Python Programming for High Performance Environments

Cloud, Robert L. *   Florisson, Mark   Daily, Jeff
rcloud@uab.edu

1   Abstract

Increasingly, the Python programming language is being used for scientific and high-performance applications. Python is attractive because of its ease of use, readability, and brevity. Prominent and large-scale organizations are adopting Python to their uses, and for exploratory programming, it is a compelling alternative to engineering suites such as MATLAB. Python is currently being used in large-scale high-performance applications, but as a glue language, controlling and wrapping modules written in compiled languages. Very recent developments in the Python community, however, now make Python a viable language to write even the computational subroutines of a large-scale scientific application. This paper seeks to highlight two recent developments in the Python ecosystem, the new parallel features of Cython, a popular interface between Python and C, and the new Python bindings for the Global Arrays Toolkit, a venerable library which allows the abstraction of communication in a manner similar to Fortran Co-Arrays. We also give a brief survey of Python's available mechanisms and libraries that one can use in scientific and high performance computing and provide references for further reading on the subject.

2   Introduction

2.1   Contemporary Uses of Python in Computation

The use of Python in the sciences dates back to the 1990s. With the development of libraries supporting efficient arrays and matrices, Python became an attractive language for numerical algorithmic exploration. In the past few years, the community using Python for the sciences has grown worldwide, evidenced by the popular SciPy conference, and companies

---

*The University of Alabama, School of Engineering

such as Enthought have taken advantage of this trend, offering ready built installations of Python complete with tools needed for scientific programming and visualization[1].

The attraction to Python comes from its readability, interactivity, extensibility, and openness. One with little experience in programming can easily begin writing and interacting with Python modules, and existing software libraries, written in low-level languages, can be efficiently wrapped, allowing one to use them in an interactive way.

While Python is well known as an effective language to do exploratory programming, this paper addresses the use of Python in the High Performance Computing ecosystem. The contemporary use of Python in large-scale, computationally expensive scientific applications has been largely limited to the role of command and control,making I/O less burdensome, and interfacing with modules written in lower level languages. Towards this end, Python has successfully been used for large-scale applications by disparate groups such as the Department of Energy, the Department of Defense, and the German Aerospace Centre[13], [15] .

The use of Python in computational settings has become popularized, especially through the work of Perez et al. in their high profile publications. They offer advice and advocate the use of Python in every stage of application development, from algorithmic exploration, to prototyping, to use in production code [11], [10]. Other work has been done in incorporating Python into high performance installations by Unpingco [16].

2.2   Challenges to Using Python at Scale

The impediments to using Python in large-scale scientific applications are twofold. First, even though there are efficient Python bindings to MPI, it has been demonstrated that CPython has problems when running on many thousands of cores. In large-scale distributed machines with a shared file system, the simple importation of Python modules can take a prohibitively long time. This is because many thousands of processes are concurrently requesting access to small files, resulting in thrashing. There is ongoing work to remedy this. Several institutions, most prominently the DoE Labs, have developed tailored solutions for their particular problem. A more general solution being worked on at Argonne National Lab, which is developing a method to use an MPI broadcast for the distribution of Python modules.

The second impediment to using Python in HPC is its limited support for shared memory parallel programming. The problems associated with this are discussed in Sverre[14], but is most prominently the limitations of the Global Interpreter Lock(GIL), which prevents multiple Python threads from being executed on the CPU concurrently. While there have been ways to circumvent this for some time, noting specifically the multiprocessing module in the Python library(though this is a solution where processes don't share the same mem-

ory space), shared memory parallel programming has more recently been made available to the Python community in the form of a Cython compiler with an OpenMP backend. This recent development is key to our argument that Python can now, more than ever, be effectively used as a primary language for large-scale parallel computational work.

2.3   Parallel Programming Facilities in the Python Ecosystem

While Python has been successfully used in large-scale applications, its role in these has been limited to the non-computational modules. Python was built with extensibility in mind, and one can easily interface between Python and C or Fortran. Thus large-scale applications perform input and output operations using Python but call numerical routines, wrapped by Python, but written a lower-level language. We argue however, that one can now make a compelling case for the direct use of Python in the computational routines, specifically by using Cython and OpenMP to create modules of comparable speed to traditional compiled languages.

