JEFF SQUYRES

Why Are There So Many MPI Implementations?

You (And Your Code) Will Be Assimilated

We are the MPI. You will be assimilated. Your code and technological distinctiveness will be added to our own. Resistance is futile. Your code will run everywhere... won't it?

The Story So Far

In each of these monthly columns, I am careful to distinguish between the MPI standards specification and the behavior of a given MPI implementation. There are many MPI implementations available - some vendors even have more than one. But why? Wasn't the goal of MPI to simplify all of this and make it easy to have portable parallel processing applications? I have personally seen clusters with over *twenty* different MPI implementations installed - it was each user's responsibility to determine which one they should use for their application (and set their `PATH` and other environment factors properly). This scenario is unfortunately not uncommon.

Indeed, with the myriad of different implementations available, independent software vendors (ISVs) attempting to sell closed-source parallel applications that use MPI typically have considerable logistical QA challenges. They already have to QA certify their application across a large number of hardware and operating system combinations; add a third dimension of MPI implementations, and the total number of platforms to QA certify against grows exponentially.

But Aren't MPI Applications Portable?

To be fair, the MPI Forum's goal was to enable *source code portability*, allowing users to recompile the same source code on different platforms with different MPI implementations. Even though some aspects of the MPI standard are not provided by all MPI implementations, MPI applications are largely source code portable across a wide variety of systems. Indeed, application source code portability is one of the largest contributing factors to the success of MPI.

Binary portability — the ability to run the same executable on multiple platforms (a la Java applets) or the ability to run the same executable with different MPI implementations on the same platform — was not one of the MPI Forum's original goals. As such, the MPI standard makes no effort to standardize the values of constants, the types of C handles, and several other surface-level aspects that make an MPI implementation distinct.

After MPI-2 was published, proposals have been periodically introduced for binary MPI interoperability (such as between the open source MPI implementations). Although these proposals have never succeeded, it has not been because the implementers think that this is a Bad Idea — reducing the logistics of users and ISVs is definitely a Good Thing™. They have failed because each MPI implementation has made fundamental design choices that preclude this kind of binary interoperability. More on this below.

Note that this goal says nothing about *performance portability* - the potential for unmodified applications to run with the same performance characteristics in multiple MPI implementations. Previous editions of this column have discussed the hazards about implied assumptions about your MPI implementation (e.g., whether `MPI_SEND` will block or not).

But the basic questions remain: why are there so many MPI implementations? And why are they so different?

To answer these questions, one really needs to look at what an MPI implementation has to provide to adhere to the standard, and then what the goals of that particular implementation are.

The Letter of the Law

As has been mentioned many times in this column, the MPI standard — consisting of two documents: MPI-1 and MPI-2 — is the bible to an MPI implementer. An implementation must adhere to all of the standard's definitions, semantics, and API details in order to be conformant.

At its core, an MPI implementation is about message passing - the seemingly simple act of moving bytes from one process to another. However, there are a large number of other services and data structures that accompany this core functionality. The MPI specification contains over 300 API functions and tens of pre-defined constants. Each of these API functions have specific, defined behavior (frequently related to other API functions) by which you must obey.

The data structures required to support such a complex web of interactions are, themselves, complex. Open MPI's internal communicator structure, for example, contains 24 members (17 of which either contain or point to other structures). The creation and run-time maintenance of these structures is an intricate task, requiring careful coding and painstaking debugging.

The Spirit of the Law

Even with the MPI standard, there are many places — both deliberate and [unfortunately] unintentional — where the text is ambiguous, and an MPI developer has to make a choice in the implementation. Should `MPI_SEND` block or return immediately? Should a given message be sent eagerly or use a rendezvous protocol? Should progress occur on an asynchronous or polling basis? Are user threads supported? Are errors handled? And if so, how?

And so on — the list is endless.

