LLNL-PRES-470231

The MPIT Interface

MPI-3 Tools Working Group

Draft Presentation: February 2011

- Ticket #266
  - MPIT Tool Interface / Reorganization of the Profiling Chapter
  - Draft Document Available At: https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/MPI3Tools/mpit

MPIT Goals

- Provide access to MPI internal information
  - Configuration and control information
  - Performance information
  - Debugging information
- Standardized access to this information
  - Build on top of the success of the PMPI interface
  - Integrate MPIT into the MPI standard
- MPIT is an MPI implementation agnostic specification
  - No particular implementation model assumed
  - Ability to provide no/varying amount of information
  - Support for production vs. debug versions of MPI

General Approach

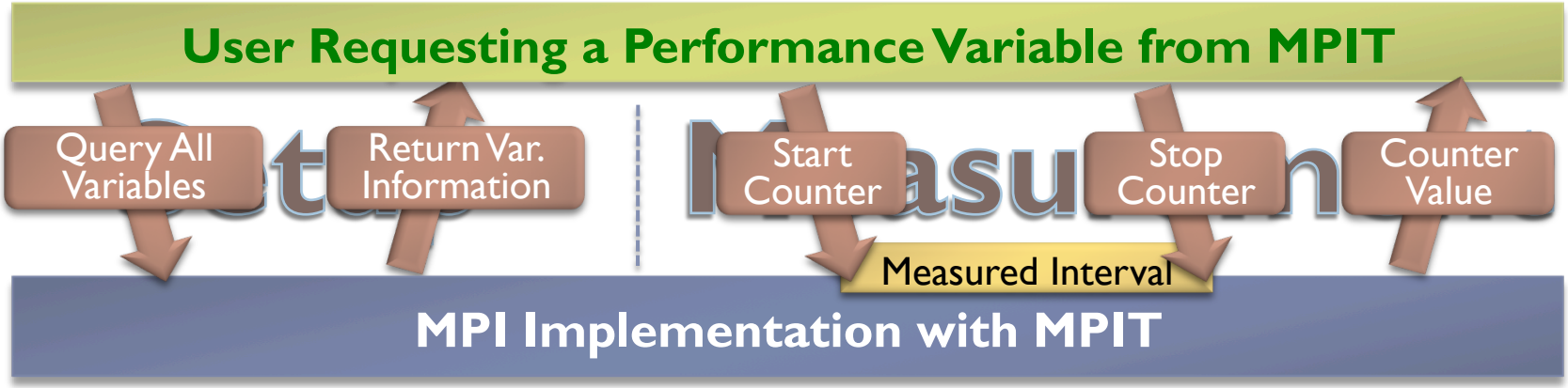
- Basic concept: a set of named variables
  - MPIT provides query functions to detect variables
  - Routines to read and write values of these variables
  - Variables split into performance and control variables

Examples of Performance Vars.

- Number of packets sent
- Time spent blocking
- Memory allocated

Examples of Control Vars.

- Parameters like Eager Limit
- Startup control
- Buffer sizes and management



Outline

- MPIT Basics
  - General workflow
  - Basic (abbreviated) calls
- MPIT Details
  - Initialization
  - Type system
  - Variable verbosity
  - Complete query calls
  - MPIT categories
  - Return codes
- The MPI Tools Chapter
  - Combining PMPI and MPIT into a single chapter

Page Number in Jan. 27 draft

Text taken from proposal draft

Example code

Page 7+ & 12+

General Concepts

- Information exported through variables
  - Two separate classes: control and performance variables
  - Variables are indexed 0 to N-1 (per class)
  - Value represented by datatype and count (similar to messages)
  - Each variable has a name (mandatory) and description (optional)
  - Each variable has metadata & attributes to describe its properties
- Workflow
  - Query number of variables N
  - Loop through all variables 0 to N-1 to find variable
    - Query name and compare
  - If necessary, bind a variable to an MPI object, then get a handle
  - Use handle to access the variable (read, write, …)

Querying Control Variables

- **int MPIT_Controlvar_get_num(*num)**
  - Returns the number of control variables available (in total)

- **int MPIT_Controlvar_get_info(index, name, namelen, verbosity, datatype, count, desc, desclen, bind, attr)**
  - Provides additional information / meta data about a variable
  - Variable specified by "index"
  - Return mandatory name and optional description
  - Report variable's verbosity and attributes (more on that later)
  - Variable specification through datatype/count pair
  - Information on which object this variable needs to be bound

Example: List All Control Variables

```
int list_all_control_vars() {
    int i, num, namelen, bind;
    char name[100];
    MPIT_Controlvar_attributes attr;

    err=MPIT_Controlvar_get_num(&num);
    if (err!=MPIT_SUCCESS) return err;

    for (i=0; i<num; i++) {
        namelen=100;
        err=MPIT_Controlvar_get_info(i, name, &namelen,
            MPIT_VERBOSITY_USER_BASIC, &datatype, &count,
            NULL, NULL, /*no description */
            &bind, &attr);
        if (err!=MPIT_SUCCESS) return err;
        printf("Var %i: &s\n", i, name);
    }
    return MPIT_SUCCESS;
}
```

Concept of Binding to MPI Objects

- Some variables have limited scope and only refer to state of individual MPI objects, e.g.,
  - Message traffic on a given communicator
  - Remote accesses to a specific RMA window
  - I/O buffer settings for a particular MPI file
- Approach: bind variables to MPI objects
  - Variable represent a general concept/template across all objects
  - Binding creates a specific instance
  - Object type returned by *_get_info calls
- Otherwise: would need separate variable for each object
  - Semantics do not allow a reuse of variable indices
  - Unlimited growth of MPIT variable space

