PPlus Documentation

Release 0.5.2

Salvatore Masecchia, Grzegorz Zycinski, Annalisa Barla

June 26, 2013

CONTENTS

Release 0.5.2

Homepage http://slipguru.disi.unige.it/Software/PPlus

Repository https://bitbucket.org/slipguru/pplus

PPlus is a simple environment to execute Python code in parallel on many machines without much effort. It is actually a fork of Parallel Python, another simple but powerful framework for parallel execution of python code, which lacks features needed for effective use in our daily research.

More specifically, `PPlus` was created to answer following needs:

- to facilitate data transport over distributed environment of usually very big file, exposing a simple interface while handling details in the background
- to separate file handling between different experiments, so one machine can participate in many computational experiments simultaneously

CHAPTER

ONE

USER DOCUMENTATION

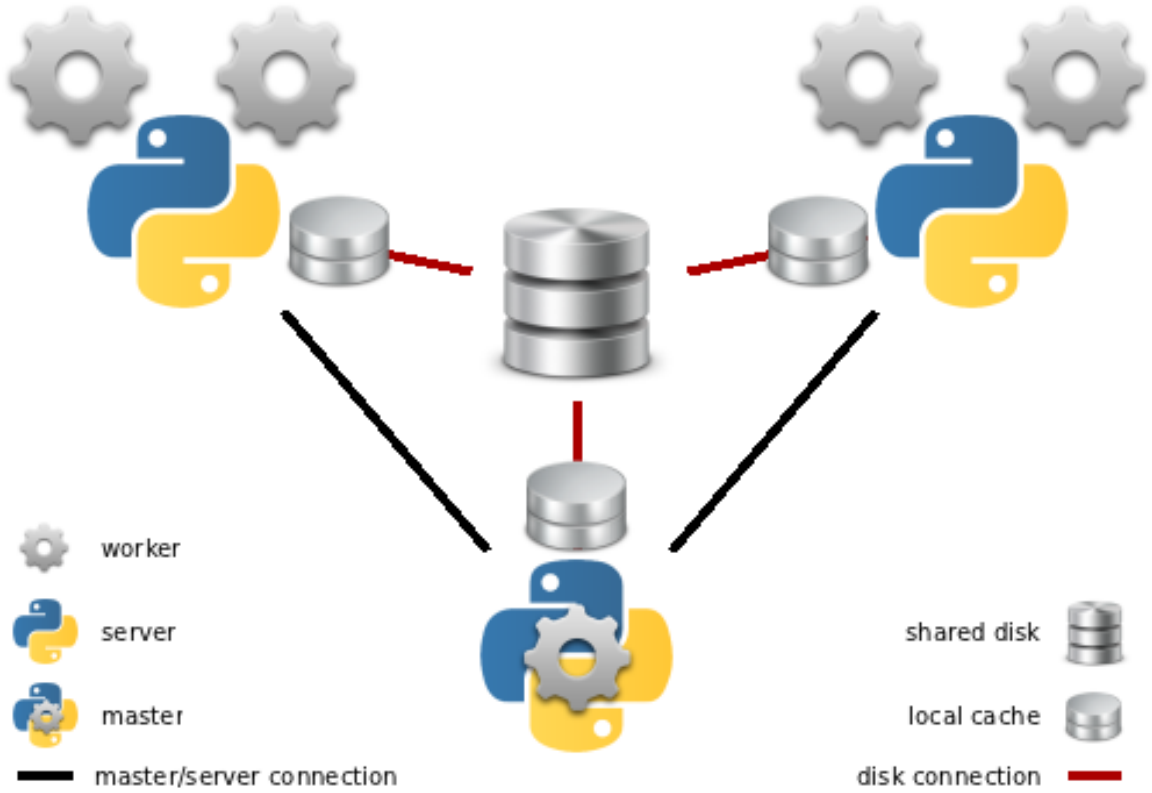
1.1 Overview

A distributed environment controlled by `PPlus` (mainly inherited from Parallel Python) is composed of a set of machines (**nodes**) that offer their resources to execute assigned tasks. All those **nodes** are running the `pplusserver.py` process in the background, that provides visibility over the local network of a prefixed number of computational **workers** and controls all data transfers (see *Using PPlus*).

The Python code to be executed by `PPlus` consists of the following conceptual pieces:

- the **worker** code is distributed over the network to the **node** machines to be executed there; it produces *partial* results saved locally and ready to be collected
- the **master** code that distributes the **worker** code pieces, collects all *partial* results and produce **master** (final) results

When a Python code needs to be executed in parallel, it is placed on one of the machines. That process is designated to be the **master** process: it distributes all parallel tasks to node **workers** and it receives all the results. Note that the machine running the master process can also provide workers (running the `pplusserver.py` script).



Both **worker** code and **master** code can do any computations, import modules (with some restrictions), and produce files.

Experiments

Internally, `PPlus` uses the concept of **experiments** to organize the code and data. The **experiment** consists of the code that performs a specific task, including pieces to be executed in parallel (i.e. master code and all worker code), as well as all regular files produced by that code. A single instance of the worker code, submitted for remote execution, is also called *worker task* or *worker job*.

To ease running of multiple subsequent experiments over the network, the experiment code can use a **shared file system resource** to store files produced during execution and to access them back if needed. This functionality is controlled by `PPlus` in a transparent way and it is exposed through a simple *API*.

Note: PPlus controls shared disk resource in the form of any network-mounted file system (e.g. NFS, Samba etc.), that can be accessed as normal directory. As a result, PPlus manipulates all remote file system resources as normal files and directories.

