

# Parallelization – when a single thread is not enough

Valentin Haenel

Strongly based on materials by Eilif Muller and Bastian Venthur

Freelance Consultant and Software Developer

<http://haenel.co>

Nov 2013

Version: 2013-11-fcl13.1 <https://github.com/fcl13/python-parallel-talk>



This work is licensed under the *Creative Commons Attribution-ShareAlike 3.0 License*.

# Outline

- 1 Introduction
  - Why Concurrency?
  - Embarrassingly Parallel Programs
  - Multicore Crisis
- 2 Symetric Multiprocessing (SMP)
- 3 Concurrency with IPython
- 4 MPI for Python with `mpi4py`

# Outline

- 1 Introduction
  - Why Concurrency?
  - Embarrassingly Parallel Programs
  - Multicore Crisis
- 2 Symetric Multiprocessing (SMP)
- 3 Concurrency with IPython
- 4 MPI for Python with `mpi4py`

# What is concurrency

- Parallel computing/processing
- Several computations executing simultaneously
- ... potentially interacting with each other

# The brain is a parallel machine

- Asynchronous
- Distributed memory
- Simple messages
- Agnostic to component failure
- Connectivity:
  - dense local
  - sparse global
- Works with long latencies

# The brain is a parallel machine

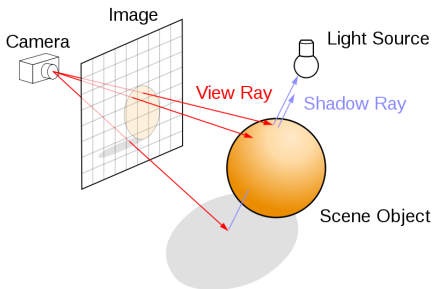
- Asynchronous
- Distributed memory
- Simple messages
- Agnostic to component failure
- Connectivity:
  - dense local
  - sparse global
- Works with long latencies
- ... and works with messy data (big data)

# Outline

- 1 Introduction
  - Why Concurrency?
  - Embarrassingly Parallel Programs
  - Multicore Crisis
- 2 Symetric Multiprocessing (SMP)
- 3 Concurrency with IPython
- 4 MPI for Python with `mpi4py`

# Useful Applications for Concurrency

## Ray Tracing



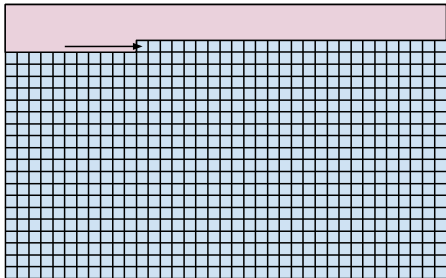
Trace the path from an imaginary eye (camera) through each pixel in a screen and calculate the color of the object(s) visible through it.



# Useful Applications for Concurrency

## Ray Tracing

Serial Execution: 1h

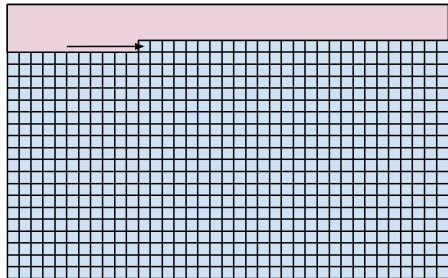


**Figure:** Ray Tracing performed by one task.

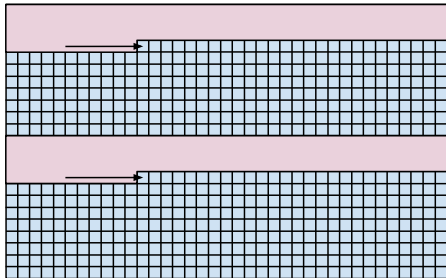
# Useful Applications for Concurrency

## Ray Tracing

Serial Execution: 1h



Parallel Execution: 0.5h



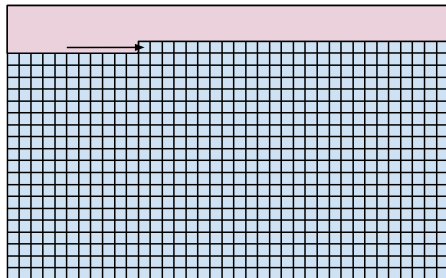
**Figure:** Ray Tracing performed by one task.

