
pathos Documentation

Release 0.2.3.dev0

Mike McKerns

Jul 26, 2018

Contents:

1	pathos: parallel graph management and execution in heterogeneous computing	1
1.1	About the Pathos Framework	1
1.2	About Pathos	2
1.3	Major Features	2
1.4	Current Release	3
1.5	Development Version	3
1.6	Installation	3
1.7	Requirements	3
1.8	More Information	4
1.9	Citation	5
2	pathos module documentation	7
2.1	abstract_launcher module	7
2.2	connection module	10
2.3	core module	11
2.4	helpers module	13
2.5	hosts module	17
2.6	mp_map module	18
2.7	multiprocessing module	18
2.8	parallel module	21
2.9	pools module	24
2.10	portpicker module	30
2.11	pp module	31
2.12	pp_map module	32
2.13	profile module	33
2.14	python module	38
2.15	secure module	39
2.16	selector module	45
2.17	serial module	46
2.18	server module	47
2.19	threading module	48
2.20	util module	51
2.21	xmlrpc module	51
3	pathos scripts documentation	55
3.1	pathos_connect script	55
3.2	portpicker script	55

4 Indices and tables	57
Python Module Index	59

pathos: parallel graph management and execution in heterogeneous computing

1.1 About the Pathos Framework

`pathos` is a framework for heterogeneous computing. It provides a consistent high-level interface for configuring and launching parallel computations across heterogeneous resources. `pathos` provides configurable launchers for parallel and distributed computing, where each launcher contains the syntactic logic to configure and launch jobs in an execution environment. Examples of launchers that plug into `pathos` are: a queue-less MPI-based launcher (in `pyina`), a ssh-based launcher (in `pathos`), and a multi-process launcher (in `multiprocess`).

`pathos` provides a consistent interface for parallel and/or distributed versions of `map` and `apply` for each launcher, thus lowering the barrier for users to extend their code to parallel and/or distributed resources. The guiding design principle behind `pathos` is that `map` and `apply` should be drop-in replacements in otherwise serial code, and thus switching to one or more of the `pathos` launchers is all that is needed to enable code to leverage the selected parallel or distributed computing resource. This not only greatly reduces the time to convert a code to parallel, but it also enables a single code-base to be maintained instead of requiring parallel, serial, and distributed versions of a code. `pathos` maps can be nested, thus hierarchical heterogeneous computing is possible by merely selecting the desired hierarchy of `map` and `pipe` (`apply`) objects.

The `pathos` framework is composed of several interoperating packages:

- `dill`: a utility to serialize all of python
- `pox`: utilities for filesystem exploration and automated builds
- `klepto`: persistent caching to memory, disk, or database
- `multiprocess`: better multiprocessing and multithreading in python
- `ppft`: distributed and parallel python
- `pyina`: MPI parallel `map` and cluster scheduling
- `pathos`: graph management and execution in heterogeneous computing

1.2 About Pathos

The `pathos` package provides a few basic tools to make parallel and distributed computing more accessible to the end user. The goal of `pathos` is to enable the user to extend their own code to parallel and distributed computing with minimal refactoring.

`pathos` provides methods for configuring, launching, monitoring, and controlling a service on a remote host. One of the most basic features of `pathos` is the ability to configure and launch a RPC-based service on a remote host. `pathos` seeds the remote host with the `portpicker` script, which allows the remote host to inform the localhost of a port that is available for communication.

Beyond the ability to establish a RPC service, and then post requests, is the ability to launch code in parallel. Unlike parallel computing performed at the node level (typically with MPI), `pathos` enables the user to launch jobs in parallel across heterogeneous distributed resources. `pathos` provides distributed `map` and `pipe` algorithms, where a mix of local processors and distributed workers can be selected. `pathos` also provides a very basic automated load balancing service, as well as the ability for the user to directly select the resources.

The high-level `pool.map` interface, yields a `map` implementation that hides the RPC internals from the user. With `pool.map`, the user can launch their code in parallel, and as a distributed service, using standard python and without writing a line of server or parallel batch code.

RPC servers and communication in general is known to be insecure. However, instead of attempting to make the RPC communication itself secure, `pathos` provides the ability to automatically wrap any distributed service or communication in a `ssh-tunnel`. `Ssh` is a universally trusted method. Using `ssh-tunnels`, `pathos` has launched several distributed calculations on national lab clusters, and to date has performed test calculations that utilize node-to-node communication between several national lab clusters and a user's laptop. `pathos` allows the user to configure and launch at a very atomistic level, through raw access to `ssh` and `scp`.

`pathos` is the core of a python framework for heterogeneous computing. `pathos` is in active development, so any user feedback, bug reports, comments, or suggestions are highly appreciated. A list of known issues is maintained at <http://trac.mystic.cacr.caltech.edu/project/pathos/query.html>, with a public ticket list at <https://github.com/uqfoundation/pathos/issues>.

1.3 Major Features

`pathos` provides a configurable distributed parallel `map` interface to launching RPC service calls, with:

- a `map` interface that meets and extends the python `map` standard
- the ability to submit service requests to a selection of servers
- the ability to tunnel server communications with `ssh`

The `pathos` core is built on low-level communication to remote hosts using `ssh`. The interface to `ssh`, `scp`, and `ssh-tunneled` connections can:

- configure and launch remote processes with `ssh`
- configure and copy file objects with `scp`
- establish and tear-down a `ssh-tunnel`

To get up and running quickly, `pathos` also provides infrastructure to:

- easily establish a `ssh-tunneled` connection to a RPC server

1.4 Current Release

This documentation is for version pathos-0.2.3.dev0.

The latest released version of pathos is available from:

<https://pypi.org/project/pathos>

pathos is distributed under a 3-clause BSD license.

```
>>> import pathos
>>> print (pathos.license())
```

1.5 Development Version

You can get the latest development version with all the shiny new features at:

<https://github.com/uqfoundation>

If you have a new contribution, please submit a pull request.

1.6 Installation

pathos is packaged to install from source, so you must download the tarball, unzip, and run the installer:

```
[download]
$ tar -xvzf pathos-0.2.2.1.tar.gz
$ cd pathos-0.2.2.1
$ python setup.py build
$ python setup.py install
```

You will be warned of any missing dependencies and/or settings after you run the “build” step above. pathos depends on dill and pox, each of which are essentially subpackages of pathos but are released independently. pathos also depends on multiprocessing and ppft. You must install all of the pathos framework packages for pathos to provide the full functionality for heterogeneous computing.

Alternately, pathos can be installed with pip or easy_install:

```
$ pip install pathos
```

1.7 Requirements

pathos requires:

- python, **version** `>= 2.6` or **version** `>= 3.1`, or pypy
- dill, **version** `>= 0.2.8.2`
- pox, **version** `>= 0.2.4`
- ppft, **version** `>= 1.6.4.8`
- multiprocessing, **version** `>= 0.70.6.1`

Optional requirements:

- `setuptools`, version ≥ 0.6
- `pyina`, version $\geq 0.2.0$
- `rpyc`, version $\geq 3.0.6$

1.8 More Information

Probably the best way to get started is to look at the documentation at <http://pathos.rtfid.io>. Also see `pathos.tests` and `pathos.examples` for a set of scripts that demonstrate the configuration and launching of communications with `ssh` and `scp`, and demonstrate the configuration and execution of jobs in a hierarchical parallel workflow. You can run the test suite with `python -m pathos.tests`. Tunnels and other connections to remote servers can be established with the `pathos_connect` script (or with `python -m pathos`). See `pathos_connect --help` for more information. `pathos` also provides a `portpicker` script to select an open port (also available with `python -m pathos.portpicker`). The source code is generally well documented, so further questions may be resolved by inspecting the code itself. Please feel free to submit a ticket on github, or ask a question on stackoverflow (@MikeMcKerns). If you would like to share how you use `pathos` in your work, please send an email (to [mmckerns at uqfoundation dot org](mailto:mmckerns@uqfoundation.org)).

Important classes and functions are found here:

- `pathos.abstract_launcher` [the worker pool API definition]
- `pathos.pools` [all of the pathos worker pools]
- `pathos.core` [the high-level command interface]
- `pathos.hosts` [the hostname registry interface]
- `pathos.serial.SerialPool` [the serial python worker pool]
- `pathos.parallel.ParallelPool` [the parallepython worker pool]
- `pathos.multiprocessing.ProcessPool` [the multiprocessing worker pool]
- `pathos.threading.ThreadPool` [the multithreading worker pool]
- `pathos.connection.Pipe` [the launcher base class]
- `pathos.secure.Pipe` [the secure launcher base class]
- `pathos.secure.Copier` [the secure copier base class]
- `pathos.secure.Tunnel` [the secure tunnel base class]
- `pathos.selector.Selector` [the selector base class]
- `pathos.server.Server` [the server base class]
- `pathos.profile` [profiling in threads and processes]

`pathos` also provides two convenience scripts that are used to establish secure distributed connections. These scripts are installed to a directory on the user's `$PATH`, and thus can be run from anywhere:

- `portpicker` [get the portnumber of an open port]
- `pathos_connect` [establish tunnel and/or RPC server]

Typing `--help` as an argument to any of the above scripts will print out an instructive help message.

1.9 Citation

If you use `pathos` to do research that leads to publication, we ask that you acknowledge use of `pathos` by citing the following in your publication:

```
M.M. McKerns, L. Strand, T. Sullivan, A. Fang, M.A.G. Aivazis,  
"Building a framework for predictive science", Proceedings of  
the 10th Python in Science Conference, 2011;  
http://arxiv.org/pdf/1202.1056  
  
Michael McKerns and Michael Aivazis,  
"pathos: a framework for heterogeneous computing", 2010- ;  
http://trac.mystic.cacr.caltech.edu/project/pathos
```

Please see <http://trac.mystic.cacr.caltech.edu/project/pathos> or <http://arxiv.org/pdf/1202.1056> for further information.

citation()
print citation

license()
print license

logger (*level=None, handler=None, **kwds*)
generate a logger instance for `pathos`

Parameters

- **level** (*int, default=None*) – the logging level.
- **handler** (*object, default=None*) – a logging handler instance.
- **name** (*str, default='pathos'*) – name of the logger instance.

Returns configured logger instance.

2.1 abstract_launcher module

This module contains the base classes for pathos pool and pipe objects, and describes the map and pipe interfaces. A pipe is defined as a connection between two ‘nodes’, where a node is something that does work. A pipe may be a one-way or two-way connection. A map is defined as a one-to-many connection between nodes. In both map and pipe connections, results from the connected nodes can be returned to the calling node. There are several variants of pipe and map, such as whether the connection is blocking, or ordered, or asynchronous. For pipes, derived methods must overwrite the ‘pipe’ method, while maps must overwrite the ‘map’ method. Pipes and maps are available from worker pool objects, where the work is done by any of the workers in the pool. For more specific point-to-point connections (such as a pipe between two specific compute nodes), use the pipe object directly.

2.1.1 Usage

A typical call to a pathos map will roughly follow this example:

```
>>> # instantiate and configure the worker pool
>>> from pathos.pools import ProcessPool
>>> pool = ProcessPool(nodes=4)
>>>
>>> # do a blocking map on the chosen function
>>> results = pool.map(pow, [1,2,3,4], [5,6,7,8])
>>>
>>> # do a non-blocking map, then extract the results from the iterator
>>> results = pool.imap(pow, [1,2,3,4], [5,6,7,8])
>>> print("...")
>>> results = list(results)
>>>
>>> # do an asynchronous map, then get the results
>>> results = pool.amap(pow, [1,2,3,4], [5,6,7,8])
>>> while not results.ready():
...     time.sleep(5); print(".", end=' ')
```

(continues on next page)

```
...
>>> results = results.get()
```

Notes

Each of the pathos worker pools rely on a different transport protocol (e.g. threads, multiprocessing, etc), where the use of each pool comes with a few caveats. See the usage documentation and examples for each worker pool for more information.

class AbstractPipeConnection (*args, **kws)

Bases: `object`

AbstractPipeConnection base class for pathos pipes.

Required input: ???

Additional inputs: ???

Important class members: ???

Other class members: ???

`__dict__ = dict_proxy({'__module__': 'pathos.abstract_launcher', '__repr__': <function`

`__init__ (*args, **kws)`

Required input: ???

Additional inputs: ???

Important class members: ???

Other class members: ???

`__module__ = 'pathos.abstract_launcher'`

`__repr__ () <==> repr(x)`

`__weakref__`

list of weak references to the object (if defined)

class AbstractWorkerPool (*args, **kws)

Bases: `object`

AbstractWorkerPool base class for pathos pools.

Important class members: nodes - number (and potentially description) of workers ncpus - number of worker processors servers - list of worker servers scheduler - the associated scheduler workdir - associated \$WORKDIR for scratch calculations/files

Other class members: scatter - True, if uses 'scatter-gather' (instead of 'worker-pool') source - False, if minimal use of TemporaryFiles is desired timeout - number of seconds to wait for return value from scheduler

`__AbstractWorkerPool__get_nodes ()`

get the number of nodes in the pool

`__AbstractWorkerPool__imap (f, *args, **kws)`

default filter for imap inputs

`__AbstractWorkerPool__init (*args, **kws)`

default filter for __init__ inputs

```

__AbstractWorkerPool__map (f, *args, **kws)
    default filter for map inputs

__AbstractWorkerPool__nodes = 1

__AbstractWorkerPool__pipe (f, *args, **kws)
    default filter for pipe inputs

__AbstractWorkerPool__set_nodes (nodes)
    set the number of nodes in the pool

__dict__ = dict_proxy({'map': <function map>, '__module__': 'pathos.abstract_launcher'})

__enter__ ()

__exit__ (*args)

__init__ (*args, **kws)

```

Important class members: nodes - number (and potentially description) of workers ncpus - number of worker processors servers - list of worker servers scheduler - the associated scheduler workdir - associated \$WORKDIR for scratch calculations/files

Other class members: scatter - True, if uses 'scatter-gather' (instead of 'worker-pool') source - False, if minimal use of TemporaryFiles is desired timeout - number of seconds to wait for return value from scheduler

```

__module__ = 'pathos.abstract_launcher'

__repr__ () <==> repr(x)

__weakref__
    list of weak references to the object (if defined)

__serve (*args, **kws)
    Create a new server if one isn't already initialized

```

amap (f, *args, **kws)
run a batch of jobs with an asynchronous map

Returns a results object which contains the results of applying the function f to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence. To retrieve the results, call the get() method on the returned results object. The call to get() is blocking, until all results are retrieved. Use the ready() method on the result object to check if all results are ready.

apipe (f, *args, **kws)
submit a job asynchronously to a queue

Returns a results object which contains the result of calling the function f on a selected worker. To retrieve the results, call the get() method on the returned results object. The call to get() is blocking, until the result is available. Use the ready() method on the results object to check if the result is ready.

clear ()
Remove server with matching state

imap (f, *args, **kws)
run a batch of jobs with a non-blocking and ordered map

Returns a list iterator of results of applying the function f to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence.

map (*f*, **args*, ***kws*)

run a batch of jobs with a blocking and ordered map

Returns a list of results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence.

pipe (*f*, **args*, ***kws*)

submit a job and block until results are available

Returns result of calling the function *f* on a selected worker. This function will block until results are available.

uimap (*f*, **args*, ***kws*)

run a batch of jobs with a non-blocking and unordered map

Returns a list iterator of results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence. The order of the resulting sequence is not guaranteed.

