Concurrency Kung-fu: Python Style

Eilif Muller

Blue Brain Project

neural ensemble .org

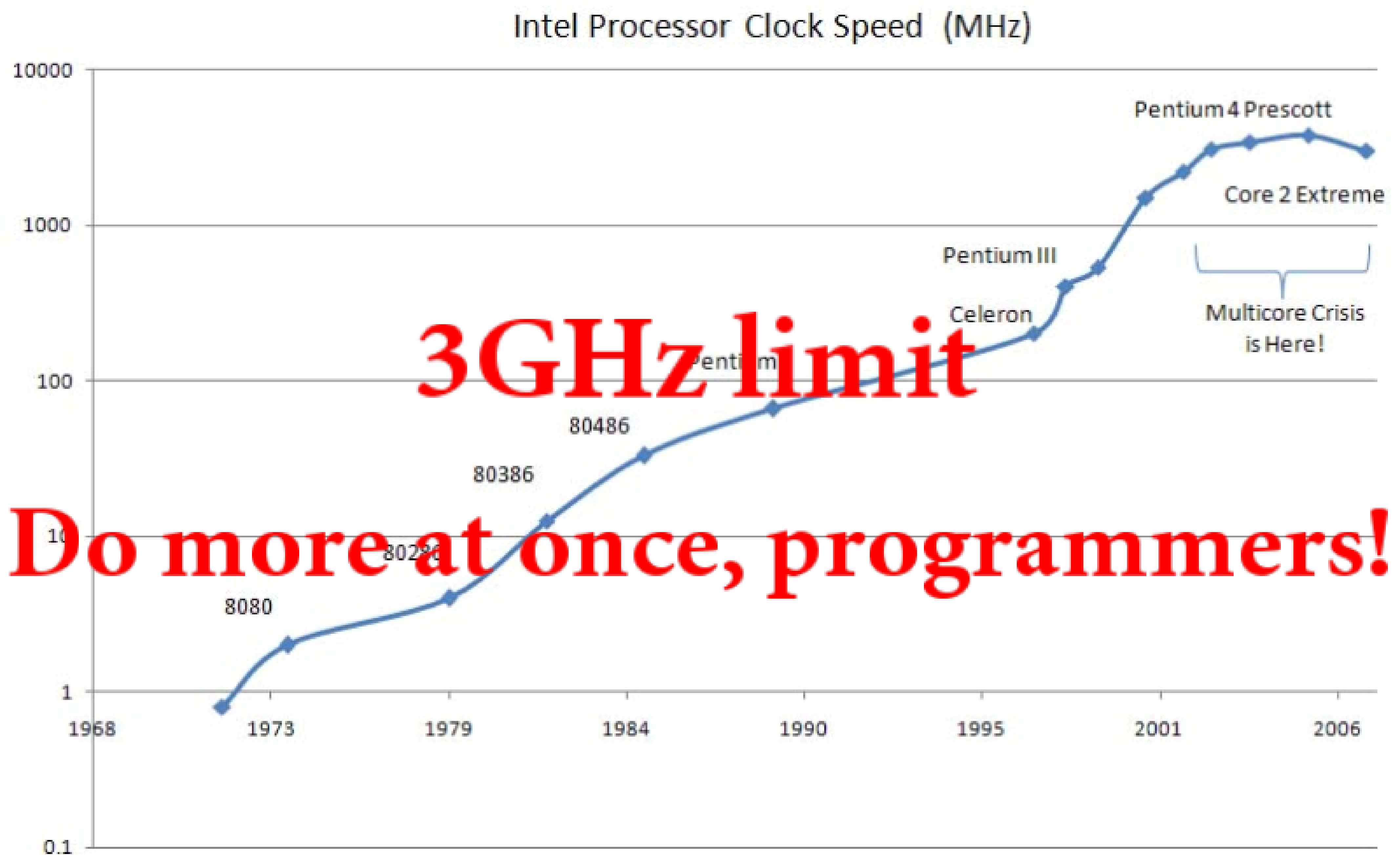
Why Concurrency?
(parallelism)

The brain is a parallel machine

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

000188

Why Concurrency?
(parallelism)


Intel Processor Clock Speed (MHz)
10000
1000
100
1
0.1
1968
1973
1979
1984
1990
1995
2001
2006
8080
80386
80486
Celeron
Pentium III
Pentium 4 Prescott
Core 2 Extreme
Multicore Crisis is Here!
3GHz limit
Do more at once, programmers!

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.

Don't Panic!

