Concurrency Kung-fu: Python Style

Eilif Muller

Blue Brain Project

www.facets-project.org

Why Concurrency? (parallelism)

The brain is a parallel machine

Asynchronous

Distributed memory

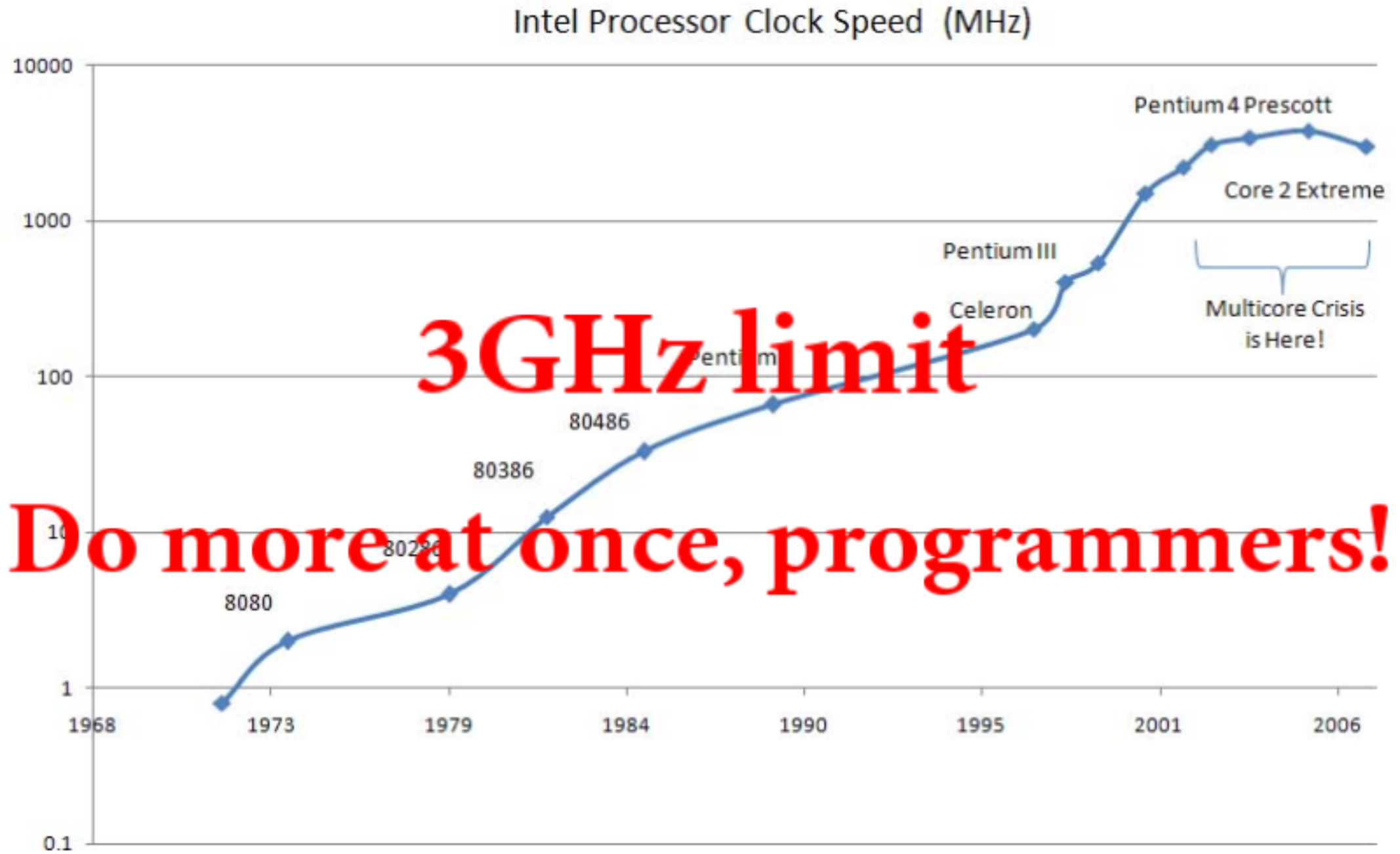
Simple messages

Agnostic to component failure

Connectivity:
dense local, sparse global

Works with long latencies

Why Concurrency? (parallelism)


Intel Processor Clock Speed (MHz)
10000
1000
100
10
1
0.1
1968
1973
1979
1984
1990
1995
2001
2006
8080
80386
80486
Pentium III
Celeron
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.

At least 3 types of Concurrency

SMP	Message passing	Stream
Shared mem. Multi-thread	MPI, sockets Linux Clusters	GPU, Cell
<8 Threads Or $	1000's of processes over network	SIMD Stream: Kernel over arrays
threads multiprocessing	mpi4py, ipython Parallel Python	PyCUDA, PyOpenCL

Others: SSEx ...


