Open MPI

Jeffrey M. Squyres

<>

15.1. Background

Open MPI [GFB+04] is an open source software implementation of The Message Passing Interface (MPI) standard. Before the architecture and innards of Open MPI will make any sense, a little background on the MPI standard must be discussed.

The Message Passing Interface (MPI)

The MPI standard is created and maintained by the MPI Forum, an open group consisting of parallel computing experts from both industry and academia. MPI defines an API that is used for a specific type of portable, high-performance inter-process communication (IPC): *message passing*. Specifically, the MPI document describes the reliable transfer of discrete, typed messages between MPI processes. Although the definition of an "MPI process" is subject to interpretation on a given platform, it usually corresponds to the operating system's concept of a process (e.g., a POSIX process). MPI is specifically intended to be implemented as middleware, meaning that upper-level applications call MPI functions to perform message passing.

MPI defines a high-level API, meaning that it abstracts away whatever underlying transport is actually used to pass messages between processes. The idea is that sending-process X can effectively say "take this array of 1,073 double precision values and send them to process Y". The corresponding receiving-process Y effectively says "receive an array of 1,073 double precision values from process X." A miracle occurs, and the array of 1,073 double precision values arrives in Y's waiting buffer.

Notice what is absent in this exchange: there is no concept of a connection occurring, no stream of bytes to interpret, and no network addresses exchanged. MPI abstracts all of that away, not only to hide such complexity from the upper-level application, but also to make the application portable across different environments and underlying message passing transports. Specifically, a correct MPI application is source-compatible across a wide variety of platforms and network types.

MPI defines not only point-to-point communication (e.g., send and receive), it also defines other communication patterns, such as *collective* communication. Collective operations are where multiple processes are involved in a single communication action. Reliable broadcast, for example, is where one process has a message at the beginning of the operation, and at the end of the operation, all processes in a group have the message. MPI also defines other concepts and communications patterns that are not described here. (As of this writing, the most recent version of the MPI standard is MPI-2.2 [For09]. Draft versions of the upcoming MPI-3 standard have been published; it may be finalized as early as late 2012.)

Uses of MPI

There are many implementations of the MPI standard that support a wide variety of platforms, operating systems, and network types. Some implementations are open source, some are closed source. Open MPI, as its name implies, is one of the open source implementations. Typical MPI transport networks include (but are not limited to): various protocols over Ethernet (e.g., TCP, iWARP, UDP, raw Ethernet frames, etc.), shared memory, and InfiniBand.

MPI implementations are typically used in so-called "high-performance computing" (HPC) environments. MPI essentially provides the IPC for simulation codes, computational algorithms, and other "big number crunching" types of applications. The input data sets on which these codes operate typically represent too much computational work for just one server; MPI jobs are spread out across tens, hundreds, or even thousands of servers, all working in concert to solve one computational problem.

That is, the applications using MPI are both parallel in nature and highly compute-intensive. It is not unusual for all the processor cores in an MPI job to run at 100% utilization. To be clear, MPI jobs typically run in dedicated environments where the MPI processes are the *only* application running on the machine (in addition to bare-bones operating system functionality, of course).

As such, MPI implementations are typically focused on providing extremely high performance, measured by metrics such as:

- Extremely low latency for short message passing. As an example, a 1-byte message can be sent from a user-level Linux process on one server, through an InfiniBand switch, and received at the target user-level Linux process on a different server in a little over 1 microsecond (i.e., 0.000001 second).
- Extremely high message network injection rate for short messages. Some vendors have MPI implementations (paired with specified hardware) that can inject up to 28 million messages per second into the network.
- Quick ramp-up (as a function of message size) to the maximum bandwidth supported by the underlying transport.
- Low resource utilization. All resources used by MPI (e.g., memory, cache, and bus bandwidth) cannot be used by the application. MPI implementations therefore try to maintain a balance of low resource utilization while still providing high performance.

Open MPI

The first version of the MPI standard, MPI-1.0, was published in 1994 [Mes93]. MPI-2.0, a set of additions on top of MPI-1, was completed in 1996 [GGHL+96].