- Writing parallel programs is easy!
  - Small and simple APIs
- Designing parallel algorithms can be easy or hard.
  - Easy: "embarassingly parallel"
  - Hard: to find the parallelism
  - Hardware: e.g. the starving CPU

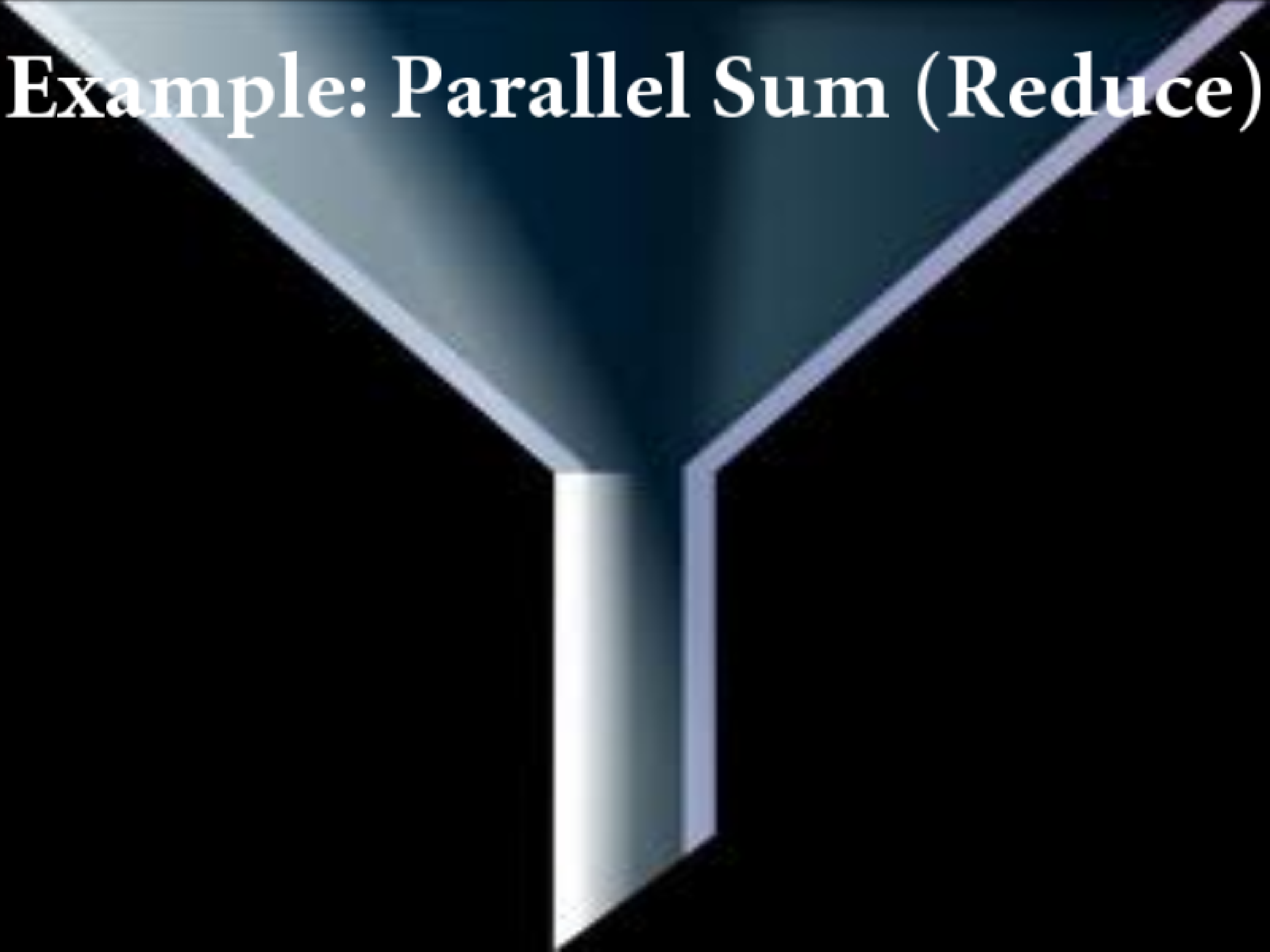
Don't Panic!