**Figure:** Ray Tracing performed by two tasks.

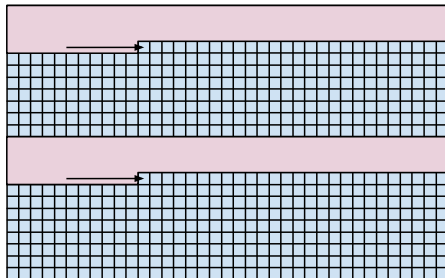
# Useful Applications for Concurrency

## Ray Tracing

Serial Execution: 1h



Parallel Execution: 0.5h



**Figure:** Ray Tracing performed by one task.

**Figure:** Ray Tracing performed by two tasks.

Ray Tracing is **embarrassingly parallel**:

- Little or no effort to separate the problem into parallel tasks
- No dependencies or communication between the tasks

# Outline

- 1 Introduction
  - Why Concurrency?
  - Embarrassingly Parallel Programs
  - Multicore Crisis
- 2 Symetric Multiprocessing (SMP)
- 3 Concurrency with IPython
- 4 MPI for Python with `mpi4py`

# The free lunch is over

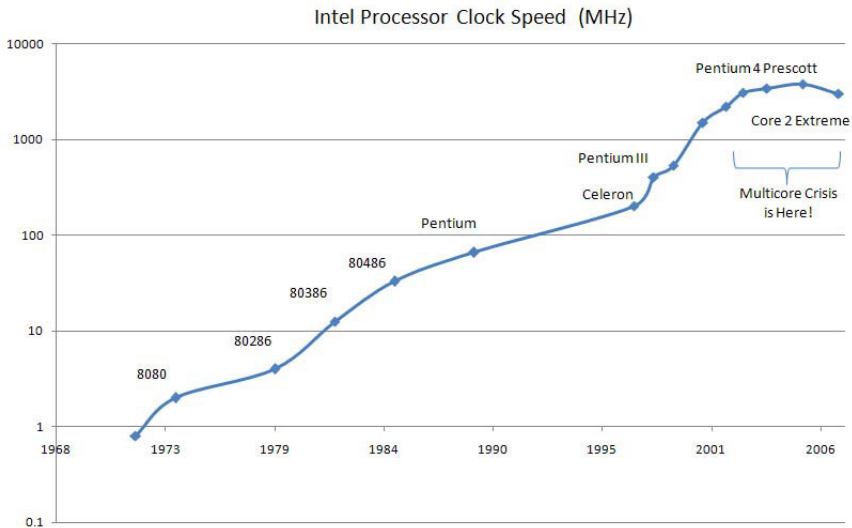
*"Concurrency is the next major revolution in how we write software [after OOP]."*

Herb Sutter, *The Free Lunch is Over: A Fundamental Turn Towards Concurrency in Software* Dr.Dobb's Journal, 30(3) March 2005.

# Multicore Crisis

- 1970-2005
  - CPUs became quicker and quicker every year
  - Moore's Law: The number of transistors [...] doubles
  - approximately every two years.
- But!
  - Physical limits:
    - Miniaturization at atomic levels
    - energy consumption
    - heat produced by CPUs, etc.
  - Stagnation in CPU clock rates since 2005
- Since 2005
  - Chip producers aimed for more cores instead of higher clock rates.
  - ⇒ Multicore crisis
  - FLOPS has been replaced by FLOPW
  - (esp. in High Performance Computing(HPC))

# Multicore Crisis

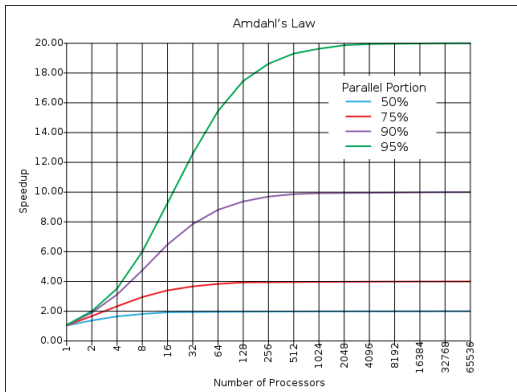


## Limitations: Amdahls Law

*The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.*



# Limitations: Amdahls Law



*if a program needs 20 hours using a single processor core, and a particular portion of the program which takes one hour to execute cannot be parallelized, while the remaining 19 hours (95) of execution time can be parallelized, then regardless of how many processors are devoted to a parallelized execution of this program, the minimum execution time cannot be less than that critical one hour. Hence the speedup is limited up to  $20\times$ .*

