A Python HPC framework: PyTrilinos, ODIN, and Seamless

K.W. Smith
Enthought, Inc.
515 Congress Ave.
Austin, TX 78701
ksmith@enthought.com

W.F. Spotz
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185
wfspotz@sandia.gov

S. Ross-Ross
Enthought, Inc.
515 Congress Ave.
Austin, TX 78701
srossross@enthought.com

***Abstract*—We present three Python software projects: PyTrilinos, for calling Trilinos distributed memory HPC solvers from Python; Optimized Distributed NumPy (ODIN), for distributed array computing; and Seamless, for automatic, Just-in-time compilation of Python source code. We argue that these three projects in combination provide a framework for high-performance computing in Python. They provide this framework by supplying necessary features (in the case of ODIN and Seamless) and algorithms (in the case of ODIN and PyTrilinos) for a user to develop HPC applications. Together they address the principal limitations (real or imagined) ascribed to Python when applied to high-performance computing. A high-level overview of each project is given, including brief explanations as to how these projects work in conjunction to the benefit of end users.**

I. Introduction

For many classes of parallel programming problems that would benefit from parallel libraries, years of experience are required to effectively use those libraries, and frequently they are difficult to use, requiring complicated programming interfaces. The large time investment and limited usability prohibit typical domain specialists, who have limited programming expertise, from creating parallel codes and benefiting from parallel resources. Compounding the difficulty is the need to master the subtleties of a powerful but complex programming language such as C++, along with the mental shift required to program using MPI, OpenCL, or another parallel programming library or paradigm. Programs that can make these time investments typically have huge resources at their disposal, such as the Advanced Simulation & Computing (ASC) campaign, or the Scientific Discovery through Advanced Computing (SciDAC) program. The motivation for industry users to access all the parallel resources at their disposal will only grow with time, as multi-core systems are becoming more widely available on commodity desktop and laptop systems, and it is not far off before hundred-core desktops and laptops are common.

A large fraction of this time investment is spent navigating the complexities of the programming interface, whether it be the Message Passing Interface [1], OpenMP [2], multithreading, or another performance-oriented parallel software technology. The remaining challenge is to sufficiently understand the numerical library routines to solve the problem at hand. A simplified interface to parallel programming libraries allows an end user to leverage the library's performance and drastically decrease the total time from development to solution.

We present three software packages that, in combination, will simplify writing parallel programs and will provide a full HPC framework for scientific computing. End users interact with all three components from the expressive and user-friendly Python [3] language.

- **PyTrilinos** For parallel scientific computing, we provide a high-level interface to the Trilinos [4], [5] Tpetra parallel linear algebra library. This makes parallel linear algebra (1) easier to use via a simplified user interface, (2) more intuitive through features such as advanced indexing, and (3) more useful by enabling access to it from the already extensive Python scientific software stack.
- **Optimized Distributed NumPy (ODIN)** ODIN builds on top of the NumPy [6] project, providing a distributed array data structure that makes parallel array-based computations as straightforward as the same computations in serial. It provides built-in functions that work with distributed arrays, and a framework for creating new functions that work with distributed arrays.
- **Seamless** Seamless aims to make node-level Python code as fast as compiled languages via dynamic compilation. It also allows effortless access to compiled libraries in Python, allowing easy integration of existing code bases written in statically typed languages.

These three components in combination will result in better utilization of existing HPC resources and will facilitate a decrease in the total time to solution, from concept to implementation.

Diverse application domains will benefit: product design and manufacturing, pharmaceuticals, thermal management, network and data transmission design, medical imaging, operational logistics, environmental project planning, and risk and performance assessment, to name some examples.

In the remainder of this paper, we give some background context of the Trilinos project, the advantages of using Python for scientific computing, and then continue with an overview of each of the three projects mentioned above, including code examples.

