Python: performance and parallelism

Fernando Pérez
Fernando.Perez@berkeley.edu

Helen Wills Neuroscience Institute, UC Berkeley

"CSE: Science Research Computing with Python", CCN 06180
Nov 1, 2010
Outline

1. HPC and parallelism
2. Python
3. Language and standard libraries
4. External Libraries/projects
5. IPython for parallelism
Outline

1. HPC and parallelism
2. Python
3. Language and standard libraries
4. External Libraries/projects
5. IPython for parallelism
Data explosion

Our data sets are getting huge!

- Sloan Digital Sky Survey Data Release 7: ~65 Terabytes
- Neuroimaging: 1 hour → a few gigabytes
- etc...

A simple problem: solve $Ax = b$ via Gaussian elimination

- For $A$ an $n \times n$ matrix, cost is $\approx \frac{1}{3} n^3$ floating point operations
- On a computer with ~100MFlops sustained performance

<table>
<thead>
<tr>
<th>n</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>$O(1)$ μs</td>
</tr>
<tr>
<td>100</td>
<td>$O(1)$ ms</td>
</tr>
<tr>
<td>1000</td>
<td>$O(1)$ s</td>
</tr>
<tr>
<td>10000</td>
<td>$O(1)$ hour</td>
</tr>
<tr>
<td>100000</td>
<td>$O(1)$ month</td>
</tr>
<tr>
<td>1000000</td>
<td>$O(1)$ century</td>
</tr>
</tbody>
</table>
Why do we need fast algorithms?

Because computers are getting bigger and faster!
Why do we need fast algorithms?

Because computers are getting **bigger and faster**!
Why do we need fast algorithms?

Because computers are getting bigger and faster!
Why do we need fast algorithms?

Because computers are getting **bigger and faster**!

![Graph showing time (hours) vs. problem size N for two algorithms: $O(N^2)$ and $O(N \log N)$](chart.png)
**Why do we need fast algorithms?**

Because computers are getting **bigger and faster!**

---

**Diagram:**

- $\mathcal{O}(N^2)$
- $\mathcal{O}(N \log N)$
- 1 day
- 2 days
- 3 days

**Graph:**

- Y-axis: Time (hours)
- X-axis: Problem size $N$

The graph shows the time required for different algorithms and problem sizes, indicating how the time increases with increasing problem size.
Parallel computing: why should we care?

Because reality looks like this:

Sources: Intel, Microsoft (Sutter), Stanford (Olukotun, Hammond) & Berkeley (Yelick)
Parallel computing: why should we care?

Because reality looks like this:

Sources: Intel, Microsoft (Sutter), Stanford (Olukotun, Hammond) & Berkeley (Yelick)
We can’t escape thermodynamics

The vendor’s solutions:

- Multicore chips: even in your laptop.
- Graphics cards for general computing: > 128 ‘processors’ per card.
- High-density clusters: SiCortex (> 5000 processors in a cabinet).
The vendor’s solutions:

- Multicore chips: even in your laptop.
- Graphics cards for general computing: $\geq 128$ ‘processors’ per card.
- High-density clusters: SiCortex ($\geq 5000$ processors in a cabinet).
Parallel programming?

There are plenty of bad news

- It is in general, *extremely* difficult.
- Scientific productivity plummets with enormous up-front efforts.
- Development, debugging, running is hard and cumbersome.
- With proprietary tools, licensing costs can go through the roof.

But not all is doom and gloom

- Many problems are *embarrassingly parallel*: uncoupled components.
- This is common in many fields:
  - Analyze many scans in an fMRI run.
  - Global parameter searches...
- Even not-so-embarrassingly parallel problems can be tractable...
  - ... with the right tools.
Parallel programming?

There are plenty of bad news

- It is in general, *extremely* difficult.
- Scientific productivity plummets with enormous up-front efforts.
- Development, debugging, running is hard and cumbersome.
- With proprietary tools, licensing costs can go through the roof.