Warning: PPlus has been developed on Linux utilizing single NFS-mounted external disk as shared file system resource; using of different remote file systems and different operating systems has not been tested at this moment.

The *experiment code* can access and store any remote files in a dedicated **experiment directory**, created specifically for that purpose on the **shared disk** resource. Both experiment master code and all worker code have an access to the experiment directory. The files produced by different experiments are physically separated; no direct support is provided for accessing data from outside the experiment. As a result, many experiments can run simultaneously without data corruption.

Note: The *experiment directory* is created automatically when the *experiment code* is started. It is possible to assign already existing directory passing manually an experiment id, but is responsibility of the user code to manage results collected in different run to avoid unexpected data corruptions or overwriting.

Sessions

Each execution of the code, of both master and worker type, on each machine, is considered a **session**. All the activity during a session is stored in a separated session log file. More specifically:

- all worker tasks are considered running in separate sessions, and will produce separate session log files;
- the *master code* is also treated as running in separate session, but it will produce **two** logs; *session log* documents the activity regarding accessing shared disk resource from within master code; *experiment log* documents the activity regarding distribution of worker tasks; see *Logging* for more details.

1.2 Installation

`PPlus` is available on PyPI and may be easily installed using standard python setup tools like pip.

Anyway, we recommend to install `PPlus` from the source distribution, to also get some useful configuration files and testing scripts shipped with it.

After extracting the compressed package, you can use the default distutils commands:

```
$ python setup.py install
```

After installation, you can launch the test suite:

```
$ python -c "import pplus; pplus.test()"
```

Note: Tests will print reports of expected raised exceptions. Check if you get `OK` or `FAILED` status at the end.

Note: You have to install `PPlus` on **each machine** you would like to use as a node worker and /or master.

You can also check the latest sources on `PPlus` code repository:

```
$ hg clone https://sabba@bitbucket.org/slipguru/pplus
```

1.2.1 Configuration

`PPlus` uses a configuration file that specifies the paths for storing and managing remote and temporary files and logging. Each machine that participates in the network should provide its own configuration file.

During the installation, a default configuration file is saved in `/etc/pplus/pplus.cfg`. The simplest way to configure `PPlus` is to edit this file which will be reached by master and workers scripts on the machine.

On the master process, the configuration file may be specified directly in `config_file_path` parameter when instantiating `PPlusConnection`:

```
pc = PPlusConnection(config_file_path='pplus.cfg')
```

If not specified that way (and in all worker sessions), `PPlus` looks for the configuration file in the following locations, in order:

1. current working directory, as obtained by `os.getcwd()`
2. `~/.pplus/pplus.cfg`, e.g. `/home/user/.pplus/pplus.cfg`
3. global configuration file `/etc/pplus/pplus.cfg` installed with `PPlus` (recommended)

Note: On workers, current working directory depends on how `pplusserver.py` script is executed.

An example of standard configuration file can be found in the source distribution of `PPlus` under `<path-to-pplus-source>/examples/pplus.cfg` or `downloaded here`:

```
# Default configuration file for PPlus client machine

# Disk configuration
[io]
DISK_PATH = disk/              ; absolute paths o relative paths with respect
CACHE_PATH = cache/            ; to this configuration file directory

# Workers configuration
[worker]
CACHE_WAITING_TIME = 2          ; in seconds
SESSION_LOG_LEVEL = DEBUG       ; [ DEBUG | INFO | WARNING | ERROR | CRITICAL ]

# Master configuration
[master]
JOB_MAX_RESUBMISSION = 0        ; a task is resubmitted after specified failures
EXPERIMENT_LOG_LEVEL = DEBUG    ; [ DEBUG | INFO | WARNING | ERROR | CRITICAL ]
```

The `DISK_PATH` is the root of remote file system resource controlled by `PPlus` (e.g. the NFS mount point).

The `CACHE_PATH` is the root directory of internal local cache used by `PPlus`.

The CACHE_WAITING_TIME is the time between locking individual file objects managed in local cache of the machine, to avoid any possibility of concurrent writing. This parameter may be configured individually for each machine, if needed.

The SESSION_LOG_LEVEL is the level of detail for log files produced by worker code on worker machines during execution of worker tasks, as well as the *session log* produced by master code on master machine.

The JOB_MAX_RESUBMISSION is the number of maximal re-submissions of single failed worker task. That is, if the worker task has failed, it will be submitted, possibly to different machine, to be executed again. If it fails again, it will be submitted back again etc, maximum up to the number of times specified by this parameter.

The EXPERIMENT_LOG_LEVEL is the level of detail for *experiment log* produced by master code on master machine.

Note: (*OS Specific*) All mount points specified in DISK_PATH on all machines participating in the single experiment, must point to the **same** physical location on disk resource. Consult the documentation specific for your OS for more details.

Note: The CACHE_PATH must be writable by the user that will execute master/worker code on that machine.

Configuration options in debug mode

PPlus has also a debug running mode:

```
pc = PPlusConnection(debug=True)
```

When a PPlusConnection is instantiated in this way no connection between master a worker processes is created and each configuration option assume its default value (e.g. JOB_MAX_RESUBMISSION=0 and CACHE_WAITING_TIME=2). Tasks will be executed on the local machine, creating a number of subprocesses equal to the number of cpu cores minus one. Moreover, two local (w.r.t. master process working directory) directories, namely disk and cache, will be created to *simulate* shared disk and local cache spaces.

1.2.2 A simple Task: counting words in a text file

With the source distribution of PPlus it is also shipped a simple example of how PPlus could be use. This script is also useful to test the installation and the definition of a valid configuration file and can be found under <path-to-pplus-source>/examples/pplus_test.py or dowloaded here.

The script defines a job function count which relies on a smaller function count_word.

