PMPI SUCCESSOR

RESTART 2017

Code Name QMPI

LLNL-PRES-xxxxxx

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

Why redo PMPI?

- Weak symbol intersection is brittle
  - Depends on the OS
- Limited to a single tool
  - One set of indirections
- Forces tools to be monolithic
  - Single shared library

Requirements

- Support multiple concurrent tools in a single process
- Link time or runtime enablement
- Low to no overhead when no tool is attached
  - Possibility for zero overhead when disabled
- No loss of functionality compared to existing PMPI
  - Complete coverage of all MPI routines (except where exempted already in MPI 3)
  - Complete coverage of all tool functionality
    - Incl. changes to parameters or routines used
  - All language bindings (C, mpif.h, use mpi, use mpif08)
- Support for pre and post call activation
  - Basically wrapper functionality
- Tools can implement functionality in C (in one place) regardless of language bindings used
- Integration with MPI thread support

QMPI:
A VIEW FROM THE TOOL

Focus: Interface the tool would see

Let’s ignore:
Configuration issues, Initialization, Complex Tool DAGs, …

Groundrules

- Maintain the wrapping concept
  - Only way to maintain all functionality
- Interface and all tools will be C only
  - Fortran mapping has to be taken care of by the MPI library
  - Tools are responsible for matching runtimes when using C++
- Support for simple tool stacks only
  - Nested wrapping
  - No DAGs, no inter-tool services, no coordination, no dependencies
- Tools are by default independent of each other
  - Can run on their own
  - Can assume a standard compliant implementation of MPI routines to build upon (the old PMPI interface)
  - Have to provide a standard compliant implementation of MPI, which consists of:
    - Wrapped routine (defined MPI routines – or close to, see below)
    - MPI routines not wrapped (when MPI=PMPI)

Tools Responsibilities

- Tools implement wrapper routines
  - Set of routines with well defined names
    - One name defined for each MPI routine
    - Extended prototype with extra fields
    - E.g.: QMPI_Comm_rank(MPI_Comm comm, int *rank, <extra data>)
  - Tool routines must call “follow on routine”, the match to the old PMPI
    - Assigned by MPI during initialization
    - Can be actual MPI or another tool – transparent
  - Routines not implemented are handled by MPI
    - Automatically set to a “follow-on” routine
- The implemented wrappers combined with all default routines (which can be assumed to adhere to the MPI standard) must again implement a coherent version of the MPI standard