Supported MPI Object Types

MPIT Constant	Corresponding MPI Object
MPIT_BIND_GLOBAL	N/A – global
MPIT_BIND_MPI_COMMUNICATOR	MPI communicators
MPIT_BIND_MPI_DATATYPE	MPI datatypes
MPIT_BIND_MPI_ERRORHANDLER	MPI error handlers
MPIT_BIND_MPI_FILE	MPI files
MPIT_BIND_MPI_GROUP	MPI groups
MPIT_BIND_MPI_INFO	MPI info object
MPIT_BIND_MPI_OPERATOR	MPI reduction operators
MPIT_BIND_MPI_REQUEST	MPI request objects
MPIT_BIND_MPI_WINDOW	MPI one-sided communication window

- List can easily be extended by adding constants
- Discussion on MPI_Status, but omitted since user allocated

Control Variable Handles

- Need to acquire a handle before using a variable
  - Includes the binding process
- **int MPIT_Controlvar_handle_allocate(index, void *object, MPIT_Controlvar_handle *handle)**
  - Request a handle for the variable specified by "index"
  - Bind to object specified by "object"
    - Pointer to object handle cast to void*
    - Caller is responsible that the right object type is used
  - Return handle

Page 9

- **int MPIT_Controlvar_handle_free(handle)**
  - Free handle and release resources
  - Set handle to MPIT_CONTROLVAR_HANDLE_NULL

Page 10

Using Control Variables

- **MPIT_Controlvar_read(handle, void *buf)**
  - Reads the variable instance referred to by handle
  - Buffer buf treated similar to MPI message buffers
  - Datatype and count provided by corresponding _get_info call

- **MPIT_Controlvar_write(handle, void *buf)**
  - Writes data to the variable instance referred to by handle
  - Buffer buf treated similar to MPI message buffers
  - Datatype and count provided by corresponding _get_info call
  - If write is not possible, MPIT will return an error
  - Control variables can have a local or global rank scope
    - If variable has a global scope, user is responsible to provide consistency

Example Get Value of a Control Variable

```
int getValue_int_comm(int index, MPI_Comm comm, int*val) {
    int err;
    MPIT_Controlvar_handle handle;

    /* Could add check if variable index can be bound to a communicator */

    err=MPIT_Controlvar_handle_allocate(index,&comm,&handle);
    if (err!=MPIT_SUCCESS) return err;

    /* Could add check if variable is represented by an integer */

    err=MPIT_Controlvar_read(handle,val);
    if (err!=MPIT_SUCCESS) return err;

    err=MPIT_Controlvar_handle_free(&handle);
    return err;
}
```

Performance Variables

- Performance variables are similar to control variables
  - Calls are prefixed with MPIT_Perfvar_...
  - Similar calls to query the number of variables and their metadata
  - Same concept of handle allocation and binding
- Differences
  - Variables are accessed within "sessions"
    - Isolate multiple uses of MPIT
  - Variables can be started and stopped
    - Enable and disable recording of state
    - Intended for easier implementation of calipers and to reduce overhead
  - Performance variables are grouped into classes
    - Provides basic semantic information about the variable's behavior
    - Specifies types used for this variable and starting values

Overview of Variable Classes

Variable Class	Types	Initial
MPIT_PERFVAR_CLASS_STATE Current state of the library or object	Enum	n/a
MPIT_PERFVAR_CLASS_RESOURCE_LEVEL Absolute usage of a resource	all types	n/a
MPIT_PERFVAR_CLASS_RESOURCE_PERCENTAGE Percentage usage of a resource	float types	n/a
MPIT_PERFVAR_CLASS_RESOURCE_HIGHWATERMARK High water mark of resource usage	all types	current usage
MPIT_PERFVAR_CLASS_RESOURCE_LOWWATERMARK Low water mark of resource usage	all types	current usage
MPIT_PERFVAR_CLASS_EVENT_COUNTER Counts the number of times an event occurs	integer types	0
MPIT_PERFVAR_CLASS_EVENT_AGGREGATE Aggregates parameters passed to events	all types	0
MPIT_PERFVAR_CLASS_EVENT_TIMER Aggregate time spent executing an event	float types	0

Complete List of Classes 1/2

- MPIT_PERFVAR_CLASS_STATE
  - Describes the current state of the MPI library or an MPI object
  - Example: is the code communicating, blocked, or computing?
  - Types: Enumeration datatype
  - Starting value: current state
- MPIT_PERFVAR_CLASS_RESOURCE_LEVEL
  - Describes the absolute usage of a resource within MPI
  - Example: number of messages in the unexpected message queue (UMQ)?
  - Types: Byte, Short, Int, Long, Long Long, Float, Double
  - Starting value: current resource level
- MPIT_PERFVAR_CLASS_RESOURCE_PERCENTAGE
  - Describes the percentage use of a resource within MPI
  - Example: how full is the unexpected message queue?
  - Types: Float, Double (between 0.0 = 0% and 1.0 = 100%)
  - Starting value: current resource level
- MPIT_PERFVAR_CLASS_RESOURCE_HIGHWATERMARK
  - Describes the highest usage of a resource since start of monitoring
  - Example: the largest number of messages ever queued in the UMQ
  - Types: Byte, Short, Int, Long, Long Long, Float, Double
  - Starting value: current resource level

Complete List of Classes 2/2

- MPIT_PERFVAR_CLASS_RESOURCE_LOWWATERMARK
  - Describes the lowest usage of a resource since start of monitoring
  - Example: the smallest number of messages queued in the UMQ since last reset
  - Types: Byte, Short, Int, Long, Long Long, Float, Double
  - Starting value: current resource level
- MPIT_PERFVAR_CLASS_EVENT_COUNTER
  - Counts the number of times a particular event occurs
  - Example: number of unexpected messages received
  - Types: Byte, Short, Int, Long, Long Long
  - Starting value: 0
- MPIT_PERFVAR_CLASS_EVENT_AGGREGATE
  - Aggregates parameters to a given event
  - Example: number of bytes in all unexpected messages received
  - Types: Byte, Short, Int, Long, Long Long, Float, Double
  - Starting value: 0
- MPIT_PERFVAR_CLASS_EVENT_TIMER
  - Describes aggregate timings spent processing a particular event
  - Example: time spent searching the unexpected message queue
  - Types: Double (matching MPI_Wtime)
  - Starting value: 0.0