II. TRILINOS AND PYTRILINOS

Trilinos underlies the PyTrilinos [7], [8] wrappers, and is a set of massively parallel C++ packages that stress performance and capability, sometimes (but not always) at the expense of usability. SciPy is a set of serial Python packages that stress easy-to-learn, powerful, high-level interfaces, sometimes (but not always) at the expense of performance. SciPy is Trilinos' nearest neighbor in the Scientific Python space, in that it is a loosely coupled collection of useful scientific computing packages.

The fundamental enabling technologies of both Trilinos and SciPy are packages that provide object-oriented access to homogeneous, contiguous numeric data. In Trilinos, these packages are Epetra and Tpetra, which are first- and second-generation distributed linear algebra packages. In SciPy, this package is NumPy, a Python module that consolidates and expands upon the early attempts at providing efficient numeric classes for Python.

High-performance computing tools such as Trilinos have been underutilized in the advanced manufacturing and engineering market sectors, in large part because of the steep learning curve associated with using them. NumPy and SciPy, on the other hand, enjoy increasing popularity in these sectors because of their power and ease of use.

The gap between Trilinos and NumPy/SciPy represents an important opportunity. The aim of the expanded PyTrilinos wrappers is to develop a parallel Python module with the distributed capabilities of the second-generation Trilinos Tpetra package that retains, to the extent possible, the ease of use of the NumPy interface.

This effort will build upon the PyTrilinos package, a Python interface of select first generation Trilinos packages. PyTrilinos has, until now, utilized a Python design philosophy of mimicking the C++ interface. This often results in non-Pythonic interfaces that are less intuitive than those found in NumPy and SciPy. With the advent of templated classes in second-generation Trilinos packages, this non-Pythonic nature of the generated interfaces would only be exacerbated. Therefore, a new design philosophy is in order for second-generation PyTrilinos packages. For array-based classes such as `Tpetra::Vector`, the design philosophy that would lead to greatest acceptance in the scientific Python community is clear: make it as much like NumPy as possible.

A. Trilinos

The Trilinos project was developed at Sandia National Laboratories for the National Nuclear Security Administration to facilitate simulation of the national nuclear weapons stockpile. It began over a decade ago as a set of three interoperable packages: Epetra for linear algebra, AztecOO for Krylov space iterative linear solvers, and Ifpack for algebraic preconditioners. It has since grown to a collection of nearly fifty software packages that include additional preconditioners, nonlinear solvers, eigensolvers, general tools, testing utilities, optimization, and a variety of other packages that form the building blocks of important classes of scientific simulation.

Epetra is the first generation linear algebra package. It provides communicators, maps to describe the distribution of data, vectors and multivectors, data importers and exporters, and operators and matrices. It was designed before C++ namespaces and templates were reliably portable, and so these features were not utilized. As a result, vectors and matrices are restricted to double precision scalar data accessible with integer ordinals.

Other first generation Trilinos packages were designed to operate solely on Epetra objects. These include Krylov space iterative linear solvers, a standard interface to a collection of third party direct solvers, algebraic and multi-level preconditioning, nonlinear solvers, testing utilities, and a standard examples package.

Tpetra is the second-generation linear algebra package. Templates allow it to store and operate on arbitrarily typed scalar data, such as floats or complex; and arbitrarily typed ordinals, to allow indexing for larger problems.

Subsequent second-generation packages provide templated interfaces to linear algebra and concrete interfaces to Epetra and Tpetra. These packages include updated versions of the packages described above, plus eigensolvers, partitioning and dynamic load balancing, meshing and mesh quality, discretization techniques, and optimization. Most of these packages are still in early development. As a whole, Trilinos forms a broad foundation for building a wide variety of important scientific simulation applications.

B. PyTrilinos

PyTrilinos is a Trilinos package that provides a Python interface to select first generation Trilinos packages. These interfaces are described briefly in Table I and include Epetra, EpetraExt, Teuchos, TriUtils, Isorropia, AztecOO, Galeri, Amesos, Ifpack, Komplex, Anasazi, ML and NOX. PyTrilinos uses the Simple Wrapper Interface Generator (SWIG) [9] to automatically generate wrapper code. The default SWIG behavior is to generate Python interfaces that shadow the source C++ classes. In many cases, this produces Python code such that the C++ documentation applies to Python as well. In other cases, it results in an interface that is inconsistent with the Python language and non-intuitive to Python programmers.