Concurrency
Kung-fu
Fundamentals

GROUP = COMM

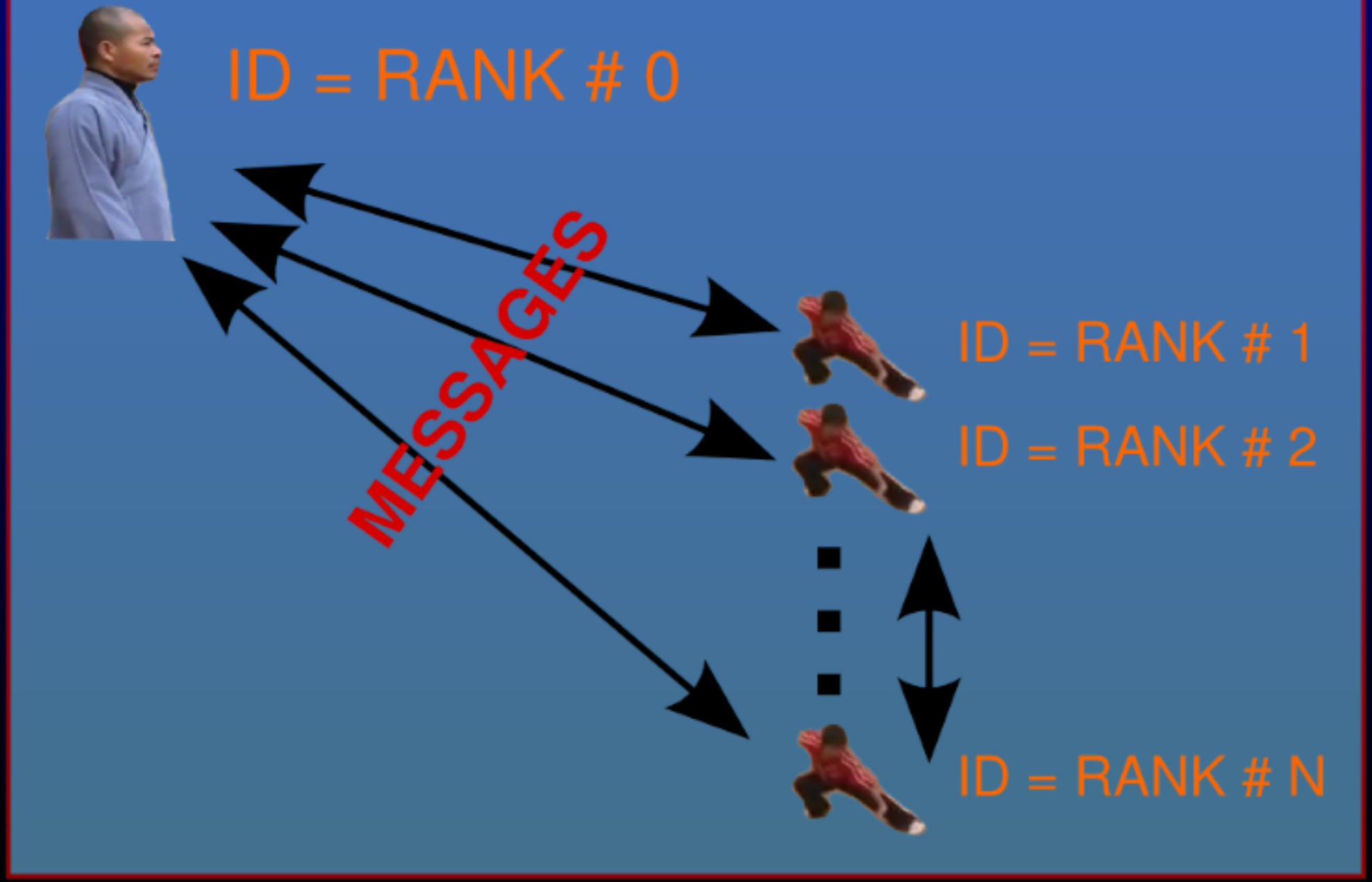
ID = RANK # 0

ID = RANK # 1

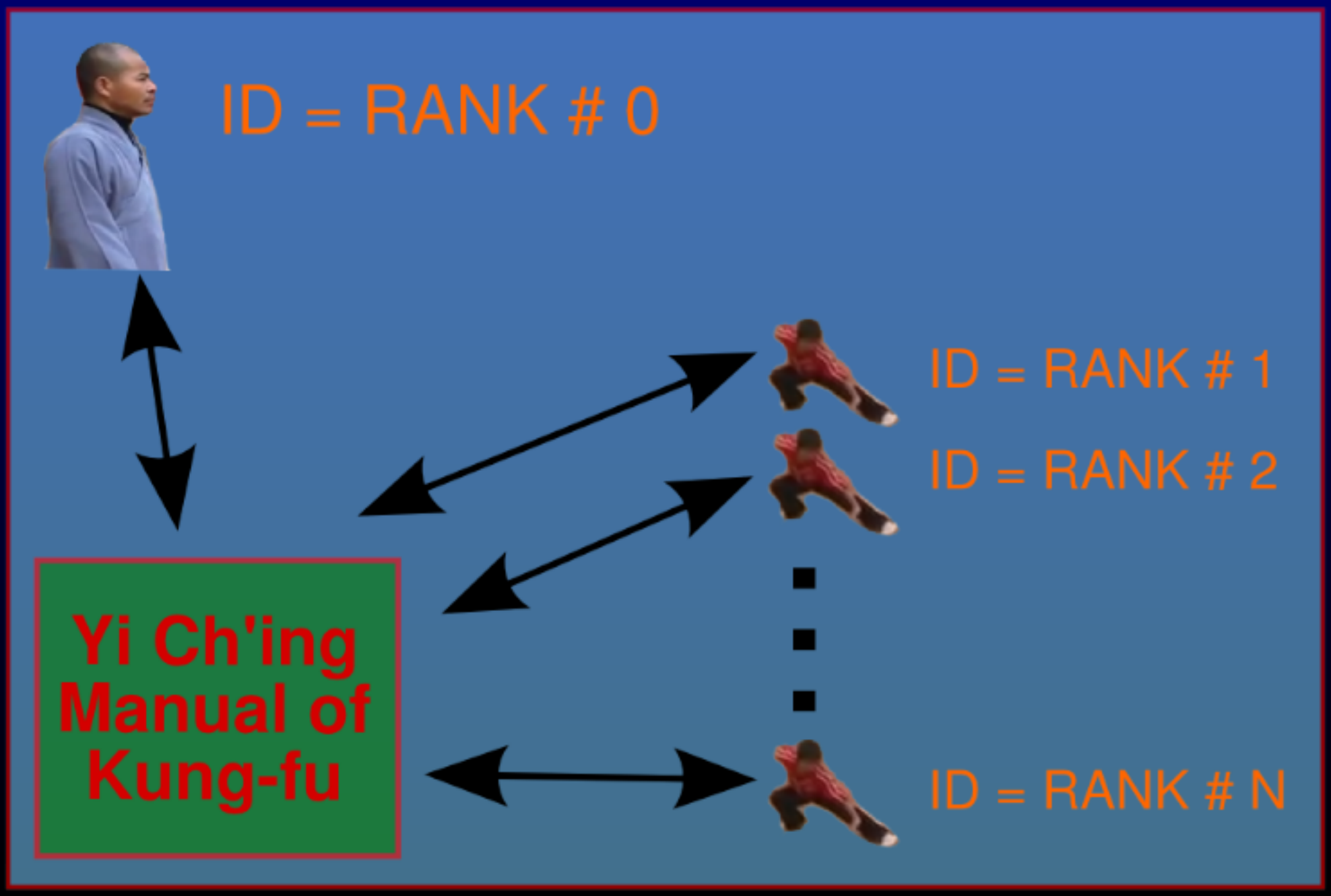
ID = RANK # 2

ID = RANK # N

Inter-Process Communication: Message passing



Inter-Process Communication: Shared Memory



Concurrency Kung-fu
Level 2

Example: Parallel Sum

Example: Parallel Sum

Example: Parallel Sum

Initially highly parallel

Becomes serial

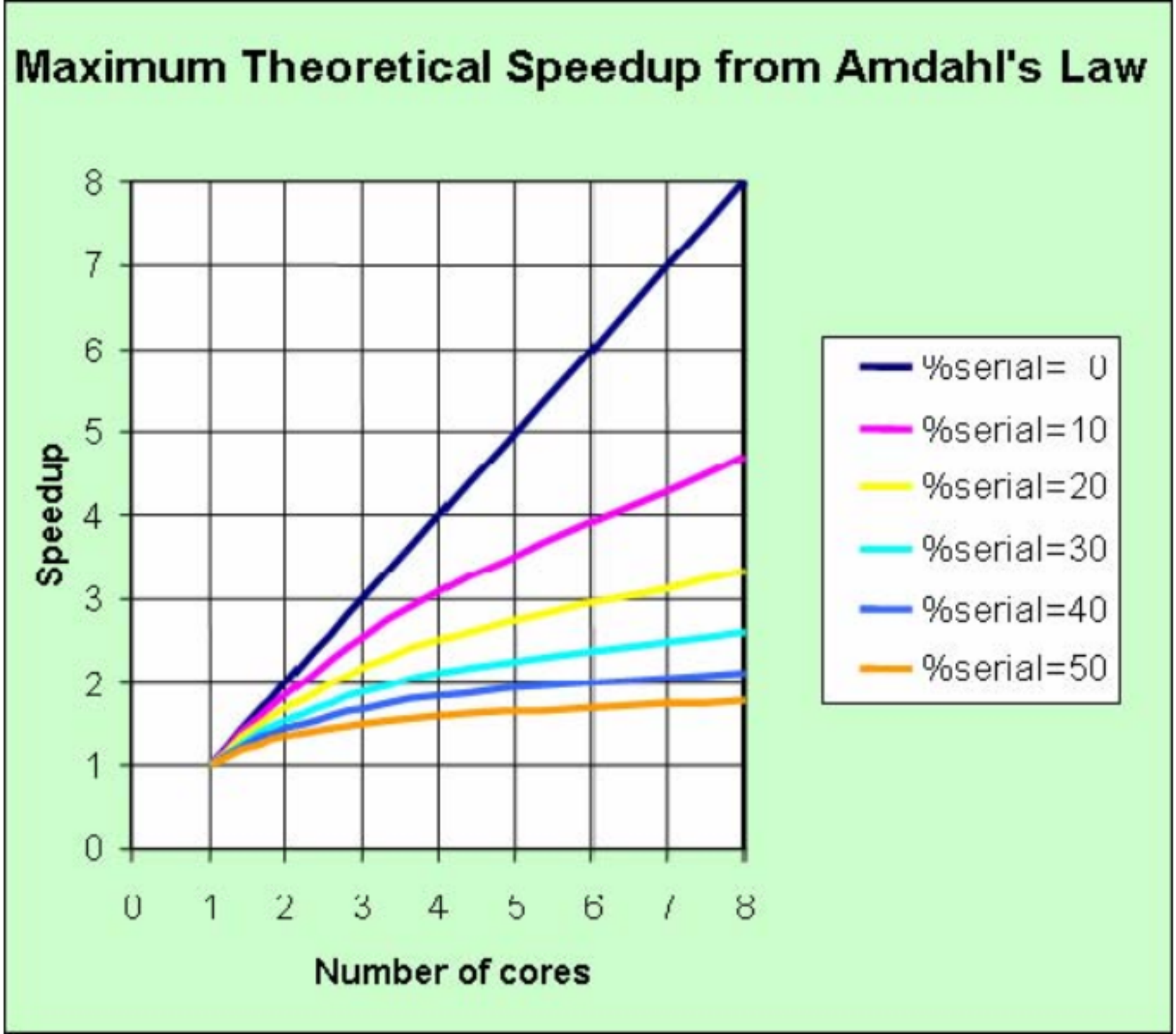
Computation is O(log(N))
or 2^q = N

As is communication ...

