Using PyTrilinos

2005 Trilinos Users Group Meeting
1 Nov 2005  3:30-4:30

Bill Spotz
A little python evangelism
Configuring, compiling and installing PyTrilinos
A word about documentation…
**PyTrilinos.Epetra**
  - Communicators and MPI
  - Maps, Vectors, CrsMatrix, etc…
**Other modules**
  - Galeri, Amesos, IFPACK, ML … (and Teuchos)
**Cross-language polymorphism**
**Efficiency issues**
**What needs to be done**
Why Python?

“Why Python?” by Eric Raymond, author of *The Cathedral and the Bazaar*:

http://www.linuxjournal.com/article/3882

Some highlights:

– “Language collector”, preferred Perl
– Initially recoiled at indentation delimiting code blocks
– Recognized Perl’s scalability problems
– Would only use C/C++ for kernel hacking, scientific computing and 3D graphics
– Tried python for writing a GUI front-end configuration tool
– Whitespace as syntax stopped feeling unnatural after ~20 min
– Mastered python in ~1 day: misstep rate ➔ 0, *before* he had learned python’s feature set
– Months after writing code, it was still readable without serious mental effort
My Own Personal Experience

• First OO design project: framework for coupling nonlinear PDEs (test bed for JFNK)
• Alfred Lorber suggested python
• After 1.5 DAYS:
  – I learned python
  – Alfred & I developed Grid and Field classes
  – Base class for PhysicsModules, two implementations from Brusselator problem (1D,FD) plus Dirichlet BCs
  – Base class for Solver that accepts arbitrary # of PhysicsModules, explicit implementation (Euler)
  – Debugged and working
  – Ready for implicit solvers...need for PyTrilinos
What Is So Great About Python?

• Interpreted, interactive, object-oriented
• Remarkable power with clear syntax
• Modules, classes, exceptions, high-level dynamic data types, dynamic typing
• Huge collection of libraries (modules)
• GUIs:
  – X11, Motif, Tk, Mac, MFC, wxWidgets…
• Extensible with compiled languages, embeddable into applications
• Portable
  – UNIX, Windows, OS/2, Mac, Amiga…
• Scalable
• Productivity: as a programmer, you can focus on problem, rather than language issues…rapid prototyping
What is PyTrilinos?

PyTrilinos is a collection of python interfaces to selected Trilinos packages

- Amesos
- Anasazi
- AztecOO
- Epetra
- EpetraExt
- Galeri
- IFPACK
- LOCA
- ML
- New_Package
- NOX
- Triutils
How Do I Build and Access PyTrilinos?

• **Prerequisites**
  – Python 2.3 or higher
  – Python Numeric module
  – SWIG 1.3.23 or higher

• **When configuring Trilinos,**
  – Use `--enable-python[=path]` or `--with-python[=path]`
  – If building NOX, you **must** use `--with-gnumake`
  – Optionally specify `--with-swig=path`

• **Building Trilinos**
  – Those packages that are enabled and have python interfaces should get built when `make` is invoked

• **Installing PyTrilinos**
  – Uses `--prefix=PREFIX` configuration option

• **Using PyTrilinos**
  – Each package is a module under the PyTrilinos package name
  – Use `from PyTrilinos import Epetra, ...`
Documentation: How Do I Use PyTrilinos?

- **Web site:**
  - FAQ, Automated listing of what header files are wrapped, some descriptions of C++/python differences, Users Guide
  - Use the C++ package doxygen pages as a first guide

- **Repository:**
  - Trilinos/packages/PyTrilinos/doc
  - Overview, ACM-TOMS paper, SciPy’05 presentation, Users Guide

- **Python:**
  - `dir()` function: lists contents of a complex python object
  - `help()` function: returns “man page” of a python object based on its documentation string(s)
  - These functions, plus the Trilinos web pages should be sufficient to figure out most of PyTrilinos
• **Communicators**
  
  - `Epetra.SerialComm()` and `Epetra.MpiComm()` supported
  
  - **New communicator: `Epetra.PyComm()`** -- returns most appropriate communicator
  
  - If Trilinos is configured with MPI, then `MPI_Init()` is called when Epetra is imported and `MPI_Finalize()` is registered with `atexit` module
  