Querying Performance Variables

Page 13

- **int MPIT_Perfvar_get_num(*num)**
  - Returns the number of performance variables available (in total)
- **int MPIT_Perfvar_get_info(index, name, namelen, verbosity, varclass, datatype, count, desc, desclen, bind, attr)**
  - Provides additional information / metadata about a variable
  - Variable specified by "index"
  - Return mandatory name and optional description
  - Return class of the variable
  - Report variable's verbosity and attributes (more on that later)
  - Variable specification through datatype/count pair
  - Information to which object this variable needs to be bound

Page 13&14

Page 14&15

Performance Variable Sessions

- Multiple libraries and/or tools may use MPIT
  - Avoid collisions and isolate state
  - Separate performance calipers
- Concept of MPIT performance sessions
  - Each "user" of MPIT allocates its own session
  - All calls to manipulate a variable instance reference this session
- **MPIT_Perfvar_session_create (MPIT_Perfvar_session *session)**
  - Start a new session and returns session identifier

Page 15

- **MPIT_Perfvar_session_free (MPIT_Perfvar_session *session)**
  - Free a session and release resources

Page 15

Performance Variable Handles

- Need to acquire a handle before using a variable
  - Includes the binding process
  - Handles are on a per session basis
- **int MPIT_Perfvar_handle_allocate(session, index, void *object, MPIT_Perfvar_handle *handle)**
  - Request a handle for variable specified by index
  - Bind to object specified by "object"
    - Pointer to object handle cast to void*
    - Caller is responsible that the right object type is used
  - Return handle
- **int MPIT_Perfvar_handle_free(session, handle)**
  - Free handle and release resources
  - Set handle to MPIT_PERFVAR_HANDLE_NULL

Page 15

Page 16

Starting/Stopping Performance Variables

- Variables can be active (started) or disabled (stopped)
  - Typical semantics used in other counter libraries
  - Easier to implement calipers
  - Potential optimizations (don't require updates of stopped variables)
- All variables are stopped initially (if possible)
- **MPIT_Perfvar_start(session, handle)**
- **MPIT_Perfvar_stop(session, handle)**
  - Start/Stop variable identified by handle
  - Effect limited to the specified session
  - Handle can be MPIT_PERFVAR_ALL_HANDLES to start/stop all valid handles in the specified session

Page 16

Reading/Writing Performance Variables

- **MPIT_Perfvar_read(session, handle, buf)**
- **MPIT_Perfvar_write(session, handle, buf)**
  - Read/write variable specified by handle
  - Effects limited to specified session
  - Buffer buf treated similar to MPI message buffers
  - Datatype and count provided by corresponding _get_info call
- **MPIT_Perfvar_reset(session, handle)**
  - Set value of variable to its starting value
  - MPIT_PERFVAR_ALL_HANDLES allowed as argument
- **MPIT_Perfvar_readreset(session, handle, buf)**
  - Combination of read & reset on same (single) variable

Page 17

Page 18

Page 18

Signal Safety

- Typical use case: sampling tools
  - Need to call MPIT routines from timer interrupt

Advice to implementors. Although MPI places no requirements on the interaction with external mechanisms such as signal handlers, it is strongly recommended that all routines to start, stop, read, write, and reset performance variables should be safe to call in asynchronous contexts. Examples of asynchronous contexts include signal handlers and interrupt handlers. Such safety permits the development of sampling-based tools. High quality implementations should strive to make the results of any such interactions intuitive to users, and attempt to document restrictions where deemed necessary.
(End of advice to implementors.)

Example: Performance Variables

- Tool to detect receive operations that are called on long unexpected message queues
  - Implementable as PMPI tool
  - Setup, variable detection and handle allocation in MPI_Init wrapper
  - Query unexpected message queue at MPI_Recv

- Global section

```
#include <mpi.h> /* Adds MPIT definitions as well */

#define THRESHOLD 5

/* Global variables for tool */
static MPIT_Perfvar_session session;
static MPIT_Perfvar_handle handle;
```

Example: Find MPIT Variable

```
int MPI_Init(int *argc, char ***argv) {
    int err, num, i, index, namelen, verb, varclass, bind;
    MPIT_Perfvar_attr attr;
    char name[100];
    MPI_Comm comm;

    err=PMPI_Init(argc,argv);
    if (err!=MPI_SUCCESS) return err;
    err=PMPIT_Init();
    if (err!=MPIT_SUCCESS) return err;

    err=PMPIT_Perfvar_get_num(&num);
    if (err!=MPIT_SUCCESS) return err;
    index=-1;
    while ((i<num) && (index<0)) {
        namelen=100;
        err=PMPIT_Perfvar_get_info(i, name, namelen, &verb, &varclass,
            &count, NULL, NULL, &bind, &attr);
        if (strcmp(name,"MPIT_UMQ_LENGTH")==0) index=i;
        i++; }
```

Example: Prepare MPIT Variable

```
    /* this could be handled in a more flexible way for a generic tool */
    ASSERT(index>=0);
    ASSERT(varclass==MPIT_VARCLASS_RESOURCE_LEVEL);
    ASSERT(datatype==MPIT_DATATYPE_INT);
    ASSERT(bind==MPIT_BIND_MPI_COMMUNICATOR);

    /* Create a session */
    err=PMPIT_Perfvar_session_create(&session);
    if (err!=MPIT_SUCCESS) return err;

    /* Get a handle and bind to MPI_COMM_WORLD */
    comm=MPI_COMM_WORLD;
    err=PMPIT_Perfvar_handle_allocate(session, index, &comm, &handle);
    if (err!=MPIT_SUCCESS) return err;

    /* Start variable */
    err=PMPIT_Perfvar_start(session, handle);
    if (err!=MPIT_SUCCESS) return err;

    return MPI_SUCCESS;
}
```