But typically op << msg

More concepts

- Load Balancing
  - Dividing the work evenly among compute units
- 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
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"

```
$ 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

```
> exe("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.kernel import client
> mec = client.MultiEngineClient()
> mec['x']
[10,10,10,10]

# kill engines + controller

> mec.kill()
```

MPI inter-operability

```
$ mpdboot
$ mpdtrace
student (your machine name)
$ ipcluster mpiexec -n 4
```

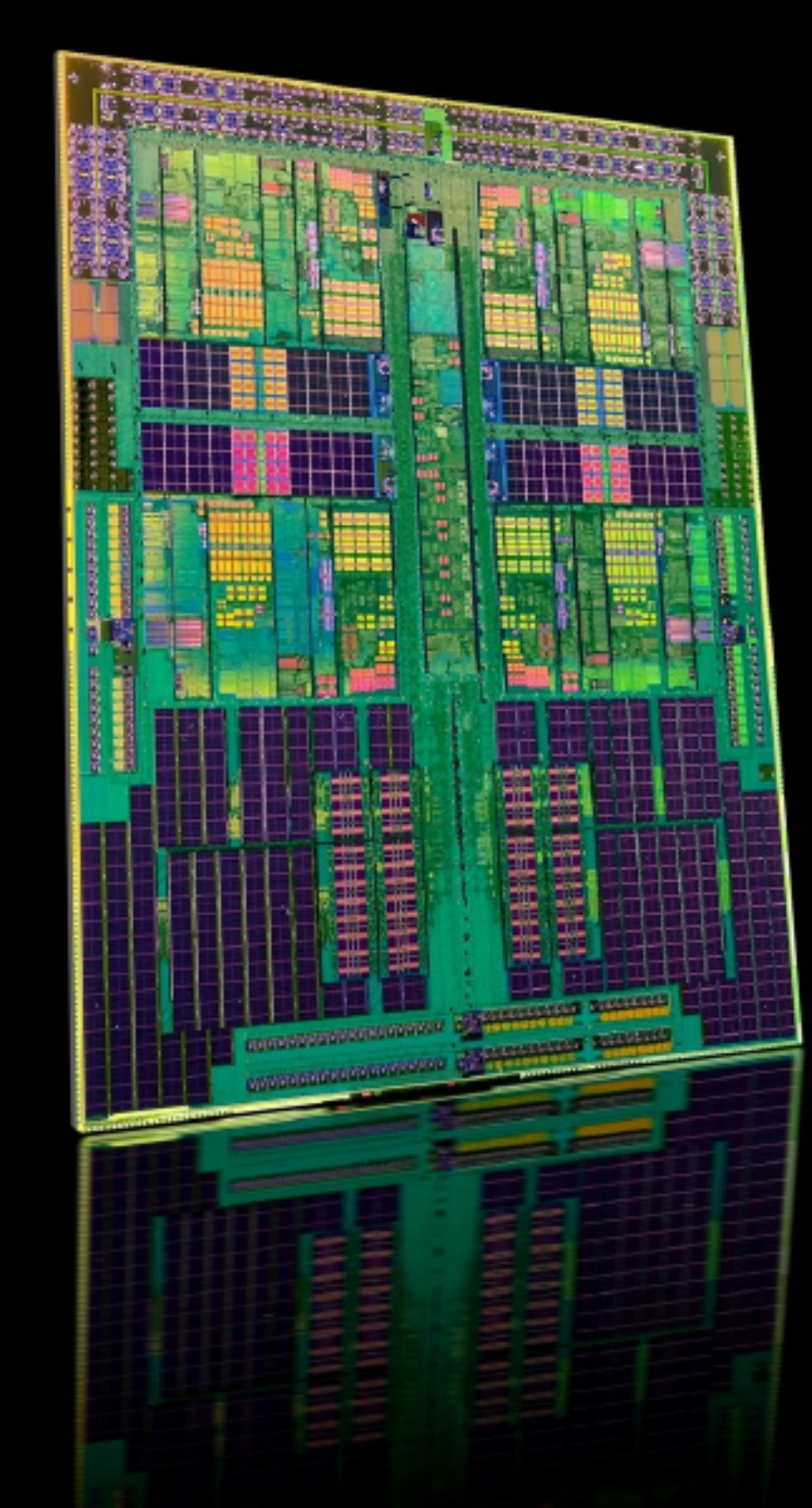
Then in python ...

```
In[]: from IPython.kernel import client
In[]: mec = client.MultiEngineClient()
In[]: mec.activate()
In[]: px from mpi4py import MPI
In[]: mec.execute("print MPI.COMM_WORLD.rank")
0
1
2
3
```

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 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()
```

```
# 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()
```

