Towards Rapid Development of Component Tools at LLNL

Todd Gamblin
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

CScADS Workshop on Tools 2011

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Leveraging existing research work at LLNL is done frequently,

- **Ph.D. students typically work on small research tool projects**
  - Need specific functionality very quickly
  - With more Ph.D. students, assisting all of them with the development work becomes less feasible.

- **Typical performance tool requires a lot of extra coding**
  - Measurement infrastructure
    - PMPI, profilers, hardware counters, timers
  - Tracking layers (MPI Requests, Datatypes, etc)
  - Actual research work is a very small part

- **Building all these tools can be the most time consuming part**
  - Research tools aren't extensively tested, tools are buggy
    - Many spend time debugging others' tools
  - Often made to work for one machine, one set of benchmarks, one app
  - Not many students know how to write a good build system

We are adopting three frameworks to enable more rapid

1. **Using P^{N}MPI for tool integration**
   - Enables us to reuse PMPI measurement modules
   - Allows modules to talk to each other
   - Can rapidly build/test PMPI modules without writing custom shim layer

PnMPI

2. **Modular build system**
   - Using CMake for tool builds
   - Pain of finding, linking, patching PnMPI modules is greatly reduced.

CMake

3. **Wrapper generator for PMPI libraries**
   - Extended existing MPE wrapper generator
   - Added lists, expression language
   - Working on more semantic information in the API

wrap.py

Quick Tool Prototyping with P^{N}MPI

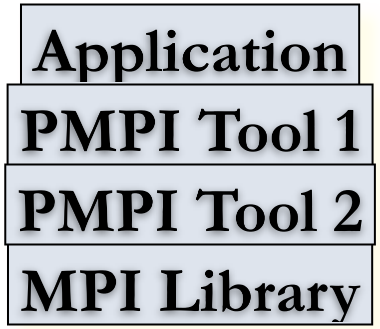
- **PMPI interception of MPI calls**
  - Used by many MPI tools
  - Limited to a single tool

Quick Tool Prototyping with P^{N}MPI

- **PMPI interception of MPI calls**
  - Used by many MPI tools
  - Limited to a single tool

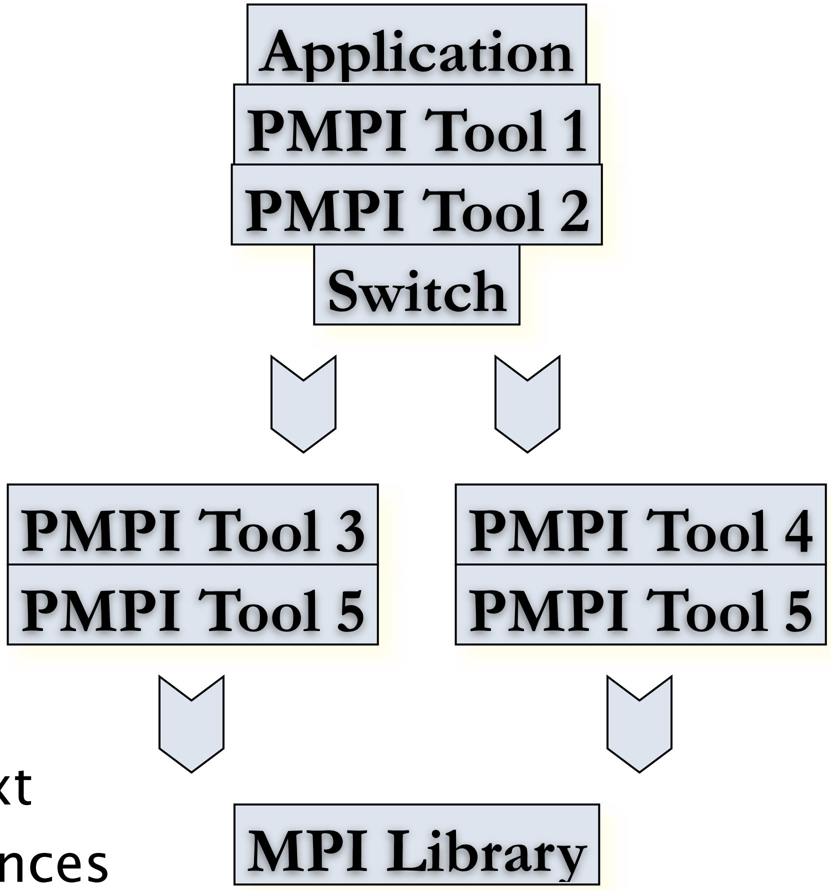
Quick Tool Prototyping with P^{N}MPI

- **PMPI interception of MPI calls**
  - Used by many MPI tools
  - Limited to a single tool
- **P^{N}MPI virtualized PMPI**
  - Multiple tools concurrently
  - Dynamic loading of tools
  - Configuration through text file
  - Tools are independent
  - Tools can collaborate

