simple authentification. Don't use a password that you use otherwise, as the `Distributor` doesn't use any encryption, *i.e.* clients transmit the password plain over the network.

Data chunks per connection (here 3) specifies how many chunks of data the server should send to the client per single connection. Depending on how fast your clients can process the data chunks, different values are sensible for this parameter.

Suppose your clients can process the data chunks pretty fast. As at every single point in time there can be only one client connecting to the server, a small amount of data per connection will result in frequent client connections. The server might not be able to handle the connections fast enough, leading to a long waiting queue of clients.

So, if your clients can process the data chunks pretty fast, you want to transmit more data items per client connection. If the clients need a long

time to process a data chunk or if the time varies a lot between different data chunks, you want to transmit only a few, maybe only one data item per connection.

nextItem-Subroutine (here `\&nextItem`). The server will repeatedly call this subroutine to retrieve the next data item. The subroutine must return a one-lined string representing a single data chunk. A one-lined string does not containt any kind of newline, *i.e.* no '`\r`' or '`\n`' characters. If there are no more data chunks, the subroutine must return `undef` instead. Note that the server might call the subroutine although it already has indicated that there are no more data items left. The subroutine is supposed to repeatedly return `undef` in that case.

Username (here `"gogarten"`) is the user name the server will use if it connects the nodes.

Nodes (here `@nodes`, can contain of course more than one machine name) is an array of names of the nodes, to which the server will connect per ssh. The machines must meet the following conditions: The server can contact them via `ssh`, it can contact them with the provided username (*i.e.* the same username is used for every node) and the server does not need a password to contact them. For the later condition, I suggest to use `ssh-agent` and public-key authentification.

Remote command (here `"./client.pl"`) is the command the server executes on the nodes. The command should eventually run a client that connects to the server, otherwise the whole setup does not make a lot of sense.

When you call `runDistribution`, two things happen simultaneously: the server is set up and the clients are launched. The setup of the server depends on the behavior of `Net::Server::Single`, but usually only means that the server is waiting for connections on the specified server hostname and port combination.

For each node to connect, *i.e.* for each client to start, `runDistribution` connects to the specified node via `ssh` and executes the remote command. It also connects stdout and stderr of the remote command to two local logfiles, to which it appends. Examine these logfiles, if your clients are not running as you supposed them. Note that clients running on the same node (which is sensible on multi-processor machines) share the same logfile.

3.2 Client setup

As you've seen from the code of the server, the server will launch a program call `./client.pl` on the nodes. Thus, we need a program of this name accessible on each node.

Here's an example for such a client:

```
#!/usr/bin/perl -w

use Collector;
my @data;
my ($remote_host, $remote_port) = @ARGV[-2,-1];
while (@data = readData($remote_host, $remote_port)) {
  for (@data) {
    print "$_\n";
  }
}
```

The script uses the Perl module `Collector` which comes with the `Distributor`. Actually, you can use your own code, as the server uses a specific protocol to transfer the files. The protocol is described in the source file of the server. However, the easiest way for you is to just use what `Collector.pm` provides you.

The example script defines the server host name and port as the last two arguments of the command line. This is because the server will call the client with its hostname and hostport. After defining the hostname and port of the server, the script enters into a `while`-loop. This loop continuously reads data items into the `@data` array with the function `readData` provided by the `Collector` module. This function will return `undef` if there no more data items and thus, the loop will be exited. If `readData` could get some data items from the server, the example script just prints them.

4 Running

Suppose you copied the scripts of the previous section into two files named `server.pl` (the server) and `client.pl` (the clients), respectively. Suppose further, that you have copied the server file in your home directory of `c1n1.math.uconn.edu`, as this is the machine that will run the server, and the client file in your home directory of `c1n2.math.uconn.edu`, as this is the machine that will run the client. You have to make sure that the client file is accessible at every node at the same location. This is easy to do, if the nodes share the same file system (as it is the case at the machines mentioned), but this might not be the case at your machines.

As mentioned before, the server will contact the nodes via `ssh` using public-key authentication. Your system administrator or some friend will help you generating and distributing public keys to the machines. At the machines mentioned above this has already been done for the user `gogarten`.

Now, login to the machine, that will run the server, here `c1n1.math.uconn.edu` and activate the public-key authentication system using the `ssh-agent`, *i.e.* you run the command

```
[gogarten@c1n1 ~]$ ssh-agent $SHELL
```

Then, you activate your public-key with `ssh-add`, *i.e.* you run the command

```
[gogarten@c1n1 ~]$ ssh-add
Identity added: /macusers/guests/gogarten/.ssh/id_dsa
 (/macusers/guests/gogarten/.ssh/id_dsa)
```

The line starting with "`Identity added`" tells you that your key has been registred by the agent.