Example: Query Variable at MPI_Recv

```
int MPI_Recv(void *buf, int count, MPI_Datatype dt, int source, int tag,
        MPI_Comm comm, MPI_Status *status)
{
        int value, err;

        if (comm==MPI_COMM_WORLD) {
                err=PMPIT_Perfvar_read(session, handle, &value);
                if ((err==MPIT_SUCCESS) && (value>THREASHOLD))
                {
                        /* gather and print call stack */
                }
        }

        return PMPI_Recv(buf, count, dt, source, tga, comm, status);
}
```

Summary / Basic Functionality 1/2

- Query interface to ask for provided variables
  - Variables numbered from 0 to N-1
  - Routine to ask for N
  - Routine to ask for metadata for each variable
- Handle allocation and free
  - Enable access to a particular variable
  - Binds an MPIT variable to an MPI object
- Binding of variables
  - Enables the restriction of a variable to a particular object
  - Instantiates the concrete variable in the context of the object
  - One variable can be bound to multiple objects

Summary / Basic Functionality 2/2

Performance Variables

- Allocate Session
- Allocate Handle
- Reset/Write Variable
- Start Variable
- Stop Variable
- Read/Readreset Variable
- Free Handle
- Free Session

Control Variables

- Allocate Handle
- Read/Write Variable
- Remark:
  Global writes must be executed by all ranks
- Free Handle

The Details

- General comments
- Initialization / Finalization
- MPIT datatype system
- Returning strings
- Variable verbosity
- Complete query calls
- Return codes

General Comments

- Separate name space for MPIT
  - All routines/types/constants are prefixed with MPIT
  - MPIT routines can be used independently from MPI routines
- MPIT only defined as C-binding
  - Mainly intended for tools, which are written in C/C++
  - All routines can be accessed after including mpi.h
- All MPIT return codes return "int"
  - MPIT_SUCCESS if completed, error code otherwise
  - Errors in MPIT do not affect MPI routines
- Unless otherwise noted, all MPI conventions hold
  - Includes the profiling interface

Page 2

Page 21/22

Page 2 & 22

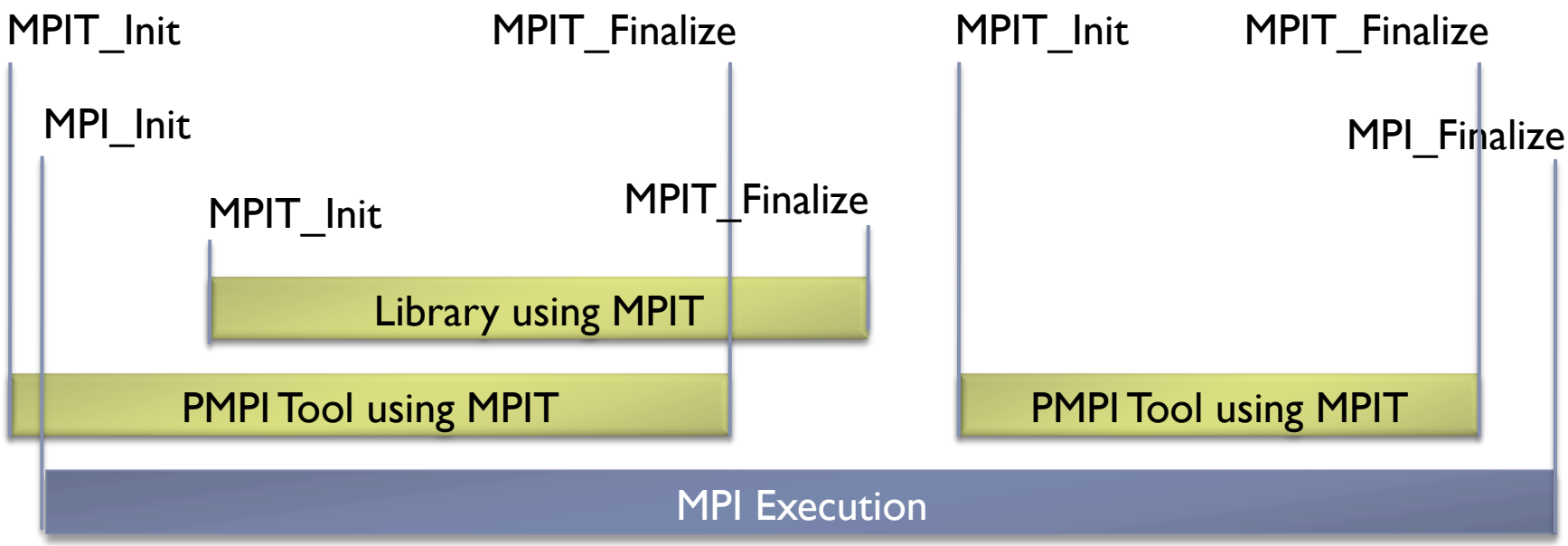
General Comments / Advice to Users

Advice to users. The number and type of control variables and performance variables can vary between MPI implementations, platforms, and even different builds of the same implementation on the same platform. Hence, any application relying on a particular variable will not be portable.