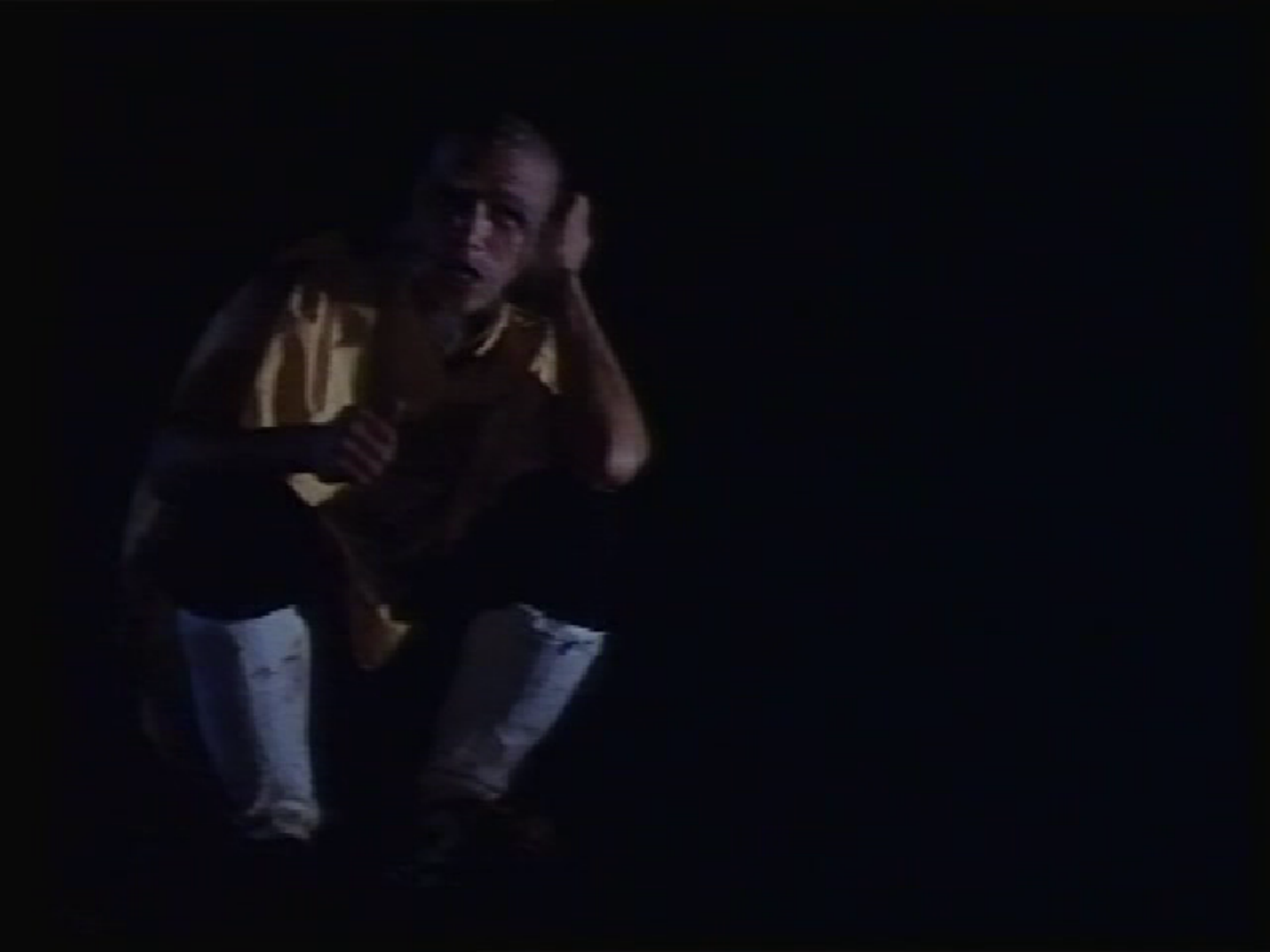
- Locks are source of deadlocks

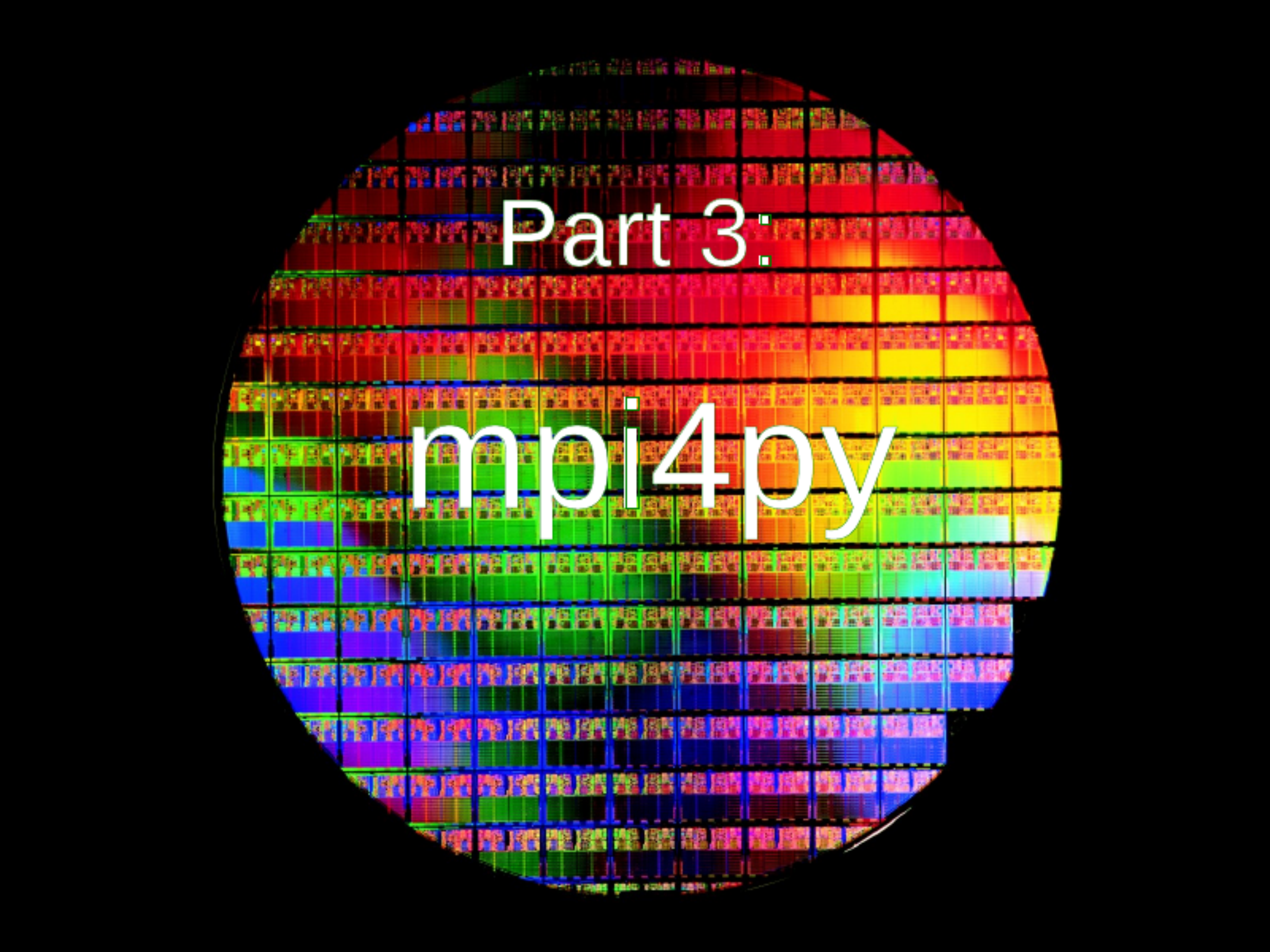
Shared memory numpy access

```
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




Part 3:
mpi4py

Scalable Message Passing Concurrency

- Multi-process execution facilities