Note: A PPlus job function has to expect as first argument a PPlusConnection instance (automatically created by PPlus)

```
# Define a dependency
def count_word(line, word):
    return line.count(word)

# Define a distribute function
def count(pc, word): # PPlusConnection instance mandatory as first argument
    bigfile = pc.get_path('BIGFILE')

    counter = 0
    with open(bigfile) as f:
        for line in f:
            counter += count_word(line, word)

    return word, counter
```

Into the main function is first created a new `PPlus` connection:

```
def main():
    # PPlus Connection instantiation in debug mode
    pc = pplus.PPlusConnection(debug=True)
    print 'Starting experiment with id %s' % pc.id
    print 'Master session id %s' % pc.session_id
```

Because a `PPlusConnection` instance is created in debug mode, no configuration file is needed.

Then, if you have an active Internet connection, the script will automatically download a sample text file, namely `bigfile.txt`, which can also be downloaded manually from the repository, and put it on the shared disk space:

```
    # Download, if not exists, the input file
    if not os.path.exists('bigfile.txt'):
        file_url = 'http://bitbucket.org/slipguru/pplus/downloads/bigfile.txt'
        print "Downloading 'bigfile.txt'...",
        urllib.urlretrieve(file_url, 'bigfile.txt')
        print "done"

    # Put the file on the shared disk
    pc.put('BIGFILE', 'bigfile.txt')
```

We are now ready to submit the jobs. The target is to count how many times a prefixed set of (actually 6) words appear into the text contained in `bigfile.txt`:

```
    # Submit counting jobs
    words = ['love', 'strong', 'year', 'than', 'is', 'and'] # 6 jobs
    for w in words:
        pc.submit(count, (w,), depfuncs=(count_word,))
```

Then we can collect and print the results:

```
    # Collect (and print) the results
    results = pc.collect()
    print '\n%d tasks on %d returned' % (len(results), len(words))
    for w, c in results:
        print "Found '%s' %d times" % (w, c)
```

From the `PPlus` source dir, if you run:

```
$ examples/pplus_test.py
```

the expected results is something like:

```
Starting experiment with id b92af2161a7511e1961f4cedde1c47b7
Master session id 31faf2377b274d42908cbe72ff66aee8
Downloading 'bigfile.txt'... done

6 tasks on 6 returned
Found 'love' 2763 times
Found 'strong' 499 times
Found 'year' 1313 times
Found 'than' 2493 times
Found 'is' 81588 times
Found 'and' 87585 times
```

Because we are in debug mode, jobs have run on the local machine cpus and you should have, in the current working directory, two new directories simulating local cache (`cache`) and shared disk (`disk`):

```
$ ls cache disk
cache:
b92af2161a7511e1961f4cedde1c47b7

disk:
b92af2161a7511e1961f4cedde1c47b7
```

Each one contains a sub-directory named as the executed `experiment id`.

The **shared disk** `disk` contains the `BIGFILE` which was putted in, and an *experiment log* `experiment.log`:

```
$ ls -R disk
disk:
b92af2161a7511e1961f4cedde1c47b7

disk/b92af2161a7511e1961f4cedde1c47b7:
BIGFILE
experiment.b92af2161a7511e1961f4cedde1c47b7.log
```

The **local cache** contains the `BIGFILE` which was got by the workers, into the job function, and a `logs` directory containing logs for each `PPlus` session:

```
$ ls -R cache
cache:
b92af2161a7511e1961f4cedde1c47b7

cache/b92af2161a7511e1961f4cedde1c47b7:
BIGFILE
logs

cache/b92af2161a7511e1961f4cedde1c47b7/logs:
rubik.master.31faf2377b274d42908cbe72ff66aee8.log
rubik.worker.118c1ec5ec634476b0ec99460df3b970.log
rubik.worker.26d78679bfb74a80b0d2ee4f72950616.log
rubik.worker.804089d287104afb87fe73cda038023f.log
rubik.worker.80e51feeb16a422d85681d5339a2bcee.log
rubik.worker.d4b0ab3a565442d8a834270ef64402a4.log
rubik.worker.dd9b8b3bdd3742df883ba11692909e66.log
```

Because the experiment ran in debug mode, here we have logs for all sessions:

- one for the master: `hostname.master.<session_id>.log` and
- one for each of (6) workers: `hostname.worker.<session_id>.log`.

Now we are ready to configure our network and run the experiment on a distributed environment as described in *Using PPlus*.

1.3 Using PPlus

In *A simple Task: counting words in a text file* you can see how to use `PPlus` in a parallel but not distributed environment (*debug mode*, see also `PPlusConnection` arguments).

In this section we assume you have correctly edit a `PPlus` configuration file adding valid disk and cache paths, as explained in *Configuration*.

To run the test distributing the tasks we have only to start a `pplusserver.py` in auto-discovery mode (see *PPlus Server options*):

```
$ pplusserver.py -a
```

and then the `pplus_test.py` script shutting-off the debug modality:

```
pc = pplus.PPlusConnection(debug=False) # line 24 of pplus_test.py
```

