Open MPI: Overview / Architecture

Jeff Squyres

CISCO

Thank you, Greenplum!

GREENPLUM®
A DIVISION OF EMC

Purpose

- An *overview* of Open MPI development
  - There's too much detail for 2 hours
- This is *not* a comprehensive guide!
  - You still need to go explore
  - You still need to go read code
  - You still need to go try things

Overview

- Overview of MPI
- Version Numbers
- Building / Installing Open MPI
- Open MPI Code Architecture
- Run-Time Parameters
- Common Code Highlights
- Hardware Locality ("hwloc")

MPI Goals

MPI goals

- High-level network API
  - Abstracts away the underlying transport
  - Simple things are simple
- API designed to be “friendly” to high performance networks
  - Ultra low latency (*nanoseconds* matter)
  - Rapid ascent to wire-rate bandwidth

MPI goals

- Typically used in High Performance Computing (HPC) environments
  - Has a bias for large compute jobs
- But:
  - "HPC" definition is evolving
  - MPI starting to be used outside of HPC
  - …because MPI is a good network IPC API

Open MPI Version Numbers

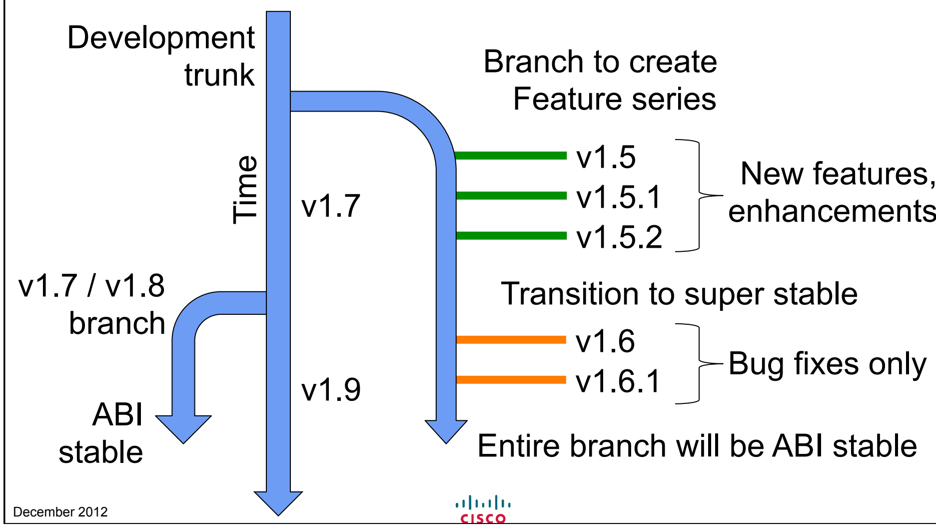
Versioning scheme

- Scheme: <major>.<minor>.<release>
- Open MPI has 2 concurrent release series
  - <minor> = odd: "Feature series"
  - <minor> = even: "Super stable series"
- Both are tested and QA'ed
  - Main difference between the two is time
  - "Stable" series are mature, time-tested

Branch goals

- Trunk: active development
  - "Mostly stable"
- <minor> = odd: feature series (branches)
  - New features added / removed
  - Controlled commits
- <minor> = even: stable series (branches)
  - Bug fixes only – no new features
  - Controlled commits

Feature / stable series



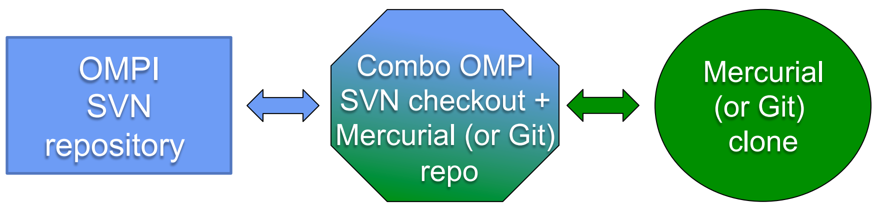
Version control

- Main Open MPI repository is Subversion
  - Hosted by Indiana University (thank you IU!)
  - https://svn.open-mpi.org/svn/ompi

INDIANA UNIVERSITY

…but you can use others

- Many Open MPI devs use Mercurial or Git
  - …and still stay in sync with SVN
- Excellent for internal development

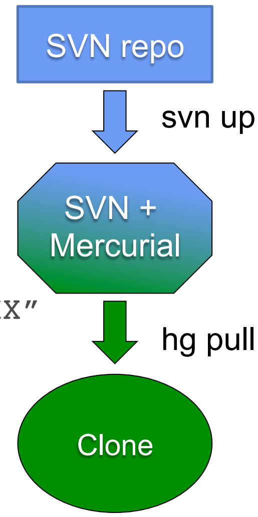


Using Mercurial (or Git)

```
$ svn co https://svn.open-mpi.org/svn/ompi/trunk
    ompi-svn-combo
$ cd ompi-svn-combo
$ hg init
$ cp contrib/hg/.hgignore .
$ hg add
$ ./contrib/hg/build-hgignore.pl
$ hg commit –m “Initial SVN rXXXX version”
$ cd ..
$ hg clone ompi-svn-combo my-work-clone
```

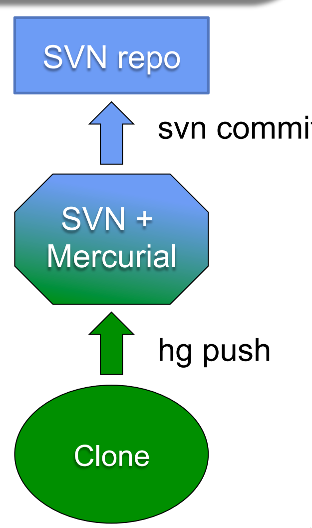
Pull down new SVN commits

```
$ cd ompi-svn-combo
$ hg up
$ svn up
```
→ Merge and resolve any conflicts
```
$ ./contrib/hg/build-hgignore.pl
$ hg addremove
$ hg commit -m "Up to SVN rXXXX"
$ cd ../my-work-clone
$ hg pull
```



Push up Mercurial commits

```
$ cd my-work-clone
…do work…
$ hg commit
$ hg push
$ cd ../ompi-svn-combo
$ hg up
```
→ Merge and resolve any conflicts
```
$ svn commit
```



Using Mercurial (or Git)

- Only use the combo for pushing / pulling!
  - Do development work in clones

- See more details on the Open MPI wiki:
  https://svn.open-mpi.org/trac/ompi/wiki

Building / Installing Open MPI

Distribution tarballs

- Built / installed very much like many other open source packages

```
$ ./configure —prefix=$HOME/ompi …
$ make —j 8 install
```


Filesystem time

- Build machine must be time-synchronized with the file server
  - If building on a local filesystem, non-issue
  - If building on a network filesystem, check this
- **WARNING:**
  - If not synced, strange build errors will occur

Suggestions where to install

- Install somewhere under $HOME
  - No root permissions necessary
- Install on a networked filesystem
  - Available on all servers
- Install to a directory by itself
  - Easy to get a clean, fresh installation

```
$ rm –rf $HOME/ompi; make install
```


Build features

- Parallel builds fully supported

```
$ make –j 8 all
```

- VPATH builds fully supported

```
$ mkdir build
$ cd build
$ ../configure … && make –j 8 …
```

- Common make targets supported
  - all, install, uninstall, clean, distclean, dist, check, …etc.

Building

- Generally only need compilers and “make”
- Defaults to gcc, but can use others

```
./configure CC=icc CXX=icpc FC=ifort …
```

- Many different configure options available

```
./configure --help
```

- Recommend building on a fast (local) disk

Sidenote: save your output!

- *Highly* recommend saving all output
  - You never know if you’ll need to examine something later

```
$ ./configure … 2>&1 | tee config.out
$ make –j 8      2>&1 | tee make.out
$ make install   2>&1 | tee install.out
```

Common configure options

- `--disable-dlopen`
  - Slurp plugins into main libs
- `--enable-mpirun-prefix-by-default`
  - Helps when using ssh
- Disable building optional parts of OMPI
  - `--disable-mpi-cxx`
  - `--disable-mpi-fortran`
  - `--disable-vt`
- `--enable-mpi-java`: Java MPI bindings

Common configure options

- Tell configure non-default locations:
  - `--with-<PACKAGE>=DIR` *(general form)*
  - `--with-jdk-dir=DIR`
  - `--with-verbs=DIR`
  - `--with-valgrind=DIR`
- General philosophy:
  - If configure finds X, build OMPI support for it
  - If configure does not find X, skip it
  - If you ask for X and OMPI does *not* find it, error

Platform files

- Roll up lots of configure options in a file
  - Simple text file with one option per line:

```
enable_mpi_java=yes
enable_vt=no
with_verbs=/usr/local/ofed
```

- Specify via --with-platform:

```
$ ./configure —with-platform=\
    greenplum/mrplus/linux
```


Developer builds

- Require more tools / setup
- SVN trunk currently requires (Dec. 2012):
  - Autoconf 2.69
  - Automake 1.12.2
  - Libtool 2.4.2
  - Flex 2.5.35 (2.5.35 strongly recommended)
- Why?
  - Old Autotools versions have bugs
  - OMPI uses new Autotools features

Don't have recent enough Autotools?