2.2 connection module

This module contains the base class for popen pipes, and describes the popen pipe interface. The ‘config’ method can be overwritten for pipe customization. The pipe’s ‘launch’ method can be overwritten with a derived pipe’s new execution algorithm. See the following for an example of standard use.

2.2.1 Usage

A typical call to a popen ‘pipe’ will roughly follow this example:

```
>>> # instantiate the pipe
>>> pipe = Pipe()
>>>
>>> # configure the pipe to stage the command
>>> pipe(command='hostname')
>>>
>>> # execute the launch and retrieve the response
>>> pipe.launch()
>>> print(pipe.response())
```

class Pipe (*name=None*, ***kws*)

Bases: `object`

a popen-based pipe for parallel and distributed computing

create a popen-pipe

Inputs: *name*: a unique identifier (string) for the pipe command: a command to send [default = ‘echo <name>’]
background: run in background [default = False] *decode*: ensure response is ‘ascii’ [default = True] *stdin*: file-like object to serve as standard input for the remote process

— **call** — (***kws*)

configure the pipe using given keywords

(Re)configure the pipe for the following inputs: *command*: a command to send [default = ‘echo <name>’] *background*: run in background [default = False] *decode*: ensure response is ‘ascii’ [default = True] *stdin*: file-like object to serve as standard input for the remote process

```

__dict__ = dict_proxy({'__module__': 'pathos.connection', 'verbose': True, '_debug':
__init__ (name=None, **kws)
    create a popen-pipe

    Inputs: name: a unique identifier (string) for the pipe command: a command to send [default = 'echo
    <name>'] background: run in background [default = False] decode: ensure response is 'ascii' [default
    = True] stdin: file-like object to serve as standard input for the remote process

__module__ = 'pathos.connection'
__repr__ () <==> repr(x)
__weakref__
    list of weak references to the object (if defined)
_debug = <logging.Logger object>
_execute ()
config (**kws)
    configure the pipe using given keywords

    (Re)configure the pipe for the following inputs: command: a command to send [default = 'echo
    <name>'] background: run in background [default = False] decode: ensure response is 'ascii' [de-
    fault = True] stdin: file-like object to serve as standard input for the remote process

kill ()
    terminate the pipe

launch ()
    launch a configured command

pid ()
    get pipe pid

response ()
    Return the response from the launched process. Return None if no response was received yet from a
    background process.

verbose = True

exception PipeException
    Bases: exceptions.Exception
    Exception for failure to launch a command

__module__ = 'pathos.connection'
__weakref__
    list of weak references to the object (if defined)

```

2.3 core module

high-level programming interface to core pathos utilities

copy (*source*, *destination=None*, ***kws*)
 copy source to (possibly) remote destination

Execute a copy, and return the copier. Use 'kill' to kill the copier, and 'pid' to get the process id for the copier.

Inputs: source – path string of source 'file' destination – path string for destination target

execute (*command, host=None, bg=True, **kwds*)

execute a command (possibly) on a remote host

Execute a process, and return the launcher. Use 'response' to retrieve the response from the executed command. Use 'kill' to kill the launcher, and 'pid' to get the process id for the launcher.

Inputs: *command* – command string to be executed *host* – hostname of execution target [default = None (i.e. run locally)] *bg* – run as background process? [default = True]

kill (*pid, host=None, **kwds*)

kill a process (possibly) on a remote host

Inputs: *pid* – process id *host* – hostname where process is running [default = None (i.e. locally)]

getpid (*target=None, host=None, all=False, **kwds*)

get the process id for a target process (possibly) running on remote host

This method should only be used as a last-ditch effort to find a process id. This method `__may__` work when a child has been spawned and the pid was not registered... but there's no guarantee.

If *target* is None, then get the process id of the `__main__` python instance.

Inputs: *target* – string name of target process *host* – hostname where process is running *all* – get all resulting lines from query? [default = False]

getppid (*pid=None, host=None, group=False*)

get parent process id (ppid) for the given process

If *pid* is None, the pid of the `__main__` python instance will be used.

Inputs: *pid* – process id *host* – hostname where process is running *group* – get parent group id (pgid) instead of direct parent id?

getchild (*pid=None, host=None, group=False*)

get all child process ids for the given parent process id (ppid)

If *pid* is None, the pid of the `__main__` python instance will be used.

Inputs: *pid* – parent process id *host* – hostname where process is running *group* – get process ids for the parent group id (pgid) instead?

serve (*server, host=None, port=None, profile='.bash_profile'*)

begin serving RPC requests

Inputs: *server*: name of RPC server (i.e. 'ppserver') *host*: hostname on which a server should be launched *port*: port number (on host) that server will accept request at *profile*: file to configure the user's environment [default='.bash_profile']

connect (*host, port=None, through=None*)

establish a secure tunnel connection to a remote host at the given port

Inputs: *host* – hostname to which a tunnel should be established *port* – port number (on host) to connect the tunnel to *through* – 'tunnel-through' hostname [default = None]

randomport (*host=None*)

select a open port on a (possibly) remote host

Inputs: *host* – hostname on which to select a open port

2.4 helpers module

ProcessPool

alias of `multiprocess.pool.Pool`

ThreadPool (*processes=None, initializer=None, initargs=()*)

cpu_count ()

Returns the number of CPUs in the system

freeze_support ()

Check whether this is a fake forked process in a frozen executable. If so then run code specified by commandline and exit.

2.4.1 pathos.helpers module documentation

mp_helper module

map helper functions

random_seed (*s=None*)

sets the seed for calls to 'random()'

random_state (*module='random', new=False, seed='!'*)

return a (optionally manually seeded) random generator

For a given module, return an object that has random number generation (RNG) methods available. If `new=False`, use the global copy of the RNG object. If `seed='!'`, do not reseed the RNG (using `seed=None` 'removes' any seeding). If `seed='*'`, use a seed that depends on the process id (PID); this is useful for building RNGs that are different across multiple threads or processes.

starargs (*f*)

decorator to convert a many-arg function to a single-arg function

pp_helper module

class ApplyResult (*task*)

Bases: `pp._pp._Task`

result object for an 'apply' method in `parallepython`

enables a `pp._Task` to mimic the `multiprocessing.pool.ApplyResult` interface

__ApplyResult__unpickle ()

Unpickles the result of the task

__call__ (*raw_result=False*)

Retrieves result of the task

__init__ (*task*)

Initializes the task

__module__ = `'pathos.helpers.pp_helper'`

callback

callbackargs

finalize (*sresult*)

Finalizes the task **internal use only**

finished

get (*timeout=None*)
Retrieves result of the task

group

lock

ready ()
Checks if the result is ready

server

successful ()
Measures whether result is ready and loaded w/o printing

tid

wait (*timeout=None*)
Waits for the task

class MapResult (*size, callback=None, callbackargs=(), group='default'*)

Bases: `object`

__MapResult__unpickle ()
Unpickles the results of the tasks

__call__ ()
Retrieve the results of the tasks

__dict__ = `dict_proxy({'__module__': 'pathos.helpers.pp_helper', '__MapResult__unpickl`

__init__ (*size, callback=None, callbackargs=(), group='default'*)
x.`__init__`(...) initializes x; see `help(type(x))` for signature

__module__ = `'pathos.helpers.pp_helper'`

__weakref__
list of weak references to the object (if defined)

__set (*i, task*)

finalize (**results*)
finalize the tasks ***internal use only***

get (*timeout=None*)
Retrieves results of the tasks

queue (**tasks*)
Fill the MapResult with ApplyResult objects

ready ()
Checks if the result is ready

successful ()
Measures whether result is ready and loaded w/o printing

wait (*timeout=None*)
Wait for the tasks

class Server (*ncpus='autodetect', ppservers=(), secret=None, restart=False, proto=2, socket_timeout=3600*)

Bases: `object`

Parallel Python SMP execution server class

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` - restart the worker process after each task completion `proto` - protocol number for pickle module `socket_timeout` - socket timeout in seconds, which is the maximum time a remote job could be executed. Increase this value if you have long running jobs or decrease if connectivity to remote ppservers is often lost.

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.

`__Server__add_to_active_tasks` (*num*)

Updates the number of active tasks

`__Server__connect` ()

Connects to all remote ppservers

`__Server__detect_ncpus` ()

Detects the number of effective CPUs in the system

`__Server__dumpsfunc` (*funcs, modules*)

Serializes functions and modules

`__Server__find_modules` (*prefix, dict*)

recursively finds all the modules in dict

`__Server__gentid` ()

Generates a unique job ID number

`__Server__get_source` (*func*)

Fetches source of the function

`__Server__scheduler` ()

Schedules jobs for execution

`__Server__stat_add_job` (*node*)

Increments job count on the node

`__Server__stat_add_time` (*node, time_add*)

Updates total runtime on the node

`__Server__update_active_rworkers` (*id, count*)

Updates list of active rworkers

`__del__` ()

`__dict__` = `dict_proxy`({'get_stats': <function get_stats>, '__module__': 'pp.pp', '_

`__init__` (*ncpus='autodetect', ppservers=(), secret=None, restart=False, proto=2, socket_timeout=3600*)

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` - restart the worker process after each task completion `proto` - protocol number for pickle module `socket_timeout` - socket timeout in seconds, which is the maximum time a remote job could be executed. Increase this value if you have long running jobs or decrease if connectivity to remote ppservers is often lost.

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.

`__module__ = 'pp._pp'`

`__weakref__`

list of weak references to the object (if defined)

`_run_local (job, sfunc, sargs, worker)`

Runs a job locally

`_run_remote (job, sfunc, sargs, rworker)`

Runs a job remotelly

`connect1 (host, port, persistent=True)`

Conects to a remote ppserver specified by host and port

`default_port = 60000`

`default_secret = 'epo20pdos1;dksldkmm'`

`destroy ()`

Kills ppworkers and closes open files

`get_active_nodes ()`

Returns active nodes as a dictionary [keys - nodes, values - ncpus]

`get_ncpus ()`

Returns the number of local worker processes (ppworkers)

`get_stats ()`

Returns job execution statistics as a dictionary

`insert (sfunc, sargs, task=None)`

Inserts function into the execution queue. It's intended for internal use only (in ppserver).

`print_stats ()`

Prints job execution statistics. Useful for benchmarking on clusters

`set_ncpus (ncpus='autodetect')`

Sets the number of local worker processes (ppworkers)

ncpus - the number of worker processes, if parammeter is omitted it will be set to the number of processors in the system

`submit (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` - dict from which all modules, functions, and classes will be imported, for instance: `globals=globals()`

`wait (group=None)`

Waits for all jobs in a given group to finish. If group is omitted waits for all jobs to finish

exception TimeoutError

Bases: `multiprocess.ProcessError`

`__module__ = 'multiprocess'`

`_ApplyResult`alias of `multiprocess.pool.ApplyResult`**`_MapResult`**alias of `multiprocess.pool.MapResult`**`class _Task`** (*server, tid, callback=None, callbackargs=(), group='default'*)Bases: `object`

Class describing single task (job)

Initializes the task

`_Task__unpickle` ()

Unpickles the result of the task

`__call__` (*raw_result=False*)

Retrieves result of the task

`__dict__` = `dict_proxy`({'**`__dict__`**': <attribute '**`__dict__`**' of '**`_Task`**' objects>, '**`__modu`****`__init__`** (*server, tid, callback=None, callbackargs=(), group='default'*)

Initializes the task

`__module__` = `'pp._pp'`**`__weakref__`**

list of weak references to the object (if defined)

`finalize` (*sresult*)

Finalizes the task.

For internal use only

`wait` ()

Waits for the task

2.5 hosts module

high-level programming interface to pathos host registry

`_profiles` = {}*For example, to register two 'known' host profiles –***`_profiles`** = { `'foobar.danse.us': '.profile'`, `'computer.cacr.caltech.edu': '.cshrc'`,
}**`get_profile`** (*rhost, assume=True*)

get the default \$PROFILE for a remote host

`get_profiles` ()

get \$PROFILE for each registered host

`register` (*rhost, profile=None*)

register a host and \$PROFILE

`register_profiles` (*profiles*)

add dict of { 'host': \$PROFILE } to registered host profiles

2.6 mp_map module

minimal interface to python's multiprocessing module

Notes

This module has been deprecated in favor of `pathos.pools`.

mp_map (*function, sequence, *args, **kwargs*)
 extend python's parallel map function to multiprocessing

Inputs: function – target function sequence – sequence to process in parallel

Additional Inputs: nproc – number of 'local' cpus to use [default = 'autodetect'] type – processing type ['blocking', 'non-blocking', 'unordered'] threads – if True, use threading instead of multiprocessing

2.7 multiprocessing module

This module contains map and pipe interfaces to python's multiprocessing module.