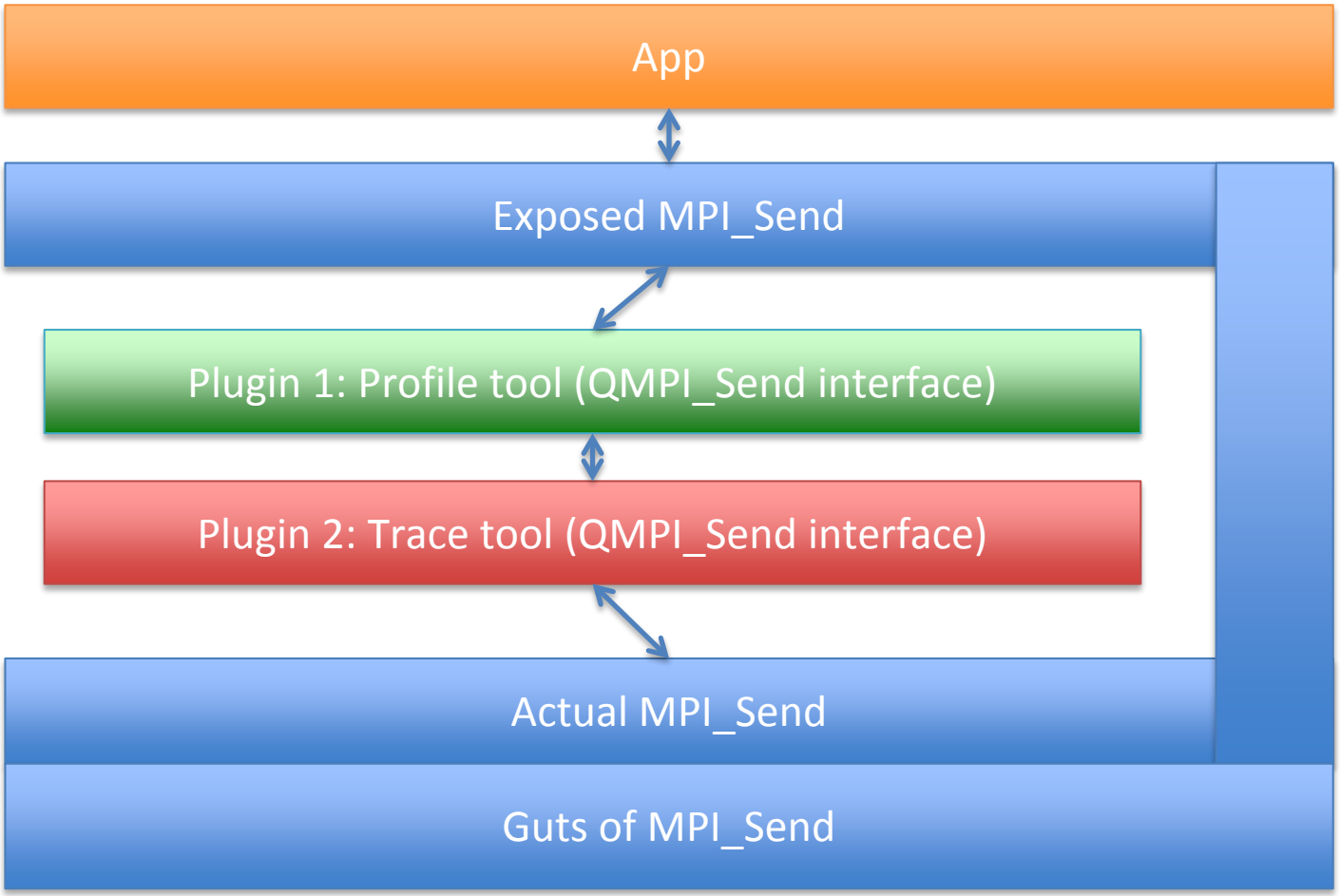
Basic Wrapping

- Each tool implements a set of routines it wraps
- Tools have independent instances
  - Separate storage space
- Each tool instance has the following “available”:
  - A functional table with all “PMPI” / follow on routines
  - A pointer to store internal information
- Wrapping process:

```
Int QMPI_X(…)
{
        qmpi_x_t pqmpi_x;
        MPI_Table_query(“QMPI_X”, &pqmpi_x,table);
        ... Do work ...
        err=pqmpi_x(...);
        ... Do work ...
        return err;
}
```

- Note: query routine should be called once during initialization

Multiple Tools



Magic Data Available to Tools

- Each tool instance needs two pieces of data
  - Pointer to its own state
  - Pointer to a table of PMPI functions to call
- Question where does it get that data from?
  - Pointer to instance state must be passed in from previous tool
    - Pointer must be used to store PMPI function table
    - Pointer must be used to store pointer to instance for follow on tool
  - Basically: context for all tools gets handed down
- Where does this data come from originally?
  - Setup during intialization routine
    - Passes in context for follow-on tool
    - Passes in PMPI function table
  - Tools then stores that information

Concept 1: Function Tables

- Used to store one function pointer for each MPI function
  - Describes complete interface
- Tools can use that to query “follow-on functions”
- Specified as opaque objects

APIs for Function Tables

- ~~Create a Table~~
  - ~~Out: table handle~~
- ~~Destroy a Table~~
  - ~~In/Out: table handle~~
- ~~Copy a Table~~
  - ~~In: src and dst table handles~~
- Query one Function
  - In: table handle, function ID or name
  - Out: function
- Set one Function
  - In: table handle, function ID or name, function
- Unset one Function
  - In: table handle, function ID or name