This interface is primarily intended for performance monitoring tools, support tools, and libraries controlling the application's environment. Application programmers should either avoid using the MPIT interface or avoid being dependent on the existence of a particular control or performance variable**. (End of advice to users.)**

MPIT Initialization

- MPIT initialization is separate from MPI initialization
  - MPIT handled as a separate name space
  - Requirement to analyze MPI_Init and MPI_Finalize calls
  - Enable multiple, independent initializations



MPIT Initialization / Finalization Calls

- **int MPIT_Init()**
  - Initialize the use of MPIT
  - Can be called at any time
    - Before MPI_Init, after MPIT_Init, after MPIT_Finalize
- **int MPIT_Finalize()**
  - Finalize the use of MPIT
  - Cannot be called more often than MPIT_Init up to this point
  - MPIT no longer initialized after as many calls to MPIT_Finalize as to MPIT_Init
  - Correct MPI programs must call MPIT_Init and MPIT_Finalize the same number of times
  - Exception: abort by MPI_Abort

Returning Strings

- Many MPIT calls return strings
  - Name of variables, types, items
  - Descriptions of variables
- Need a "safe" way to return strings
  - Current MPI routines returning strings require fixed buffer lengths
  - Not suitable for long descriptions
- New approach taken for MPIT
  - Caller passes a user allocated buffer with length argument N
  - MPIT writes up to N-1 characters into buffer + zero character
  - If N is smaller than length of string, the string is truncated
  - MPIT returns length of string -1 as the length argument (also if buffer is NULL or length is passed as 0)

Returning Strings (cont.)

Several MPIT function return one or more strings. These functions have two arguments for each string to be returned: one that identifies a pointer to the buffer in which the string will be returned, and one to pass the length of the buffer. The latter is used as an IN/OUT argument. The user is responsible for the memory allocation of the buffer and must pass the size of the buffer as the length argument. Let n be the length value specified to the function. On return, the function writes at most n-1 of the string's characters into the buffer, followed by a null terminator. If the returned string's length is greater than or equal to n, the string will be truncated to n-1 characters. In this case, the length of the string plus one (for the terminating null character) is returned in the length argument. The buffer is always null-terminated. If the user passes the null pointer as the buffer argument or passes 0 as the length argument, the function does not return the string and only returns the length of the string plus one in the length argument. If the user passes the null pointer for both arguments, no string is returned.

MPIT Datatype System

- MPIT has separate initialization/finalization
  - Cannot rely on MPI's datatype systems (may not be available, yet)
  - Need a separate MPIT datatype system, but does not have to be as comprehensive as MPI's datatypes
- Two datatype classes
  - Predefined types (similar to MPI basic types)
  - Enumeration types (similar to C enum's)
- Each MPIT datatype has an equivalent MPI datatype
  - Valid once both MPI and MPIT datatype systems are initialized
  - Ability to communicate MPIT information via MPI
  - Prerequisite: MPI and MPIT datatypes have the same size

MPIT Datatype Functions

- **MPIT_Datatype_get_class(datatype,datatypeclass)**
  - Return the class of the MPIT datatype
  - On return, datatypeclass will be set
    - MPIT_DATATYPECLASS_PREDEFINED
    - MPIT_DATATYPECLASS_ENUMERATION

Page 5

- **MPIT_Datatype_get_size(datatype,size)**
  - Return the size of datatype in bytes
  - Value return in the size argument

Page 6

MPIT Predefined Datatypes

MPIT Datatypes	Equivalent MPI Datatype
MPIT_BOOLEAN	MPI_INT
MPIT_BYTE	MPI_BYTE
MPIT_SHORT	MPI_SHORT
MPIT_INT	MPI_INT
MPIT_LONG	MPI_LONG
MPIT_LONG_LONG	MPI_LONG_LONG
MPIT_CHAR	MPI_CHAR
MPIT_FLOAT	MPI_FLOAT
MPIT_DOUBLE	MPI_DOUBLE

- Open questions
  - Equivalent type for MPIT_BOOLEAN
  - Handling of strings

MPIT Enumeration Datatypes

- Used to represent collection of items
  - Comparable to C-style enum types
  - Each item with a distinct name
- Examples
  - Finite number of states
    (e.g., MPI state as Computing, Blocked, Communicating)
  - Finite number of configuration options
    (e.g., communication protocols)
- Routines to query more information
  - Used to display information in tools
  - Query name of the type itself (and number of items)
  - Query name for each item for a particular type

MPIT Enumeration Datatype Functions

- **MPIT_Datatype_enum_get_info (datatype, num, name, name_len)**
  - Query more information about an enumeration type
  - num = number of items in the enumeration type
  - name = string showing the name of the type (mandatory)
  - Names must be unique among all enumeration types

Page 6

- **MPIT_Datatype_enum_get_item (datatype, item, name, name_len)**
  - Query name if a particular item is an enumeration item
  - Item identified by datatype/item pair
  - Names must be unique among all items in this datatype

Page 6

Variable Verbosity

- The number of variables can be large for particular MPI implementations
  - Can be hard for users to sift through
  - Some variables can be very specialized and not useful for basic performance analysis
- Need for subsetting
  - Each variable is assigned to one of nine verbosity levels
  - Returned by the MPIT_*_GET_INFO calls
  - Three groups of levels based on target audience
    - User, Tuner, MPI Implementor
  - Three sublevels
    - Basic, Detailed, Verbose

Verbosity constants

MPIT Verbosity Constants	Level Descriptions
MPIT_VERBOSITY_USER_BASIC	Basic information of interest for end users
MPIT_VERBOSITY_USER_DETAILED	Detailed information –" –
MPIT_VERBOSITY_USER_VERBOSE	Verbose information –" –
MPIT_VERBOSITY_TUNER_BASIC	Basic information required for tuning
MPIT_VERBOSITY_TUNER_DETAILED	Detailed information –" –
MPIT_VERBOSITY_TUNER_VERBOSE	Verbose information –" –
MPIT_VERBOSITY_MPIDEV_BASIC	Basic information for MPI deveopers
MPIT_VERBOSITY_MPIDEV_DETAILED	Detailed information –" –
MPIT_VERBOSITY_MPIDEV_VERBOSE	Verbose information –" –