```
$ mpdboot
$ mpdtrace
student (names in process ring)
$ 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
```

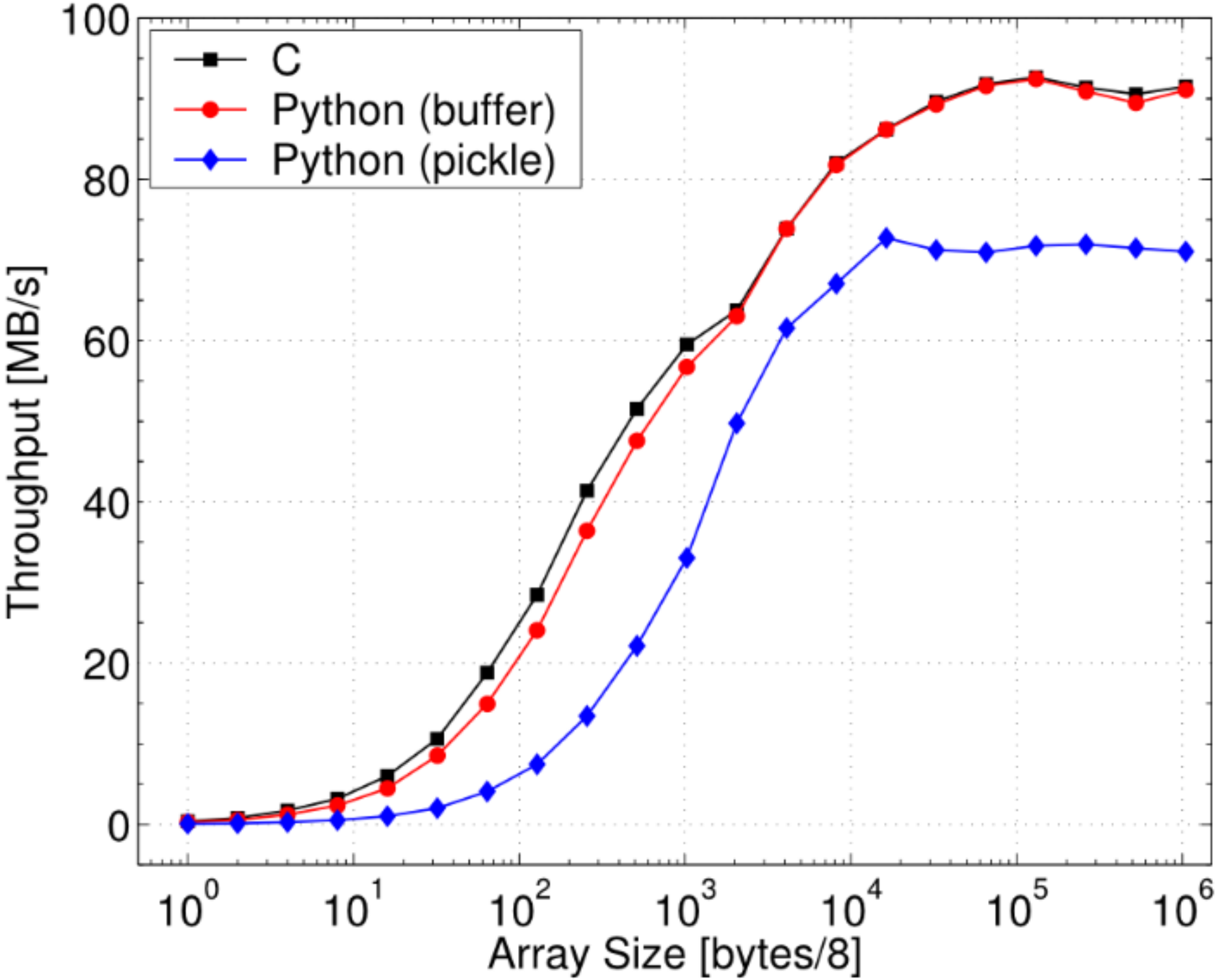
```
Hello, World! I am process 232 of 512 on Rank 232 of 512 <2,2,3,0>  R00-M0-N09-J18.
Hello, World! I am process 133 of 512 on Rank 133 of 512 <1,0,2,1>  R00-M0-N09-J04.
Hello, World! I am process 208 of 512 on Rank 208 of 512 <0,1,3,0>  R00-M0-N09-J26.
Hello, World! I am process 145 of 512 on Rank 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

```
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 $\rightarrow$ [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).

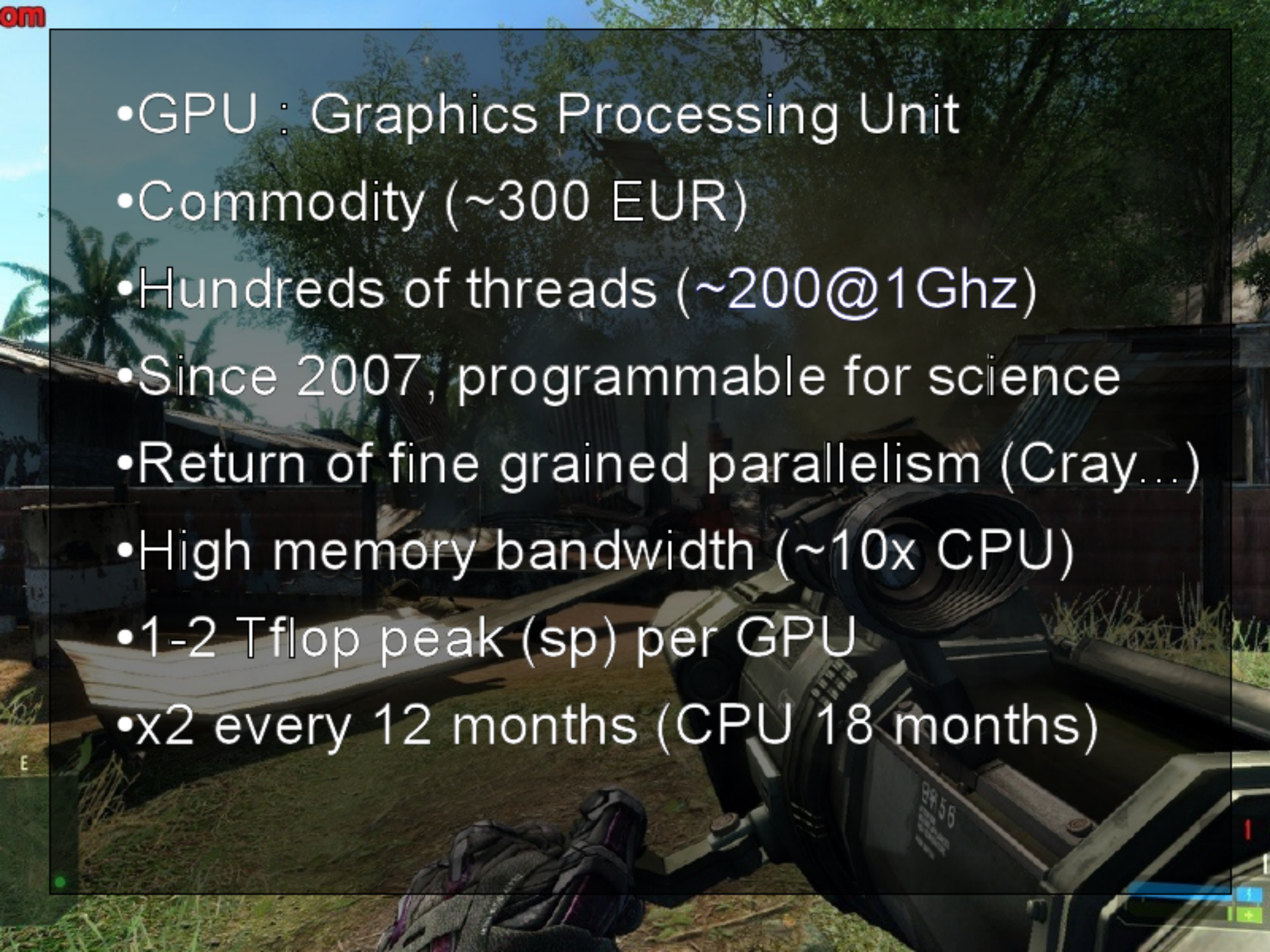
Features Summary (cont.)

- Integration with IPython

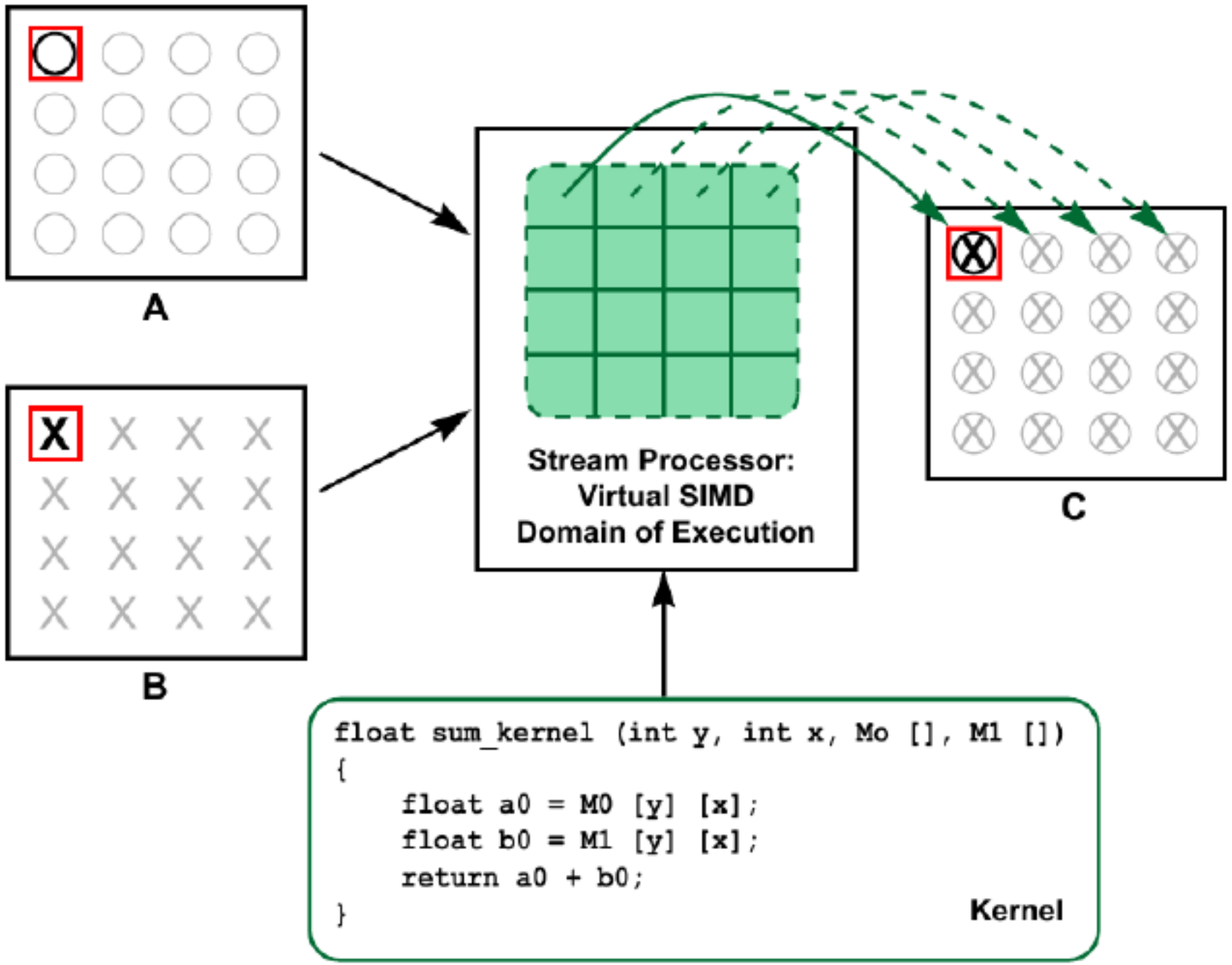
```
$ ipcluster mpiexec
```

  - enables MPI applications to be used *interactively*.