- Easy to obtain and install
  - Download from ftp.gnu.org

```
$ ./configure --prefix=$HOME/gnu
$ make install
```

- **WARNINGS:**
  - You *may* need to install recent GNU m4, too
    - Recent Autoconf versions require recent GNU m4
  - Install all the tools into a single prefix
  - Do *not* overwrite system-installed Autotools!

Developer builds

- Make sure Autotools are in your $PATH
- Run `./autogen.pl` in OMPI top directory
  - More on this script later

- Now `./configure` and `make` just like distribution tarballs

Developer builds

- *Much* debugging is enabled by default
  - Auto-activated if `./configure` sees `.svn`, `.hg`, or `.git` directory
  - Results in lower performance
  - …but (much) easier to debug
- To create an optimized build, either:
  - Build from a distribution tarball, or
  - Do a VPATH build, or
  - Configure `--with-platform=optimized`

December 2012 31

The role of `autogen.pl`

- Prepares the tree and runs the Autotools
  - Takes a minute or three to run
  - You do *not* need to run it every build
- Generally only need to run `autogen.pl`:
  - If you change `VERSION`
  - If you change `configure.ac`
  - If you change any `*.m4` file
  - If `svn up` changes any of these files

December 2012 32

The role of `configure`

- Tests system and prepares to build
  - Configures all plugins and subsystems
  - May take multiple minutes to run
  - You do *not* need to run it every build
- Generally only need to run `configure`:
  - If you re-run `autogen.pl`
  - If you add / remove a framework or plugin

The role of `make`

- Generates a *small* number of source files
  - Flex parsers
  - Fortran modules
- Auto-generate C header dependencies
  - If you edit a C .h file, a top-level `make` will rebuild everything that includes that .h file
- Build and install Open MPI

Where to run `make`

- Top-level directory
- Top-level project directories
  - Only sometimes – more on this later
- Individual plugin directories
  - This saves a *lot* of time
- Popular targets:
  - all, install

What gets installed

- What users need to compile/run MPI apps
- Libraries, plugins, MPI header files
  - E.g., mpi.h, mpif.h, mpi.mod, mpi_f08.mod
- Text config and help files
- Man pages
- Open MPI utility executables
  - E.g., mpicc, mpirun, etc.

What does not get installed

- NO: Autoconf-generated config.h files
- NO: component header files
- NO: project core header files
- NO: libtool convenience libraries

→ If it isn't needed to compile / run MPI apps, it does not get installed

Open MPI Code Architecture

Included 3rd party packages

- Hardware Locality (hwloc)
  - Server topology / locality information
- libevent
  - File descriptor, timer, signal event engine
- libltdl (part of GNU Libtool)
  - Portable “dlopen”, “dlsym”, etc.
- VampirTrace
  - Optional MPI trace library

→ All are configured / built as part of OMPI

Code breakdown

- Vast majority of code base is C
  - A few Flex (.l) files that generate C
- Lots of m4 / sh / Autoconf / Automake
  - Configure / build system only
- A few others
  - MPI Fortran, C++, Java bindings
    - Top-level APIs only; mostly call C underneath
  - Soon: Perl/Python to generate Fortran code

Code breakdown from ohloh.net

Language	LOC	Percent
C:	572,312	74.0%
C++:	58,566	7.6%
Autoconf:	48,923	6.3%
Shell script:	30,520	3.9%
Fortran:	23,121	3.0%
Automake:	12,829	1.7%

Code style guidelines

- 4 space tabs
  - Spaces, not tabs
- Curly braces on first line of the block
  - if (a < b) { …
- Preprocessor macros in all upper case
- Not many other style rules enforced
  - Too much religious debate; not worth it

Defensive programming

- All blocks use curly braces
  - Even one-line blocks
- Constants on the left side of ==
  - if (NULL == foo) { …
- Functions with no arguments are (void)
- No C++-style comments in C code
  - No GCC extensions except in GCC-only code
- No C++ code in libraries
  - Discouraged in components

Defensive programming

- Always define preprocessor macros
  - Define logicals to 0 or 1 (vs. define or not define)
  - Use "#if FOO", not "#ifdef FOO"
  - Gives compiler assistance for mistakes
- Not possible for some generated macros
  - Autoconf and friends

Name conventions

- No CamelCase
- Use multi-word names
  - (Usually) Use full words, not abbreviations
  - Separated by underscores

    `orte_plm_base_receive_process_msg()`

    `opal_hwloc_base_get_local_cpuset()`
- Yes, they're long
  - But you know exactly *what* and *where* they are

Name conventions

- Type names follow the prefix rule
  (described later)
- Most structs are typedef'ed

  `typedef struct ompi_foo_t { …} ompi_foo_t`
- Typically use the typedef name
  - Type names generally end in `_t`
  - Function pointer typedefs end in `_fn_t`

#include statements

- System files are in <>
  - Most should be protected with macros

```
#if HAVE_UNISTD_H
#include <unistd.h>
#endif
```

- OMPI files in “”
  - Always use full pathname

```
#include “opal/mca/base.h”
#include “ompi/group/group.h”
```


Header files

- Always protect with preprocessor macros

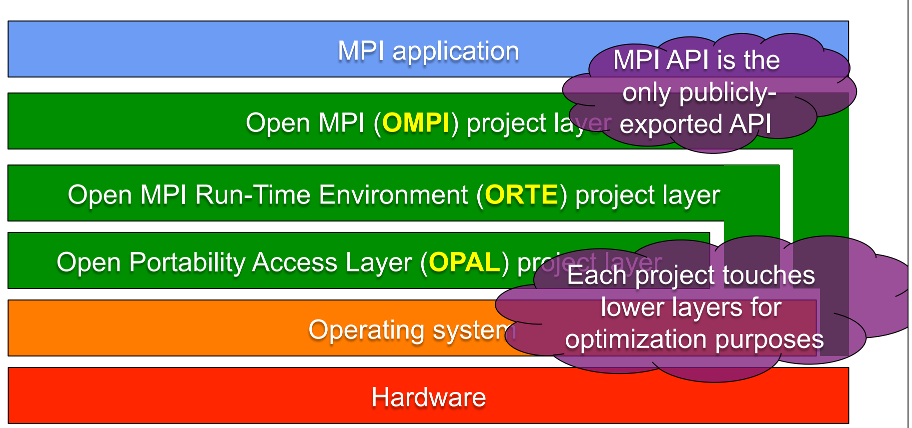
```
#ifndef _THIS_HEADER_FILE_NAME_H
#define _THIS_HEADER_FILE_NAME_H
/* …contents of header file… */
#endif
```

- Only access external symbols through their header files
  - Do not “extern” external variables in .c files
  - Do not prototype external functions in .c files

Compiler warnings

- Fix warnings on all platforms, compilers
- Default GCC developer build
  - Maximum pickyness
- Exceptions granted where warnings cannot be avoided, such as:
  - OpenFabrics header files
  - Flex-generated code

Project architecture view



Projects (layers)

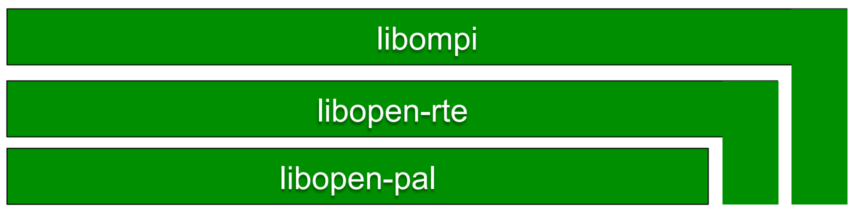
- OMPI *(pronounced: oom-pee)*
  - Public MPI API
  - Back-end MPI semantics and supporting logic
- ORTE *(pronounced: or-tay)*
  - No knowledge of MPI
  - Parallel run-time system
    - Launch, monitor individual processes
    - Group individual processes into "jobs"
  - Forward stdin / stdout / stderr

Projects

- OPAL *(pronounced: o-pull)*
  - Single-process semantics only
  - Portable OS-level functionality
  - Basic utilities (linked lists, etc.)