- Recommendations to MPI implementors
  - Use MPIT_VERBOSITY_USER_BASIC if only one level
  - Use *_BASIC if only one level for each target group

MPIT_Controlvar_get_info

```
MPIT_Controlvar_get_info(
    int index,                          /* index of variable to query */
    char *name,
    int *name_len,                      /* unique name of variable */
    int *verbosity,                     /* verbosity level of variable */
    MPIT_Datatype *dt,                  /*MPIT datatype representing variable*/
    int *count,                         /* number of datatype elements */
    char *desc,
    int *desc_len,                      /* optional description */
    int *bind,                          /* MPI object to be bound */
    MPIT_Controlvar_attributes *attr /* additional attributes */
    )
```

Control Variable Attributes

- Additional attributes are returned as an opaque object
  - Additional query routines
  - Rationale: extensible
- **MPIT_Controlvar_attr_get_scope (MPIT_Controlvar_attributes attr, int *scope)**
  - Retrieve scope from attributes returned by _get_info
  - Scope describes if and when a variable can be changed
    - MPIT_SCOPE_READONLY – cannot be changed
    - MPIT_SCOPE_LOCAL – can be changed on a per node level
    - MPIT_SCOPE_GLOBAL – can be changed, but is a global operation
  - MPIT user is responsible to make global changes consistently
  - It is not guaranteed that a variable can be changed at any time

MPIT_Perfvar_get_info

```
MPIT_Perfvar_get_info(
    int index, /* index of variable to query */
    char *name,
    int *name_len,                  /* unique name of variable */
    int *verbosity,                 /* verbosity level of variable */
    int *varclass,                  /* class of the performance variable */
    MPIT_Datatype *dt,              /*MPIT datatype representing variable*/
    int *count,                     /* number of datatype elements */
    char *desc,
    int *desc_len,                  /* optional description */
    int *bind,                      /* MPI object to be bound */
    MPIT_Perfvar_attributes *attr /* additional attributes */
    )
```

Performance Variable Attributes

- Access routines to query attributes for performance variables as returned by the _get_info call

- **MPIT_Perfvar_attr_get_readonly (MPIT_Perfvar_attributes attr, int *readonly)**

  Page 14

  - On return "readonly" will indicate whether
    - the variable can be written and reset (if value is 0)
    - the variable can only be read (if value is 1)

- **MPIT_Perfvar_attr_get_continuous (MPIT_Perfvar_attributes attr, int *continuous)**

  Page 14

  - On return "continuous" will indicate whether
    - the variable can be started and stopped (if value is 0)
    - the variable is always started (if value is 1)