C. Current Status of PyTrilinos

PyTrilinos provides a fairly complete interface to first generation Trilinos packages. In terms of licensing, over three-fifths of Trilinos packages are BSD-compatible, and the PyTrilinos wrappers are distributed with the same license as the package that they wrap. The first necessary step to supporting second generation packages is to wrap Tpetra. This presents certain interface design issues. As an example, the Tpetra Vector class has the following signature:

```
template<class Scalar, class LocalOrdinal,
                       class GlobalOrdinal>
class Vector {...};
```

Package	Description
Epetra	Linear algebra vector and operator classes
EpetraExt	Extensions to Epetra (I/O, sparse transposes, coloring, etc.)
Teuchos	General tools (parameter lists, reference counted pointers, XML I/O, etc.)
TriUtils	Testing utilities
Isorropia	Partitioning algorithms
AztecOO	Iterative Krylov-space linear solvers
Galeri	Examples of common maps and matrices
Amesos	Uniform interface to third party direct linear solvers
Ifpack	Algebraic preconditioners
Komplex	Complex vectors and matrices via real Epetra objects
Anasazi	Eigensolver package
ML	Multi-level (algebraic multigrid) preconditioners
NOX	Nonlinear solvers

TABLE I
TRILINOS PACKAGES INCLUDED IN PYTRILINOS

The `LocalOrdinal` and `GlobalOrdinal` types support indexing using `long` integers (or any integer type) in addition to the traditional integer indexing. Since the Python integer type corresponds to the C `long` integer, it would be logical to always set `LocalOrdinal` and `GlobalOrdinal` to be `long` within the Python interface. However, the `Scalar` template type presents an opportunity to support a variety of data types, whether real, complex, integer, or potentially more exotic data types as well, just as NumPy does.

This type of interface is under current research and development under a Department of Energy Small Business Innovation Research Phase I grant awarded to Enthought, Inc. Preliminary results, achieved using Cython instead of SWIG, have been encouraging so far, and have led to a Phase II proposal, currently in preparation.

As mentioned in the introduction, the expanded PyTrilinos wrappers are standalone, and will allow an end user to perform massively parallel computations via a user-friendly interface to several Trilinos packages. PyTrilinos will become even more powerful when combined with a general distributed array data structure, allowing an end user to easily initialize a problem with NumPy-like ODIN distributed arrays and then pass those arrays to a PyTrilinos solution algorithm, leveraging Trilinos' optimizations and scalability.

III. ODIN

Scientists approach large array-structured data in two primary ways: from a global perspective (or *mode*), in which the details of the array distribution over a collection of nodes are abstracted and computations are applied on an entire-array basis, and from a local perspective, in which computations are performed on a local segment of a distributed array. MPI, for example, requires users to program in the second model, although there are several MPI routines to allow users to accomplish entire array computations. A frequent challenge in MPI-based programs is to compute entire-array quantities and to perform non-trivial entire-array computations while restricted to node-level operations.

The overall goal of ODIN is to provide a distributed array data structure and associated functions that operate on distributed arrays. Users can interact with these distributed arrays in two primary *modes*, as described above. The first–and most straightforward–mode of interaction is on the global level, in which case ODIN arrays feel very much like regular NumPy arrays, even though computations are carried out in a distributed fashion. The second mode of interaction is on the local level, in which case functions and operations are applied on the local segment of a distributed array, much like MPI. These two modes of interaction are designed to work with each other: the global mode of interaction builds on top of the local computations, and the local computations receive their instructions and data segment specifications from the global mode.

