SCOOP Documentation

Release dev

Marc Parizeau, Olivier Gagnon, Marc-André Gardner, Yannick Holc

Sep 27, 2017

Contents

SCOOP (Scalable COncurrent Operations in Python) is a distributed task module allowing concurrent parallel programming on various environments, from heterogeneous grids to supercomputers.

Our philosophy is based on these ideas:

- The **future** is parallel;
- **Simple** is beautiful;
- **Parallelism** should be simpler.

These tenets are translated concretely in a minimum number of functions allowing maximum parallel efficiency while keeping at minimum the inner knowledge required to use them. It is implemented with Python 3 in mind while being compatible with 2.6+ to allow fast prototyping without sacrificing efficiency and speed.

CHAPTER 1

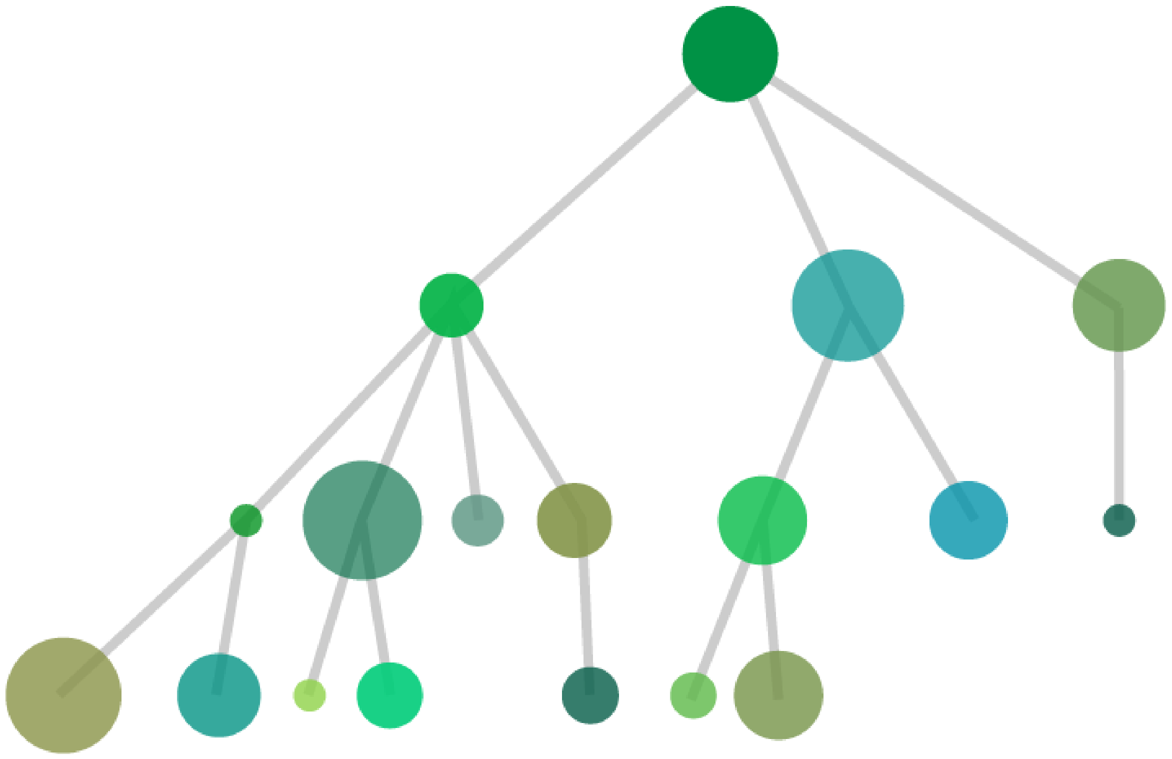
Features

SCOOP has many features and advantages over Futures, multiprocessing and similar modules, such as:

- Harness the power of **multiple computers** over network;
- Ability to spawn subtasks within tasks;
- API compatible with **PEP 3148**;
- Parallelizing serial programs with only minor modifications;
- Efficient load-balancing.

Anatomy of a SCOOPed program

SCOOP can handle multiple diversified multi-layered tasks. You can submit your different functions and data simultaneously and effortlessly while the framework executes them locally or remotely. Contrarily to most multiprocessing frameworks, it allows to launch subtasks within tasks.



Through SCOOP, you can simultaneously execute tasks that are of different nature (Discs of different colors) or different by complexity (Discs radiuses). The module will handle the physical considerations of parallelization such as task distribution over your resources (load balancing), communications, etc.