Pipe methods provided: pipe - blocking communication pipe [returns: value] apipe - asynchronous communication pipe [returns: object]

Map methods provided: map - blocking and ordered worker pool [returns: list] imap - non-blocking and ordered worker pool [returns: iterator] uimap - non-blocking and unordered worker pool [returns: iterator] amap - asynchronous worker pool [returns: object]

2.7.1 Usage

A typical call to a pathos multiprocessing map will roughly follow this example:

```
>>> # instantiate and configure the worker pool
>>> from pathos.multiprocessing import ProcessPool
>>> pool = ProcessPool(nodes=4)
>>>
>>> # do a blocking map on the chosen function
>>> print(pool.map(pow, [1,2,3,4], [5,6,7,8]))
>>>
>>> # do a non-blocking map, then extract the results from the iterator
>>> results = pool.imap(pow, [1,2,3,4], [5,6,7,8])
>>> print("...")
>>> print(list(results))
>>>
>>> # do an asynchronous map, then get the results
>>> results = pool.amap(pow, [1,2,3,4], [5,6,7,8])
>>> while not results.ready():
...     time.sleep(5); print(".", end=' ')
...
>>> print(results.get())
>>>
>>> # do one item at a time, using a pipe
>>> print(pool.pipe(pow, 1, 5))
>>> print(pool.pipe(pow, 2, 6))
>>>
```

(continues on next page)

(continued from previous page)

```

>>> # do one item at a time, using an asynchronous pipe
>>> result1 = pool.apipe(pow, 1, 5)
>>> result2 = pool.apipe(pow, 2, 6)
>>> print(result1.get())
>>> print(result2.get())

```

Notes

This worker pool leverages the python's multiprocessing module, and thus has many of the limitations associated with that module. The function `f` and the sequences in `args` must be serializable. The maps in this worker pool have full functionality whether run from a script or in the python interpreter, and work reliably for both imported and interactively-defined functions. Unlike python's multiprocessing module, `pathos.multiprocessing` maps can directly utilize functions that require multiple arguments.

class ProcessPool (**args, **kws*)

Bases: `pathos.abstract_launcher.AbstractWorkerPool`

Mapper that leverages python's multiprocessing.

Important class members: `nodes` - number (and potentially description) of workers `ncpus` - number of worker processors `servers` - list of worker servers `scheduler` - the associated scheduler `workdir` - associated \$WORKDIR for scratch calculations/files

Other class members: `scatter` - True, if uses 'scatter-gather' (instead of 'worker-pool') `source` - False, if minimal use of TemporaryFiles is desired `timeout` - number of seconds to wait for return value from scheduler

NOTE: if number of nodes is not given, will autodetect processors

`__ProcessPool__get_nodes` ()

get the number of nodes used in the map

`__ProcessPool__set_nodes` (*nodes*)

set the number of nodes used in the map

`__init__` (**args, **kws*)

Important class members: `nodes` - number (and potentially description) of workers `ncpus` - number of worker processors `servers` - list of worker servers `scheduler` - the associated scheduler `workdir` - associated \$WORKDIR for scratch calculations/files

Other class members: `scatter` - True, if uses 'scatter-gather' (instead of 'worker-pool') `source` - False, if minimal use of TemporaryFiles is desired `timeout` - number of seconds to wait for return value from scheduler

NOTE: if number of nodes is not given, will autodetect processors

`__module__` = 'pathos.multiprocessing'

`__repr__` () <==> `repr(x)`

`__clear` ()

Remove server with matching state

`__serve` (*nodes=None*)

Create a new server if one isn't already initialized

`amap` (*f, *args, **kws*)

run a batch of jobs with an asynchronous map

Returns a results object which contains the results of applying the function `f` to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence. To retrieve the results, call the `get()` method on the returned results object. The call to `get()` is blocking, until all results are retrieved. Use the `ready()` method on the result object to check if all results are ready.

apipe (*f*, *args, **kws)

submit a job asynchronously to a queue

Returns a results object which contains the result of calling the function `f` on a selected worker. To retrieve the results, call the `get()` method on the returned results object. The call to `get()` is blocking, until the result is available. Use the `ready()` method on the results object to check if the result is ready.

clear ()

Remove server with matching state

close ()

close the pool to any new jobs

imap (*f*, *args, **kws)

run a batch of jobs with a non-blocking and ordered map

Returns a list iterator of results of applying the function `f` to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence.

join ()

cleanup the closed worker processes

map (*f*, *args, **kws)

run a batch of jobs with a blocking and ordered map

Returns a list of results of applying the function `f` to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence.

ncpus

get the number of nodes used in the map

nodes

get the number of nodes used in the map

pipe (*f*, *args, **kws)

submit a job and block until results are available

Returns result of calling the function `f` on a selected worker. This function will block until results are available.

restart (*force=False*)

restart a closed pool

terminate ()

a more abrupt close

uimap (*f*, *args, **kws)

run a batch of jobs with a non-blocking and unordered map

Returns a list iterator of results of applying the function `f` to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence. The order of the resulting sequence is not guaranteed.

ProcessPool

alias of `multiprocess.pool.Pool`

2.8 parallel module

This module contains map and pipe interfaces to the `parallepython` (`pp`) module.

Pipe methods provided: `pipe` - blocking communication pipe [returns: value] `apipe` - asynchronous communication pipe [returns: object]

Map methods provided: `map` - blocking and ordered worker pool [returns: list] `imap` - non-blocking and ordered worker pool [returns: iterator] `uimap` - non-blocking and unordered worker pool [returns: iterator] `amap` - asynchronous worker pool [returns: object]

2.8.1 Usage

A typical call to a pathos `pp` map will roughly follow this example:

```
>>> # instantiate and configure the worker pool
>>> from pathos.pp import ParallelPool
>>> pool = ParallelPool(nodes=4)
>>>
>>> # do a blocking map on the chosen function
>>> print(pool.map(pow, [1,2,3,4], [5,6,7,8]))
>>>
>>> # do a non-blocking map, then extract the results from the iterator
>>> results = pool.imap(pow, [1,2,3,4], [5,6,7,8])
>>> print("...")
>>> print(list(results))
>>>
>>> # do an asynchronous map, then get the results
>>> results = pool.amap(pow, [1,2,3,4], [5,6,7,8])
>>> while not results.ready():
...     time.sleep(5); print(".", end=' ')
...
>>> print(results.get())
>>>
>>> # do one item at a time, using a pipe
>>> print(pool.pipe(pow, 1, 5))
>>> print(pool.pipe(pow, 2, 6))
>>>
>>> # do one item at a time, using an asynchronous pipe
>>> result1 = pool.apipe(pow, 1, 5)
>>> result2 = pool.apipe(pow, 2, 6)
>>> print(result1.get())
>>> print(result2.get())
```

Notes

This worker pool leverages the `parallepython` (`pp`) module, and thus has many of the limitations associated with that module. The function `f` and the sequences in `args` must be serializable. The maps in this worker pool have full functionality when run from a script, but may be somewhat limited when used in the python interpreter. Both imported and interactively-defined functions in the interpreter session may fail due to the pool failing to find the source code for the target function. For a work-around, try:

```
>>> # instantiate and configure the worker pool
>>> from pathos.pp import ParallelPool
```

(continues on next page)

(continued from previous page)

```

>>> pool = ParallelPool(nodes=4)
>>>
>>> # wrap the function, so it can be used interactively by the pool
>>> def wrapsin(*args, **kwds):
>>>     from math import sin
>>>     return sin(*args, **kwds)
>>>
>>> # do a blocking map using the wrapped function
>>> results = pool.map(wrapsin, [1,2,3,4,5])

```

class ParallelPool (*args, **kwds)

Bases: *pathos.abstract_launcher.AbstractWorkerPool*

Mapper that leverages parallepython (i.e. pp) maps.

Important class members: nodes - number (and potentially description) of workers ncpus - number of worker processors servers - list of worker servers scheduler - the associated scheduler workdir - associated \$WORKDIR for scratch calculations/files

Other class members: scatter - True, if uses 'scatter-gather' (instead of 'worker-pool') source - False, if minimal use of TemporaryFiles is desired timeout - number of seconds to wait for return value from scheduler

NOTE: if number of nodes is not given, will autodetect processors NOTE: if a tuple of servers is not provided, defaults to localhost only

__ParallelPool__get_nodes ()
get the number of nodes used in the map

__ParallelPool__get_servers ()
get the servers used in the map

__ParallelPool__set_nodes (nodes)
set the number of nodes used in the map

__ParallelPool__set_servers (servers)
set the servers used in the map

__init__ (*args, **kwds)

Important class members: nodes - number (and potentially description) of workers ncpus - number of worker processors servers - list of worker servers scheduler - the associated scheduler workdir - associated \$WORKDIR for scratch calculations/files

Other class members: scatter - True, if uses 'scatter-gather' (instead of 'worker-pool') source - False, if minimal use of TemporaryFiles is desired timeout - number of seconds to wait for return value from scheduler

NOTE: if number of nodes is not given, will autodetect processors NOTE: if a tuple of servers is not provided, defaults to localhost only

__module__ = 'pathos.parallel'

__repr__ () <==> repr(x)

__clear ()
Remove server with matching state

__equals (server)
check if the server is compatible

__is_alive (server=None, negate=False, run=True)

_serve (*nodes=None, servers=None*)

Create a new server if one isn't already initialized

amap (*f, *args, **kws*)

run a batch of jobs with an asynchronous map

Returns a results object which contains the results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence. To retrieve the results, call the `get()` method on the returned results object. The call to `get()` is blocking, until all results are retrieved. Use the `ready()` method on the result object to check if all results are ready.

apipe (*f, *args, **kws*)

submit a job asynchronously to a queue

Returns a results object which contains the result of calling the function *f* on a selected worker. To retrieve the results, call the `get()` method on the returned results object. The call to `get()` is blocking, until the result is available. Use the `ready()` method on the results object to check if the result is ready.

clear ()

Remove server with matching state

close ()

close the pool to any new jobs

imap (*f, *args, **kws*)

run a batch of jobs with a non-blocking and ordered map

Returns a list iterator of results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence.

join ()

cleanup the closed worker processes

map (*f, *args, **kws*)

run a batch of jobs with a blocking and ordered map

Returns a list of results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence.

ncpus

get the number of nodes used in the map

nodes

get the number of nodes used in the map

pipe (*f, *args, **kws*)

submit a job and block until results are available

Returns result of calling the function *f* on a selected worker. This function will block until results are available.

restart (*force=False*)

restart a closed pool

servers

get the servers used in the map

terminate ()

a more abrupt close

uimap (*f*, **args*, ***kws*)

run a batch of jobs with a non-blocking and unordered map

Returns a list iterator of results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence. The order of the resulting sequence is not guaranteed.

stats (*pool=None*)

return a string containing stats response from the pp.Server

2.9 pools module

pools: pools of pathos workers, providing map and pipe constructs

class ParallelPool (**args*, ***kws*)

Bases: *pathos.abstract_launcher.AbstractWorkerPool*

Mapper that leverages parallelpython (i.e. pp) maps.

Important class members: nodes - number (and potentially description) of workers ncpus - number of worker processors servers - list of worker servers scheduler - the associated scheduler workdir - associated \$WORKDIR for scratch calculations/files

Other class members: scatter - True, if uses 'scatter-gather' (instead of 'worker-pool') source - False, if minimal use of TemporaryFiles is desired timeout - number of seconds to wait for return value from scheduler

NOTE: if number of nodes is not given, will autodetect processors NOTE: if a tuple of servers is not provided, defaults to localhost only

__ParallelPool__get_nodes ()

get the number of nodes used in the map

__ParallelPool__get_servers ()

get the servers used in the map

__ParallelPool__set_nodes (*nodes*)

set the number of nodes used in the map

__ParallelPool__set_servers (*servers*)

set the servers used in the map

__init__ (**args*, ***kws*)

Important class members: nodes - number (and potentially description) of workers ncpus - number of worker processors servers - list of worker servers scheduler - the associated scheduler workdir - associated \$WORKDIR for scratch calculations/files

Other class members: scatter - True, if uses 'scatter-gather' (instead of 'worker-pool') source - False, if minimal use of TemporaryFiles is desired timeout - number of seconds to wait for return value from scheduler

NOTE: if number of nodes is not given, will autodetect processors NOTE: if a tuple of servers is not provided, defaults to localhost only

__module__ = 'pathos.parallel'

__repr__ () <==> repr(*x*)

__clear ()

Remove server with matching state

_equals (*server*)

check if the server is compatible

_is_alive (*server=None, negate=False, run=True*)

_serve (*nodes=None, servers=None*)

Create a new server if one isn't already initialized

amap (*f, *args, **kws*)

run a batch of jobs with an asynchronous map

Returns a results object which contains the results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence. To retrieve the results, call the `get()` method on the returned results object. The call to `get()` is blocking, until all results are retrieved. Use the `ready()` method on the result object to check if all results are ready.

apipe (*f, *args, **kws*)

submit a job asynchronously to a queue

Returns a results object which contains the result of calling the function *f* on a selected worker. To retrieve the results, call the `get()` method on the returned results object. The call to `get()` is blocking, until the result is available. Use the `ready()` method on the results object to check if the result is ready.

clear ()

Remove server with matching state

close ()

close the pool to any new jobs

imap (*f, *args, **kws*)

run a batch of jobs with a non-blocking and ordered map

Returns a list iterator of results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence.

join ()

cleanup the closed worker processes

map (*f, *args, **kws*)

run a batch of jobs with a blocking and ordered map

Returns a list of results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence.

ncpus

get the number of nodes used in the map

nodes

get the number of nodes used in the map

pipe (*f, *args, **kws*)

submit a job and block until results are available

Returns result of calling the function *f* on a selected worker. This function will block until results are available.

restart (*force=False*)

restart a closed pool

servers

get the servers used in the map

terminate ()

a more abrupt close

uimap (*f*, **args*, ***kws*)

run a batch of jobs with a non-blocking and unordered map

Returns a list iterator of results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence. The order of the resulting sequence is not guaranteed.

class ProcessPool (**args*, ***kws*)

Bases: *pathos.abstract_launcher.AbstractWorkerPool*

Mapper that leverages python's multiprocessing.

