Efficient Programming for Parallel Python

Mike McKerns

June 9, 2012

Start with Good Coding Practices

ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

Brian Kernighan

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"

- We all write buggy code. Accept it. Deal with it.
- Write your code with testing and debugging in mind.
- Keep It Simple, Stupid (KISS).
  - What is the simplest thing that could possibly work?
- Don't Repeat Yourself (DRY).
  - Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.
  - Constants, algorithms, etc...
- Try to limit interdependencies of your code. (Loose Coupling)
- Give your variables, functions and modules meaningful names (not mathematics names)

pyflakes: fast static analysis

ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

…catch bugs even before you run your code

There are several static analysis tools in Python; to name a few: pylint, pychecker, and pyflakes. Here we focus on pyflakes, which is the simplest tool.

- **Fast, simple**
- Detects syntax errors, missing imports, typos on names.

Integrating pyflakes in your editor is highly recommended, it **does yield productivity gains**.

debugging workflow

If you do have a non trivial bug, this is when debugging strategies kick in. There is no silver bullet. Yet, strategies help:

For debugging a given problem, the favorable situation is when the problem is isolated in a small number of lines of code, outside framework or application code, with short modify-run-fail cycles

1. Make it fail reliably. Find a test case that makes the code fail every time.
2. Divide and Conquer. Once you have a failing test case, isolate the failing code.
    - Which module.
    - Which function.
    - Which line of code.

    => isolate a small reproducible failure: a test case
3. Change one thing at a time and re-run the failing test case.
4. Use the debugger to understand what is going wrong.
5. Take notes and be patient. It may take a while.

Note: Once you have gone through this process: isolated a tight piece of code reproducing the bug and fix the bug using this piece of code, add the corresponding code to your test suite.

The python debugger

The python debugger, `pdb`: http://docs.python.org/library/pdb.html, allows you to inspect your code interactively.

Specifically it allows you to:

- View the source code.
- Walk up and down the call stack.
- Inspect values of variables.
- Modify values of variables.
- Set breakpoints.

print

Yes, `print` statements do work as a debugging tool. However to inspect runtime, it is often more efficient to use the debugger.

And …gdb

If you have a segmentation fault, you cannot debug it with pdb, as it crashes the Python interpreter before it can drop in the debugger. Similarly, if you have a bug in C code embedded in Python, pdb is useless. For this we turn to the gnu debugger, gdb, available on Linux.

```
$ gdb python
...
(gdb) run segfault.py
Starting program: /usr/bin/python segfault.py
[Thread debugging using libthread_db enabled]

Program received signal SIGSEGV, Segmentation fault.
_strided_byte_copy (dst=0x8537478 "\360\343G", outstrides=4, src=
    0x86c0690 <Address 0x86c0690 out of bounds>, instrides=32, N=3,
    elsize=4)
        at numpy/core/src/multiarray/ctors.c:365
365             _FAST_MOVE(Int32);
(gdb)
```

We get a segfault, and gdb captures it for post-mortem debugging in the C level stack (not the Python call stack). We can debug the C call stack using gdb's commands:

code optimization

ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

Donald Knuth

"Premature optimization is the root of all evil"

1. Make it work: write the code in a simple **legible** ways.
2. Make it work reliably: write automated test cases, make really sure that your algorithm is right and that if you break it, the tests will capture the breakage.
3. Optimize the code by profiling simple use-cases to find the bottlenecks and speeding up these bottleneck, finding a better algorithm or implementation. Keep in mind that a trade off should be found between profiling on a realistic example and the simplicity and speed of execution of the code. For efficient work, it is best to work with profiling runs lasting around 10s.

No optimization without measuring!

- **Measure:** profiling, timing
- You'll have surprises: the fastest code is not always what you think

timeit

ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

In IPython, use `timeit` (http://docs.python.org/library/timeit.html) to time elementary operations:

```
In [1]: import numpy as np

In [2]: a = np.arange(1000)

In [3]: %timeit a ** 2
100000 loops, best of 3: 5.73 us per loop

In [4]: %timeit a ** 2.1
1000 loops, best of 3: 154 us per loop

In [5]: %timeit a * a
100000 loops, best of 3: 5.56 us per loop
```

Use this to guide your choice between strategies.

Note: For long running calls, using `%time` instead of `%timeit`; it is less precise but faster

...more timing

Useful when you have a large program to profile, for example the following file:

```
import numpy as np
from scipy import linalg
from ica import myfunc

def test():
    data = np.random.random((5000, 100))
    u, s, v = linalg.svd(data)
    pca = np.dot(u[:10, :], data)
    results = myfunc(pca.T)

test()
```

In IPython we can time the script:

```
In [1]: %run -t demo.py

IPython CPU timings (estimated):
    User   :     14.3929 s.
    System:    0.256016 s.
```

…and profiling

and profile it:

```
In [2]: %run -p demo.py

         916 function calls in 14.551 CPU seconds

Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1   14.457   14.457   14.479   14.479 decomp.py:849(svd)
     1    0.054    0.054    0.054    0.054 {method 'random_sample' of 'mtrand.Ra
     1    0.017    0.017    0.021    0.021 function_base.py:645(asarray_chkfinit
    54    0.011    0.000    0.011    0.000 {numpy.core._dotblas.dot}
     2    0.005    0.002    0.005    0.002 {method 'any' of 'numpy.ndarray' obje
     6    0.001    0.000    0.001    0.000 ica.py:195(gprime)
     6    0.001    0.000    0.001    0.000 ica.py:192(g)
    14    0.001    0.000    0.001    0.000 {numpy.linalg.lapack_lite.dsyevd}
    19    0.001    0.000    0.001    0.000 twodim_base.py:204(diag)
     1    0.001    0.001    0.008    0.008 ica.py:69(_ica_par)
     1    0.001    0.001   14.551   14.551 {execfile}
   107    0.000    0.000    0.001    0.000 defmatrix.py:239(__array_finalize__)
     7    0.000    0.000    0.004    0.001 ica.py:58(_sym_decorrelation)
     7    0.000    0.000    0.002    0.000 linalg.py:841(eigh)
   172    0.000    0.000    0.000    0.000 {isinstance}
     1    0.000    0.000   14.551   14.551 demo.py:1(<module>)
    29    0.000    0.000    0.000    0.000 numeric.py:180(asarray)
    35    0.000    0.000    0.000    0.000 defmatrix.py:193(__new__)
```

The profiler is great: it tells us which function takes most of the time, but not where it is called.

For this, we use the line_profiler: in the source file, we decorate a few functions that we want to inspect with `@profile` (no need to import it):

```
@profile
def test():
    data = np.random.random((5000, 100))
    u, s, v = linalg.svd(data)
    pca = np.dot(u[:10, :], data)
    results = myfunc(pca.T)
```

Then we run the script using the kernprof.py program, with switches `-` and `-v`:

```
~ $ kernprof.py -l -v demo.py

Wrote profile results to demo.py.lprof
Timer unit: 1e-06 s

File: demo.py
Function: test at line 5
Total time: 14.2793 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     5                                           @profile
     6                                           def test():
     7         1        19015  19015.0      0.1      data = np.random.random((500(
     8         1     14242163 14242163.0     99.7      u, s, v = linalg.svd(data)
     9         1        10282  10282.0      0.1      pca = np.dot(u[:10, :], data)
    10         1         7799   7799.0      0.1      results = myfunc(pca.T)
```

Make sure to try alternatives

ENTHOUGHT

For certain algorithms, many of the bottlenecks will be linear algebra computations. In this case, using the right function to solve the right problem is key. For instance, an eigenvalue problem with a symmetric matrix is easier to solve than with a general matrix. Also, most often, you can avoid inverting a matrix and use a less costly (and more numerically stable) operation.

```
In [3]: %timeit np.linalg.svd(data)
1 loops, best of 3: 14.5 s per loop

In [4]: from scipy import linalg

In [5]: %timeit linalg.svd(data)
1 loops, best of 3: 14.2 s per loop

In [6]: %timeit linalg.svd(data, full_matrices=False)
1 loops, best of 3: 295 ms per loop

In [7]: %timeit np.linalg.svd(data, full_matrices=False)
1 loops, best of 3: 293 ms per loop
```

For a high-level view of the problem, a good understanding of the maths behind the algorithm helps. However, it is not uncommon to find simple changes, like **moving computation or memory allocation outside a for loop**, that bring in big gains.

looping constructs

```
newlist = []
for word in oldlist:
    newlist.append(word.upper())
```

you can use `map` to push the loop from the interpreter into compiled C code:

```
newlist = map(str.upper, oldlist)
```

List comprehensions were added to Python in version 2.0 as well. They provide a syntactically more compact and more efficient way of writing the above for loop:

```
newlist = [s.upper() for s in oldlist]
```

Generator expressions were added to Python in version 2.4. They function more-or-less like list comprehensions or `map` but avoid the overhead of generating the entire list at once. Instead, they return a generator object which can be iterated over bit-by-bit:

```
iterator = (s.upper() for s in oldlist)
```

avoid “.”

ENTHOUGHT

Avoiding dots…

Suppose you can't use `map` or a list comprehension? You may be stuck with the for loop. The for loop example has another inefficiency. Both `newlist.append` and `word.upper` are function references that are reevaluated each time through the loop. The original loop can be replaced with:

```
upper = str.upper
newlist = []
append = newlist.append
for word in oldlist:
    append(upper(word))
```

This technique should be used with caution. It gets more difficult to maintain if the loop is large. Unless you are intimately familiar with that piece of code you will find yourself scanning up to check the definitions of `append` and `upper`.