1.3.1 PPlus Server options

Executing `pplusserver.py` with the `--help` option:

```
$ pplusserver.py --help
```

you get a complete list of options (see also PPlusConnection):

```
usage: pplusserver.py [-h] [-d, --debug] [-a, --auto] [-r, --restart]
                      [--version] [-i, --interface INTERFACE]
                      [-p, --port PORT] [-b, --broadcast BROADCAST]
                      [-w, --workers {1,2,3,4}] [-s, --secret SECRET]
                      [-t, --timeout TIMEOUT] [-n, --nproto PROTONUM]

PPlus Network Server (pplus-0.5.0-hg)

optional arguments:
  -h, --help            show this help message and exit
  -d, --debug           set log level to debug
  -a, --auto            enable auto-discovery service
  -r, --restart         restart worker process after each task completion
  --version             show program's version number and exit
  -i, --interface INTERFACE
                        interface to listen
  -p, --port PORT       port to listen
  -b, --broadcast BROADCAST
                        broadcast address for auto-discovery service
  -w, --workers {1,2,3,4}
                        number of workers to start
  -s, --secret SECRET   secret for authentication
  -t, --timeout TIMEOUT
                        timeout to exit if no connections withclients exist
  -n, --nproto PROTONUM
                        protocol number for pickle module
```

1.3.2 Example: a working environment

To facilitate computational experiments performed by Computational Biology and Biostatistics branch of SlipGURU research group, a complete environment for parallel computations was created in our lab.

It consists of a few personal Linux workstations, accessing each other in within the same local network. A separate workstation has been dedicated to provide only the external disk space, accessible via NFS.

On all participating workstations, running various flavors of Ubuntu, a dedicated login-less user `pplus` was created and a system service (`pplus_services`) was installed on each participating workstation, that mounts external disk via NFS and starts `pplusserver.py`, so the machine can re-enter the environment automatically even after manual restart.

The NFS mount point and location of local cache are in `pplus` home directory, and user `pplus` will execute any incoming worker code.

Master code of the experiment can be executed on any workstation from any user.

All the operation needed to install a compete environment are performed using the `pplus_add_node.py` script shipped with `PPlus` source distribution into the `install` directory together with the `pplus_services` Linux script. Run:

```
$ pplus_add_node.py --help
```

to see all the options:

```
Usage: pplusserver.py [options] nfs_mount.

    NFS mount point must be in remote_host:remote_dir format.
    If not found, NFS driver will be automatically installed.
```

```
Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -u USER, --user=USER  workers owner user (*default pplus*). If the specified
                        user does not exist, it will be created (login
                        disabled)
  -w WORKERS, --workers=WORKERS
                        number of workers to manage, default 4
```

1.4 PPlus Insight

In this section we describe some information about PPlus development that could be useful for advanced users.

1.4.1 File Management

Remote files are managed based on *file keys*. They serve as identifiers for accessing physical files without knowing their precise location, regardless of the network protocols.

The following rules apply to file keys regarding experiment level:

- between different experiment directories, the same file keys may be used; that is, key 'BIGFILE' used in experiment A, and key 'BIGFILE' used in experiment B, are both referring to two different files
- within experiment directory, if the same file key is used for opening new remote file for writing, by default the content of the existing file will be overwritten without warning; otherwise, an error will be reported

As a result, **within experiment all file keys must be unique** to avoid unwanted data corruption.

Note: It is strongly advisable *not* to access any physical files in the locations affected by experiment code that is running: *experiment directory* on shared disk resource, and *local cache* directories for all participating worker machines. Doing so may result in data corruption.

Note: (*OS Specific*) By convention, typical file keys are composed of capital letters, digits and underscore, for instance `CFG`, `PROBESET_2_GENEID`. However, it is possible to use, as file key, any regular file name *that is acceptable on the OS that handles shared disk resource*. Consult OS specific documentation for more details.

Note: Internally, in the current implementation, the file objects are still stored as normal files, and file keys are used as real file names. Therefore, knowing the OS specific details of shared disk resource, as well as the file keys themselves, it is still possible to access the real file objects, in case of untraceable crash.

1.4.2 Logging

PPlus uses `logging` to record its activity during the execution of experiment code.

The following logs are used:

- *experiment log*

This log is created by master code when *experiment ID* is granted. It documents the activity of the master code regarding control of worker tasks and interaction with Parallel Python. Also, all errors in worker tasks will be logged here. It is considered *private* and is *not* exposed through public *API*. When experiment is finished, it is available in the following location:

```
<SHARED_DISK_PATH>/<experiment_ID>/experiment.log
```

- *master session log*

This log is created by master code when *experiment ID* and *session ID* are granted. It primarily documents the activity of the master code regarding remote file access. It is considered *public* and is exposed through public *API*. When experiment is finished, it is available in the following location on master machine:

```
<LOCAL_CACHE_PATH>/<experiment_ID>/logs/<machine_name>.master.<session_ID>.log
```

- *session log*

This log is created by each single worker task, with *experiment ID* given and *session ID* granted. It documents the activity of the worker code regarding remote file access. It is considered *public* and is exposed through public *API*. When experiment is finished, it is available in the following location on worker machine:

```
<LOCAL_CACHE_PATH>/<experiment_ID>/logs/<machine_name>.worker.<session_ID>.log
```

Note: Logs produced in `<LOCAL_CACHE_PATH>` are never transferred to shared disk resource after the experiment has been finished. They must be accessed manually on each machine.

1.4.3 PPlus Execution modes

Debug Mode

Debug mode is intended to check the correctness of the experiment code, by executing it as *local experiment*. Instead of distributing worker tasks to remote machines, all of them will be executed on local machine, along with master task.

In this mode:

1. PPlus ignores all configuration files and creates `disk` and `cache` directories in current working directory:

```
>>> import os
>>> import pplus
>>> cwd = os.getcwd()
>>> pc = pplus.PPlusConnection(debug=True)
>>> os.path.exists(os.path.join(cwd, 'disk'))
True
>>> os.path.exists(os.path.join(cwd, 'cache'))
True
```

2. The master code is executed normally, and it 'distributes' all worker code pieces as usual, producing all regular files normally
3. When any exception is thrown during the execution of master code, the experiment code flow is interrupted, and the error is reported
4. When any exception is thrown during the execution of any worker task, the task is **not** resubmitted for another execution, the experiment code flow is interrupted, and the error is reported

Normal Mode

Normal mode is intended to run the experiment code over fully configured parallel environment.