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

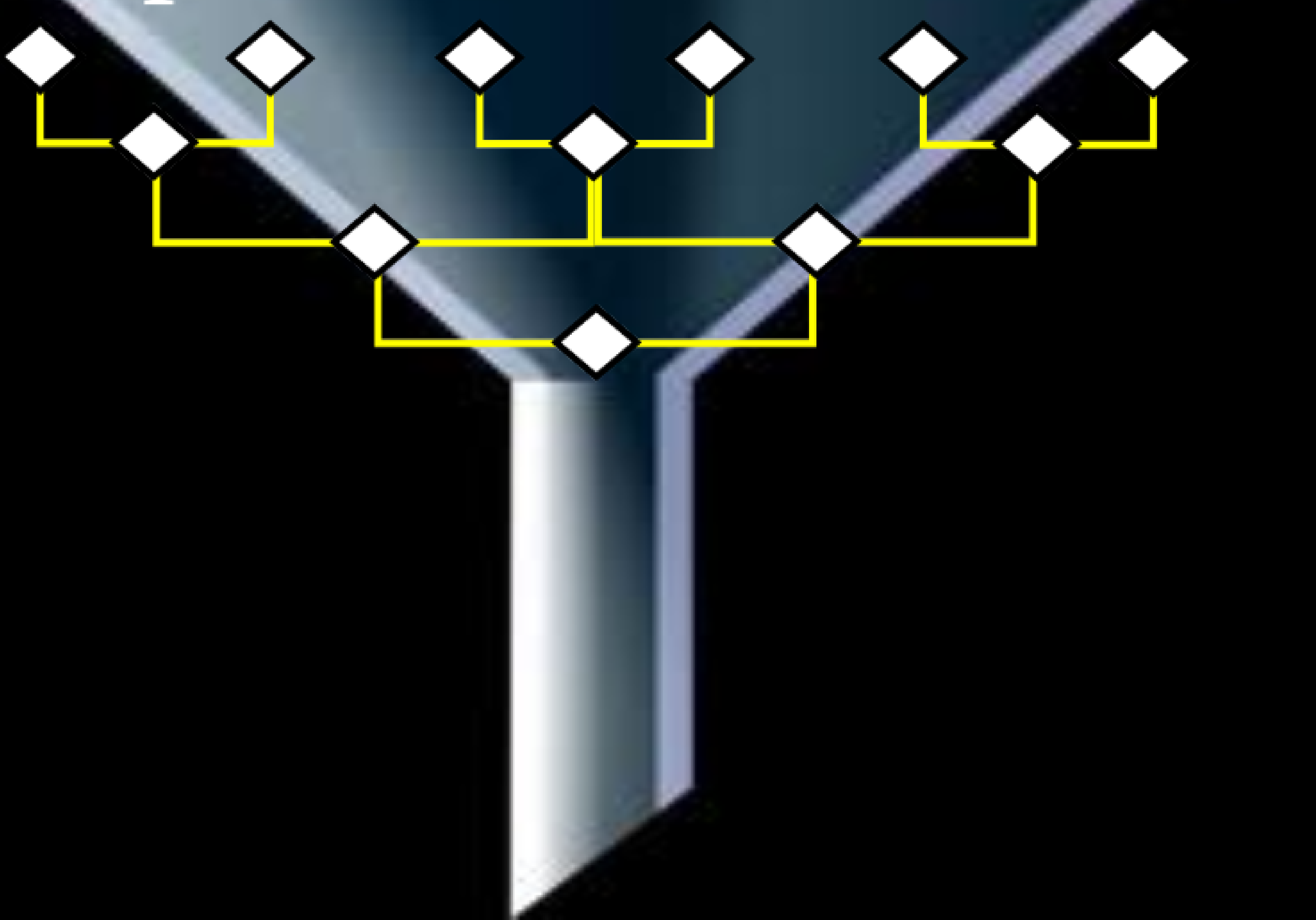
Three Objectives

- How to run parallel programs
- APIs: ipython, mpi4py, multiprocessing
  - ~8 functions each
- Think parallel
  - You already know how!

Example: Parallel Sum (Reduce)



Example: Parallel Sum (Reduce)



Example: Parallel Sum (Reduce)