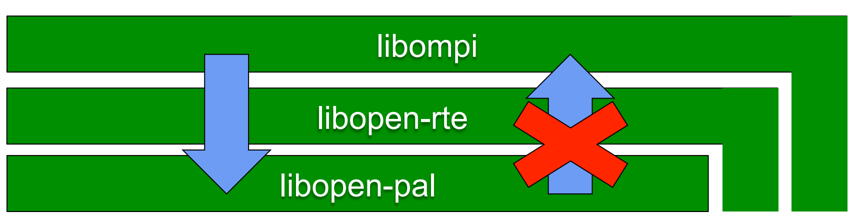
Project separation

- Each project is a separate library



Dependencies

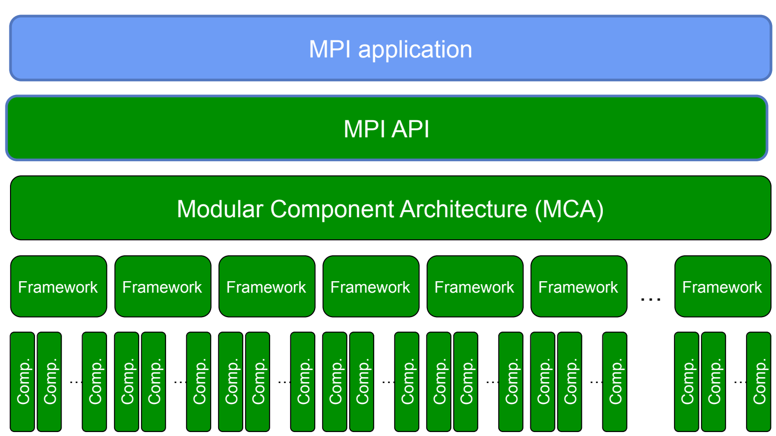
- Downward only!
  - Violations punished by the linker



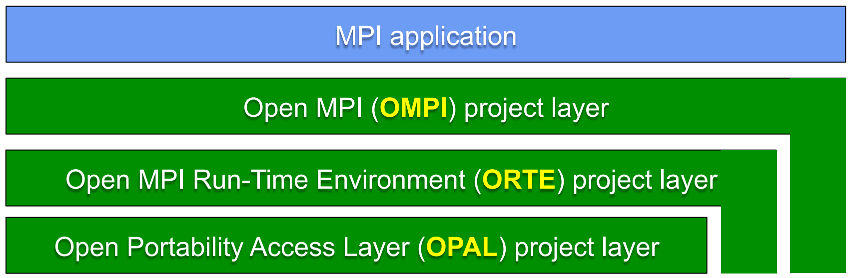
Plugin architecture

- Each project is structured similarly:
  - Main / core code
  - Components (a.k.a. “plugins”)
  - Frameworks
- Plugins are a fundamental design decision
  - Governed by the Modular Component Architecture (MCA)

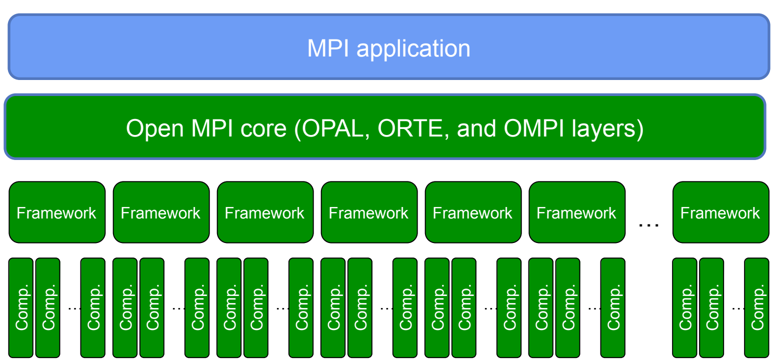
MCA architecture view



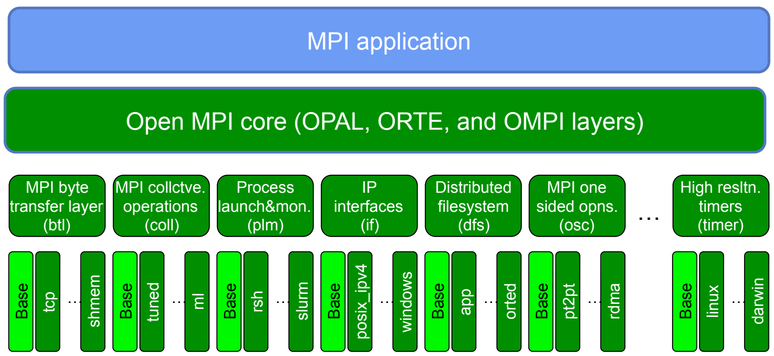
Project architectural view (for comparison)



Merged architecture views



Merged architecture views, showing some actual frameworks and components



Why components (plugins)?

- Better software engineering
  - Enforce strict abstraction barriers
- Small, discrete chunks of code
  - Good for learning / new developers
  - Easier to maintain and extend
- Separate user apps from back-end libraries
  - E.g., MPI apps not compiled against libibverbs.so / libportals.so / libpbs.a

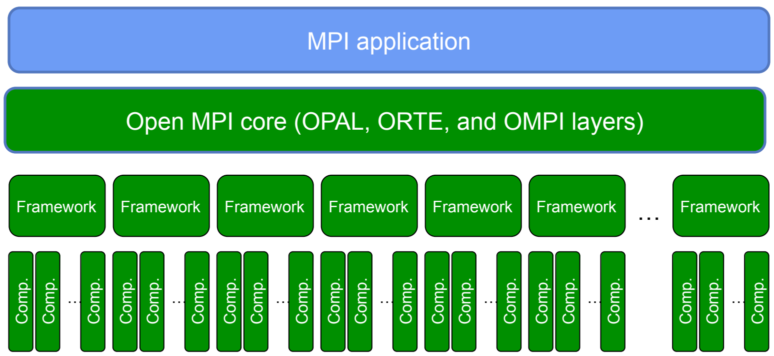
MCA layout

- MCA
  - Top-level architecture for component services
  - Find, load, unload components
- Frameworks
  - Targeted set of functionality
  - Defined interfaces
  - Essentially: a grouping of one type of plugins
  - E.g., MPI point-to-point, high-resolution timers

MCA layout

- Components
  - Code that exports a specific interface
  - Loaded / unloaded at run-time (usually)
  - Think "plugins"
- Modules
  - A component paired with resources
  - E.g., "TCP" component loaded, finds 2 IP interfaces (eth0, eth1), makes 2 TCP modules

Merged architecture views (review)



December 2012 63

MCA code organization

- Frameworks
  - Have unique string names
- Components
  - Belong to exactly one framework
  - Have unique string names
  - Namespace is per framework
- All names must be valid C variable names

December 2012 64

Organized by directory

- <project>/mca/<framework>/<component>
  - Project = opal, orte, ompi
  - Framework = framework name, or "base"
  - Component = component name, or "base"
- Directory names must match
  - Framework name
  - Component name
- Examples
  - ompi/mca/btl/tcp, ompi/mca/btl/sm

"Base"

- Reserved name: "base"
  - opal/mca/base: the MCA itself
  - orte/mca/plm/base: the PLM framework
  - ompi/mca/btl/base: the BTL framework
- Helper functions / header files
  - Common to all components in that framework
  - Public data / methods to be invoked from outside the framework