The most common way to program contemporary supercomputers, i.e. distributed memory based clusters, is to use the Message Passing Interface(MPI). There are efficient bindings to MPI, mpi4py [5], developed by Lisandro Dalcin. Large-scale distributed applications have been successfully implemented using these bindings, but often, the computational routines are being run through wrappers to a low level language.

For shared memory parallel programming, the choices have been rather limited. One could use the multiprocessing module in the Python library or call modules written in compiled code that make use of threading, but both methods are rather restrictive, the first being that processes dont share a single memory space, and the second having the associated inconvenience of wrapping modules written in other languages. While it is certainly true that many MPI implementations can make use of shared memory machines, programming MPI, even in Python, can be cumbersome, difficult, and hard to debug. OpenMP[7] is the de facto standard for shared memory parallel programming for multi-core machines. Cython 0.15, released in mid 2011, supports shared memory parallel programming using OpenMP as a backend, enabling one to easily utilize the resources of contemporary multicore machines.

There has been much discussion [12] of hybrid MPI and OpenMP programming and the challenges that follow, but it will become more important as the number of cores in machines continue to grow. While hybrid programming is outside of the scope of this paper, we will demonstrate that one can get excellent performance by making use of the Cython's new OpenMP facilities while retaining many of the conveniences associated with Python, and this is key to our argument for the use of Python in even the computationally expensive modules of a scientific application.

2.4   A Scientific Application

Laplaces Equation is a well-known partial differential equation and is useful as a pedagogic tool because it is easily solved with computational methods, easily understood, and shares common characteristics with many other scientific problems. The equation models the steady-state distribution of independent variables in space.

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0$$

Taylors Theorem and the derivation of finite difference schemes lead directly to simple iterative methods such as Jacobi Iteration which can be used to find a solution of the equation when provided with boundary conditions. Though there are other alternatives which converge faster, e.g., Gauss Seidel Iteration with successive overrelaxation, The Jacobi iterative method is extremely amenable to parallelization and we will use it for this reason.

$$U_{i,j} = \frac{U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1}}{4}$$

2.5   Outline

The rest of this paper is as follows: we will first implement Laplace's Equation in pure Python. Then we will take a look at a Cython implementation followed by a parallel Cython implementation using OpenMP. We will then demonstrate an implementation with the Global Arrays toolkit. We will conclude our paper by surveying other contemporary parallel programming facilities in Python.

3   Implementations

3.1   A Straight Python Implementation

We first implement our Laplace solver using pure Python with the extension of multi-dimensional NumPy arrays. We perform basic Python loops using xrange(). It is well known that Python is very inefficient when operating in this manner, and this is demonstrated in the figure. Our solution module is derived directly from the algebraic Jacobi formulation.

```
def solve(u, u_new):
    dim_y = u.shape[0]
    dim_x = u.shape[1]
    for y in xrange(1, dim_y - 1):
        for x in xrange(1, dim_x - 1):
            u_new[y,x] = (u[y + 1,x] +
                                       u[y - 1,x] +
                                       u[y,x + 1] +
                                       u[y,x - 1]) / 4
```

3.2 Vectorizing with NumPy for Higher Performance

As it is well known that iteration in Python over NumPy arrays is inefficient, NumPy offers slicing semantics which correspond to efficient C loops on the backend. Performance of our solve routine is vastly improved, and at this point, one may stop and consider whether further performance gains are necessary.

```
def solve(u, u_new):
    u_new[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1]) +
                                     (u[1:-1,0:-2] + u[1:-1, 2:])) / 4
```

3.3 Cython Implementation

Cython was developed by a team including Behnel, Bradshaw, Citro, Dalcin, Seljebotn, and Smith as well as others[2]. It is often difficult or impossible to formulate algorithms in terms of NumPy vectorization. In these cases, if one wishes to have high performance, one must resort to interfacing with compiled languages. There are a great magnitude of ways to do this, and Wilbers and Langtangen discuss many of them[17]. One can interface with Fortran routines or C modules using f2py or Python's C extensions. If one doesn't wish to go this route, they establish that the most compelling option is to use Cython.

Cython is both a library and a program, capable of compiling Python code into a C file. Even when just run on a straight Python file, we can see moderate improvements in performance as is discussed in Seljebotn[14].

The real gains in performance are made when we add C types to a module. One of the great benefits of Cython is that one can gradually optimize until one achieves the performance level that is desired. This is done by adding flags, typing variables with C types, and manipulating(releasing) the GIL.