ODIN's approach has several advantages:

- Users have access to arrays in the same way that they think about them: either globally or locally. This flexibility allows one to pick the appropriate level to design and implement an algorithm, and the expression of that algorithm does not have to work around limitations of the language or library.
- As ODIN arrays are easier to use and reason about than the MPI-equivalent, this leads to faster iterative cycles, more flexibility when exploring parallel algorithms, and an overall reduction in total time-to-solution.
- ODIN is designed to work with existing MPI programs–regardless of implementation language–allowing users to leverage tested and optimized MPI routines.
- By using Python, ODIN can leverage the ecosystem of speed-related third party packages, either to wrap external code (f2py for Fortran, Cython, ctypes, SWIG, and several others for C/C++) or to accelerate existing Python code. The Seamless project, described in a later section, is particularly well-suited for this role.
- With the power and expressiveness of NumPy array slicing, ODIN can optimize distributed array expressions. These optimizations include: loop fusion, array expression analysis to select the appropriate communication strategy between worker nodes,

To illustrate how ODIN will be useful for performing distributed array computations in a number of domains, a survey of use cases follow, after which we briefly discuss the current implementation of ODIN.

A. Distributed array creation

All NumPy array creation routines are supported by ODIN, and the resulting arrays are distributed. Routines that create a new array take optional arguments to control the distribution. Some aspects of the distribution that can be controlled are: which nodes to distribute over for a given array, which dimension or dimensions to distribute over, apportion non-uniform sections of an array to each node, and either block,

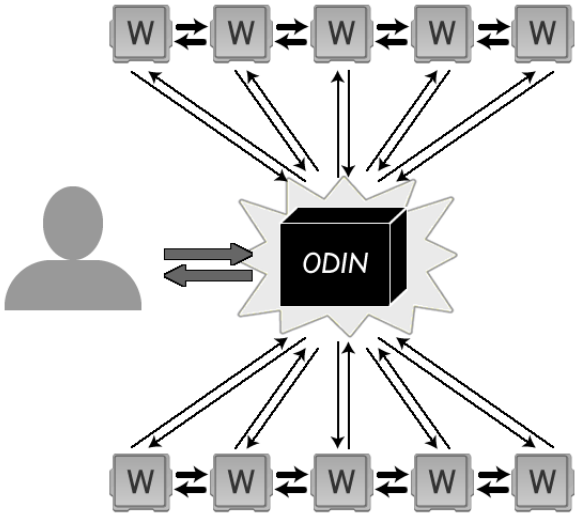

Fig. 1. A schematic depiction of ODIN. The end user interacts with the "ODIN Process", which determines what allocations and calculations to run on the worker nodes, indicated with a "W". The user can specify local functions to run on each worker node, which the ODIN process will send to the worker nodes and make available to the end user via a global function call. The worker nodes can communicate directly with each other bypassing the ODIN process. For performance critical routines, users are encouraged to create local functions that communicate directly with other worker nodes so as to ensure that the ODIN process does not become a performance bottleneck.

cyclic, block-cyclic, or another arbitrary global-to-local index mapping can be specified.

B. Global array operations

The global mode of interaction with ODIN arrays provides a straightforward, NumPy-like interface to distributed array computations. The array creation and manipulation routines are issued by the end user either at an interactive prompt or from a Python script, and these global-level routines treat distributed arrays as globally-addressable entities. Each operation sends one or more messages to the appropriate worker nodes, and these messages control what local array sections are allocated, what index range they cover, or what local operation to perform. These messages are designed to be small operations; very little to no array *data* is associated with them. For instance, when calling the `odin.rand(shape)` routine, a message is sent to all participating nodes to create a local section of an ODIN array with specified shape and with a specified random seed, different for each node. Other necessary information is stored with the local array data structure to indicate the global context in which this local array section is embedded. All array data is allocated and initialized on each node; the only communication from the top-level node is a short message, at most tens of bytes. For efficiency, several messages can be buffered and sent at once for the frequent case when communication latency is significant and when throughput is large.