After setting up the public-key authentication system, you are ready to run the `Distributor`. Say

```
[gogarten@c1n1 ~]$ ./server.pl
```

and the server will print out some diagnostic messages like:

```
Launching on node: c1n2.math.uconn.edu
2004/02/18-17:05:37 Distributor (type Net::Server::Single) starting!
pid(14601)
Binding to TCP port 49876 on host c1n1.math.uconn.edu
Setting gid to "99 99"
Couldn't become gid "99" (2000)
All data items delivered (Wed Feb 18 17:05:37 2004)
```

After the line "All data items delivered" appears, you know exactly that, *i.e.* every data item your `nextItem` function was delivering to the server has been delivered to a client. You *don't* know however, if every client has finished its job. If you know, that every client has finished its job, you can stop the server. The easiest way to do that is to abort the corresponding process, *i.e.* hitting Ctrl+C on your keyboard.

The output of our client is saved in a log file called. The logfile has the name of the node with the extension `.log` appended to it. Its contents look like that:

```
*** Wed Feb 18 17:05:37 2004: New subprocess started.
3
6
9
12
```

The client has printed out every data item. Of course, usually the result of your client's calculation won't be some output, but a file the client has created.

If there are errors, you can find them in another logfile with the extension `.err.log` instead of `.log`. In this example there was no error, so the error logfile is virtually empty:

```
*** Wed Feb 18 17:05:37 2004: New subprocess started.
```

It just tells you, that a client has been launched.

If you alter the server script and use more nodes, you will get more logfiles, two for each node. If your nodes are multiprocessor machines (as it is the case with the `.math.uconn.edu` cluster), you can specify a node twice (or more often, if your nodes have more processors) to use every processor of your node. However, the clients on each node will share the same logfile, so if you depend on the clients output to the logfile, you want to take some measure to ensure you know which output is from which client.

5 Recipe

Here's a short step-by-step description on how to setup your own distributor, using the example scripts given above.

5.1 Recipe for the server

Use the example script mentioned above and do the following:

1. Load the `@data` array with the values you want to pass to the clients, either by hardcoding it into the script or loading it from some source, *e.g.* a database. Unless you have a lot of data items which don't fit into the server's memory, I suggest to load every data item into the `@data` array before launching the server. Particularly, using the diamond operator in the `nextItem` subroutine did not work for me (for some reason I don't know). As an examle, I loaded the `@data` array with filenames that I used as input for some other program and copied the necessary files to the nodes.

2. Write the list of the nodes into the `@nodes` array. Note you can mention nodes several times. This makes (only) sense if your nodes have more than one CPU.

3. Maybe rewrite the `nextItem` subroutine. If you use the `@data` array approach, you don't have to modify the subroutine, otherwise you have to make sure the subroutine reportedly returns `undef` once it has delivered all data items (`shift has this feature`).

4. Adjust the arguments of `runDistribution`. Change the server name to the name of the machine the server will be running on. Adjust the number of data items to deliver to your needs; 3 is not suitable for every case (see above for details). Make sure you use a username with which you can log into every node. Maybe adjust the remote command, if the script is not in your home directory, but in some other directory.

5. Upload the server script to the machine mentioned in the first argument of `runDistribution`. Make sure, `Distributor.pm` is in the library path of Perl, *i.e.* in a directory that is in the array `@INC`. If not, insert a line

`use lib "path/to-Distributor"` as the second line of the server script, with `path/to-Distributor` being the path in which `Distributor.pm` is.[1]

5.2 Recipe for the client

Use the example script mentioned above and do the following:

1. Change the body of the `for` loop to your needs. Note that you need *both* the `while` and the `for` loop to ensure that you process all data items.

2. Upload the client script to every node mentioned in the server script, so that the remote command of the server will execute it. Make sure, `Collector.pm` is in the library path of Perl, *i.e.* in a directory that is in the array `@INC`. If not, insert a line `use lib "path/to-Collector"` as the second line of the client script, with `path/to-Collector` being the path in which `Collector.pm` is.

5.3 Recipe for running the `Distributor`

Follow the following steps to run the `Distributor`:

1. Log into the machine of the server script.

2. Make sure, your authentication agent is running. If in doubt, start it with the commands

```
[gogarten@c1n1 ~]$ ssh-agent $SHELL
[gogarten@c1n1 ~]$ ssh-add
Identity added: /macusers/guests/gogarten/.ssh/id_dsa
 (/macusers/guests/gogarten/.ssh/id_dsa)
```

3. Change to the directory where the server script is and launch the server:

```
[gogarten@c1n1 ~]$ ./server.pl
```

   You will get bunch of diagnostic messages in your terminal and a bunch of logfile created in the directory of the server script. A message like "All data items delivered." will appear, after all data items have been delivered. There is program called `checkstat.pl` which can tell you how many data items are already delivered. Contact the author, if you want to use it.

If any error occurs or you don't get the results you expected, check the error logfiles of each node.

[1] At the example nodes, `c1n1.math.uconn.edu`, I've already done that.

6 Summary

This article showed with the help of an example setup, how to use the `Distributor` to distribute calculation jobs that can be run independently from each other on several nodes. A user writes to scripts, one for the server and one for the clients. After the server script is launched, the server script will launch the client script on each node, collecting its normal and error output into a logfile.

The clients are supposed to contact the server and ask for further data items until there are no more data items. Thus, the data items can be processed as fast as possible, as no client has to wait for another client to finish its job.

Questions and comments can be sent via email to the author at `andreas@carrot.mcb.uconn.edu`.