- Needs enumeration of all MPI functions
  - Alternative: strings (?)
- Likely to contain other tool routines (init/finalize/…)

Concept 2: Tools and Instances

- Tools expose a set of intercepted routines
  - Typically implemented as a shared library
  - Magically gets loaded
    - Let’s discuss that later
  - Doesn’t do anything by itself
    - No automatic interception
    - Just provides routines
- Tool instances are built from tools
  - Use the set of intercepted routines
  - Instances have their own context
    - Instances don’t share context
  - Context = void pointer to store information

Bringup steps (before MPI_Init)

- Step 1: tools get registered
  - This loads the tools
  - MPI now knows about them
- Step 2: tools get initialized
  - Make their wrapped functions known
  - Provide meta data
- Step 3: MPI creates a list of tool to instantiate
  - Let’s leave that to magic
  - Result is an ordered list of tools to create instances for
- Step 4: tools instances get initialized
  - Tools get context
  - Done bottom up (see late why)

Step 1: Tool Registration

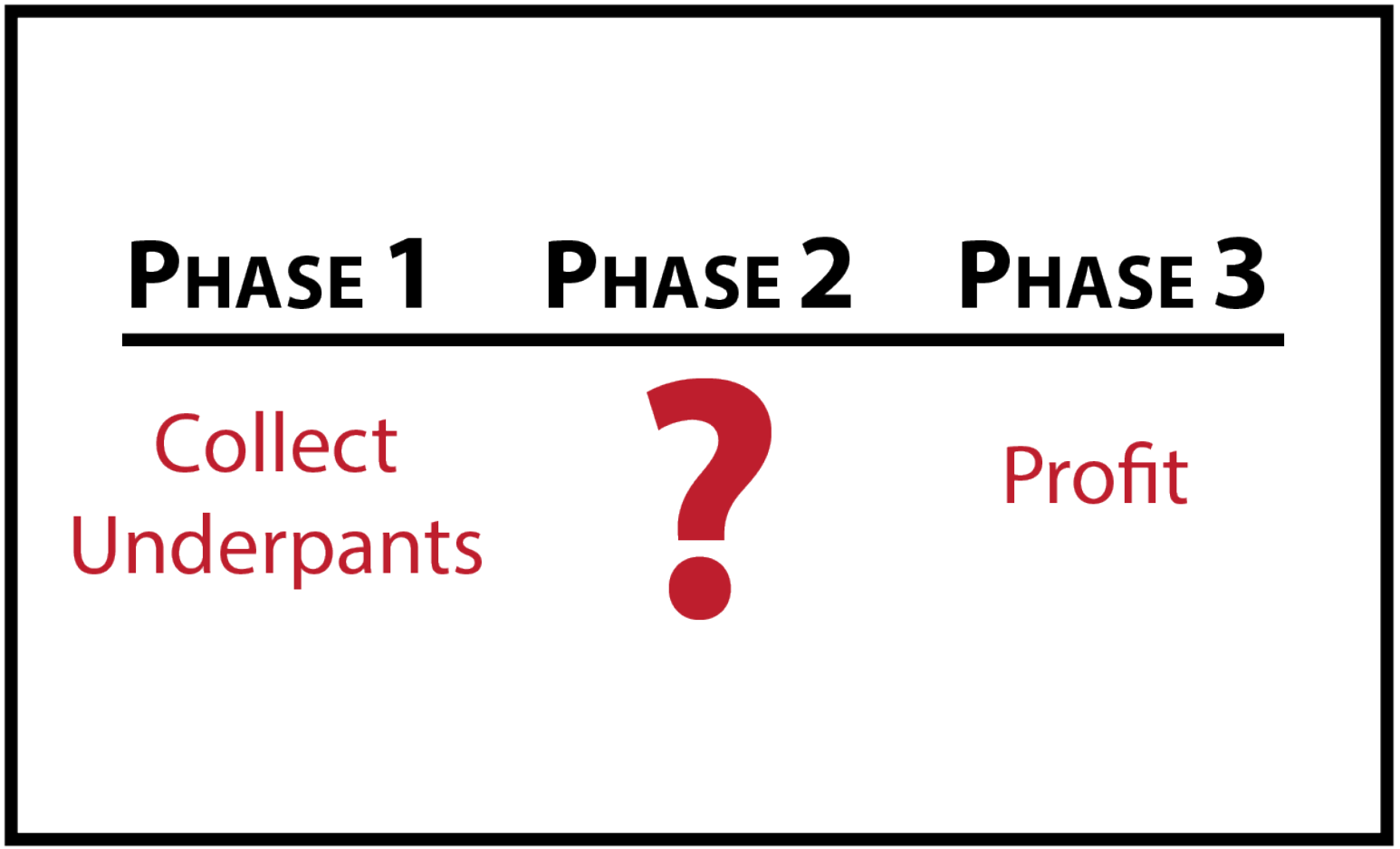
- Option 1 / Callback
  - MPI searches for available intercepts and remembers them
  - Tool implements a well defined callback routine (special QMPI routine)
  - Routine gets called from MPI
  - Tools fills out a “who am I” record and returns it
  - Returns pointer to initialization routine
- Option 2 / Self registration
  - Tool gets “magically” initialized (ini routines)
  - Tool registers itself by calling the MPI function for tool registration
  - Tool passes that table to MPI
  - Tools fills out a “who am I” record and passes it
  - Provides pointer to initialization routine