# Don't Panic

- Writing parallel programs is easy!
- Small and simple APIs
- Designing parallel algorithms can be **easy** or **hard**
- **Easy**: embarrassingly parallel
- **Hard**: to find the parallelism

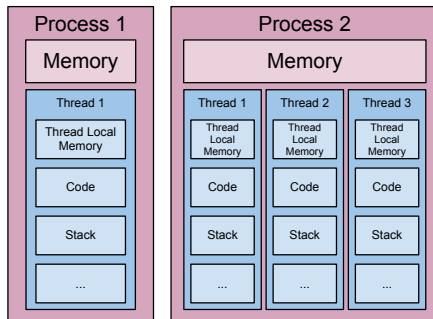
# Don't Panic

- Scientific parallel programs are easy!
- **Parallelism**: Number crunching over large datasets
- **Prior-art**: Many algorithms already exist
- **Hardware**: HPC is traditionally academic

# Outline

- 1 Introduction
  - Why Concurrency?
  - Embarrassingly Parallel Programs
  - Multicore Crisis
- 2 Symetric Multiprocessing (SMP)
- 3 Concurrency with IPython
- 4 MPI for Python with `mpi4py`

# Two Kinds of Tasks: Threads and Processes



- A process has **one or more** threads
- Processes have their **own** memory (Variables, etc.)
- Threads share the memory of the process they belong to
- Threads are also called **lightweight** processes:
  - They spawn faster than processes
  - Context switches (if necessary) are faster

# Communication between Tasks

## Shared Memory and Message Passing

Basically you have two paradigms:

### ① Shared Memory

- Tasks A and B share some memory
- Whenever a task modifies a variable in the shared memory, the other task(s) see that change immediately

### ② Message Passing

- Task A sends a message to Task B
- Task B receives the message and does something with it

The former paradigm is usually used with threads and the latter one with processes (more on that later).

# Symmetric Multiprocessing (SMP)

- Homogeneous Multi-core and/or cpu
- Physically shared memory
- e.g. 8-way 6-core Opteron = 48 cores
- Most of you have a multicore CPU in your laptop

# CPython and the Global Interpreter Lock (GIL)

- CPython suffers from the *Gil Problem*
- The interpreter has a global state storage
- Not thread-safe!
  
- Can have threads
- ... but only one can run at a time
- No real, concurrent threads in CPython



# CPython and the Global Interpreter Lock (GIL)

- CPython suffers from the *Gil Problem*
- The interpreter has a global state storage
- Not thread-safe!
  
- Can have threads
- ... but only one can run at a time
- No real, concurrent threads in CPython
  
- Jython and IronPython do not suffer from this limitation

# SMP in Python

- Standard as of Python 2.6

```
>>> import multiprocessing
```

- Like threads, but in separate processes
- Avoids GIL but higher process creation cost
- Package exists for 2.5

# Race

- When unpredictable order of completion affects output
  - Hardware latency
  - Unpredictable algorithm run-time
- Difficult to debug because problematic case maybe infrequent

# Race: the setup

```
import multiprocessing as mp
import time
import random

def mult(arg):
    time.sleep(random.uniform(1,5))
    arg.value *= 10

def add(arg):
    time.sleep(random.uniform(1,5))
    arg.value += 10

arg = mp.Value('d', 0.0)  # synchronized shared object, type double

p1 = mp.Process(target=mult, args=(arg,))
p2 = mp.Process(target=add, args=(arg,))
p1.start(); p2.start(); p1.join(); p2.join()
print arg.value
```

# Race: in action

```
$ python ex_smp_race.py
```

```
10.0
```

```
$ python ex_smp_race.py
```

```
100.0
```

```
$ python ex_smp_race.py
```

```
10.0
```

# The apparent solution

- Locks can be a solution to enforce atomicity:

```
l = Lock()  
l.acquire()  
# <code>  
l.release()
```

- However, Locks are source of deadlocks

# Deadlock: the setup

```
import multiprocessing as mp

def compute(arg, lock):
    lock.acquire()
    arg.value += 10

arg = mp.Value('d', 0.0)  # synchronized shared object, type double
lock = mp.Lock()          # non-recursive lock object

p1 = mp.Process(target=compute, args=(arg, lock))
p2 = mp.Process(target=compute, args=(arg, lock))
p1.start()
p2.start()
p1.join()
```