C. Local array operations

The local mode of interaction with distributed arrays complements the global mode. Users specify, via the `odin.local` decorator, that a function will be run in a single process on a worker node, and arguments to local functions can be the local segment of a distributed array. ODIN's infrastructure will ensure that the global array is distributed appropriately, and that the array segment will be passed to the local function. Here is an example:

```
@odin.local
def hypot(x, y):
    return odin.sqrt(x**2 + y**2)

x = odin.random((10**6, 10**6))
y = odin.random((10**6, 10**6))

h = hypot(x, y)
```

In the above, we define a simple function `hypot` to compute the element-by-element hypotenuse of two ND arrays `x` and `y`. A local function could perform any arbitrary operation, including communication with another node, calling a wrapped serial or MPI routine, or loading a section of an array from a file. In this case, the computation could be performed at the global level with the arrays `x` and `y`; the current example is simply for illustration. The `hypot` function is made local via the `odin.local` decorator. Its purpose is twofold: after the function is defined, it broadcasts the resulting function object to all worker nodes and injects it into their namespace, so it is able to be called from the global level. The decorator's second task is to create a global version of the `hypot` function so that, when called from the global level, a message is broadcast to all worker nodes to call their local `hypot` function. If a distributed array is passed as an argument to the global function, then the ODIN infrastructure ensures that the local function call sees the local segment of the distributed array, as one would expect.

D. Automatically parallelize large NumPy array calculations

NumPy array calculations are often easily parallelized, although NumPy itself does not implement parallelism, leaving it to third party packages [10] and libraries [11], [12]. Being a distributed data structure with support for NumPy array expressions, ODIN does support parallelization of NumPy computations, frequently with little to no modification of the original serial NumPy code. All of NumPy's unary ufuncs are able to be trivially parallelized. Binary ufuncs are trivially parallelizable for the case when the argument arrays are conformable, i.e., when they have the same distribution pattern. For the case when array arguments do not share the same distribution, the ufunc requires node-level communication to perform the computation. A number of different options present themselves in this case, and ODIN will choose a strategy that will minimize communication, while allowing the knowledgeable user to modify its behavior via Python context managers and function decorators.

E. Access Trilinos HPC solvers and other external libraries of parallel algorithms

ODIN arrays are designed to be optionally compatible with Trilinos distributed Vectors and MultiVectors and their associated global-to-local mapping class, allowing ODIN users to use Trilinos packages via the expanded PyTrilinos wrappers described elsewhere in this paper. Just as several libraries are able to be used with NumPy's array data structure, so too can ODIN arrays be used with the massively parallel Trilinos libraries. This will further remove barriers to entry for users who would benefit from Trilinos solvers.

F. Finite element calculations with unstructured meshes

Trilinos has packages for finite element calculations, and ODIN will allow the easy creation, initialization, and manipulation of sparse arrays to be passed to the wrapped Trilinos solvers.

G. Finite difference calculations on structured grid

Another domain, largely orthogonal to finite element calculations, are finite difference calculations on N-dimensional structured grids. This case is nearer to the layout of NumPy's existing N-dimensional array objects, and one can succinctly perform finite difference calculations with NumPy array expressions using slicing syntax. ODIN will support distributed array slicing, which will allow users to do distributed finite difference calculations globally, with a single NumPy-like expression. For example, if `x` and `y` are distributed ODIN arrays, defined thusly:

```
x = odin.linspace(1, 2*pi, 10**8)
y = odin.sin(x)
```

Here, `x` and `y` are distributed ODIN arrays, and each uses a default block distribution, as no other distribution was specified, and `y` has the same distribution as `x`, as it is a simple application of `sin` to each element of `x`.