…and shop locally

Local Variables

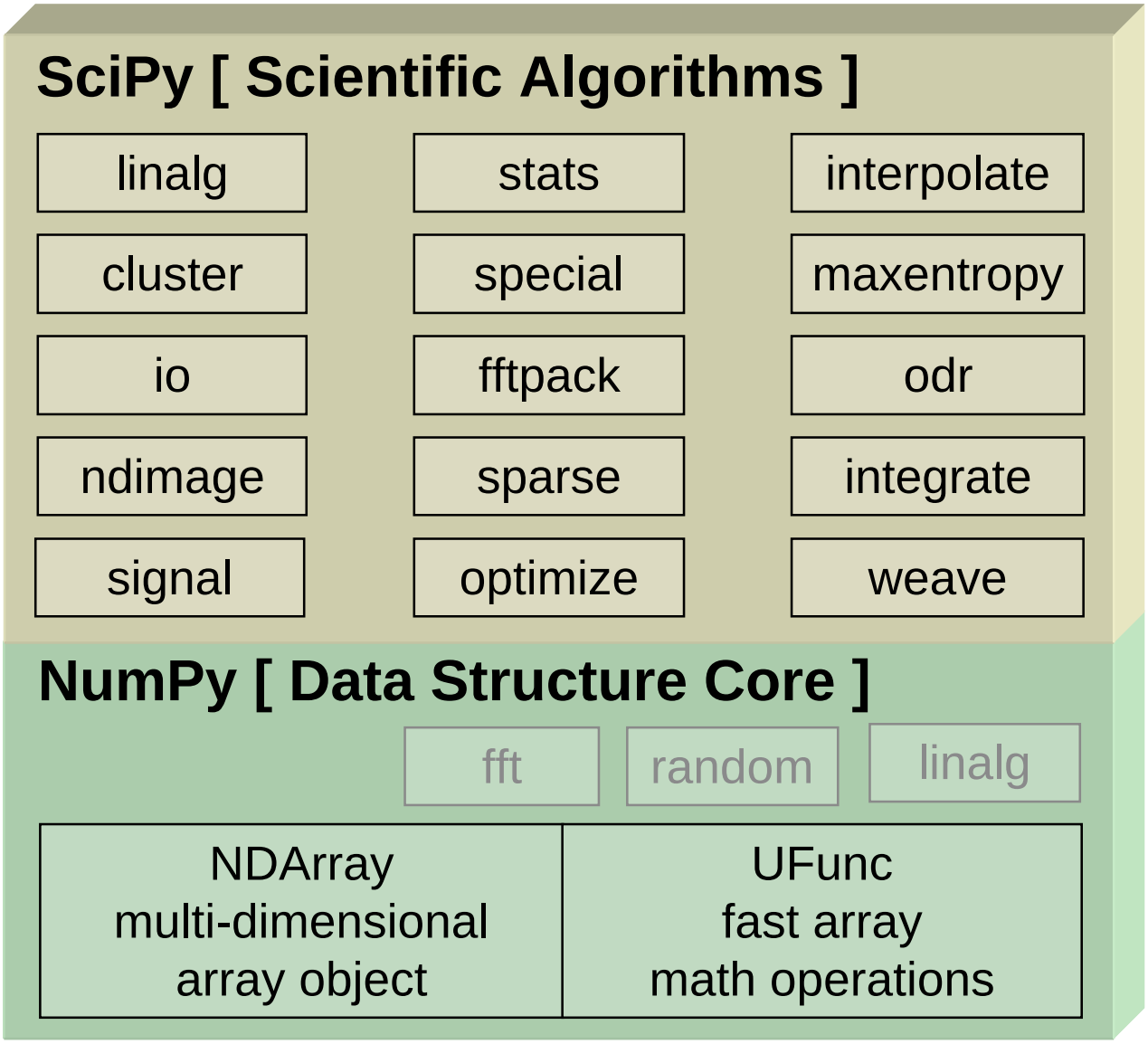
The final speedup available to us for the non-`map` version of the `for` loop is to use local variables wherever possible. If the above loop is cast as a function, `append` and `upper` become local variables. Python accesses local variables much more efficiently than global variables.

```
def func():
    upper = str.upper
    newlist = []
    append = newlist.append
    for word in oldlist:
        append(upper(word))
    return newlist
```

Import Statement Overhead

`import` statements can be executed just about anywhere. It's often useful to place them inside functions to restrict their visibility and/or reduce initial startup time. Although Python's interpreter is optimized to not import the same module multiple times, repeatedly executing an import statement can seriously affect performance in some circumstances.

NumPy and SciPy



Array Calculation Methods

SUM FUNCTION

```
>>> a = array([[1,2,3],
               [4,5,6]], float)

# sum() defaults to adding up
# all the values in an array.
>>> sum(a)
21.

# supply the keyword axis to
# sum along the 0th axis
>>> sum(a, axis=0)
array([5., 7., 9.])

# supply the keyword axis to
# sum along the last axis
>>> sum(a, axis=-1)
array([6., 15.])
```

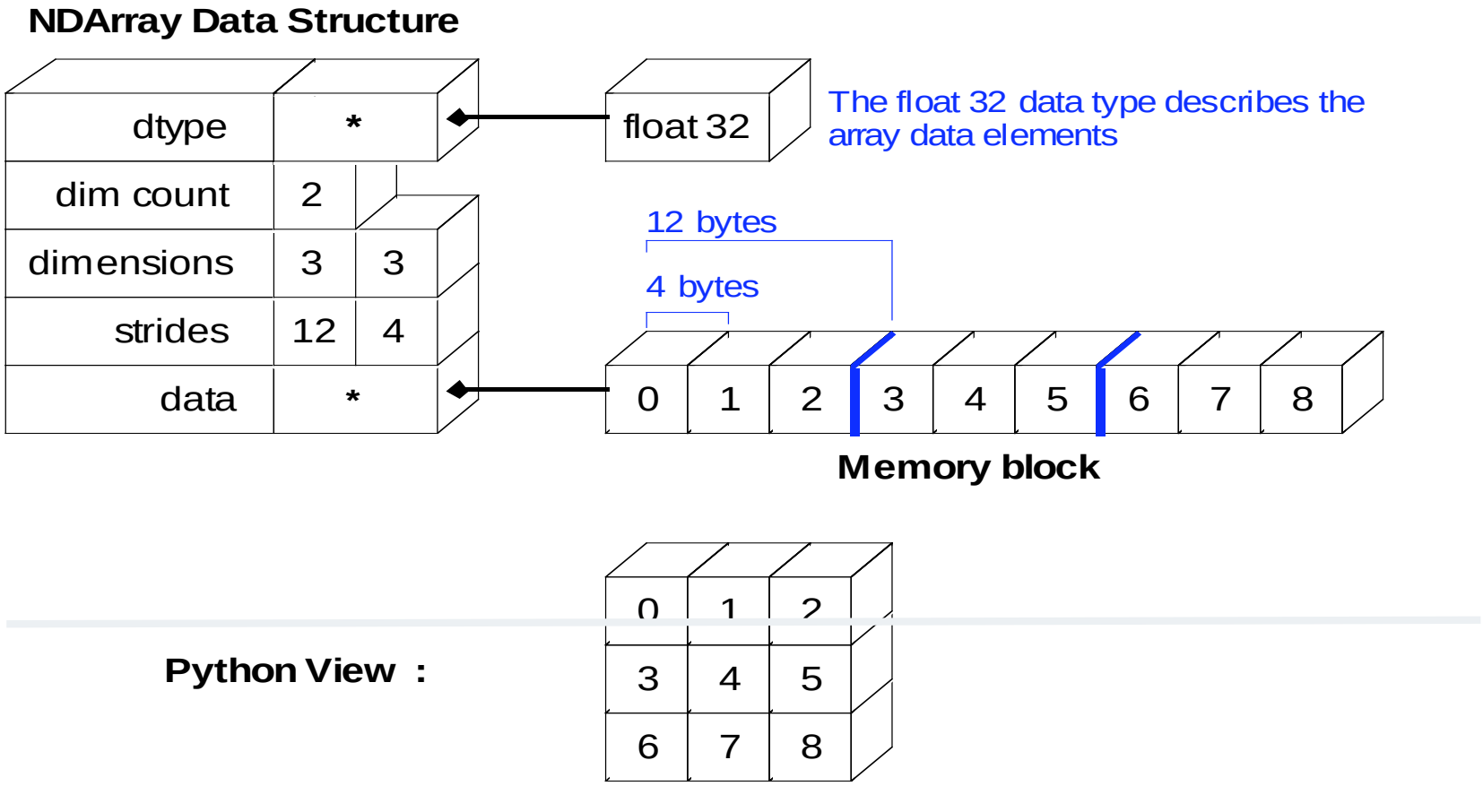
SUM ARRAY METHOD

```
# a.sum() defaults to adding
# up all values in an array.
>>> a.sum()
21.

# supply an axis argument to
# sum along a specific axis
>>> a.sum(axis=0)
array([5., 7., 9.])
```

PRODUCT

```
# product along columns
>>> a.prod(axis=0)
array([ 4., 10., 18.])

# functional form
>>> prod(a, axis=0)
array([ 4., 10., 18.])
```

Array Data Structure



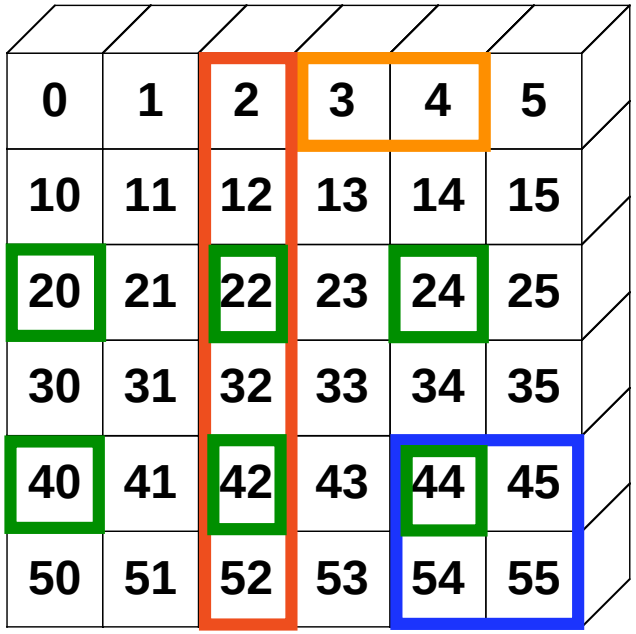
Array Slicing

SLICING WORKS MUCH LIKE STANDARD PYTHON SLICING

```
>>> a[0,3:5]
array([3, 4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])
```

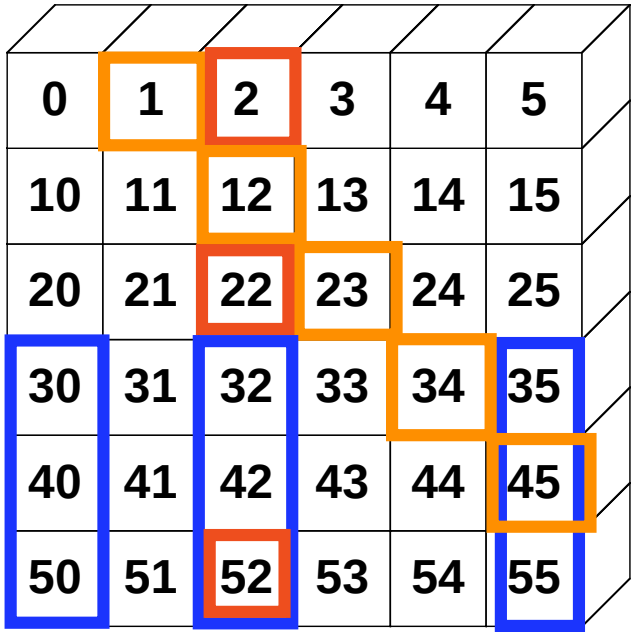
STRIDES ARE ALSO POSSIBLE

```
>>> a[2::2,::2]
array([[20, 22, 24],
       [40, 42, 44]])
```



Fancy Indexing in 2-D

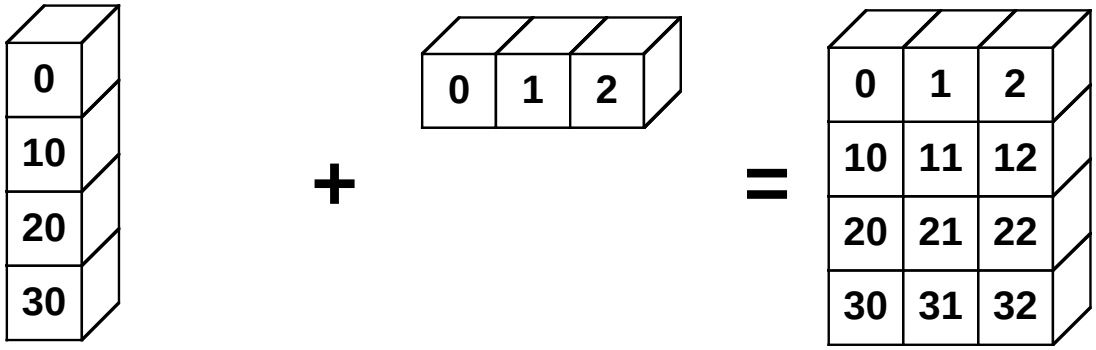
```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])

>>> a[3:,[0, 2, 5]]
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])

>>> mask = array([1,0,1,0,0,1],
                 dtype=bool)
>>> a[mask,2]
array([2,22,52])
```



Unlike slicing, fancy indexing creates copies instead of a view into original array.