Initially highly parallel

Becomes serial

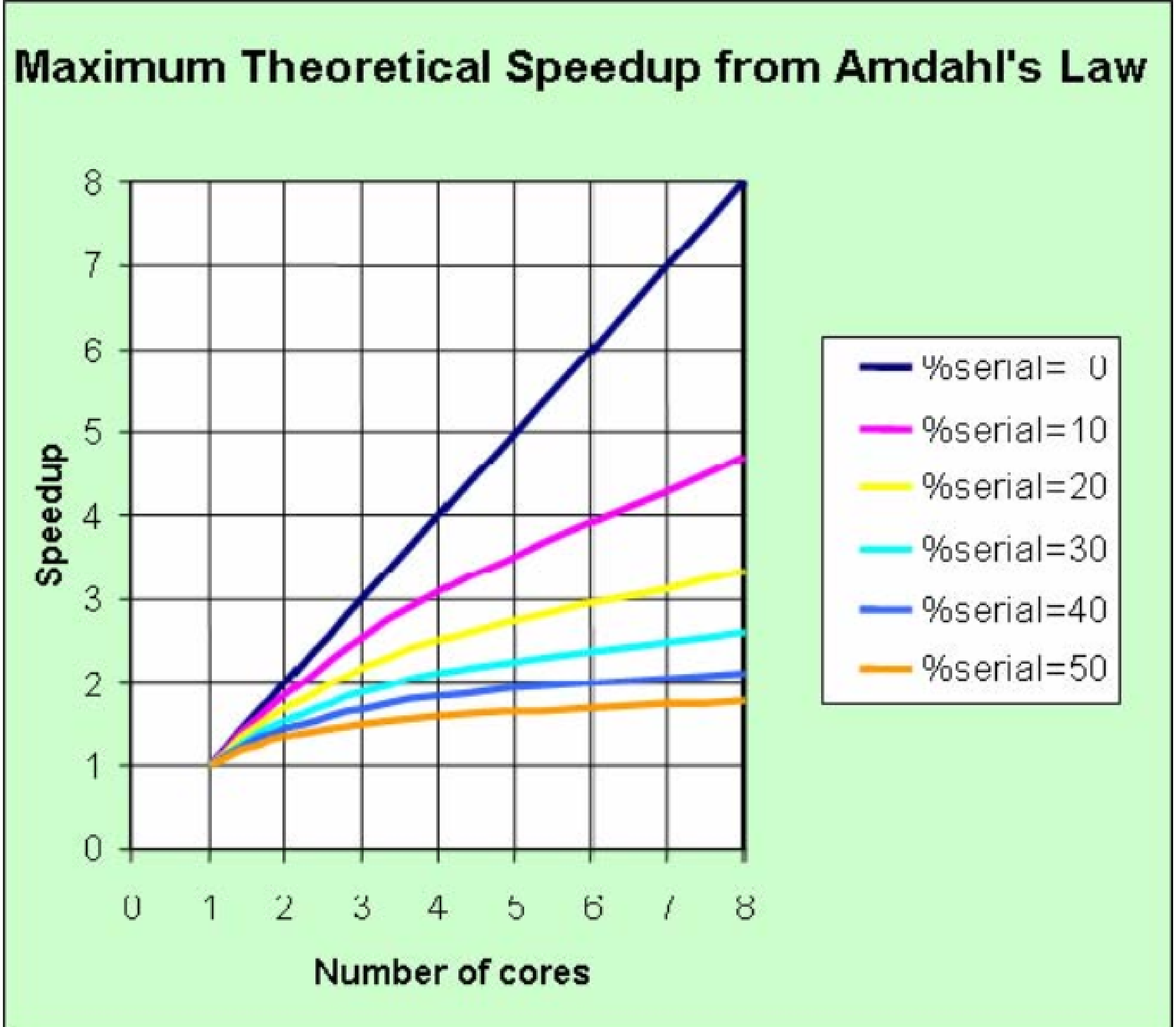
Computation is O(log(N))
or 2^q = N

As is communication ...

and t(op) << t(msg)

Glossary

- Load balancing
  - Divide work so all threads finish on time
- Speedup = T_serial/T_parallel(n_threads)
- Scalability
  - Does speedup continue to improve with increasing n_threads?


Maximum Theoretical Speedup from Amdahl's Law
Speedup
0
1
2
3
4
5
6
7
8
Number of cores
%serial= 0
%serial=10
%serial=20
%serial=30
%serial=40
%serial=50


Part 1:
Easy Concurrency
with IPython

Start IPython“slaves” (version>=0.12.1)

```
$ ipcluster start -n 2
```

Command them in IPython:

```
$ ipython -pylab
```

```
> from IPython.parallel import Client
> rc = Client()
> mec = rc[:]
> mec.block=True
> mec.targets
[0,1,2,3]
```

Start IPython "slaves" (version<=0.10.1)

```
$ ipcluster local
```

Command them in IPython:

```
$ ipython -pylab
```

```
> from IPython.kernel import client
> mec = client.MultiEngineClient()
> mec.get_ids()
[0,1,2,3]
```

Slaves have local namespaces

```
> exe = mec.execute
> exe("x = 10")
> x = 5
> mec['x']
[10,10,10,10]
```

Embarrassingly Parallel

```
> from scipy import factorial
> mec.map(factorial, range(4))
[1.0,1.0,2.0,6.0]
```

Scatter

```
> mec.scatter("a",'hello world')
> mec['a']
['hel', 'lo ', 'wor', 'ld']
> mec.execute("a = a.upper()",
                targets=[2,3])
```

Gather

```
> ''.join(mec.gather("a"))
'hello WORLD'
> <crtl-D> # quit
```

Reconnect

Engines are NOT reset

```
> from IPython.parallel import Client
> rc = Client()
> mec = rc[:]
> mec.block=True
> mec['x']
[10,10,10,10]
```