With these distributed arrays defined, it is trivial to compute the first-order finite-difference derivative of `sin(x)`:

```
dx = x[1] - x[0]
dy = y[1:] - y[:-1]
dydx = dy / dx
```

In the above, we take advantage of the fact that the step size in the `x` array is the same throughout, therefore `dx` is a Python scalar. Handling the more general case of non-uniform step sizes requires a trivial modification of the above. The `dy` array above is another distributed ODIN array, and its computation requires some small amount of inter-node communication, since it is the subtraction of shifted array slices. The equivalent MPI code would require several calls to communication routines, whereas here, ODIN performs this communication automatically.

In this example, we see how the expressive NumPy slicing syntax allows us to perform several operations in a single line of code. Combined with ODIN's distributed arrays, we can perform distributed computations with ease.

H. File IO of distributed data sets

ODIN, being compatible with MPI, can make use of MPI's distributed IO routines. For custom formats, access to node-level computations allows full control to read or write any arbitrary distributed file format.

I. Distributed tabular data

ODIN supports distributed structured or tabular data sets, building on the powerful dtype features of NumPy. In combination with ODIN's distributed function interface, distributed structured arrays provide the fundamental components for parallel Map-Reduce style computations.

J. ODIN current status

We emphasize that this paper gives an overall vision of what ODIN will be–the descriptions thus far do not reference ODIN's current implementation status, for which we give more detail here.

ODIN's basic features—distributed array creation, unary and binary ufunc application, global and local modes of interaction—are prototyped and are currently being tested on systems and clusters with small to mid-range number of nodes. The use cases currently covered by this prototype are simple, and do not yet address array slicing or advanced array communication patterns. The emphasis for this prototype implementation is settling on a distributed array protocol, determining the global and local node APIs, and instrumentation to help identify performance bottlenecks associated with different communication patterns.

Expression analysis, loop fusion, and PyTrilinos support are features that are currently in the design phase. In terms of licensing, ODIN will be released under a BSD-style open-source license.

IV. Seamless

The goals of the Seamless project are fourfold: to provide a Just-In-Time compiler for Python, and specifically, NumPy-centric Python, code; to allow Python to be statically compiled to a useful library for use with other projects; to allow external language constructs (C structs, C functions, C++ classes, etc.) to be easily accessible to Python code; and to make Python into an Algorithm specification language. Seamless accomplishes each of these by building on top of LLVM [13], making its features and flexibility available to Python.

A. Python and NumPy JIT

Just-in-time compilation for Python is not new [14], [15], however, widespread adoption of this technology in the Python world has yet to occur, for reasons beyond the scope of this overview. Seamless' implementation of JIT compilation is different from the cited attempts in that Seamless' approach works from within the existing CPython interpreter, and is not a reimplementation of the Python runtime. This allows a staged and incremental approach, focusing on the parts of Python and NumPy that yield the greatest performance benefits. End

users can access Seamless' JIT by adding simple function decorators, and, optionally, type hints.

```
from seamless import jit
@jit
def sum(it):
    res = 0.0
    for i in range(len(it)):
        res += it[i]
    return res
```

Seamless, via LLVM, compiles Python code to be run on the native CPU instruction set; Seamless' design is such that it will be possible to JIT compile code to run on GPGPU hardware.

B. Statically compile Python

Rather than *dynamically* compiling Python source to machine code via a JIT compiler, Seamless also allows the *static* compilation of Python code to a library that can be used in conjunction with other languages. This feature is intentionally similar to the functionality of the Cython [16] project, which takes type-annotated Python code and generates a source file that can be compiled into an extension module. One primary difference between Seamless' static compilation and Cython is that Cython adds to the Python language to add `cdef` type annotations, while Seamless maintains Python language compatibility[1]. The syntax to accomplish this requires no modification of the JIT example above. The above will work and use type discovery to type `res` as a floating point variable and to type `i` as an integer type. If desired, it is possible to give explicit type information as arguments to a `jit.compile` decorator, and specify that the `it` argument should be restricted to a list of integers, for example. One would use the `seamless` command line utility to generate the extension module.