Should we allow (or enable for the future) dynamic registration?
What about dynamic addition of routines?

Step 2: Tool Initialization

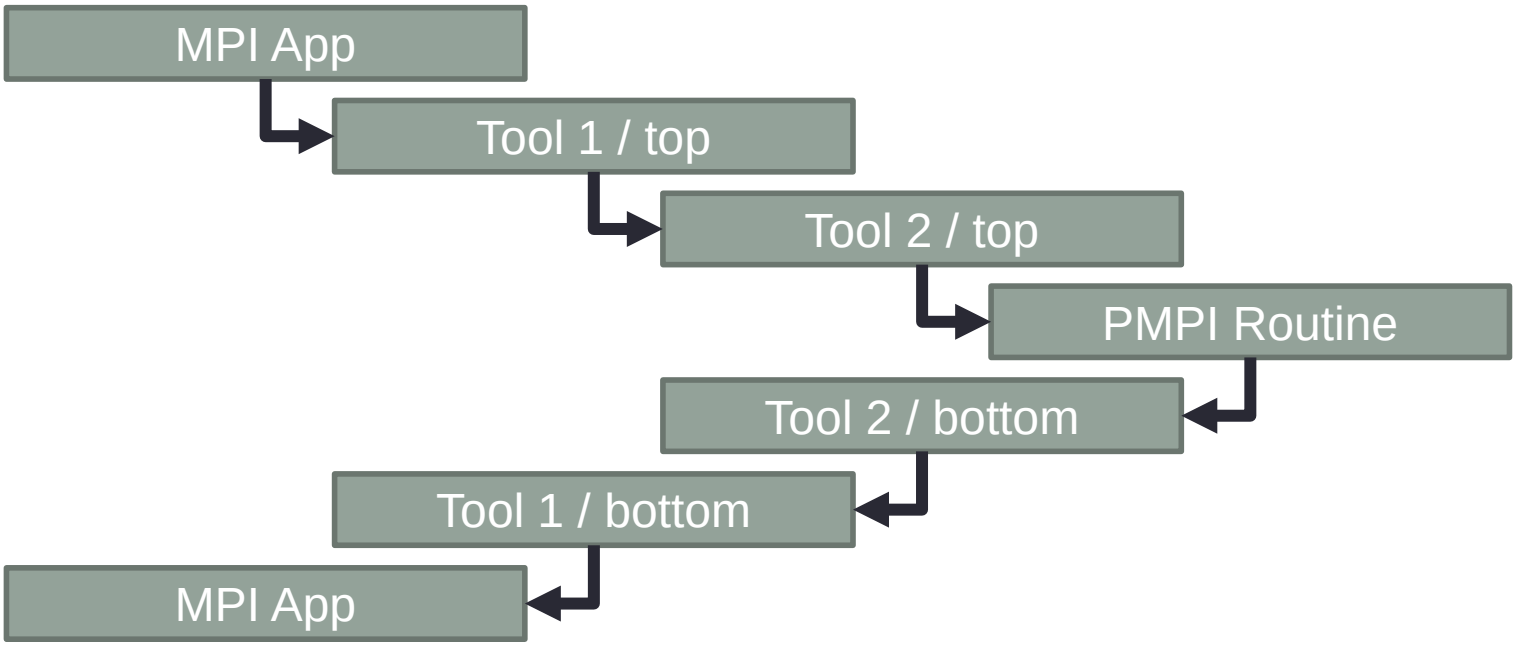
- MPI calls initialization routines
  - Arbitrary order
  - MPI passes function table
    - Represents all intercepted routines MPI could find
      - Typically used for option 1
    - Can be manipulated during the initialization
      - Typically used for option 2
- This step could be combined with Step 1

