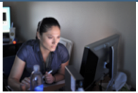
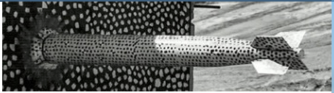
14688C

PyTrilinos2: automatic (re)generation of a Python interface for Trilinos





Presented by:

K. Liegeois & C. Glusa

Trilinos User Group, October 2022

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell

dia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly ow subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Outline

- Motivations for a Python interface for Trilinos;
- Overview of current PyTrilinos;
- Proposed approach for PyTrilinos2;
- Example;
- Future work and discussion.

Motivations for a Python interface for Trilinos

Python is a glue language that allows to:

- ease the prototyping of new methods and applications,
- improve the pre and post processing workflow,
- drive complex software through user-friendly front-ends,
- manage inter-operation between libraries and software written in different languages.

Motivations for a Python interface for Trilinos

In particular, a Python interface for Trilinos could allow to:

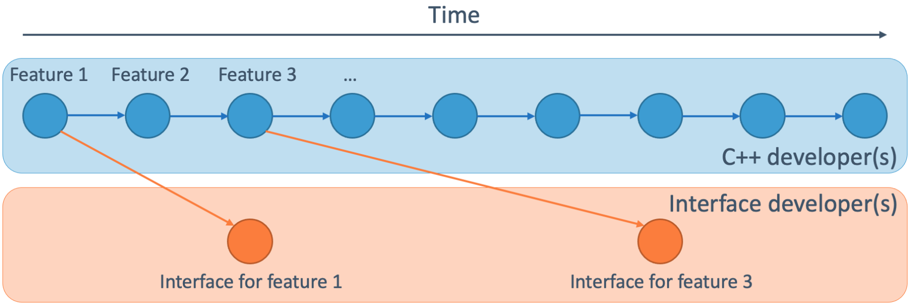
- drive parametric computations from Python:
  - perform uncertainty quantification analysis,
  - optimize parameter values,
  - compute sensitivity w.r.t parameters using finite differences, ...
- prototype new solvers while relying on existing C++ features:
  - new multigrid cycles for MueLu,
  - new mixed precision linear solvers for Belos,
  - new solvers for Ifpack2, ...
- embed Machine Learning (ML) models written in Python into existing C++ code:
  - use a ML model at an integration point to evaluate as a constitutive model while assembling matrices, ...
- lower the learning curve associated to the usage of Trilinos and potentially increase the user base.

Current PyTrilinos: overview

There is a currently available Python interface called PyTrilinos. PyTrilinos:

- provides an interface for a few legacy packages of Trilinos,
- was developed by Bill Spotz,
- uses SWIG to create the interface between C++ and Python,
- wraps mostly Epetra related packages (some Tpetra classes such as `Map`, `Vector`, and `MultiVector` have an interface too).

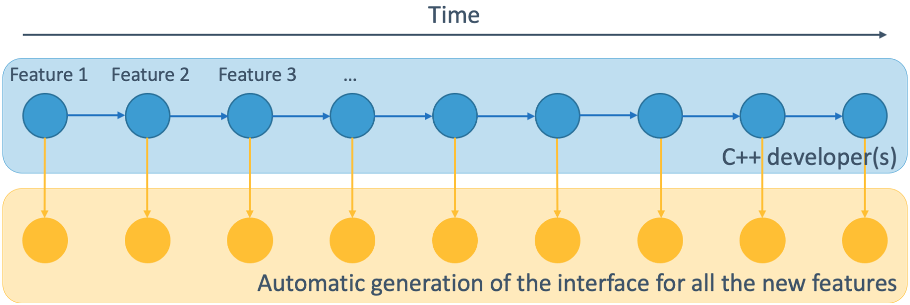
PyTrilinos: overview



Drawbacks:

- Extra language required (SWIG).
- Latency in the wrapping of features in the Python interface.
- Not actively maintained nor tested automatically.

PyTrilinos2: goals for a reboot



Goals:

- Automatic generation of the interface for new packages.
- Automatic update of the interface to reflect C++ changes.
- No latency between new C++ features and their Python interface.

Proposed approach for PyTrilinos2

Instead of relying on the developers to write the interface in SWIG, we propose to rely on the combination of two tools: Pybind11 and Binder.

Pybind11:

- a lightweight header-only library that exposes C++ types in Python and vice versa,
- similar to Boost.Python,
- supports custom smart pointers (such as `Teuchos::RCP`), template arguments, inheritance, ...
- can be easily install using pip.

Binder:

- tool for the automatic generation of Python bindings using Pybind11 and Clang LibTooling libraries,
- generates the Pybind11 lines in `.cpp` files but does not compile them,
- developed at the Johns Hopkins University by Sergey Lyskov.