# Deadlock: in action

```
>>> %run ex_smp.py
# p2 is still waiting for release (deadlock)
>>> p2.is_alive()
True
>>> arg.value()
10.0
# resolve the deadlock, without killing processes
>>> lock.release()
>>> p2.is_alive()
False
>>> p2.join()
>>> num.value
20
```

See also: Dining Philosophers



# Shared Memory: Numpy and Multiprocessing

```
import ctypes
from multiprocessing import sharedctypes, Process
import numpy
from numpy import ctypeslib

def f(cta):
    from numpy import ctypeslib
    npa = ctypeslib.as_array(cta._obj)
    npa[0] = npa.sum()

cta = sharedctypes.Array(ctypes.c_double, numpy.arange(1e6))
npa = ctypeslib.as_array(cta._obj)
p1 = Process(target=f, args=(cta,))
```

- cta is a synchronized shared memory buffer
- npa is a numpy array that shares memory with cta

# Shared Memory: Numpy and Multiprocessing

```
>>> %run ex_smp_numpy.py
>>> npa
array([ 0.00000000e+00,  1.00000000e+00,  2.00000000e+00, ...,
        9.99997000e+05,  9.99998000e+05,  9.99999000e+05])
>>> p1.start(); p1.join()
In [6]: npa
array([ 4.99999500e+11,  1.00000000e+00,  2.00000000e+00, ...,
        9.99997000e+05,  9.99998000e+05,  9.99999000e+05])
```

# Embarrassingly Parallel - Multiprocessing

- Imagine the following function that does work

```
def f(x):  
    import time  
    import random  
    time.sleep(random.uniform(0.1, 0.2))  
    return x*x  
  
nums = range(10)
```

# The Pool class and map function

- Python provides a builtin `map` function
- The `Pool` class comes in handy for embarrassingly parallel problems
- Provides a `map` function across worker processes
- `import` statements need to be in the function
- If you need one with a progressbar: `embparpar`

# The Pool class and map function

```
>>> nums = range(10)
>>> %timeit map(f, nums)
1 loops, best of 3: 1.37 s per loop
>>> pool1 = Pool(1)
>>> %timeit pool1.map(f, nums)
1 loops, best of 3: 1.27 s per loop
>>> pool6 = Pool(6)
>>> %timeit pool6.map(f, nums)
1 loops, best of 3: 281 ms per loop
>>> pool10 = Pool(10)
>>> %timeit pool10.map(f, nums)
10 loops, best of 3: 194 ms per loop
>>> pool16 = Pool(16)
>>> %timeit pool16.map(f, nums)
10 loops, best of 3: 186 ms per loop
```

# Outline

- 1 Introduction
  - Why Concurrency?
  - Embarrassingly Parallel Programs
  - Multicore Crisis
- 2 Symetric Multiprocessing (SMP)
- 3 Concurrency with IPython
- 4 MPI for Python with `mpi4py`

# Starting IPython engines

- To launch a small (or large) cluster of IPython engines

```
$ ipcluster start -n 24
```

```
2013-02-05 16:35:56.367 [IPClusterStart]
```

```
    Using existing profile dir: u'/home/val/.ipython/profile_default'
```

```
2013-02-05 16:35:56.372 [IPClusterStart]
```

```
    Starting ipcluster with [daemon=False]
```

```
2013-02-05 16:35:56.373 [IPClusterStart]
```

```
    Creating pid file: /home/val/.ipython/profile_default/pid/ipcluster.
```

```
2013-02-05 16:35:56.376 [IPClusterStart]
```

```
    Starting Controller with LocalControllerLauncher
```

```
2013-02-05 16:35:57.372 [IPClusterStart]
```

```
    Starting 24 Engines with LocalEngineSetLauncher
```

```
2013-02-05 16:36:29.888 [IPClusterStart]
```

```
    Engines appear to have started successfully
```

# Command them from IPython

- Use the `Client` class to interface with the engines

```
>>> from IPython.parallel import Client
>>> client = Client()
>>> len(client.ids)
24
```



# Command them from IPython

- Easy example: using an embarrassingly parallel map
- Obtain a DirectView using slicing on the client

```
>>> dview = client[:]  
>>> type(dview)  
IPython.parallel.client.view.DirectView  
>>> %timeit dview.map_sync(f, range(10)) # using the f from before  
1 loops, best of 3: 187 ms per loop
```