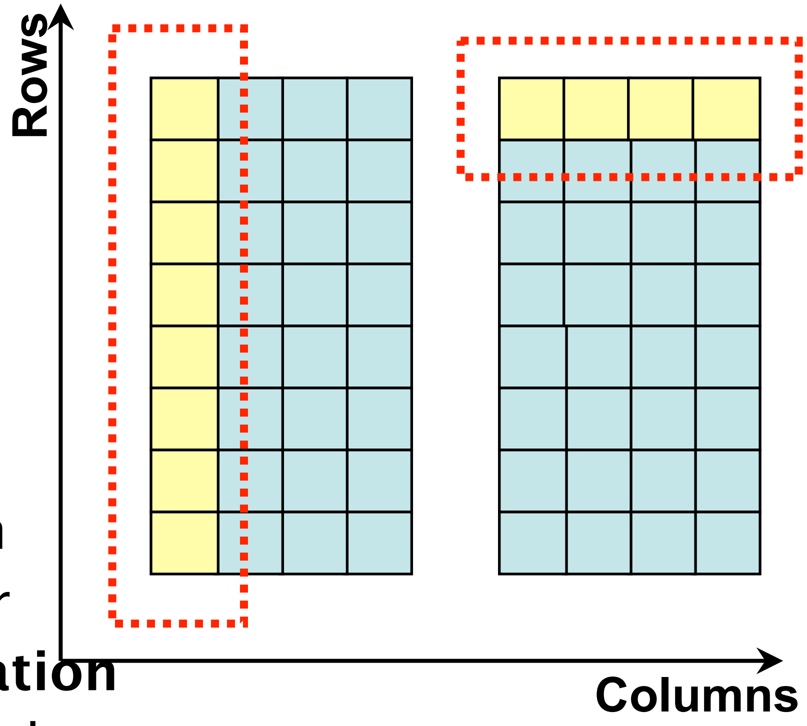


Quick Tool Prototyping with P^{N}MPI

- **PMPI interception of MPI calls**
  - Used by many MPI tools
  - Limited to a single tool
- **P^{N}MPI virtualized PMPI**
  - Multiple tools concurrently
  - Dynamic loading of tools
  - Configuration through text file
  - Tools are independent
  - Tools can collaborate
- **Transparently adding context**
  - Select tool based on MPI context
  - Transparently isolate tool instances


Application
PMPI Tool 1
PMPI Tool 2
Switch
PMPI Tool 3
PMPI Tool 5
PMPI Tool 4
PMPI Tool 5
MPI Library