In this mode:

1. The master code is executed; during the initial phase, the following specific activities occur:
   - the master `PPlusConnection` instance is created, that reads properly specified configuration file (see *Configuration*), obtaining, among others, `DISK_PATH` and `CACHE_PATH` locations *for that particular machine*
   - the experiment ID is granted, in the form of `uuid`

- the session ID is granted, in the form of `uuid`
- the *experiment directory* is created:

  `<DISK_PATH>/<experiment_ID>`

  all remote files produced by the whole experiment code will be stored there
- the *local cache* for the experiment is created on that machine:

  `<CACHE_PATH>/<experiment_ID>`

  all temporary copies of remote files accessed by master code will be stored there

3. The master code continues its execution, eventually worker code pieces are distributed over worker machines. The master code keeps track of all distributed worker tasks, as well as of all completed worker tasks.
4. When some worker piece of code is distributed, together with experiment ID, to worker machine, then reconstructed according to Parallel Python rules, and started, the following specific activities occur:
   - from within worker code, the worker `PPlusConnection` instance is created that reads properly specified configuration file, obtaining, among others, `DISK_PATH` and `CACHE_PATH` locations *for that particular machine*
   - the experiment ID is re-used to access shared *experiment directory* in:

     `<DISK_PATH>/<experiment_ID>`
   - the worker session ID is granted, in the form of `uuid`
   - if does not exists, the *local cache* for the experiment is created for that machine:

     `<CACHE_PATH>/<experiment_ID>`

     all temporary copies of remote files, accessed by any worker code running on that machine within the experiment, will be stored there
   - the worker code piece continues its execution until the formal end (i.e. when the last statement has been processed, and/or function end has been reached)

     Note: When any exception is thrown inside worker task, it is considered an *error* and the task is considered as *not completed*. Therefore, all worker tasks must be self-contained; deliberate exception propagation will lead to error.

     ---

     when the execution passes without errors, the worker task is considered *completed*
5. Master code, in the meanwhile, controls execution status of all distributed worker tasks periodically ('collects' them).

   When some worker task is marked as *not completed*, it is *resubmitted* for another execution, until it is marked as *completed*.

   Note: The maximum number of re-submissions is controlled by `JOB_MAX_RESUBMISSION` parameter, specified for master machine (see *Configuration*). Note that by default, the failed worker tasks are **not** resubmitted.

   Note: Although the limit of re-submissions is available, the unnecessary overhead of computation time is still present for particular long tasks (that is, when task is failing constantly because of programming error). Therefore, it is advisable to design parallel code with caution using Debug Mode, before trying it with Normal Mode.

   ---

6. When any exception is thrown during the execution of master code, the experiment code flow is interrupted, and the error is reported
7. When master code has collected all distributed worker tasks, it finishes its execution until the formal end (i.e. when the last statement has been processed, and/or function end has been reached)
8. The experiment code has finished; assuming all configuration files pointed to the same shared disk resource, all the shared data are available in one *experiment directory*:

   `<DISK_PATH>/<experiment_ID>`

1.4.4 PPlus job submission

Starting with PPlus 0.5.2, jobs can be submitted in two ways.

Direct Mode

In direct mode, one passes *function object* to `submit()`.

This way, while convenient, has some limitations regarding scope of function in Python code.

Technically, function object must be *unpicklable* by local/remote Parallel Python worker. In case of unpickling error, the exception may vary; see `pickle` for details on how *function objects* are pickled/unpickled.

This also means that function must be essentially *importable* on remote worker machine(s), as well as for local worker(s). Most frequently this means proper installation of module/package the function is defined in. See `distutils` for more information about installing Python modules/packages.

Note: If job function is in properly installed module/package, direct mode should be sufficient and less troublesome to use. For example, if job functions are from **numpy/scipy** packages, most likely those packages will be installed anyway on all involved PPlus machines.

Indirect Mode

In indirect mode, essentially ANY Python function can be submitted as job to be executed by Parallel Python, bypassing the limitations of direct mode.

Here, the function object is NOT passed directly into `submit()`. Instead, the following procedure is applied:

1. Full scope of function object (i.e. the module it is defined in) *must be* accessible through `sys.path`; if not, it may be dynamically added.

   For example, if function is defined as "function1" in package "a.b.c", then "a.b.c" must be accessible through `sys.path`.
2. Function object is added as one of the *depfuncs* (i.e. function object is added to *depfuncs* argument iterable).
3. *Function name* is passed as "function" argument to `submit()`.

Technically, the job function itself is then transported by Parallel Python as *any other depfunc*, and then is recovered in proper scope on proper machine by its name.

The indirect mode allows circumventing scope problems e.g. with submission of functions from deep submodules.

Note: The indirect mode is important in the following cases:

- PPlus is used with Python module/package that is **not installed**, either on remote worker machine(s) or on local machine, or even both
- PPlus user is **not in full control** of what could, or could not, be installed on involved machines

---

Example

Note: In this example, we assume that module/package **mysoftware** is *not* installed, i.e. it is not accessible through *sys.path*.