5

Wilbers and Langtangen establish Cython as a very compelling option to accelerate Python code. Seljebotn gives an overview of how one can achieve high performance using Cython for a matrix multiplication application as well as discussing the future potential of parallel computation with Cython.

3.4 Parallel Cython Implementation

Recently, with the release of Cython version 0.15, an OpenMP backend was added to Cython enabling the use of shared memory parallel processing. This is a very positive development for the Python community as it crosses a great preexisting impasse for shared memory parallel computing in Python. Because the GIL disallows Python threads to run in parallel, threads were only useful for concurrency purposes. Cython gives the ability to release the GIL for sections of code, and by doing this multiple threads can run in parallel on a multicore computer but we lose access to native Python functionality. Nevertheless, even when releasing the GIL, Cython can convert Python statements to C, allowing one to continue using Python's extraordinary brevity in the source (before being converted to C) while at the same time realizing high performance.

The functionality is a subset of OpenMP exposed through a Pythonic interface. The functionality is especially useful to execute computational loops in parallel, through the `prange` function. The compiler automatically figures out which variables should be private or shared, and which ones should be reduced at the end. The OpenMP runtime will typically figure out how many CPU cores a machine has and will distribute the work over a team of threads. The number of threads may also be specified at runtime, either through keyword arguments, the use of OpenMP functions, or through environment variables.

Depending on how the work is distributed over the iterations a sensible scheduling scheme may be chosen. For instance, one may want to distribute all the work evenly before starting the loop, by assigning a portion of work to each thread, or one may want to have a set of tasks that get executed by the threads when the amount of work per iteration varies enough. Besides the parallel loop, a general parallel `with` block may be used, which can be useful to setup or post-process thread-local data that is operated on by a parallel loop, or to allow multiple worksharing loops without the overhead of starting parallel regions, or to allow efficient parallalization of an inner loop. In the future these regions may be useful for task generation as well.

The user must be wary of false sharing and utilize private data as much as possible [7]. For instance, declaring an array of the size of the thread team and assigning to individual elements depending on the thread ID will lead to bad performance, as the array elements are contiguous in memory and end up in the same cache line. Whenever one thread assigns to their element in the array, they invalidate the cached copy of the other threads. In the

future non-scalar reductions may be implemented to avoid false sharing or the need for atomic operations or locking for certain kind of operations.

The following code may perform bad depending on the scheduling scheme and chunk sizes involved:

```
cdef int i
cdef double array[N]
for i in prange(N):
    array[i] = compute(i)
```

For this algorithm a cyclic data distribution scheme would perform bad as it maximizes false sharing. It would be better to use a block-cyclic or block distribution scheme. For instance, if `compute` takes approximately the same time for each `i`, then `static` would be a good choice. If the time to do each computation varies too much, a tradeoff has to be made between optimal load distribution and false sharing. One may for instance choose `dynamic` scheduling with a chunksize of 16 (so that 16 * sizeof(double) is the size of your cache lines), if that provides good load balancing as well. If these arrays were reduction variables it would not matter at all which scheduling scheme were chosen, although we would then need to use `+=` for the assignment, and we would need to initialize our array to zero.

In the future barriers, tasks (for producer/consumer-like functionality) and some form of mutual exclusion (akin to critical sections) will be available, and possibly atomic constructs.

3.4.1   Cython 0.16

In Cython 0.16 there will be more functionality available to deal with arrays (PEP 3118 [9]), and it will allow the user to do more without the GIL and with a clean syntax. Aditionally, the new-gained fused types functionality will allow the user to operate on arrays with different base types in the same code, as well as allowing template-like funcionality in general.

3.5   Parallel Features of IPython

IPython[10] is a very popular Python shell developed by Fernando Perez and Brian Granger et. al. Recently, with version 0.11, the parallel capabilities of IPython were rewritten by Min Ragan-Kelley making use of ZeroMQ for networking. With IPython, one can interactively manipulate and run modules across a distributed environment. This feature brings Python's exploratory nature to parallel environments and can be very useful for

developing and debugging parallel applications. We refer the interested reader to the
IPython documentation.