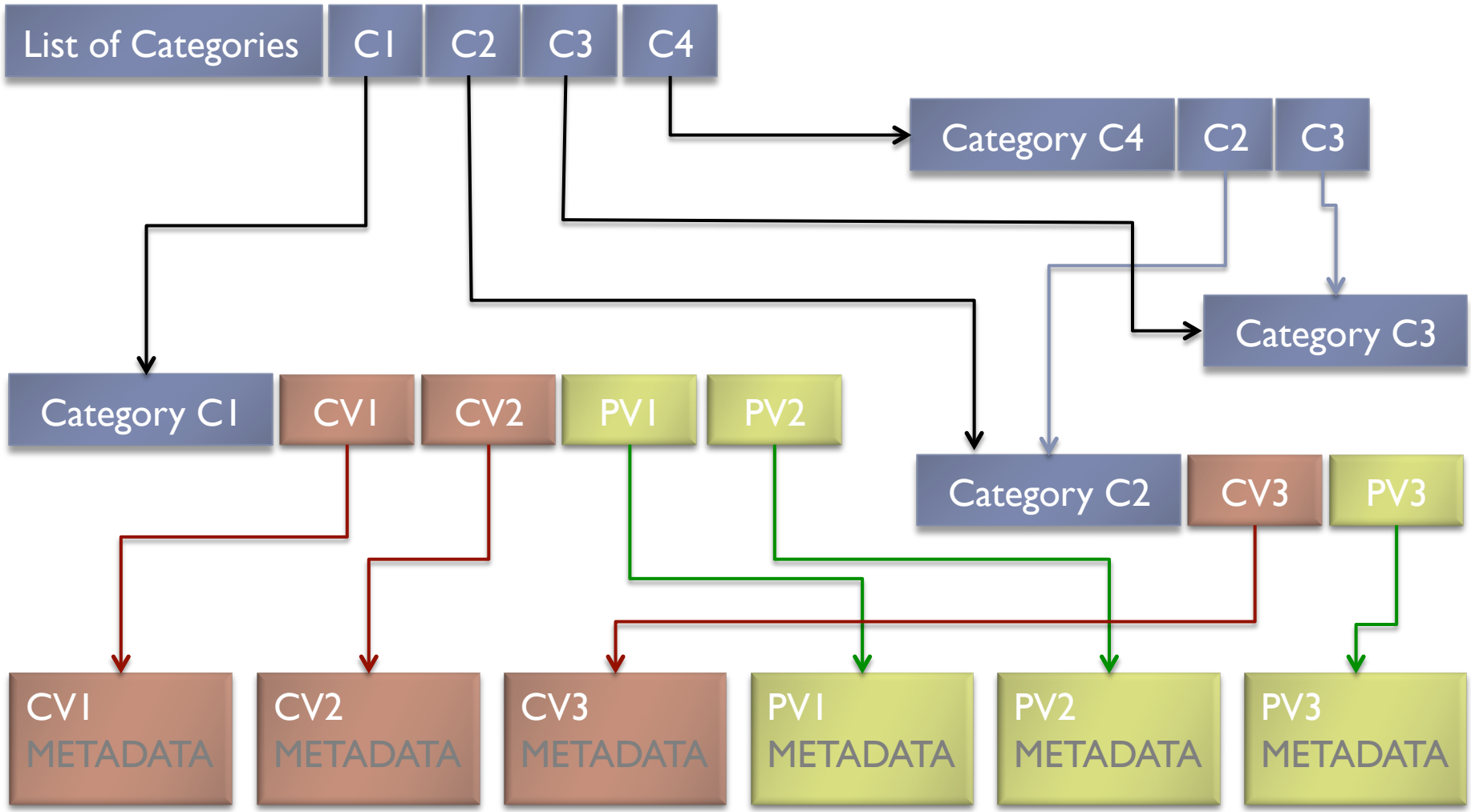
Dynamically Adding Variables

- MPIT implementations can add variables at runtime
  - Initialization at MPI_Init
  - Dynamic loading of modules
- MPIT implementations are not allowed to remove or change existing variables
  - Would cause consistency problems
  - If variables need to be removed, access to them can return errors
- MPIT implementations can only add variables
  - Extend range of variable index
  - Can be checked by calling MPIT_*_get_num and comparing the returned value with previous results

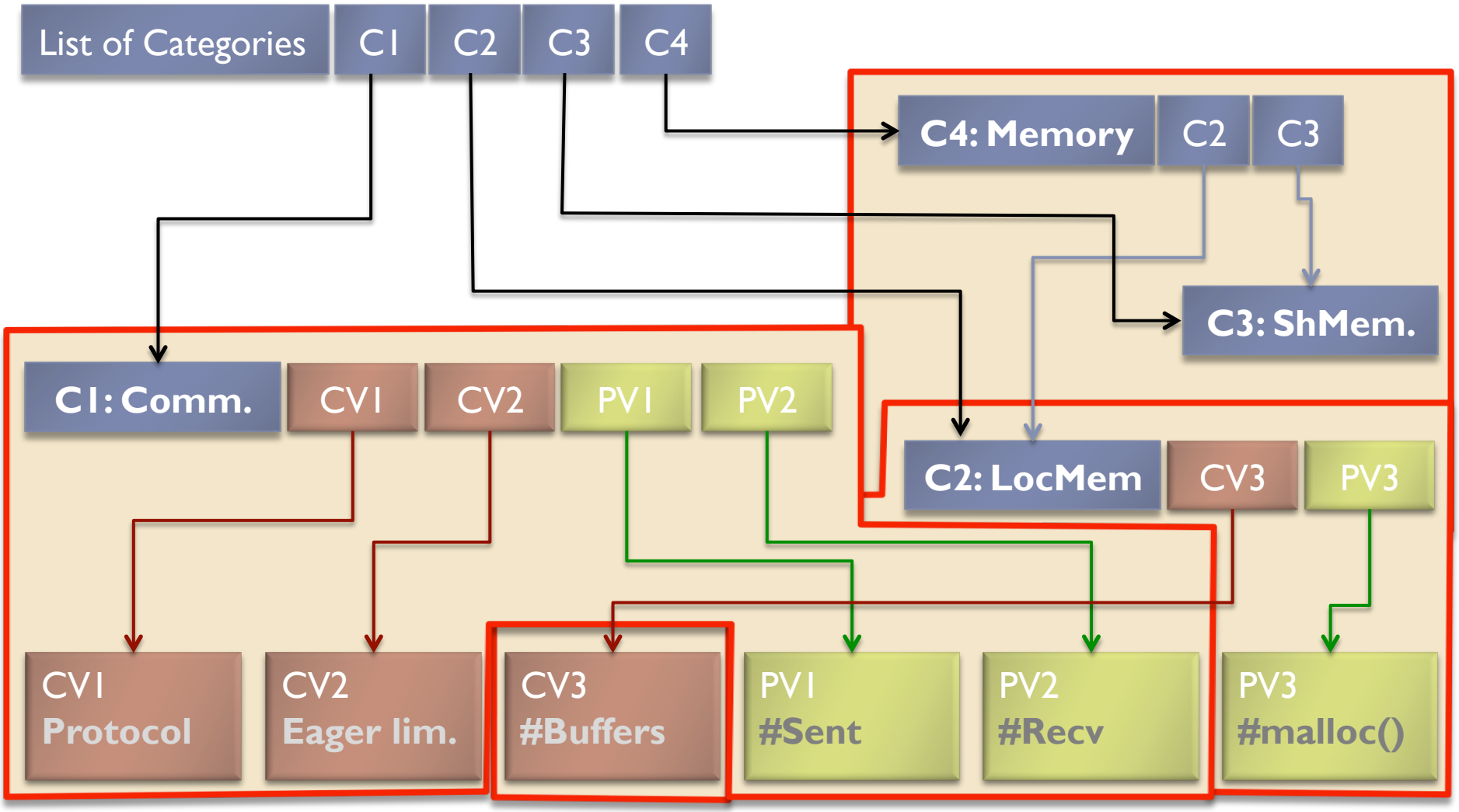
MPIT Categories

- Goal: provide some semantic information to tools
  - Tools know/can know little about individual variables
  - Categories provide an hierarchical grouping of variables
  - Categories can contain variables and other categories
  - Example: group all communication related variables
- Categories organized similarly to variables
  - Indexed from 0 to N-1 with potentially growing N
  - MPIT implementations can add categories, but not remove them
  - Routines to query metadata and contents of categories
- Limitations/Restrictions/Recommendations
  - Categories can not by cyclic and contain themselves
  - Categories should either contain variables OR other categories

Conceptual Idea of Categories



Examples: Categories



Getting Information on Categories

MPIT_Category_get_num(int *num)

Returns the number of categories (in total)