  - Global operator methods (Broadcast, MaxAll, MinAll, SumAll, ...) typically take an arbitrary python object and return a Numeric array of doubles, ints or longs (integer return codes are checked internally).
from Numeric import *
from PyTrilinos import Epetra

comm = Epetra.PyComm()
n = comm.NumProc()
data = comm.MyPID() + 1

sum = comm.SumAll(data) # One argument, returns array not int
assert(sum[0] == n*(n+1)/2)

vals = arange(5) # [0, 1, 2, 3, 4]
myVals = vals * data
maxVals = vals * n

maxAll = comm.MaxAll(myVals)
assert(maxAll == maxVals)

minAll = comm.MinAll(myVals)
assert(minAll == vals)
PyTrilinos MPI Support

- Uses standard python interpreter (some python MPI implementations require new python executable: `mpipython`)
- Standard parallel invocation
  - `mpirun -np 4 python script.py`
- Marzio claims:
  - It seems to work with LAM/MPI only
  - It seems to require shared MPI library
  - Easier with GCC-4.0
  - It works fine on MAC OS X 10.4
PyTrilinos.Epetra

• Maps
  - `Epetra.Map()`, `Epetra_BlockMap()` and `Epetra_LocalMap()` are supported

• Vectors
  - `Epetra.Vector()` supports some of the standard constructors, but can also take an arbitrary python object
  - Other classes that *should* support this model:
    - `Epetra.MultiVector`, `Epetra_IntVector`, `Epetra.SerialDenseVector`,
    - `Epetra.SerialDenseMatrix`, `Epetra.IntSerialDenseVector`,
    - `Epetra.IntSerialDenseMatrix`

• Import/Export available
Two Ways of Building an Epetra.CrsMatrix

• The easy way…

```python
from PyTrilinos import Epetra

comm = Epetra.PyComm()
nGlobal = 10 * comm.NumProc()
map = Epetra.Map(nGlobal, 0, comm)
matrix = Epetra.CrsMatrix(Epetra.Copy, map, 0)
myElements = map.MyGlobalElements()

for gid in myElements:
    matrix[gid, gid] = 2.0
    if gid > 0:
        matrix[gid, gid-1] = -1.0  # Like MATLAB!
    if gid < nGlobal - 1:
        matrix[gid, gid+1] = -1.0

matrix.FillComplete()
```
Two Ways of Building an Epetra.CrsMatrix

• The efficient way...

```python
from PyTrilinos import Epetra

comm = Epetra.PyComm()
nGlobal = 10 * comm.NumProc()
map = Epetra.Map(nGlobal, 0, comm)
matrix = Epetra.CrsMatrix(Epetra.Copy, map, 0)
nLocal = map.NumMyElements()

for lid in xrange(nLocal):
    gid = map.GID(lid)
    if gid != nGlobal-1 and gid != 0:
        indices = [gid-1, gid, gid+1]
        values = [-1.0, 2.0, -1.0]
    else:
        indices = [gid]
        values = [1.0]
    matrix.InsertGlobalValues(gid, values, indices) # As Epetra

matrix.FillComplete()
```
The Galeri Package

- The `Trilinos_Util_CrsMatrixGallery` and `VbrMatrixGallery` classes are being replaced by the Galeri package
  - Better documentation, easier to introduce new matrices
  - PyTrilinos interface available
- Several finite difference matrices available
- Examples:

```python
from PyTrilinos import Epetra, Galeri

comm = Epetra.PyComm()
pList = {"n": 100}
map1 = Galeri.CreateMap("Linear", comm, pList)
matrix1 = Galeri.CreateCrsMatrix("Tridiag", map, pList)
map2, matrix2, x, b, exact = Galeri.ReadHB("gre__115.rua", comm)
```
Parameter Lists

• **Goal**: wherever Trilinos expects a parameter list, accept a python dictionary

```python
# Create a Cartesian map, containing nx x ny x NumProcs nodes
comm = Epetra.PyComm()

nx = 2
ny = 2 * comm.NumProc()
pList = {
    "nx": nx,  # Number of nodes in the X-direction
    "ny": ny,  # Number of nodes in the Y-direction
    "mx": 1,   # Number of processors in the X-direction
    "my": comm.NumProc() # Number of processors in the Y-direction
}

map = Galeri.CreateMap("Cartesian2D", comm, pList)
```

• **Currently input only, no sublists**
Linear Solvers and Preconditioners

• **Amesos:**
  – All classes and the Factory are wrapped
  – You can use KLU, SuperLU, SuperLU_DIST, UMFPACK, MUMPS, DSCPACK, TAUCS, PARDISO within Python