3.6   MPI Implementation with mpi4py

Python bindings for MPI have been available since the early 2000's with the PyMPI wrappers from Lawrence Livermore National Labs[8]. A more contemporary and more fully featured library is available through mpi4py developed by Lisandro Dalcin[5]. mpi4py provides a complete set of bindings to the MPI 2 specification and has been used successfully in very large scale applications. mpi4py enables one to both pass messages composed of regular Python objects through pickling and the efficient messaging of NumPy buffers. In order to achieve both high performance and convenience, one should make use of both features.

```
iteration = 0
    while cont_vals[-1] == True: # We sort this list, True last element unless all False
        if iteration % 2 == 0: #even
            if rank != 0:
                comm.Send(u[1], dest=(rank - 1)) # send top row to proc above
            if rank != size - 1:
                comm.Recv(u[-1], source=(rank + 1)) # recv boundary from proc below
            if rank != size - 1:
                comm.Send(u[-2], dest=(rank + 1)) # send bottom row to proc below
            if rank != 0:
                comm.Recv(u[0], source=(rank - 1)) # recv top row from proc above
            solve(u, u_new, u.shape[0], u.shape[1])


        else:
            if rank != 0:
                comm.Send(u_new[1], dest=(rank - 1)) # send top row to proc above
            if rank != size - 1:
                comm.Recv(u_new[-1], source=(rank + 1)) # recv boundary from proc below
            if rank != size - 1:
                comm.Send(u_new[-2], dest=(rank + 1)) # send bottom row to proc below
            if rank != 0:
                comm.Recv(u_new[0], source=(rank - 1)) # recv top row from proc above
            solve(u_new, u, u.shape[0], u.shape[1])

        #test for convergence
        if iteration % 1000 == 0:
            cont = False
```

```
            if(np.max(u_new - u) > eps):
                cont = True
            cont_vals = comm.allgather(cont)
            cont_vals.sort()
        iteration = iteration + 1
```

Here, the communicational part is the necessary addition to our solver. We must communicate the boundaries between the processes using the Send and Recv methods of the communicator objects. We continue the iteration until the difference between the two solution matrices fall below a specified threshold epsilon. This is managed by the communication of a boolean Python list object contvals which is gathered from each process to rank 0. We send the rows as NumPy buffers using the optimized methods which begin with a capital letter. We test for convergence every 1000 iterations, sort contvals and if any of the items in the list evaluate to True, continue the iteration.

We have abstracted away our solve routine. By doing this, we can implement it in a number of ways. We could, for instance, make use of our straight Python solve routine, our vectorized NumPy solve routine, or a solve routine written in Cython.

Furthermore, we never have to store the complete matrix in the memory of a single process. Rather, initialization data is broadcasted to each process which initializes its own sub matrix using the following routine:

```
def initialize(dim, comm, rank, size):
    init_vals = None
    if rank == 0:
        top = random.random() * 100
        left = random.random() * 75
        right = random.random() * 50
        bottom = random.random() * 25
        init_vals = {'top':top, 'left':left, 'right':right, 'bottom':bottom}
    init_vals = comm.bcast(obj=init_vals)
    u = np.zeros((dim / size, dim))
    if rank == 0:
        u = np.vstack((u, np.zeros(dim)))
        u[0, :] = init_vals['top']
        u[:, 0] = init_vals['left']
        u[:, -1] = init_vals['right']

    elif rank == size - 1:
        u = np.vstack((np.zeros(dim), u))
        u[-1, :] = init_vals['bottom']
        u[:, 0] = init_vals['left']
```

```
        u[:, -1] = init_vals['right']
    else:
        u = np.vstack((np.zeros(dim), u, np.zeros(dim)))
        u[:, 0] = init_vals['left']
        u[:, -1] = init_vals['right']

    u_new = np.copy(u)
    return u, u_new
```

Again, taking advantage of mpi4py's ability to send native Python objects, we utilize a dictionary to store the semi random initialization values. Beginning with equally sized NumPy arrays(total m=n dimensions divided by number of processes that we specify when submitting to the cluster's queue), we stack them with the boundary rows, on the bottom for the top process, the top row for the bottom process and both the top and bottom for intermediate processes. By using semi random boundary conditions we can make our problem more similar to a real world unsteady system, where we would solve multiple times, each iterating until convergence, to produce data for the changing conditions.