In the first decade after MPI-1 was published, a variety of MPI implementations sprung up. Many were provided by vendors for their proprietary network interconnects. Many other implementations arose from the research and academic communities. Such implementations were typically "research-quality," meaning that their purpose was to investigate various high-performance networking concepts and provide proofs-of-concept of their work. However, some were high enough quality that they gained popularity and a number of users.

Open MPI represents the union of four research/academic, open source MPI implementations: LAM/MPI, LA/MPI (Los Alamos MPI), and FT-MPI (Fault-Tolerant MPI). The members of the PACX-MPI team joined the Open MPI group shortly after its inception.

The members of these four development teams decided to collaborate when we had the collective realization that, aside from minor differences in optimizations and features, our software code bases were quite similar. Each of the four code bases had their own strengths and weaknesses, but on the whole, they more-or-less did the same things. So why compete? Why not pool our resources, work together, and make an *even better* MPI implementation?

After much discussion, the decision was made to abandon our four existing code bases and take only the best *ideas* from the prior projects. This decision was mainly predicated upon the following premises:

- Even though many of the underlying algorithms and techniques were similar among the four code bases, they each had radically different implementation architectures, and would be incredible difficult (if not impossible) to merge.
- Each of the four also had their own (significant) strengths and (significant) weaknesses. Specifically, there were features and architecture decisions from each of the four that were desirable to carry forward. Likewise, there were poorly optimized and badly designed code in each of the four that were desirable to leave behind.
- The members of the four developer groups had not worked directly together before. Starting with an entirely new code base (rather than advancing one of the existing code bases) put all developers on equal ground.

Thus, Open MPI was born. Its first Subversion commit was on November 22, 2003.

15.2. Architecture

For a variety of reasons (mostly related to either performance or portability), C and C++ were the only two possibilities for the primary implementation language. C++ was eventually discarded because different C++ compilers tend to lay out structs/classes in memory according to different optimization algorithms, leading to different on-the-wire network representations. C was therefore chosen as the primary implementation language, which influenced several architectural design decisions.

When Open MPI was started, we knew that it would be a large, complex code base:

- In 2003, the current version of the MPI standard, MPI-2.0, defined over 300 API functions.
- Each of the four prior projects were large in themselves. For example, LAM/MPI had over 1,900 files of source code, comprising over 300,000 lines of code (including comments and blanks).
- We wanted Open MPI to support more features, environments, and networks than all four prior projects put together.

We therefore spent a good deal of time designing an architecture that focused on three things:

1. Grouping similar functionality together in distinct abstraction layers.
2. Using run-time loadable plugins and run-time parameters to choose between multiple different implementations of the same behavior.
3. Not allowing abstraction to get in the way of performance.

Abstraction Layer Architecture

Open MPI has three main abstraction layers, shown in Figure 15.1:

- *Open, Portable Access Layer (OPAL)*: OPAL is the bottom layer of Open MPI's abstractions. Its

abstractions are focused on individual processes (versus parallel jobs). It provides utility and glue code such as generic linked lists, string manipulation, debugging controls, and other mundane—yet necessary—functionality.
OPAL also provides Open MPI's core portability between different operating systems, such as discovering IP interfaces, sharing memory between processes on the same server, processor and memory affinity, high-precision timers, etc.

- *Open MPI Run-Time Environment (ORTE)* (pronounced "or-tay"): An MPI implementation must provide not only the required message passing API, but also an accompanying run-time system to launch, monitor, and kill parallel jobs. In Open MPI's case, a parallel job is comprised of one or more processes that may span multiple operating system instances, and are bound together to act as a single, cohesive unit.
In simple environments with little or no distributed computational support, ORTE uses `rsh` or `ssh` to launch the individual processes in parallel jobs. More advanced, HPC-dedicated environments typically have schedulers and resource managers for fairly sharing computational resources between many users. Such environments usually provide specialized APIs to launch and regulate processes on compute servers. ORTE supports a wide variety of such managed environments, such as (but not limited to): Torque/PBS Pro, SLURM, Oracle Grid Engine, and LSF.
- *Open MPI (OMPI)*: The MPI layer is the highest abstraction layer, and is the only one exposed to applications. The MPI API is implemented in this layer, as are all the message passing semantics defined by the MPI standard.
Since portability is a primary requirement, the MPI layer supports a wide variety of network types and underlying protocols. Some networks are similar in their underlying characteristics and abstractions; some are not.