Pybind11 example

myadd.cpp

```cpp
#include <pybind11/pybind11.h>

int add(int i, int j) {
    return i + j;
}

PYBIND11_MODULE(example, m) {
    m.doc() = "pybind11 example plugin"; // optional module docstring
    m.def("add", &add, "A function that adds two numbers");
}
```

Example of usage:

```python
import example
print(example.add(1, 2))
```

Binder example

```
namespace example {
    // @brief A function that adds two numbers
    int add(int i, int j) {
        return i + j;
    }
}
```

```
#include <pybind11/pybind11.h>
#include <myadd.hpp>

void bind_myadd(std::function< pybind11::module &
 (std::string const &namespace_) > &M)
{
// example::add(int, int) file:myadd.hpp line:5
M("example").def("add", (int (*)(int, int)) &example::add,
 "A function that adds two numbers\n\n C++: example::add(int, int) --> int",
 pybind11::arg("i"), pybind11::arg("j"));
}
```

Example

The presented example uses Teuchos, Tpetra, and MueLu.

Why have we started with Tpetra? Tpetra:

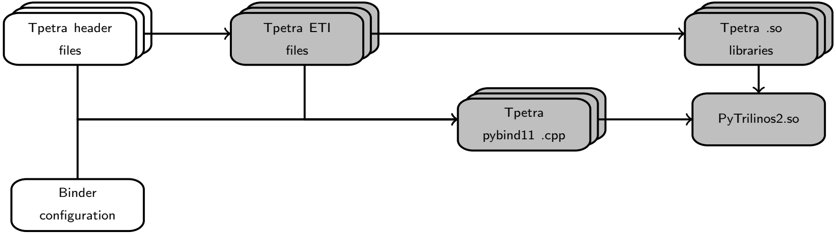
- relies heavily on templates,
- uses MPI communicator,
- is the base package of a stack of solvers,
- relies on Kokkos for shared memory parallelism,
- uses ETI and allows to instantiate multiple instances of one class within one build.

The current work branch of PyTrilinos2 provides binding for `MultiVector`, `Vector`, `Map`, `CrsGraph`, `CrsMatrix`, `Export`, `Import`, ...

Example

Work done to make it work:

- Generalize Binder to accept a custom shared pointer implementation.
- Add a constructor to `Teuchos::RCP` needed for the multiple inheritance using Pybind11: **pybind11/issues/1306**.
- Modify Binder not to use virtual base classes.
- Use the ETI files generated by CMake as inputs for Binder to have binding for the defined symbols.
- Create utility Python functions that return the Python type name associated to a particular instance of a class and default values for the template arguments.



Example: configuration and custom bindings

```
+include <pybind11/stl.h>
+include <pybind11/stl_bind.h>
+include <Teuchos_RCP.hpp>
+custom_shared Teuchos::RCP
+include_for_namespace Teuchos <PyTrilinos2_Teuchos_Custom.hpp>
-function Teuchos::ParameterList::sublist
-function Teuchos::ParameterList::set
-function Teuchos::ParameterList::get
+add_on_binder Teuchos::ParameterList def_ParameterList_member_functions
+include_for_namespace Tpetra <PyTrilinos2_Tpetra_Custom.hpp>
#...
```

```
template <typename T>
void def_ParameterList_member_functions(T cl) {
  cl.def("__setitem__", [](Teuchos::RCP<Teuchos::ParameterList> &m,
    const std::string &name, Teuchos::ParameterList value) { m->set(name,value);  });
  cl.def("set", [](Teuchos::RCP<Teuchos::ParameterList> &m, const std::string &name,
    Teuchos::ParameterList value) { m->set(name,value);  });
  cl.def("sublist", [](Teuchos::RCP<Teuchos::ParameterList> &m, const std::string &name) {
    if (m->isSublist(name))
      return pybind11::cast(sublist(m, name));
    return pybind11::cast("Invalid sublist name");
  }, pybind11::return_value_policy::reference);
  cl.def("__setitem__", [](Teuchos::RCP<Teuchos::ParameterList> &m, const std::string &name,
    pybind11::object value) { setPythonParameter(m,name,value);  });
  //...
```

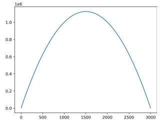
Example: **automatically** generated bindings

```
{ // Tpetra::Vector file: line:292
pybind11::class_<Tpetra::Vector<double,int,long long,Kokkos::Compat::KokkosDeviceWrapperNode<Kokkos::Cuda,
 Kokkos::CudaSpace>>, Teuchos::RCP<Tpetra::Vector<double,int,long long,
 Kokkos::Compat::KokkosDeviceWrapperNode<Kokkos::Cuda, Kokkos::CudaSpace>>>,
 PyCallBack_Tpetra_Vector_double_int_long_long_Kokkos_Compat_KokkosDeviceWrapperNode_Kokkos_Cuda_Kokkos_CudaSpace_t,
 Tpetra::MultiVector<double,int,long long,
 Kokkos::Compat::KokkosDeviceWrapperNode<Kokkos::Cuda, Kokkos::CudaSpace>>>
 cl(M("Tpetra"),
 "Vector_double_int_long_long_Kokkos_Compat_KokkosDeviceWrapperNode_Kokkos_Cuda_Kokkos_CudaSpace_t", "");
 cl.def("dot", [](//...
}
{ // Tpetra::Vector file: line:292
pybind11::class_<Tpetra::Vector<double,int,long long,Kokkos::Compat::KokkosDeviceWrapperNode<Kokkos::Serial,
 Kokkos::HostSpace>>, Teuchos::RCP<Tpetra::Vector<double,int,long long,
 Kokkos::Compat::KokkosDeviceWrapperNode<Kokkos::Serial, Kokkos::HostSpace>>>,
 PyCallBack_Tpetra_Vector_double_int_long_long_Kokkos_Compat_KokkosDeviceWrapperNode_Kokkos_Serial_Kokkos_HostSpace_t,
 Tpetra::MultiVector<double,int,long long,
 Kokkos::Compat::KokkosDeviceWrapperNode<Kokkos::Serial, Kokkos::HostSpace>>>
 cl(M("Tpetra"),
 "Vector_double_int_long_long_Kokkos_Compat_KokkosDeviceWrapperNode_Kokkos_Serial_Kokkos_HostSpace_t", "");
 cl.def("dot", [](//...
}
```