3.7 Hybrid Parallel Programming with OpenMP and MPI

With the combination of mpi4py and the new OpenMP backend of Cython, it becomes possible to develop hybrid OpenMP/MPI applications which take full advantage of shared memory architectures in a distributed environment. This will become ever more important as multicore computers reach the level of many-core(see e.g. AMD's Bulldozer with 16 cores or Intel's Xeon with 10 cores and 20 threads).

The advantages and pitfalls of hybrid programming are discussed in some detail in Rabenseifner et. al. [12]

3.8 Adding Brevity to Distributed Computing with Global Arrays

The Global Arrays toolkit was developed at Pacific Northwest National Labs in the 1990s as an abstraction for distributed memory clusters enabling shared memory like accesses. Its primary development purpose was to support high performance chemistry applications, but it has since been ported to a number of architectures and interconnects. Recently, Python bindings have been developed for the language, and a tutorial was done on these by Jeff Daily at the SciPy 2011 conference[4].

With Global Arrays, communication is implicit, removing the need to perform communications like that shown in the MPI implementation. This resolves in a program with much more brevity than an program requiring explicit MPI communication.

10

```python
from mpi4py import MPI
from ga4py import ga
import numpy as np
import sys

EPSILON = .0001
HOW_MANY_STEPS_BEFORE_CONVERGENCE_TEST = 2
DEBUG = False

if len(sys.argv) != 2:
    print "supply dimension"
    sys.exit()

dim = int(sys.argv[1])
rank = ga.nodeid()
size = ga.nnodes()

def print_sync(obj):
    for proc in range(size):
        if rank == proc:
            print "%d: %s" % (proc, obj)
        ga.sync()

def convergence_test(g_a, g_b):
    ### PROBLEM: g_b might contain zeros which causes GA to terminate
    # subtract g_b from g_a, results stored in g_b
    ga.add(g_a, g_b, g_b, beta=-1)
    # divide g_b by g_a, results stored in g_b
    ga.elem_divide(g_b, g_a, g_b)
    # find the largets element and compare to epsilon
    value,index = ga.select_elem_max(g_b)
    if DEBUG:
        print_sync(value)
    return value < EPSILON

def convergence_test_L2(g_a, g_b):
    # compute L2 norm of change
    # subtract g_b from g_a, results stored in g_b
    ga.add(g_a, g_b, g_b, beta=-1)
    # compute elementwise dot product (i.e. treats N-d arrays as vectors)
    value = ga.dot(g_b, g_b)
```

11

```python
    if DEBUG:
        print_sync(value)
    return value < EPSILON


# create GA, distribute entire rows
g_a = ga.create(ga.C_FLOAT, (dim,dim), chunk=(0,dim))
# create a duplicate GA for the convergence test
g_b = ga.duplicate(g_a)


# process 0 initializes global array
# Note: alternatively, each process could initialize its local data using
# ga.access() and ga.distribution()
a = np.zeros((dim,dim), dtype=np.float32)
if rank == 0:
    a[0,:] = 100 #top row
    a[:,0] = 75 #left column
    a[:,a.shape[0] - 1] = 50 #right column
    ga.put(g_a, a)
ga.sync()


# which piece of array do I own?
# note that rhi and chi follow python range conventions i.e. [lo,hi)
(rlo,clo),(rhi,chi) = ga.distribution(g_a)

iteration = 0
start = ga.wtime()
while True:
    ga.sync()
    iteration += 1
    if iteration % 200 == 0:
        # check for convergence will occur, so make a copy of the GA
        ga.copy(g_a, g_b)
    # the iteration
    if rlo == 0:
        # I own the top rows, so get top row of next domain
        next_domain_row = ga.get(g_a, (rhi,0), (rhi+1,dim))
        ga.sync()
        my_array = ga.access(g_a)
        combined = np.vstack((my_array,next_domain_row))
        my_array[1:,1:-1] = (
                combined[0:-2, 1:-1] +
```