# IPython engines have local namespaces

- DirectView object provides dictionary access to engine namespaces

```
>>> dview.execute('x=1')
>>> dview['x']
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>> dview8 = client[8]
>>> dview8.execute('x=23')
>>> dview['x']
[1, 1, 1, 1, 1, 1, 1, 1, 23, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

# IPython engines have local namespaces

- Using apply in asynchronous and synchronous mode

```
>>> result = dview.apply(lambda y: x+y, 19)
>>> type(result)
IPython.parallel.client.asyncresult.AsyncResult
>>> result.get()[7:12]
[20, 42, 20, 20, 20]
>>> dview.block = True
>>> dview.apply(lambda y: x+y, 19)
...
```

# Partition and Collection data

- Send using scatter

```
>>> dview4 = client[:4]
>>> dview4.block = True
>>> dview4.scatter('a', 'Hello World!')
['Hel', 'lo ', 'Wor', 'ld!']
```

- Now operate on the data using execute

```
>>> dview4.execute('a = a.upper()', targets=[0, 1])
>>> dview4['a']
['HEL', 'LO ', 'Wor', 'ld!']
```

- Collect using gather

```
>>> "".join(dview4.gather('a'))
'HELLO World!'
```

# Warnings

- Engines are not reset after IPython restart

```
>>> from IPython.parallel import Client
>>> client = Client()
>>> dview4 = client[:4]
>>> dview4['a']
['HEL', 'LO ', 'Wor', 'ld!']
```

- Errors propagate as usual

```
>>> dview8['b']
```

```
-----
NameError                                Traceback (most recent call last)
/home/val/anaconda/lib/python2.7/site-packages/IPython/parallel/util.py in
    249         else:
    250             if not user_ns.has_key(keys):
--> 251                 raise NameError("name '%s' is not defined"%keys)
    252             return user_ns.get(keys)
    253
NameError: name 'b' is not defined
```

# Pros and Cons

## Pros

- Interactive
- Plays nicely with MPI
- Re-connectible
- Debugging output from engines
- Can be run over other batch queueing systems
  - (e.g. Sun Grid Engine)

## Cons

- Slow for large messages
- No shared memory

# Outline

- 1 Introduction
  - Why Concurrency?
  - Embarrassingly Parallel Programs
  - Multicore Crisis
- 2 Symetric Multiprocessing (SMP)
- 3 Concurrency with IPython
- 4 MPI for Python with `mpi4py`

# Scalable Message Passing Concurrency

- MPI - the Message Passing Interface
- Multi-process execution facilities

\$ `mpiexec -n 16 python helloworld.py`

- API for inter-process message exchanges
  - eg. Basic P2P: Send (emitter), Recv (consumer)
- Very common on large HPC installations



# What is MPI for Python? (mpi4py)

- A wrapper for widely used MPI (MPICH2, OpenMPI, LAM/MPI)
- MPI supported by wide range of vendors, hardware, languages
- API based on the standard MPI-2 C++ bindings.
- Almost all MPI calls are supported.
  - targeted to MPI-2 implementations.
  - also works with MPI-1 implementations.

# Basic stuff

```
from mpi4py import MPI  
comm = MPI.COMM_WORLD
```

- Communicator = comm
  - Manages processes and communication between them
- MPI.COMM\_WORLD
  - all processes defined at execution time
- Comm.size – total number of available processes
- Comm.rank – ID of the current process

# Basic stuff

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
print "Hello from %s, %d of %d"\
      % (MPI.Get_processor_name(),
         comm.rank, comm.size)
```

```
$ mpiexec -n 4 python ex_mpi.py
```

```
Hello from teik, 1 of 4
```

```
Hello from teik, 3 of 4
```

```
Hello from teik, 2 of 4
```

```
Hello from teik, 0 of 4
```