Example: 1D Laplacian solved with CG

```
def main():
    comm = Teuchos.getTeuchosComm(MPI.COMM_WORLD)

    n = 3000
    A = assemble1DLaplacian(n, comm)
    mapT = A.getRowMap()

    x = getTypeName('Vector')(mapT, True)
    b = getTypeName('Vector')(mapT, False)
    b.putScalar(1.)
    pl = Teuchos.ParameterList()
    P = MueLu.CreateTpetraPreconditioner(A, pl)

    CG(A, x, b, P, max_iter=30)
    n0 = 0
    if comm.getRank() == 0:
        n0 = n
    mapT0=type(mapT)(n, n0, 0, comm)

    x0 = getTypeName('Vector')(mapT0, True)
    export = getTypeName('Export')(mapT0, mapT)
    x0.doImport(source=x, exporter=export,\
      CM=Tpetra.CombineMode.REPLACE)

    if comm.getRank() == 0:
        plt.figure()
        plt.plot(x0.getLocalViewHost())
        plt.savefig('x0_view.png')
```

- Solve a 1D Laplacian using CG,
- the CG is written in Python using Tpetra function calls,
- the script can be run with MPI,
- the postprocess is done on the rank 0 after importing the solution vector and getting its local view host (as a NumPy array),
- Kokkos is used under the hood and works with CUDA backend.



Future work and discussion

Future works:

- Create the interface for other packages/features.
- Create a PR with the current work.
- Potentially couple the work with PyKokkos.

Discussion:

- Which team/package is interested into a Python interface?
- What are the thoughts of the Trilinos developer community:
  - on having an interface,
  - on the reboot,
  - and on the automatic (re)generation of the interface?

Example: Tpetra CG

```
def CG(A, x, b, prec, max_iter=20, tol=1e-8):
    r = type(b)(b, Teuchos.DataAccess.Copy)
    A.apply(x,r,Teuchos.ETransp.NO_TRANS,alpha=-1,beta=1)

    p = type(r)(r, Teuchos.DataAccess.Copy)
    q = type(r)(r, Teuchos.DataAccess.Copy)

    Br = type(r)(r, Teuchos.DataAccess.Copy)
    prec.apply(r, p)
    gamma = sqrt(r.dot(p))

    if gamma < tol:
        return 0
    for j in range(max_iter):
        A.apply(p, q)
        c = q.dot(p)
        alpha = gamma**2 / c
        x.update(alpha, p, 1)
        r.update(-alpha, q, 1)
        prec.apply(r, Br)
        gamma_next = sqrt(Br.dot(r))
        beta = gamma_next**2/gamma**2
        gamma = gamma_next
        if gamma < tol:
            return j+1
        p.update(1, Br, beta)
    return max_iter
```

Example: 1D Laplacian

```
def assemble1DLaplacian(n, comm):
    mapT=getTypeName('Map')(n, 0, comm)
    graph = getTypeName('CrsGraph')(mapT, 3)
    for i in range(mapT.getMinLocalIndex(), mapT.getMaxLocalIndex()+1):
        global_i = mapT.getGlobalElement(i)
        indices = [global_i]
        if global_i > 0:
            indices.append(global_i-1)
        if global_i < mapT.getMaxAllGlobalIndex():
            indices.append(global_i+1)
        graph.insertGlobalIndices(global_i, indices)
    graph.fillComplete()

    A = getTypeName('CrsMatrix')(graph)
    for i in range(mapT.getMinLocalIndex(), mapT.getMaxLocalIndex()+1):
        global_i = mapT.getGlobalElement(i)
        indices = [global_i]
        vals = [2.]
        if global_i > 0:
            indices.append(global_i-1)
            vals.append(-1.)
        if global_i < mapT.getMaxAllGlobalIndex():
            indices.append(global_i+1)
            vals.append(-1.)
        A.replaceGlobalValues(global_i, indices, vals)
    A.fillComplete()
    return A
```