But not all is doom and gloom

- Many problems are *embarrassingly parallel*: uncoupled components.
- This is common in many fields:
  - Analyze many scans in an fMRI run.
  - Global parameter searches...
- Even not-so-embarrassingly parallel problems can be tractable...
  - ... with the right tools.
Basic terms

- **CPU (Central Processing Unit)**: unit capable of executing the arithmetic and logical instructions in a program.

- **Core**: single unit capable of executing a program in a computer.

- **Multicore**: single processor that contains multiple physical cores on one chip (they share some resources)

- **SIMD (Single Instruction Multiple Data)**: all units execute the same code on different datasets.

- **MIMD (Multiple Instruction Multiple Data)**: each unit executes different code.

- **Process**: separate program executed in isolation by the Operating system.

- **Thread**: stream of execution within a program, can share resources with other threads.

- **Shared memory**: all units have read/write access to the same data (typically living in the same process).
OpenMP: Open MultiProcessing

- Threads-based, shared-memory multiprocessing.
- All caveats of threads apply

OpenMP example

```c
int main(int argc, char *argv[]) {
    const int N = 100000;
    int i, a[N];

    #pragma omp parallel for
    for (i = 0; i < N; i++)
        a[i] = 2 * i;

    return 0;
}
```
MPI_Init(&argc,&argv); /* all MPI programs start with MPI_Init; 
    all 'N' processes exist thereafter */
MPI_Comm_size(MPI_COMM_WORLD,&numprocs); /* find out how big the 
    SPMD world is */
MPI_Comm_rank(MPI_COMM_WORLD,&myid); /* and this processes' rank 
    is */
/* At this point, all programs are running equivalently, the rank 
    distinguishes 
    the roles of the programs in the SPMD model, with rank 0 often 
    used specially... */
if(myid == 0) {
    printf("%d: We have %d processors\n", myid, numprocs);
    for(i=1;i<numprocs;i++) {
        sprintf(buff, "Hello %d! ", i);
        MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD); }
}
else { /* receive from rank 0: */
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat );
    sprintf(idstr, "Processor %d ", myid);
}
MPI_Finalize(); /* MPI Programs end with MPI Finalize; this is a 
    weak synchronization point */
How fast can we go?

Exercise: Derive and plot Amdahl’s law

Upper bound on speedup achievable via parallelization

- $s$: serial fraction of total work to be done
- $1 - s$: parallelizable fraction
- $p$: number of processors used
Amdahl’s law: a logical limit

Consider solving a problem where communication has zero cost

- Using 1 processor: \( T_1 = s + (1 - s) = 1 \)
- Using \( p \) processors: \( T_p = s + \frac{1-s}{p} \)

The total speedup possible:

\[
\frac{T_1}{T_p} = \frac{1}{s + \frac{1-s}{p}} < \frac{1}{s}
\]

Lesson: The serial fraction is a hard limit!
Amdahl’s law: a logical limit

Consider solving a problem where communication has zero cost

- Using 1 processor: \( T_1 = s + (1 - s) = 1 \)
- Using \( p \) processors: \( T_p = s + \frac{1-s}{p} \)

The total speedup possible:

\[
\frac{T_1}{T_p} = \frac{1}{s + \frac{1-s}{p}} < \frac{1}{s}
\]

Lesson: The serial fraction is a hard limit!
```python
def amdahl(s, p):
    return 1.0/(s+(1.0-s)/p)

p = np.logspace(0, 16, 9, base=2)
ax = plt.subplot(111)

for s in [0.5, 0.25, 0.1, 0.05]:
    sp = amdahl(s, p)
    ax.semilogx(p, sp,'-o', label='s=%.2f' % s, basex=2.0)

ax.set_xlabel('Processors')
ax.set_ylabel('Speedup')
ax.set_title('Speedup as a function of serial fraction')
ax.legend()
ax.grid()
plt.show()
```
Amdahl’s law