- Process needs to check which rank it is

# What can this look like in practice

```
Hello, World! I am process 232 of 512 on Rank 232 of 512 <2,2,3,0> R00-M0-N09-J18.
Hello, World! I am process 133 of 512 on Rank 133 of 512 <1,0,2,1> R00-M0-N09-J04.
Hello, World! I am process 208 of 512 on Rank 208 of 512 <0,1,3,0> R00-M0-N09-J26.
Hello, World! I am process 145 of 512 on Rank 145 of 512 <0,1,2,1> R00-M0-N09-J22.
Hello, World! I am process 224 of 512 on Rank 224 of 512 <0,2,3,0> R00-M0-N09-J25.
Hello, World! I am process 215 of 512 on Rank 215 of 512 <1,1,3,3> R00-M0-N09-J09.
Hello, World! I am process 225 of 512 on Rank 225 of 512 <0,2,3,1> R00-M0-N09-J25.
Hello, World! I am process 148 of 512 on Rank 148 of 512 <1,1,2,0> R00-M0-N09-J05.
Hello, World! I am process 218 of 512 on Rank 218 of 512 <2,1,3,2> R00-M0-N09-J17.
Hello, World! I am process 253 of 512 on Rank 253 of 512 <3,3,3,1> R00-M0-N09-J32.
Hello, World! I am process 212 of 512 on Rank 212 of 512 <1,1,3,0> R00-M0-N09-J09.
Hello, World! I am process 249 of 512 on Rank 249 of 512 <2,3,3,1> R00-M0-N09-J19.
Hello, World! I am process 132 of 512 on Rank 132 of 512 <1,0,2,0> R00-M0-N09-J04.
Hello, World! I am process 130 of 512 on Rank 130 of 512 <0,0,2,2> R00-M0-N09-J23.
Hello, World! I am process 150 of 512 on Rank 150 of 512 <1,1,2,2> R00-M0-N09-J05.
Hello, World! I am process 149 of 512 on Rank 149 of 512 <1,1,2,1> R00-M0-N09-J05.
Hello, World! I am process 184 of 512 on Rank 184 of 512 <2,3,2,0> R00-M0-N09-J15.
Hello, World! I am process 159 of 512 on Rank 159 of 512 <3,1,2,3> R00-M0-N09-J30.
Hello, World! I am process 211 of 512 on Rank 211 of 512 <0,1,3,3> R00-M0-N09-J26.
Hello, World! I am process 163 of 512 on Rank 163 of 512 <0,2,2,3> R00-M0-N09-J21.
Hello, World! I am process 142 of 512 on Rank 142 of 512 <3,0,2,2> R00-M0-N09-J31.
Hello, World! I am process 178 of 512 on Rank 178 of 512 <0,3,2,2> R00-M0-N09-J20.
Hello, World! I am process 136 of 512 on Rank 136 of 512 <2,0,2,0> R00-M0-N09-J12.
Hello, World! I am process 193 of 512 on Rank 193 of 512 <0,0,3,1> R00-M0-N09-J27.
Hello, World! I am process 190 of 512 on Rank 190 of 512 <3,3,2,2> R00-M0-N09-J28.
Hello, World! I am process 204 of 512 on Rank 204 of 512 <3,0,3,0> R00-M0-N09-J35.
Hello, World! I am process 216 of 512 on Rank 216 of 512 <2,1,3,0> R00-M0-N09-J17.
Hello, World! I am process 185 of 512 on Rank 185 of 512 <2,3,2,1> R00-M0-N09-J15.
```

# Point-to-Point (P2P): Python objects

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
```

- The tag information allows selectivity of messages at the receiving end.

## P2P: (NumPy) array data

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = numpy.arange(23, dtype='i')
    comm.Send([data, MPI.INT], dest=1, tag=77)
elif rank == 1:
    data = numpy.empty(23, dtype='i')
    comm.Recv([data, MPI.INT], source=0, tag=77)
```

- Array data buffer notation: [`<BUFFER>`, `MPI.<DATATYPE>`]
- Can also infer the correct datatype from the numpy array
- MPI will write into the Numpy array

# Collective Messages

Involve the whole COMM

- **Scatter**
  - Spread a sequence over processes
- **Gather**
  - Collect a sequence scattered over processes
- **Broadcast**
  - Send a message to all processes
- **Barrier**
  - Block till all processes arrive

# Scatter and Gather

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

N = 10
# rank 0 should send the message
if comm.rank == 0:
    msg = numpy.arange(N, dtype=int)
else:
    msg = None

# dest should hold the scattered message
dest = numpy.empty(N/comm.size, dtype=int)
# ans should hold the gathered result
ans = numpy.empty(comm.size, dtype=int)
```



# Scatter and Gather

```
# scatter the array, from rank 0  
comm.Scatter([msg, MPI.INT],  
             [dest, MPI.INT], root=0)
```