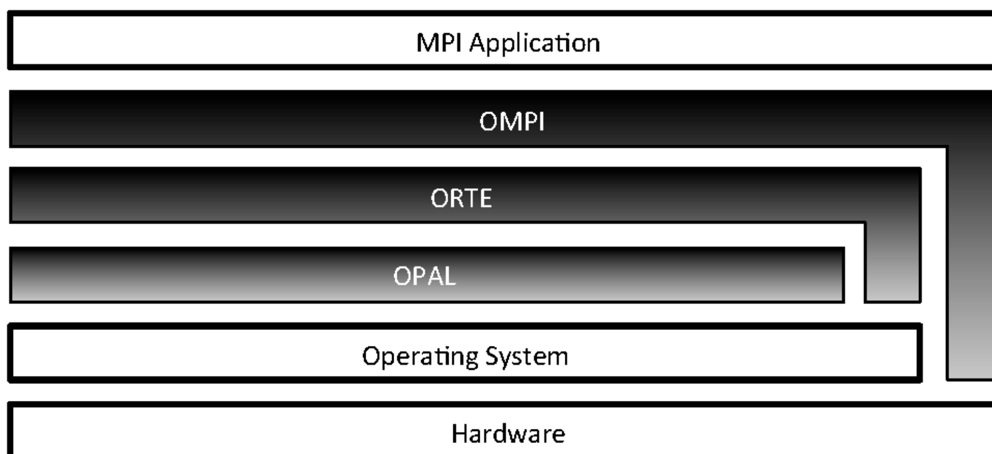

Figure 15.1: Abstraction layer architectural view of Open MPI showing its three main layers: OPAL, ORTE, and OMPI

Although each abstraction is layered on top of the one below it, for performance reasons the ORTE and OMPI layers can bypass the underlying abstraction layers and interact directly with the operating system and/or hardware when needed (as depicted in Figure 15.1). For example, the OMPI layer uses OS-bypass methods to communicate with certain types of NIC hardware to obtain maximum networking performance.

Each layer is built into a standalone library. The ORTE library depends on the OPAL library; the OMPI library depends on the ORTE library. Separating the layers into their own libraries has acted as a wonderful tool for preventing abstraction violations. Specifically, applications will fail to link if one layer incorrectly attempts to use a symbol in a higher layer. Over the years, this abstraction enforcement mechanism has saved many developers from inadvertently blurring the lines between the three layers.

Plugin Architecture

Although the initial members of the Open MPI collaboration shared a similar core goal (produce a portable, high-performance implementation of the MPI standard), our organizational backgrounds, opinions, and agendas were—and still are—wildly different. We therefore spent a considerable amount of time designing an architecture that would allow us to be different, even while sharing a common code base.

Run-time loadable *components* were a natural choice (a.k.a., dynamic shared objects, or "DSOs", or "plugins"). Components enforce a common API but place few limitations on the implementation of that API. Specifically: the same interface behavior can be implemented multiple different ways. Users can then choose, at run time, which plugin(s) to use. This even allows third parties to independently develop and distribute their own Open MPI plugins outside of the core Open MPI package. Allowing arbitrary extensibility is quite a liberating policy, both within the immediate set of Open MPI developers and in the greater Open MPI community.

This run-time flexibility is a key component of the Open MPI design philosophy and is deeply integrated throughout the entire architecture. Case in point: the Open MPI v1.5 series includes 155 plugins. To list just a few examples, there are plugins for different `memcpy()` implementations, plugins for how to

launch processes on remote servers, and plugins for how to communicate on different types of underlying networks.

One of the major benefits of using plugins is that multiple groups of developers have freedom to experiment with alternate implementations without affecting the core of Open MPI. This was a critical feature, particularly in the early days of the Open MPI project. Sometimes the developers didn't always know what was the right way to implement something, or sometimes they just disagreed. In both cases, each party would implement their solution in a component, allowing the rest of the developer community to easily compare and contrast. Code comparisons can be done without components, of course, but the component concept helps guarantee that all implementations expose exactly the same external API, and therefore provide exactly the same required semantics.