Broadcasting in Action

```
>>> a = array((0,10,20,30))
>>> b = array((0,1,2))
>>> y = a[:, newaxis] + b
```



coding for speed

- **Vectorizing for loops**

  Find tricks to avoid for loops using numpy arrays. For this, masks and indices arrays can be useful.

- **Broadcasting**

  Use *broadcasting* to do operations on arrays as small as possible before combining them.

- **In place operations**

```
In [1]: a = np.zeros(1e7)

In [2]: %timeit global a ; a = 0*a
10 loops, best of 3: 111 ms per loop

In [3]: %timeit global a ; a *= 0
10 loops, best of 3: 48.4 ms per loop
```

note: we need `global a` in the timeit so that it work, as it is assigning to `a`, and thus considers it as a local variable.

more speed tips

- **Be easy on the memory: use views, and not copies**

  Copying big arrays is as costly as making simple numerical operations on them:

```
In [1]: a = np.zeros(1e7)

In [2]: %timeit a.copy()
10 loops, best of 3: 124 ms per loop

In [3]: %timeit a + 1
10 loops, best of 3: 112 ms per loop
```

- **Beware of cache effects**

  Memory access is cheaper when it is grouped: accessing a big array in a continuous way is much faster than random access. This implies amongst other things that **smaller strides are faster**

```
In [1]: c = np.zeros((1e4, 1e4), order='C')

In [2]: %timeit c.sum(axis=0)
1 loops, best of 3: 3.89 s per loop

In [3]: %timeit c.sum(axis=1)
1 loops, best of 3: 188 ms per loop
```

More numpy speed

ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

This is the reason why Fortran ordering or C ordering may make a big difference on operations:

```
In [5]: a = np.random.rand(20, 2**18)

In [6]: b = np.random.rand(20, 2**18)

In [7]: %timeit np.dot(b, a.T)
1 loops, best of 3: 194 ms per loop

In [8]: c = np.ascontiguousarray(a.T)

In [9]: %timeit np.dot(b, c)
10 loops, best of 3: 84.2 ms per loop
```

Note that copying the data to work around this effect may not be worth it:

```
In [10]: %timeit c = np.ascontiguousarray(a.T)
10 loops, best of 3: 106 ms per loop
```

Using numexpr can be useful to automatically optimize code for such effects.

…when all else fails

- **Use compiled code**

  The last resort, once you are sure that all the high-level optimizations have been explored, is to transfer the hot spots, i.e. the few lines or functions in which most of the time is spent, to compiled code. For compiled code, the preferred option is to use Cython: it is easy to transform exiting Python code in compiled code, and with a good use of the numpy support yields efficient code on numpy arrays, for instance by unrolling loops.

Hand Wrapping C/C++

The Wrapper Function

fact.h

```
#ifndef FACT_H
#define FACT_H
int fact(int n);
#endif
```

fact.c

```
#include "fact.h"
int fact(int n)
{
        if (n <=1) return 1;
        else return n*fact(n-1);
}
```

WRAPPER FUNCTION

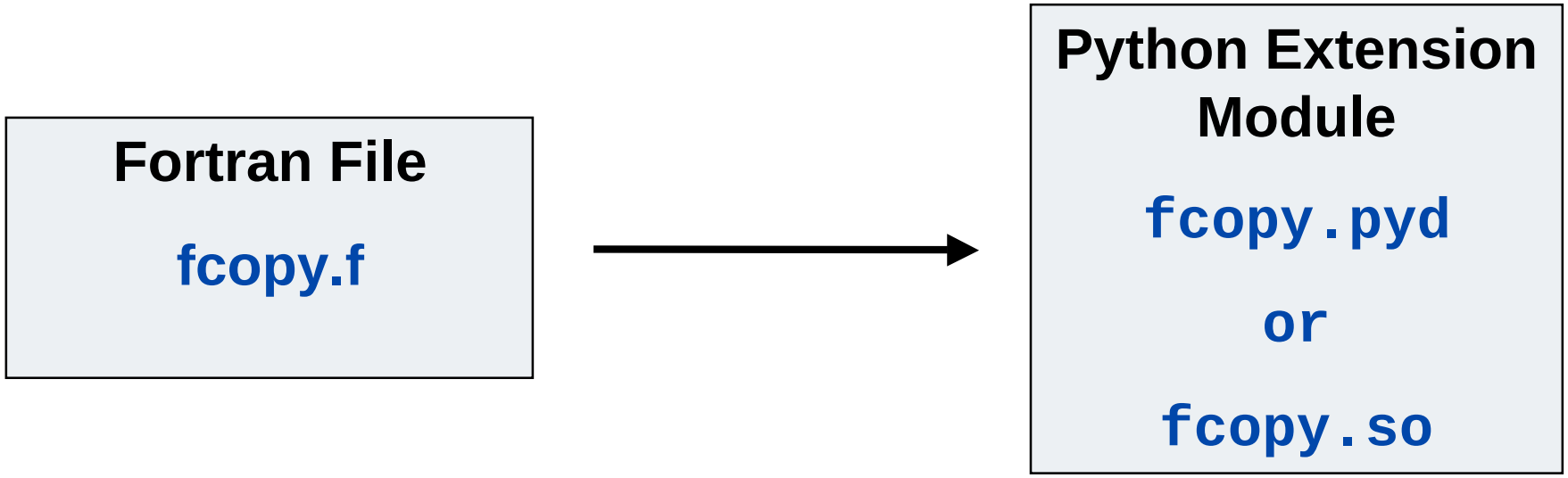
```
static PyObject* wrap_fact(PyObject *self, PyObject *args)
{
    /* Python-C data conversion */
    int n, result;
    if (!PyArg_ParseTuple(args, "i", &n))
        return NULL;
    /* C Function Call */
    result = fact(n);
    /* C->Python data conversion */
    return Py_BuildValue("i", result);
}
```

Complete Example

```
/* Must include Python.h before any standard headers! */
#include "Python.h"
#include "fact.h"

/* Define the wrapper functions exposed to Python (must be static) */
static PyObject* wrap_fact(PyObject *self, PyObject *args) {
    int n, result;
    /* Python->C Conversion */
    if (!PyArg_ParseTuple(args, "i", &n))
        return NULL;
    /* Call our function */
    result = fact(n);
    /* C->Python Conversion */
    return Py_BuildValue("i", result);
}
/* Method table declaring the names of functions exposed to Python */
static PyMethodDef ExampleMethods[] = {
    {"fact",  wrap_fact, METH_VARARGS, "Calculate the factorial of n"},
    {NULL, NULL, 0, NULL}     /* Sentinel */
};
/* Module initialization function called at "import example" */
PyMODINIT_FUNC initexample(void) {
        (void) Py_InitModule("example", ExampleMethods);  }
```

f2py

Simplest f2py Usage




```
f2py -c -m fcopy fcopy.f -compiler=mingw32
```

- Compile code and build an extension module
- Name the extension module fcopy
- Specify the Fortran file to use
- On Windows, specifying mingw32 uses gcc tool chain

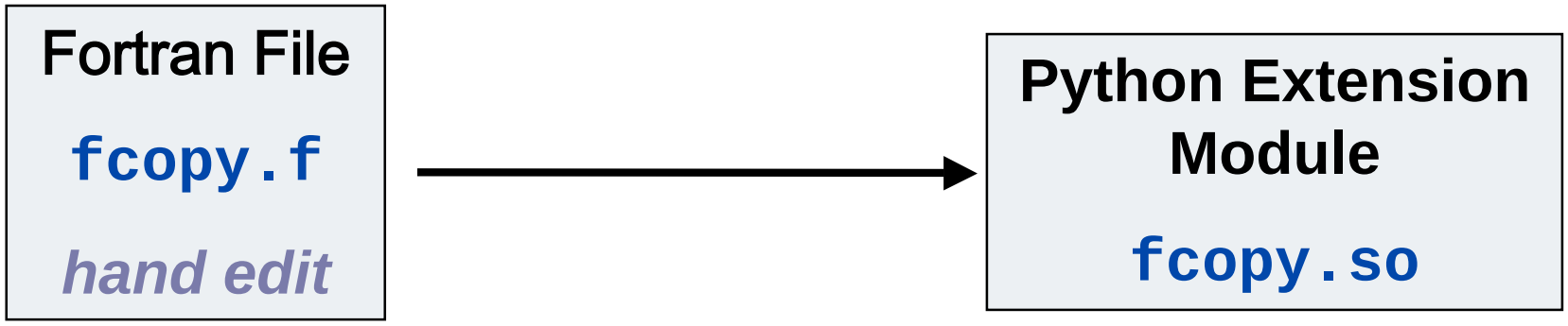
Simplest Usage Result

```
Fortran file fcopy.f
C
      SUBROUTINE FCOPY(AIN,N,AOUT)
C
      DOUBLE COMPLEX AIN(*)
      INTEGER N
      DOUBLE COMPLEX AOUT(*)
      DO 20 J = 1, N
         AOUT(J) = AIN(J)
20    CONTINUE
      END
```

```
>>> a = rand(1000) + 1j*rand(1000)
>>> b = zeros((1000,), dtype=complex128)
>>> fcopy.fcopy(a,1000,b)
>>> alltrue(a==b)
True
```