Speedup as a function of serial fraction

- $s=0.50$
- $s=0.25$
- $s=0.10$
- $s=0.05$

Processors vs. Speedup

- Processors: $2^0, 2^2, 2^4, 2^6, 2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}$
- Speedup: $0, 5, 10$
- Finding enough parallelism (Amdahl’s law)
- Granularity: bite-sized chunks for each unit...
  - But need large enough amount of work to hide the overhead
- Locality
  - large memories are slow, fast memories are small.
- Load balance
- Coordination and synchronization
  - Communication is expensive...
  - But getting the wrong answer fast doesn’t cut it.
- Model performance, profile, profile, profile...
  - If you didn’t measure it, you don’t actually know.
  - Your intuition is wrong.
<table>
<thead>
<tr>
<th></th>
<th>Dense Linear Algebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Sparse Linear Algebra</td>
</tr>
<tr>
<td>3</td>
<td>Spectral Methods</td>
</tr>
<tr>
<td>4</td>
<td>N-Body Methods</td>
</tr>
<tr>
<td>5</td>
<td>Structured Grids</td>
</tr>
<tr>
<td>6</td>
<td>Unstructured Grids</td>
</tr>
<tr>
<td>7</td>
<td>MapReduce</td>
</tr>
<tr>
<td>8</td>
<td>Combinational Logic</td>
</tr>
<tr>
<td>9</td>
<td>Graph Traversal</td>
</tr>
<tr>
<td>10</td>
<td>Dynamic Programming</td>
</tr>
<tr>
<td>11</td>
<td>Backtrack and Branch-and-Bound</td>
</tr>
<tr>
<td>12</td>
<td>Graphical Models</td>
</tr>
<tr>
<td>13</td>
<td>Finite State Machines</td>
</tr>
</tbody>
</table>

http://view.eecs.berkeley.edu/wiki/Dwarf_Mine
Outline

1. HPC and parallelism
2. Python
3. Language and standard libraries
4. External Libraries/projects
5. IPython for parallelism
Python: highly dynamic language

Python is strongly but dynamically typed

```python
In [5]: x=42
    ...: print 'x=', x, 'type(x)=', type(x), 'x*2=', x*2
    ...: print
    ...: x="Flexible!"
    ...: print 'x=', x, 'type(x)=', type(x), 'x*2=', x*2
    ...:
    x= 42 type(x)= <type 'int'> x*2= 84
```
```
In [6]: x="Flexible!"
    ...: print 'x=', x, 'type(x)=', type(x), 'x*2=', x*2
```
```python
x= Flexible! type(x)= <type 'str'> x*2= Flexible!Flexible!
```

- Types are rich but removed from the hardware
  - ints: arbitrary precision
  - floats: wrapped C doubles
  - lists, tuples: far from double*

- **Very simple, stack-based** Virtual Machine
  - minimal optimization
  - VM overview: [http://www.troeger.eu/teaching/pythonvm08.pdf](http://www.troeger.eu/teaching/pythonvm08.pdf)
The Python VM: at your fingertips

In [5]: import dis  # Python’s disassembler
   ...: src='''
   ...: s=0
   ...: for i in range(10):
   ...:     s += i
   ...: '''
   ...: code = compile(src, '<input>', 'exec')
   ...: print dis.dis(code)