The following function is defined in module **mysoftware.mod1.impl.Job**. The function has no arguments and no dependencies (in sense of Parallel Python's depfuncs):

```
# module mysoftware.mod1.impl.Job
# file Job.py

# ...

# job function
def jobFunc():
    # job code here
    return result

# ...
```

To submit it in direct mode, it is necessary to call `submit()` within the scope of **mysoftware.mod1.impl.Job**, so the function object is pickled and unpickled properly:

```
# module mysoftware.mod1.impl.Job
# file Job.py

import pplus

# ...

# job function
def jobFunc():
    # job code here
    return result

# ...

if __name__ == '__main__':
    # ...
    # create PPlus connection, here in debug mode
    pc = pplus.PPlusConnection(debug=True)
    # submit in direct mode
    pc.submit(jobFunc)
    pc.collect()
```

This works since current directory is always added to `sys.path`.

If we have our submission code implemented in **mysoftware.main.job.Job**, submission in direct mode is no longer possible, since the enclosing module (mysoftware.mod1.impl.Job) is not visible and import is not sufficient:

```
# module mysoftware.main.job.Job
# file Job.py

import pplus
##### -----> fails with import error
from mysoftware.mod1.impl import jobFunc

if __name__ == '__main__':
    # ...
    # create PPlus connection, here in debug mode
    pc = pplus.PPlusConnection(debug=True)
    # submit in direct mode
```

```
    pc.submit(jobFunc)
    pc.collect()
```

One can add *mysoftware.mod1.impl* to `sys.path`, and dynamically import 'Job' (see `__import__()` for more details), but this will cause different trouble:

```
# module mysoftware.main.job.Job
# file Job.py

import pplus

if __name__ == '__main__':
    # ...
    # create PPlus connection, here in debug mode
    pc = pplus.PPlusConnection(debug=True)
    # path to 'mysoftware' package
    MYSOFTWARE_ROOT_PATH = ...
    # add it to sys.path
    if MYSOFTWARE_ROOT_PATH not in sys.path:
        sys.path.append(MYSOFTWARE_ROOT_PATH)
    # dynamically import 'mysoftware.mod1.impl.Job' and obtain it
    jmod = __import__('mysoftware.mod1.impl', fromlist=['Job'])
    jmod = getattr(jmod, 'Job')
    # obtain job function
    jobFunc = jmod.jobFunc
    # submit in direct mode
    pc.submit(jobFunc)
    ##### -----> fails during job execution, most likely with ImportError
    pc.collect()
```

This time, function object is visible and properly submitted, but local/remote worker(s) cannot *unpickle* it properly, and (most likely) result will be ImportError thrown from within PP worker.

To submit such function successfully from within **mysoftware.main.job**, we need indirect mode. Note that we still need to modify `sys.path`:

```
# module mysoftware.main.job.Job
# file Job.py

import pplus
import sys
import os
from mysoftware.mod1.impl import jobFunc

if __name__ == '__main__':
    # ...
    # create PPlus connection, here in debug mode
    pc = pplus.PPlusConnection(debug=True)
    # path to 'mysoftware' package
    MYSOFTWARE_ROOT_PATH = ...
    # add it to sys.path
    if MYSOFTWARE_ROOT_PATH not in sys.path:
        sys.path.append(MYSOFTWARE_ROOT_PATH)
    # dynamically import 'mysoftware.mod1.impl.Job' and obtain it
    jmod = __import__('mysoftware.mod1.impl', fromlist=['Job'])
    jmod = getattr(jmod, 'Job')
    # obtain job function
    jobFunc = jmod.jobFunc
    # INDIRECT MODE SUBMISSION
    depfuncs = list()
    # jobFunc is passed as Parallel Python depfunc
    depfuncs.append(jobFunc)
    # submit in indirect mode
```

```
    pc.submit(jobFunc.func_name, depfuncs=depfuncs)
    pc.collect()
```

Here, `func_name` is a string that contains function name. See *Data model* for more details.

Essentially, any Python function located anywhere can be submitted indirectly, providing that it is accessible through `sys.path` in the moment of submission. Note that after submission, any dynamic entries added to `sys.path` may be removed if not needed anymore.

Indirect mode on PPlus earlier than 0.5.2

It is still possible to use indirect mode on PPlus with version earlier than 0.5.2, since indirect mode is inherent feature of Parallel Python, and PPlus 0.5.2 offers only *support* for it. In such case, PPlus will very likely throw exception similar to this one (as seen on Windows, Python 2.6):

```
Traceback (most recent call last):
  <...>
  <...>
  File "C:\Python26\lib\site-packages\pplus\_connection.py", line 480, in submit
    self._submit_task(function, args, depfuncs, modules)
  File "C:\Python26\lib\site-packages\pplus\_connection.py", line 597, in _submit_task
    tid=task_id)
  File "C:\Python26\lib\site-packages\pplus\core\pp.py", line 441, in submit
    (tid, args[0].func_name))
AttributeError: 'str' object has no attribute 'func_name'
```

1.4.5 Name clash with dependent functions

As a rule of thumb, **avoid using the same name for any depfunc and job function at the same time**.

For example, if **mysoftware.main.jobs.func** depends on **mysoftware.impl.deps.func**, in indirect mode *both* functions must be passed as depfuncs, and *both* will be resolvable on target machine by Parallel Python only by their name *func*. At the end, since both functions are instantiated in the same scope, only one of them can be pickup by name, and second one is simply discarded and never executed.

In such cases, what exactly happens during job execution, depends on may independent factors, among them: how many arguments dependance function has, the relative order of presence of both functions in "depfuncs" argument, etc. The end result is unpredictable.

To avoid this, simply use different names whenever possible, e.g. "**mysoftware.impl.deps.func**" and "**mysoftware.main.jobs.job_func**".

CHAPTER

TWO

PPLUS API

2.1 PPlusConnection class

class pplus.**PPlusConnection**(*id=None*, *config_file_path=None*, *debug=True*, *local_workers_number=None*, *workers_servers=None*, *secret=None*, *experiment_label=None*)

Implements common end point for accessing PPlus environment.