As a direct result of the flexibility that it provides, the component concept is utilized heavily throughout all three layers of Open MPI; in each layer there are many different types of components. Each type of component is enclosed in a *framework*. A component belongs to exactly one framework, and a framework supports exactly one kind of component. Figure 15.2 is a template of Open MPI's architectural layout; it shows a few of Open MPI's frameworks and some of the components that they contain. (The rest of Open MPI's frameworks and components are laid out in the same manner.) Open MPI's set of layers, frameworks, and components is referred to as the Modular Component Architecture (MCA).

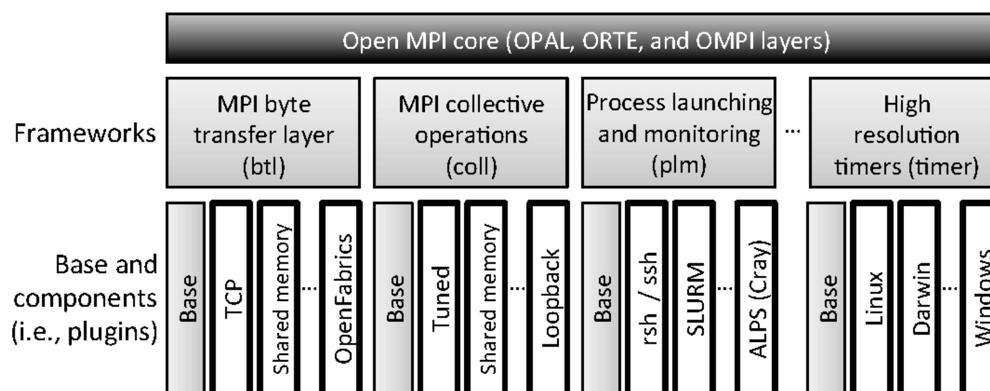

Figure 15.2: Framework architectural view of Open MPI, showing just a few of Open MPI's frameworks and components (i.e., plugins). Each framework contains a *base* and one or more components. This structure is replicated in each of the layers shown in Figure 15.1. The sample frameworks listed in this figure are spread across all three layers: `btl` and `coll` are in the OMPI layer, `plm` is in the ORTE layer, and `timer` is in the OPAL layer.

Finally, another major advantage of using frameworks and components is their inherent composability. With over 40 frameworks in Open MPI v1.5, giving users the ability to mix-n-match different plugins of different types allows them to create a software stack that is effectively tailored to their individual system.

Plugin Frameworks

Each framework is fully self-contained in its own subdirectory in the Open MPI source code tree. The name of the subdirectory is the same name as the framework; for example, the `memory` framework is in the `memory` directory. Framework directories contain at least the following three items:

1. *Component interface definition:* A header file named `<framework>.h` will be located in the top-level framework directory (e.g., the Memory framework contains `memory/memory.h`). This well-known header file defines the interfaces that each component in the framework must support. This header includes function pointer typedefs for the interface functions, structs for marshaling these function pointers, and any other necessary types, attribute fields, macros, declarations, etc.
2. *Base code*: The `base` subdirectory contains the glue code that provides the core functionality of the framework. For example, the `memory` framework's base directory is `memory/base`. The base is typically comprised of logistical grunt work such as finding and opening components at run-time, common utility functionality that may be utilized by multiple components, etc.
3. *Components*: All other subdirectories in the framework directory are assumed to be components. Just like the framework, the names of the components are the same names as their subdirectories (e.g., the `memory/posix` subdirectory contains the POSIX component in the Memory framework).

Similar to how each framework defines the interfaces to which its components must adhere, frameworks also define other operational aspects, such as how they bootstrap themselves, how they pick components to use, and how they are shut down. Two common examples of how frameworks differ in their setup are many-of-many versus one-of-many frameworks, and static versus dynamic frameworks.

Many-of-many frameworks.

Some frameworks have functionality that can be implemented multiple different ways in the same process. For example, Open MPI's point-to-point network framework will load multiple driver plugins to allow a single process to send and receive messages on multiple network types.

Such frameworks will typically open all components that they can find and then query each component, effectively asking, "Do you want to run?" The components determine whether they want to run by

examining the system on which they are running. For example, a point-to-point network component will look to see if the network type it supports is both available and active on the system. If it is not, the component will reply "No, I do not want to run", causing the framework to close and unload that component. If that network type *is* available, the component will reply "Yes, I want to run", causing the framework to keep the component open for further use.