Step 3: Magic



Step 3: Magic

- MPI decides on a list of tool instances to use
  - Creates a tool stack
  - Mechanism open (e.g., Env. Variable)
- These tools are nested



Step 4: Tool Instance Initialization

- Each tool gets initialized in reverse order
  - Instance initialization routine stored in function table for tool
- Initialization routine gets passed
  - A void pointer to store tool instance context
  - An opaque void pointer
    - This is the context for the follow on tool
  - A function table containing the PMPI routines
    - Who owns this table?
    - When can it change and who can do this?
    - Or can this just be a tool ID
- Tool responsibilities
  - Create storage structure and hook it to void pointer
  - Store opaque pointer and function table in that structure

Wrapping

```
Int QMPI_X(<MPI arguments>, void* context, MPI_blob blob)
{
        retrieve function table from context -> next_table
        retrieve opaque pointer from context -> next_context
        qmpi_x_t pqmpi_x;
        MPI_Table_query("QMPI_X", &pqmpi_x,next_table);
        ... Do work ...
        err=pqmpi_x(<MPI arguments>, next_context, blob);
        ... Do work ...
        return err;
}
```

- Note: query routine should be called once during initialization
- Blob contains MPI internal information, such Fortran flag

What Needs to be Standardized

- Function Tables
  - Type and access function
- Shadow API
  - All MPI functions with slight modifications
  - ”Chance to get it right”
- Initialization routines for tools and tool instances
  - Includes the “who am I” record
- Query routines
  - Ability to see which tools (and tool instances?) are present
- Specification of tools to load
  - Do we need that or is that out of the specification?

Issues

- Transition between PMPI and QMPI
  - Deprecate PMPI
  - For a while this needs to be interoperable
    - PMPI is likely on top of the new infrastructure
    - In intercepts the actual MPI calls from the app
- Threading support
  - What if tool needs thread support itself
- Who am I field
  - Versioning
  - Unique name

Dynamic Tools / Open Issues

- Changing “next tables”
  - When can it happen
  - Who gets notified
  - Thread safety
- Probably should have dynamic updates in tables
  - Opaque objects may be too slow
- Newly loaded tools need to be initialized
  - Need a barrier?
- What about removing
  - Can a tool just finalized itself?

Now What About Complex Tools?

- We do want full P^{n}MPI functionality
  - Tool DAGs / Diamond stacks
  - Services for intra tool communication
  - Support for cooperating tool modules
- BUT: this is not MPI's task
  - Research
  - Requires external specification
- However, this proposal allows developers to build tools that
  - Are implemented as a tool themselves
  - Provide this functionality
  - Can integrate existing tools without changes
    - No more patching like in P^{n}MPI

Complex Tool Use Cases

- This should be implementable by a new tool that uses the interface, without changing the interface
- Only run on certain nodes/processes/communicator
  - Argument for centralized table registry?
- Create arbitrary tool DAGs