

Parallel Python usage examples

Example #1: sum_primes.py
 Example #2: reverse_md5.py
 Example #3: dynamic_ncpus.py
 Example #4: callback.py
 Example #5: auto_diff.py
 ^

Download all examples Example #1: sum_primes.py ^ ^

```
#!/usr/bin/python
# File: sum_primes.py
# Author: Vitalii Vanovschi
# Desc: This program demonstrates parallel computations with pp module
# It calculates the sum of prime numbers below a given integer in parallel
# Parallel Python Software: http://www.parallelpython.com

import math, sys, time
import pp

def isprime(n):
    """Returns True if n is prime and False otherwise"""
    if not isinstance(n, int):
        raise TypeError("argument passed to is_prime is not of 'int' type")
    if n < 2:
        return False
    if n == 2:
        return True
    max = int(math.ceil(math.sqrt(n)))
    i = 2
    while i <= max:
        if n % i == 0:
            return False
        i += 1
    return True

def sum_primes(n):
    """Calculates sum of all primes below given integer n"""
    return sum([x for x in xrange(2,n) if isprime(x)])

print """Usage: python sum_primes.py [ncpus]
[ncpus] - the number of workers to run in parallel,
if omitted it will be set to the number of processors in the system
"""

# tuple of all parallel python servers to connect with
ppservers = ()
#ppservers = ("10.0.0.1",)

if len(sys.argv) > 1:
    ncpus = int(sys.argv[1])
    # Creates jobserver with ncpus workers
    job_server = pp.Server(ncpus, ppservers=ppservers)
else:
    # Creates jobserver with automatically detected number of workers
    job_server = pp.Server(ppservers=ppservers)

print "Starting pp with", job_server.get_ncpus(), "workers"

# Submit a job of calculating sum_primes(100) for execution.
# sum_primes - the function
# (100,) - tuple with arguments for sum_primes
```

```
# (isprime,) - tuple with functions on which function sum_primes depends
# ("math",) - tuple with module names which must be imported before sum_primes execution
# Execution starts as soon as one of the workers will become available
job1 = job_server.submit(sum_primes, (100,), (isprime,), ("math",))
```

```
# Retrieves the result calculated by job1
# The value of job1() is the same as sum_primes(100)
# If the job has not been finished yet, execution will wait here until result is available
result = job1()
```

```
print "Sum of primes below 100 is", result
```

```
start_time = time.time()
```

```
# The following submits 8 jobs and then retrieves the results
inputs = (100000, 100100, 100200, 100300, 100400, 100500, 100600, 100700)
jobs = [(input, job_server.submit(sum_primes, (input,), (isprime,), ("math",))) for input in inputs]
for input, job in jobs:
    print "Sum of primes below", input, "is", job()
```

```
print "Time elapsed: ", time.time() - start_time, "s"
job_server.print_stats()
```

```
# Parallel Python Software: http://www.parallelpython.com
```

```
Example #2: reverse_md5.py #!/usr/bin/python
# File: reverse_md5.py
# Author: Vitalii Vanovschi
# Desc: This program demonstrates parallel computations with pp module
# It tries to reverse an md5 hash in parallel
# Parallel Python Software: http://www.parallelpython.com
```

```
import math, sys, md5, time
import pp
```

```
def md5test(hash, start, end):
    """Calculates md5 of the integers between 'start' and 'end' and compares it with 'hash'"""
    for x in xrange(start, end):
        if md5.new(str(x)).hexdigest() == hash:
            return x
```

```
print """Usage: python reverse_md5.py [ncpus]
[ncpus] - the number of workers to run in parallel,
if omitted it will be set to the number of processors in the system
"""
```

```
# tuple of all parallel python servers to connect with
ppservers = ()
ppservers = ("10.0.0.1",)
```

```
if len(sys.argv) > 1:
    ncpus = int(sys.argv[1])
    # Creates jobserver with ncpus workers
    job_server = pp.Server(ncpus, ppservers=ppservers)
else:
    # Creates jobserver with automatically detected number of workers
    job_server = pp.Server(ppservers=ppservers)
```

```
print "Starting pp with", job_server.get_ncpus(), "workers"
```

```
#Calculates md5 hash from the given number
hash = md5.new("1829182").hexdigest()
print "hash =", hash
```

```

#Now we will try to find the number with this hash value

start_time = time.time()
start = 1
end = 2000000

# Since jobs are not equal in the execution time, division of the problem
# into a 100 of small subproblems leads to a better load balancing
parts = 128

step = (end - start) / parts + 1
jobs = []

for index in xrange(parts):
    starti = start + index * step
    endi = min(start + (index + 1) * step, end)
    # Submit a job which will test if a number in the range starti-endi has given md5 hash
    # md5test - the function
    # (hash, starti, endi) - tuple with arguments for md5test
    # () - tuple with functions on which function md5test depends
    # ("md5",) - tuple with module names which must be imported before md5test execution
    jobs.append(job_server.submit(md5test, (hash, starti, endi), (), ("md5",)))

# Retrieve results of all submitted jobs
for job in jobs:
    result = job()
    if result:
        break

# Print the results
if result:
    print "Reverse md5 for", hash, "is", result
else:
    print "Reverse md5 for", hash, "has not been found"

print "Time elapsed:", time.time() - start_time, "s"
job_server.print_stats()

# Parallel Python Software: http://www.parallelpython.com
# Example #3: dynamic_ncpus.py

#!/usr/bin/python
# File: dynamic_ncpus.py
# Author: Vitalii Vanovschi
# Desc: This program demonstrates parallel computations with pp module
# and dynamic cpu allocation feature.
# Program calculates the partial sum 1-1/2+1/3-1/4+1/5-1/6+... (in the limit it is ln(2))
# Parallel Python Software: http://www.parallelpython.com

import math, sys, md5, time
import pp

def part_sum(start, end):
    """Calculates partial sum"""
    sum = 0
    for x in xrange(start, end):
        if x % 2 == 0:
            sum -= 1.0 / x
        else:
            sum += 1.0 / x
    return sum

print """Usage: python dynamic_ncpus.py"""
print

