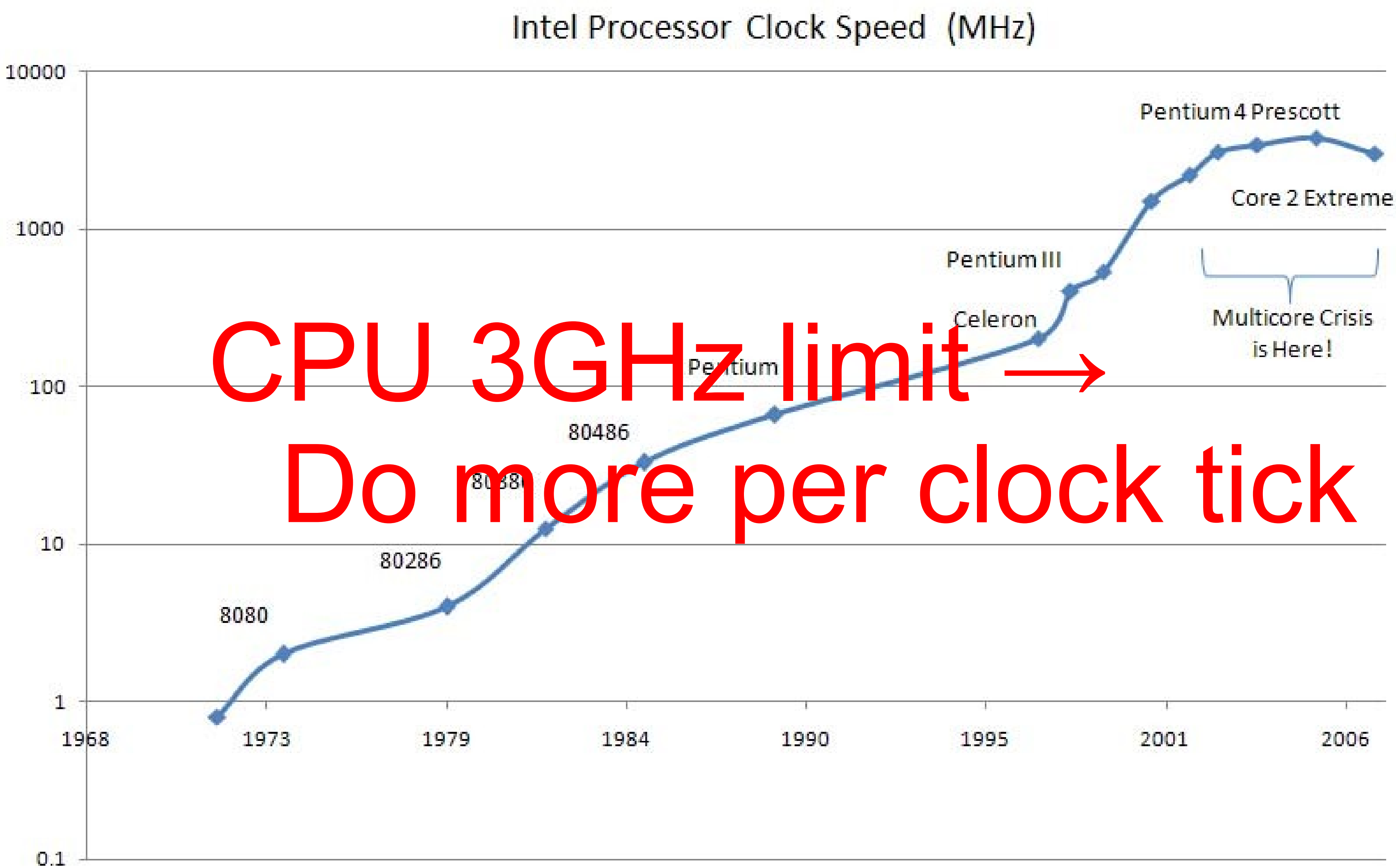
Day 3: Practical Concurrency with Python

Eilif Muller, LCN-EPFL

Why Concurrency?