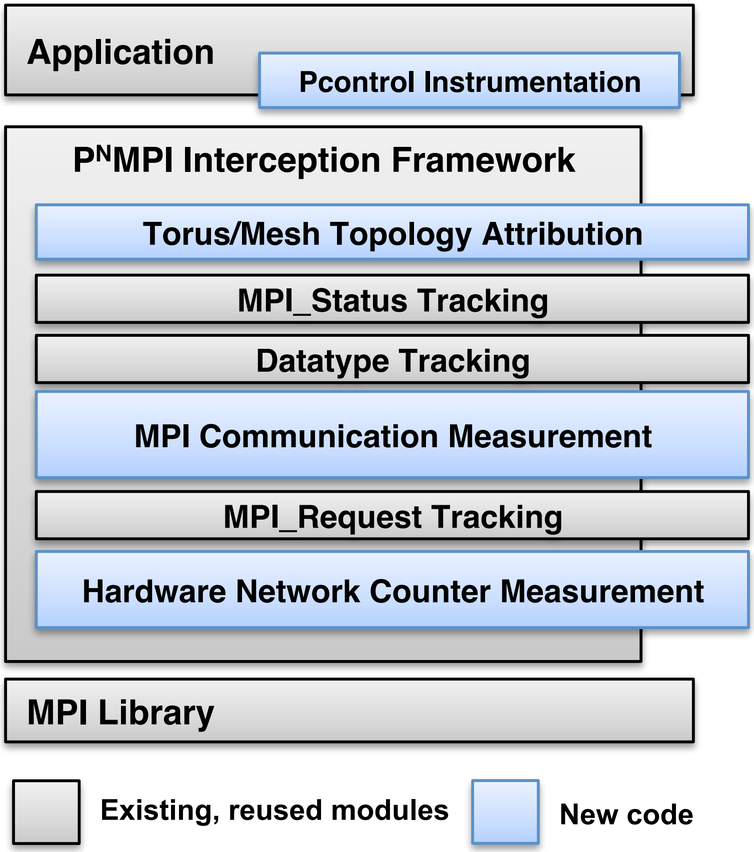
Example: Optimizing an FPMD Code



- **Data structure: dense matrix**
  - Row and column communicators
  - Additional global operations
  - Standard profiles aggregate data
- **Need to profile separately**
  - Potentially different operations
  - May lead to separate optimization
  - BUT: don't want to rewrite profiler
- **Switch module to split communication**
  - Create three independent tool stacks
  - Apply unmodified profiler (mpiP) in each stack
  - Transparent to profiler, application & MPI library

P^{N}MPI allows PAVE tools to be factored into modules and run concurrently

- **We are able to leverage existing communication measurement for P^{N}MPI**
  - BG/P network counter module
  - Request, datatype tracking modules
- **Allows application-specific analysis with minimal additional work**
  - Using existing modules is as simple as adding lines to a configuration file
  - Don't need to modify PMPI code
- **Building and integrating new modules can be painful**



Application

Pcontrol Instrumentation

P^{N}MPI Interception Framework

Torus/Mesh Topology Attribution

MPI_Status Tracking

Datatype Tracking

MPI Communication Measurement

MPI_Request Tracking

Hardware Network Counter Measurement

MPI Library

⬜ **Existing, reused modules** 🟦 **New code**

We use CMake to streamline our build process

- **Finding packages and modules on large systems is difficult**
  - Aren't detected easily by other tools if not part of the system
  - Writing portable, custom m4 for autotools can be painful
  - Keeping versions, LD_LIBRARY_PATHs straight is painful

- **Finding external packages built with CMake is easier**
  - CMake allows projects to export key build information
  - Modules simply tell other modules where to find libraries and headers, rather than requiring the user to do this
  - Exporting this information in CMake is very easy

- **Integrates well with dot kits on LLNL machines**
  - Should also integrate with modules (untested)

Building P^{N}MPI Modules with Make vs. CMake

- **Makefile shown at right:**
  - Requires 4 environment variables to be set by user
  - User must know how to write rules to patch PnMPI modules manually
  - User needs platform-specific knowledge of linking shared libraries
  - User manually writes wrapper generator rules
- **None of this is difficult if you are experienced with builds**
  - **BUT**, it can be very tedious
  - Requires build/link/tool-specific knowledge

```
include $(PNMPIBASE)/common/Makefile.common

MOD     = virtual.so
MPISPEC = mpi_pnmpi
#MPISPEC = mpi_def
WRAPDIR = ../../wrapper
PROTOFILE = $(WRAPDIR)/$(MPISPEC)_proto
FCTFILE   = $(WRAPDIR)/$(MPISPEC)_fct
WRAPPERC = wrapper_c.w
WRAPPERH = wrapper_h.w
WR = ../../wrappergen/wrappergen
CFLAGS += -I$(PNMPI_INC_PATH) -fPIC
CCFLAGS += -I$(PNMPI_INC_PATH) -fPIC

all: $(MOD) install
virtual.so: virtual.o
$(CROSSLD) -o $@ $(SFLAGS) $<
virtual.o: virtual.c virtual.h
$(MPICC) -c $(CFLAGS) $<
virtual.h: virtual.w
$(WR) -p $(PROTOFILE) -f $(FCTFILE) -w $< -o $@
install: $(MOD)
for mymod in $(MOD); do \
(../../patch/patch $$mymod $(PNMPI_LIB_PATH)/$$mymod );
done
clean:
rm -f $(MOD) *.o virtual.h
clobber: clean
rm -f *~
```

Building P^{N}MPI Modules with Make vs. CMake

```
find_package(PnMPI REQUIRED)
Find_package(MPI REQUIRED)

add_pnmpi_module(virtual virtual.c)
add_wrapped_file(virtual.c virtual.w)

install(TARGETS virtual DESTINATION ${PnMPI_MODULES_DIR})

include_directories(
${PnMPI_INCLUDE_PATH}
${MPI_C_INCLUDE_PATH})
```