<table>
<thead>
<tr>
<th></th>
<th>opcode</th>
<th>arg</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>LOAD_CONST</td>
<td>0 (0)</td>
</tr>
<tr>
<td>3</td>
<td>STORE_NAME</td>
<td>0 (s)</td>
</tr>
<tr>
<td>3</td>
<td>SETUP_LOOP</td>
<td>30 (to 39)</td>
</tr>
<tr>
<td>9</td>
<td>LOAD_NAME</td>
<td>1 (range)</td>
</tr>
<tr>
<td>12</td>
<td>LOAD_CONST</td>
<td>1 (10)</td>
</tr>
<tr>
<td>15</td>
<td>CALL_FUNCTION</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>GET_ITER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FOR_ITER</td>
<td>16 (to 38)</td>
</tr>
<tr>
<td>22</td>
<td>STORE_NAME</td>
<td>2 (i)</td>
</tr>
<tr>
<td>4</td>
<td>LOAD_NAME</td>
<td>0 (s)</td>
</tr>
<tr>
<td>28</td>
<td>LOAD_NAME</td>
<td>2 (i)</td>
</tr>
<tr>
<td>31</td>
<td>INPLACE_ADD</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>STORE_NAME</td>
<td>0 (s)</td>
</tr>
<tr>
<td>35</td>
<td>JUMP_ABSOLUTE</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>POP_BLOCK</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOAD_CONST</td>
<td>2 (None)</td>
</tr>
<tr>
<td>42</td>
<td>RETURN_VALUE</td>
<td></td>
</tr>
</tbody>
</table>
In [5]: import dis  # Python’s disassembler

...: src='''
...: s=0
...: for i in range(10):
...:     s += i
...: '''

...: code = compile(src, '<input>', 'exec')

...: print dis.dis(code)
Multiple implementations of the Virtual Machine:

- CPython: pure C, ‘reference’
- IronPython: .NET
- Jython: Java

Their threading behaviors differ, I’ll focus on CPython.

Native threads supported, but of limited use.

**Global interpreter lock (GIL):** only one thread can modify any python data structure.

No language-specific primitives for parallelism.
The infamous Global Interpreter Lock in CPython

- Historical reasons, simplicity of implementation
- All attempts at removing it have failed
  - 2x loss of performance is not acceptable
- Threads only good for i/o bound tasks.
- Mostly useless for CPU-bound ones.
- Can operate on pre-allocated arrays, but:
  - code must be in C/C++/Fortran/Cython
  - be very careful with locking if code is not atomic at Python level

The best possible reference on the GIL: David Beazley's work

http://www.dabeaz.com/GIL
The infamous Global Interpreter Lock in CPython

- Historical reasons, simplicity of implementation
- All attempts at removing it have failed
  - 2x loss of performance is not acceptable
- Threads only good for i/o bound tasks.
- Mostly useless for CPU-bound ones.
- Can operate on pre-allocated arrays, but:
  - code must be in C/C++/Fortran/Cython
  - be very careful with locking if code is not atomic at Python level

The best possible reference on the GIL: David Beazley's work

http://www.dabeaz.com/GIL
With these limitations, why should you care?

- Very dynamic, introspective language

```python
In [13]: def f(x, y=1, **kw):
   ...:     """A docstring""
   ...:     return x+y

In [14]: f.func_code
Out[14]: <code object f at 0xac8cec0, file "<ipython console>",
  line 1>

In [15]: f.func_defaults
Out[15]: (1,)
```

- It’s open source: the perfect playground
  - Create a modified VM if you want
  - Google’s Unladen Swallow:
    - http://code.google.com/p/unladen-swallow

- It’s use in numerical/scientific computing is exploding
  - There’s a real need and much to be done.
  - Your ideas will have a real impact!
  - GPUs, local multicore, clusters... even large scale supercomputing?
Parallelism in Python

- **In-process (mind the GIL)**
  - Data parallelism with threaded libraries
  - Numpy/scipy can use a threaded ATLAS
  - Numexpr: a 'numpy VM'
  - Theano: a library that thinks it's a compiler
  - GPU-based solutions: PyCuda/PyOpenCL, scikits.cuda.
  - Hand-written threaded code...