One-of-many frameworks.

Other frameworks provide functionality for which it does not make sense to have more than one implementation available at run-time. For example, the creation of a consistent checkpoint of a parallel job—meaning that the job is effectively "frozen" and can be arbitrarily resumed later—must be performed using the same back-end checkpointing system for each process in the job. The plugin that interfaces to the desired back-end checkpointing system is the *only* checkpoint plugin that must be loaded in each process—all others are unnecessary.

Dynamic frameworks.

Most frameworks allow their components to be loaded at run-time via DSOs. This is the most flexible method of finding and loading components; it allows features such as explicitly *not* loading certain components, loading third-party components that were not included in the main-line Open MPI distribution, etc.

Static frameworks.

Some one-of-many frameworks have additional constraints that force their one-and-only-one component to be selected at compile time (versus run time). Statically linking one-of-many components allows direct invocation of its member functions (versus invocation via function pointer), which may be important in highly performance-sensitive functionality. One example is the `memcpy` framework, which provides platform-optimized `memcpy()` implementations.

Additionally, some frameworks provide functionality that may need to be utilized before Open MPI is fully initialized. For example, the use of some network stacks require complicated memory registration models, which, in turn, require replacing the C library's default memory management routines. Since memory management is intrinsic to an entire process, replacing the default scheme can only be done pre-`main`. Therefore, such components must be statically linked into Open MPI processes so that they can be available for pre-`main` hooks, long before MPI has even been initialized.

Plugin Components

Open MPI plugins are divided into two parts: a *component* struct and a *module* struct. The component struct and the functions to which it refers are typically collectively referred to as "the component." Similarly, "the module" collectively refers to the module struct and its functions. The division is somewhat analogous to C++ classes and objects. There is only one component per process; it describes the overall plugin with some fields that are common to all components (regardless of framework). If the component elects to run, it is used to generate one or more modules, which typically perform the bulk of the functionality required by the framework.

Throughout the next few sections, we'll build up the structures necessary for the TCP component in the BTL (byte transfer layer) framework. The BTL framework effects point-to-point message transfers; the TCP component, not surprisingly, uses TCP as its underlying transport for message passing.

Component struct.

Regardless of framework, each component contains a well-known, statically allocated and initialized component struct. The struct must be named according to the template `mca_<framework>_<component>_component`. For example, the TCP network driver component's struct in the BTL framework is named `mca_btl_tcp_component`.

Having templated component symbols both guarantees that there will be no name collisions between components, and allows the MCA core to find any arbitrary component struct via `dlsym(2)` (or the appropriate equivalent in each supported operating system).

The base component struct contains some logistical information, such as the component's formal name, version, framework version adherence, etc. This data is used for debugging purposes, inventory listing, and run-time compliance and compatibility checking.

```
struct mca_base_component_2_0_0_t {
    /* Component struct version number */
    int mca_major_version, mca_minor_version, mca_release_version;

    /* The string name of the framework that this component belongs to,
       and the framework's API version that this component adheres to */
    char mca_type_name[MCA_BASE_MAX_TYPE_NAME_LEN + 1];
    int mca_type_major_version, mca_type_minor_version,
        mca_type_release_version;

    /* This component's name and version number */
    char mca_component_name[MCA_BASE_MAX_COMPONENT_NAME_LEN + 1];
```

```
    int mca_component_major_version, mca_component_minor_version,
        mca_component_release_version;

    /* Function pointers */
    mca_base_open_component_1_0_0_fn_t mca_open_component;
    mca_base_close_component_1_0_0_fn_t mca_close_component;
    mca_base_query_component_2_0_0_fn_t mca_query_component;
    mca_base_register_component_params_2_0_0_fn_t
        mca_register_component_params;
};
```

The base component struct is the core of the TCP BTL component; it contains the following function pointers:

- *Open.* The *open* call is the initial query function invoked on a component. It allows a component to initialize itself, look around the system where it is running, and determine whether it wants to run. If a component can always be run, it can provide a `NULL` open function pointer.
  The TCP BTL component *open* function mainly initializes some data structures and ensures that invalid parameters were not set by the user.
- *Close.* When a framework decides that a component is no longer needed, it calls the *close* function to allow the component to release any resources that it has allocated. The close function is invoked on all remaining components when processes are shutting down. However, *close* can also be invoked on components that are rejected at run time so that they can be closed and ignored for the duration of the process.
  The TCP BTL component *close* function closes listening sockets and frees resources (e.g., receiving buffers).
- *Query.* This call is a generalized "Do you want to run?" function. Not all frameworks utilize this specific call—some need more specialized query functions.
  The BTL framework does not use the generic *query* function (it defines its own; see below), so the TCP BTL does not fill it in.
- *Parameter registration.* This function is typically the first function called on a component. It allows the component to register any relevant run-time, user-settable parameters. Run-time parameters are discussed further below.
  The TCP BTL component *register* function creates a variety of user-settable run-time parameters, such as one which allows the user to specify which IP interface(s) to use.

The component structure can also be extended on a per-framework and/or per-component basis. Frameworks typically create a new component struct with the component base struct as the first member. This nesting allows frameworks to add their own attributes and function pointers. For example, a framework that needs a more specialized query function (as compared to the *query* function provided on the basic component) can add a function pointer in its framework-specific component struct.

The MPI `btl` framework, which provides point-to-point MPI messaging functionality, uses this technique.

```
struct mca_btl_base_component_2_0_0_t {
    /* Base component struct */
    mca_base_component_t btl_version;
    /* Base component data block */
    mca_base_component_data_t btl_data;

    /* btl-framework specific query functions */
    mca_btl_base_component_init_fn_t btl_init;
    mca_btl_base_component_progress_fn_t btl_progress;
};
```

As an example of the TCP BTL framework query functions, the TCP BTL component `btl_init` function does several things:

- Creates a listening socket for each "up" IPv4 and IPv6 interface.
- Creates a module for each "up" IP interface.
- Registers the tuple `(IP address, port)` for each "up" IP interface with a central repository so that other MPI processes know how to contact it.

Similarly, plugins can extend the framework-specific component struct with their own members. The `tcp` component in the `btl` framework does this; it caches many data members in its component struct.

```
struct mca_btl_tcp_component_t {
    /* btl framework-specific component struct */
    mca_btl_base_component_2_0_0_t super;
```

```
    /* Some of the TCP BTL component's specific data members */
    /* Number of TCP interfaces on this server */
    uint32_t tcp_addr_count;

    /* IPv4 listening socket descriptor */
    int tcp_listen_sd;

    /* ...and many more not shown here */
};
```

This struct-nesting technique is effectively a simple emulation of C++ single inheritance: a pointer to an instance of a `struct mca_btl_tcp_component_t` can be cast to any of the three types such that it can be used by an abstraction layer than does not understand the "derived" types.

That being said, casting is generally frowned upon in Open MPI because it can lead to incredibly subtle, difficult-to-find bugs. An exception was made for this C++-emulation technique because it has well-defined behaviors and helps enforce abstraction barriers.

Module struct.

Module structs are individually defined by each framework; there is little commonality between them. Depending on the framework, components generate one or more module struct instances to indicate that they want to be used.

For example, in the BTL framework, one module usually corresponds to a single network device. If an MPI process is running on a Linux server with three "up" Ethernet devices, the TCP BTL component will generate three TCP BTL modules; one corresponding to each Linux Ethernet device. Each module will then be wholly responsible for all sending and receiving to and from its Ethernet device.

Tying it all together.

Figure 15.3 shows the nesting of the structures in the TCP BTL component, and how it generates one module for each of the three Ethernet devices.

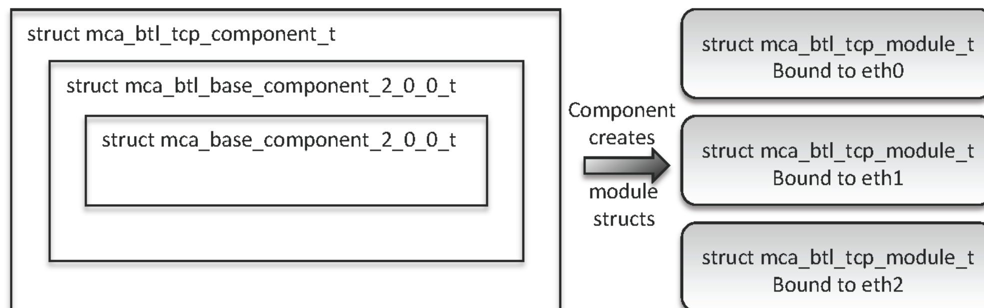

Figure 15.3: The left side shows the nesting of structures in the TCP BTL component. The right side shows how the component generates one module struct for each "up" Ethernet interface.