12

```
                combined[2:, 1:-1] +
                combined[1:-1,0:-2] +
                combined[1:-1, 2:]) / 4
        ga.release(g_a)
    elif rank == size-1:
        # I own the bottom rows, so get bottom row of previous domain
        prev_domain_row = ga.get(g_a, (rlo-1,0), (rlo,dim))
        ga.sync()
        my_array = ga.access(g_a)
        combined = np.vstack((prev_domain_row,my_array))
        my_array[0:-1,1:-1] = (
                combined[0:-2, 1:-1] +
                combined[2:, 1:-1] +
                combined[1:-1,0:-2] +
                combined[1:-1, 2:]) / 4
        ga.release(g_a)
    else:
        # I own the middle rows, so get top and bottom row of neighboring domain
        next_domain_row = ga.get(g_a, (rhi,0), (rhi+1,dim))
        prev_domain_row = ga.get(g_a, (rlo-1,0), (rlo,dim))
        ga.sync()
        my_array = ga.access(g_a)
        combined = np.vstack((prev_domain_row,my_array,next_domain_row))
        my_array[0:,1:-1] = (
                combined[0:-2, 1:-1] +
                combined[2:, 1:-1] +
                combined[1:-1,0:-2] +
                combined[1:-1, 2:]) / 4
        ga.release(g_a)
    if iteration % HOW_MANY_STEPS_BEFORE_CONVERGENCE_TEST == 0:
        if convergence_test_L2(g_a, g_b):
            break

if DEBUG and rank == 0:
    print ga.get(g_a)

if rank == 0:
    print iteration
    print ga.wtime() - start, "seconds"
```

3.9 Heterogeneous Programming with PyCUDA and PyOpenCL

Heterogeneous computing is an extremely hot topic in the high performance computing world and very good Python bindings have been developed for the main languages being used to program novel architectures. PyCUDA, developed by Andreas Kloeckner, fully supports the latest implementation of NVIDIA CUDA, and PyOpenCL, also developed by Kloeckner et. al., supports OpenCL 1.1(the most recent revision of the specification) as well as adding additional functionality such as the ability to manipulate device arrays from the host [6]. Both bindings are elegantly designed, are very Pythonic, and make what would otherwise be the cumbersome task easy. True, the challenge of heterogeneous programming is writing and optimizing the kernels for an architecture, but PyCUDA and PyOpenCL enable one to do so by manipulating Python strings as well as diminishing the burden of the necessary host side boilerplate code.

One great feature about OpenCL is that if an implementation meets the specification, then an OpenCL kernel will run on it. This allows for the development of cross-architecture code. However, one must be aware that performance is not necessarily portable. To achieve optimal performance, one must develop to the architecture under consideration. One may find, however, that performance can be "good enough." We have successfully run our OpenCL kernel on the AMD, Intel, and NVIDIA implementations, and while the results on each are correct, performance varies widely. Out of all of our implementations, we see that our NVIDIA Tesla C1060 GPU(240 Processing Elements @ 1.3 GHz) gives the best performance. The performance of the AMD and Intel implementation are both below our Cython OpenMP application, though the Intel implementation gives the better performance over AMD for large domains.

Our OpenCL implementation is given in full:

```
import pyopencl as cl
import numpy as np

def get_kernel(size):
    width_macro = "#define WIDTH %s\n" %str(size)
    index_macro = "#define TD(u, y, x) (u[(y) * WIDTH + (x)])\n"
    kernel_source = width_macro + index_macro + """
    __kernel void solve(__global float *u,
                    __global float *u_new)
                    {
```

14

```
                    int id = get_global_id(0);

                    int y = id / WIDTH;
                    int x = id % WIDTH;

                    if (y != 0 && y != WIDTH - 1 && x != 0 && x != WIDTH - 1)
                    {
                        TD(u_new, y, x)   = (TD(u, y + 1, x) +
                                              TD(u, y - 1, x) +
                                              TD(u, y, x + 1) +
                                              TD(u, y, x - 1)) / 4;
                    }
                    }
    """

    return kernel_source

def initialize(a):
    #initialized using Chapra's values
    a[0,:] = 100 #top row
    a[:,0] = 75 #left column
    a[:,a.shape[0] - 1] = 50 #right column

def convergence_test(u_new, u):
    eps = .0001
    cont = False
    if(np.max(u_new - u) > eps):
        cont = True
    return cont

def solve(size):
    """
    size is the m=n domain to be solved
    """
    ctx = cl.create_some_context()
    """
    for Jacobian iteration we need two arrays, one to store the
    values from timestep i and one for timestep values i+1
    """
    u = np.zeros((size,size), dtype=np.float32)
    initialize(u)
    u_new=np.copy(u)
```

```
program = cl.Program(ctx, get_kernel(size)).build()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
#create the memory objects on the device
u_dev = cl.Buffer(ctx, mf.READ_WRITE | mf.COPY_HOST_PTR, hostbuf=u)
u_new_dev = cl.Buffer(ctx, mf.READ_WRITE | mf.COPY_HOST_PTR, hostbuf=u_new)
cont = True
iteration = 0
import time
start = time.time()
while cont == True:
    if iteration % 2 == 0:
        program.solve(queue, (size * size,), None, u_dev, u_new_dev)
    else:
        program.solve(queue, (size * size,), None, u_new_dev, u_dev)
    if iteration % 1000 == 0:
        cl.enqueue_copy(queue, u_new, u_new_dev)
        cl.enqueue_copy(queue, u, u_dev)
        cont = convergence_test(u_new, u)
    iteration = iteration + 1
cl.enqueue_copy(queue, u_new, u_new_dev)
end = time.time()
print end - start
return u_new
```

