PSim and web2py

Eric D. Bratt, DePaul University, ebratt@mail.depaul.edu

Abstract

As part of this project, I have implemented a web front-end to Massimo DiPierro’s parallel algorithm simulator, PSim, using web2py[[1]](#footnote-2). The objective is to provide the user with the ability to setup some basic parameters for PSim, run some simulations of some basic algorithms with different types of input data, and see the output of the algorithms along with their respective run times (clock-on-the-wall). The primary mechanism for executing these simulations is web2py’s built-in Scheduler[[2]](#footnote-3). A set of helper scripts have been developed to wrap around the PSim library. These two features allow PSim to fork multiple processes on the server’s operating system separate from the web2py process.

1. Introduction

Massimo Di Pierro created a way to simulate parallel processing with a python module called PSim. PSim is written in Python. Having experimented with both web2py and PSim at DePaul University, I decided to marry the two with a web2py application called psim2web2py[[3]](#footnote-4). What is web2py? Web2py is a web application framework, written in Python. According to the website’s About page[[4]](#footnote-5),

web2py was inspired by Ruby on Rails and, as Rails, it focuses on rapid development and follows a Model View Controller design. web2py differs from Rails because it is based on Python (thus it is faster and more scalable), because it provides a comprehensive web-based administrative interface (thus there is no need to ever type shell commands unless you wish), includes libraries to handle more protocols (for example XML-RPC and RSS feeds), and can run on the Google App Engine.

web2py was also inspired by Django and, as Django, it has the ability to generate forms from database tables and it includes an extensive set of validators. web2py differs from Django because it is more compact, easier to learn and does not have any project-level configuration files.

web2py is less verbose than Java-based frameworks and its syntax is much cleaner than PHP-based frameworks. This makes applications simpler to develop, and easier to read and maintain.

Readers can visit a page[[5]](#footnote-6) on the website to see how web2py compares to other well-known web application frameworks.

1. Approach / Methodology
   1. Hardware

The table below demonstrates the specific hardware used in this experiment.

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Version** | **Source** |
| Machine | ACPI x86-based PC | Latitude | Dell |
| Virtual Client | Oracle VirtualBox | 4.3.20 r96997 | Oracle Corporation |
| Host Operating System | 32-Bit | Windows 7 Professional, SP 1 | Microsoft |
| Processor | 4 x Intel® Core™ | i7 | Intel |
| Memory | 4.00 GB (3.24 GB Usable) |  |  |
| Hard-Disk | 298 GB (219 GB Free) |  |  |

* 1. Software

While pwim2web2py can run on many operating systems, the table below demonstrates the specific software used in this experiment.

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Version** | **Source** |
| Client Operating System | Linux crunchbang | 3.2.0-4-686 pae #1 Debian 3.2.65-1+deb7u1 i686 GNU/Linux | crunchbang.org |
| IDE | PyCharm Professional Ediction 4 | Build #PY-139.1001, built on January 7, 2015 | jetbrains.com/pycharm |
| Runtime | Java™ Runtime SE Environment | 1.6.0\_34-b34 i386 | www.oracle.com[[6]](#footnote-7) |
| Virtual Machine | Java™Virtual Machine | Open JDK Server VM | Sun Microsystems Inc. |
| Web Application Framework | web2py | 2.9.12-stable+timestamp.2015.01.17.06.11.03 | web2py.com |
| Interpreter | Python | 2.7.3 | python.org/download/releases/2.7.3 |
| Version Control | git | 1.7.10.4 | packages.debian.org/wheezy/git |

* 1. Implementation

PSim calls os.fork() to spawn multiple processes for parallel algorithm execution. Since os.fork() is not available for the Windows operating system[[7]](#footnote-8), I decided the smoothest way to implement psim2web2py was to start with a \*nix virtual machine. I have had many positive experiences with Oracle’s Virtual Box and it is free so I decided to install Virtual Box on my Windows laptop. Next I had to decide which \*nix distribution to use. I tried a few (Ubuntu, slitaz, and curnchbang) and found that Ubuntu provided way too many features that I didn’t need and was bigger than crunchbang (981 MB) and slitaz had trouble playing nicely with Virtual Box. The crunchbang ISO file is small (806,850,560 bytes on my machine) and I was able to get it to work very well with Virtual Box (1024 MB of base memory, 2 processors, 128 MB of video memory, 3D acceleration, and 8 GB of storage in a VDI file). Additionally, installing Virtual Box’s Guest Additions allowed me to work nicely in full-screen mode. The end result is a very responsive, small and quick virtual Linux machine running on my Windows OS.