Parameters **id** : str, optional (default: None)

experiment id. if None, create master connection instance; otherwise, create worker connection instance.

Experiment ID is currently generated as `uuid.uuid1()` (for master connection) and `uuid.uuid4()` (for worker connection). See also "experiment_label" option.

config_file_path : str, optional (default: None)

path to PPlus configuration file.

If None the configuration file will be searched in the current working directory and then in standard current and global location, '~/.pplus/pplus.cfg' and '/etc/pplus/pplus.cfg'.

In debug mode this option is ignored.

debug : bool, optional (default: True)

if True create connection instance in PPlus debug mode.

local_workers_number : int, optional (default: None)

number of local workers forked by the master process.

In debug mode, None means that *local_workers_number* is equal to the number of available cpus/cores. In normal mode, None means 0.

workers_servers : list, optional (default: None)

list of pplus servers addresses. It is possible to specify also port by using the format 'address:port'. Default port number is 60000 If None, servers are automatically discovered (if they are running in auto-discovery mode).

In debug mode this option is ignored.

secret : str, optional (default: None)

pass-phrase used to authenticate connections with workers servers.

In debug mode this option is ignored.

experiment_label : str, optional (default: None)

label assigned to experiment

If not None, the label will be prepended to experiment id, and experiment will be identified as '<label>_<id>'. This may ease the identification of particular experiment among others, since id is unique but not human readable. See "id" option notes for more details.

Raises **PPlusError** :

when specified (or default) configuration file does not exist or when id does not contain proper mandatory option.

Attributes

`id`	Experiment ID of the current connection instance.
`session_id`	Session ID of the current connection instance.
`is_debug`	Indicates if the current connection instance is in debug mode.
`disk_path`	Full path related with the current experiment on remote disk.
`cache_path`	Full path related with the current experiment on local cache.
`session_logger`	Instance of `loggers.PPlusLogger`.

Members

`put`(key, src_path[, overwrite])	Store specified file on currently configured disk resource, using specified file i
`remove`(key)	Remove remote file, associated with specified file identifier, from currently co
`get_path`(key)	Get valid physical local path of the remote file associated with specified file id
`write_remotely`(key[, binary, buffering, ...])	Returns a file descriptor open for writing, associated with specified file identif
`submit`(function[, args, depfuncs, modules])	Submit requested callable/deepfunction as a task to be executed in parallel
`collect`([clean_executed])	Wait for all submitted but not completed tasks; when all tasks are completed,

put (*key*, *src_path*, *overwrite=True*)

Store specified file on currently configured disk resource, using specified file identifier.

Parameters **key** : str

PPlus disk file identifier to use on remote side.

src_path : str

physical path to the file to be stored on remote disk resource.

overwrite : bool, optional (default: True)

if True, overwrite any existing remote file already associated with 'key' silently; otherwise, raise an error.

Raises **PPlusError** :

when remote file is overwritten and silent overwriting of existing remote files is turned off.

remove (*key*)

Remove remote file, associated with specified file identifier, from currently configured disk resource.

If requested remote file does not exist, do nothing.

Parameters **key** : string

PPlus remote file identifier of requested file

get_path (*key*)

Get valid physical local path of the remote file associated with specified file identifier.

This path is intended for opening the file **only** in read mode; use `open_remotely()` for write mode.

The remote file is accessed through local cache. The first time it has been requested, it is transferred silently to the local cache. For every subsequent request, a copy from local cache will be accessed.

Parameters **key** : str

PPlus remote file identifier of requested file.

Returns **physical_path** : str

physical path of remote file in the local cache, for opening the file in read mode **only**.

Raises **PPlusError** :

if requested remote file cannot be accessed.

write_remotely(*key*, *binary=False*, *buffering=-1*, *overwrite=True*)

Returns a file descriptor open for writing, associated with specified file identifier.

The object returned by the function could also be used within a *with* statement.

Warning: The method returns a file-like object which actually write in worker's local cache. The file is transferred on the central disk only when you **close** it. Moreover, be aware that PPlus **does not check** for concurrent writing of the file.

Parameters **key** : str

PPlus remote file identifier of requested file.

binary : bool, optional (default: False)

if True, force opening remote file in binary mode.

buffering : int, optional (default: -1, system default)

buffer size to use while opening remote file. If the buffering argument is given, 0 means unbuffered, 1 means line buffered, and larger numbers specify the buffer size.

overwrite : bool

if True, overwrite any existing remote file already associated with 'key' silently; otherwise, raise an error

Returns **fd** : file

file descriptor for the opened file

Raises **PPlusError** :

when any error was reported during opening of remote file or when remote file is overwritten and silent overwriting of existing remote files is turned off

See Also:

`PPlusTemporaryCachedFile`

submit(*function*, *args=()*, *depfuncs=()*, *modules=()*)

Submit requested callable/deepfunction as a task to be executed in parallel within current PPlus environment.

The callable **has to accept** as first parameter an instance of `PPlusConnection` which will be instantiated and passed automatically by PPlus and may return any valid Python pickeable object.

Note: For *direct* submission, "function" is a callable. However, callable can be also submitted in *indirect* mode, where callable itself is added to *depfuncs* and callable name is given as "function". During job execution, callable is then reconstructed in correct scope by its name. This mechanism allows

circumventing natural limitations of callable submission in Parallel Python (which stems from limitations of function pickling in Python itself), and allows submission of *any* function, if only accessible through `sys.path`. See *PPlus job submission* for more details.

Warning: If your job function produce a very big result may be more efficient to save the result on the disk.