Important class members: nodes - number (and potentially description) of workers ncpus - number of worker processors servers - list of worker servers scheduler - the associated scheduler workdir - associated \$WORKDIR for scratch calculations/files

Other class members: scatter - True, if uses 'scatter-gather' (instead of 'worker-pool') source - False, if minimal use of TemporaryFiles is desired timeout - number of seconds to wait for return value from scheduler

NOTE: if number of nodes is not given, will autodetect processors

__ProcessPool__get_nodes ()

get the number of nodes used in the map

__ProcessPool__set_nodes (*nodes*)

set the number of nodes used in the map

__init__ (**args*, ***kws*)

Important class members: nodes - number (and potentially description) of workers ncpus - number of worker processors servers - list of worker servers scheduler - the associated scheduler workdir - associated \$WORKDIR for scratch calculations/files

Other class members: scatter - True, if uses 'scatter-gather' (instead of 'worker-pool') source - False, if minimal use of TemporaryFiles is desired timeout - number of seconds to wait for return value from scheduler

NOTE: if number of nodes is not given, will autodetect processors

__module__ = 'pathos.multiprocessing'**__repr__** () <==> *repr(x)***__clear** ()

Remove server with matching state

__serve (*nodes=None*)

Create a new server if one isn't already initialized

amap (*f*, **args*, ***kws*)

run a batch of jobs with an asynchronous map

Returns a results object which contains the results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence. To retrieve the results, call the *get()* method on the returned results object. The call to *get()* is blocking, until all results are retrieved. Use the *ready()* method on the result object to check if all results are ready.

apipe (*f, *args, **kws*)

submit a job asynchronously to a queue

Returns a results object which contains the result of calling the function *f* on a selected worker. To retrieve the results, call the `get()` method on the returned results object. The call to `get()` is blocking, until the result is available. Use the `ready()` method on the results object to check if the result is ready.

clear ()

Remove server with matching state

close ()

close the pool to any new jobs

imap (*f, *args, **kws*)

run a batch of jobs with a non-blocking and ordered map

Returns a list iterator of results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence.

join ()

cleanup the closed worker processes

map (*f, *args, **kws*)

run a batch of jobs with a blocking and ordered map

Returns a list of results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence.

ncpus

get the number of nodes used in the map

nodes

get the number of nodes used in the map

pipe (*f, *args, **kws*)

submit a job and block until results are available

Returns result of calling the function *f* on a selected worker. This function will block until results are available.

restart (*force=False*)

restart a closed pool

terminate ()

a more abrupt close

uimap (*f, *args, **kws*)

run a batch of jobs with a non-blocking and unordered map

Returns a list iterator of results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence. The order of the resulting sequence is not guaranteed.

class SerialPool (**args, **kws*)

Bases: `pathos.abstract_launcher.AbstractWorkerPool`

Mapper that leverages standard (i.e. serial) python maps.

Important class members: `nodes` - number (and potentially description) of workers `ncpus` - number of worker processors `servers` - list of worker servers `scheduler` - the associated scheduler `workdir` - associated \$WORKDIR for scratch calculations/files

Other class members: scatter - True, if uses 'scatter-gather' (instead of 'worker-pool') source - False, if minimal use of TemporaryFiles is desired timeout - number of seconds to wait for return value from scheduler

`__SerialPool__get_nodes()`
get the number of nodes in the pool

`__SerialPool__set_nodes(nodes)`
set the number of nodes in the pool

`__module__ = 'pathos.serial'`

`__exiting = False`

`__is_alive(negate=False, run=True)`

`clear()`
hard restart

`close()`
close the pool to any new jobs

`imap(f, *args, **kws)`
run a batch of jobs with a non-blocking and ordered map

Returns a list iterator of results of applying the function f to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence.

`join()`
cleanup the closed worker processes

`map(f, *args, **kws)`
run a batch of jobs with a blocking and ordered map

Returns a list of results of applying the function f to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence.

`nodes`
get the number of nodes in the pool

`pipe(f, *args, **kws)`
submit a job and block until results are available

Returns result of calling the function f on a selected worker. This function will block until results are available.

`restart(force=False)`
restart a closed pool

`terminate()`
a more abrupt close

`class ThreadPool(*args, **kws)`
Bases: `pathos.abstract_launcher.AbstractWorkerPool`

Mapper that leverages python's threading.

Important class members: nodes - number (and potentially description) of workers ncpus - number of worker processors servers - list of worker servers scheduler - the associated scheduler workdir - associated \$WORKDIR for scratch calculations/files

Other class members: scatter - True, if uses 'scatter-gather' (instead of 'worker-pool') source - False, if minimal use of TemporaryFiles is desired timeout - number of seconds to wait for return value from scheduler

NOTE: if number of nodes is not given, will autodetect processors

`ThreadPool.get_nodes()`
get the number of nodes used in the map

`ThreadPool.set_nodes(nodes)`
set the number of nodes used in the map

`__init__(*args, **kws)`

Important class members: nodes - number (and potentially description) of workers ncpus - number of worker processors servers - list of worker servers scheduler - the associated scheduler workdir - associated \$WORKDIR for scratch calculations/files

Other class members: scatter - True, if uses 'scatter-gather' (instead of 'worker-pool') source - False, if minimal use of TemporaryFiles is desired timeout - number of seconds to wait for return value from scheduler

NOTE: if number of nodes is not given, will autodetect processors

`__module__ = 'pathos.threading'`

`__repr__()` $\langle == \rangle repr(x)$

`clear()`
Remove server with matching state

`_serve(nodes=None)`
Create a new server if one isn't already initialized

`amap(f, *args, **kws)`
run a batch of jobs with an asynchronous map

Returns a results object which contains the results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence. To retrieve the results, call the `get()` method on the returned results object. The call to `get()` is blocking, until all results are retrieved. Use the `ready()` method on the result object to check if all results are ready.

`apipe(f, *args, **kws)`
submit a job asynchronously to a queue

Returns a results object which contains the result of calling the function *f* on a selected worker. To retrieve the results, call the `get()` method on the returned results object. The call to `get()` is blocking, until the result is available. Use the `ready()` method on the results object to check if the result is ready.

`clear()`
Remove server with matching state

`close()`
close the pool to any new jobs

`imap(f, *args, **kws)`
run a batch of jobs with a non-blocking and ordered map

Returns a list iterator of results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence.

`join()`
cleanup the closed worker processes

`map(f, *args, **kws)`
run a batch of jobs with a blocking and ordered map

Returns a list of results of applying the function `f` to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence.

nodes

get the number of nodes used in the map

nthreads

get the number of nodes used in the map

pipe (*f, *args, **kws*)

submit a job and block until results are available

Returns result of calling the function `f` on a selected worker. This function will block until results are available.

restart (*force=False*)

restart a closed pool

terminate ()

a more abrupt close

uimap (*f, *args, **kws*)

run a batch of jobs with a non-blocking and unordered map

Returns a list iterator of results of applying the function `f` to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence. The order of the resulting sequence is not guaranteed.

ProcessPool

alias of `multiprocess.pool.Pool`

ThreadPool (*processes=None, initializer=None, initargs=()*)

2.10 portpicker module

This script prints out an available port number.

class portnumber (*min=0, max=65536*)

Bases: `object`

port selector

Usage:

```
>>> pick = portnumber(min=1024,max=65535)
>>> print( pick() )
```

select a port number from a given range.

The first call will return a random number from the available range, and each subsequent call will return the next number in the range.

Inputs: `min` – minimum port number [default = 0] `max` – maximum port number [default = 65536]

__call__ (...) <==> `x(...)`

__dict__ = `dict_proxy({'__module__': 'pathos.portpicker', '__dict__': <attribute '...'>})`

__init__ (*min=0, max=65536*)

select a port number from a given range.

The first call will return a random number from the available range, and each subsequent call will return the next number in the range.

Inputs: min – minimum port number [default = 0] max – maximum port number [default = 65536]

`__module__ = 'pathos.portpicker'`

`__weakref__`

list of weak references to the object (if defined)

randomport (*min=1024, max=65536*)
select a random port number

Inputs: min – minimum port number [default = 1024] max – maximum port number [default = 65536]

2.11 pp module

This module contains map and pipe interfaces to the `parallepython` (`pp`) module.

Pipe methods provided: `pipe` - blocking communication pipe [returns: value] `apipe` - asynchronous communication pipe [returns: object]

Map methods provided: `map` - blocking and ordered worker pool [returns: list] `imap` - non-blocking and ordered worker pool [returns: iterator] `uimap` - non-blocking and unordered worker pool [returns: iterator] `amap` - asynchronous worker pool [returns: object]

2.11.1 Usage

A typical call to a `pathos pp` map will roughly follow this example:

```
>>> # instantiate and configure the worker pool
>>> from pathos.pp import ParallelPool
>>> pool = ParallelPool(nodes=4)
>>>
>>> # do a blocking map on the chosen function
>>> print(pool.map(pow, [1,2,3,4], [5,6,7,8]))
>>>
>>> # do a non-blocking map, then extract the results from the iterator
>>> results = pool.imap(pow, [1,2,3,4], [5,6,7,8])
>>> print("...")
>>> print(list(results))
>>>
>>> # do an asynchronous map, then get the results
>>> results = pool.amap(pow, [1,2,3,4], [5,6,7,8])
>>> while not results.ready():
...     time.sleep(5); print(".", end=' ')
...
>>> print(results.get())
>>>
>>> # do one item at a time, using a pipe
>>> print(pool.pipe(pow, 1, 5))
>>> print(pool.pipe(pow, 2, 6))
>>>
>>> # do one item at a time, using an asynchronous pipe
>>> result1 = pool.apipe(pow, 1, 5)
>>> result2 = pool.apipe(pow, 2, 6)
```

(continues on next page)

```
>>> print(result1.get())
>>> print(result2.get())
```

Notes

This worker pool leverages the `paralelpython` (`pp`) module, and thus has many of the limitations associated with that module. The function `f` and the sequences in `args` must be serializable. The maps in this worker pool have full functionality when run from a script, but may be somewhat limited when used in the python interpreter. Both imported and interactively-defined functions in the interpreter session may fail due to the pool failing to find the source code for the target function. For a work-around, try:

```
>>> # instantiate and configure the worker pool
>>> from pathos.pp import ParallelPool
>>> pool = ParallelPool(nodes=4)
>>>
>>> # wrap the function, so it can be used interactively by the pool
>>> def wrapsin(*args, **kwds):
>>>     from math import sin
>>>     return sin(*args, **kwds)
>>>
>>> # do a blocking map using the wrapped function
>>> results = pool.map(wrapsin, [1,2,3,4,5])
```

ParallelPythonPool

alias of `pathos.parallel.ParallelPool`

stats (*pool=None*)

return a string containing stats response from the `pp.Server`

2.12 pp_map module

minimal interface to python's `pp` (parallel python) module

Implements a work-alike of the builtin `map` function that distributes work across many processes. As it uses `ppft` to do the actual parallel processing, code using this must conform to the usual `ppft` restrictions (arguments must be serializable, etc).

Notes

This module has been deprecated in favor of `pathos.pools`.

ppServer

alias of `pp._pp.Server`

pp_map (*function, sequence, *args, **kwds*)

extend python's parallel map function to parallel python

Inputs: `function` – target function `sequence` – sequence to process in parallel

Additional Inputs: `ncpus` – number of 'local' processors to use [default = 'autodetect'] `servers` – available distributed parallel python servers [default = ()]

ppmap (*processes, function, sequence, *sequences*)

Split the work of 'function' across the given number of processes. Set 'processes' to None to let Parallel Python autodetect the number of children to use.

Although the calling semantics should be identical to `__builtin__.map` (even using `__builtin__.map` to process arguments), it differs in that it returns a generator instead of a list. This enables lazy evaluation of the results so that other work can be done while the subprocesses are still running.

```
>>> def rangetotal(n): return n, sum(range(n))
>>> list(map(rangetotal, range(1, 6)))
[(1, 0), (2, 1), (3, 3), (4, 6), (5, 10)]
>>> list(ppmap(1, rangetotal, range(1, 6)))
[(1, 0), (2, 1), (3, 3), (4, 6), (5, 10)]
```

print_stats (*servers=None*)

print stats from the pp.Server

stats (*pool=None*)

return a string containing stats response from the pp.Server

2.13 profile module

This module contains functions for profiling in other threads and processes.

Functions for identifying a thread/process: `process_id` - get the identifier (process id) for the current process
`thread_id` - get the identifier for the current thread

Functions for controlling profiling: `enable_profiling` - initialize a profiler in the current thread/process
`start_profiling` - begin profiling everything in the current thread/process
`stop_profiling` - stop profiling everything in the current thread/process
`disable_profiling` - remove the profiler from the current thread/process

Functions that control profile statistics (pstats) output `clear_stats` - clear stored pstats from the current thread/process
`get_stats` - get stored pstats for the current thread/process
`print_stats` - print stored pstats for the current thread/process
`dump_stats` - dump stored pstats for the current thread/process

Functions that add/remove profiling: `profiled` - decorator to add profiling to a function
`not_profiled` - decorator to remove profiling from a function
`profile` - decorator for profiling a function (will enable_profiling)

2.13.1 Usage

Typical calls to pathos profiling will roughly follow this example:

```
>>> import time
>>> import random
>>> import pathos.profile as pr
>>>
>>> # build a worker function
>>> def _work(i):
...     x = random.random()
...     time.sleep(x)
...     return (i,x)
>>>
>>> # generate a 'profiled' work function
>>> config = dict(gen=pr.process_id)
>>> work = pr.profiled(**config)(_work)
>>>
```

(continues on next page)

(continued from previous page)

```

>>> # enable profiling
>>> pr.enable_profiling()
>>>
>>> # profile the work (not the map internals) in the main process
>>> from itertools import imap
>>> for i in imap(work, range(-10,0)):
...     print(i)
...
>>> # profile the map in the main process, and work in the other process
>>> from pathos.helpers import mp
>>> pool = mp.Pool(10)
>>> _uimap = pr.profiled(**config)(pool.imap_unordered)
>>> for i in _uimap(work, range(-10,0)):
...     print(i)
...
>>> # deactivate all profiling
>>> pr.disable_profiling() # in the main process
>>> tuple(_uimap(pr.disable_profiling, range(10))) # in the workers
>>> for i in _uimap(work, range(-20,-10)):
...     print(i)
...
>>> # re-activate profiling
>>> pr.enable_profiling()
>>>
>>> # print stats for profile of 'import math' in another process
>>> def test_import(module):
...     __import__(module)
...
>>> import pathos.pools as pp
>>> pool = pp.ProcessPool(1)
>>> pr.profile('cumulative', pipe=pool.pipe)(test_import, 'pox')
    10 function calls in 0.003 seconds

```

Ordered by: cumulative time

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.003	0.003	<stdin>:1(test_import)
1	0.002	0.002	0.003	0.003	{__import__}
1	0.001	0.001	0.001	0.001	__init__.py:8(<module>)
1	0.000	0.000	0.000	0.000	shutils.py:11(<module>)
1	0.000	0.000	0.000	0.000	_disk.py:15(<module>)
1	0.000	0.000	0.000	0.000	{eval}
1	0.000	0.000	0.000	0.000	utils.py:11(<module>)
1	0.000	0.000	0.000	0.000	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	info.py:2(<module>)
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' _

↪objects}

```

>>> pool.close()
>>> pool.join()
>>> pool.clear()

```

Notes

This module leverages the python's cProfile module, and is primarily a high-level interface to that module that strives to make profiling in a different thread or process easier. The use of pathos.pools are suggested, however are not required (as seen in the example above).