Composing BTL modules this way allows the upper-layer MPI progression engine both to treat all network devices equally, and to perform user-level channel bonding.

For example, consider sending a large message across the three-device configuration described above. Assume that each of the three Ethernet devices can be used to reach the intended receiver (reachability is determined by TCP networks and netmasks, and some well-defined heuristics). In this case, the sender will split the large message into multiple fragments. Each fragment will be assigned—in a round-robin fashion—to one of the TCP BTL modules (each module will therefore be assigned roughly one third of the fragments). Each module then sends its fragments over its corresponding Ethernet device.

This may seem like a complex scheme, but it is surprisingly effective. By pipelining the sends of a large message across the multiple TCP BTL modules, typical HPC environments (e.g., where each Ethernet device is on a separate PCI bus) can sustain nearly maximum bandwidth speeds across multiple Ethernet devices.

Run-Time Parameters

Developers commonly make decisions when writing code, such as:

- Should I use algorithm A or algorithm B?
- How large of a buffer should I preallocate?
- How long should the timeout be?
- At what message size should I change network protocols?
- …and so on.

Users tend to assume that the developers will answer such questions in a way that is generally suitable for most types of systems. However, the HPC community is full of scientist and engineer power users who want to aggressively tweak their hardware and software stacks to eke out every possible compute cycle. Although these users typically do not want to tinker with the actual code of their MPI implementation, they *do* want to tinker by selecting different internal algorithms, choosing different

resource consumption patterns, or forcing specific network protocols in different circumstances.

Therefore, the MCA parameter system was included when designing Open MPI; the system is a flexible mechanism that allows users to change internal Open MPI parameter values at run time. Specifically, developers register string and integer MCA parameters throughout the Open MPI code base, along with an associated default value and descriptive string defining what the parameter is and how it is used. The general rule of thumb is that rather than hard-coding constants, developers use run-time-settable MCA parameters, thereby allowing power users to tweak run-time behavior.

There are a number of MCA parameters in the base code of the three abstraction layers, but the bulk of Open MPI's MCA parameters are located in individual components. For example, the TCL BTL plugin has a parameter that specifies whether only TCPv4 interfaces, only TCPv6 interfaces, or both types of interfaces should be used. Alternatively, another TCP BTL parameter can be set to specify exactly which Ethernet devices to use.

Users can discover what parameters are available via a user-level command line tool (`ompi_info`). Parameter values can be set in multiple ways: on the command line, via environment variables, via the Windows registry, or in system- or user-level INI-style files.

The MCA parameter system complements the idea of run-time plugin selection flexibility, and has proved to be quite valuable to users. Although Open MPI developers try hard to choose reasonable defaults for a wide variety of situations, every HPC environment is different. There are inevitably environments where Open MPI's default parameter values will be unsuitable—and possibly even detrimental to performance. The MCA parameter system allows users to be proactive and tweak Open MPI's behavior for their environment. Not only does this alleviate many upstream requests for changes and/or bug reports, it allows users to experiment with the parameter space to find the best configuration for their specific system.

15.3. Lessons Learned

With such a varied group of core Open MPI members, it is inevitable that we would each learn *something*, and that as a group, we would learn many things. The following list describes just a few of these lessons.

Performance

Message-passing performance and resource utilization are the king and queen of high-performance computing. Open MPI was specifically designed in such a way that it could operate at the very bleeding edge of high performance: incredibly low latencies for sending short messages, extremely high short message injection rates on supported networks, fast ramp-ups to maximum bandwidth for large messages, etc. Abstraction is good (for many reasons), but it must be designed with care so that it does not get in the way of performance. Or, put differently: carefully choose abstractions that lend themselves to shallow, performant call stacks (versus deep, feature-rich API call stacks).