CPU 3GHz limit →

Do more per clock tick

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

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

At least 3 types of Concurrency

SMP	Message passing	SIMD + Stream
Shared mem. Multi-thread	MPI, sockets Linux Clusters	GPU, Cell, SSEx
<8 Threads Or $	1000's of processes over network	SIMD Stream: Kernel over arrays
threads multiprocessing	Mpi4py, ipython Parallel Python	PyCUDA, numpy

Concepts

- Thread & Execution Management

```
> group = ProcessGroup(n=4)
$ mpiexec -n 16 python main.py
```

- Mixing Serial & Parallel: Synchronization

```
> wait_all_done(group)
> MPI.Barrier()   # Blocks till all processes arrive
```

- Data exchange, asynchronous, atomicity:

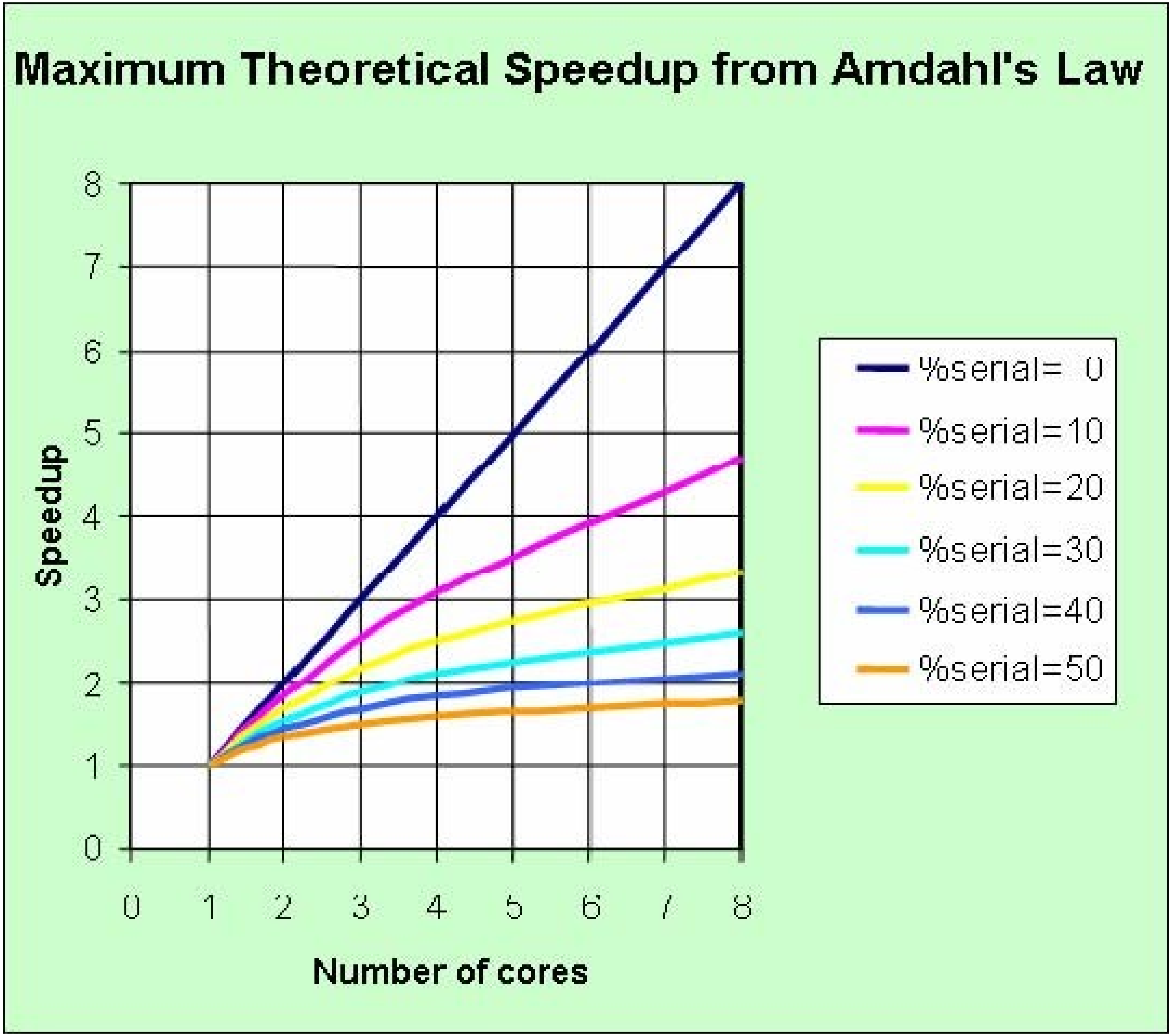
```
> msg = MPI.Isend(receiver=5, object, tag)
> while msg.pending() == True: something_else()
```