Looks exactly like
the Fortran —
but now it is callable
from Python

Simply Sophisticated



f2py –c fcopy.f –m fcopy

Compile code and build an extension module

Name the extension module fcopy

Simply Sophisticated

Directives help f2py interpret the source

```
Fortran file fcopy2.f
C
      SUBROUTINE FCOPY(AIN,N,AOUT)
C
CF2PY DOUBLE COMPLEX DIMENSION(N), INTENT(IN) :: AIN
CF2PY DOUBLE COMPLEX DIMENSION(N), INTENT(OUT) :: AOUT
CF2PY INTEGER, INTENT(HIDE), DEPEND(AIN) :: N=LEN(AIN)
      DOUBLE COMPLEX AIN(*)
      INTEGER N
      DOUBLE COMPLEX AOUT(*)
      DO 20 J = 1, N
         AOUT(J) = AIN(J)
 20   CONTINUE
      END
```

The resulting interface is more *Pythonic*

```
>>> import fcopy
>>> info(fcopy.fcopy)
fcopy - Function signature:
  aout = fcopy(ain)
Required arguments:
  ain : input rank-1 array('D') with
bounds (n)
Return objects:
  aout : rank-1 array('D') with bounds (n)
>>> a = rand(1000)
>>> import fcopy
>>> b = fcopy.fcopy(a)
```

Weave

weave

- `weave.blitz()`

Translation of NumPy array expressions to C/C++ for fast execution

- `weave.inline()`

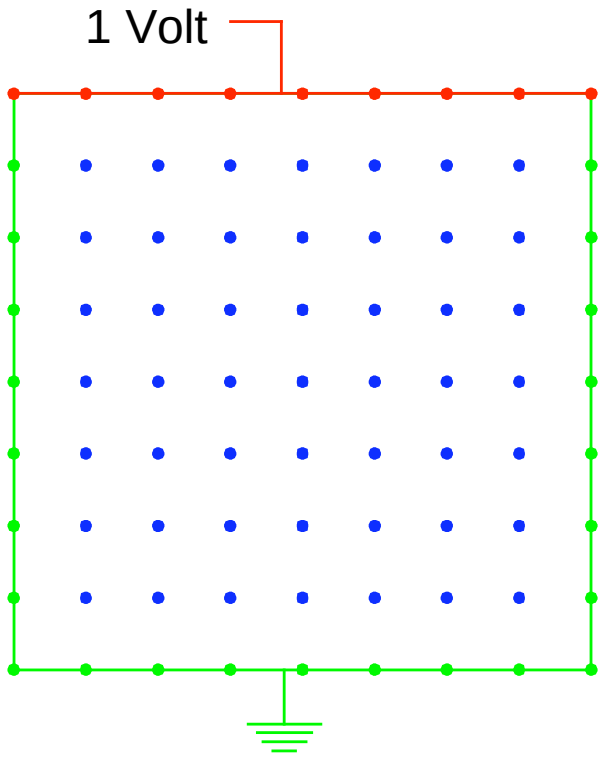
Include C/C++ code directly in Python code for on-the-fly execution

- `weave.ext_tools`

Classes for building C/C++ extension modules in Python

weave and Laplace's equation

Weave case study: An iterative solver for Laplace's Equation



PURE PYTHON **133 SECONDS**

```
for i in range(1, nx-1):
  for j in range(1, ny-1):
    tmp = u[i,j]
    u[i,j] = ((u[i-1, j] + u[i+1, j])*dy2 +
              (u[i, j-1] + u[i, j+1])*dx2)
             / (2.0*(dx2 + dy2))
    diff = u[i,j] - tmp
    err = err + diff**2
```

weave and Laplace's equation

USING NUMPY — 1.36 SECONDS

```
old_u = u.copy() # needed to compute the error.
u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
                 (u[1:-1, 0:-2] + u[1:-1, 2:])*dx2)
                * dnr_inv
err = sum(dot(old_u - u))
```

WEAVE.BLITZ — 0.56 SECONDS

```
old_u = u.copy() # needed to compute the error.
expr = """ \
       u[1:-1, 1:-1] =  ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
                         (u[1:-1, 0:-2] + u[1:-1, 2:])*dx2)
                       * dnr_inv
       """
weave.blitz(expr,size_check=0)
err = sum((old_u - u)**2)
```

weave and Laplace's equation

WEAVE.INLINE **0.25 SECONDS**

```
code = """
       #line 120 "laplace.py" (This is only useful for debugging)
       double tmp, err, diff;
       err = 0.0;
       for (int i=1; i<nx-1; ++i) {
         for (int j=1; j<ny-1; ++j) {
           tmp = u(i,j);
           u(i,j) = ((u(i-1,j) + u(i+1,j))*dy2 +
                     (u(i,j-1) + u(i,j+1))*dx2)*dnr_inv;
           diff = u(i,j) - tmp;
           err += diff*diff;
         }
       }
       return_val = sqrt(err);
       """
err = weave.inline(code, ['u','dx2','dy2','dnr_inv','nx','ny'],
                         type_converters = converters.blitz,
                         compiler = 'gcc',
                         extra_compile_args = ['-O3','-malign-double'])
```

Cython

What is Cython?

Cython is a Python-like language for writing extension modules. It lets you mix Python and C data types any way you want, and compiles to (reasonably) fast code.

http://www.cython.org/

It is a (friendly) fork of Pyrex:

http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex

Problem: Make This Fast!

```
def mandelbrot_escape(x, y, n):
    z_x = x
    z_y = y
    for i in range(n):
        z_x, z_y = z_x**2 - z_y**2 + x, 2*z_x*z_y + y
        if z_x**2 + z_y**2 >= 4.0:
            break
    else:
        i = -1
    return i

def generate_mandelbrot(xs, ys, n):
    d = empty(shape=(len(ys), len(xs)))
    for j in range(len(ys)):
        for i in range(len(xs)):
            d[j,i] = mandelbrot_escape(xs[i], ys[j], n)
    return d
```

Step 1: Add Type Information

Type information can be added to function signatures:

```
def mandelbrot_escape(double x, double y, int n):
    ...
def generate_mandelbrot(xs, ys, int n):
    ...
```

Variables can be declared to have a type using 'cdef':

```
def generate_mandelbrot(xs, ys, int n):
    cdef int i,j
    cdef int N = len(xs)
    cdef int M = len(ys)
    ...
```

Step 2: Use Cython C Functions

In Cython you can declare functions to be C functions using 'cdef' instead of 'def':

```
cdef int mandelbrot_escape(float x, float y, int n):
    ...
```

This makes the functions:

- Generate actual C functions, so they are much faster.
- Not visible to Python, but freely usable in your Cython module.

Arbitrary Python objects can still be passed in and out of C functions using the 'object' type.

Solution: This is Fast!

```
cdef int mandelbrot_escape(double x, double y, int n):
    cdef double z_x = x
    cdef double z_y = y
    cdef int i
    for i in range(n):
        z_x, z_y = z_x**2 - z_y**2 + x, 2*z_x*z_y + y
        if z_x**2 + z_y**2 >= 4.0:
            break
    else:
        i = -1
    return i
def generate_mandelbrot(xs, ys, int n):
    cdef int i,j
    cdef int N = len(xs)
    cdef int M = len(ys)
    d = empty(dtype=int, shape=(N, M))
    for j in range(M):
        for i in range(N):
            d[j,i] = mandelbrot_escape(xs[i], ys[j], n)
    return d
```

Step 3: Use NumPy in Cython

In our example, we are still using Python-level numpy calls to do our array indexing:

```
d[j,i] = mandelbrot_escape(xs[i], ys[j], n)
```

If we use the Cython interface to NumPy, we can declare our arrays to be C-level numpy extension types, and gain even more speed.

NumPy arrays are declared using a special buffer notation:

```
cimport numpy as np
...
cdef np.ndarray[int, ndim=2] my_array
```

You must declare both the type of the array, and the number of dimensions. All the standard numpy types are declared in the numpy cython declarations.

Solution 2: This is *Really* Fast!

mandel.pyx

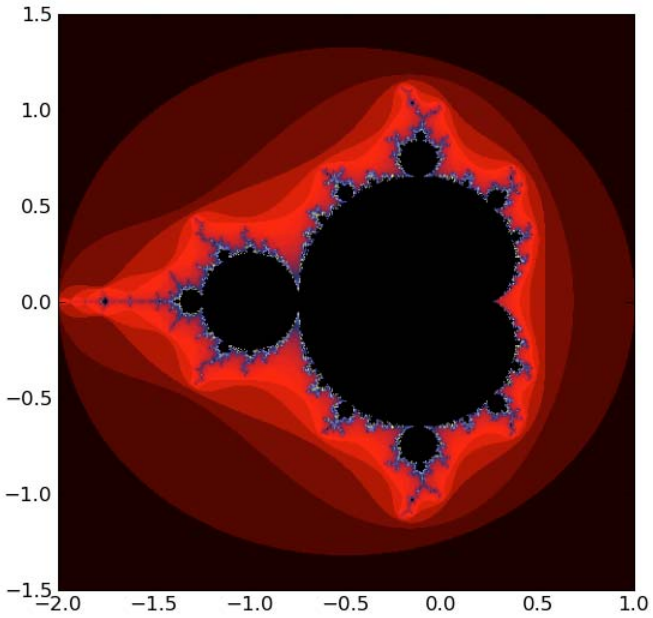
```
cimport numpy as np
...
def generate_mandelbrot(np.ndarray[double, ndim=1] xs,
        np.ndarray[double, ndim=1] ys, int n):
    cdef int i,j
    cdef int N = len(xs)
    cdef int M = len(ys)
    cdef np.ndarray[int, ndim=2] d = empty(dtype=int, shape=(N, M))
    for j in range(M):
        for i in range(N):
            d[j,i] = mandelbrot_escape(xs[i], ys[j], n)
    return d
```

setup.py

```
...
import numpy
...
ext = Extension("mandel", ["mandel.pyx"],
    include_dirs = [numpy.get_include()])
```

Conclusion

Solution	Time	Speed-up
Pure Python	74.3 s	x 1.0
Cython (Step 1)	0.291 s	x 255
Cython (Step 2)	0.213 s	x 350
Cython+Numpy (Step 3)	0.0397 s	x 1870



Timing performed on a dual core 2.33 GHz MacBook Pro with 2GB RAM using a 500x500 array and an escape time of n=100.

Batch Processing in Python

What's a second worth?

hera@llnl: 12,800 opteron cores

lobo@lanl:4352 opteron cores

coyote@lanl:2676 opteron cores

A model requiring 1 minute of compute time may be evaluated 10,000 times in a global optimization. We need to get results in less than 10,000 minutes (1 week).

"Realistic" models may require several days of runtime on a large parallel cluster.

Is it possible to do global optimization and rigorous sensitivity analysis on problems of this size?

(yes…)

pickle – Object Serialization

BASIC FUNCTIONS

`dump(obj, file, protocol=0)`

Write the pickled object to the file.

`load(file)`

Read and return a previously pickled object from the file.

`dumps(obj, protocol=0)`

Return the string containing the pickled object.

`loads(string)`

Unpickle the string and return the python object.

PROTOCOL

The `protocol` argument determines the format of the pickle string.

There are currently three different protocols which can be used for pickling.

