

# Parallel Python documentation

## Module API

Quick start guide, SMP  
 Quick start guide, clusters  
 Quick start guide, clusters with auto-discovery  
 Advanced guide, clusters  
 Command line arguments, ppserver.py  
 Security and secret key  
 PP FAQ

pp 1.6.0 module API

class Server Parallel Python SMP execution server class

Methods defined here:

`__init__(self, ncpus='autodetect', ppservers=(), secret=None, restart=False, proto=2)`Creates Server instance

ncpus - the number of worker processes to start on the local computer, if parameter is omitted it will be set to the number of processors in the system

ppservers - list of active parallel python execution servers to connect with

secret - passphrase for network connections, if omitted a default passphrase will be used. It's highly recommended to use a custom passphrase for all network connections.

restart - whether to restart worker process after each task completion

proto - protocol number for pickle module

With ncpus = 1 all tasks are executed consequently

For the best performance either use the default "autodetect" value

or set ncpus to the total number of processors in the system `destroy(self)`Kills ppworkers and closes open files

`get_active_nodes(self)`Returns active nodes as a dictionary

[keys - nodes, values - ncpus] `get_ncpus(self)`Returns the number of local worker processes (ppworkers)

`get_stats(self)`Returns job execution statistics as a dictionary `print_stats(self)`Prints job execution statistics. Useful for benchmarking on

clusters `set_ncpus(self, ncpus='autodetect')`Sets the number of local worker processes (ppworkers)

ncpus - the number of worker processes, if parameter is omitted

it will be set to the number of processors in the system `submit(self, func, args=(), depfuncs=(), modules=(), callback=None,`

`callbackargs=(), group='default', globals=None)`Submits function to the execution queue

func - function to be executed

args - tuple with arguments of the 'func'

depfuncs - tuple with functions which might be called from 'func'

modules - tuple with module names to import

callback - callback function which will be called with argument

list equal to `callbackargs+(result,)`

as soon as calculation is done

callbackargs - additional arguments for callback function

group - job group, is used when `wait(group)` is called to wait for

jobs in a given group to finish

globals - dictionary from which all modules, functions and classes

will be imported, for instance: `globals=globals()` `wait(self, group=None)`Waits for all jobs in a given group to finish.

If group is omitted waits for all jobs to finish `default_port = 60000` `default_secret = 'epo20pdosl;dkslldkmm'`

class Template Template class

Methods defined here:

`__init__(self, job_server, func, depfuncs=(), modules=(), callback=None, callbackargs=(), group='default',`

`globals=None)`Creates Template instance

job\_server - pp server for submitting jobs

func - function to be executed

depfuncs - tuple with functions which might be called from 'func'

modules - tuple with module names to import  
callback - callback function which will be called with argument  
list equal to callbackargs+(result,)  
as soon as calculation is done  
callbackargs - additional arguments for callback function  
group - job group, is used when wait(group) is called to wait for  
jobs in a given group to finish  
globals - dictionary from which all modules, functions and classes  
will be imported, for instance: globals=globals() submit(self, \*args)Submits function with \*arg arguments to the execution queue

Data copyright = 'Copyright (c) 2005-2010 Vitalii Vanovschi. All rights reserved'  
version = '1.6.0' Quick start guide, SMP

1) Import pp module:

```
import pp
```

2) Start pp execution server with the number of workers set to the number of processors in the system

```
job_server = pp.Server()
```

3) Submit all the tasks for parallel execution:

```
f1 = job_server.submit(func1, args1, depfuncs1, modules1)
```

```
f2 = job_server.submit(func1, args2, depfuncs1, modules1)
```

```
f3 = job_server.submit(func2, args3, depfuncs2, modules2)
```

...etc...

4) Retrieve the results as needed:

```
r1 = f1()
```

```
r2 = f2()
```

```
r3 = f3()
```

...etc...