C. Trivially import external functions into Python

Several existing projects, to varying degrees of automation, help wrap existing code bases written in C, C++, or Fortran and expose them to Python. All require either the explicit specification of the foreign function's interfaces or an explicit compilation step external to Python. Seamless allows users to access foreign functions without an explicit compilation step and without the manual specification of the function's interface in Python. To make use of the `math` C library, one has only to do:

```
class cmath(CModule):
    Header = 'math.h'

libm = cmath('m')
libm.atan2(1.0, 2.0)
```

After instantiating the `cmath` class with a specific library, all of the `math` library is available to use. The call to the `cmath` constructor will find the system's built-in math library, similarly to how the built-in `ctypes` module finds built in system libraries. This feature of Seamless is intentionally similar to the interface of the Ctypes module. An enhancement over Ctypes is that the argument types and return types of the exposed functions are automatically discovered. One has only to specify the header file location in the class body and instantiate the class with a library, and all functions defined in the header file are immediately available for use.

D. Python as an Algorithm specification language

The last high-level feature of Seamless is its ability to allow Python to become a general algorithm specification language whose algorithms can be used from another language, such as C++. This is the inverse of the previous feature, in that one is now accessing Python-defined functions from a statically typed language. This feature allows one to define a general algorithm in the expressive Python language and use it from another language as if it were defined in that language originally.

While this feature may, at first, seem similar to the static compilation feature described previously, it is in actuality something novel. For instance, one can run the following from a C++ source file:

```
#include <seamless>
...
int arr[100];
// initialize arr's contents...
seamless::numpy::sum(arr);
...
std::vector<double> darr(100);
// initialize darr's contents...
seamless::numpy::sum(darr);
```

We emphasize that, in the above example, Seamless allows existing Python code to be accessed *from within a different language* and to be used to perform some computation within that source file. The Python code being used, in this case NumPy's `sum()` function, can be completely unaware of the fact that it is being compiled to C++ code and used from another language. This feature extends Python's reach into a much wider domain.

V. Discussion and Conclusion

We have presented a very high level overview of three standalone projects, each designed to address one core component relevant to High-Performance computing. PyTrilinos provides wrappers to the Trilinos suite of HPC solvers; ODIN provides a flexible and optimized distributed array data structure; and Seamless provides a way to speed up Python code and access pre-compiled libraries written in statically-typed languages. Our vision is that, by using these three projects in concert, users will have a flexible and easy to use framework in which to solve any problem in the high-performance computing domain.

An example use case may go as follows: a user/developer has a large scientific problem to solve. The user allocates,

[1]It should be noted that Cython also has a "Pure Python" mode that supports annotations from within valid Python syntax.

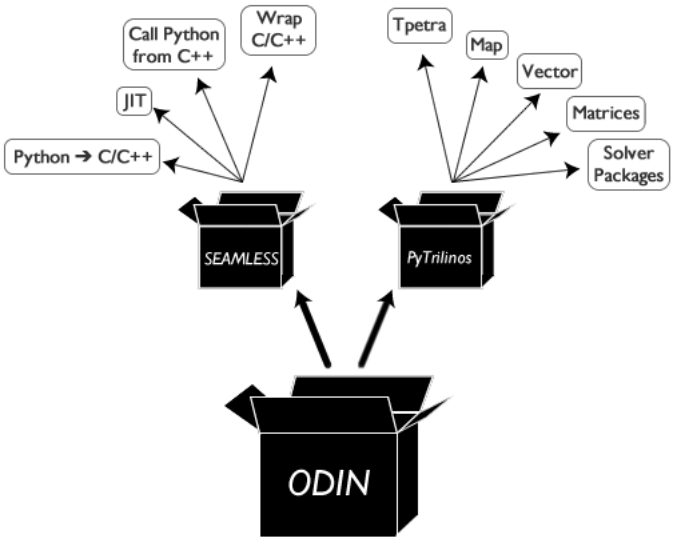

Fig. 2. Schematic relation between PyTrilinos, ODIN, and Seamless. Each of the three packages is standalone. ODIN can use Seamless and PyTrilinos and the functionality that these two packages provide. Seamless provides four principal features, while PyTrilinos wraps several Trilinos solver packages. See the text descriptions for more details.