- `protocol=0`: the original ASCII protocol, backwards compatible with earlier versions of Python.
- `protocol=1`: the old binary format, also compatible with earlier versions of Python.
- `protocol=2`: introduced in Python 2.3; it provides much more efficient pickling of new-style classes.

The constant `pickle.HIGHEST_PROTOCOL` holds the highest protocol version available.

Multiprocessing

What is Multiprocessing?

The multiprocessing module provides access to task-based parallel computing using separate Python processes.

Multiprocessing has several features:
•part of Python's standard library,
•cross-platform,
•low-level parallel constructs: locks, queues, pipes, semaphores, shared memory access
•higher-level constructs: task pools with automatic scheduling, shared memory object wrappers, functional and iterator-like design support, etc.

The multiprocessing module is useful for:
•batch computing, embarrassingly parallel computations
•parallel computing with minimal shared state
•GUI programming: do calculations in a background process
•many, many others.

`http://docs.python.org/library/multiprocessing.html`

Task Pools

`Pool()` objects provide a convenient way to perform embarrassingly parallel computations with automatic load balancing.

`Pool(processes=None, initializer=None, initargs=())`

`processes` — int, number of processes in pool, cpu_count() by default

`initializer` — callable, if not None, initializer(*initargs) is called by each process when starting.

Methods:

`map(func, iterable, chunksize=None)` — parallel equivalent of the map() builtin function. Blocks until result is ready.

`map_async(func, iterable, chunksize=None, callback=None)`

Same as **`map()`**, nonblocking. Returns a result object.

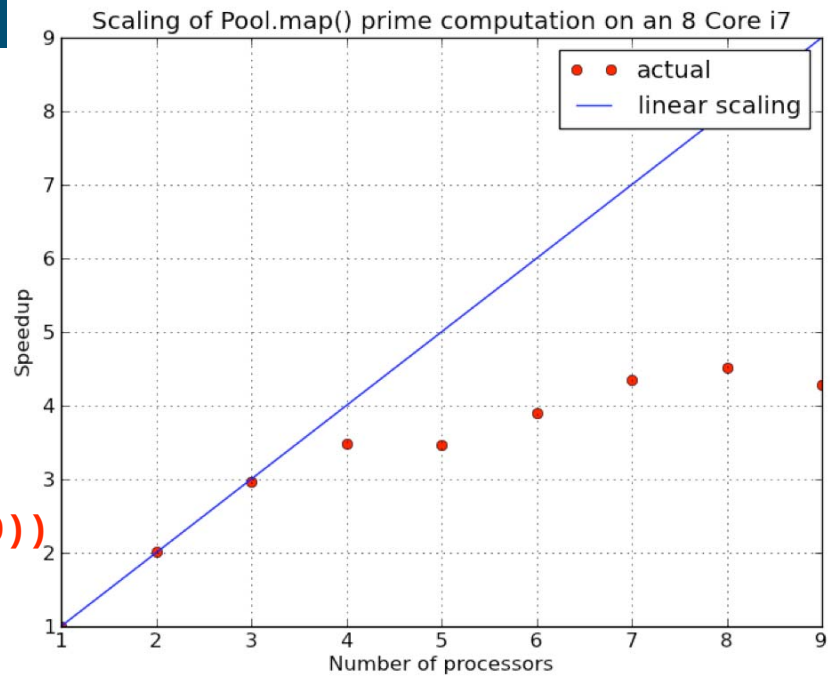
Task Pools

BASIC POOL USAGE

```
import multiprocessing as mp

def is_prime(num):
    # determine whether num is prime
    return (num, isp)

if __name__ == '__main__':
    # create a pool with cpu_count() procs.
    pl = mp.Pool()
    results = pl.map(is_prime, range(50, 100))
    for num, isp in results:
        if isp: print num, “is prime”
```

OUTPUT

```
$ python fermat_prime.py
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
...
```



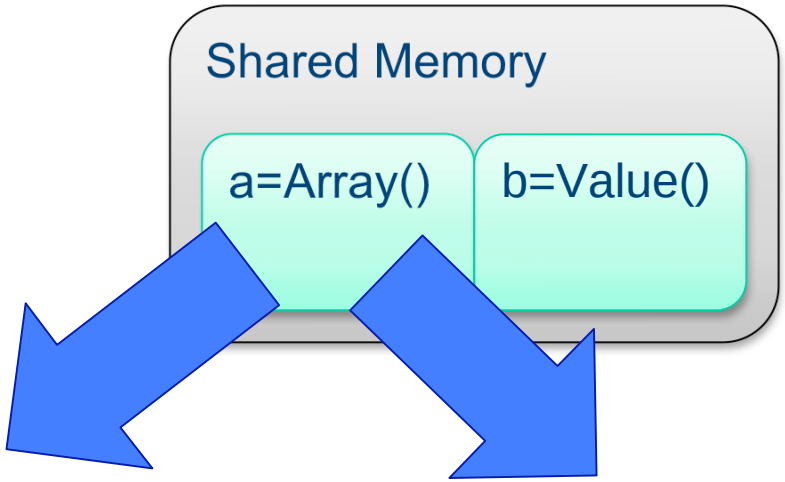
Scaling computed for 10^6 integers while varying pool size. Max speedup is ~4.5X on 8 cores. This problem has high communication overhead.

Shared Memory

Multiprocessing provides the **`Value`** and **`Array`** classes for shared memory programming; there is also the **`multiprocessing.sharedctypes`** module to create ctypes objects and arrays in shared memory.

Passing a shared memory object to the args of a **`Process()`** object will pass a *proxy* to the target; the actual memory location is in shared memory. Any modifications to the proxy object will be seen by all processes.

This allows parallel array computations without much communication overhead.




```
def worker1(a):
    # modify a
```

```
def worker2(a):
    # can see worker1's changes
```

Multiprocessing + NumPy

NumPy arrays can be made to view multiprocessing shared memory objects.

SHARED MEMORY VIEW

```
import multiprocessing as mp
from multiprocessing import sharedctypes
from numpy import ctypeslib

def fill_arr(arr_view, i):
    arr_view.fill(i)

if __name__ == '__main__':
    ra = sharedctypes.RawArray('i', 4)
    arr = ctypeslib.as_array(ra)
    arr.shape = (2, 2)
    p1 = mp.Process(target=fill_arr,
                    args=(arr[:1, :], 1))
    p2 = mp.Process(target=fill_arr,
                    args=(arr[1:, :], 2))
    p1.start(); p2.start()
    p1.join(); p2.join()
    print arr
```

OUTPUT

```
$ python shared_numpy.py
[[1 1]
 [2 2]]
```

Multiprocessing and others

The multiprocessing module is typically used for non-numerical calculations, when Python's role as a glue between several concurrent processes can be used.

For numerical calculations, numerically-optimized third-party libraries such as PyMPI, MPI4Py, PyOpenCL, CLyther, PyCuda, etc. are typically used.

Multiprocessing is useful when one needs to add concurrency to an application without requiring external dependencies.

Python, MPI, & MPI4Py

What is MPI4Py? Why use it?

MPI4Py wraps the the Message Passing Interface library in Python, allowing a user to develop, wrap, or interface with MPI programs from Python. It provides a clean and flexible interface, and has extra support to work nicely with NumPy arrays.

MPI4Py provides access to the entirety of MPI:

- Automatic initialization and finalization
- COMM_WORLD, custom communicators, groups
- send(), recv(), sendrecv() (blocking an non-blocking variants)
- Bcast(), scatter(), gather(), alltoall() (and variants)
- One-sided communication, window creation, put(), get()
- Custom datatype creation

`http://mpi4py.scipy.org/docs/usrman/index.html`

MPI4Py Basics

The **`mpi4py.MPI`** module contains everything for MPI interaction; importing it calls **`MPI_Init()`** under the hood.

THE MPI MODULE

```
# mpi_basic.py

# MPI_Init() called on MPI import.
from mpi4py import MPI
import os

print “Hello, World!”, os.getpid()
# MPI_Finalize() called at exit.
```

RUNNING MPI4PY PROGRAMS

```
$ mpiexec -n 4 python mpi_basic.py
Hello, World! 39011
Hello, World! 39014
Hello, World! 39012
Hello, World! 39013
```

COMM_WORLD

```
from mpi4py import MPI
From os import getpid

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

print “pid”, getpid(),
print “size”, size,
print “rank”, rank
```

OUTPUT

```
$ mpiexec -n 2 python comm_world.py
pid 39342 size 2 rank 1
pid 39341 size 2 rank 0
```

Collective Communication

Point-to-point communication is between one source and one destination.

Collective communication is one-to-many, many-to-one, or many-to-many. Object and buffer versions of each.

One-to-many:

`comm.Bcast(…)`

`comm.Scatter(…)`

many-to-one:

`comm.Gather(…)`

`comm.Reduce(…)`

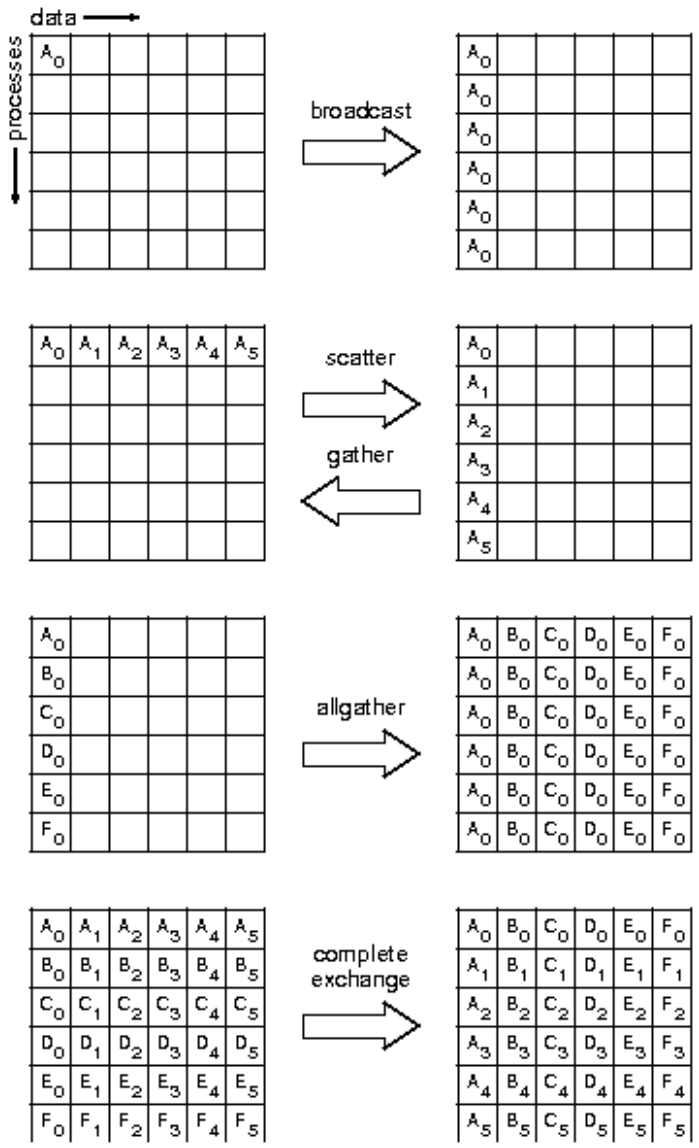
many-to-many:

`comm.Allgather(…)`

`comm.Alltoall(…)`

`comm.Allreduce(…)`

`comm.Reduce_scatter(…)`



Collective Communication

BARRIER

`comm.Barrier()` – blocks until all processes synchronize.