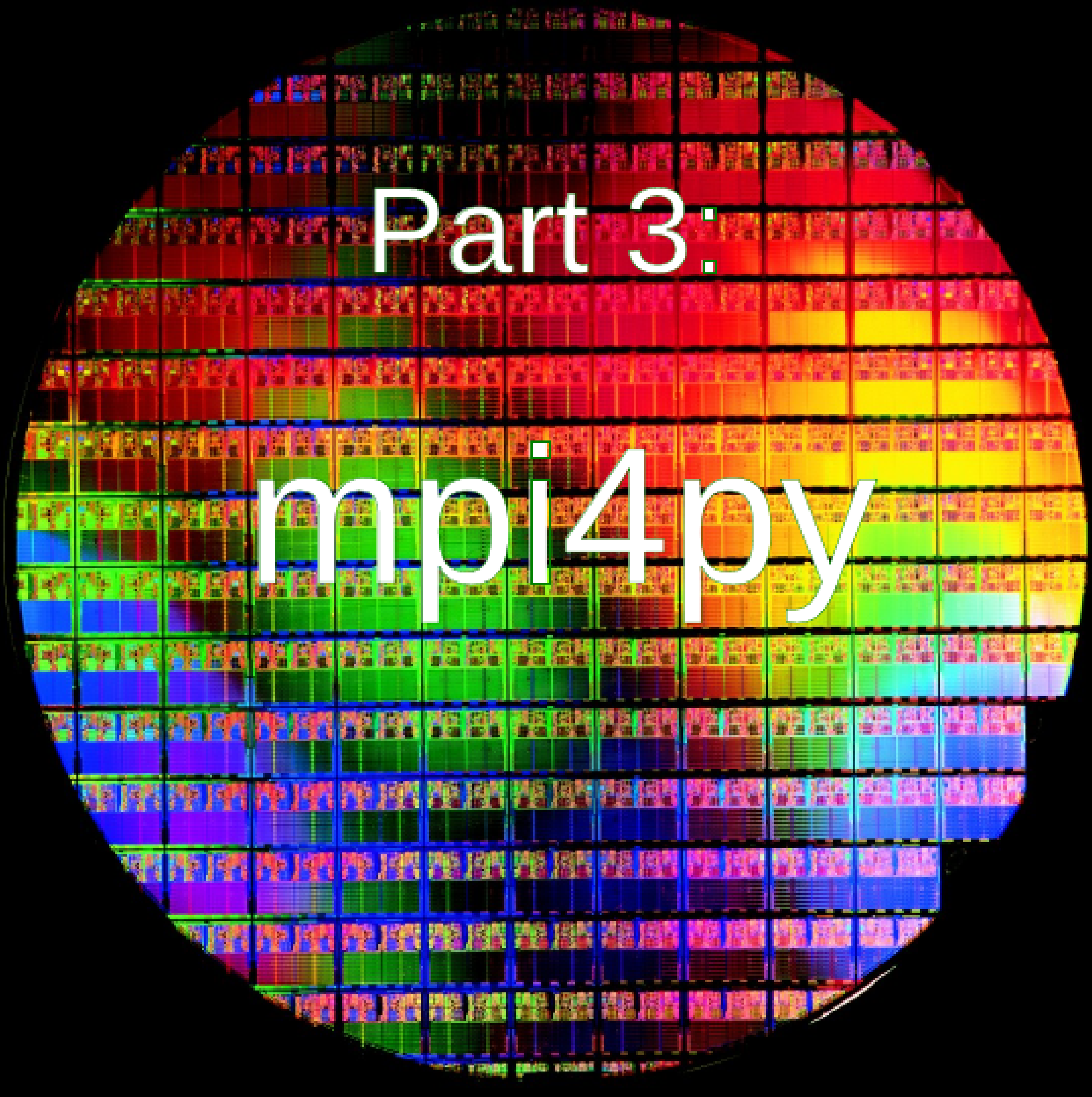
Pros and Cons

- Interactive
- Plays with MPI
- Re-connectible
- Debugging output from slaves

- Slow for large messages
- 2 Step execution
- No shared memory
- Inter-slave: MPI

Part 3:

mpi4py



Scalable Message Passing Concurrency

- Multi-process execution facilities

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

- API for inter-process message exchanges
  - eg. Basic P2P: Send (emitter), Recv (consumer)

What is MPI for Python?

- 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 (recall: kung-fu group)
  - all processes defined at exec.time
- Comm.size, Comm.rank

Basic stuff (cont.)