Applications

The common applications of SCOOP consist of, but is not limited to:

- Evolutionary Algorithms
- Monte Carlo simulations
- Data mining
- Data processing
- I/O processing
- Graph traversal

CHAPTER 2

Manual

Install

Dependencies

The software requirements for SCOOP are as follows:

- Python >= 2.6 or >= 3.2
- Distribute >= 0.6.2 or setuptools >= 0.7
- Greenlet >= 0.3.4
- pyzmq >= 13.1.0 and libzmq >= 3.2.0
- **ssh** for remote execution

Prerequisites

Linux

You must have the Python headers (to compile pyzmq and greenlet) and pip installed. These should be simple to install using the package manager provided with your distribution.

To get the prerequisites on an Ubuntu system, execute the following in a console:

```
sudo apt-get install python-dev python-pip
```

Ensure that your compiler is GCC as it is the tested compiler for pyzmq and greenlet.

Mac

The easiest way to get started is by using Homebrew. Once you've brewed your Python version and ZeroMQ, you are ready to install SCOOP.

Windows

Please download and install pyzmq before installing SCOOP. This can be done by using the binary installer provided at their download page. These installers will provide libzmq alongside pyzmq.

You can install pip on windows using either Christoph Gohlke windows installers or the get-pip.py script as shown in the pip-installer.org webpage.

Installation

To install SCOOP, use pip as such:

```
pip install scoop
```

POSIX Operating systems

Connection to remote hosts is done using SSH. An implementation of SSH must be installed in order to be able to use this feature.

Windows Operating System

On Windows, this will try to compile libzmq. You can skip this compilation by installing pyzmq using the installer available at their download page. This installer installs libzmq alongside pyzmq.

Furthermore, to be able to use the multi-system capabilities of SCOOP, a SSH implementation must be available. This may be done either by using Cygwin or OpenSSH for Windows.

Remote usage

Because remote host connection needs to be done without a prompt, you must use ssh keys to allow **passwordless authentication between every computing node**. You should make sure that your public ssh key is contained in the `~/.ssh/authorized_keys` file on the remote systems (Refer to the ssh manual). If you have a shared `/home/` over your systems, you can do as such:

```
[~]$ mkdir ~/.ssh; cd ~/.ssh
[.ssh]$ ssh-keygen -t dsa
[.ssh]$ cat id_dsa.pub >> authorized_keys
[.ssh]$ chmod 700 ~/.ssh ; chmod 600 ./id_dsa ; chmod 644 ./id_dsa.pub ./authorized_
↪keys
```

Note: If your remote hosts needs special configuration (non-default port, some specified username, etc.), you should do it in your ssh client configuration file (by default `~/.ssh/config`).

Note: The following parameters of `ssh` are used by SCOOP:

- -x : Deactivates X forwarding
- -n : Prevents reading from stdin (batch mode)
- -oStrictHostKeyChecking=no : Allow the connection to hosts `ssh` sees for the first time. Without it, `ssh` interactively asks to accept the identity of the peer.

HPC usage

If you use an Infiniband network, you may want to use an RDMA accelerated socket alternative instead of TCP over IB. In order to do so, you can use libsdp. This can be done by performing the following steps:

```
$ wget https://www.openfabrics.org/downloads/libsdp/libsdp-1.1.108-0.17.ga6958ef.tar.
↪gz
$ tar xfvz libsdp-1.1.108-0.17.ga6958ef.tar.gz
$ cd libsdp-1.1.108
$ ./configure --prefix=$HOME && make && make install
```

Once the compilation is done, you can use it by creating a file containing this (for bash):

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/lib/
export LD_PRELOAD=libsdp.so
```

By passing this file to the `--prolog` parameter of SCOOP, SDP sockets will be used instead of TCP over IB.

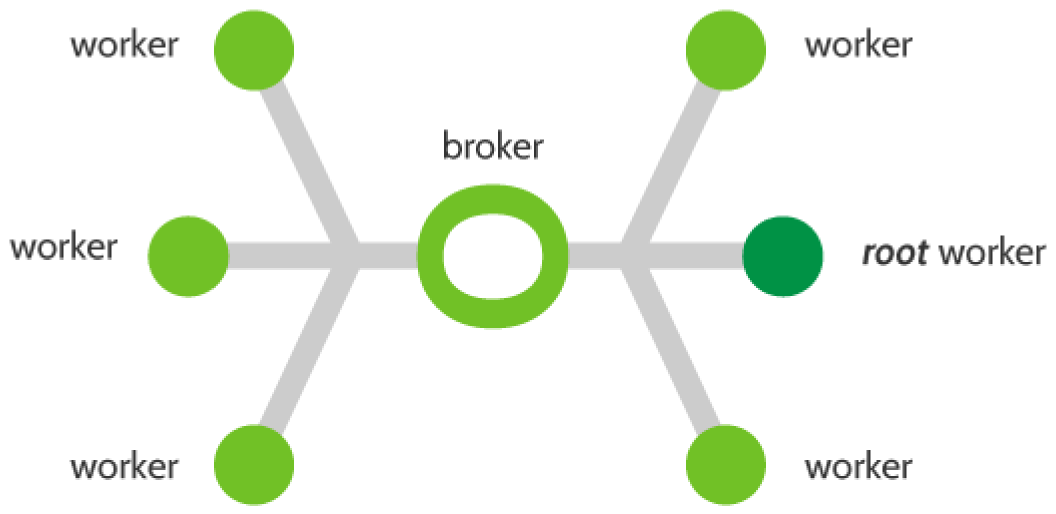
Usage

Nomenclature

Keyword	Description
Future(s)	The Future class encapsulates the asynchronous execution of a callable.
Broker	Process dispatching Futures.
Worker	Process executing Futures.
Root	The worker executing the root Future, your main program.

Architecture diagram

The future(s) distribution over workers is done by a variation of the Broker pattern. In such a pattern, workers act as independant elements that interact with a broker to mediate their communications.



Mapping API

The philosophy of SCOOP is loosely built around the *futures* module proposed by **PEP 3148**. It primarily defines a `map()` and a `submit()` function allowing asynchroneous computation that SCOOP will propagate to its workers.

Map

A *map()* function applies multiple parameters to a single function. For example, if you want to apply the *abs()* function to every number of a list:

```python
import random
data = [random.randint(-1000, 1000) for r in range(1000)]

# Without Map
result = []
for i in data:
  result.append(abs(i))

# Using a Map
result = list(map(abs, data))
```

SCOOP's `map()` returns a generator iterating over the results in the same order as its inputs. It can thus act as a parallel substitute to the standard *map()*, for instance:

```python
# Script to be launched with: python -m scoop scriptName.py
import random
from scoop import futures
data = [random.randint(-1000, 1000) for r in range(1000)]

if __name__ == '__main__':
    # Python's standard serial function
    dataSerial = list(map(abs, data))

    # SCOOP's parallel function
    dataParallel = list(futures.map(abs, data))

    assert dataSerial == dataParallel
```

Warning: In your root program, you *must* check `if __name__ == '__main__'` as shown above. Failure to do so will result in every worker trying to run their own instance of the program. This ensures that every worker waits for parallelized tasks spawned by the root worker.

Note: Your callable function passed to SCOOP must be picklable in its entirety.

Note that the pickle module is limited to **top level functions and classes** as stated in the documentation.

Note: Keep in mind that objects are not shared between workers and that changes made to an object in a function are not seen by other workers.

Map_as_completed

The *map_as_completed()* function is used exactly in the same way as the *map()* function. The only difference is that this function will yield results as soon as they are made available.

Submit

SCOOP's `submit()` returns a *Future* instance. This allows a finer control over the Futures, such as out-of-order results retrieval.

Reduction API

mapReduce

The `mapReduce()` function allows to parallelize a reduction function after applying the aforementioned *map()* function. It returns a single element.

A reduction function takes the map results and applies a function cumulatively to it. For example, applying `reduce(lambda x, y: x+y, ["a", "b", "c", "d"])` would execute `((("a")+"b")+"c")+"d")` give you the result `"abcd"`.

More information is available in the standard Python documentation on the reduce function.

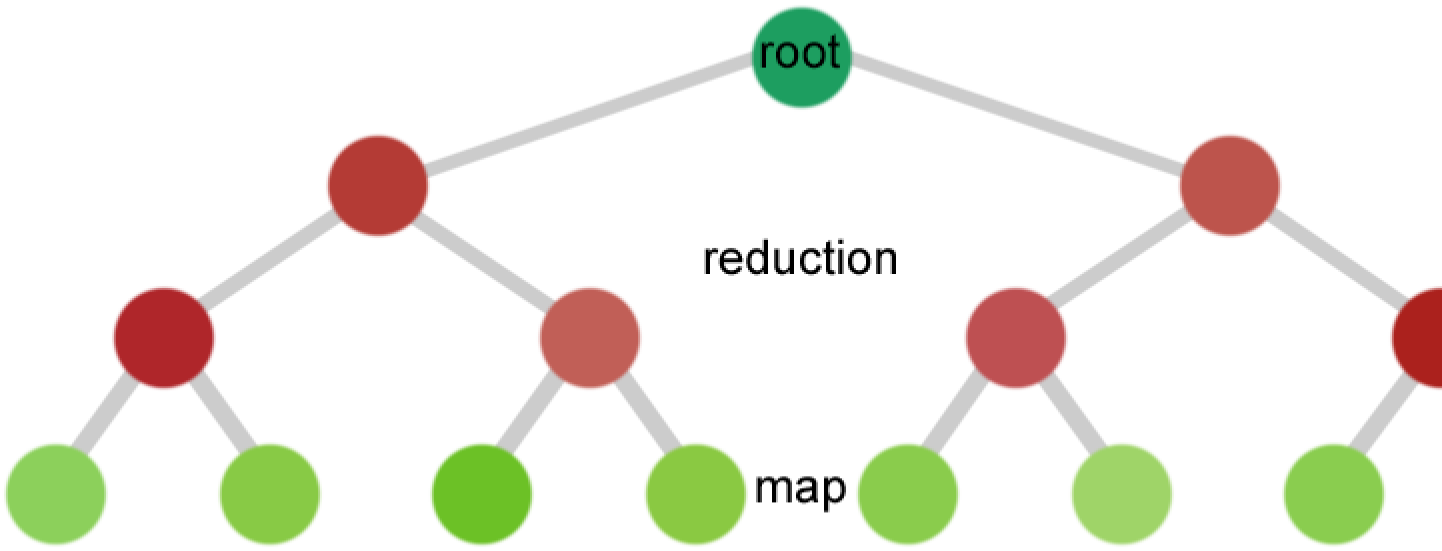
A common reduction usage consist of a sum as the following example:

```
# Script to be launched with: python -m scoop scriptName.py
import random
import operator
from scoop import futures
data = [random.randint(-1000, 1000) for r in range(1000)]


if __name__ == '__main__':
    # Python's standard serial function
    serialSum = sum(map(abs, data))

    # SCOOP's parallel function
    parallelSum = futures.mapReduce(abs, operator.add, data)

    assert serialSum == parallelSum
```

Note: You can pass any arbitrary reduction function, not only operator ones.

Architecture

SCOOP will automatically generate a binary reduction tree and submit it. Every level of the tree contain reduction nodes except for the bottom-most which contains the mapped function.



Utilities

Object sharing API

Sharing constant objects between workers is available using the `shared` module.

Its functionnalities are summarised in this example:

```
from scoop import futures, shared

def myParallelFunc(inValue):
    myValue = shared.getConst('myValue')
    return inValue + myValue


if __name__ == '__main__':
    shared.setConst(myValue=5)
    print(list(futures.map(myParallelFunc, range(10))))
```

Note: A constant can only be defined once on the entire pool of workers. More information in the *Shared module* reference.

Logging

You can use the *scoop.logger* to output useful information alongside your log messages such as the time, the worker name which emitted the message and the module in which the message was emitted.

Here is a sample usage:

```
import scoop

scoop.logger.warn("This is a warning!")
```

How to launch SCOOP programs

Programs using SCOOP, such as the ones in the `examples/` directory, need to be launched with the `-m scoop` parameter passed to Python, as such:

```
cd scoop/examples/
python -m scoop fullTree.py
```

Note: When using a Python version prior to 2.7, you must start SCOOP using *-m scoop.__main__* .

You should also consider using an up-to-date version of Python.

Launch in details

The SCOOP module spawns the needed broker(s) and worker(s) on the given list of computers, including remote ones via **ssh**.

Every worker imports your program with a *__name__* variable different than *__main__* then awaits orders given by the root node to execute available functions. This is necessary to have references over your functions and variables in the global scope.

This means that everything (definitions, assignments, operations, etc.) in the global scope of your program will be executed by every worker. To ensure a section of your code is only executed once, you must place a conditional barrier such as this one:

```
if __name__ == '__main__':
```

An example with remote workers may be as follow:

```
python -m scoop --hostfile hosts -vv -n 6 your_program.py [your arguments]
```

Argument	Meaning
-m scoop	**Mandatory** Uses SCOOP to run program.
–hostfile	hosts is a file containing a list of host to launch SCOOP
-vv	Double verbosity flag.
-n 6	Launch a total of 6 workers.
your_program.py	The program to be launched.
[your arguments]	The arguments that needs to be passed to your program.

Note: Your local hostname must be externally routable for remote hosts to be able to connect to it. If you don't have a DNS properly set up on your local network or a system hosts file, consider using the `--broker-hostname` argument to provide your externally routable IP or DNS name to SCOOP. You may as well be interested in the `-e` argument for testing purposes.

Hostfile format

You can specify the hosts with a hostfile and pass it to SCOOP using the `--hostfile` argument. The hostfile should use the following syntax:

```
hostname_or_ip 4
other_hostname
third_hostname 2
```

The name being the system hostname and the number being the number of workers to launch on this host. The number of workers to launch is optional. If omitted, SCOOP will launch as many workers as there are cores on the machine.

Using a list of host

You can also use a list of host with the `--host [...]` flag. In this case, you must put every host separated by a space the number of time you wish to have a worker on each of the node. For example:

```
python -m scoop --host machine_a machine_a machine_b machine_b your_program.py
```

This example would start two workers on `machine_a` and two workers on `machine_b`.

Choosing the number of workers

The number of workers started should be equal to the number of cores you have on each machine. If you wish to start more or less workers than specified in your hostfile or in your hostlist, you can use the `-n` parameter.

Be aware that tinkering with this parameter may hinder performances.

Note: The `-n` parameter overrides any previously specified worker amount.

If `-n` is less than the sum of workers specified in the hostfile or hostlist, the workers are launched in batch by host until the parameter is reached. This behavior may ignore latters hosts.

If `-n` is more than the sum of workers specified in the hostfile or hostlist, the remaining workers are distributed using a Round-Robin algorithm. Each host will increment its worker amount until the parameter is reached.

Use with a scheduler

You must provide a startup script on systems using a scheduler such as supercomputers or laboratory grids. Here are some example startup scripts using different grid task managers. Some example startup scripts are available in the `examples/submit_files` directory.

SCOOP natively supports Sun Grid Engine (SGE), Torque (PBS-compatible, Moab, Maui) and SLURM. That means that a minimum launch file is needed while the framework recognizes automatically the nodes assigned to your task.

Note: **These are only examples**. Refer to the documentation of your scheduler for the list of arguments needed to run the task on your grid or cluster.

Use on cloud services

Pitfalls

Program scope

As a good Python practice (see **PEP 395#what-s-in-a-name**), you should always wrap the executable part of your program using:

```
if __name__ == '__main__':
```

This is mandatory when using parallel frameworks such as multiprocessing or SCOOP. For an explanation why, read the *Launch in details* section.

If your program lacks this conditional barrier, your whole program will be executed as many times as there are workers, meaning duplicate work is being done.

Unpicklable Future

Only functions or classes declared at the top level of your program are picklables. This is a limitation of Python's pickle module. Here are some examples of non-working map invocations:

```
from scoop import futures


class myClass(object):
    @staticmethod
    def myFunction(x):
        return x


if __name__ == '__main__':
    def mySecondFunction(x):
        return x

    # Both of these calls won't work because Python pickle won't be able to
    # pickle or unpickle the function references.
    wrongCall1 = futures.map(myClass.myFunction, [1, 2, 3, 4, 5])
    wrongCall2 = futures.map(mySecondFunction, [1, 2, 3, 4, 5])
```

Launching a faulty program will result in this error being displayed:

```
[...] This element could not be pickled: [...]
```

Mutable arguments

In standard programs, modifying a mutable function argument also modifies it in the caller scope because objects are passed by reference. This side-effect is not simulated in SCOOP. Function arguments are not serialized back along its answer.

Lazy-like evaluation

The `map()` and `submit()` will distribute their Futures both locally and remotely. Futures executed locally will be computed upon access (iteration for the `map()` and `result()` for `submit()`). Futures distributed remotely will be executed right away.

Large datasets

Every parameter sent to a function by a `map()` or `submit()` gets serialized and sent within the Future to its worker. Sending large elements as parameter(s) to your function(s) results in slow speeds and network overload.

You should consider using a global variable in your module scope for passing large elements. It will then be loaded on launch by every worker and won't overload your network.

Unefficient:

```
from scoop import futures


def mySum(inData):
    """The worker will receive all its data from network."""
    return sum(inData)

if __name__ == '__main__':
    data = [[i for i in range(x, x + 1000)] for x in range(0, 8001, 1000)]
    results = list(futures.map(mySum, data))
```

Better efficiency:

```
from scoop import futures

data = [[i for i in range(x, x + 1000)] for x in range(0, 8001, 1000)]


def mySum(inIndex):
    """The worker will only receive an index from network."""
    return sum(data[inIndex])

if __name__ == '__main__':
    results = list(futures.map(mySum, range(len(data))))
```

SCOOP and greenlets

Warning: Since SCOOP uses greenlets to schedule and run futures, programs that use their own greenlets won't work with SCOOP. However, you should consider replacing the greenlets in your code by SCOOP functions.

Examples

You can find the examples detailed on this page and more in the `examples/` directory of SCOOP.

Please check the *API Reference* for any implentation detail of the proposed functions.

Introduction to the `map()` function

A core concept of task-based parallelism as presented in SCOOP is the map. An introductory example to map working is presented in `examples/map_doc.py`.