BCAST

`comm.Bcast(buf, root)` – sends object from root process to all processes.

SCATTER

`comm.Scatter(sendbuf, recvbuf, root)` – split up object or buffer in even pieces on root, send corresp. piece to recvbuf on all processes.

GATHER

`comm.Gather(sendbuf, recvbuf, root)` – inverse of comm.Scatter().

REDUCE

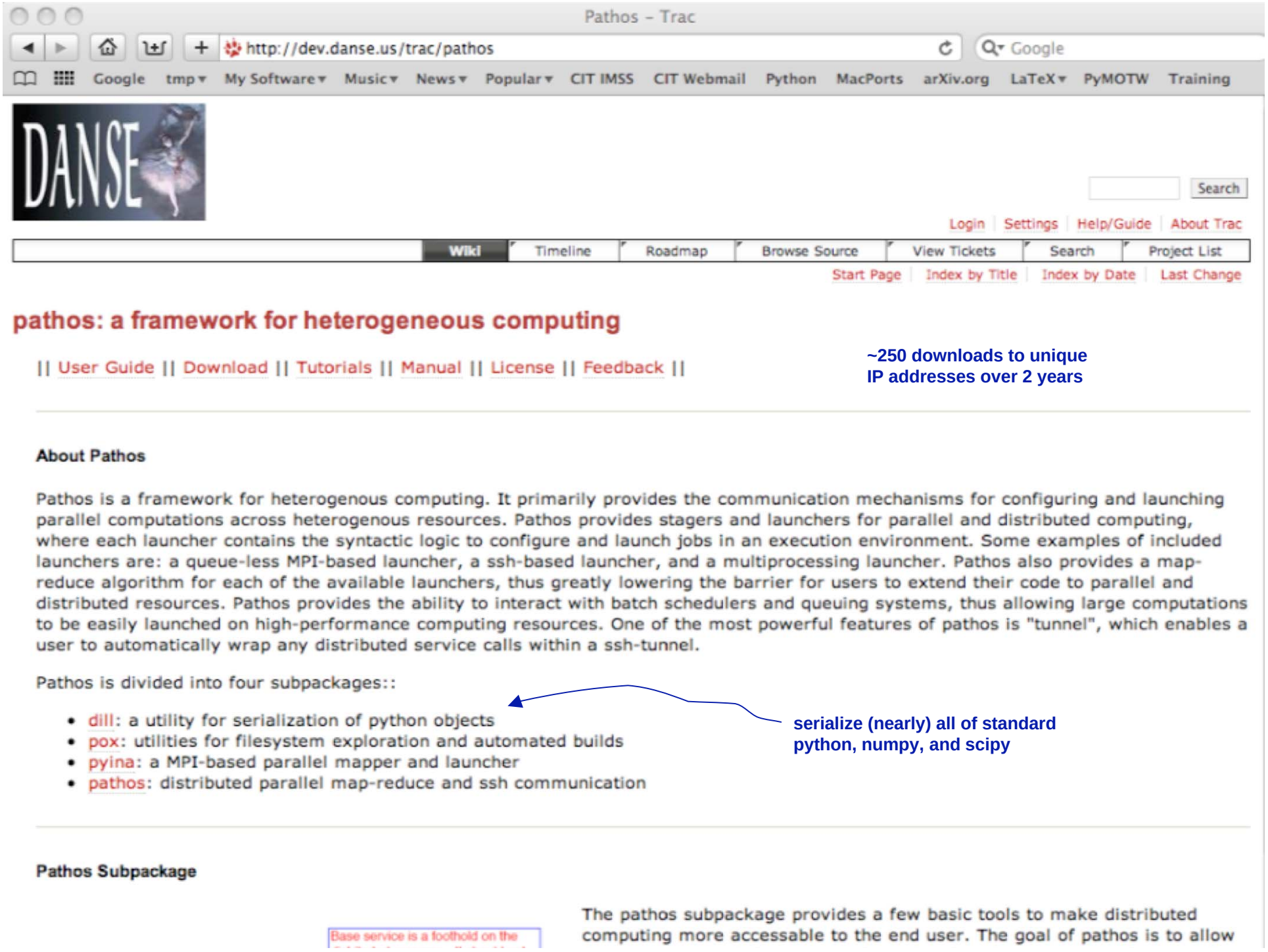
`comm.Reduce(send, recv, op, root)` – combine send buffer across processes using operation op, storing result in recv on root process.

`comm.Allreduce(send, recv, op)`

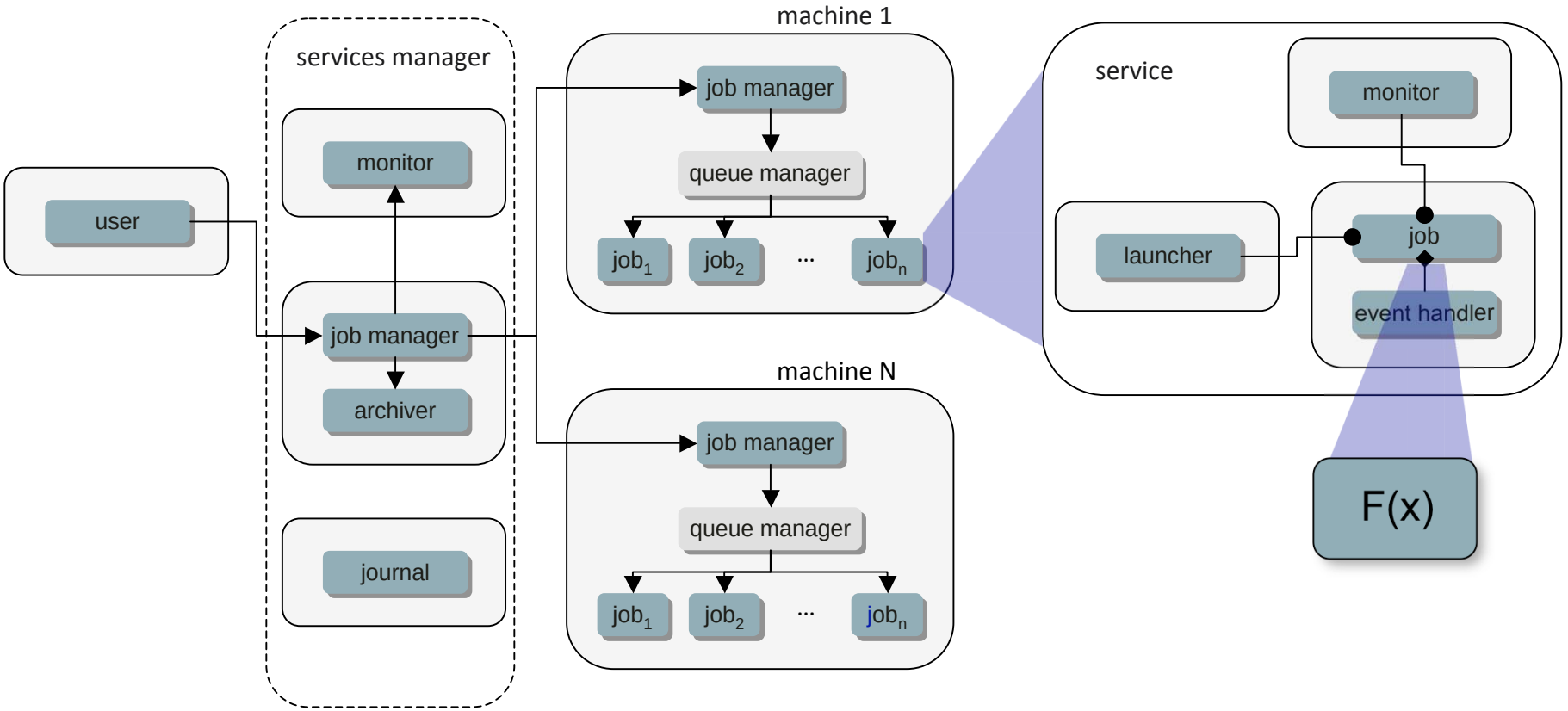
`comm.Reduce_scatter(send, recv, op)`– Useful variants of Reduce(…).

ALLTOALL

`comm.Alltoall(sendbuf, recvbuf)` – the jth block of sendbuf sent from process i is received by process j and placed in the ith block of recvbuf. (Expensive!)


Pathos – Trac
http://dev.danse.us/trac/pathos
Google
Google tmp My Software Music News Popular CIT IMSS CIT Webmail Python MacPorts arXiv.org LaTeX PyMOTW Training
DANSE
Search
Login Settings Help/Guide About Trac
Wiki Timeline Roadmap Browse Source View Tickets Search Project List
Start Page Index by Title Index by Date Last Change
pathos: a framework for heterogeneous computing
|| User Guide || Download || Tutorials || Manual || License || Feedback ||
~250 downloads to unique IP addresses over 2 years
About Pathos
Pathos is a framework for heterogenous computing. It primarily provides the communication mechanisms for configuring and launching parallel computations across heterogenous resources. Pathos provides stagers and launchers for parallel and distributed computing, where each launcher contains the syntactic logic to configure and launch jobs in an execution environment. Some examples of included launchers are: a queue-less MPI-based launcher, a ssh-based launcher, and a multiprocessing launcher. Pathos also provides a map-reduce algorithm for each of the available launchers, thus greatly lowering the barrier for users to extend their code to parallel and distributed resources. Pathos provides the ability to interact with batch schedulers and queuing systems, thus allowing large computations to be easily launched on high-performance computing resources. One of the most powerful features of pathos is "tunnel", which enables a user to automatically wrap any distributed service calls within a ssh-tunnel.
Pathos is divided into four subpackages::
dill: a utility for serialization of python objects
pox: utilities for filesystem exploration and automated builds
pyina: a MPI-based parallel mapper and launcher
pathos: distributed parallel map-reduce and ssh communication
serialize (nearly) all of standard python, numpy, and scipy
Pathos Subpackage
Base service is a foothold on the
The pathos subpackage provides a few basic tools to make distributed computing more accessable to the end user. The goal of pathos is to allow

lowering the barrier to parallel computing



goal: make the configuration, deployment, and management of F(x) on heterogeneous resources as easy as possible

pathos framework “subpackages”

pathos.pathos

- workflow API: launchers, services, maps, couplers, …
- map API: strategy, conditional, (a)synchronous, load balancing, …
- server plugins: sockets, rpc, ssh-tunneled
- distributed monitoring and lookup (e.g. logs and simple database)

pathos.hydra

- proxies and distributed graph management
- service, proxy, and server factories

pathos.hpbs

- explicit scheduler management (torque and pbs only)

pathos.pyina

- scheduler API: scheduler and batch queue management
- parallel processing plugins: threads, multiprocessing, MPI, …