More concepts

- Loop parallelism → Functional programming
  - > ans = group.map(dot,zip(a,b))
- Thread/process local namespace
  - > import numpy
  - > group.execute('import numpy')
- Parallel Specific Bugs: Deadlock, Race

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?



Source: “Is the free lunch really over? Scalbility in multi-core systems”, Intel Whitepaper

Part 1:

Easy Concurrency with IPython

Start “slaves”:

```
$ ipcluster -n 4
```

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])
```

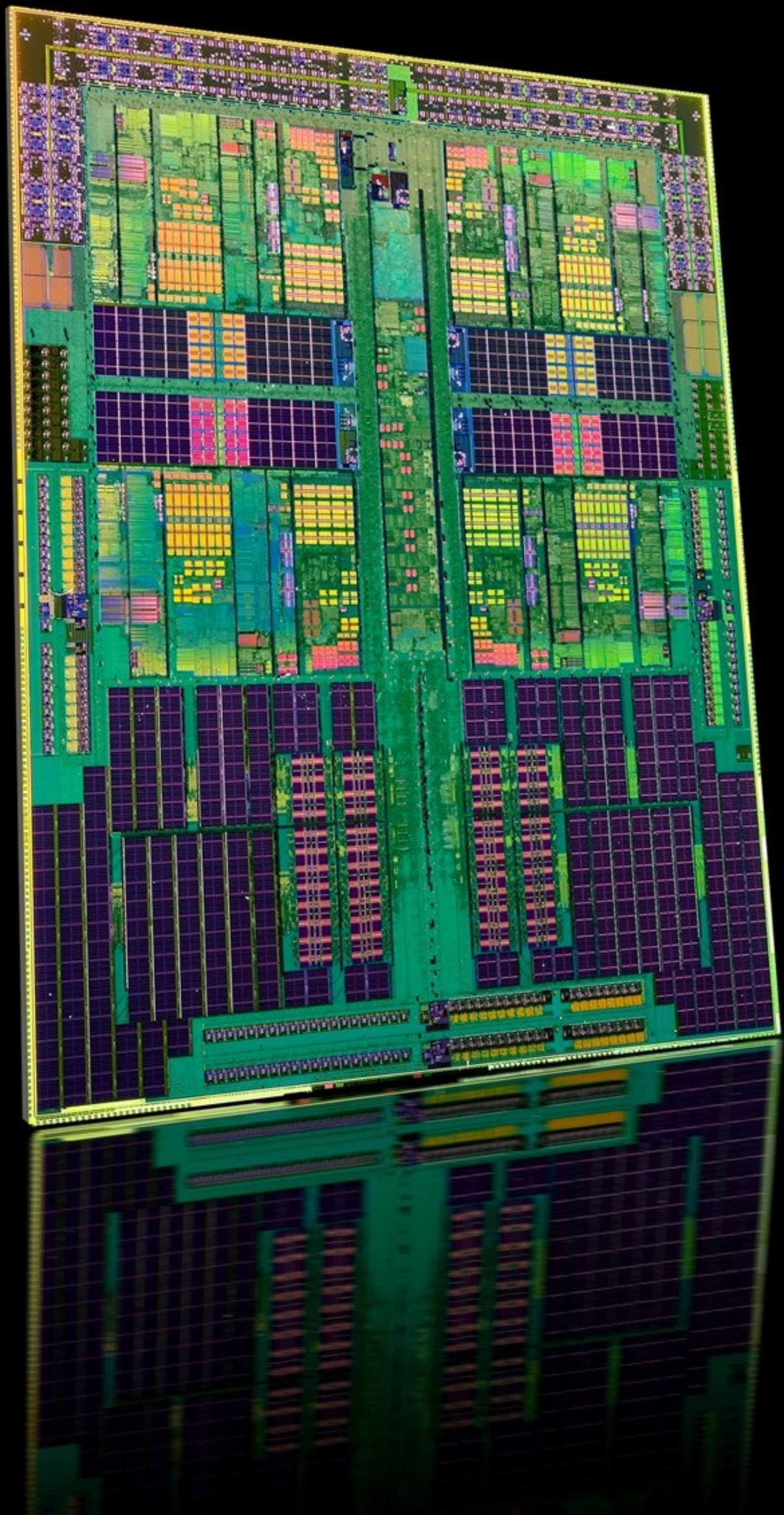
Gather