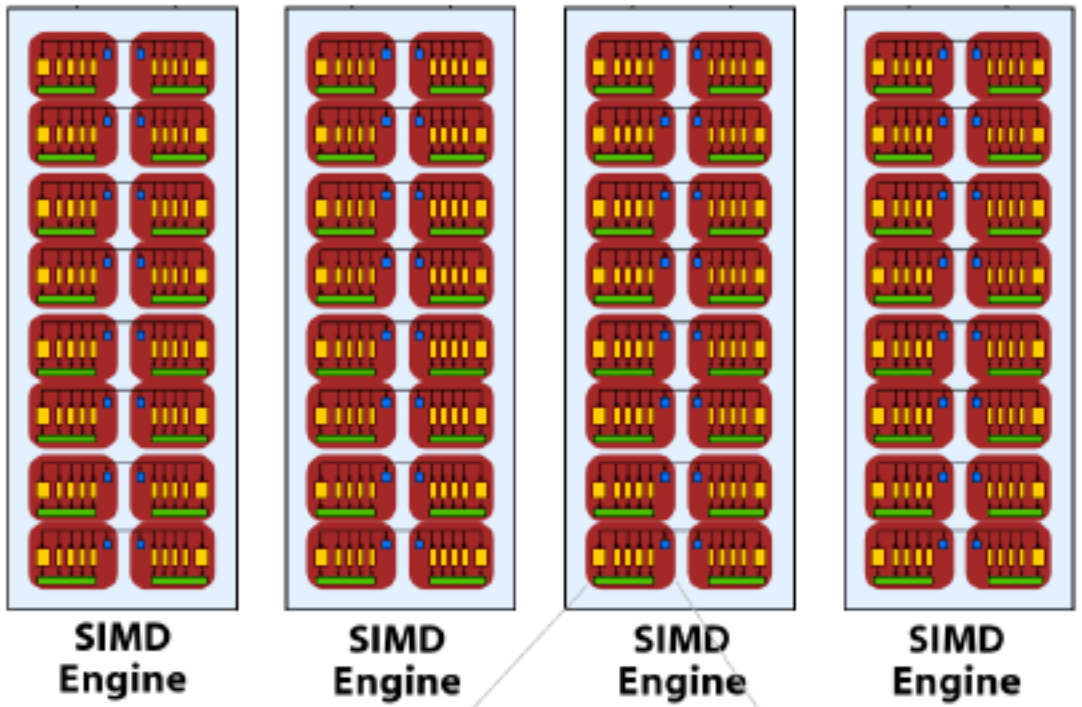

Part 4:
Stream

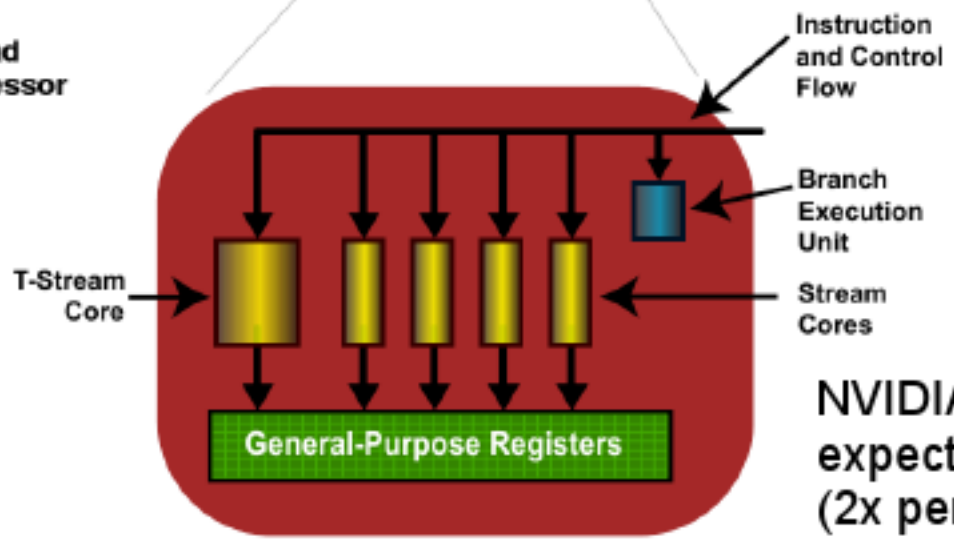

•GPU : Graphics Processing Unit
•Commodity (~300 EUR)
•Hundreds of threads (~200@1Ghz)
•Since 2007, programmable for science
•Return of fine grained parallelism (Cray…)
•High memory bandwidth (~10x CPU)
•1-2 Tflop peak (sp) per GPU
•x2 every 12 months (CPU 18 months)

GPGPU: Stream computing



AMD/ATI Platform





 # SIMDs

10 @ 850 MHz
1.36 Tflop MAD

2x20x16x5=3.2k
cores @ 725 MHz
2x2.32 Tflop MAD

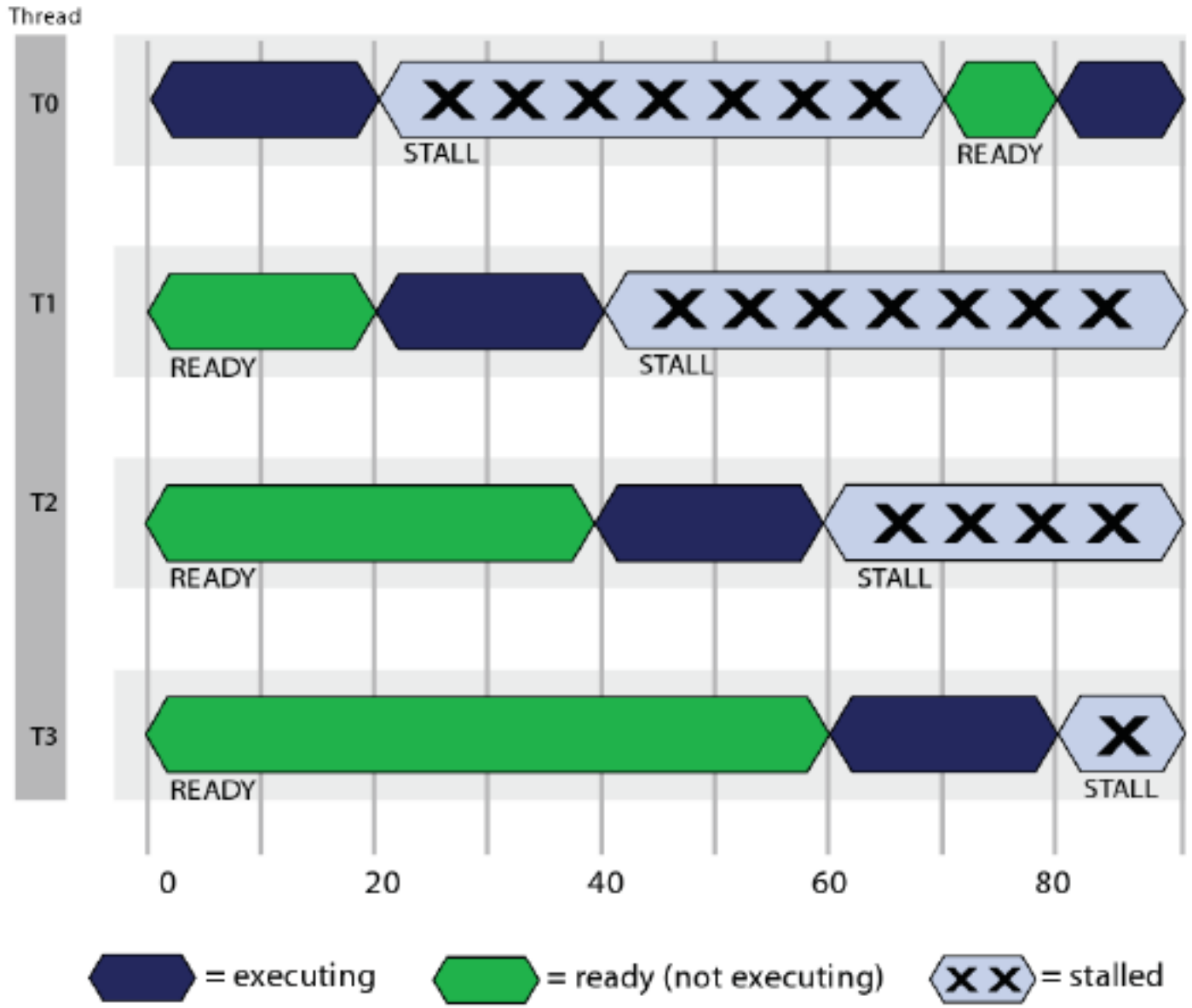
2x480 cores @
1242 MHz
2x596 Gflop MAD

NVIDIA next gen. "FERMI"
expected 1st quarter 2010
(2x perf + new compute features)

NVIDIA

Hiding memory latency

By concurrent execution of >1 threads per core (typically 8)



GPU Compute: 3 Choices

- NVIDA + CUDA
  - Large community
- ATI + Brook or CAL/IL
  - Present hardware leader
  - IL: Access to nifty hardware features