• **AztecOO**
  – AztecOO class is wrapped

• **IFPACK:**
  – All capabilities of the Factory class are wrapped

• **ML:**
  – The MultiLevelPreconditioner class is wrapped; therefore all “stable” ML capabilities can be tested within PyTrilinos
  – Still missing the support for Maxwell equations
Cross-Language Polymorphism

• Consider a pure virtual base class such as `Epetra_Operator`
• Python “shadow class” is `Epetra.Operator` (actually `Epetra.PyOperator` for now)
• C++ methods and functions that take a `Epetra_Operator` will expect methods such as `Apply()`, `OperatorRangeMap()`, etc, to be implemented
• Python classes that inherit from `Epetra.PyOperator` can define `Apply()`, `OperatorRangeMap()`, etc. *in python*
• Wrapper generator SWIG has “director” feature that directs callbacks between languages
• Class `Epetra.PyRowMatrix` also implemented
• Proceed step-by-step to localize errors
from PyTrilinos import Epetra, AztecOO

class MyOperator(Epetra.PyOperator):

    def __init__(self, map):
        Epetra.PyOperator.__init__(self, map.Comm())
        self.__map = map

    def Apply(*args):
        LHS = args[1]
        RHS = args[2]
        n = RHS.MyLength()
        RHS[0, 0] = 2.0 * LHS[0, 0] - LHS[0, 1]
        RHS[0, n-1] = 2.0 * LHS[0, n-1] - LHS[0, n-2]
        for i in xrange(1, n-1):
            RHS[0, i] = 2.0 * LHS[0, i] - LHS[0, i-1] - LHS[0, i+1]
        return 0

    def OperatorRangeMap(*args):
        # First argument is always self
        self = args[0]
        return self.__map
Cross-Language Polymorphism, Continued

```python
comm = Epetra.PyComm()
n  = 100 * comm.NumProc()  # Scaled problem size
map  = Epetra.Map(n, 0, comm)
op  = MyOperator(map)

LHS = Epetra.MultiVector(map,1)
RHS = Epetra.MultiVector(map,1)

# Now create an AztecOO solver, and solve the problem using
# the C++ code. Could be done with Epetra_RowMatrix as well.
problem = Epetra.LinearProblem(op, LHS, RHS)
solver = AztecOO.AztecOO(problem)

solver.SetAztecOption(AztecOO.AZ_solver, AztecOO.AZ_cg)
solver.SetAztecOption(AztecOO.AZ_precond, AztecOO.AZ_none)
solver.SetAztecOption(AztecOO.AZ_output, 16)
solver.Iterate(1550, 1e-5)
```

For more examples, see `ml/python/examples/exPyOperator.py` and `exPyRowMatrix.py`
Building a More Efficient Epetra.PyOperator

• C++ code calls a virtual method of Trilinos class
• Using the SWIG director feature and callback facility, the Trilinos wrapper calls a user-written python method
• The python method implements a python loop, which is inefficient
• If efficiency is an issue, you can try slice syntax or use the weave module to have the python method execute compiled C/C++ code
• Typically see factor of 10-100 speedup
Using slice syntax:

```python
def Apply(*args):
    LHS = args[1]
    RHS = args[2]
    n = RHS.MyLength()

    RHS[0, 0] = 2.0 * LHS[0, 0] - LHS[0, 1]
    RHS[0, n-1] = 2.0 * LHS[0, n-1] - LHS[0, n-2]
    RHS[0, 1:-1] = 2.0 * LHS[0, 1:-1] - LHS[0, :-2] - LHS[0, 2:]

    return 0
```
def Apply(*args):
    LHS = args[1]
    RHS = args[2]
    n = RHS.MyLength()

    code = ""
    RHS[0,0 ] = 2.0 * LHS[0,0 ] - LHS[0,1 ];
    RHS[0,n-1] = 2.0 * LHS[0,n-1] - LHS[0,n-2];
    for (int i=0; i<n; i++) {
        RHS[0,i] = 2.0 * LHS[0,i] - LHS[0,i-1] - LHS [0,i+1];
    }
    ""

    weave.inline(code, [‘RHS’,’LHS’,’n’],
                  type_converters = weave.converters.blitz)