```

```

start = 1
end = 20000000

# Divide the task into 64 subtasks
parts = 64
step = (end - start) / parts + 1

# Create jobserver
job_server = pp.Server()

# Execute the same task with different amount of active workers and measure the time
for ncpus in (1, 2, 4, 8, 16, 32):
    job_server.set_ncpus(ncpus)
    jobs = []
    start_time = time.time()
    print "Starting ", job_server.get_ncpus(), " workers"
    for index in xrange(parts):
        start = start + index * step
        end = min(start + (index + 1) * step, end)
        # Submit a job which will calculate partial sum
        # part_sum - the function
        # (start, end) - tuple with arguments for part_sum
        # () - tuple with functions on which function part_sum depends
        # () - tuple with module names which must be imported before part_sum execution
        jobs.append(job_server.submit(part_sum, (start, end)))
    # Retrieve all the results and calculate their sum
    part_sum1 = sum([job() for job in jobs])
    # Print the partial sum
    print "Partial sum is", part_sum1, "| diff =", math.log(2) - part_sum1

    print "Time elapsed:", time.time() - start_time, "s"
    print
job_server.print_stats()

# Parallel Python Software: http://www.parallelpython.com

# Example #4: callback.py

#!/usr/bin/python
# File: callback.py
# Author: Vitalii Vanovschi
# Desc: This program demonstrates parallel computations with pp module
# using callbacks (available since pp 1.3).
# Program calculates the partial sum 1-1/2+1/3-1/4+1/5-1/6+... (in the limit it is ln(2))
# Parallel Python Software: http://www.parallelpython.com

import math, time, thread, sys
import pp

# class for callbacks
class Sum:
    def __init__(self):
        self.value = 0.0
        self.lock = thread.allocate_lock()
        self.count = 0

    # the callback function
    def add(self, value):
        # we must use lock here because += is not atomic
        self.count += 1
        self.lock.acquire()
        self.value += value
        self.lock.release()

```

```

def part_sum(start, end):
    """Calculates partial sum"""
    sum = 0
    for x in xrange(start, end):
        if x % 2 == 0:
            sum -= 1.0 / x
        else:
            sum += 1.0 / x
    return sum

print """Usage: python callback.py [ncpus]
[ncpus] - the number of workers to run in parallel,
if omitted it will be set to the number of processors in the system
"""

start = 1
end = 20000000

# Divide the task into 128 subtasks
parts = 128
step = (end - start) / parts + 1

# tuple of all parallel python servers to connect with
ppservers = ()
#ppservers = ("localhost",)

if len(sys.argv) > 1:
    ncpus = int(sys.argv[1])
    # Creates jobserver with ncpus workers
    job_server = pp.Server(ncpus, ppservers=ppservers)
else:
    # Creates jobserver with automatically detected number of workers
    job_server = pp.Server(ppservers=ppservers)

print "Starting pp with", job_server.get_ncpus(), "workers"

# Create an instance of callback class
sum = Sum()

# Execute the same task with different amount of active workers and measure the time
start_time = time.time()
for index in xrange(parts):
    start = start + index * step
    end = min(start + (index + 1) * step, end)
    # Submit a job which will calculate partial sum
    # part_sum - the function
    # (start, end) - tuple with arguments for part_sum
    # callback=sum.add - callback function
    job_server.submit(part_sum, (start, end), callback=sum.add)
#wait for jobs in all groups to finish
job_server.wait()
# Print the partial sum
print "Partial sum is", sum.value, "| diff =", math.log(2) - sum.value

job_server.print_stats()

# Parallel Python Software: http://www.parallelpython.com

# Example #5: auto_diff.py
# File: auto_diff.py
# Author: Vitalii Vanovschi
# Desc: This program demonstrates parallel computations with pp module
# using class methods as parallel functions (available since pp 1.4).