Directory layout

top →
- configure
- README
- NEWS
- VERSION
- …others…
- ompi
- orte
- opal

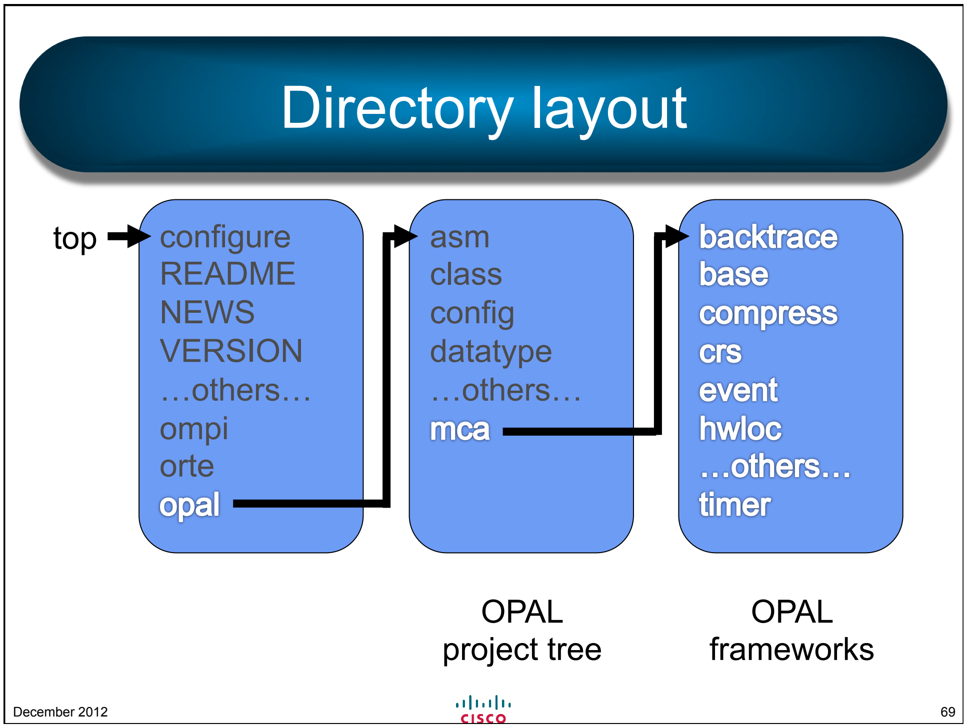
December 2012 67

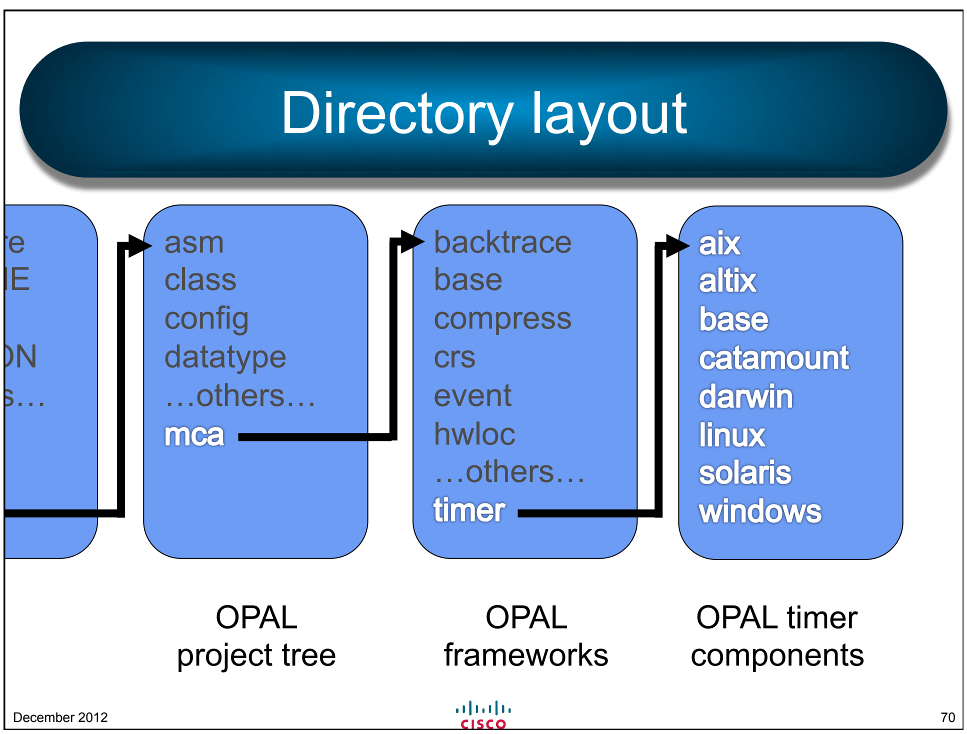
Directory layout

top →
- configure
- README
- NEWS
- VERSION
- …others…
- ompi
- orte
- opal →
  - asm
  - class
  - config
  - datatype
  - …others…
  - mca

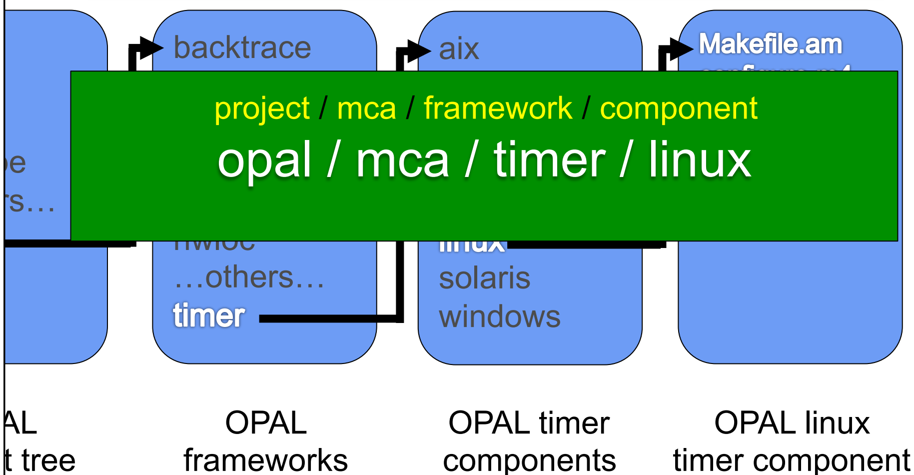
OPAL
project tree

December 2012 68

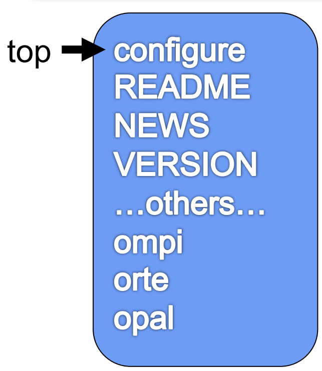

Directory layout
top
configure
README
NEWS
VERSION
…others…
ompi
orte
opal
asm
class
config
datatype
…others…
mca
backtrace
base
compress
crs
event
hwloc
…others…
timer
OPAL project tree
OPAL frameworks
December 2012
69

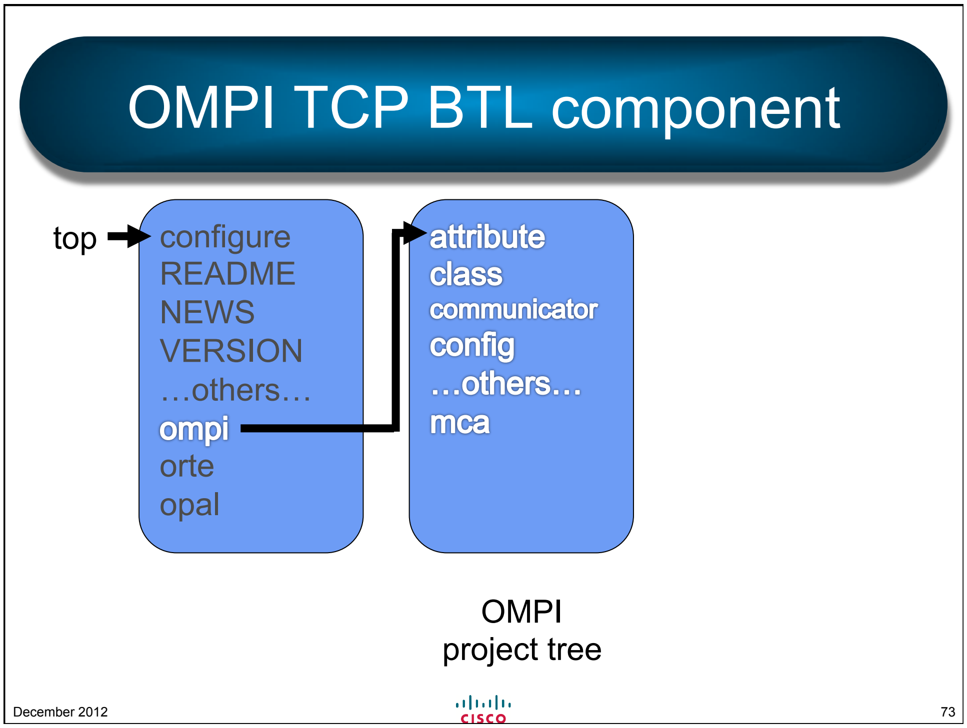

Directory layout
asm
class
config
datatype
…others…
mca
backtrace
base
compress
crs
event
hwloc
…others…
timer
aix
altix
base
catamount
darwin
linux
solaris
windows
OPAL project tree
OPAL frameworks
OPAL timer components
December 2012
70