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

→ test.py

```
$ mpiexec -n 2 python test.py
Hello from rucola, 0 of 2
Hello from rucola, 1 of 2
```

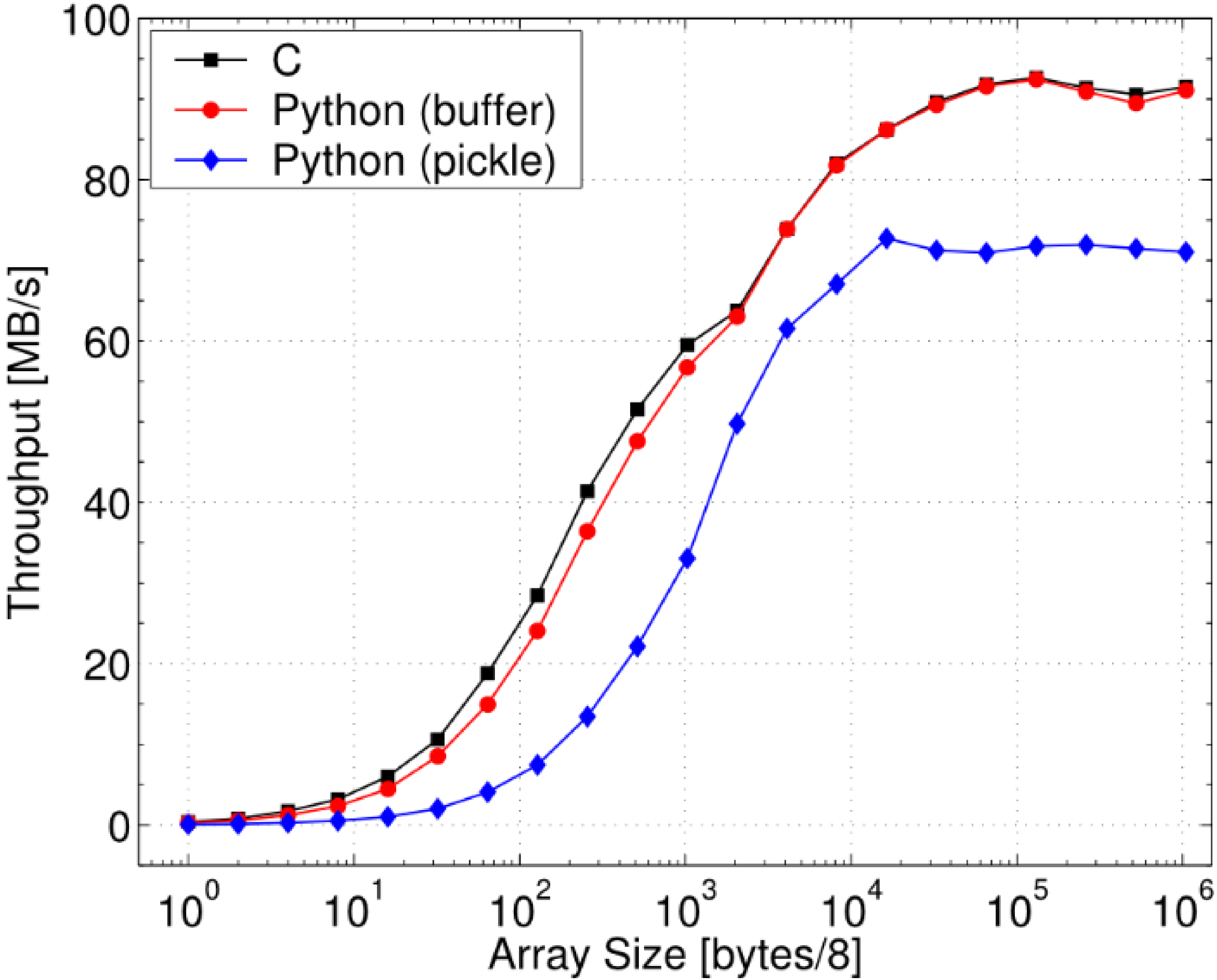
```
Hello, World! I am process 232 of 512 on Rank 232 of 512 <2,2,3,0>  R00-M0-N09-J18.
Hello, World! I am process 133 of 512 on Rank 133 of 512 <1,0,2,1>  R00-M0-N09-J04.
Hello, World! I am process 208 of 512 on Rank 208 of 512 <0,1,3,0>  R00-M0-N09-J26.
Hello, World! I am process 145 of 512 on Rank 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.
Hello, World! I am process 196 of 512 on Rank 196 of 512 <1,0,3,0>  R00-M0-N09-J08.
Hello, World! I am process 229 of 512 on Rank 229 of 512 <1,2,3,1>  R00-M0-N09-J10.
Hello, World! I am process 180 of 512 on Rank 180 of 512 <1,3,2,0>  R00-M0-N09-J07.
Hello, World! I am process 175 of 512 on Rank 175 of 512 <3,2,2,3>  R00-M0-N09-J29.
```