- **Equivalent CMake file:**
  - No environment variables needed
  - P^{N}MPI is automatically located by the build
    - — P^{N}MPI exports build information, build can simply import this
- **Build uses variables and functions supplied by P^{N}MPI**
  - `add_pnmpi_module()`
  - `add_wrapped_file()`
  - `${PNMPI_MODULES_DIR}` is the install location

Building P^{N}MPI Modules with Make vs. CMake

```
find_package(PnMPI REQUIRED)
Find_package(MPI REQUIRED)

add_pnmpi_modu
add_wrapped_fil

install(TARGETS virtual D

includ
${PnMP
${MPI_
```

This code is provided by PnM

```
function(add_pnmpi_module targetname)
   # Add a library for the module
  add_library(${targetname} MODULE ${ARGN})

  # Patch the library in place once it's built
get_target_property(lib   ${targetname} LOCATION)
get_target_property(patch pnmpi-patch   LOCATION)
   set(tmplib ${targetname}-unpatched.so)

add_custom_command(TARGET ${targetname} POST_BUILD
     COMMAND mv       ARGS ${lib} ${tmplib}
     COMMAND ${patch} ARGS ${tmplib} ${lib}
       COMMAND rm        ARGS -f ${tmplib}
   WORKING_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
        COMMENT "Patching ${targetname}"
               VERBATIM)

  # Make sure that PnMPI lib and patch tool
     # are built before this module.
add_dependencies(${targetname} pnmpi-patch pnmpi)
          endfunction()
```

- **Equivalent CMake file:**
  - No environment variables needed
  - P^{N}MPI is automatically located by th
    - — P^{N}MPI exports build information
- Build uses variables and functions supplied by P^{N}MPI
  - `add_pnmpi_module()`
  - `add_wrapped_file()`
  - `${PNMPI_MODULES_DIR}` is the install location

Exporting build information in CMake projects is simple

Code doing the exporting:

```
add_library(MyModule module.c wrapper.c)
add_wrapped_file(wrapper.c wrapper.w)
install(TARGETS MyModule EXPORT MyModule-libs DESTINATION lib)
install(EXPORT MyModule-libs DESTINATION share/cmake/MyModule)

install(FILES MyModule-config.cmake DESTINATION share/cmake/MyModule)
```

Client project attempting to find above library:

```
find_package(MyModule REQUIRED)
add_executable(myexe myexe.c)
target_link_libraries(myexe MyModule)
```

- **Each project exports a file that supplies build information (library, include locations, etc.)**
  - Other projects can use MyModule’s libraries easily
  - Client projects simply import information from MyProject
    - MyProject location is supplied in environment or at build time
    - Environment variables are easy to set in dotkits or environment modules
- **Makes integration of our own tools simple**
  - No custom m4 scripts needed for CMake projects

Exporting build information in CMake projects is simple

Code doing the exporting:

```
add_library(MyModule module.c wrapper.c)
add_wrapped_file(wrapper.c wrapper.w)
install(TARGETS MyModule EXPORT MyModule-libs DESTINATION lib)
install(EXPORT MyModule-libs DESTINATION share/cmake/MyModule)

install(FILES MyModuleConfig.cmake DESTINATION share/cmake/MyModule)
```

Client project attemp

```
# Various important directories in the PnMPI installation.
set(MyModule_INSTALL_PREFIX          @CMAKE_INSTALL_PREFIX@)
set(MyModule_INCLUDE_DIR             @CMAKE_INSTALL_PREFIX@/include)
set(MyModule_LIBRARY_DIR             @CMAKE_INSTALL_PREFIX@/lib)
set(MyModule_CMAKE_INCLUDE_DIR       @CMAKE_INSTALL_PREFIX@/share/cmake/MyModule)

include(${MyModule_CMAKE_INCLUDE_DIR}/MyModule-libs.cmake)
```

- **Each project ex locations, etc.)**
  - Other projects can use MyModule's libraries easily
  - Client projects simply import information from MyProject
    - MyProject location is supplied in environment or at build time
    - Environment variables are easy to set in dotkits or environment modules
- **Makes integration of our own tools simple**
  - No custom m4 scripts needed for CMake projects