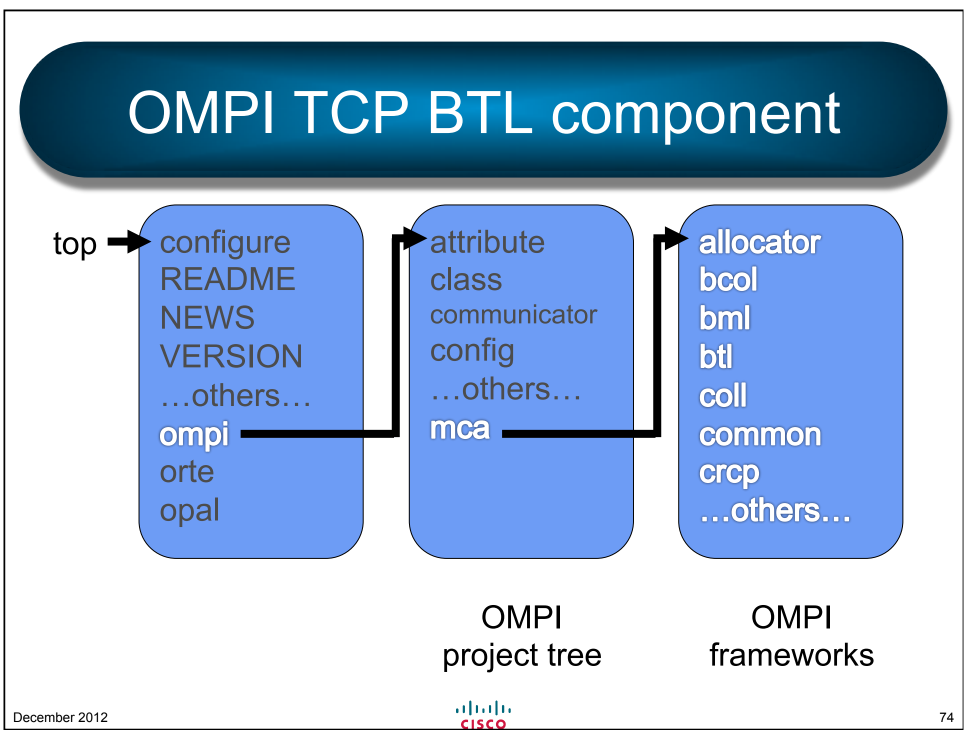
OPAL Linux timer compoment

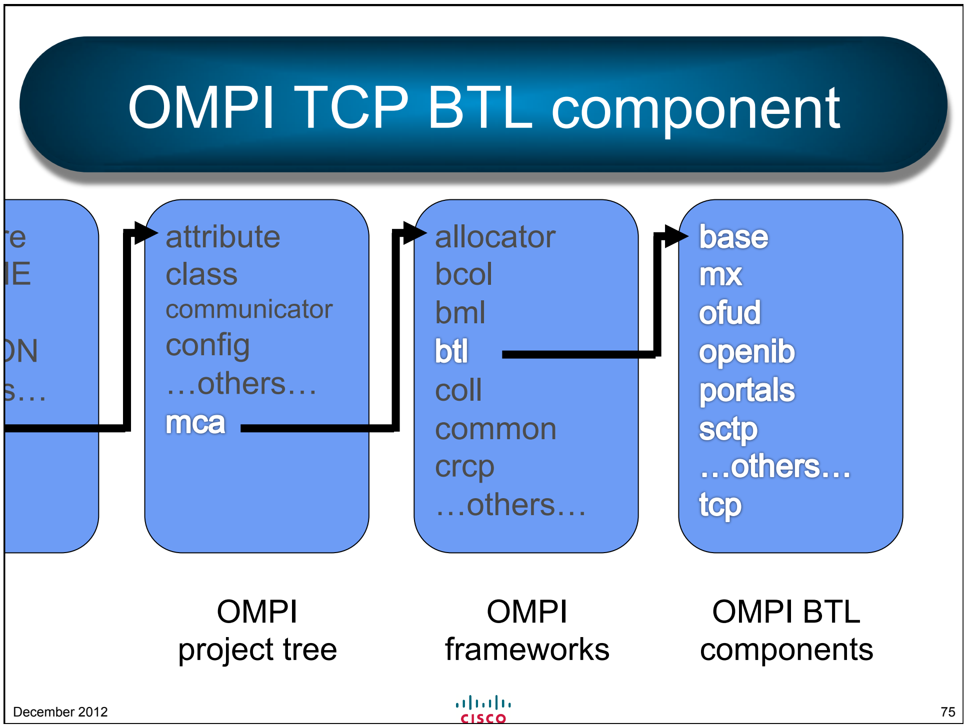


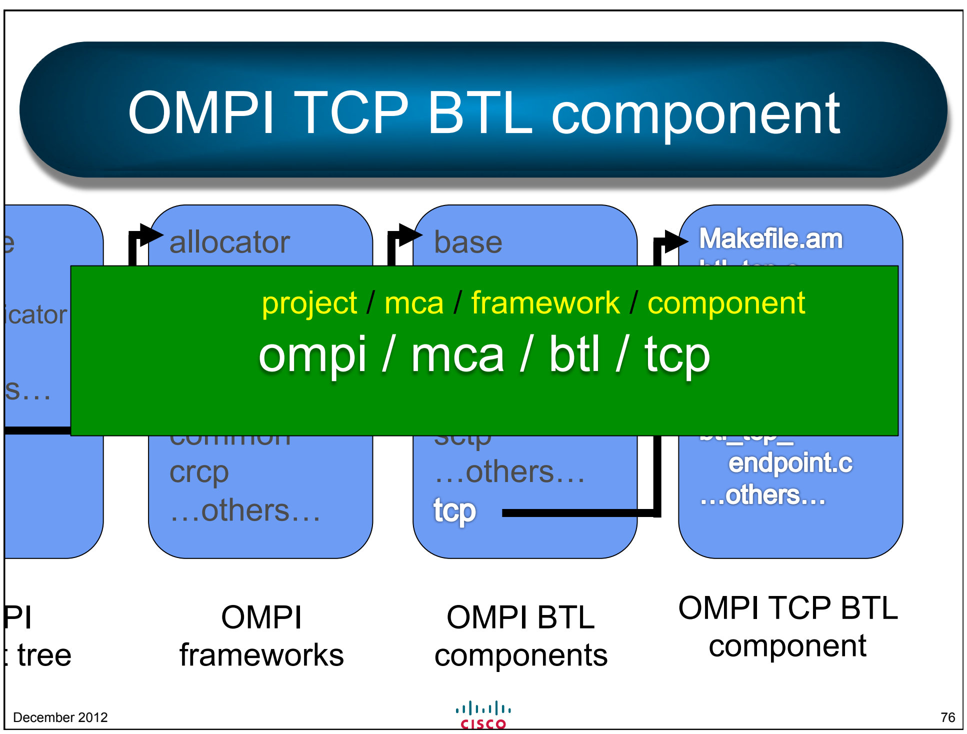
OMPI TCP BTL component

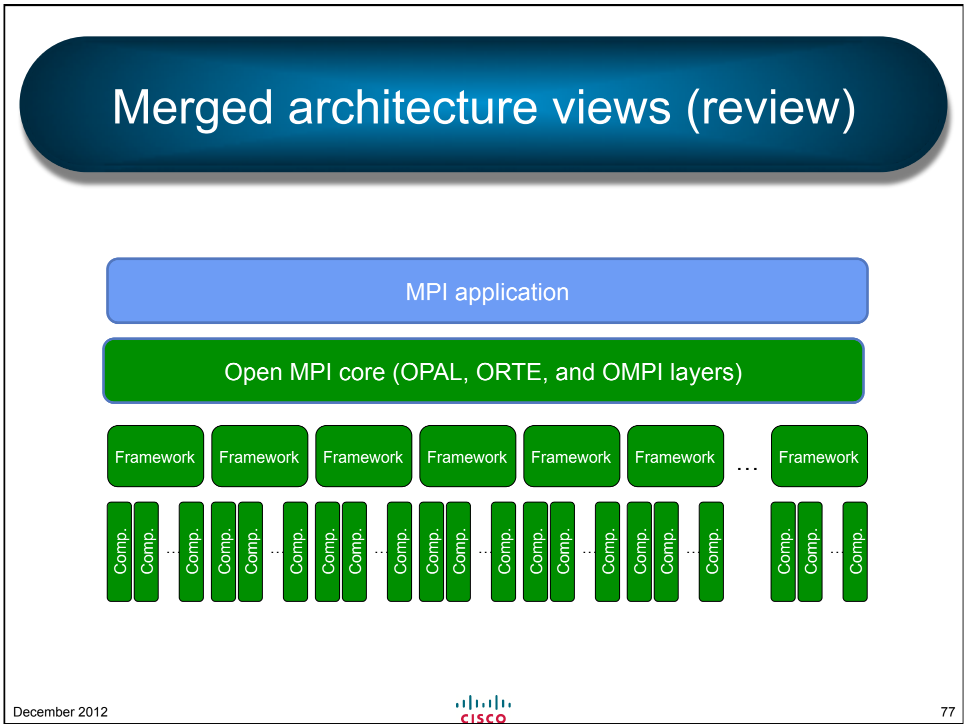


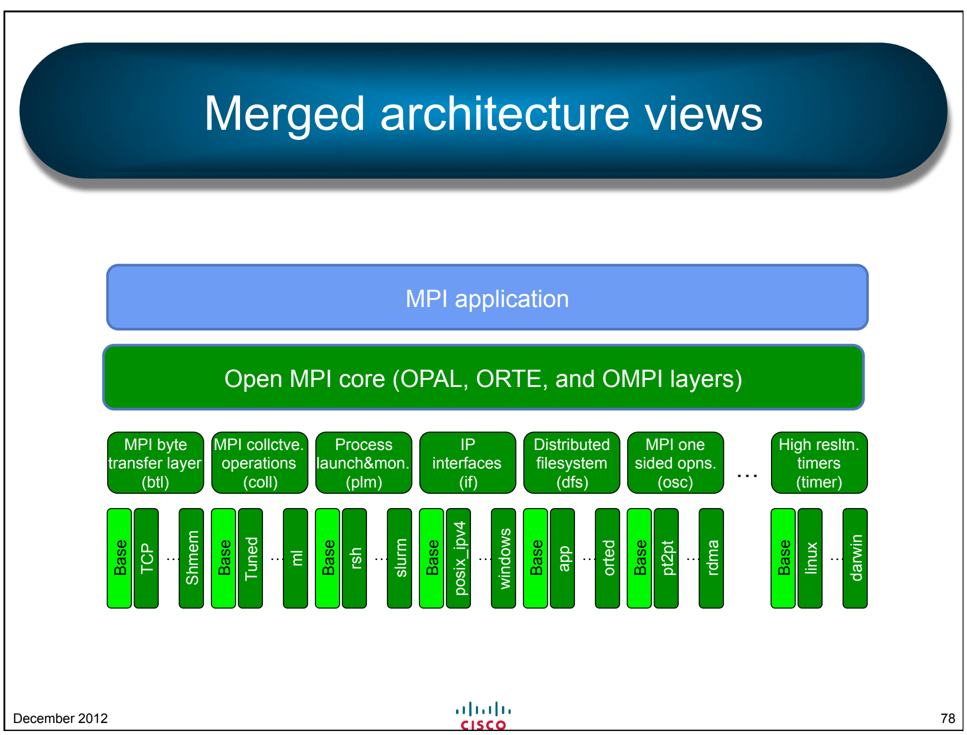

OMPI TCP BTL component
top
configure
README
NEWS
VERSION
…others…
ompi
orte
opal
attribute
class
communicator
config
…others…
mca
OMPI
project tree
December 2012
73


OMPI TCP BTL component
top
configure
README
NEWS
VERSION
…others…
ompi
orte
opal
attribute
class
communicator
config
…others…
mca
allocator
bcol
bml
btl
coll
common
crcp
…others…
OMPI
project tree
OMPI
frameworks
December 2012
74


OMPI TCP BTL component
attribute
class
communicator
config
…others…
mca
allocator
bcol
bml
btl
coll
common
crcp
…others…
base
mx
ofud
openib
portals
sctp
…others…
tcp
OMPI
project tree
OMPI
frameworks
OMPI BTL
components
December 2012
75


OMPI TCP BTL component
allocator
base
Makefile.am
project / mca / framework / component
ompi / mca / btl / tcp
crcp
…others…
…others…
tcp
endpoint.c
…others…
OMPI
frameworks
OMPI BTL
components
OMPI TCP BTL
component
December 2012
76

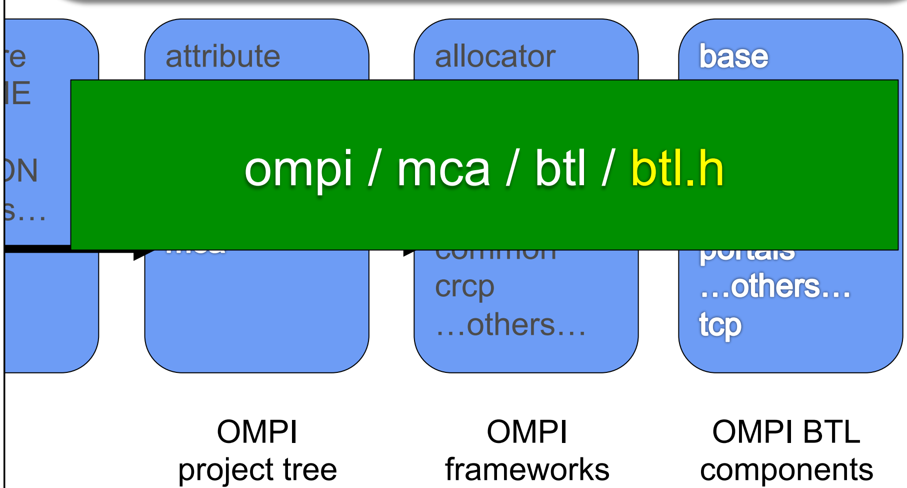

Merged architecture views (review)
MPI application
Open MPI core (OPAL, ORTE, and OMPI layers)
Framework
Framework
Framework
Framework
Framework
Framework
…
Framework
Comp.
December 2012
77


Merged architecture views
MPI application
Open MPI core (OPAL, ORTE, and OMPI layers)
MPI byte transfer layer (btl)
MPI collctve. operations (coll)
Process launch&mon. (plm)
IP interfaces (if)
Distributed filesystem (dfs)
MPI one sided opns. (osc)
…
High resltn. timers (timer)
Base
TCP
Shmem
Base
Tuned
ml
Base
rsh
slurm
Base
posix_ipv4
windows
Base
app
orted
Base
pt2pt
rdma
Base
linux
darwin
December 2012
78

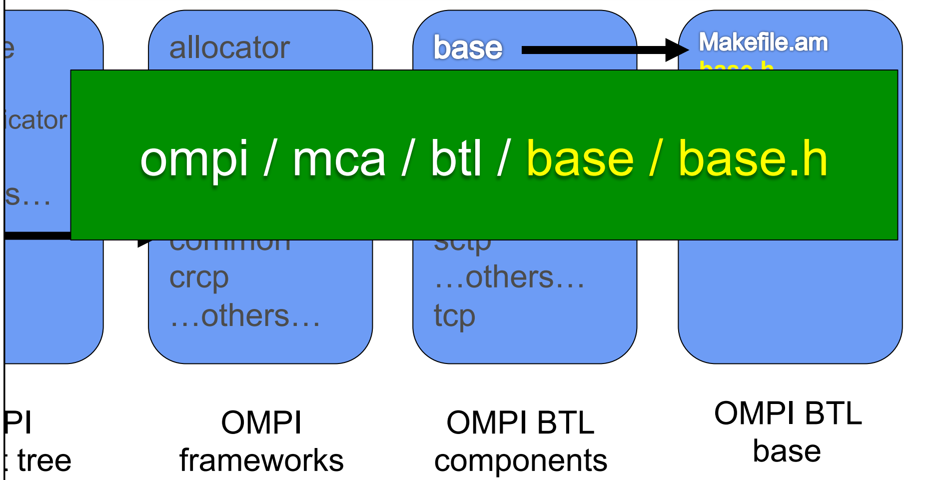