The PyCharm IDE allows me to setup a Run/Debug configuration calling the web2py.py script with python, passing-it the necessary parameters to a) have access to the web2py admin console and b) run a Scheduler process alongside the web2py server. With web2py and Python, the implementation went smoothly and efficiently. I didn’t have to deal with a lot of headaches that many developers have (using the right interpreter, compiler options, build/make, etc.). I was able to implement a feature at a time, test it in a web browser, commit my changes, and move on to the next feature in rapid fashion.

* 1. Background Information

For this project, I had to brush-up on Python, web2py (specifically the Scheduler), and interacting with PSim. My primary sources of information were the Python docs[[8]](#footnote-9), the web2py book[[9]](#footnote-10) (they are permanently bookmarked in all my browsers), and my class notes for PSim. I spent most of my time getting to know the web2py Scheduler. I discovered that the web2py community is very strong and there are many great resources on the web if one is trying to learn about a particular web2py topic in a very short period of time (many thanks to Niphlod[[10]](#footnote-11) for his Scheduler Tests tutorial[[11]](#footnote-12)). This application gave me the background and insight i needed to implement the Scheduler for psim2web2py.

What is the web2py Scheduler? [*Much of the following is paraphrased from the web2py documentation, primarily for my own edification*]

The Scheduler is a set of tables and an API to a library of functions and a Python class that provides a way to run multiple Python processes (called Workers) on different machines (or the same machine). Each Worker is assigned a unit of work (called a Task) that - along with the Worker’s state - is persisted in the database. This allows a web2py administrator to easily monitor Tasks and Workers. The tables that come along with the Scheduler are:

|  |
| --- |
| scheduler\_worker(  id, # id of the worker  worker\_name, # name of the worker  first\_heartbeat, # date-time of the first heartbeat of the worker  last\_heartbeat, # date-time of the last heartbeat of the worker  status, # status of the worker (ACTIVE, DISABLED, TERMINATE, KILLED)  is\_ticker, # determines if the next available task will be given to the worker  group\_names, # groups to whom the worker belongs  worker\_stats # statistics about the worker stored in JSON  ) |
| scheduler\_task(  id, # id of the task  application\_name, # in case there are multiple apps using the same scheduler  task\_name, # name of the task  group\_name, # task group name  status, # tasks status (QUEUED, ASSIGNED, EXPIRED, RUNNING, TIMEOUT, # FAILED, COMPLETED)  function\_name, # name of the function to call  uuid, # The UUID of the task  args, # arguments passed to the task as a Python list  vars, # variables passed to the task as a Python dictionary  enabled, # tells whether the task is enabled or not  start\_time, # date-time to start the task  next\_run\_time, # date-time the task is scheduled to run next  stop\_time, # date-time that the task is scheduled to stop  repeats, # number of times to repeat the task  retry\_failed, # number of times to retry the task if it fails  period, # number of seconds to pass between repeated executions  prevent\_drift, # overrides the period with start\_time  timeout, # number of seconds to wait until the task times-out  sync\_output, # allows the reporting of percentages complete  times\_run, # number of times the task was run  times\_failed, # number of times the task failed  last\_run\_time, # date-time of the last time the task was run  assigned\_worker\_name # name of the worker assigned to the task  ) |
| scheduler\_run(  id, # id of the run  task\_id, # id of the task  status, # status of the run (RUNNING, COMPLETED, FAILED, TIMOUT)  start\_time, # date-time when the run started  stop\_time, # date-time when the run stopped  run\_output, # output of the run, serialized in JSON  run\_result, # result of the run  traceback, # traceback of the thrown Exception, if any  worker\_name # name of the worker for whom the task is run  ) |

A Task is scheduled in web2py with:

|  |
| --- |
| scheduler.queue\_task(  function,  pargs, # stored as a Python list  pvars, # stored as a Python dictionary  start\_time=now,  stop\_time=None,  timeout=60,  prevent\_drift=False,  period=60,  immediate=False,  repeats=1  ) |

The Workers are started when the web2py server is started with:

|  |
| --- |
| python web2py.py -K psim2web2py |

I used the following to allow the Scheduler to run alongside the web2py server:

|  |
| --- |
| python web2py.py -a password -K psim2web2py –X |

By default, the Scheduler will check its status on the scheduler\_worker table every 3 seconds and see if there are any **ASSIGNED** Tasks to itself to process. There are other default settings when instantiating the Scheduler class; I chose to leave all the defaults alone.