Point-to-Point: 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)
```

P2P: (NumPy) array data

```python
from mpi4py import MPI
import numpy

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

if rank == 0:
    data = numpy.arange(1000, dtype='i')
    comm.Send([data, MPI.INT], dest=1, tag=77)
elif rank == 1:
    data = numpy.empty(1000, dtype='i')
    comm.Recv([data, MPI.INT], source=0, tag=77)
```



C
Python (buffer)
Python (pickle)
Throughput [MB/s]
Array Size [bytes/8]
0
20
40
60
80
100
10^0
10^1
10^2
10^3
10^4
10^5
10^6

Non-blocking P2P

```
if rank == 0:
    data = numpy.arange(1000, dtype='i')
    req = comm.Isend([data, MPI.INT],dest=1,...)
elif rank == 1:
    data = numpy.empty(1000, dtype='i')
    req1 = comm.Irecv([data, MPI.INT],source=0,...)
```

< do something, even another Irecv, etc. >

```
if rank == 1:
    status = [MPI.Status(), ... ]
    MPI.Request.Waitall([req1, ...], status)
```

Persitant P2P

Store messaging paramters as a Prequest to be used in a loop:

```
request = comm.Recv_init([msg,MPI.INT],
                         partner_rank)
for i in xrange(10):

    MPI.Prequest.Startall(request)

    do_something()

    MPI.Request.Waitall([request])

    do_something_with(msg)
```

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 & Gather

```
N = 100
assert(N%com.size==0)
if com.rank==0:
    msg = numpy.arange(N,dtype=float)
else: msg = None

dest = numpy.empty(N/com.size, dtype=float)
ans = numpy.empty(com.size, dtype=float)

com.Scatter([msg,MPI.DOUBLE],
            [dest,MPI.DOUBLE],root=0)
mysum = numpy.sum(dest)

com.Gather([mysum, MPI.DOUBLE],
           [ans,MPI.DOUBLE], root=0)
```

ans → [1225. 3725.]

Array data buffer notation

Basic: [buf, MPI datatype]

```
a = numpy.empty(10,dtype=float)
comm.Send([a, MPI.DOUBLE], dest=1, tag=77)
```

Vector collectives: [buf, count, displ, MPI datatype]

```
comm.Scatterv([msg,counts,None,MPI.DOUBLE],
              [b,MPI.DOUBLE])
comm.Allgatherv([b,MPI.DOUBLE],
                [c,counts,None,MPI.DOUBLE])
```

Implementation

Implemented with Cython http://www.cython.org

- 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 !

Portability

- 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.

Cool things I don't have time for

- Dynamic Process Management (spawn, accept/connect).
- Parallel I/O (files, read/write).
- One-sided (windows, get/put/accumulate).

Features Summary (cont.)

- 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).

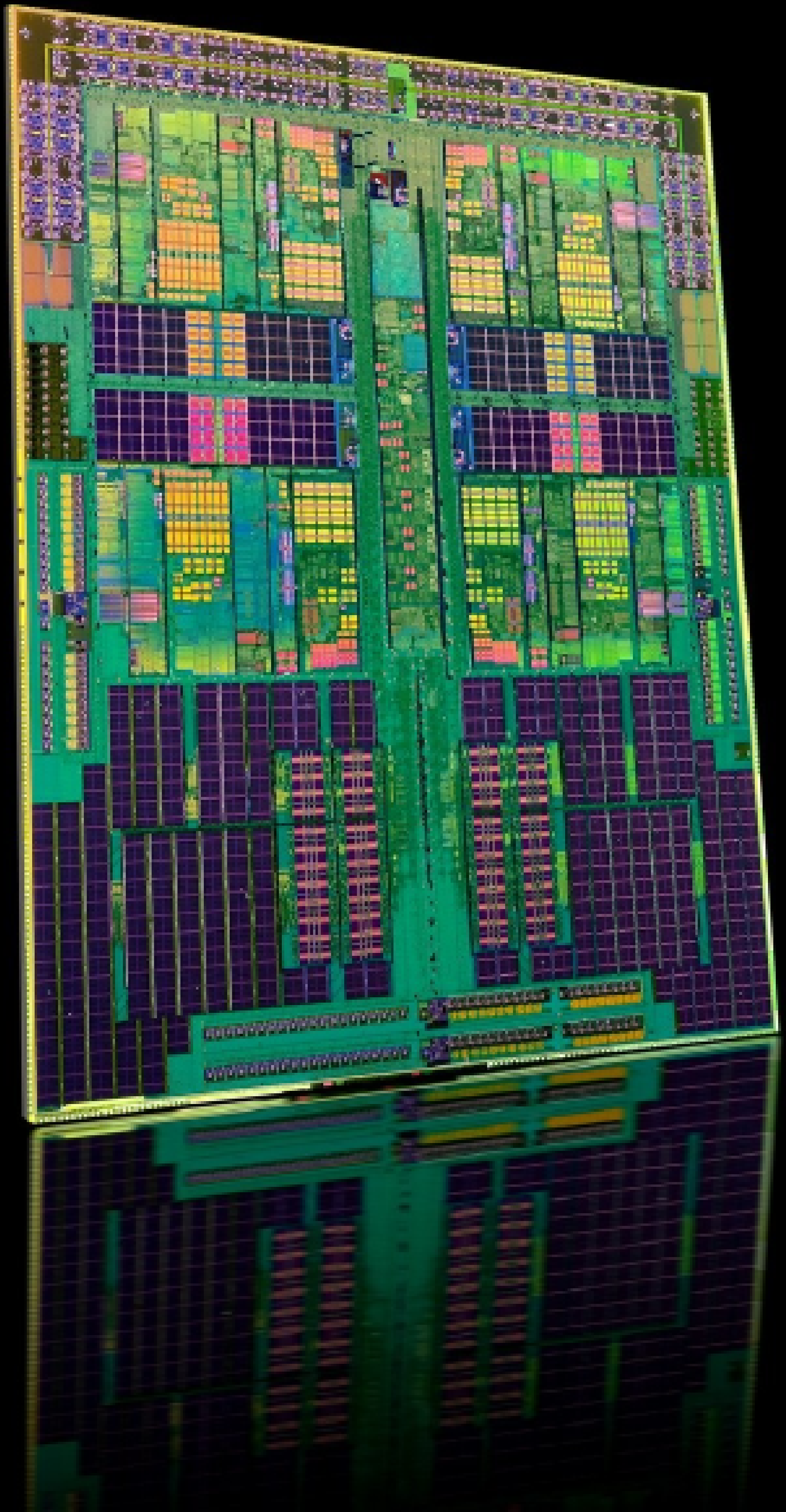
MPI-IPython inter-operability

Start your MPI engines