Header File Conventions

- Framework interface defined in
  - <project>/mca/<framework>/<framework>.h
  - This is mandatory
- Public base functions declared in
  - <project>/mca/<framework>/base/base.h
  - This is common, but not mandatory

BTL framework header



BTL base public header



Components

- Back-end component magic
  - Function pointers
  - Usually compiled as dynamic shared objects (DSO’s) in .so files (“plugins”)
  - But can be included in libmpi (etc.)
- Use GNU Libtool “ltdl” library
  - Portable dlopen(), dlsym()
  - Even works on Windows
  - Not GPL (!)

Component implementations

- Build system requirements:
  - `configure.m4`
  - `Makefile.am`
  - Will not discuss these in detail today
- Details of component build requirements:

  https://svn.open-mpi.org/trac/ompi/wiki/devel/CreateComponent

December 2012 83

Component implementations

- Freedom of implementation
  - As many `.c` and `.h` files as you want
  - Can even have subdirectories
- End result, needs to produce `mca_<framework>_<component>.so`
  - Examples

    `mca_btl_tcp.so`

    `mca_plm_rsh.so`

December 2012 84

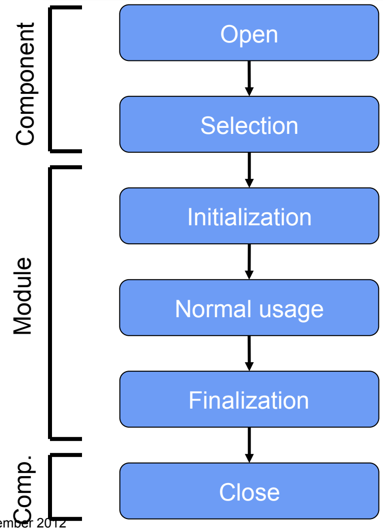
Each framework is unique

- The MCA base is strictly defined
- Each framework builds upon the base
  - But definitions are framework-specific
  - Every framework is different
  - Depends on what the framework is for
- Therefore somewhat difficult to describe
- But most follow common conventions

Component Interface

- Defined by the framework
- Typically has some kind of selection function
- Framework asks each component:
  - "Do you want to be used with X?"
  - Where "X" is relevant to the framework
- Examples
  - BTL: "Do you want to be used with this process?"
  - Coll: "Do you want to be used with MPI communicator X?"