- **Out-of-process**
  - The multiprocessing module
  - Python futures
  - Communicating Sequential Processes, ParallelPython, ... many more
  - IPython (I’m obviously biased)
Outline

1. HPC and parallelism
2. Python
3. Language and standard libraries
4. External Libraries/projects
5. IPython for parallelism
Multiprocessing
Module: multiprocessing

- Built-in since version 2.6 (available for earlier versions)
- An API that closely follows the threading API, but using processes
- Useful high-level objects
  - Process, Process pool, Namespaces, Listeners, ...
- Uses fork() on posix (hence there are some limitations)

A simple example

```python
from multiprocessing import Process

def f(name):
    print 'hello', name

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```
Python futures
In Python 3.2 as concurrent.futures

- High-level interface for asynchronously executing callables.
- Executors and Futures are the key objects

A simple example

```python
from shutil import copy
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(copy, 'src1.txt', 'dest1.txt')  # returns a Future
    e.submit(copy, 'src2.txt', 'dest2.txt')
    e.submit(copy, 'src3.txt', 'dest3.txt')
    e.submit(copy, 'src3.txt', 'dest4.txt')
```

Futures have useful methods:

- `f.cancel()`
- `f.running()`
- `f.result(timeout=None)`
- `f.add_done_callback(func)`
import time

def timed(func):
    def wrapper(n, **kw):
        st = time.clock()
        out = func(n, **kw)
        print "Time used: %.2f s" % (time.clock()-st)
        return out
    return wrapper

@timed
def ssq(n):
    "Sum of squares"
    return sum(i**2 for i in range(n))

Produces

In [3]: ssq(100000)
Time used: 0.12 s
Out[3]: 333328333350000L

In [4]: ssq(1000000)
Time used: 1.84 s
Out[4]: 333332833333500000L
Decorators normally return a modified function...

But they can do whatever they want!

```python
def funnydeco(func):
    return 'Hi, I am a decorator...'

@funnydeco
def f(x):
    return x+1
```

This decorator produces:

```text
In [2]: f(10)
Traceback (most recent call last):
  File "<ipython console>" , line 1, in <module>
TypeError: 'str' object is not callable

In [3]: print f
Hi, I am a decorator...
```
What does this have to do with parallelism?

Consider a simple pair of 'loop body' and 'loop summary' functions:

```python
def do_work(data, i):
    return data[i]/2

def summarize(results):
    return sum(results)
```

and some 'dataset' (here just a list of 10 numbers)

```python
count = 10
data = [3.0*j for j in range(count)]
```

that has to be processed, done here with a serial function:

```python
def loop_serial():
    results = [None]*count

    for i in range(count):
        results[i] = do_work(data, i)

    return summarize(results)
```
Now let’s look for clean syntax to do this in parallel...

```python
def for_each(iterable):
    """This decorator-based loop does a normal serial run.
    But in principle it could be doing the dispatch remotely""
    def call(func):
        map(func, iterable)  # This could be IPython’s parallel map
        # or a gpu dispatch...
    return call
```

This is the actual code of the decorator-based loop:

```python
def loop_deco():
    results = [None]*count

    @for_each(range(count))
    def loop(i):
        results[i] = do_work(data, i)

    return summarize(results)
```

Validate that both versions really do the same thing

```python
In [34]: assert loop_serial() == loop_deco()
    ...: print 'OK'
OK
```
Compare normal and decorator based syntax

The serial loop (just the body of the loop)

```python
for i in range(count):
    results[i] = do_work(data, i)
```

The equivalent part in the decorator version

```python
@for_each(range(count))
def loop(i):
    results[i] = do_work(data, i)
```

Decorator benefits

- A named closure
- With controlled access to parameters
- With access to enclosing scope
- With optional return values

This provides semantics extremely similar to Apple’s *Grand Central Dispatch* (and their GCC extensions that go along with GCD)
Outline

1. HPC and parallelism
2. Python
3. Language and standard libraries
4. External Libraries/projects
5. IPython for parallelism
Numpy and Scipy: ‘Out of the box’ parallelism?

- Not great...

- Can be built against a threaded ATLAS or the Intel Math Kernel Library (MKL)
  - This can give multithreaded support to many linear algebra operations.

- Manual effort with C/Fortran + OpenMP can give you some gains...
  - but with a fair amount of pain
Numexpr
An expression compiler for numpy

Approach
- Compile Numpy expressions to equivalent Python code...
- Block operations carefully
- execute on a special-purpose mini-VM (written in C)

Benefits
- Reduce the use of temporaries.
- Be cache-friendly.
- Support threads natively for all operations.
- Support Intel Vector Math Library and MKL.
### Evaluating simple expressions