```
# do the work  
mysum = numpy.sum(dest)
```

```
# gather the result, into rank 0  
comm.Gather([mysum, MPI.INT],  
           [ans, MPI.INT], root=0)
```

# Scatter and Gather

```
if comm.rank == 0:  
    print "Answer from rank 0: %s" % str(ans)  
else:  
    print "Answer from rank %d: %s" % (comm.rank, str(ans))
```

# What is happening here

- Imagine `comm.size == 2`
- `len(dest)` is 5 everywhere
- `msg` is `array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])`
- `dest` on 0 becomes `array([0, 1, 2, 3, 4])`
- `dest` on 1 becomes `array([5, 6, 7, 8, 9])`

# What is happening here

- Imagine `comm.size == 2`
- `len(dest)` is 5 everywhere
- `msg` is `array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])`
- `dest` on 0 becomes `array([0, 1, 2, 3, 4])`
- `dest` on 1 becomes `array([5, 6, 7, 8, 9])`
- What is the `len` of `ans`?

# What is happening here

- Imagine `comm.size == 2`
- `len(dest)` is 5 everywhere
- `msg` is `array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])`
- `dest` on 0 becomes `array([0, 1, 2, 3, 4])`
- `dest` on 1 becomes `array([5, 6, 7, 8, 9])`
  
- What is the `len` of `ans`?
- 2 is correct

# What is happening here

- Imagine `comm.size == 2`
- `len(dest)` is 5 everywhere
- `msg` is `array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])`
- `dest` on 0 becomes `array([0, 1, 2, 3, 4])`
- `dest` on 1 becomes `array([5, 6, 7, 8, 9])`
  
- What is the `len` of `ans`?
- 2 is correct
- What is the final value of `ans`?

# What is happening here

- `Imagine comm.size == 2`
- `len(dest)` is 5 everywhere
- `msg` is `array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])`
- `dest` on 0 becomes `array([0, 1, 2, 3, 4])`
- `dest` on 1 becomes `array([5, 6, 7, 8, 9])`
  
- What is the `len` of `ans`?
- 2 is correct
- What is the final value of `ans`?
- `[10, 35]` is correct

# What is happening here

```
$ mpiexec -n 2 python ex_mpi_scatter_gather.py
```

```
Answer from rank 1: [139713299173080 139713299173080]
```

```
Answer from rank 0: [10 35]
```



# The import problem

- Importing Python module from each MPI process may cause problems on big installations
- `MPI_import` provides specialised import statements that use MPI under the hood e.g. `mpi_import`

```
from MPI_Import import mpi_import, MPI
with mpi_import():
    import os
    import re
    import math
    import decimal
    import optparse
    import platform
    import string
    import zipfile
    import numpy

comm = MPI.COMM_WORLD
rank, size = comm.Get_rank(), comm.Get_size()
print "Hello World! I am rank %d of size %d" % (rank, size)
```

# Implementation

- Implemented with **Cython**
- Code base far easier to write, maintain, and extend.
- Faster than other solutions (mixed Python and C codes).
- A **pythonic** API that runs at C speed !

- Tested on all major platforms (Linux, Mac OS X, Windows).
- Works with the open-source MPI's (MPICH2, Open MPI, MPICH1, LAM).
- Should work with vendor-provided MPI's (HP, IBM, SGI).
- Works on Python 2.3 to 3.0 (Cython is just great!).

# Interoperability

- Good support for wrapping other MPI-based codes.
  - You can use Cython (`cimport` statement).
  - You can use boost.
  - You can use SWIG (*typemaps* provided).
  - You can use F2Py (`"py2f()"/"f2py()"` methods).
  - You can use hand-written C (C-API provided).
- `mpi4py` will allow you to use virtually **any** MPI based C/C++/Fortran code from Python.

# Features Summary

- Classical MPI-1 Point-to-Point.
  - blocking (`send/recv`)
  - non-blocking (`isend/irecv`, `test/wait`).
- Classical MPI-1 and Extended MPI-2 Collectives.
- Communication of general Python objects (pickle).
  - very convenient, as general as pickle can be.
  - can be slow for large data (CPU and memory consuming).
- Communication of array data (Python's buffer interface).
  - MPI datatypes have to be explicitly specified.
  - Very fast, almost C speed (for messages above 5-10 kB).