That being said, we also had to accept that in some cases, abstraction—not architecture—must be thrown out the window. Case in point: Open MPI has hand-coded assembly for some of its most performance-critical operations, such as shared memory locking and atomic operations.

It is worth noting that Figures 15.1 and 15.2 show two different *architectural* views of Open MPI. They do not represent the run-time call stacks or calling invocation layering for the high performance code sections.

Lesson learned:

It is acceptable (albeit undesirable) and unfortunately sometimes necessary to have gross, complex code in the name of performance (e.g., the aforementioned assembly code). However, it is *always* preferable to spend time trying to figure out how to have good abstractions to discretize and hide complexity whenever possible. A few weeks of design can save literally hundreds or thousands of developer-hours of maintenance on tangled, subtle, spaghetti code.

Standing on the Shoulders of Giants

We actively tried to avoid re-inventing code in Open MPI that someone else has already written (when such code is compatible with Open MPI's BSD licensing). Specifically, we have no compunctions about either directly re-using or interfacing to someone else's code.

There is no place for the "not invented here" religion when trying to solve highly complex engineering problems; it only makes good logistical sense to re-use external code whenever possible. Such re-use frees developers to focus on the problems unique to Open MPI; there is no sense re-solving a problem that someone else has solved already.

A good example of this kind of code re-use is the GNU Libtool Libltdl package. Libltdl is a small library that provides a portable API for opening DSOs and finding symbols in them. Libltdl is supported on a wide variety of operating systems and environments, including Microsoft Windows.

Open MPI *could* have provided this functionality itself—but why? Libltdl is a fine piece of software, is actively maintained, is compatible with Open MPI's license, and provides exactly the functionality that was needed. Given these points, there is no realistic gain for Open MPI developers to re-write this functionality.

Lesson learned:

When a suitable solution exists elsewhere, do not hesitate to integrate it and stop wasting time trying to re-invent it.

Optimize for the Common Case

Another guiding architectural principle has been to optimize for the common case. For example, emphasis is placed on splitting many operations into two phases: setup and repeated action. The assumption is that setup may be expensive (meaning: slow). So do it *once* and get it over with. Optimize for the much more common case: repeated operation.

For example, `malloc()` can be slow, especially if pages need to be allocated from the operating system. So instead of allocating just enough bytes for a single incoming network message, allocate enough space for a *bunch* of incoming messages, divide the result up into individual message buffers, and set up a freelist to maintain them. In this way, the *first* request for a message buffer may be slow, but *successive* requests will be much faster because they will just be de-queues from a freelist.

Lesson learned:

Split common operations into (at least) two phases: setup and repeated action. Not only will the code perform better, it may be easier to maintain over time because the distinct actions are separated.

Miscellaneous

There are too many more lessons learned to describe in detail here; the following are a few more lessons that can be summed up briefly:

- We were fortunate to draw upon 15+ years of HPC research and make designs that have (mostly) successfully carried us for more than eight years. When embarking on a new software project, *look to the past*. Be sure to understand what has already been done, *why* it was done, and what its strengths and weaknesses were.
- The concept of components—allowing multiple different implementations of the same functionality—has saved us many times, both technically and politically. Plugins are good.
- Similarly, we continually add and remove frameworks as necessary. When developers start arguing about the "right" way to implement a new feature, add a framework that fronts components that implement that feature. Or when newer ideas come along that obsolete older frameworks, don't hesitate to delete such kruft.

Conclusion

If we had to list the three *most* important things that we've learned from the Open MPI project, I think they would be as follows:

- One size does not fit all (users). The run-time plugin and companion MCA parameter system allow users flexibility that is necessary in the world of portable software. Complex software systems cannot (always) magically adapt to a given system; providing user-level controls allows a human to figure out—and override—when the software behaves sub-optimally.
- Differences are good. Developer disagreements are good. Embrace challenges to the status quo; do not get complacent. A plucky grad student saying "Hey, check this out…" can lead to the basis of a whole new feature or a major evolution of the product.
- Although outside the scope of this book, people and community matter. A lot.

This work is made available under the Creative Commons Attribution 3.0 Unported license. Please see the full description of the license for details.

Back to top
Back to *The Architecture of Open Source Applications*.