Tasks have a life-cycle and may be in one of seven statuses:

* **Queued**: this is the initial status
  + **Assigned**: if a Task is assigned to a Worker before its designated stop time or
  + **Expired**: if a Task is not picked up by a Worker before the designated stop time
    - **Running**: this is the status when the Task is picked up by a Worker
      * **Timeout**: when more seconds pass than the designated timeout period allows
      * **Failed**: when an exception is detected
      * **Completed**: when the Tasks is successfully completed on-time and without exceptions

In psim2web2py I decided to NOT designate any non-default parameters for the Task methods. For example, if a Task fails or times-out, psim2web2py does NOT re-queue it again. Tasks are assigned immediately and Workers pickup what they can when they become available. Note that this is NOT what is meant by parallel processing in the context of PSim. This just has to do with having multiple Workers pickup scheduled Tasks submitted by one or more psim2web2py users.

A Tasks’ status can be retrieved with:

|  |
| --- |
| scheduler.task\_status(  ref, # reference to the Task by scheduler\_Task.id,  # scheduler\_Task.uuid, or by query string  output=False  ) |

Workers also have a life-cycle and may be in one of four statuses (commands):

* **Active**: this is the initial status when the Worker is created (or when it is resumed)
  + **Disabled**: this is when a Worker is asleep because it hasn’t been given a Task for 30 seconds
  + **Terminate**: the Worker will die as soon as it can
  + **Kill**: the Worker will die right away

The Workers in psim2web2py do not prioritize any Tasks over others. All simulation Tasks are treated the same. Scheduling of all simulation Tasks in psim2web2py is done using the API and NOT by inserting rows in the db.

Each Task can have multiple runs (stored in the scheduler\_run table in the Scheduler's db). Runs have a life-cycle and may be in one of four statuses:

* **Running**: when a Task's Job is running
* **Completed**: when a Task's Job is completed the Task is marked as Queued or Completed
* **Failed**: when a Task's Job failed
* **Timeout**: when a Task's Job timed-out

The Run table can be used to see the results and output of a Task. The output of the Task is serialized in JSON format in the Run table. If the Task throws an exception, the Run and the Task are marked as **Failed**. Additionally, the Run shows the traceback as to what Exception was thrown, where it was thrown, and why it was thrown. When a Run exceeds the timeout the Run and the Task are marked as **Timeout**.

When queuing a Task in psim2web2py, I decided to set the retry\_failed to 3 in most cases. This is so that, if multiple Tasks are queued simultaneously and there aren't enough Workers to carry-out all the Tasks AND a Task times-out, it will be re-queued at least 3 times before failing.

1. Experimental Design
   1. Task Functions
   2. Tasks Shell Scripts
2. Experimental Results
   1. Performance Comparisons

1. http://www.web2py.com [↑](#footnote-ref-2)
2. http://web2py.com/books/default/chapter/29/04/the-core?search=Scheduler#web2py-Scheduler [↑](#footnote-ref-3)
3. https://github.com/ebratt/psim2web2py [↑](#footnote-ref-4)
4. http://www.web2py.com/init/default/what [↑](#footnote-ref-5)
5. http://www.web2py.com/examples/static/web2py\_vs\_others.pdf [↑](#footnote-ref-6)
6. http://www.oracle.com/technetwork/java/javase/downloads/java-archive-downloads-javase6-419409.html#jre-6u34-oth-JPR [↑](#footnote-ref-7)
7. https://docs.python.org/2/library/os.html#os.fork [↑](#footnote-ref-8)
8. https://docs.python.org/2/library/index.html [↑](#footnote-ref-9)
9. http://web2py.com/book [↑](#footnote-ref-10)
10. http://www.web2pyslices.com/person/show/niphlod [↑](#footnote-ref-11)
11. http://www.web2pyslices.com/slice/show/1940/scheduler-tests [↑](#footnote-ref-12)