- OpenCL (ATI or NVIDIA)
  - 1/10 performance

GPGPU: A lot of hype, but why bother?

- Programmer resources more expensive than compute resources
- Any new science?

New science: The real-time barrier

- CPU simulations (NEST, BlueBrain, CSIM) which fill memory run at:

  ~1/100th - 1/200th real-time

- 1hr of learning takes 8 days
- Off-line, non-interactive workflows
- No real-world input or output

Consider the following...

$$c_m \frac{dv(t)}{dt} = g_l(E_l - v(t)) + g_e(t)(E_e - v(t)) + g_i(t)(E_i - v(t))$$
$$+ g_s(t)(E_s - v(t)) + g_r(t)(E_r - v(t)).$$

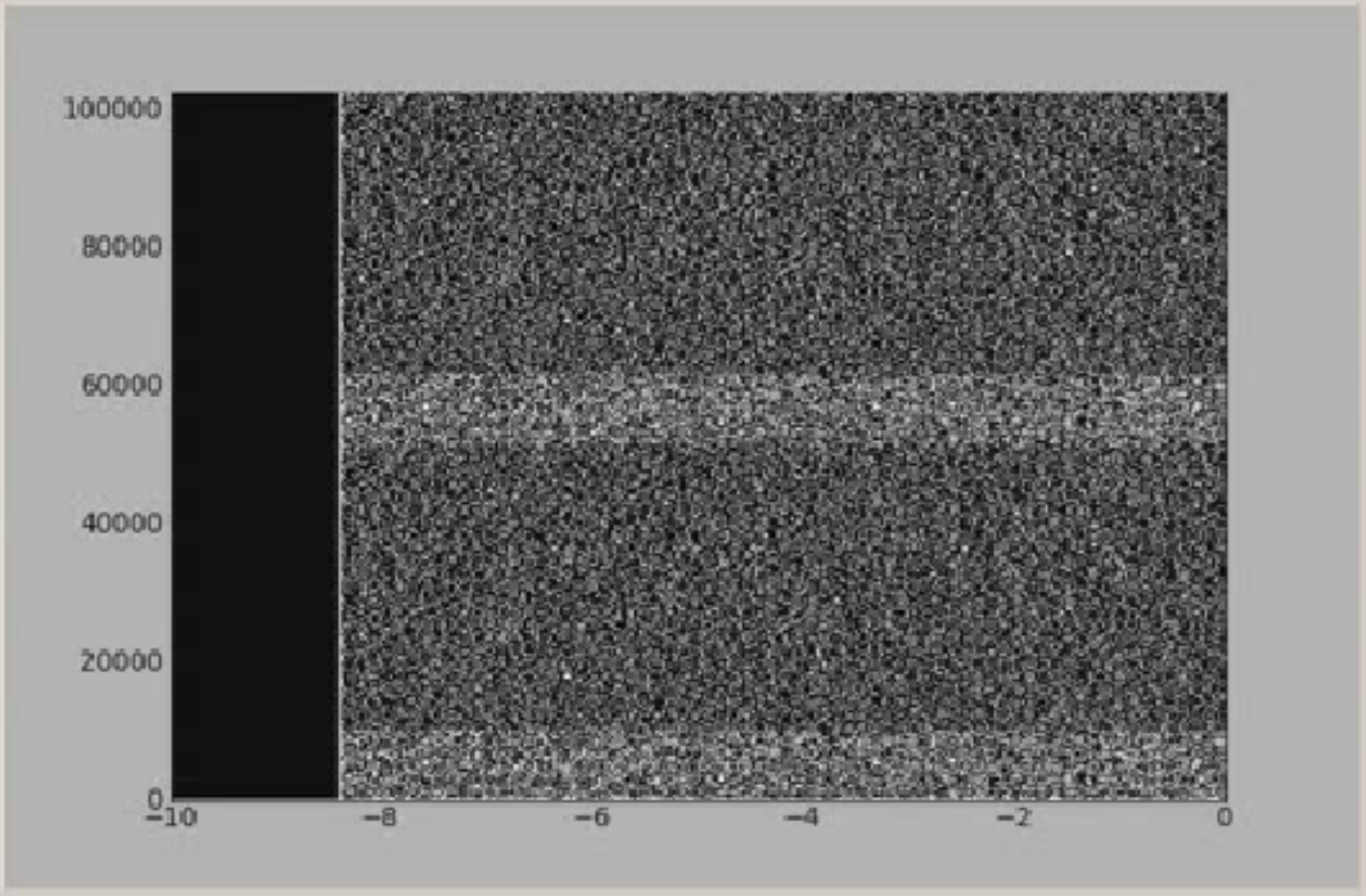
- 10k cells in the high-conductance state
- unconnected
- Poisson input to static synapses (exc:1000*6Hz, inh:250*11Hz)
- NEST, dt=0.1 ms, float64, static synapses
- Numerics: GSL Runge-Kutta-Fehlberg (4,5), RNG: GSL MT19937
- 1 CPU core (i7-920)