    return 0
**Speedup Experiment: \((\psi, \zeta)\)**

<table>
<thead>
<tr>
<th>#</th>
<th>Equation</th>
<th>Method</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>All</td>
<td>Naive</td>
<td>29.12</td>
<td>Baseline</td>
</tr>
<tr>
<td>2</td>
<td>BCs</td>
<td>Slice syntax</td>
<td>26.05</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>BCs</td>
<td>weave</td>
<td>28.90</td>
<td>Slice syntax preferable</td>
</tr>
<tr>
<td>4</td>
<td>(\psi)</td>
<td>Slice syntax</td>
<td>19.99</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(\psi)</td>
<td>weave</td>
<td>19.20</td>
<td>weave preferable</td>
</tr>
<tr>
<td>6</td>
<td>(\zeta)</td>
<td>weave</td>
<td>6.99</td>
<td>Slice syntax complicated</td>
</tr>
<tr>
<td>7</td>
<td>((u,v))</td>
<td>Slice syntax</td>
<td>1.23</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>((u,v))</td>
<td>weave</td>
<td>1.44</td>
<td>More flexible</td>
</tr>
</tbody>
</table>
PyTrilinos Performance vs. MATLAB

- CPU sec to fill $n \times n$ dense matrix

<table>
<thead>
<tr>
<th>$n$</th>
<th>MATLAB</th>
<th>PyTrilinos</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.00001</td>
<td>0.000416</td>
</tr>
<tr>
<td>100</td>
<td>0.0025</td>
<td>0.0357</td>
</tr>
<tr>
<td>1000</td>
<td>0.0478</td>
<td>3.857</td>
</tr>
</tbody>
</table>

- CPU sec to fill $n \times n$ diagonal matrix

<table>
<thead>
<tr>
<th>$n$</th>
<th>MATLAB</th>
<th>PyTrilinos</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.00006</td>
<td>0.000159</td>
</tr>
<tr>
<td>1000</td>
<td>0.00397</td>
<td>0.0059</td>
</tr>
<tr>
<td>10,000</td>
<td>0.449</td>
<td>0.060</td>
</tr>
<tr>
<td>50,000</td>
<td>11.05</td>
<td>0.313</td>
</tr>
<tr>
<td>100,000</td>
<td>50.98</td>
<td>0.603</td>
</tr>
</tbody>
</table>

- CPU sec for 100 MatVecs

<table>
<thead>
<tr>
<th>$n$</th>
<th>MATLAB</th>
<th>PyTrilinos</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.02</td>
<td>0.0053</td>
</tr>
<tr>
<td>100</td>
<td>0.110</td>
<td>0.0288</td>
</tr>
<tr>
<td>500</td>
<td>3.130</td>
<td>1.782</td>
</tr>
<tr>
<td>1000</td>
<td>12.720</td>
<td>7.150</td>
</tr>
</tbody>
</table>
PyTrilinos Performance vs. Trilinos

- **Fine-grained script:**

<table>
<thead>
<tr>
<th>$n$</th>
<th>Trilinos</th>
<th>PyTrilinos</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.010</td>
<td>0.15</td>
</tr>
<tr>
<td>10,000</td>
<td>0.113</td>
<td>0.241</td>
</tr>
<tr>
<td>100,000</td>
<td>0.280</td>
<td>1.238</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1.925</td>
<td>11.28</td>
</tr>
</tbody>
</table>

- **Course-grained script:**
What Needs to Be Done

• Handling *all* methods with C array arguments
  – SWIG library *numeric.i*
  – Package by package, class by class, method by method
• Treating *all* array-type objects as Numeric arrays
• Accepting python dictionaries for parameter lists
  – *(Teuchos::Parameter::List/PyObject)* class
  – Implement *everywhere*
• Handling *Teuchos::RefCountPtrs* transparently
• Test and example scripts
• PyTrilinos refactor
  – Move PyTrilinos classes to *Teuchos*
  – Change PyTrilinos to be a “cleanup” package
• Robustness
  – Portability, parallelism, advanced features
• Documentation
• New Packages…
Some Final Words

• **Python takes the $@::{};! out of programming**

• **PyTrilinos allows you to**
  – Play around with Trilinos objects interactively
  – Quickly develop parallel scripts
  – Rapidly prototype applications
  – Glue Trilinos to other packages easily
  – Developers can write unit tests that cover large parts of the code very quickly

• **SciPy provides tons of scientific computing capabilities**

• **Several excellent plotting packages exist**

• **Legitimate, free, object-oriented alternative to MATLAB**