Component / Module Lifecycle

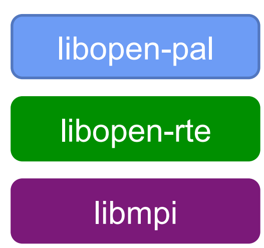


- Component
  - Open DSO (if necessary)
  - Open: per-process initialization
  - Selection: per-scope determination if want to use
  - Close: per-process finalization
  - Close DSO (if necessary)
- Module
  - Initialization: per-scope, if component is selected
  - Normal usage
  - Finalization: per-scope cleanup

Where to run `make` (redux)

- Top-level directory
  - Makes everything

```
$ make all
```



Where to run `make` (redux)

- Top-level directory
  - Makes everything
- Top-level project directories
  - Builds entire project library

```
$ cd opal
$ make all
```

libopen-pal

December 2012 89

Where to run `make` (redux)

- Top-level directory
  - Makes everything
- Top-level project directories
  - Builds entire project library

```
$ cd orte
$ make all
```

libopen-rte

December 2012 90

THIS SLIDE IS OBSOLETE!

After we recorded the video, we made changes to the Open MPI build system that made this slide be incorrect.

Specifically: libopen-rte does *not* include libopen-pal, and libmpi does not include libopen-rte.

So you can "make" in in project directory, and even "make install".

Where to run `make` (redux)

- In individual component directories
  - E.g., `make all` or `make install`
  - Saves a *lot* of time
- Example

```
$ cd ompi/mca/btl/tcp
…modify the TCP BTL…
$ make install
```

More related wiki pages

- The role of autogen.pl
  https://svn.open-mpi.org/trac/ompi/wiki/devel/Autogen
- How to add a component
  https://svn.open-mpi.org/trac/ompi/wiki/devel/CreateComponent
- How to add a framework
  https://svn.open-mpi.org/trac/ompi/wiki/devel/CreateFramework

Framework / component prefix rule

- Public names / symbols must be prefixed
  - `project_framework_component_<name>` (usually)
  - `framework_component_<name>`
  - `mca_framework_component_<name>`
    - Component struct only – special case

Framework / component prefix rule

- **WARNING** (historical note):
  - <project> prefix was only added recently
  - Many component files and symbols do not have <project> prefix
  - All *new* names should be project-prefixed
  - Will be fixed over time

Prefix rule examples

- Public function: `opal_timer_linux_init()`
- Public symbol: `orte_plm_rsh_started`
- Filename: `btl_tcp_component.c`
  - Note lack of <project> -- should be updated!

Prefix rule rationale

- All the .c→.o files exist in a single process
  - Cannot have filename collisions
  - Cannot have symbol collisions (variables, functions, or types)
- Also cannot collide with user app symbols

Prefix rule in project cores

- Outside of frameworks / components
  - Use `<project>` prefix for symbols
  - Subset as appropriate

    Func: `ompi_free_list_init()`

    Variable: `orte_plm_base`

    Type: `opal_list_t`
- Same rationale applies:
  - Avoid symbol collisions in OMPI
  - Avoid symbol collisions with MPI application

Public vs. private symbols

- Remember: this is middleware
  - Only make public what you need to
- OMPI defaults to private symbols
  - Must declare symbols to be public
  - Use “DECLSPEC” macro (per project)

```
ORTE_DECLSPEC bool orte_plm_rsh_started;
```

- Components invoked by function pointers
  - *Most symbols do not need to be public*

Portability

- Beware of Linux / GCC-specific-isms
  - Non-portable code goes in components
  - Or surrounded by `#if`
- All .c files must have code that is *called*
  - Do not have “constants.c” with no functions
  - Some linkers will drop .o’s with no callable code (e.g., OS X)

Run-Time Parameters

December 2012

101

Tunable parameters

- Philosophy: do not use constants
  - Use run-time parameters instead
- Referred to as "MCA parameters"
  - Somewhat misleading name
  - Means: service provided by the MCA base
  - Does not mean that they are restricted to MCA components or frameworks
  - OPAL, ORTE, and OMPI projects have "base" parameters, too

December 2012

102

Rationale

- Make everything a run-time decision
  - Give every param a "sensible" default
  - …where possible
- Parameters usually indicate:
  - Values (e.g., short/long message size)
  - Behavior (e.g., selection of algorithm)
- Much easier than recompiling

Intrinsic MCA param: framework name

- Each framework name is an MCA param
  - Specifies which components to open
- MCA base automatically registers it
  - Comma-delimited list of component names
  - Default value is empty (meaning "all")
- Inclusionary or exclusionary behavior

```
btl=tcp,self,sm
btl=^tcp
```

MCA param lookup order

1. "Override" value (set by API)
2. mpirun command line
   - `mpirun –mca <name> <value>`
3. Environment variable
   - `setenv OMPI_MCA_<name> <value>`
4. File
   - `$HOME/.openmpi/mca-params.conf`
   - `$prefix/etc/openmpi-mca-params.conf`

   (these locations are themselves tunable)
5. Default value

Using MCA parameters

- Characteristics
  - Strings and integers
  - Read-only (information) and read-write
  - Private and public
- **WARNING:** Lookup is slow!
  - Do not put in critical performance path
  - Do lookups at beginning of scope

MCA param examples

- btl_udverbs_version
  - Read-only, string version of the Verbs library that udverbs BTL was compiled against
- btl_tcp_if_include
  - Read-write, string list of IP interfaces to use
- btl
  - Read-write, list of BTL components to use
- orte_base_singleton
  - Private, whether this process is a singleton

Sidenote: `ompi_info` command

- Tells everything about OMPI installation
  - Finds all components and all params
  - Great for debugging
- Can look up specific component
  - `ompi_info --param <framework> <component`
  - Shows params, current values, where set from
  - Can also use keyword "all"
- `--parsable` option

MCA param API

- See `opal/mca/base/mca_base_param.h`
- Register and lookup functions
  - Several variations of each
- Components register params during component register (or open; deprecated)
  - `ompi_info` calls register/open/close on every component that it finds (to discover parameters)

Prefix rule and MCA params

- MCA params must be prefixed
  - Does not include the project name

  <framework>_<component>_<param_name>
- Examples

  `btl_tcp_mtu`

  `coll_basic_bcast_crossover`
- Register API function takes 3 strings
  - When registering in core, use:
    - Framework = project name
    - Component = "base"

Common Code Highlights

Init / finalize

- `<foo>_init()` to initialize something
- `<foo>_finalize()` to finalize something
- Examples:
  - `ompi_mpi_init()`: initializes OMPI layer, calls
  - `orte_init()`: initializes ORTE layer, calls
  - `opal_init()`: initializes OPAL layer
- Paired with `ompi_mpi_finalize()`, etc.
  - Frees resources, etc.

Init / finalize

- Not just used for overall projects
- Also used for individual subsystems

```
ompi_op_init()
  → ompi_op_finalize()
opal_datatype_init()
  → opal_datatype_finalize()
```


Utility code

- `<project>/util/*.[h,c]`
- E.g., OPAL has lots of compatibility code
  - asprintf, qsort, basename, strncpy
- Useful "add-on" code
  - Manipulate argv arrays (opal/util/argv.h)
  - printf debugging code (opal/util/output.h)
  - Error reporting (opal/util/show_help.h)
  - IP interfaces (opal/util/if.h)

Arrays of strings

- See `opal/util/arg.h`: `opal_argv_*()`
- Simple functions for maintaining argv-style arrays of strings
  - Prepend / append (resize if necessary)
  - Insert / remove (resize if necessary)
  - Split / join
  - Get length of array
  - Free array (and all strings)

`opal_output()` debugging code

- Function to emit debugging / error messages to stderr, stdout, file, syslog, …
  - Versions to simplify debugging output
  - Stream 0 prepends host, PID
- Printf-like arguments

```
opal_output(0, "hello, world");
opal_output_verbose(0, 10, "debugging…");
OPAL_OUTPUT(0, "--enable-debug only");
OPAL_OUTPUT_VERBOSE(….);
```

Friendly error messages

- `opal/util/opal_show_help.[h,c]`
- Print friendly messages for users
  - Message in text file rather than in source code
  - Can use printf substitutions (%s, %d, etc.)
  - De-duplicates messages
- Example
  - `opal_show_help(“help-mpi-btl-tcp.txt”, “invalid minimum port”, true, “ipv4”, default_value, hostname, port_num);`

Friendly error messages

- Contents of help-mpi-btl-tcp.txt:

```
[invalid minimum port]
WARNING: An invalid value was given for the
btl_tcp_port_min_%s. Legal values are in the
range [1 .. 2^16-1]. This value will be
ignored; OMPI will use the default value of
%d.

  Local host: %s
  Value:      %d
```

Discover IP interfaces

- See `opal/util/if.h`: `opal_if_*()`
- STL-like iteration over OS IP interfaces
  - Get info about each interface
  - Name, flags, netmask, loopback, etc.