MPIT_Category_get_info(

```
int index,                  /* category to be queried */
char *name,
int *name_len,              /* mandatory name of category */
char *desc,
int * desc_len,             /* optional description */
int *num_controlvars,       /* number of controlvars in category */
int *num_perfvars,          /* number of perfvars in category */
int *num_categories         /* number of categories in category */
)
```

Reading MPIT Categories

- Routines to query contents of a category
  - **MPIT_Category_get_controlvars (int cat_index, int len, int indices[])** Page 20
  - **MPIT_Category_get_perfvars (int cat_index, int len, int indices[])** Page 21
  - **MPIT_Category_get_categories (int cat_index, int len, int indices[])** Page 21
- All routines return an array of indices that lists all variables or categories (respectively)
  - Caller is responsible to pass a sufficiently long array
  - The required length is returned by _get_info call
  - If array is too small, a truncated list is returned

Return codes

- Each MPIT call returns an error or return code
  - Modeled after MPI return/error codes
  - Return type "int"
- Successful completion
  - **MPIT_SUCCESS**
- Errors in MPIT routines are indicated by returning an error code (next slides) instead of MPIT_SUCCESS
  - Errors are not fatal to the MPI implementation itself
- Wrong parameters to MPIT routines are not required to be caught by the MPIT implementation, but if caught, the correct error code must be returned

List of Return Codes 1/2

Return Code	Description
Return Codes for all MPIT Functions	
MPIT_SUCCESS	No error, call completed
MPIT_ERR_MEMORY	Out of memory
MPIT_ERR_NOTINITIALIZED	MPIT not initialized
MPIT_ERR_CANTINIT	MPIT not in the state to be initialized
Return Codes for Datatype Functions: MPIT_DATATYPE_*	
MPIT_ERR_PREDEFINED	Datatype is a predefined datatype and not an enumeration
MPIT_ERR_INVALIDDATATYPE	Datatype is not a valid datatype
MPIT_ERR_INVALIDITEM	The item index queried is out of range (for MPIT_DATATYPE_ENUMITEM only)
Return Codes for variable and category query functions: MPIT_*_GET_INFO	
MPIT_ERR_INVALIDINDEX	The variable or category index is invalid
Return Codes for Handle Functions: MPIT_*_ALLOCATE,FREE	
MPIT_ERR_INVALIDINDEX	The variable index is invalid
MPIT_ERR_INVALIDHANDLE	The handle is invalid
MPIT_ERR_OUTOFHANDLES	No more handles available
Return Codes for Session Functions: MPIT_PERFVAR_SESSION_*	
MPIT_ERR_OUTOFSESSIONS	No more sessions available
MPIT_ERR_INVALIDSESSION	Session argument is not a valid session

List of Return Codes 2/2

Return Codes for Control Variable Access Functions: MPIT_CONTROLVAR_READ,WRITE	
MPIT_ERR_SETNOTNOW	Variable cannot be set at this moment
MPIT_ERR_SETNEVER	Variable cannot be set until end of execution
MPIT_ERR_INVALIDVAR	Control variable does not exist
MPIT_ERR_INVALIDHANDLE	The handle is invalid

Return Codes for Performance Variable Access and Control: MPIT_PERFVAR_START,STOP,READ,WRITE,RESET,READRESET	
MPIT_ERR_INVALIDHANDLE	The handle is invalid
MPIT_ERR_INVALIDSESSION	Session argument is not a valid session
MPIT_ERR_NOSTARTSTOP	Variable can not be started or stopped for MPIT_PERFVAR_START and MPIT_PERFVAR_STOP
MPIT_ERR_NOWRITE	Variable can not be written or reset for MPIT_PERFVAR_WRITE and MPIT_PERFVAR_RESET

Return Codes for Category Functions: MPIT_CATEGORY_*	
MPIT_ERR_INVALIDCATEGORY	The specified category index does not exist

MPI Tools Chapter

- Question: where to place MPIT in the standard?
  - Fits well together with PMPI
  - Other tool related extensions may come in the future
- Proposal: create a new tools chapter
  - Replaces the existing profiling interface chapter
  - Profiling interface will be a section of this chapter
  - Commonly shared preamble that allows more tool interfaces to be added at a later time
- Question: how should this be handled?
  - One large ticket?
  - Need separate ticket for changes in PMPI text?

Tools Chapter Preamble Text

This chapter discusses a set of interfaces that allows debuggers, performance analyzers, and other tools to extract information about the operation of MPI processes. Specifically, this chapter defines both the PMPI profiling interface (Section 1.2) for transparently intercepting and inspecting any MPI call, and the MPIT tool information interface (Section 1.3) for querying MPI control and performance variables. The interfaces described in this chapter are all defined in the context of an MPI process, i.e., are callable from the same code that invokes other MPI functions.

Addendum to Draft

- Add MPI info object to list of supported objects
- Distinguish different meanings of global
  - Objects to bind vs. control variable scope
- Allow passing of NULL as string length
- Changed boolean type in MPIT type system
  - Planning to add explanation about type size and equivalence
- Variables that return strings
  - How to handle multiple length strings
- Need a clear connection between timer variable class and the values returned by routines like MPI_Wtime
- Examples still to be added to the text
- Explicit values for verbosity states constants
- Interoperability/Non-overlap of MPI and MPIT error codes

Summary & Status

- MPIT enables access to MPI internal information
  - Designed for control and performance variables
- Impact on MPI standard/implementations
  - MPI implementation agnostic
  - Separate name space and initialization
  - Isolation to MPI routines maintained as much as possible
- Status
  - Draft complete (aside from issues on previous slide)
  - Hopefully ready for first reading once comments are addressed
- Prototype implementation
  - Early experiments with previous version of MPIT within MVAPICH
  - Prototype within MPICH-2 underway (Dave)