```python
>>> import numpy as np
>>> import numexpr as ne

>>> a = np.arange(1e6)  # Choose large arrays for high performance
>>> b = np.arange(1e6)

>>> ne.evaluate("a + 1")  # a simple expression
array([ 1.00000000e+00, 2.00000000e+00, 3.00000000e+00, ..., 9.99998000e+05, 9.99999000e+05, 1.00000000e+06])

>>> ne.evaluate('a*b-4.1*a > 2.5*b')  # a more complex one
array([False, False, False, ..., True, True, True], dtype=bool)
```
Numexpr timings

Comparisons to Numpy and thread usage

```python
>>> timeit a**2 + b**2 + 2*a*b
10 loops, best of 3: 35.9 ms per loop

>>> ne.set_num_threads(1) # use 1 thread (on a 6-core machine)

>>> timeit ne.evaluate("a**2 + b**2 + 2*a*b")
100 loops, best of 3: 9.28 ms per loop    # 3.9x faster than NumPy

>>> ne.set_num_threads(4) # use 4 threads (on a 6-core machine)

>>> timeit ne.evaluate("a**2 + b**2 + 2*a*b")
100 loops, best of 3: 4.17 ms per loop    # 8.6x faster than NumPy
```
PiCloud
High level parallelism on an EC2 backend, as a service - http://picloud.com

Note: PiCloud is not open-source. I’ve only seen demos of it, I haven’t used it.
Theano
http://deeplearning.net/software/theano

A library that thinks it’s the child of a compiler and a Computer Algebra System

- Declare and construct mathematical expressions (including numpy)
- Emit highly optimized code for them:
  - use of GPU for computations
  - constant folding
  - merging of similar subgraphs, to avoid redundant calculation
  - arithmetic simplification (e.g. \( x \times y / x \rightarrow y \), \( -x \rightarrow x \))
  - inserting efficient BLAS operations (e.g. GEMM) in a variety of contexts
  - using inplace operations wherever it does not interfere with aliasing
  - loop fusion for elementwise sub-expressions
  - ...more
Declaring a simple function

```python
import theano
from theano import tensor

# declare two symbolic floating-point scalars
a = tensor.dscalar()
b = tensor.dscalar()

# create a simple expression
c = a + b

# convert the expression into a callable object that takes (a,b) values as input and computes a value for c
f = theano.function([a,b], c)

# bind 1.5 to 'a', 2.5 to 'b', and evaluate 'c'
assert 4.0 == f(1.5, 2.5)
```

SciPy 2010 presentation: **Transparent GPU Computing with Theano**
Outline

1. HPC and parallelism
2. Python
3. Language and standard libraries
4. External Libraries/projects
5. IPython for parallelism
Parallel computing: **fully interactive**
- development, debugging, testing, execution, monitoring,…

Easy things should be easy, difficult things possible

Make parallel computing **collaborative**

More dynamic model for **load balancing and fault tolerance**

Seamless integration with other tools: plotting/visualization, system shell.

Also want to keep the benefits of traditional approaches:
- Should integrate with threads/MPI if appropriate
- Should be easy to integrate compiled code and libraries

Support many types of parallelism
Network-aware IPython

The "IPython VM"

Kernel Controller
- Non-blocking
- World-visible

Kernel Engine
**MAY BLOCK!**

Twisted

Networks
IPython’s architecture
Easy reuse and distribution of existing serial (‘normal’) codes.

High-level abstractions for ‘embarrassingly parallel’ problems.

Direct execution of code over the network: multiengine interface.
Out-of-the box task farming tools: task interface.

Task farming system is “low-latency” (not in the Myrinet sense...)

- can be integrated into more complex codes.

Implement any approach to parallelism you want:

- Synchronous or asynchronous execution of code on nodes.
- Task farming.
- Traditional Message Passing (MPI).
- Integrate hybrid codes.
- BYO.

Actively developed.
Use cases

**Multicore**
- Run controller, engines, client on the same system.
- Zero setup
- Can continue using an active IPython session