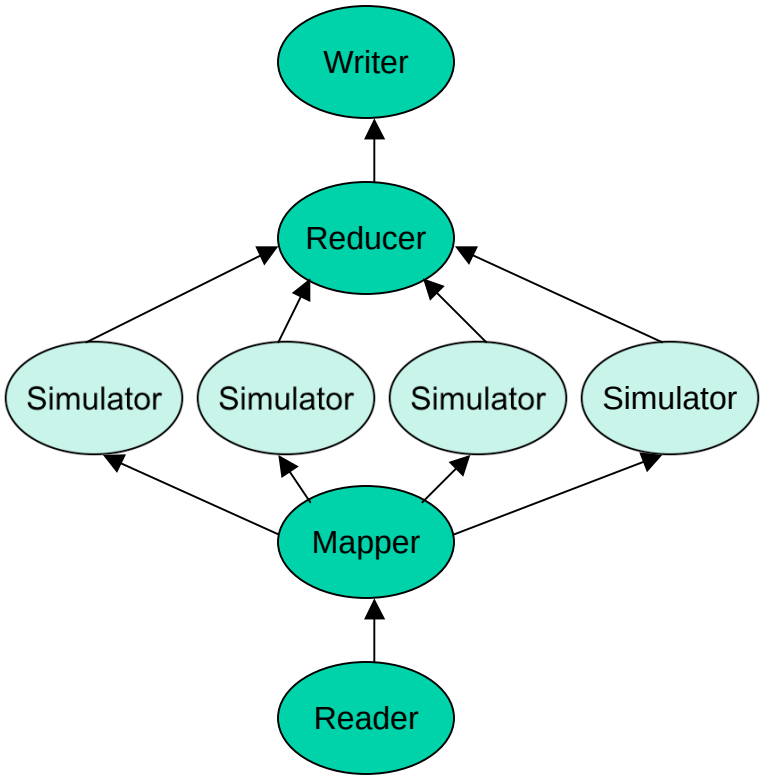
pathos.dill

- serialize all of standard python (i.e. pickle)

pathos.pox

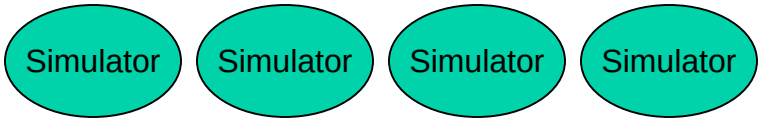
- “remote” file-system interaction and utilities (i.e. shutils)

graph management with proxies



resource 1

scheduler needs only manage nodes on it's own domain

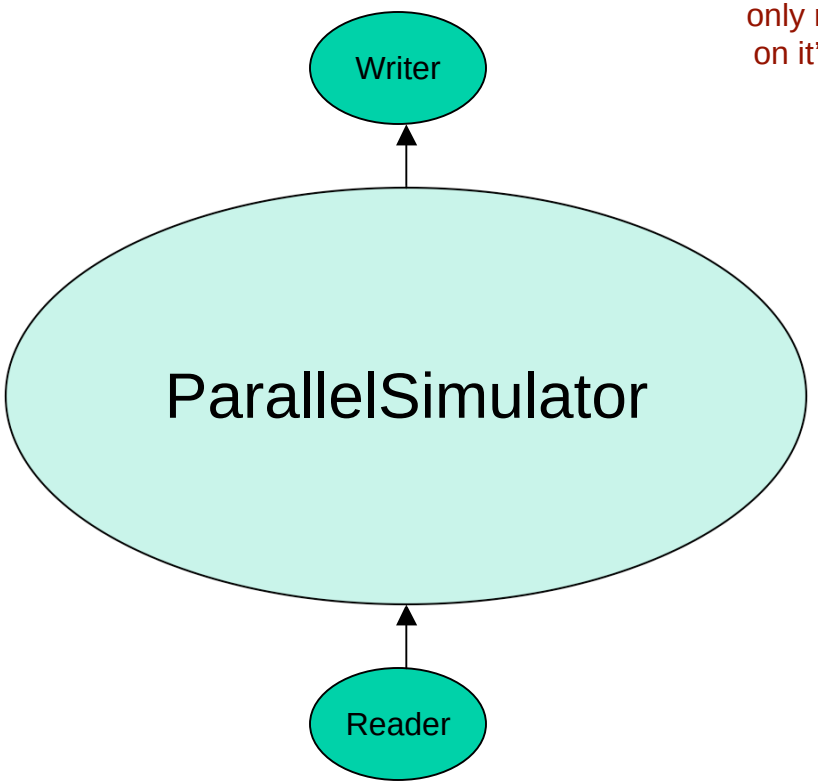


resource 2,3,…

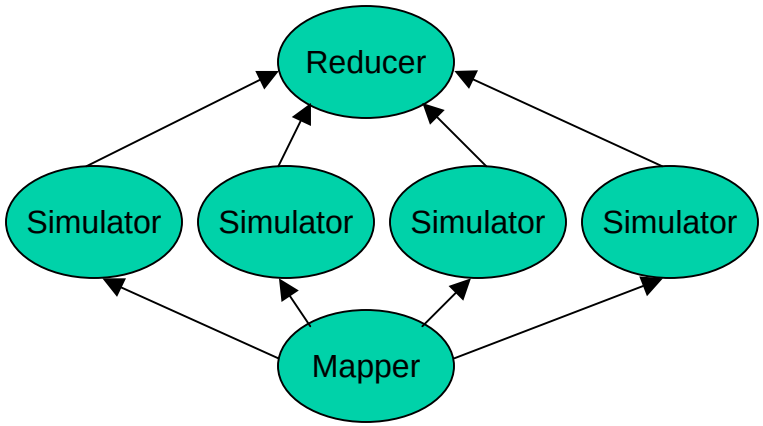
proxy calls to a service at a given URI

alternate use case with proxies

scheduler needs
only manage nodes
on it's own domain



resource 1



resource 2

untangles workflow logic into
asynchronously coupled services

abstract assembly of analysis circuits

```
# a user-provided model function
def identify(x)
  return x

# add pathos infrastructure (included in mystic)
from mystic.tools import modelFactory, Monitor
evalmon = Monitor()
my_model = modelFactory(identify, monitor=evalmon)

# evaluate the model
y = my_model(x)

# evaluate the model with a map function
from mystic.tools import PythonMap
my_map = PythonMap()
z = my_map(my_model, range(10))
```

map provides batch processing on an potentially distributed or parallel service

---

```
# select and configure a parallel map
from pathos.maps import ipcPool
my_map = ipcPool(2, servers=['foo.caltech.edu'])

# evaluate the model in parallel
z = my_map(identify, range(10))
```

map itself provides distributed or parallel infrastructure

- available launchers:
  - multiprocessing, threading
  - MPI-based (mpi4py)
  - IPC-based (pp)
  - SSH-based
  - GPU-based (pyCUDA)
  - cloud-based
- available schedulers:
  - torque, slurm, lsf
- compound services can be built manually or with a coupling strategy
  - sequential
  - nested
  - pool map
  - equalportion map
  - carddealer map

mystic: a simple model-independent inversion framework

|| User Guide || Download || Tutorials || Manual || License || Feedback ||

~900 downloads to unique IP addresses over 2 years

About Mystic

The mystic framework provides a collection of optimization algorithms and tools that allows the user to more robustly (and readily) solve optimization problems. All optimization algorithms included in mystic provide workflow at the fitting layer, not just access to the algorithms as function calls. Mystic gives the user fine-grained power to both monitor and steer optimizations as the fit processes are running.

Where possible, mystic optimizers share a common interface, and thus can be easily swapped without the user having to write any new code. Mystic solvers all conform to a solver API, thus also have common method calls to configure and launch an optimization job. For more details, see `mystic.abstract_solver`. The API also makes it easy to bind a favorite 3rd party solver into the mystic framework.

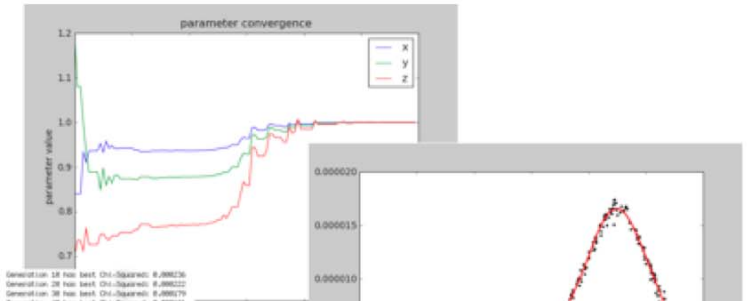
By providing a robust interface designed to allow the user to easily configure and control solvers, mystic reduces the barrier to implementing a target fitting problem as stable code. Thus the user can focus on building their physical models, and not spend time hacking together an interface to optimization code.

Mystic is in the early development stages, and any user feedback is highly appreciated. Contact Mike McKerns [mmckerns at caltech dot edu] with comments, suggestions, and any bugs you may find. A list of known issues is maintained at http://dev.danse.us/trac/mystic/query.