In many cases, profiling in another thread is not necessary, and either of the following can be sufficient/better for timing and profiling:

```
$ python -c "import time; s=time.time(); import pathos; print time.time()-s"
$ python -c "import cProfile; p=cProfile.Profile(); p.enable(); import pathos; p.
→print_stats('cumulative')"
```

This module was inspired by: <http://stackoverflow.com/a/32522579/4646678>.

clear_stats (*args)
clear all stored profiling results from the current thread/process

disable_profiling (*args)
remove the profiler instance from the current thread/process

dump_stats (*args, **kwds)
dump all stored profiling results for the current thread/process

Notes

see pathos.profile.profiled for settings for *args and **kwds

enable_profiling (*args)
initialize a profiler instance in the current thread/process

get_stats (*args)
get all stored profiling results for the current thread/process

not_profiled (f)
decorator to remove profiling (due to 'profiled') from a function

print_stats (*args, **kwds)
print all stored profiling results for the current thread/process

process_id ()
get the identifier (process id) for the current process

class profile (sort=None, **config)
Bases: object

decorator for profiling a function (will enable profiling)

sort is integer index of column in pstats output for sorting

Important class members: pipe - pipe instance in which profiling is active

Example

```
>>> import time
>>> import random
>>> import pathos.profile as pr
>>>
```

(continues on next page)

(continued from previous page)

```

... def work():
...     x = random.random()
...     time.sleep(x)
...     return x
...
>>> # profile the work; print pstats info
>>> pr.profile()(work)
      4 function calls in 0.136 seconds

```

Ordered by: standard name

```

ncalls tottime  percall  cumtime  percall filename:lineno(function)  1  0.000  0.000  0.136  0.136
<stdin>:1(work)  1  0.000  0.000  0.000  0.000 {method 'disable' of '_lsprof.Profiler' objects}  1
0.000  0.000  0.000  0.000 {method 'random' of '_random.Random' objects}  1  0.136  0.136  0.136
0.136 {time.sleep}

```

0.1350568110491419 >>>

NOTE: pipe provided should come from pool built with nodes=1. Other configuration keywords (config) are passed to 'pr.profiled'. Output can be ordered by setting sort to one of the following: 'calls' - call count 'cumulative' - cumulative time 'cumtime' - cumulative time 'file' - file name 'filename' - file name 'module' - file name 'ncalls' - call count 'pcalls' - primitive call count 'line' - line number 'name' - function name 'nfl' - name/file/line 'stdname' - standard name 'time' - internal time 'tottime' - internal time

`__call__` (...) <==> x(...)

`__dict__` = dict_proxy({'__module__': 'pathos.profile', '__dict__': <attribute '__dict__' of 'pathos.profile' objects>})

`__init__` (sort=None, **config)

sort is integer index of column in pstats output for sorting

Important class members: pipe - pipe instance in which profiling is active

Example

```

>>> import time
>>> import random
>>> import pathos.profile as pr
>>>
... def work():
...     x = random.random()
...     time.sleep(x)
...     return x
...
>>> # profile the work; print pstats info
>>> pr.profile()(work)
      4 function calls in 0.136 seconds

```

Ordered by: standard name

```

ncalls tottime  percall  cumtime  percall filename:lineno(function)  1  0.000  0.000  0.136  0.136
<stdin>:1(work)  1  0.000  0.000  0.000  0.000 {method 'disable' of '_lsprof.Profiler' objects}  1
1  0.000  0.000  0.000  0.000 {method 'random' of '_random.Random' objects}  1  0.136  0.136
0.136  0.136 {time.sleep}

```

0.1350568110491419 >>>

NOTE: pipe provided should come from pool built with nodes=1. Other configuration keywords (config) are passed to 'pr.profiled'. Output can be ordered by setting sort to one of the following: 'calls' - call count 'cumulative' - cumulative time 'cumtime' - cumulative time 'file' - file name 'filename' - file name 'module' - file name 'ncalls' - call count 'pcalls' - primitive call count 'line' - line number 'name' - function name 'nfl' - name/file/line 'stdname' - standard name 'time' - internal time 'tottime' - internal time

`__module__ = 'pathos.profile'`

`__weakref__`

list of weak references to the object (if defined)

class profiled (*gen=None, prefix='id-', suffix='.prof'*)

Bases: `object`

decorator for profiling a function (does not call enable profiling)

y=gen(), with y an identifier (e.g. `current_process().pid`)

Important class members: `prefix` - string prefix for pstats filename [default: 'id-'] `suffix` - string suffix for pstats filename [default: '.prof'] `pid` - function for obtaining id of current process/thread `sort` - integer index of column in pstats output for sorting

Example

```
>>> import time
>>> import random
>>> import pathos.profile as pr
>>>
>>> config = dict(gen=pr.process_id)
>>> @pr.profiled(**config)
... def work(i):
...     x = random.random()
...     time.sleep(x)
...     return (i,x)
...
>>> pr.enable_profiling()
>>> from itertools import imap
>>> # profile the work (not the map internals); write to file for pstats
>>> for i in imap(work, range(-10,0)):
...     print(i)
...
...

```

NOTE: If gen is a bool or string, then sort=gen and pid is not used. Otherwise, pid=gen and sort is not used. Output can be ordered by setting gen to one of the following: 'calls' - call count 'cumulative' - cumulative time 'cumtime' - cumulative time 'file' - file name 'filename' - file name 'module' - file name 'ncalls' - call count 'pcalls' - primitive call count 'line' - line number 'name' - function name 'nfl' - name/file/line 'stdname' - standard name 'time' - internal time 'tottime' - internal time

`__call__` (...) <==> x(...)

`__dict__` = `dict_proxy`({'__module__': 'pathos.profile', '__dict__': <attribute '__dic

`__init__` (*gen=None, prefix='id-', suffix='.prof'*)

y=gen(), with y an identifier (e.g. `current_process().pid`)

Important class members: prefix - string prefix for pstats filename [default: 'id-'] suffix - string suffix for pstats filename [default: '.prof'] pid - function for obtaining id of current process/thread sort - integer index of column in pstats output for sorting

Example

```
>>> import time
>>> import random
>>> import pathos.profile as pr
>>>
>>> config = dict(gen=pr.process_id)
>>> @pr.profiled(**config)
... def work(i):
...     x = random.random()
...     time.sleep(x)
...     return (i,x)
...
>>> pr.enable_profiling()
>>> from itertools import imap
>>> # profile the work (not the map internals); write to file for pstats
>>> for i in imap(work, range(-10,0)):
...     print(i)
...
...

```

NOTE: If gen is a bool or string, then sort=gen and pid is not used. Otherwise, pid=gen and sort is not used. Output can be ordered by setting gen to one of the following: 'calls' - call count 'cumulative' - cumulative time 'cumtime' - cumulative time 'file' - file name 'filename' - file name 'module' - file name 'ncalls' - call count 'pcalls' - primitive call count 'line' - line number 'name' - function name 'nfl' - name/file/line 'stdname' - standard name 'time' - internal time 'tottime' - internal time

`__module__` = 'pathos.profile'

`__weakref__`

list of weak references to the object (if defined)

`start_profiling(*args)`

begin profiling everything in the current thread/process

`stop_profiling(*args)`

stop profiling everything in the current thread/process

`thread_id()`

get the identifier for the current thread

2.14 python module

This module contains map and pipe interfaces to standard (i.e. serial) python.

Pipe methods provided: pipe - blocking communication pipe [returns: value]

Map methods provided: map - blocking and ordered worker pool [returns: list] imap - non-blocking and ordered worker pool [returns: iterator]

2.14.1 Usage

A typical call to a pathos python map will roughly follow this example:

```
>>> # instantiate and configure the worker pool
>>> from pathos.serial import SerialPool
>>> pool = SerialPool()
>>>
>>> # do a blocking map on the chosen function
>>> print(pool.map(pow, [1,2,3,4], [5,6,7,8]))
>>>
>>> # do a non-blocking map, then extract the results from the iterator
>>> results = pool.imap(pow, [1,2,3,4], [5,6,7,8])
>>> print("...")
>>> print(list(results))
>>>
>>> # do one item at a time, using a pipe
>>> print(pool.pipe(pow, 1, 5))
>>> print(pool.pipe(pow, 2, 6))
```

Notes

This worker pool leverages the built-in python maps, and thus does not have limitations due to serialization of the function *f* or the sequences in *args*. The maps in this worker pool have full functionality whether run from a script or in the python interpreter, and work reliably for both imported and interactively-defined functions.

PythonSerial

alias of `pathos.serial.SerialPool`

2.15 secure module

class Copier (*name=None, **kwds*)

Bases: `pathos.connection.Pipe`

a popen-based copier for parallel and distributed computing.

create a copier

Inputs: *name*: a unique identifier (string) for the launcher *source*: hostname:path of original [user@host:path is also valid] *destination*: hostname:path for copy [user@host:path is also valid] *launcher*: remote service mechanism (i.e. scp, cp) [default = 'scp'] *options*: remote service options (i.e. -v, -P) [default = ''] *background*: run in background [default = False] *decode*: ensure response is 'ascii' [default = True] *stdin*: file-like object to serve as standard input for the remote process

`__call__` (***kwds*)

configure the copier using given keywords:

(Re)configure the copier for the following inputs: *source*: hostname:path of original [user@host:path is also valid] *destination*: hostname:path for copy [user@host:path is also valid] *launcher*: remote service mechanism (i.e. scp, cp) [default = 'scp'] *options*: remote service options (i.e. -v, -P) [default = ''] *background*: run in background [default = False] *decode*: ensure response is 'ascii' [default = True] *stdin*: file-like object to serve as standard input for the remote process

`__init__` (*name=None, **kwds*)

create a copier

Inputs: name: a unique identifier (string) for the launcher source: hostname:path of original [user@host:path is also valid] destination: hostname:path for copy [user@host:path is also valid] launcher: remote service mechanism (i.e. scp, cp) [default = 'scp'] options: remote service options (i.e. -v, -P) [default = ''] background: run in background [default = False] decode: ensure response is 'ascii' [default = True] stdin: file-like object to serve as standard input for the remote process

`__module__ = 'pathos.secure.copier'`

`config (**kws)`

configure the copier using given keywords:

(Re)configure the copier for the following inputs: source: hostname:path of original [user@host:path is also valid] destination: hostname:path for copy [user@host:path is also valid] launcher: remote service mechanism (i.e. scp, cp) [default = 'scp'] options: remote service options (i.e. -v, -P) [default = ''] background: run in background [default = False] decode: ensure response is 'ascii' [default = True] stdin: file-like object to serve as standard input for the remote process

`class Pipe (name=None, **kws)`

Bases: `pathos.connection.Pipe`

a popen-based ssh-pipe for parallel and distributed computing.

create a ssh pipe

Inputs: name: a unique identifier (string) for the pipe host: hostname to receive command [user@host is also valid] command: a command to send [default = 'echo <name>'] launcher: remote service mechanism (i.e. ssh, rsh) [default = 'ssh'] options: remote service options (i.e. -v, -N, -L) [default = ''] background: run in background [default = False] decode: ensure response is 'ascii' [default = True] stdin: file-like object to serve as standard input for the remote process

`__call__ (**kws)`

configure a remote command using given keywords:

(Re)configure the copier for the following inputs: host: hostname to receive command [user@host is also valid] command: a command to send [default = 'echo <name>'] launcher: remote service mechanism (i.e. ssh, rsh) [default = 'ssh'] options: remote service options (i.e. -v, -N, -L) [default = ''] background: run in background [default = False] decode: ensure response is 'ascii' [default = True] stdin: file-like object to serve as standard input for the remote process

`__init__ (name=None, **kws)`

create a ssh pipe

Inputs: name: a unique identifier (string) for the pipe host: hostname to receive command [user@host is also valid] command: a command to send [default = 'echo <name>'] launcher: remote service mechanism (i.e. ssh, rsh) [default = 'ssh'] options: remote service options (i.e. -v, -N, -L) [default = ''] background: run in background [default = False] decode: ensure response is 'ascii' [default = True] stdin: file-like object to serve as standard input for the remote process

`__module__ = 'pathos.secure.connection'`

`config (**kws)`

configure a remote command using given keywords:

(Re)configure the copier for the following inputs: host: hostname to receive command [user@host is also valid] command: a command to send [default = 'echo <name>'] launcher: remote service mechanism (i.e. ssh, rsh) [default = 'ssh'] options: remote service options (i.e. -v, -N, -L) [default = ''] background: run in background [default = False] decode: ensure response is 'ascii' [default = True] stdin: file-like object to serve as standard input for the remote process

`class Tunnel (name=None, **kws)`

Bases: `object`

a ssh-tunnel launcher for parallel and distributed computing.

create a ssh tunnel launcher

Inputs: name – a unique identifier (string) for the launcher

MAXPORT = 65535

MINPORT = 1024

__Tunnel__disconnect ()

disconnect tunnel internals

__dict__ = dict_proxy({'__module__': 'pathos.secure.tunnel', '_connect': <function __

__init__ (name=None, **kws)

create a ssh tunnel launcher

Inputs: name – a unique identifier (string) for the launcher

__module__ = 'pathos.secure.tunnel'

__repr__ () <==> repr(x)

__weakref__

list of weak references to the object (if defined)

__connect (localport, remotehost, remoteport, through=None)

connect (host, port=None, through=None)

establish a secure shell tunnel between local and remote host

Input: host – remote hostname [user@host:path is also valid] port – remote port number

Additional Input: through – ‘tunnel-through’ hostname [default = None]

disconnect ()

destroy the ssh tunnel

verbose = True

exception TunnelException

Bases: exceptions.Exception

Exception for failure to establish ssh tunnel

__module__ = 'pathos.secure.tunnel'

__weakref__

list of weak references to the object (if defined)

2.15.1 pathos.secure module documentation

connection module

This module contains the derived class for secure shell (ssh) launchers See the following for an example.