```
$ ipcluster local -n 4
```

**Cluster with MPI: existing C/Fortran MPI code**
- Wrap for python use (mpi4py)
- Start controller and engines with MPI/PBS support
- Connect to Controller from IPython session and use interactively

```
$ ipcluster mpirun -n 64 --mpi=mpi4py
$ ipcluster pbs -n 64 --pbs-script=myscript.sh
```
Use cases

### Multicore
- Run controller, engines, client on the same system.
- Zero setup
- Can continue using an active IPython session

$ ipcluster local -n 4

### Cluster with MPI: existing C/Fortran MPI code
- Wrap for python use (mpi4py)
- Start controller and engines with MPI/PBS support
- Connect to Controller from IPython session and use interactively

$ ipcluster mpirun -n 64 --mpi=mpi4py
$ ipcluster pbs -n 64 --pbs-script=myscript.sh
Vision: visual programming
Michel Sanner, Scripps Institute, La Jolla.
IPVision: visual distributed computing
Michel Sanner, Jose Unpingco, Ananth Devulapalli [Ohio Supercomputing Center/OSU]
Some technical notes

- Networking: **Twisted**
  - High-level interfaces: no need to learn Twisted.
- RPC: Twisted’s **foolscap**
- **Security**: foolscap supports SSL (pyOpenSSL) and a capabilities model.
  - Review/improvements welcome, we’re not security experts!
- MPI support is there, use **mpi4py** bindings.
- Integration with queuing systems, better process control coming...
Distributed symbolic computing

**Sympy: interactive distributed factorization of symbolic polynomials**

In [1]: \texttt{import sympy}

In [2]: \texttt{def factorit(n):}
   ...: \texttt{x = sympy.var(\textquoteleft x\textquoteright )}
   ...: \texttt{return sympy.factor(x**n-1, x)}

In [3]: factorit(5)
Out[3]: \texttt{-(1 - x)*(1 + x + x**2 + x**3 + x**4)}

In [4]: \texttt{from IPython.kernel import client}
   ...: mec = client.MultiEngineClient()
   ...: mec.execute(\textquoteleft import sympy\textquoteright )
   ...: f = mec.map(factorit, range(100, 110))

In [5]: f[0]
Out[5]: \texttt{-(1 + x)*(1 + x**2)*(1 - x)*(1 + x + x**2 + x**3 + x**4)*}
   \texttt{*(1 - x + x**2 - x**3 + x**4)*(1 + x**5 + x**10 + x**15 + x**20)*}
   \texttt{*(1 - x**5 + x**10 - x**15 + x**20)*(1 - x**10 + x**20 - x**30 + x**40)*}
   \texttt{(1 - x**2 + x**4 - x**6 + x**8)}

Replace MultiEngineClient by TaskClient to get load balancing.
“Sockets done right”

- The socket library that acts as a concurrency framework.
- Fast enough for clustered products and supercomputing.
- Asynch I/O for scalable multicore message-passing apps.
- Python bindings in Cython (Brian Granger, Min RK)
- Python bindings run messaging in native threads - no GIL
- Abstractions are at the message delivery level, not the raw-bytes level.
- Socket types encapsulate messaging patterns

Interactive IPython on ØMQ

- Kernel raw_input
- Requests to kernel
- Kernel output broadcast
- Request/Reply direction
- Lots of Sockets
- 1 Socket = 1 type of action
- More complicated picture
- Simpler code
Performance: raw throughput
Send No-op tasks as fast as possible, wait for results
Performance: arrays
Echo 16 random arrays of given size

16 echo np array

Tasks/s

Size (B)

zmq
lru
twisted
sent
Thank you!

Paul Ivanov on GPU programming next...

Some good resources:

http://parlab.eecs.berkeley.edu/

http://cs.nyu.edu/courses/fall10/G22.2945-001/