1.Create manually a file in the local cache (that we can assume writable):

```
# pc = PPlusConnection instance given by PPlus
tmp_file = os.path.join(pc.cache_path, 'UNIQUE_FILE_NAME')

with open(tmp_file) as f:
    # do something with f

pc.put('KEY', tmp_file)
```

2.Use the `write_remotely()` method:

```
# pc = PPlusConnection instance given by PPlus
with pc.write_remotely('KEY')) as f:
    # do something with f
```

Parameters **function** : callable/string

callable object/callable name to be sent as task for parallel execution.

If string, "function" is interpreted as callable *name*. Subsequently, the corresponding callable MUST be added to *depfuncs*, otherwise an error will occur!

args : iterable

any positional arguments for requested callable.

depfuncs : iterable

any depended callables ('deep functions') for requested callable.

modules : iterable

list of module names that requested callable imports when executed on remote side.

collect(*clean_executed=False*)

Wait for all submitted but not completed tasks; when all tasks are completed, collect their partial results and return them.

If *clean_executed* is True new tasks can be submitted afterwards, allowing submission cycles:

```
submit()       # 1
...
submit()       # 10
collect(True)  # get 1..10
collect(True)  # get NULL
submit()       # 11
...
submit()       # 20
collect(True)  # get 11..20
collect(True)  # get NULL
```

if False, preserve content of internal cache between calls, so older partial results will be available with newer ones:

```
submit()        # 1
...
submit()        # 10
collect(False)  # get 1..10
```

```
collect(False) # get 1..10
submit()       # 11
...
submit()       # 20
collect(False) # get 1..20
collect(False) # get 1..20
```

Parameters **clean_executed** : bool, optional (default: True)

if True, clean internal cache immediately after collection succeeded.

Returns **results** : iterable

partial results from each submitted and completed task

Raises **PPlusError** :

when no task has been submitted yet

exception pplus.**PPlusError**
General exception used for reporting PPlus errors.

2.2 Utility functions and classes

2.2.1 Logging

class pplus.loggers.**PPlusLogger**(*name*, *path*, *level*)
PPlus File Logger.

This class is used to manage sessions and experiment logs. The only responsibility of this class is to properly format log messages. Through the PPlusConnection.session_logger attribute of a pplus.PPlusConnection instance, client code can add messages into the current session log.

Parameters **name** : str

log *hierarchical* name. PPlus names all the log with a root prefix 'pplus'. A session log will have name 'pplus.<session_id>'

path : str

log file path

level : str, ['DEBUG' | 'INFO' | 'WARNING' | 'ERROR' | 'CRITICAL']

sets the log level

2.2.2 Local Cache

All the functions working on the local cache rely on a file system locker (mutex) to avoid file corruption during concurrent operations.

class pplus.lockfile.**FilesystemLock**(*name*)
A file system mutex used in the local cache.

This relies on the filesystem property that creating a symlink is an atomic operation and that it will fail if the symlink already exists. Deleting the symlink will release the lock.

The implementation is based on the lockfile module (version 10.2.0) shipped with the [Twisted] library.

Parameters **name** : str

name of the file associated with this lock.

clean : bool

indicates whether this lock was released cleanly by its last owner. Only meaningful after `lock` has been called and returns True.

locked : bool

indicates whether the lock is currently held by this object.

class `pplus.ioutils.`**`PPlusTemporaryCachedFile`**(*pc*, *key*, *mode*, *buffering*, *overwrite*)

Temporary writable cached file.

An instance of this class is returned by the `pplus.PPlusConnection.write_remotely()` method.

The class wraps a standard temporary file opened in the local cache. The user code may use it as a standard file object (even in a `with` statement) which will be transferred on the shared disk when closed.

Parameters **pc** : PPlusConnection object

instance of PPlusConnection

key : str

identifier of the file to use when transferred on shared disk

mode : str, ['wb', 'w']

file mode, 'b' means binary mode. The mode is not checked, because it is correctly passed by `pplus.PPlusConnection`

buffering : int

buffer size to use while opening file.

overwrite : bool

if True, overwrite any existing remote file already associated with 'key' silently; otherwise, raise an error Check is done only during transferring.

`pplus.ioutils.`**`copy_if_not_exists`**(*src_path*, *dst_path*, *clean=True*)

Copy `src_path` to `dst_path` only if the latter does not exist.

`clean` parameters refers to the cache mutex status.

`pplus.ioutils.`**`remove_if_exists`**(*file_path*, *clean=True*)

Remove `file_path` from the local cache.

`clean` parameters refers to the cache mutex status.

`pplus.ioutils.`**`create_local_experiment_dirs`**(*path*, *clean=True*)

Create local cache directories structure having `path` as root.

`clean` parameters refers to the cache mutex status.

2.2.3 Configuration file

`pplus.ioutils.`**`read_config_file`**(*config_file_path*)

Reads configuration file and returns a dictionary representation.

For each 'section' and for each 'option' the resulting dictionary contain the read value mapped with the key 'SECTION.OPTION'.

2.3 Quick Reference

`pplus.PPlusConnection`([id, ...])	Implements common end point for accessing PPlus environment.
`pplus.PPlusError`	General exception used for reporting PPlus errors.

Continued on next page

Table 2.3 – continued from previous page

pplus.loggers	Logging utilities (for sessions and experiment logs).
pplus.ioutils	File System related utilities (caching and configuration file parsing).
pplus.lockfile	Filesystem-based interprocess mutex.

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *search*

BIBLIOGRAPHY

[Twisted] http://twistedmatrix.com

PYTHON MODULE INDEX

p

PYTHON MODULE INDEX

p

INDEX