10 s bio-time = 1800 s simulation time

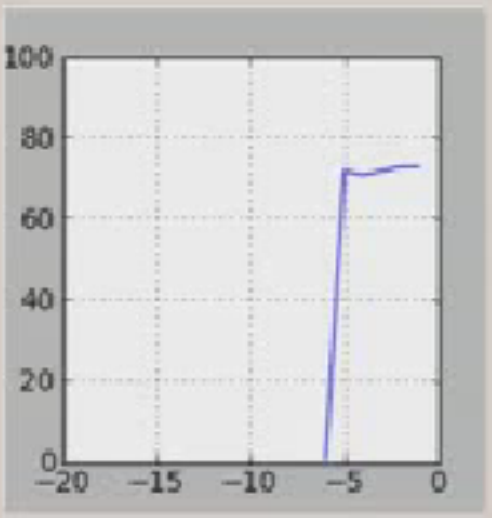
And on ATI/AMD? ...

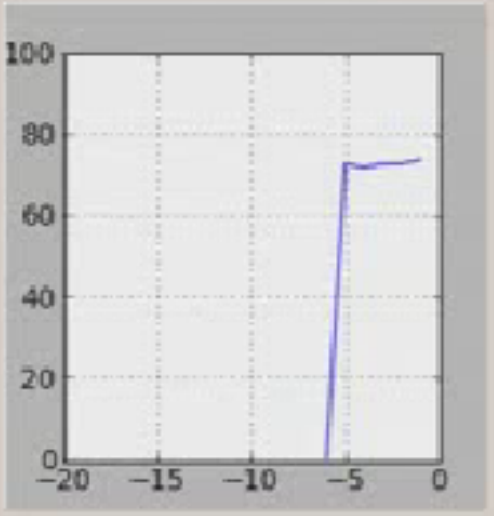
mp_gumon.py

go gumon!



GPU loads





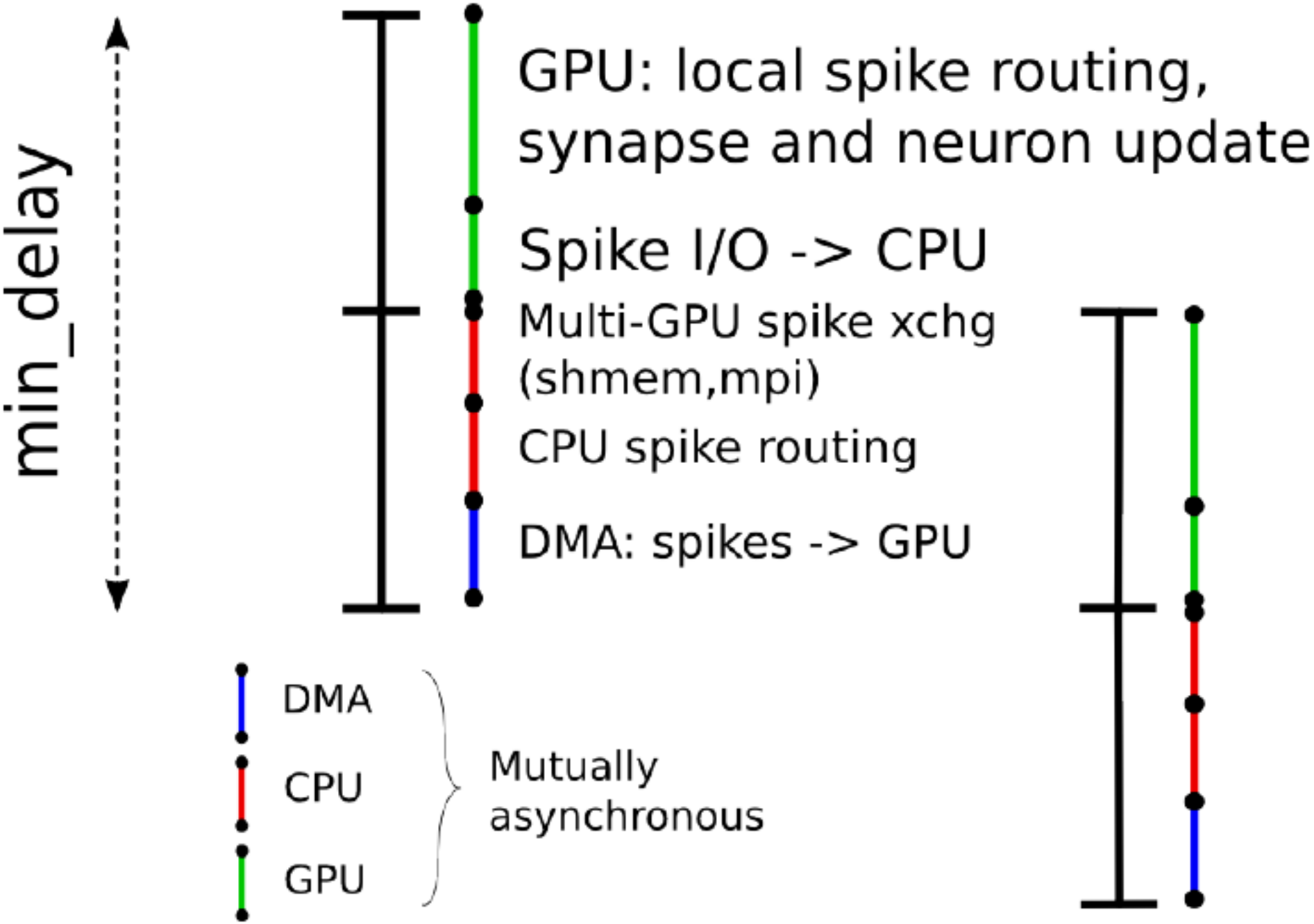
GPGPU impl for AMD/ATI

$$c_m \frac{dv(t)}{dt} = g_l(E_l - v(t)) + g_e(t)(E_e - v(t)) + g_i(t)(E_i - v(t))$$
$$+ g_s(t)(E_s - v(t)) + g_r(t)(E_r - v(t)).$$

- 50k cells in the high-conductance state
- unconnected
- Poisson input to static synapses (exc:1000*6Hz, inh:250*11Hz)
- dt=0.1 ms, float32, static synapses
- Numerics: Euler, RNG: WarpStandard
- 4 cells/thread, 12800 threads
- Asynchronous spike I/O

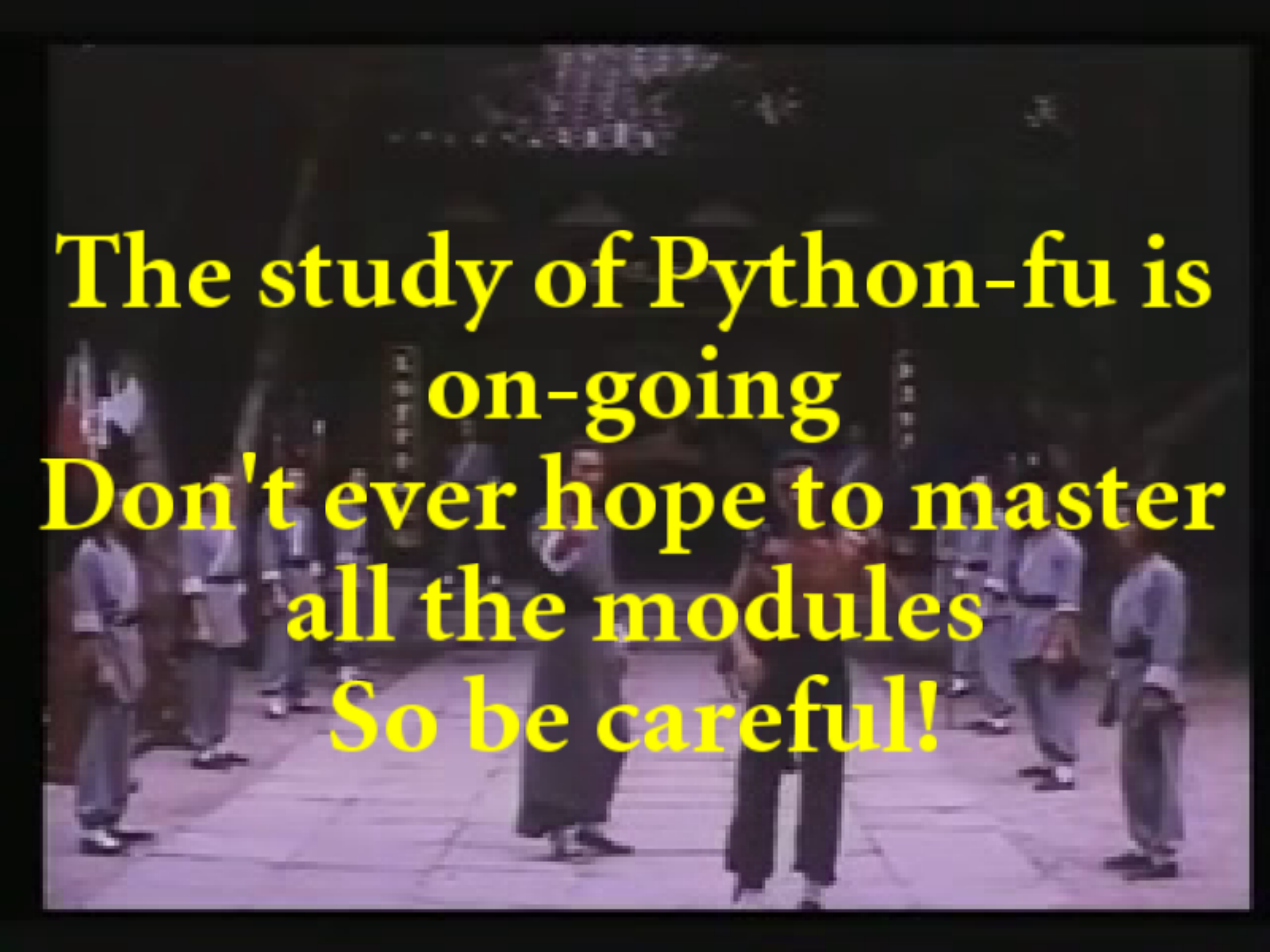
10 s bio-time = ~10 s simulation time

GPU is worker, CPU is manager



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


The study of Python-fu is
on-going
Don't ever hope to master
all the modules
So be careful!