```
from __future__ import print_function
from scoop import futures

def helloWorld(value):
    return "Hello World from Future #{0}".format(value)

if __name__ == "__main__":
    returnValues = list(futures.map(helloWorld, range(16)))
    print("\n".join(returnValues))
```

Line *1* allows Python 2 users to have a print function compatible with Python 3.

On line *2*, SCOOP is imported.

On line *4-5*, the function that will be mapped is declared.

The condition on line *7* is a safety barrier that prevents the main program to be executed on every workers. It ensures that the map is issued only by one worker, the root.

The `map()` function is located on line *8*. It launches the *helloWorld* function 16 times, each time with a different argument value selected from the *range(16)* argument. This method is compatible with the standard Python *map()* function and thus can be seamlessly interchanged without modifying its arguments.

The example then prints the return values of every calls on line *9*.

You can launch this program using **`python -m scoop`**. The output should look like this:

```
~/scoop/examples$ python -m scoop -n 8 map_doc.py
Hello World from Future #0
Hello World from Future #1
Hello World from Future #2
[...]
```

Note: Results of a map are always ordered even if their computation was made asynchronously on multiple computers.

Note: You can toy around with the previous example by changing the second parameter of the `map()` function. Is it working with string arrays, pure strings or other variable types?

Computation of π

A Monte-Carlo method to calculate π using SCOOP to parallelize its computation is found in `examples/pi_calc.py`. You should familiarize yourself with Monte-Carlo methods before going forth with this example.

Fig. 2.1: Image from Wikipedia made by CaitlinJo that shows the Monte Carlo computation of π.

First, we need to import the needed functions as such:

```python
from math import hypot
from random import random
from scoop import futures
```

The Monte-Carlo method is then defined. It spawns two pseudo-random numbers that are fed to the hypot function which calculates the hypotenuse of its parameters. This step computes the Pythagorean equation ($\sqrt{x^2 + y^2}$) of the given parameters to find the distance from the origin (0,0) to the randomly placed point (which X and Y values were generated from the two pseudo-random values). Then, the result is compared to one to evaluate if this point is inside or outside the unit disk. If it is inside (have a distance from the origin lesser than one), a value of one is produced (red dots in the figure), otherwise the value is zero (blue dots in the figure). The experiment is repeated `tries` number of times with new random values.

The function returns the number times a pseudo-randomly generated point fell inside the unit disk for a given number of tries.

```python
def test(tries):
    return sum(hypot(random(), random()) < 1 for _ in range(tries))
```

One way to obtain a more precise result with a Monte-Carlo method is to perform the method multiple times. The following function executes repeatedly the previous function to gain more precision. These calls are handled by SCOOP using it's `map()` function. The results, that is the number of times a random distribution over a 1x1 square hits the unit disk over a given number of tries, are then summed and divided by the total of tries. Since we only covered the upper right quadrant of the unit disk because both parameters are positive in a cartesian map, the result must be multiplied by 4 to get the relation between area and circumference, namely π.

```python
def calcPi(nbFutures, tries):
    expr = futures.map(test, [tries] * nbFutures)
    return 4. * sum(expr) / float(nbFutures * tries)
```

As *previously stated*, you *must* wrap your code with a test for the __main__ name.

```python
if __name__ == "__main__":
    print("pi = {}".format(calcPi(3000, 5000)))
```

You can now run your code using the command **`python -m scoop`**.

Sharing Constant

A typical usage of the shared constants is to broadcast a value or an object that must be created at runtime and read by every worker, as the following:

```python

def getValue(words):
    """Computes the sum of the values of the words."""
    value = 0
    for word in words:
        for letter in word:
            # shared.getConst will evaluate to the dictionary broadcasted by
            # the root Future
            value += shared.getConst('lettersValue')[letter]
    return value


if __name__ == "__main__":
    # Set the values of the letters according to the language and broadcast it
    # This list is set at runtime
    import sys
    if len(sys.argv) > 1 and sys.argv[1] == 'francais':
        shared.setConst(lettersValue={'a': 1, 'b': 3, 'c': 3, 'd': 2, 'e': 1,
        'f': 4, 'g': 2, 'h': 4, 'i': 1, 'j': 8, 'k':10, 'l': 1, 'm': 2, 'n': 1,
        'o': 1, 'p': 3, 'q': 8, 'r': 1, 'r': 1, 's': 1, 't': 1, 'u': 1, 'v': 4,
        'w':10, 'x':10, 'y':10, 'z': 10})
        print("French letter values used.")
    else:
        shared.setConst(lettersValue={'a': 1, 'b': 3, 'c': 3, 'd': 2, 'e': 1,
        'f': 4, 'g': 2, 'h': 4, 'i': 1, 'j': 8, 'k': 5, 'l': 1, 'm': 3, 'n': 1,
        'o': 1, 'p': 3, 'q':10, 'r': 1, 'r': 1, 's': 1, 't': 1, 'u': 1, 'v': 4,
        'w': 4, 'x': 8, 'y': 4, 'z': 10})
        print("English letter values used.")