Cmake has more robust compile/link options

- Knows about GNU, Intel, XL, Pathscale, PGI, Visual Compilers
- Full support for rpath
  - Used extensively at LLNL due to number/versions of installed packages
- Platform/compiler/language-specific flags for:
  CMAKE_SHARED_LIBRARY_${lang}_FLAGS
  CMAKE_SHARED_LIBRARY_CREATE_${lang}_FLAGS
  CMAKE_SHARED_LIBRARY_RUNTIME_${lang}_FLAG

- Full control over link line for exe's and libs via
  - CMAKE_${lang}_LINK_EXECUTABLE
  - Useful for special XL/GNU flags used for dynamic executables on BG/P

- Platform support files are relatively easy to write
  - We did BlueGene/P support for static and dynamic libs
    - boost-cmake build for BG/P worked out of the box
  - Cross compiling is reasonably well supported
    - Still need to do hacky things for hybrid builds
  - Compare to libtool!

We have developed a wrapper generator to speed generation of boilerplate code in PMPI

```
#define swap_comm(comm) \
    if (comm == MPI_COMM_WORLD) comm = virtual_comm;

{{fnall fn_name}}
    {{apply_to_type MPI_Comm swap_comm}}
{{endfnall}}
```

Communicator virtualization in 5 lines with wrap.py

- **wrap.py: LLNL Wrapper Generator**
  - Based on wrapper generator in MPE toolkit (came with MPICH 1)
  - Extensible
    - written in python; each wrap.py macro is a python function.
  - Used extensively in the P^{N}MPI build
  - Adopted by Allinea for use in DDT debugger

Basic wrapper generation

Simple code for timing all functions

```
{{fnall foo MPI_Send MPI_Recv}}
  double start_time = get_time_in_nanoseconds();
  {{callfn}}
  double end_time = get_time_in_nanoseconds();
  printf("{{foo}} took %f nanoseconds to run!\n", (end_time - start_time));
{{endfnall}}
```

Wrap just a couple functions to store their addresses in a global:

```
{{fn foo MPI_Send MPI_Recv}}
  // 'foo' here evaluates to just the name of the function.
  my_global_function_pointer = {{foo}};
  {{callfn}}
{{endfn}}
```

- **wrap.py parses mpi.h and extracts info on types, args, of declarations**
  - Has some a priori knowledge
  - Doesn’t require extra prototype files with descriptions of functions
- **Generates both C and Fortran bindings for same functions**
  - Handles special cases like Fortran mpi_init.

Generating non-wrapper code for each MPI function

Generate enum ids for each MPI function

```
typedef enum {
  {{forallfn foo}}
    {{foo}}_id,
  {{endforallfn}}
} mpi_fn_id_t;
```

String ids for all functions

```
{{forallfn foo}}
static const char *{{foo}}_name = “{{foo}}”;
{{endforallfn}}
```

- **These don’t generate wrappers**
  - Allow same iteration over prototypes and type/arg information
- **Can also use these to generate non-C code**
  - Used by Allinea to generate XML API description files for DDT

Simple syntax for lists and list expressions

Simple list of strings

```
{{list foo bar baz}}
```

Some built in lists, and indexing them for particular elements:

```
// Formal params:
{{formals}}
{{formals 0}}
{{formals 1}}
```

```
// Types of formals:
{{types}}
{{types 0}}
{{types 1}}
```

```
// Argument names:
{{args}}
{{args 0}}
{{args 1}}
```

Substitution, and filtering lists with regular expressions:

```
// Get a list of only those formal parameters that have MPI handle types:
{{filter '^MPI_' {{formals}} }}

// replace void with FOO in the first type in the parameter list
{{sub {{types 0}} void FOO}}

// replace any MPI type with MPI_Foo in the parameter list
{{ret_type}} {{foo}}({{zip {{sub {{types}} 'MPI_.*' MPI_Foo}} {{args}} }});
```

wrap.py is easy to extend

```
#define swap_comm(comm) \
    if (comm == MPI_COMM_WORLD) comm = virtual_comm;

{{fnall fn_name}}
    {{apply_to_type MPI_Comm swap_comm}}
{{endfnall}}
```

Communicator virtualization in 5 lines with wrap.py

- Above code swaps out MPI_COMM_WORLD for another communicator
- Allows applications to run in a subpartitionof their MPI allocation
- Easy to implement in Python
  - Other such functions can be added quickly

wrap.py is easy to extend

```
#define swap_comm(comm) \
    if (comm == MPI_COMM_WORLD) comm = virtual_comm;

{{fnall fn_name}}
    {{apply_to_type MPI_Comm swap_comm}}
{{endfnall}}
```