Usage

A typical call to a ‘ssh pipe’ will roughly follow this example:

```
>>> # instantiate the pipe, providing it with a unique identifier
>>> pipe = Pipe('launcher')
>>>
>>> # configure the pipe to perform the command on the selected host
>>> pipe(command='hostname', host='remote.host.edu')
>>>
>>> # execute the launch and retrieve the response
>>> pipe.launch()
>>> print(pipe.response())
```

class Pipe (*name=None, **kws*)

Bases: *pathos.connection.Pipe*

a popen-based ssh-pipe for parallel and distributed computing.

create a ssh pipe

Inputs: name: a unique identifier (string) for the pipe host: hostname to receive command [*user@host* is also valid] command: a command to send [default = 'echo <name>'] launcher: remote service mechanism (i.e. ssh, rsh) [default = 'ssh'] options: remote service options (i.e. -v, -N, -L) [default = ''] background: run in background [default = False] decode: ensure response is 'ascii' [default = True] stdin: file-like object to serve as standard input for the remote process

__call__ (***kws*)

configure a remote command using given keywords:

(Re)configure the copier for the following inputs: host: hostname to receive command [*user@host* is also valid] command: a command to send [default = 'echo <name>'] launcher: remote service mechanism (i.e. ssh, rsh) [default = 'ssh'] options: remote service options (i.e. -v, -N, -L) [default = ''] background: run in background [default = False] decode: ensure response is 'ascii' [default = True] stdin: file-like object to serve as standard input for the remote process

__init__ (*name=None, **kws*)

create a ssh pipe

Inputs: name: a unique identifier (string) for the pipe host: hostname to receive command [*user@host* is also valid] command: a command to send [default = 'echo <name>'] launcher: remote service mechanism (i.e. ssh, rsh) [default = 'ssh'] options: remote service options (i.e. -v, -N, -L) [default = ''] background: run in background [default = False] decode: ensure response is 'ascii' [default = True] stdin: file-like object to serve as standard input for the remote process

__module__ = 'pathos.secure.connection'

config (***kws*)

configure a remote command using given keywords:

(Re)configure the copier for the following inputs: host: hostname to receive command [*user@host* is also valid] command: a command to send [default = 'echo <name>'] launcher: remote service mechanism (i.e. ssh, rsh) [default = 'ssh'] options: remote service options (i.e. -v, -N, -L) [default = ''] background: run in background [default = False] decode: ensure response is 'ascii' [default = True] stdin: file-like object to serve as standard input for the remote process

copier module

This module contains the derived class for launching secure copy (scp) commands. See the following for an example.

Usage

A typical call to a 'scp launcher' will roughly follow this example:

```
>>> # instantiate the launcher, providing it with a unique identifier
>>> copier = Copier('copier')
>>>
>>> # configure and launch the copy to the selected destination
>>> copier(source='~/foo.txt', destination='remote.host.edu:~')
>>> copier.launch()
>>>
>>> # configure and launch the copied file to a new destination
>>> copier(source='remote.host.edu:~/foo.txt', destination='.')
>>> copier.launch()
>>> print(copier.response())
```

exception FileNotFoundError

Bases: `exceptions.Exception`

Exception for improper source or destination format

`__module__` = `'pathos.secure.copier'`

`__weakref__`

list of weak references to the object (if defined)

class Copier (name=None, **kws)

Bases: `pathos.connection.Pipe`

a popen-based copier for parallel and distributed computing.

create a copier

Inputs: name: a unique identifier (string) for the launcher source: hostname:path of original [user@host:path is also valid] destination: hostname:path for copy [user@host:path is also valid] launcher: remote service mechanism (i.e. scp, cp) [default = 'scp'] options: remote service options (i.e. -v, -P) [default = ''] background: run in background [default = False] decode: ensure response is 'ascii' [default = True] stdin: file-like object to serve as standard input for the remote process

`__call__` (**kws)

configure the copier using given keywords:

(Re)configure the copier for the following inputs: source: hostname:path of original [user@host:path is also valid] destination: hostname:path for copy [user@host:path is also valid] launcher: remote service mechanism (i.e. scp, cp) [default = 'scp'] options: remote service options (i.e. -v, -P) [default = ''] background: run in background [default = False] decode: ensure response is 'ascii' [default = True] stdin: file-like object to serve as standard input for the remote process

`__init__` (name=None, **kws)

create a copier

Inputs: name: a unique identifier (string) for the launcher source: hostname:path of original [user@host:path is also valid] destination: hostname:path for copy [user@host:path is also valid] launcher: remote service mechanism (i.e. scp, cp) [default = 'scp'] options: remote service options (i.e. -v, -P) [default = ''] background: run in background [default = False] decode: ensure response is 'ascii' [default = True] stdin: file-like object to serve as standard input for the remote process

`__module__` = `'pathos.secure.copier'`

`config` (**kws)

configure the copier using given keywords:

(Re)configure the copier for the following inputs: source: hostname:path of original [user@host:path is also valid] destination: hostname:path for copy [user@host:path is also valid] launcher: remote service mechanism (i.e. scp, cp) [default = 'scp'] options: remote service options (i.e. -v, -P) [default = ''] background: run in background [default = False] decode: ensure response is 'ascii' [default = True] stdin: file-like object to serve as standard input for the remote process

tunnel module

This module contains the base class for secure tunnel connections, and describes the pathos tunnel interface. See the following for an example.

Usage

A typical call to a pathos 'tunnel' will roughly follow this example:

```
>>> # instantiate the tunnel, providing it with a unique identifier
>>> tunnel = Tunnel('tunnel')
>>>
>>> # establish a tunnel to the remote host and port
>>> remotehost = 'remote.host.edu'
>>> remoteport = 12345
>>> localport = tunnel.connect(remotehost, remoteport)
>>> print("Tunnel connected at local port: %s" % tunnel._lport)
>>>
>>> # pause script execution to maintain the tunnel (i.e. do something)
>>> sys.stdin.readline()
>>>
>>> # tear-down the tunneled connection
>>> tunnel.disconnect()
```

class Tunnel (name=None, **kws)

Bases: object

a ssh-tunnel launcher for parallel and distributed computing.

create a ssh tunnel launcher

Inputs: name – a unique identifier (string) for the launcher

MAXPORT = 65535

MINPORT = 1024

__Tunnel__disconnect ()
disconnect tunnel internals

__dict__ = dict_proxy({'__module__': 'pathos.secure.tunnel', '__connect': <function __

__init__ (name=None, **kws)
create a ssh tunnel launcher

Inputs: name – a unique identifier (string) for the launcher

__module__ = 'pathos.secure.tunnel'

__repr__ () <==> repr(x)

__weakref__
list of weak references to the object (if defined)

`_connect` (*localport, remotehost, remoteport, through=None*)

`connect` (*host, port=None, through=None*)

establish a secure shell tunnel between local and remote host

Input: host – remote hostname [`user@host:port` is also valid] port – remote port number

Additional Input: through – ‘tunnel-through’ hostname [default = None]

`disconnect` ()

destroy the ssh tunnel

verbose = True

exception TunnelException

Bases: `exceptions.Exception`

Exception for failure to establish ssh tunnel

`__module__` = `'pathos.secure.tunnel'`

`__weakref__`

list of weak references to the object (if defined)

2.16 selector module

This module implements a selector class, which can be used to dispatch events and for event handler wrangling.

class Selector

Bases: `object`

Selector object for watching and event notification.

Takes no initial input.

`_TIMEOUT` = 0.5

`__dict__` = `dict_proxy({'__module__': 'pathos.selector', 'notifyOnReadReady': <function`

`__init__` ()

Takes no initial input.

`__module__` = `'pathos.selector'`

`__weakref__`

list of weak references to the object (if defined)

`_cleanup` ()

`_debug` = `<logging.Logger object>`

`_dispatch` (*handlers, entities*)

`_watch` ()

`notifyOnException` (*fd, handler*)

add <handler> to the list of routines to call when <fd> raises an exception

`notifyOnInterrupt` (*handler*)

add <handler> to the list of routines to call when a signal arrives

`notifyOnReadReady` (*fd, handler*)

add <handler> to the list of routines to call when <fd> is read ready

notifyOnWriteReady (*fd, handler*)
add <handler> to the list of routines to call when <fd> is write ready

notifyWhenIdle (*handler*)
add <handler> to the list of routines to call when a timeout occurs

watch (*timeout=None*)
dispatch events to the registered handlers

2.17 serial module

This module contains map and pipe interfaces to standard (i.e. serial) python.

Pipe methods provided: pipe - blocking communication pipe [returns: value]

Map methods provided: map - blocking and ordered worker pool [returns: list] imap - non-blocking and ordered worker pool [returns: iterator]

2.17.1 Usage

A typical call to a pathos python map will roughly follow this example:

```
>>> # instantiate and configure the worker pool
>>> from pathos.serial import SerialPool
>>> pool = SerialPool()
>>>
>>> # do a blocking map on the chosen function
>>> print(pool.map(pow, [1,2,3,4], [5,6,7,8]))
>>>
>>> # do a non-blocking map, then extract the results from the iterator
>>> results = pool.imap(pow, [1,2,3,4], [5,6,7,8])
>>> print("...")
>>> print(list(results))
>>>
>>> # do one item at a time, using a pipe
>>> print(pool.pipe(pow, 1, 5))
>>> print(pool.pipe(pow, 2, 6))
```

Notes

This worker pool leverages the built-in python maps, and thus does not have limitations due to serialization of the function *f* or the sequences in *args*. The maps in this worker pool have full functionality whether run from a script or in the python interpreter, and work reliably for both imported and interactively-defined functions.

class SerialPool (**args, **kws*)
Bases: *pathos.abstract_launcher.AbstractWorkerPool*

Mapper that leverages standard (i.e. serial) python maps.

Important class members: nodes - number (and potentially description) of workers ncpus - number of worker processors servers - list of worker servers scheduler - the associated scheduler workdir - associated \$WORKDIR for scratch calculations/files

Other class members: scatter - True, if uses 'scatter-gather' (instead of 'worker-pool') source - False, if minimal use of TemporaryFiles is desired timeout - number of seconds to wait for return value from scheduler

```

__SerialPool__get_nodes ()
    get the number of nodes in the pool

__SerialPool__set_nodes (nodes)
    set the number of nodes in the pool

__module__ = 'pathos.serial'

__exiting = False

__is_alive (negate=False, run=True)

clear ()
    hard restart

close ()
    close the pool to any new jobs

imap (f, *args, **kws)
    run a batch of jobs with a non-blocking and ordered map

    Returns a list iterator of results of applying the function f to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence.

join ()
    cleanup the closed worker processes

map (f, *args, **kws)
    run a batch of jobs with a blocking and ordered map

    Returns a list of results of applying the function f to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence.

nodes
    get the number of nodes in the pool

pipe (f, *args, **kws)
    submit a job and block until results are available

    Returns result of calling the function f on a selected worker. This function will block until results are available.

restart (force=False)
    restart a closed pool

terminate ()
    a more abrupt close

```

2.18 server module

This module contains the base class for pathos servers, and describes the pathos server interface. If a third-party RPC server is selected, such as 'parallel python' (i.e. 'pp') or 'RPyC', direct calls to the third-party interface are currently used.

class Server

Bases: `object`

Server base class for pathos servers for parallel and distributed computing.

Takes no initial input.

```
__dict__ = dict_proxy({'__dict__': <attribute '__dict__' of 'Server' objects>, '__mod
__init__ ()
    Takes no initial input.
__module__ = 'pathos.server'
__weakref__
    list of weak references to the object (if defined)
activate (onTimeout=None, selector=None)
    configure the selector and install the timeout callback
deactivate ()
    turn off the selector
selector ()
    get the selector
serve (timeout)
    begin serving, and set the timeout
```

2.19 threading module

This module contains map and pipe interfaces to python's threading module.

Pipe methods provided: pipe - blocking communication pipe [returns: value] apipe - asynchronous communication pipe [returns: object]

Map methods provided: map - blocking and ordered worker pool [returns: list] imap - non-blocking and ordered worker pool [returns: iterator] uimap - non-blocking and unordered worker pool [returns: iterator] amap - asynchronous worker pool [returns: object]

2.19.1 Usage

A typical call to a pathos threading map will roughly follow this example:

```
>>> # instantiate and configure the worker pool
>>> from pathos.threading import ThreadPool
>>> pool = ThreadPool(nodes=4)
>>>
>>> # do a blocking map on the chosen function
>>> print(pool.map(pow, [1,2,3,4], [5,6,7,8]))
>>>
>>> # do a non-blocking map, then extract the results from the iterator
>>> results = pool.imap(pow, [1,2,3,4], [5,6,7,8])
>>> print("...")
>>> print(list(results))
>>>
>>> # do an asynchronous map, then get the results
>>> results = pool.amap(pow, [1,2,3,4], [5,6,7,8])
>>> while not results.ready():
...     time.sleep(5); print(".", end=' ')
...
>>> print(results.get())
>>>
>>> # do one item at a time, using a pipe
```

(continues on next page)

(continued from previous page)

```

>>> print(pool.pipe(pow, 1, 5))
>>> print(pool.pipe(pow, 2, 6))
>>>
>>> # do one item at a time, using an asynchronous pipe
>>> result1 = pool.apipe(pow, 1, 5)
>>> result2 = pool.apipe(pow, 2, 6)
>>> print(result1.get())
>>> print(result2.get())

```

Notes

This worker pool leverages the python's multiprocessing.dummy module, and thus has many of the limitations associated with that module. The function *f* and the sequences in *args* must be serializable. The maps in this worker pool have full functionality whether run from a script or in the python interpreter, and work reliably for both imported and interactively-defined functions. Unlike python's multiprocessing.dummy module, pathos.threading maps can directly utilize functions that require multiple arguments.

class ThreadPool (*args, **kws)

Bases: *pathos.abstract_launcher.AbstractWorkerPool*

Mapper that leverages python's threading.