```
$ ipcluster start -n 4 --engines=MPIEngineSetLauncher
```

Then in python ...

```
> from IPython.parallel import Client
> rc = Client()
> mec = rc[:2]
> mec.block=True
> mec.activate()
> px from mpi4py import MPI
> px print MPI.COMM_WORLD.rank
Parallel execution on engine(s): [0, 1]
[stdout:0] 0
[stdout:1] 1
```



Part 2:

SMP

SMP: Symmetric Multiprocessing

- Homogeneous Multi-core,-cpu
- Shared memory
- Numbers:

  x86: 8-way 6-core Opteron = 48 cores
- Exotic & expensive scaling >8

  Sun SPARC+SGI MIPS ~ double #s

SMP in Python

- Standard as of python 2.6

```
> import multiprocessing
```

- Avoids GIL but higher process creation cost
- Package exists for 2.5

Deadlock

```
import multiprocessing as mp
import time

def h(n, lock):
    lock.acquire()
    n.value += 10

num = mp.Value('d', 0.0)

# lock obj for read and write
lock = mp.Lock()

p1 = mp.Process(target=h, args=(num, lock))
p2 = mp.Process(target=h, args=(num, lock))
p1.start(); p2.start()
p1.join()
```

```python
# p2 is still waiting for release (deadlock)
print p2.is_alive()

# resolve the deadlock
# without killing processes
lock.release()
time.sleep(1)

print p2.is_alive()
p2.join()

print num.value # 10+10=20
```

Race

- When unpredictable order of completion affects output
- Difficult to debug because problematic case maybe infrequent
- Locks can be a solution to enforce atomicity:

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

- Locks are source of deadlocks

Shared memory numpy access

```
def f(a):
    from numpy import ctypeslib
    nd_a = ctypeslib.as_array(a).reshape(dims)
    nd_a[0] = numpy.sum(a)

from multiprocessing import sharedctypes
a = sharedctypes.Array(ctypes.c_double,array)
p = Process(target=f, args=a)
```

Example in SVN:

day3/examples/matmul/mp_matmul_shared.py

Mat mul 4000x2000 * 2000x4000

- 1 CPU (no ATLAS)

  ~20s
- 1 CPU (ATLAS)

  6.48 s
- MP 4 CPU
  - 2.64s
- MP 4 CPU sh. mem.
  - ~1.8s
- IPython → unusable
  - >60s
- naïve weave loop
  - 540s
- MPI 4 CPU
  - 2.07s
- MPI 4 local 4 remote
  - ~8s
- PyBrook(ATI) 0.47s
- PyCUDA(NV) 0.67s

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