initializes and manipulates a large simulation data set using ODIN. Depending on the size of the data set and the scaling requirements of the problem, the user may prototype on an 8-core desktop machine, and move to a full 100-node cluster deployment. He then devises a solution approach using PyTrilinos solvers that accept ODIN arrays and chooses an approach where the solver calls back to Python to evaluate a model. This model is prototyped and debugged in pure Python, but when the time comes to solve one or more large problems, Seamless is used convert this callback into a highly efficient numerical kernel.

A user can create a function designed to work on array data, compile it with Seamless' JIT compiler or static compiler, and use that function as the node-level function for a distributed array computation with ODIN. ODIN will allow this function to be accessible as any other globally defined function, and will allow the function to be applied to arrays that are distributed across a large number of nodes. By providing a built-in library of optimized distributed functions, ODIN gives access to a built-in suite of useful routines. For the cases when an end user needs access to distributed HPC solvers found in Trilinos, ODIN arrays are optionally compatible with Trilinos distributed data structures, and will allow easy interaction with these solvers via the PyTrilinos wrappers.

While the PyTrilinos project is more advanced in terms of development, the ODIN and Seamless projects, when feature complete, will fill in the remaining components of the entire framework. This framework removes several commonly-cited limitations of Python when applied to HPC computing:

- **Python is too slow.** Seamless allows compilation to fast machine code, either dynamically or statically.
- **Python is yet another language to integrate with existing software.** Seamless allows easy interaction between Python and other languages, and removes nearly all barriers to inter-language programming.
- **The Python HPC ecosystem is too small.** PyTrilinos provides access to a comprehensive suite of HPC solvers. Further, ODIN will provide a library of functions and methods designed to work with distributed arrays, and its design allows access to any existing MPI routines.
- **Integrating all components is too difficult.** ODIN provides a common framework to integrate disparate components for distributed computing.

We believe the functionality provided by these three projects will provide a compelling reason to consider Python as *the* go-to language for HPC in coming years.

ACKNOWLEDGMENT

The PyTrilinos expansion was funded under a grant from the Department of Energy's Small Business Innovation and Research program for FY2012.

REFERENCES

[1] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 2 ed., November 1999.
[2] "OpenMP." http://openmp.org/wp/.
[3] G. van Rossum and F. Drake, *Python Reference Manual*. PythonLabs, 2001.
[4] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An overview of the Trilinos project," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, 2005.
[5] "Trilinos." `http://trilinos.org`.
[6] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: A structure for efficient numerical computation," *Computing in Science and Engineering*, vol. 13, no. 2, pp. 22–30, 2011.
[7] W. F. Spotz, "PyTrilinos: Recent advances in the Python interface to Trilinos," *Scientific Programming*, vol. 20, no. 3, pp. 311–325, 2012.
[8] "Pytrilinos." `http://trilinos.sandia.gov/packages/pytrilinos`.
[9] "SWIG." http://www.swig.org.
[10] "numexpr: Fast numerical array expression evaluator for Python and NumPy." `http://code.google.com/p/numexpr/`.
[11] "Intel Math Kernel Library." `http://software.intel.com/en-us/intel-mk`
[12] "Automatically Tuned Linear Algebra Software (ATLAS)." `http://math-atlas.sourceforge.net/`.
[13] C. A. Lattner, "Llvm: An infrastructure for multi-stage optimization," tech. rep., 2002.
[14] C. F. Bolz, A. Cuni, and M. Fijalkowski, "Tracing the meta-level: PyPy's Tracing JIT Compiler," in *Proceedings of ICOOOLPS*, ACM Press, 2009.
[15] "Unladen-Swallow: A faster implementation of Python." `http://code.google.com/p/unladen-swallow/`.
[16] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science and Engineering*, vol. 13, no. 2, pp. 31–39, 2011.