    # Get the player words (generate a list of random letters
    import random
    import string
    random.seed(3141592653)
    words = []
    player_quantity = 4
    words_per_player = 10
    word_letters = (1, 6)
    for pid in range(player_quantity):
        player = []
        for _ in range(words_per_player):
            word = "".join(random.choice(string.ascii_lowercase) for _ in␣
↪range(random.randint(*word_letters)))
            player.append(word)
        print("Player {pid} played words: {player}".format(**locals()))
        words.append(player)

    # Compute the score of every player and display it
    results = list(futures.map(getValue, words))
    for pid, result in enumerate(results):
        print("Player {pid}: {result}".format(**locals()))
```

Overall example

The `examples/fullTree.py` example holds a wrap-up of available SCOOP functionnalities. It notably shows that SCOOP is capable of handling twisted and complex hierarchical requirements.

Getting acquainted with the previous examples is fairly enough to use SCOOP, no need to dive into this complicated example.

API Reference

Futures module

The following methods are part of the futures module. They can be accessed like so:

```
from scoop import futures

results = futures.map(func, data)
futureObject = futures.submit(func, arg)
...
```

More informations are available in the *Usage* document.

`scoop.futures.`**`as_completed`**(*fs*, *timeout=None*)

Iterates over the given futures that yields each as it completes. This call is blocking.

Parameters

- **`fs`** – The sequence of Futures to wait upon.
- **`timeout`** – The maximum number of seconds to wait. If None, then there is no limit on the wait time.

Returns An iterator that yields the given Futures as they complete (finished or cancelled).

`scoop.futures.`**`map`**(*func*, **iterables*)

Equivalent to map(func, *iterables, ...) but *func* is executed asynchronously and several calls to func may be made concurrently. This non-blocking call returns an iterator which raises a TimeoutError if *__next__()* is called and the result isn't available after timeout seconds from the original call to *map()*. If timeout is not specified or

None then there is no limit to the wait time. If a call raises an exception then that exception will be raised when its value is retrieved from the iterator.

Parameters

- **`func`** – Any picklable callable object (function or class object with *__call__* method); this object will be called to execute the Futures. The callable must return a value.
- **`iterables`** – Iterable objects; each will be zipped to form an iterable of arguments tuples that will be passed to the callable object as a separate Future.
- **`timeout`** – The maximum number of seconds to wait. If None, then there is no limit on the wait time.

Returns A generator of map results, each corresponding to one map iteration.

`scoop.futures.`**`map_as_completed`**(*func*, **iterables*)

Equivalent to map, but the results are returned as soon as they are made available.

Parameters

- **`func`** – Any picklable callable object (function or class object with *__call__* method); this object will be called to execute the Futures. The callable must return a value.
- **`iterables`** – Iterable objects; each will be zipped to form an iterable of arguments tuples that will be passed to the callable object as a separate Future.
- **`timeout`** – The maximum number of seconds to wait. If None, then there is no limit on the wait time.

Returns A generator of map results, each corresponding to one map iteration.

`scoop.futures.`**`shutdown`**(*wait=True*)

This function is here for compatibility with *futures* (PEP 3148) and doesn't have any behavior.

Parameters **`wait`** – Unapplied parameter.

`scoop.futures.`**`wait`**(*fs*, *timeout=-1*, *return_when='ALL_COMPLETED'*)

Wait for the futures in the given sequence to complete. Using this function may prevent a worker from executing.

Parameters

- **`fs`** – The sequence of Futures to wait upon.
- **`timeout`** – The maximum number of seconds to wait. If negative or not specified, then there is no limit on the wait time.
- **`return_when`** – Indicates when this function should return. The options are:

FIRST_COMPLETED	Return when any future finishes or is cancelled.
FIRST_EXCEPTION	Return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to ALL_COMPLETED.
ALL_COMPLETED	Return when all futures finish or are cancelled.

Returns A named 2-tuple of sets. The first set, named 'done', contains the futures that completed (is finished or cancelled) before the wait completed. The second set, named 'not_done', contains uncompleted futures.

Future class

The `submit()` function returns a *`Future`* object. This instance possesses the following methods.

class `scoop._types.`**`Future`**(*parentId*, *callable*, **args*, ***kargs*)

This class encapsulates an independent future that can be executed in parallel. A future can spawn other parallel futures which themselves can recursively spawn other futures.

`add_done_callback`(*callable_*, *inCallbackType='standard'*, *inCallbackGroup=None*)

Attach a callable to the future that will be called when the future is cancelled or finishes running. Callable will be called with the future as its only argument.

Added callables are called in the order that they were added and are always called in a thread belonging to the process that added them. If the callable raises an Exception then it will be logged and ignored. If the callable raises another BaseException then behavior is not defined.

If the future has already completed or been cancelled then callable will be called immediately.

`cancel`()

If the call is currently being executed or sent for remote execution, then it cannot be cancelled and the method will return False, otherwise the call will be cancelled and the method will return True.

`cancelled`()

Returns True if the call was successfully cancelled, False otherwise.

`done`()

Returns True if the call was successfully cancelled or finished running, False otherwise. This function updates the executionQueue so it receives all the awaiting message.

`exception`(*timeout=None*)

Return the exception raised by the call. If the call hasn't yet completed then this method will wait up to timeout seconds. More information in the *Usage* page. If the call hasn't completed in timeout seconds then a TimeoutError will be raised. If timeout is not specified or None then there is no limit to the wait time.

If the future is cancelled before completing then CancelledError will be raised.

If the call completed without raising then None is returned.

Returns The exception raised by the call.

`result`(*timeout=None*)

Return the value returned by the call. If the call hasn't yet completed then this method will wait up to ''timeout'' seconds. More information in the *Usage* page. If the call hasn't completed in timeout seconds then a TimeoutError will be raised. If timeout is not specified or None then there is no limit to the wait time.

If the future is cancelled before completing then CancelledError will be raised.

If the call raised an exception then this method will raise the same exception.

Returns The value returned by the callable object.

`running`()

Returns True if the call is currently being executed and cannot be cancelled.

Shared module

This module provides the `setConst()` and *`getConst()`* functions allowing arbitrary object sharing between futures. These objects can only be defined once and cannot be modified once shared, hence the name constant.

class `scoop.shared.`**`SharedElementEncapsulation`**(*element*)

Encapsulates a reference to an element available in the shared module.

This is used by Futures (map on lambda, for instance).

`scoop.shared.getConst(name, timeout=0.1)`

Get a shared constant.

Parameters

- **`name`** – The name of the shared variable to retrieve.
- **`timeout`** – The maximum time to wait in seconds for the propagation of the constant.

Returns The shared object.

Usage: value = getConst('name')

SCOOP Constants and objects

The following objects are available to a program that was launched using SCOOP.

Note: Please note that using these is considered as advanced usage. You should not rely on these for other purposes than debugging.

Constants	Description
scoop.IS_ORIGIN	Boolean value. True if current instance is the root worker.
scoop.BROKER	broker.broker.BrokerInfo namedtuple. Address, ports and hostname of the broker.
scoop.DEBUG	Boolean value. True if debug mode is enabled, false otherwise.
scoop.IS_RUNNING	Boolean value. True if SCOOP is currently running, false otherwise.
scoop.worker	2-tuple. Unique identifier of the current instance in the pool.
scoop.logger	Logger object. Provides log formatting and redirection facilities. See the official documentation for more information on its usage.

Contributing

Reporting a bug

You can report a bug on the issue tracker on google code or on the mailing list.

Retrieving the latest code

You can check the latest sources with the command:

```
hg clone https://code.google.com/p/scoop/
```

Bear in mind that this development code may be partially broken or unfinished. To get a stable version of the code, update to a release tag using *hg update <tag name>*.

Coding guidelines

Most of those conventions are base on Python PEP8.

> *A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.*

Code layout

Same as PEP8.

Imports

Standard library imports must be first, followed by SCOOP imports and finally custom modules. Each section should be separated by an empty line as such:

```
import system

from scoop import futures

import myModule
```

Whitespace in Expressions and Statements

Same as PEP8.

Comments

Same as PEP8

Documentation Strings

Same as PEP8

Naming Conventions

- **Module**: lowercase convention.
- **Class**: CapWords (upper camel case) convention (ie. AnExample).
- **Function** / Procedure: mixedCase (lower camel case) convention. First word should be an action verb.
- **Variable**: lower_case_with_underscores convention. Should be as short possible as.

If a name already exists in the standard library, an underscore is appended to it. (ie. a custom *range* function could be called *range_*. A custom *type* function could be called *type_*.)

Architecture

Communication protocols

Here are the message types from the point of view of a broker. Message coming from workers are always from their Task socket.

Message name	Socket	Arguments	Description
INIT	Task		Handshake from a worker: allows a broker to recognize a new worker and propagate the currently shared variables.
CON-NECT	Task	Addresses	Notify a broker of the existence of other brokers.
REQUEST	Task		Worker requesting task(s).
TASK	Task	Task	A task (future) to be executed.
REPLY	Task*	Task, Destination	The result of a task to be sent to its parent. Communicated directly between workers if possible.
SHUT-DOWN	Info		Request a shutdown of the entire worker pool.
VARI-ABLE	Info	Key, Value, Source	A worker requested the share of a variable. The broker propagates it to its fellow workers.
TASK-END	Info	askResult, groupID	A collaborative task (scan, reduce, etc.) have ended, memory can be freed on workers.
BRO-KER_INFO	Info		Propagate information about other brokers to workers.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index