Object system

- C-style reference counting object system
- "Poor man's C++"
  - Single inheritance
  - Constructors / destructors associated with each object instance
- Statically or dynamically allocated objects

Object system example

- Define class in header

```
typedef struct ompi_foo_t   {
  ompi_parent_t parent;
  void *first_member;
  ...
} ompi_foo_t;
OBJ_CLASS_DECLARATION(ompi_foo_t);
```

- `ompi_parent_t` must be a object
  - Root object is `opal_object_t`

December 2012 121

Object system example

- Must instantiate class descriptor in .c file

```
OBJ_CLASS_INSTANCE(ompi_foo_t,
    ompi_parent_t, foo_construct,
    foo_destruct);
```

- Local constructor / destructor functions
  - Both take one param: pointer to the object
- Constructors and destructors called recursively up the object stack

December 2012 122

Dynamic objects

- Create dynamically allocated object
  - Initial reference count set to 1

```
ompi_foo_t *foo = OBJ_NEW(ompi_foo_t);
```

- Increase reference count

```
OBJ_RETAIN(foo);
```

- Decrease reference count

```
OBJ_RELEASE(foo);
```

- Object destroyed and freed when reference count hits 0

Static objects

- Construct object

```
ompi_foo_t foo;
OBJ_CONSTRUCT(&foo, ompi_foo_t);
```

- Destruct object:

```
OBJ_DESTRUCT(&foo);
```

- Can use `OBJ_RETAIN/OBJ_RELEASE`, but
  - "Badness" if reference count hits 0
  - No automatic destruction if object goes out of scope

Object-based containers

- Lists, free lists, hash tables, value array, atomic LIFO list
- OMPI provide additional functionality
  - Shared memory fifo, red-black tree
- Such OBJ-based code usually found in `<project>/class`

Linked List

- `opal_list_t` is a doubly-linked list
- Item ownership transferred
  - No copies like in STL
  - Item only belong to one list
- Pointers to items never invalidated by `opal_list` functions
- O(1) insert, delete, join, get size
- Splice and sort routines
- Large debugging performance impact

…and others

- Go explore:
  - <project>/util
  - <project>/class
- If you find yourself writing “glue” code
  - Look first in util directories
  - If not there, consider if you should put it in util

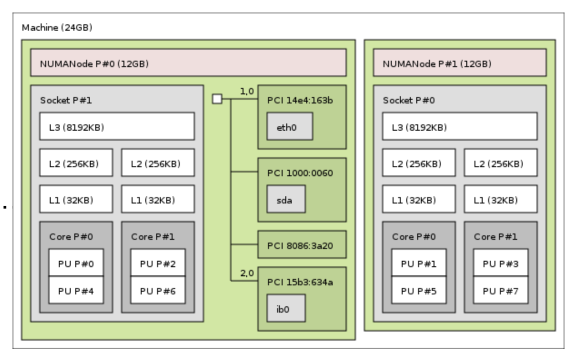
Hardware Locality (“hwloc”)

Hardware Locality (hwloc)

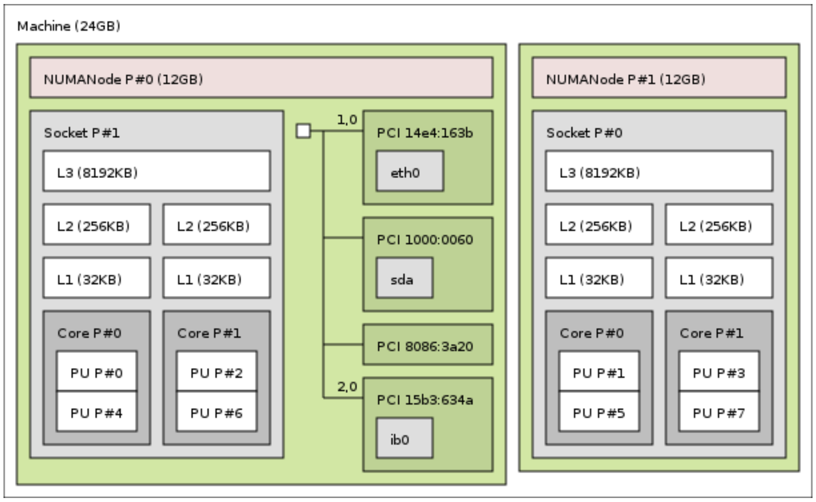
- High performance computing is all about location, Location, LOCATION!
  - NUMA is now common
  - Can consider network as next (several) level(s) of locality: NUNA
- Performant code must understand locality

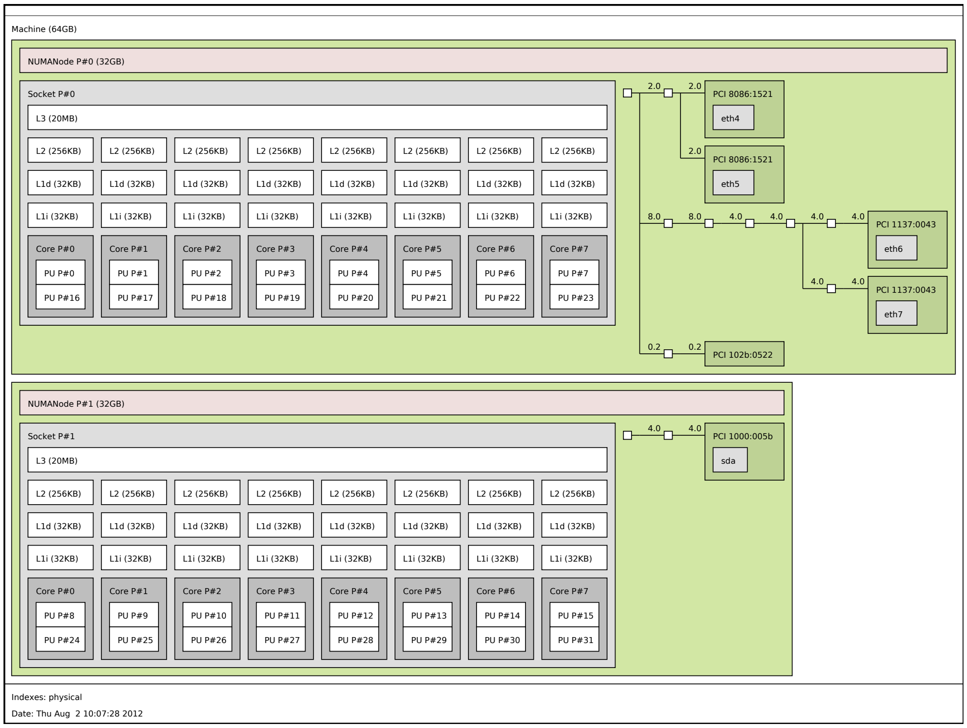
Hardware Locality (hwloc)

- Hwloc provides inside-the-server topology
  - CLI
    - Prettyprint
    - JPG, PNG, PDF, …
  - XML
  - C API
- lstopo(1) draws these pictures



Hwloc example





Hwloc capabilities

- Query topology information
  - As shown in previous pictures
  - C API provides tree of all that information
- Memory and processor affinity
  - hwloc-bind(1) *much mo' betta* than numactl(1)

```
$ hwloc-bind socket:0.core:3 my_program
hwloc_set_cpubind(…)
```

- Works on many different Oss
  - Linux, OS X, Windows, BSDs, …etc.

Hwloc sub-project

- An official sub-project of Open MPI
  - Has its own SVN repository
  - Developed mainly by INRIA (France)
  - A full copy of it is maintained on OMPI's SVN
- Fully documented
  - *Excellent* stand-alone tool (unrelated to MPI)
  - Highly encourage you to check it out

Open MPI's use of hwloc

- Wholly embeds a copy of hwloc
  - Can be compiled to use external hwloc
  - Embedded hwloc is certified to work properly
- Used to discover server topology
  - Effect processor and memory affinity
  - Query cache sizes
  - Query process peer locality (same socket, NUMA node, etc.)
  - Query PCI device locality

Open MPI's use of hwloc

- …and we're just getting started
- Anticipate much more use of the hwloc API over time
  - MPI collective algorithms
  - MPI shared memory point-to-point communications
  - …etc.

Questions?

Thank you!

CISCO

GREENPLUM®
A DIVISION OF EMC