4   Conclusion

We have implemented our Laplace solver in Python, NumPy, Cython, both parallel and
sequential, MPI, OpenCL and with Global Arrays. We have evaluated their relative per-
formance to a baseline Fortran implementation. We have argued that Python has now
matured into a language capable of writing high performance code with a syntax primarily
characterized by its brevity. While it must be emphasized that we must retain and sup-
port legacy scientific codes, we believe that developers of new applications should seriously
consider writing computational modules in Python and especially Cython. Furthermore,
the difficulties associated with programming heterogeneous architectures can be to an ex-
tent overcome by using Python wrappers. Parallel programming is difficult and Python,
especially IPython provides a convenient mechanism to interactively test and debug par-
allel modules. Python's bindings to Global Arrays enable one to use the resources of a

distributed cluster while abstracting away the challenges of passing messages. Hybrid programming will become more important in the future, and current Python tools, such as mpi4py and parallel Cython, enable one to exploit this programming model.

References

[1] A. Alghamdi, A. Ahmadia, D. Ketcheson, M. Knepley, K. Mandli, and L. Dalcin. Petclaw: A scalable parallel nonlinear wave propagation solver for python, acm, 2011. In *proceedings of HPC*, 2011.

[2] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, (99):1–1, 2011.

[3] X. Cai, H. P. Langtangen, and H. Moe. On the performance of the python programming language for serial and parallel scientific computations. *Scientific Programming*, 13(1):31–56, 2005.

[4] Jeff Daily. High performance parallel computing in python using numpy and the global arrays toolkit. In *SciPy 2011*, 2011.

[5] L. Dalcn, R. Paz, M. Storti, and J. D'Ela. Mpi for python: Performance improvements and mpi-2 extensions. *Journal of Parallel and Distributed Computing*, 68(5):655–662, 2008.

[6] A. Klckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation. *Parallel Computing*, 2011.

[7] T. Mattson and L. Meadows. A "hands-on" introduction to openmp. page 120.

[8] P. Miller. pympi–an introduction to parallel python using mpi. *Lawrence Livermore National Laboratory report UCRL-WEB-151052*, 2002.

[9] T. Oliphant and C. Banks. Pep 3118 - revising the buffer protocol.

[10] F. Perez and B. E. Granger. Ipython: a system for interactive scientific computing. *Computing in Science & Engineering*, pages 21–29, 2007.

[11] F. Prez, B. Granger, and J. Hunter. Python: an ecosystem for scientific computing. *Computing in Science & Engineering*, (99):1–1, 2011.

[12] R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436. IEEE, 2009.

[13] M. Sala, W. F. Spotz, and M. A. Heroux. Pytrilinos: High-performance distributed-memory solvers for python. In *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*, pages 966–975. Springer-Verlag, 2006.

[14] D. Sverre. Fast numerical computations with cython. In *Proceedings of the 8th Python in Science Conference*, number SciPy, pages 15–22, 2009.

[15] Matt Terry and Joseph Koning. Automation of inertial fusion target design with python. In *SciPy 2011*, 2011.

[16] J. Unpingco. User friendly high productivity computational workflows using the vision/hpc prototype. In *DoD HPCMP Users Group Conference 2008*, pages 387–390. IEEE, 2008.

[17] I. M. Wilbers, H. P. Langtangen, and . degrd. Using cython to speed up numerical python programs. *Proceedings of MekIT*, 9, 2009.