Important class members: nodes - number (and potentially description) of workers ncpus - number of worker processors servers - list of worker servers scheduler - the associated scheduler workdir - associated \$WORKDIR for scratch calculations/files

Other class members: scatter - True, if uses 'scatter-gather' (instead of 'worker-pool') source - False, if minimal use of TemporaryFiles is desired timeout - number of seconds to wait for return value from scheduler

NOTE: if number of nodes is not given, will autodetect processors

__ThreadPool__get_nodes ()

get the number of nodes used in the map

__ThreadPool__set_nodes (nodes)

set the number of nodes used in the map

__init__ (*args, **kws)

Important class members: nodes - number (and potentially description) of workers ncpus - number of worker processors servers - list of worker servers scheduler - the associated scheduler workdir - associated \$WORKDIR for scratch calculations/files

Other class members: scatter - True, if uses 'scatter-gather' (instead of 'worker-pool') source - False, if minimal use of TemporaryFiles is desired timeout - number of seconds to wait for return value from scheduler

NOTE: if number of nodes is not given, will autodetect processors

__module__ = 'pathos.threading'

__repr__ () <==> repr(x)

__clear ()

Remove server with matching state

__serve (nodes=None)

Create a new server if one isn't already initialized

amap (*f*, *args, **kws)

run a batch of jobs with an asynchronous map

Returns a results object which contains the results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence. To retrieve the results, call the `get()` method on the returned results object. The call to `get()` is blocking, until all results are retrieved. Use the `ready()` method on the result object to check if all results are ready.

apipe (*f*, *args, **kws)

submit a job asynchronously to a queue

Returns a results object which contains the result of calling the function *f* on a selected worker. To retrieve the results, call the `get()` method on the returned results object. The call to `get()` is blocking, until the result is available. Use the `ready()` method on the results object to check if the result is ready.

clear ()

Remove server with matching state

close ()

close the pool to any new jobs

imap (*f*, *args, **kws)

run a batch of jobs with a non-blocking and ordered map

Returns a list iterator of results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence.

join ()

cleanup the closed worker processes

map (*f*, *args, **kws)

run a batch of jobs with a blocking and ordered map

Returns a list of results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence.

nodes

get the number of nodes used in the map

nthreads

get the number of nodes used in the map

pipe (*f*, *args, **kws)

submit a job and block until results are available

Returns result of calling the function *f* on a selected worker. This function will block until results are available.

restart (*force=False*)

restart a closed pool

terminate ()

a more abrupt close

uimap (*f*, *args, **kws)

run a batch of jobs with a non-blocking and unordered map

Returns a list iterator of results of applying the function *f* to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence. The order of the resulting sequence is not guaranteed.

`_ThreadPool` (*processes=None, initializer=None, initargs=()*)

2.20 util module

utilities for distributed computing

`_b` (*string, codec=None*)
convert string to bytes using the given codec (default is 'ascii')

`_str` (*byte, codec=None*)
convert bytes to string using the given codec (default is 'ascii')

`print_exc_info` ()
thread-safe return of string from print_exception call

`spawn` (*onParent, onChild*)
a unidirectional fork wrapper
Calls onParent(pid, fromchild) in parent process, onChild(pid, toparent) in child process.

`spawn2` (*onParent, onChild*)
a bidirectional fork wrapper
Calls onParent(pid, fromchild, tochild) in parent process, onChild(pid, fromparent, toparent) in child process.

2.21 xmlrpc module

class XMLRPCRequestHandler (*server, socket*)
Bases: BaseHTTPServer.BaseHTTPRequestHandler

create a XML-RPC request handler

Override BaseHTTPRequestHandler.__init__(): we need to be able to have (potentially) multiple handler objects at a given time.

Inputs: server – server object to handle requests for socket – socket connection

`__init__` (*server, socket*)
Override BaseHTTPRequestHandler.__init__(): we need to be able to have (potentially) multiple handler objects at a given time.

Inputs: server – server object to handle requests for socket – socket connection

`__module__` = 'pathos.xmlrpc.server'

`_debug` = <logging.Logger object>

`_sendResponse` (*response*)
Write the XML-RPC response

`do_POST` ()
Access point from HTTP handler

`log_message` (*format, *args*)
Overriding BaseHTTPRequestHandler.log_message()

class XMLRPCServer (*host, port*)

Bases: *pathos.server.Server*, *SimpleXMLRPCServer.SimpleXMLRPCDispatcher*

extends base pathos server to an XML-RPC dispatcher

create a XML-RPC server

Takes two initial inputs: *host* – hostname of XML-RPC server *port* – port number for server requests

__init__ (*host, port*)

create a XML-RPC server

Takes two initial inputs: *host* – hostname of XML-RPC server *port* – port number for server requests

__module__ = `'pathos.xmlrpc.server'`

_handleMessageFromChild (*selector, fd*)

handler for message from a child process

_installSocket (*host, port*)

prepare a listening socket

_marshaled_dispatch (*data, dispatch_method=None*)

override *SimpleXMLRPCDispatcher._marshaled_dispatch()* fault string

_onConnection (*selector, fd*)

upon socket connection, establish a request handler

_onSelectorIdle (*selector*)

something to do when there's no requests

_onSocketConnection (*socket*)

upon socket connections, establish a request handler

_registerChild (*pid, fromchild*)

register a child process so it can be retrieved on select events

_unregisterChild (*fd*)

remove a child process from active process register

activate ()

install callbacks

serve ()

enter the select loop... and wait for service requests

2.21.1 pathos.xmlrpc module documentation

server module

This module contains the base class for pathos XML-RPC servers, and derives from python's *SimpleXMLRPCServer*, and the base class for XML-RPC request handlers, which derives from python's base HTTP request handler.

Usage

A typical setup for an XML-RPC server will roughly follow this example:


```

>>> # establish a XML-RPC server on a host at a given port
>>> host = 'localhost'
>>> port = 1234
>>> server = XMLRPCServer(host, port)
>>> print('port=%d' % server.port)
>>>
>>> # register a method the server can handle requests for
>>> def add(x, y):
...     return x + y
>>> server.register_function(add)
>>>
>>> # activate the callback methods and begin serving requests
>>> server.activate()
>>> server.serve()

```

The following is an example of how to make requests to the above server:

```

>>> # establish a proxy connection to the server at (host,port)
>>> host = 'localhost'
>>> port = 1234
>>> proxy = xmlrpclib.ServerProxy('http://%s:%d' % (host, port))
>>> print('1 + 2 = %s' % proxy.add(1, 2))
>>> print('3 + 4 = %s' % proxy.add(3, 4))

```

class XMLRPCServer (*host, port*)

Bases: *pathos.server.Server*, *SimpleXMLRPCServer.SimpleXMLRPCDispatcher*

extends base pathos server to an XML-RPC dispatcher

create a XML-RPC server

Takes two initial inputs: *host* – hostname of XML-RPC server *port* – port number for server requests

__init__ (*host, port*)

create a XML-RPC server

Takes two initial inputs: *host* – hostname of XML-RPC server *port* – port number for server requests

__module__ = 'pathos.xmlrpc.server'

_handleMessageFromChild (*selector, fd*)

handler for message from a child process

_installSocket (*host, port*)

prepare a listening socket

_marshaled_dispatch (*data, dispatch_method=None*)

override *SimpleXMLRPCDispatcher._marshaled_dispatch()* fault string

_onConnection (*selector, fd*)

upon socket connection, establish a request handler

_onSelectorIdle (*selector*)

something to do when there's no requests

_onSocketConnection (*socket*)

upon socket connections, establish a request handler

_registerChild (*pid, fromchild*)

register a child process so it can be retrieved on select events

`_unRegisterChild` (*fd*)
remove a child process from active process register

`activate` ()
install callbacks

`serve` ()
enter the select loop... and wait for service requests

class `XMLRPCRequestHandler` (*server, socket*)
Bases: `BaseHTTPServer.BaseHTTPRequestHandler`
create a XML-RPC request handler

Override `BaseHTTPRequestHandler.__init__()`: we need to be able to have (potentially) multiple handler objects at a given time.

Inputs: `server` – server object to handle requests for `socket` – socket connection

`__init__` (*server, socket*)
Override `BaseHTTPRequestHandler.__init__()`: we need to be able to have (potentially) multiple handler objects at a given time.

Inputs: `server` – server object to handle requests for `socket` – socket connection

`__module__` = `'pathos.xmlrpc.server'`

`_debug` = `<logging.Logger object>`

`_sendResponse` (*response*)
Write the XML-RPC response

`do_POST` ()
Access point from HTTP handler

`log_message` (*format, *args*)
Overriding `BaseHTTPRequestHandler.log_message()`

3.1 pathos_connect script

connect to the specified machine and start a ‘server’, ‘tunnel’, or both

Notes

Usage: `pathos_connect [hostname] [server] [remoteport] [profile]` [hostname] - name of the host to connect to
[server] - name of RPC server (assumes is installed on host) or ‘tunnel’ [remoteport] - remote port to use
for communication or ‘tunnel’ [profile] – name of shell profile to source on remote environment

Examples:

```
$ pathos_connect computer.college.edu ppserver tunnel
Usage: pathos_connect [hostname] [server] [remoteport] [profile]
  [hostname] - name of the host to connect to
  [server] - name of RPC server (assumes is installed on host) or 'tunnel'
  [remoteport] - remote port to use for communication or 'tunnel'
  [profile] -- name of shell profile to source on remote environment
  defaults are: "localhost" "tunnel" "" ""
executing {ssh -N -L 22921:computer.college.edu:15058}'

Server running at port=15058 with pid=4110
Connected to localhost at port=22921
Press <Enter> to kill server
```

3.2 portpicker script

This script prints out an available port number.

CHAPTER 4

Indices and tables

- `genindex`
- `modindex`
- `search`

—
_pathos_connect, 55
_portpicker, 55

a

pathos.abstract_launcher, 7

c

pathos.connection, 10
pathos.core, 11

h

pathos.helpers, 13
pathos.helpers.mp_helper, 13
pathos.helpers.pp_helper, 13
pathos.hosts, 17

m

pathos.mp_map, 18
pathos.multiprocessing, 18

p

pathos, ??
pathos.parallel, 21
pathos.pools, 24
pathos.portpicker, 30
pathos.pp, 31
pathos.pp_map, 32
pathos.profile, 33
pathos.python, 38

s

pathos.secure, 39
pathos.secure.connection, 41
pathos.secure.copier, 42
pathos.secure.tunnel, 44
pathos.selector, 45
pathos.serial, 46
pathos.server, 47

t

pathos.threading, 48

u

pathos.util, 51

x

pathos.xmlrpc, 51
pathos.xmlrpc.server, 52

Symbols

- `_AbstractWorkerPool__get_nodes()` (AbstractWorkerPool method), 8
- `_AbstractWorkerPool__imap()` (AbstractWorkerPool method), 8
- `_AbstractWorkerPool__init()` (AbstractWorkerPool method), 8
- `_AbstractWorkerPool__map()` (AbstractWorkerPool method), 8
- `_AbstractWorkerPool__nodes` (AbstractWorkerPool attribute), 9
- `_AbstractWorkerPool__pipe()` (AbstractWorkerPool method), 9
- `_AbstractWorkerPool__set_nodes()` (AbstractWorkerPool method), 9
- `_ApplyResult` (in module `pathos.helpers.pp_helper`), 16
- `_ApplyResult__unpickle()` (ApplyResult method), 13
- `_MapResult` (in module `pathos.helpers.pp_helper`), 17
- `_MapResult__unpickle()` (MapResult method), 14
- `_ParallelPool__get_nodes()` (ParallelPool method), 22, 24
- `_ParallelPool__get_servers()` (ParallelPool method), 22, 24
- `_ParallelPool__set_nodes()` (ParallelPool method), 22, 24
- `_ParallelPool__set_servers()` (ParallelPool method), 22, 24
- `_ProcessPool` (in module `pathos.multiprocessing`), 20
- `_ProcessPool` (in module `pathos.pools`), 30
- `_ProcessPool__get_nodes()` (ProcessPool method), 19, 26
- `_ProcessPool__set_nodes()` (ProcessPool method), 19, 26
- `_SerialPool__get_nodes()` (SerialPool method), 28, 46
- `_SerialPool__set_nodes()` (SerialPool method), 28, 47
- `_Server__add_to_active_tasks()` (Server method), 15
- `_Server__connect()` (Server method), 15
- `_Server__detect_ncpus()` (Server method), 15
- `_Server__dumpsfunc()` (Server method), 15
- `_Server__find_modules()` (Server method), 15
- `_Server__gentid()` (Server method), 15
- `_Server__get_source()` (Server method), 15
- `_Server__scheduler()` (Server method), 15
- `_Server__stat_add_job()` (Server method), 15
- `_Server__stat_add_time()` (Server method), 15
- `_Server__update_active_rworkers()` (Server method), 15
- `_TIMEOUT` (Selector attribute), 45
- `_Task` (class in `pathos.helpers.pp_helper`), 17
- `_Task__unpickle()` (_Task method), 17
- `_ThreadPool()` (in module `pathos.pools`), 30
- `_ThreadPool()` (in module `pathos.threading`), 50
- `_ThreadPool__get_nodes()` (ThreadPool method), 29, 49
- `_ThreadPool__set_nodes()` (ThreadPool method), 29, 49
- `_Tunnel__disconnect()` (Tunnel method), 41, 44
- `__call__()` (ApplyResult method), 13
- `__call__()` (Copier method), 39, 43
- `__call__()` (MapResult method), 14
- `__call__()` (Pipe method), 10, 40, 42
- `__call__()` (_Task method), 17
- `__call__()` (portnumber method), 30
- `__call__()` (profile method), 36
- `__call__()` (profiled method), 37
- `__del__()` (Server method), 15
- `__dict__` (AbstractPipeConnection attribute), 8
- `__dict__` (AbstractWorkerPool attribute), 9
- `__dict__` (MapResult attribute), 14
- `__dict__` (Pipe attribute), 10
- `__dict__` (Selector attribute), 45
- `__dict__` (Server attribute), 15, 47
- `__dict__` (Tunnel attribute), 41, 44
- `__dict__` (_Task attribute), 17
- `__dict__` (portnumber attribute), 30
- `__dict__` (profile attribute), 36
- `__dict__` (profiled attribute), 37
- `__enter__()` (AbstractWorkerPool method), 9
- `__exit__()` (AbstractWorkerPool method), 9
- `__init__()` (AbstractPipeConnection method), 8
- `__init__()` (AbstractWorkerPool method), 9
- `__init__()` (ApplyResult method), 13
- `__init__()` (Copier method), 39, 43
- `__init__()` (MapResult method), 14
- `__init__()` (ParallelPool method), 22, 24
- `__init__()` (Pipe method), 11, 40, 42