```

```
# Program calculates the partial sums of  $f(x) = x - x^{2/2} + x^{3/3} - x^{4/4} + \dots$ 
# and first derivatives  $f'(x)$  using automatic differentiation technique.
# In the limit  $f(x) = \ln(x+1)$  and  $f'(x) = 1/(x+1)$ .
# Parallel Python Software: http://www.parallelpython.com
```

```
import math, sys
import pp
```

```
# Partial implementation of automatic differentiation class
```

```
class AD:
```

```
    def __init__(self, x, dx=0.0):
```

```
        self.x = float(x)
```

```
        self.dx = float(dx)
```

```
    def __pow__(self, val):
```

```
        if isinstance(val, int):
```

```
            p = self.x**val
```

```
            return AD(self.x**val, val*self.x**(val-1)*self.dx)
```

```
        else:
```

```
            raise TypeError("Second argument must be an integer")
```

```
    def __add__(self, val):
```

```
        if isinstance(val, AD):
```

```
            return AD(self.x+val.x, self.dx+val.dx)
```

```
        else:
```

```
            return AD(self.x+val, self.dx)
```

```
    def
```

```
    def __radd__(self, val):
```

```
        return self+val
```

```
    def __mul__(self, val):
```

```
        if isinstance(val, AD):
```

```
            return AD(self.x*val.x, self.x*val.dx+val.x*self.dx)
```

```
        else:
```

```
            return AD(self.x*val, val*self.dx)
```

```
    def __rmul__(self, val):
```

```
        return self*val
```

```
    def
```

```
    def __div__(self, val):
```

```
        if isinstance(val, AD):
```

```
            return self*AD(1/val.x, -val.dx/val.x**2)
```

```
        else:
```

```
            return self*(1/float(val))
```

```
    def __rdiv__(self, val):
```

```
        return AD(val)/self
```

```
    def
```

```
    def __sub__(self, val):
```

```
        if isinstance(val, AD):
```

```
            return AD(self.x-val.x, self.dx-val.dx)
```

```
        else:
```

```
            return AD(self.x-val, self.dx)
```

```
    def __repr__(self):
```

```
        return str((self.x, self.dx))
```

```
# This class contains methods which will be executed in parallel
```

```
class PartialSum:
```

```
    def __init__(self, n):
```

```
        self.n = n
```

```
    def
```

```
    #truncated natural logarithm
```

```
    def t_log(self, x):
```

```
        return self.partial_sum(x-1)
```

```

# partial sum for truncated natural logarithm
def partial_sum(self, x):
    return sum([float(i%2 and 1 or -1)*x**i/i for i in xrange(1,self.n)])

print """Usage: python auto_diff.py [ncpus]
[ncpus] - the number of workers to run in parallel,
if omitted it will be set to the number of processors in the system
"""

# tuple of all parallel python servers to connect with
ppservers = ()
#ppservers = ("10.0.0.1",)

if len(sys.argv) > 1:
    ncpus = int(sys.argv[1])
    # Creates jobserver with ncpus workers
    job_server = pp.Server(ncpus, ppservers=ppservers)
else:
    # Creates jobserver with automatically detected number of workers
    job_server = pp.Server(ppservers=ppservers)

print "Starting pp with", job_server.get_ncpus(), "workers"

proc = PartialSum(20000)

results = []
for i in range(32):
    # Creates an object with x = float(i)/32+1 and dx = 1.0
    ad_x = AD(float(i)/32+1, 1.0)
    # Submits a job of calculating proc.t_log(x).
    f = job_server.submit(proc.t_log, (ad_x,))
    results.append((ad_x.x, f))

for x, f in results:
    # Retrieves the result of the calculation
    val = f()
    print "t_log(%f) = %f, t_log'(%f) = %f" % (x, val.x, x, val.dx)

# Print execution statistics
job_server.print_stats()

# Parallel Python Software: http://www.parallelpython.com

```