Major Features

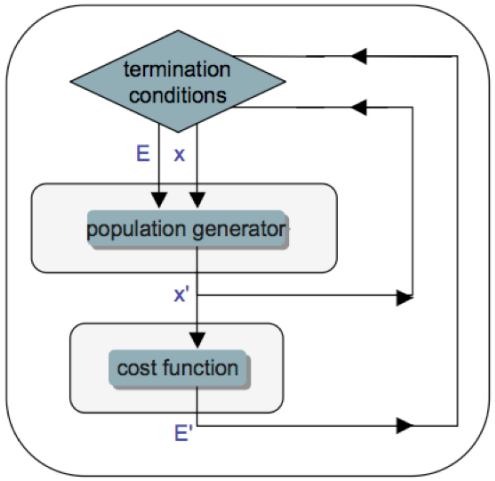
Mystic provides a stock set of configurable, controllable solvers with::

- a common interface
- the ability to impose solver-independent bounds constraints
- the ability to apply solver-independent monitors
- the ability to configure solver-independent termination conditions
- a control handler yielding: [pause, continue, exit, and user_callback]
- ease in selecting initial conditions: [initial_guess, random]



optimization with an expanded interface

```
# the function to be minimized and initial values
from mystic.models import rosen as my_model
x0 = [0.8, 1.2, 0.7]

# get monitor and termination condition objects
from mystic.monitors import Monitor, VerboseMonitor
stepmon = VerboseMonitor(5)
evalmon = Monitor()
from mystic.termination import ChangeOverGeneration
COG = ChangeOverGeneration()

# instantiate and configure the solver
from mystic.solvers import NelderMeadSimplexSolver
solver = NelderMeadSimplexSolver(len(x0))
solver.SetInitialPoints(x0)
solver.SetGenerationMonitor(stepmon)
solver.SetEvaluationMonitor(evalmon)
solver.Solve(my_model, COG)

# obtain the solution
solution = solver.bestSolution
```



an optimizer is composed of a population generator and termination conditions, acting on a cost function

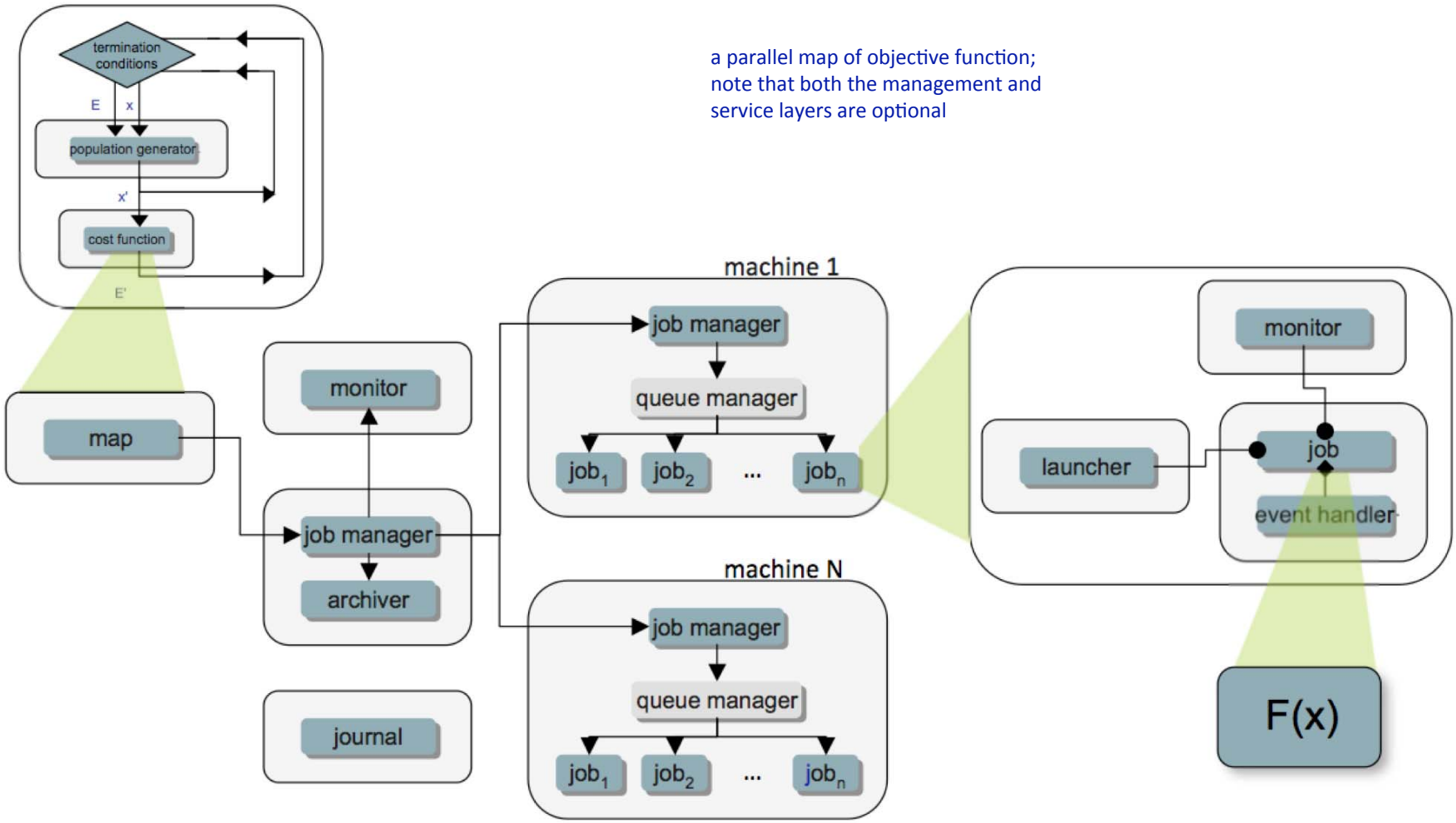
a cost function provides a difference metric
- $E = |F(x) - G|^2$

optimizer independent:

- monitoring & logging
- termination conditions
- penalties & constraints

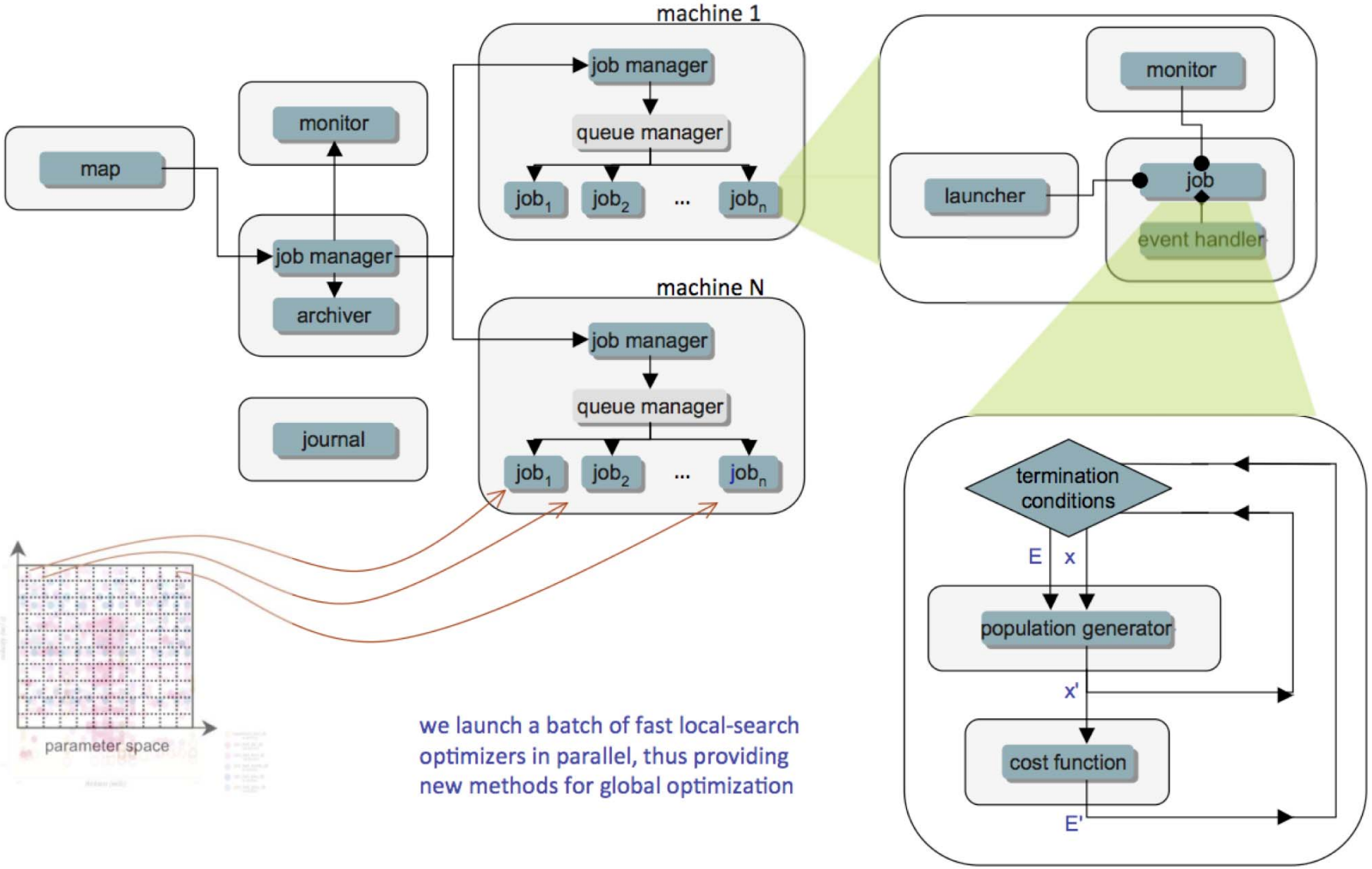
```
# obtain diagnostic information
function_evals = solver.evaluations
iterations = solver.generations
cost = solver.bestEnergy

# modify the solver configuration; continue
COG = ChangeOverGeneration(tolerance=1e-8)
solver.Solve(my_model, COG)

# obtain the new solution
solution = solver.bestSolution
```

a parallel mapped genetic optimizer



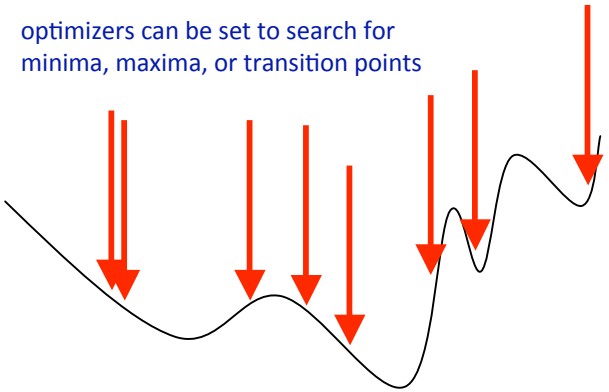
a parallel map of objective function;
note that both the management and
service layers are optional

the buckshot-Powell batch optimizer



new massively-parallel optimizers

```
# the function to be minimized and the bounds
from mystic.models import rosen as my_model
lb = [0.0, 0.0, 0.0]; ub = [2.0, 2.0, 2.0]

# get monitor and termination condition objects
from mystic.monitors import LoggingMonitor
stepmon = LoggingMonitor(1, 'log.txt')
from mystic.termination import ChangeOverGeneration
COG = ChangeOverGeneration()

# select the parallel launch configuration
from pyina.maps import TorqueMpirunCarddealer
my_map = TorqueMpirunCarddealer('5:ppn=4')

# instantiate and configure the nested solver
from mystic.solvers import PowellDirectionalSolver
my_solver = PowellDirectionalSolver(len(lb))
my_solver.SetStrictRanges(lb, ub)
my_solver.SetEvaluationLimits(50)

# instantiate and configure the outer solver
from mystic.solvers import BuckshotSolver
solver = BuckshotSolver(len(lb), 20)
solver.SetRandomInitialPoints(lb, ub)
solver.SetGenerationMonitor(stepmon)
solver.SetNestedSolver(my_solver)
solver.SetSolverMap(my_map)
solver.Solve(my_model, COG)
# obtain the solution
solution = solver.bestSolution
```



- New optimizers can also be created by coupling optimizers or applying constraints between parallel optimizers
  - Powell-*nested*-DE
  - *penalty*-ParticleSwarm

End Presentation