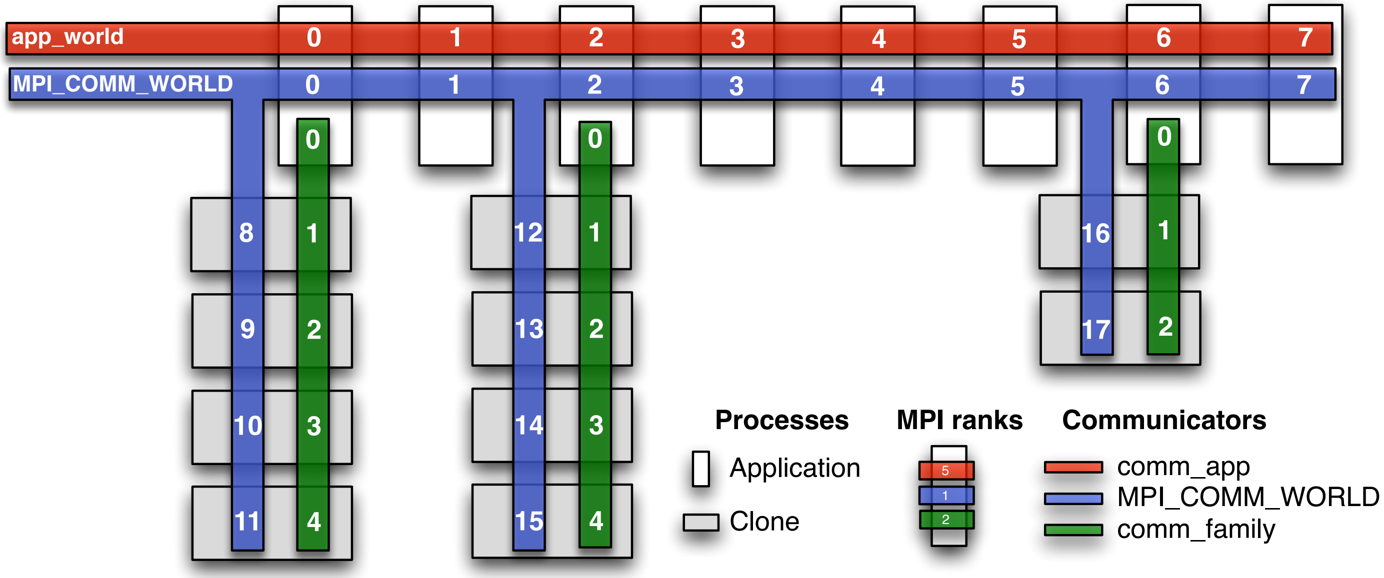
Communicator virtualization in 5 lines with wrap.py

```
class TypeApplier:
    """This class implements a Macro function for applying something
       callable to args in a decl with a particular type.
    """
    def __init__(self, decl):
        self.decl = decl

    def __call__(self, out, scope, args, children):
        len(args) == 2 or syntax_error("Wrong number of args in apply.")
        type, macro_name = args
        for arg in self.decl.args:
            if arg.cType() == type:
                out.write("%s(%s);\n" % (macro_name, arg.name))
```

Python code that implements above macro

We are looking into convenient ways to add more semantics to wrappers



- **MPIEcho** tool was developed in ~2 weeks using wrap.py
  - Allows MPI ranks to be cloned so that heavyweight instrumentation can be spread out
  - Implemented with simple PnMPI virtualization modules in tool stack
- Tool needs semantics of MPI operations in addition to wrapper generation
  - Specific information about args (in parameters, out parameters etc).

Current and Future Projects

- Preparing releases of a number of tool frameworks components using build system described here
  - P^{N}MPI
  - Muster scalable clustering library
  - Nami Wavelet compression library
  - Generic, annotatable Call Tree library
  - Effort library for modeling source code phases/regions
  - Others
  - Libraries used by PAVE project
    - BG/P counters
    - Communication measurement and collective modeling
  - Libraries used by debugging tools
    - Online control flow modeling
- We are extending wrap.py for:
  - Richer semantic information about specific APIs available in the wrapper generator
  - Generic interception of other language runtimes
    - e.g. given a header, wrap every function in it