To find out how to achieve efficient parallelization with pp please take a look at examples [Quick start guide](#), [clusters](#)

On the nodes

1) Start parallel python execution server on all your remote computational nodes:

```
node-1> ./ppserver.py
```

```
node-2> ./ppserver.py
```

```
node-3> ./ppserver.py
```

On the client

2) Import pp module:

```
import pp
```

3) Create a list of all the nodes in your cluster (computers where you've run ppserver.py)

```
ppservers=("node-1", "node-2", "node-3")
```

4) Start pp execution server with the number of workers set to the number of processors in the system and list of ppservers to connect with :

```
job_server = pp.Server(ppservers=ppservers)
```

5) Submit all the tasks for parallel execution:

```
f1 = job_server.submit(func1, args1, depfuncs1, modules1)
```

```
f2 = job_server.submit(func1, args2, depfuncs1, modules1)
```

```
f3 = job_server.submit(func2, args3, depfuncs2, modules2)
```

...etc...

6) Retrieve the results as needed:

```
r1 = f1()
```

```
r2 = f2()
```

```
r3 = f3()
```

...etc...

To find out how to achieve efficient parallelization with pp please take a look at examples [Quick start guide](#), [clusters with autodiscovery](#)

On the nodes

1) Start parallel python execution server on all your remote computational nodes:

```
node-1> ./ppserver.py -a
```

```
node-2> ./ppserver.py -a
```

```
node-3> ./ppserver.py -a
```

On the client

2) Import pp module:

```
import pp
```

3) Set ppservers list to auto-discovery:

```
ppservers=("*",)
```

4) Start pp execution server with the number of workers set to the number of processors in the system and list of ppservers to connect with :

```
job_server = pp.Server(ppservers=ppservers)
```

5) Submit all the tasks for parallel execution:

```
f1 = job_server.submit(func1, args1, depfuncs1, modules1)
f2 = job_server.submit(func1, args2, depfuncs1, modules1)
f3 = job_server.submit(func2, args3, depfuncs2, modules2)
```

...etc...

6) Retrieve the results as needed:

```
r1 = f1()
r2 = f2()
r3 = f3()
```

...etc...

To find out how to achieve efficient parallelization with pp please take a look at examples [Advanced guide, clusters](#)

On the nodes

1) Start parallel python execution server on all your remote computational nodes (listen to a given port 35000, and local network interface only, accept only connections which know correct secret):

```
node-1> ./ppserver.py -p 35000 -i 192.168.0.101 -s "mysecret"
node-2> ./ppserver.py -p 35000 -i 192.168.0.102 -s "mysecret"
node-3> ./ppserver.py -p 35000 -i 192.168.0.103 -s "mysecret"
```

On the client

2) Import pp module:

```
import pp
```

3) Create a list of all the nodes in your cluster (computers where you've run ppserver.py)

```
ppservers=("node-1:35000", "node-2:35000", "node-3:35000")
```

4) Start pp execution server with the number of workers set to the number of processors in the system, list of ppservers to connect with and secret key to authorize the connection:

```
job_server = pp.Server(ppservers=ppservers, secret="mysecret")
```

5) Submit all the tasks for parallel execution:

```
f1 = job_server.submit(func1, args1, depfuncs1, modules1)
f2 = job_server.submit(func1, args2, depfuncs1, modules1)
f3 = job_server.submit(func2, args3, depfuncs2, modules2)
```

...etc...

6) Retrieve the results as needed:

```
r1 = f1()
```

```
r2 = f2()
```

```
r3 = f3()
```

```
...etc...
```

7) Print the execution statistics:

```
job_server.print_stats()
```

To find out how to achieve efficient parallelization with pp please take a look at examples Command line options, ppserver.py Usage: ppserver.py [-hda] [-i interface] [-b broadcast] [-p port] [-w nworkers] [-s secret] [-t seconds]

Options:

-h : this help message

-d : debug

-a : enable auto-discovery service

-i interface : interface to listen

-b broadcast : broadcast address for auto-discovery service

-p port : port to listen

-w nworkers : number of workers to start

-s secret : secret for authentication

-t seconds : timeout to exit if no connections with clients exist

Security and secret key

Due to the security concerns it is highly recommended to run ppserver.py with a non-trivial secret key (-s command line argument) which should be paired with the matching secret keyword of PP Server class constructor. Since PP 1.5.3 it is possible to set secret key by assigning pp\_secret variable in the configuration file .pythonrc.py which should be located in the user home directory (please make this file readable and writable only by user). The key set in .pythonrc.py could be overridden by command line argument (for ppserver.py) and secret keyword (for PP Server class constructor).