- `__init__()` (ProcessPool method), 19, 26
- `__init__()` (Selector method), 45
- `__init__()` (Server method), 15, 48
- `__init__()` (ThreadPool method), 29, 49
- `__init__()` (Tunnel method), 41, 44
- `__init__()` (XMLRPCRequestHandler method), 51, 54
- `__init__()` (XMLRPCServer method), 52, 53
- `__init__()` (`_Task` method), 17
- `__init__()` (portnumber method), 30
- `__init__()` (profile method), 36
- `__init__()` (profiled method), 37
- `__module__` (AbstractPipeConnection attribute), 8
- `__module__` (AbstractWorkerPool attribute), 9
- `__module__` (ApplyResult attribute), 13
- `__module__` (Copier attribute), 40, 43
- `__module__` (FileNotFound attribute), 43
- `__module__` (MapResult attribute), 14
- `__module__` (ParallelPool attribute), 22, 24
- `__module__` (Pipe attribute), 11, 40, 42
- `__module__` (PipeException attribute), 11
- `__module__` (ProcessPool attribute), 19, 26
- `__module__` (Selector attribute), 45
- `__module__` (SerialPool attribute), 28, 47
- `__module__` (Server attribute), 16, 48
- `__module__` (ThreadPool attribute), 29, 49
- `__module__` (TimeoutError attribute), 16
- `__module__` (Tunnel attribute), 41, 44
- `__module__` (TunnelException attribute), 41, 45
- `__module__` (XMLRPCRequestHandler attribute), 51, 54
- `__module__` (XMLRPCServer attribute), 52, 53
- `__module__` (`_Task` attribute), 17
- `__module__` (portnumber attribute), 31
- `__module__` (profile attribute), 37
- `__module__` (profiled attribute), 38
- `__repr__()` (AbstractPipeConnection method), 8
- `__repr__()` (AbstractWorkerPool method), 9
- `__repr__()` (ParallelPool method), 22, 24
- `__repr__()` (Pipe method), 11
- `__repr__()` (ProcessPool method), 19, 26
- `__repr__()` (ThreadPool method), 29, 49
- `__repr__()` (Tunnel method), 41, 44
- `__weakref__` (AbstractPipeConnection attribute), 8
- `__weakref__` (AbstractWorkerPool attribute), 9
- `__weakref__` (FileNotFound attribute), 43
- `__weakref__` (MapResult attribute), 14
- `__weakref__` (Pipe attribute), 11
- `__weakref__` (PipeException attribute), 11
- `__weakref__` (Selector attribute), 45
- `__weakref__` (Server attribute), 16, 48
- `__weakref__` (Tunnel attribute), 41, 44
- `__weakref__` (TunnelException attribute), 41, 45
- `__weakref__` (`_Task` attribute), 17
- `__weakref__` (portnumber attribute), 31
- `__weakref__` (profile attribute), 37
- `__weakref__` (profiled attribute), 38
- `_b()` (in module `pathos.util`), 51
- `_cleanup()` (Selector method), 45
- `_clear()` (ParallelPool method), 22, 24
- `_clear()` (ProcessPool method), 19, 26
- `_clear()` (ThreadPool method), 29, 49
- `_connect()` (Tunnel method), 41, 44
- `_debug` (Pipe attribute), 11
- `_debug` (Selector attribute), 45
- `_debug` (XMLRPCRequestHandler attribute), 51, 54
- `_dispatch()` (Selector method), 45
- `_equals()` (ParallelPool method), 22, 24
- `_execute()` (Pipe method), 11
- `_exiting` (SerialPool attribute), 28, 47
- `_handleMessageFromChild()` (XMLRPCServer method), 52, 53
- `_installSocket()` (XMLRPCServer method), 52, 53
- `_is_alive()` (ParallelPool method), 22, 25
- `_is_alive()` (SerialPool method), 28, 47
- `_marshaled_dispatch()` (XMLRPCServer method), 52, 53
- `_onConnection()` (XMLRPCServer method), 52, 53
- `_onSelectorIdle()` (XMLRPCServer method), 52, 53
- `_onSocketConnection()` (XMLRPCServer method), 52, 53
- `_pathos_connect` (module), 55
- `_portpicker` (module), 55
- `_profiles` (in module `pathos.hosts`), 17
- `_registerChild()` (XMLRPCServer method), 52, 53
- `_run_local()` (Server method), 16
- `_run_remote()` (Server method), 16
- `_sendResponse()` (XMLRPCRequestHandler method), 51, 54
- `_serve()` (AbstractWorkerPool method), 9
- `_serve()` (ParallelPool method), 22, 25
- `_serve()` (ProcessPool method), 19, 26
- `_serve()` (ThreadPool method), 29, 49
- `_set()` (MapResult method), 14
- `_str()` (in module `pathos.util`), 51
- `_unRegisterChild()` (XMLRPCServer method), 52, 53
- `_watch()` (Selector method), 45

A

- AbstractPipeConnection (class in `pathos.abstract_launcher`), 8
- AbstractWorkerPool (class in `pathos.abstract_launcher`), 8
- `activate()` (Server method), 48
- `activate()` (XMLRPCServer method), 52, 54
- `amap()` (AbstractWorkerPool method), 9
- `amap()` (ParallelPool method), 23, 25
- `amap()` (ProcessPool method), 19, 26
- `amap()` (ThreadPool method), 29, 49
- `apipe()` (AbstractWorkerPool method), 9
- `apipe()` (ParallelPool method), 23, 25

apipe() (ProcessPool method), 20, 26
 apipe() (ThreadPool method), 29, 50
 ApplyResult (class in pathos.helpers.pp_helper), 13

C

callback (ApplyResult attribute), 13
 callbackargs (ApplyResult attribute), 13
 citation() (in module pathos), 5
 clear() (AbstractWorkerPool method), 9
 clear() (ParallelPool method), 23, 25
 clear() (ProcessPool method), 20, 27
 clear() (SerialPool method), 28, 47
 clear() (ThreadPool method), 29, 50
 clear_stats() (in module pathos.profile), 35
 close() (ParallelPool method), 23, 25
 close() (ProcessPool method), 20, 27
 close() (SerialPool method), 28, 47
 close() (ThreadPool method), 29, 50
 config() (Copier method), 40, 43
 config() (Pipe method), 11, 40, 42
 connect() (in module pathos.core), 12
 connect() (Tunnel method), 41, 45
 connect1() (Server method), 16
 Copier (class in pathos.secure), 39
 Copier (class in pathos.secure.copier), 43
 copy() (in module pathos.core), 11
 cpu_count() (in module pathos.helpers), 13

D

deactivate() (Server method), 48
 default_port (Server attribute), 16
 default_secret (Server attribute), 16
 destroy() (Server method), 16
 disable_profiling() (in module pathos.profile), 35
 disconnect() (Tunnel method), 41, 45
 do_POST() (XMLRPCRequestHandler method), 51, 54
 dump_stats() (in module pathos.profile), 35

E

enable_profiling() (in module pathos.profile), 35
 execute() (in module pathos.core), 11

F

FileNotFound, 43
 finalize() (_Task method), 17
 finalize() (ApplyResult method), 13
 finalize() (MapResult method), 14
 finished (ApplyResult attribute), 14
 freeze_support() (in module pathos.helpers), 13

G

get() (ApplyResult method), 14
 get() (MapResult method), 14

get_active_nodes() (Server method), 16
 get_ncpus() (Server method), 16
 get_profile() (in module pathos.hosts), 17
 get_profiles() (in module pathos.hosts), 17
 get_stats() (in module pathos.profile), 35
 get_stats() (Server method), 16
 getchild() (in module pathos.core), 12
 getpid() (in module pathos.core), 12
 getppid() (in module pathos.core), 12
 group (ApplyResult attribute), 14

I

imap() (AbstractWorkerPool method), 9
 imap() (ParallelPool method), 23, 25
 imap() (ProcessPool method), 20, 27
 imap() (SerialPool method), 28, 47
 imap() (ThreadPool method), 29, 50
 insert() (Server method), 16

J

join() (ParallelPool method), 23, 25
 join() (ProcessPool method), 20, 27
 join() (SerialPool method), 28, 47
 join() (ThreadPool method), 29, 50

K

kill() (in module pathos.core), 12
 kill() (Pipe method), 11

L

launch() (Pipe method), 11
 license() (in module pathos), 5
 lock (ApplyResult attribute), 14
 log_message() (XMLRPCRequestHandler method), 51, 54
 logger() (in module pathos), 5

M

map() (AbstractWorkerPool method), 9
 map() (ParallelPool method), 23, 25
 map() (ProcessPool method), 20, 27
 map() (SerialPool method), 28, 47
 map() (ThreadPool method), 29, 50
 MapResult (class in pathos.helpers.pp_helper), 14
 MAXPORT (Tunnel attribute), 41, 44
 MINPORT (Tunnel attribute), 41, 44
 mp_map() (in module pathos.mp_map), 18

N

ncpus (ParallelPool attribute), 23, 25
 ncpus (ProcessPool attribute), 20, 27
 nodes (ParallelPool attribute), 23, 25
 nodes (ProcessPool attribute), 20, 27

nodes (SerialPool attribute), 28, 47
 nodes (ThreadPool attribute), 30, 50
 not_profiled() (in module pathos.profile), 35
 notifyOnException() (Selector method), 45
 notifyOnInterrupt() (Selector method), 45
 notifyOnReadReady() (Selector method), 45
 notifyOnWriteReady() (Selector method), 45
 notifyWhenIdle() (Selector method), 46
 nthreads (ThreadPool attribute), 30, 50

P

ParallelPool (class in pathos.parallel), 22
 ParallelPool (class in pathos.pools), 24
 ParallelPythonPool (in module pathos.pp), 32
 pathos (module), 1
 pathos.abstract_launcher (module), 7
 pathos.connection (module), 10
 pathos.core (module), 11
 pathos.helpers (module), 13
 pathos.helpers.mp_helper (module), 13
 pathos.helpers.pp_helper (module), 13
 pathos.hosts (module), 17
 pathos.mp_map (module), 18
 pathos.multiprocessing (module), 18
 pathos.parallel (module), 21
 pathos.pools (module), 24
 pathos.portpicker (module), 30
 pathos.pp (module), 31
 pathos.pp_map (module), 32
 pathos.profile (module), 33
 pathos.python (module), 38
 pathos.secure (module), 39
 pathos.secure.connection (module), 41
 pathos.secure.copier (module), 42
 pathos.secure.tunnel (module), 44
 pathos.selector (module), 45
 pathos.serial (module), 46
 pathos.server (module), 47
 pathos.threading (module), 48
 pathos.util (module), 51
 pathos.xmlrpc (module), 51
 pathos.xmlrpc.server (module), 52
 pid() (Pipe method), 11
 Pipe (class in pathos.connection), 10
 Pipe (class in pathos.secure), 40
 Pipe (class in pathos.secure.connection), 42
 pipe() (AbstractWorkerPool method), 10
 pipe() (ParallelPool method), 23, 25
 pipe() (ProcessPool method), 20, 27
 pipe() (SerialPool method), 28, 47
 pipe() (ThreadPool method), 30, 50
 PipeException, 11
 portnumber (class in pathos.portpicker), 30
 pp_map() (in module pathos.pp_map), 32

ppmap() (in module pathos.pp_map), 32
 ppServer (in module pathos.pp_map), 32
 print_exc_info() (in module pathos.util), 51
 print_stats() (in module pathos.pp_map), 33
 print_stats() (in module pathos.profile), 35
 print_stats() (Server method), 16
 process_id() (in module pathos.profile), 35
 ProcessPool (class in pathos.multiprocessing), 19
 ProcessPool (class in pathos.pools), 26
 ProcessPool (in module pathos.helpers), 13
 profile (class in pathos.profile), 35
 profiled (class in pathos.profile), 37
 PythonSerial (in module pathos.python), 39

Q

queue() (MapResult method), 14

R

random_seed() (in module pathos.helpers.mp_helper), 13
 random_state() (in module pathos.helpers.mp_helper), 13
 randomport() (in module pathos.core), 12
 randomport() (in module pathos.portpicker), 31
 ready() (ApplyResult method), 14
 ready() (MapResult method), 14
 register() (in module pathos.hosts), 17
 register_profiles() (in module pathos.hosts), 17
 response() (Pipe method), 11
 restart() (ParallelPool method), 23, 25
 restart() (ProcessPool method), 20, 27
 restart() (SerialPool method), 28, 47
 restart() (ThreadPool method), 30, 50

S

Selector (class in pathos.selector), 45
 selector() (Server method), 48
 SerialPool (class in pathos.pools), 27
 SerialPool (class in pathos.serial), 46
 serve() (in module pathos.core), 12
 serve() (Server method), 48
 serve() (XMLRPCServer method), 52, 54
 server (ApplyResult attribute), 14
 Server (class in pathos.helpers.pp_helper), 14
 Server (class in pathos.server), 47
 servers (ParallelPool attribute), 23, 25
 set_ncpus() (Server method), 16
 spawn() (in module pathos.util), 51
 spawn2() (in module pathos.util), 51
 starargs() (in module pathos.helpers.mp_helper), 13
 start_profiling() (in module pathos.profile), 38
 stats() (in module pathos.parallel), 24
 stats() (in module pathos.pp), 32
 stats() (in module pathos.pp_map), 33
 stop_profiling() (in module pathos.profile), 38
 submit() (Server method), 16

successful() (ApplyResult method), 14
successful() (MapResult method), 14

T

terminate() (ParallelPool method), 23, 26
terminate() (ProcessPool method), 20, 27
terminate() (SerialPool method), 28, 47
terminate() (ThreadPool method), 30, 50
thread_id() (in module pathos.profile), 38
ThreadPool (class in pathos.pools), 28
ThreadPool (class in pathos.threading), 49
ThreadPool() (in module pathos.helpers), 13
tid (ApplyResult attribute), 14
TimeoutError, 16
Tunnel (class in pathos.secure), 40
Tunnel (class in pathos.secure.tunnel), 44
TunnelException, 41, 45

U

uimap() (AbstractWorkerPool method), 10
uimap() (ParallelPool method), 23, 26
uimap() (ProcessPool method), 20, 27
uimap() (ThreadPool method), 30, 50

V

verbose (Pipe attribute), 11
verbose (Tunnel attribute), 41, 45

W

wait() (_Task method), 17
wait() (ApplyResult method), 14
wait() (MapResult method), 14
wait() (Server method), 16
watch() (Selector method), 46

X

XMLRPCRequestHandler (class in pathos.xmlrpc), 51
XMLRPCRequestHandler (class in pathos.xmlrpc.server), 54
XMLRPCServer (class in pathos.xmlrpc), 51
XMLRPCServer (class in pathos.xmlrpc.server), 53