```
> ''.join(mec.gather(“a”))
'hello WORLD'
> mec.kill(controller=True)
```

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

- As of python 2.6
  - > import multiprocessing
  - back-port exists for 2.5

Examples:
mp_race.py, mp_norace.py,
mp_deadlock.py

Race condition

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

  > l = Lock()

  > l.acquire(); <code>; l.release()
- Locks are source of deadlocks

Shared memory numpy access

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

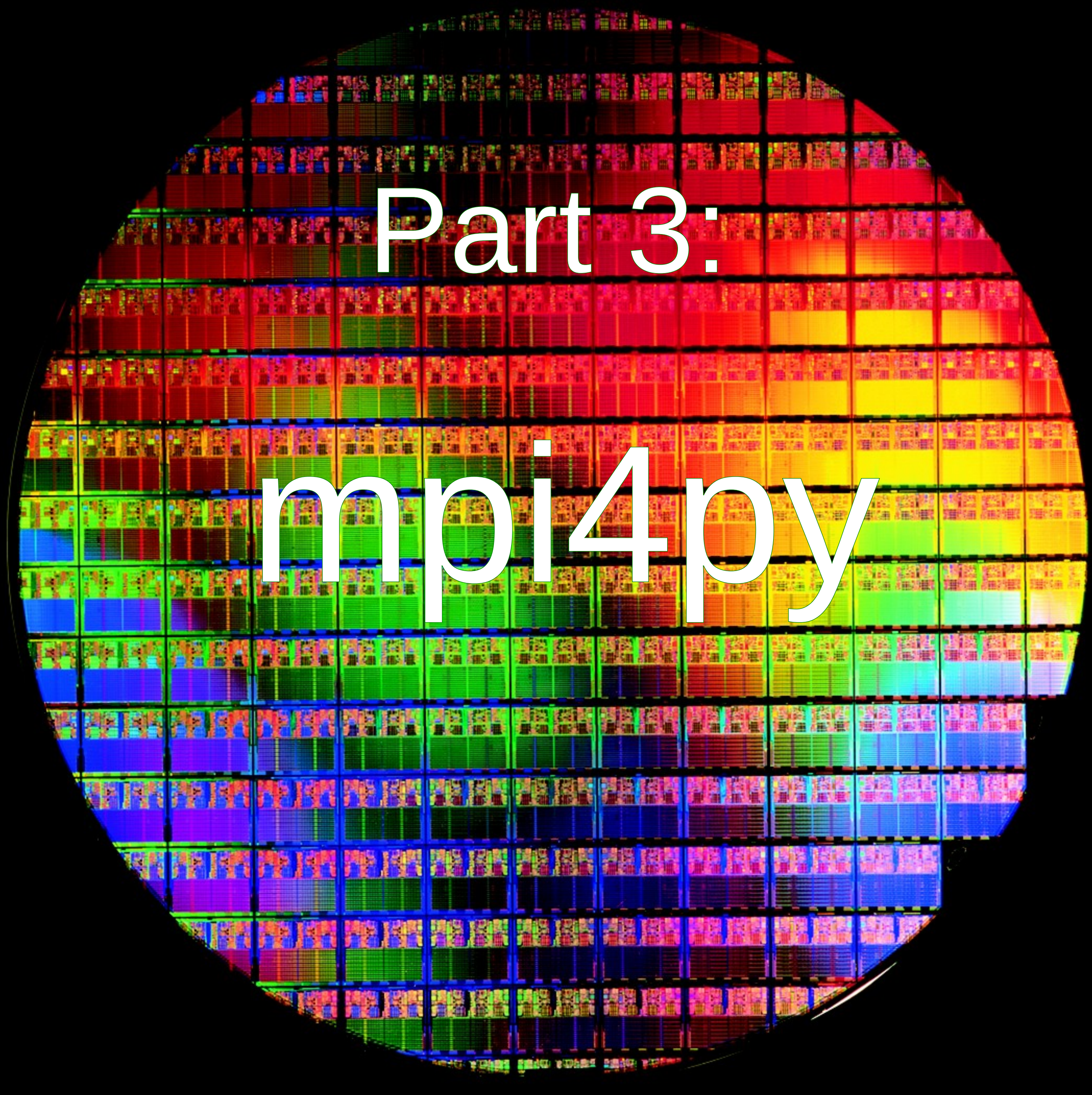
  from numpy import ctypeslib

  nd_a = ctypeslib.as_array(a).reshape(dims)

  nd_a[0] = numpy.sum(a)
- Example in SVN: day3/examples/matmul/mp_matmul_shared.py

Part 3:

mpi4py



mpi4py = MPI for Python

- MPI = Message Passing Interface
- A wrapper for widely used MPI
  - MPICH2, OpenMPI, LAM/MPI
- MPI support by wide range of vendors, hardware, languages
- High-performance
- Heterogeneous clusters

> from mpi4py import MPI

- Communicator = Comm
  - Manages processes and communication between them
  - MPI.COMM_WORLD
    - all processes defined at exec.time
- Comm.size, Comm.rank

```
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
```

mpiexec needs mpd

- mpd – the process management daemon
- Starting (single machine)

  $ mpdboot
- Testing

  $ mpdtrace

    rucola
- Stopping

  $ mpdallexit

Blocking Messages

```
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)
```

- Use comm.send, comm.recv for Python objects (will be pickled)
- Process waits for completion
- Send&Recv must come in pairs → deadlocks

Non-Blocking Messages

```
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. >

   status = [MPI.Status(), MPI.Status()]
   MPI.Request.Waitall([req1, ...], status)
```

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
- Example: examples/matmul/mpi_matmul.py

Part 4:

Stream

GPU Stream computing

- Hundreds of hardware threads
- Thread runs simple “kernel” compiled for GPU
  - Written in DSL like CUDA or Brook
- 100x theoretical throughput of CPU
- Fine grained parallelism
- Data must transit PCIe bus
  - Can stay on GPU over kernel calls
- Example → matrix multiply

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.
  - ?
- IPython → unusable
  - >60s
- naïve weave loop
  - 540s
- MPI 4 CPU
  - 2.07s
- MPI 4 local 4 remote
  - ~8s
- ATI Stream 0.47s
- PyCUDA 0.67s