Each implementer answers these questions differently, largely depending on the goals of the specific implementation. Some MPI implementations are "research quality" and were created to study a specific set of experimental issues. Such implementations are likely to take short cuts in many areas and concentrate on their particular research topic(s). Other implementations are hardened/production quality, and must be able to run large parallel jobs for weeks at a time without leaking resources or crashing.

Some implementations are targeted at specific platforms, interconnects, run-time systems, etc., while others are designed to be portable across some subset of the (platform, network, run-time system) tuple. In some ways, writing single-purpose MPI implementations (e.g., for a specific set of hardware/network/run-time system) can be dramatically simpler than writing portable systems. Since it only has to work on one operating system, with one compiler, and one network, the code is *far* less complex than a portable system.

That being said, I've had discussions with developers of such single-system implementations and, despite the homogeneity of their target systems, their job is not easy. I've known developers who cheerfully break out logic analyzers to watch bus activity during an MPI run in order to fully understand *all* activity on the machine in order to further optimize their MPI. I even know of one [unnamed] vendor's implementation that used self-modifying code in order to avoid two cache misses and reduce latency by a few tens of nanoseconds. That particular trick had to get sign-offs from several levels of management in order to pass QA, but in the end, contributed to delivering an extremely high-performing MPI to the company's customers.

Let's take a short tour of some other choices that an MPI implementer has to make.

MPI Handles: Pointers or Integers?

This may seem like a trivial matter, but it has wide-reaching effects throughout the entire MPI implementation. A communicator, for example, has a bunch of internal data associated with it (the members of the group, the error-handler associated with it, whether the communicator is an inter- or intra-communicator, and so on). An implementation typically bundles all this information together in a C structure (or C++ object) and provides the application with some kind of handle to it. The handle that the application sees is of type `MPI_Comm` — but what should its real type be: a pointer to the structure/object, or an integer index into an array of all currently-allocated communicators?

Surprisingly, this issue incurs deep religious rifts between MPI implementers.

Using integers for handles means that there is no loss of performance between the C and Fortran bindings — both sets use indirect addressing to find the back-end structure (note that MPI specifically defines Fortran handles to be integers because Fortran — at least Fortran 77 — has no concept of a pointer). Note, however, that in multi-threaded environments, it is necessary to obtain a lock before ex-

The Penalty of Fortran

Most MPI implementations are written in C and/or C++. In addition to C and C++ bindings, the MPI standard specifies language bindings in two flavors of Fortran: one that will work with Fortran 77 (and later) compilers and one that will work with Fortran 90 (and later) compilers.

For MPI implementations that provide them, the Fortran bindings are typically "wrapper" functions, meaning that they are actually written in C (or C++) and simply translate the Fortran arguments to C/C++ conventions before invoking a back-end implementation function. In many cases, the back-end function is the corresponding C function. For example, the Fortran binding for `MPI_SEND` performs argument translation and then invokes the C binding for `MPI_SEND`.

The argument translation may also involve some lookups — for example, converting Fortran integer handles into back-end structures or objects. In a threaded environment, this likely involves some form of locking.

Not all implementations work this way, but many do. It is worth investigating your MPI implementation's behavior if you are trying to squeeze every picosecond of performance out of your parallel environment.

amining the array because another thread may have grown (and therefore moved) the array.

Conversely, using pointers means that the Fortran bindings may have to perform translation from the integer to a pointer (probably through indirect addressing), but the C bindings can access the back-end data directly and have no need for additional lookup or locking of index arrays. Finally, on platforms where the size of a Fortran `INTEGER` is the same size as a pointer, this is a non-issue — each can be used interchangeably (e.g., the Fortran integer handle can actually be the C pointer value). This case is not true for all platforms, however.

The size of MPI handles is visible in `mpi.h`, and is therefore a key aspect of the MPI implementation's interface to user applications.

What's in an MPI_Status?

The `MPI_Status` object, as defined by the MPI standard, is different than all other MPI objects: not only does it have public data members, the user is responsible for allocating and freeing `MPI_Status` objects. This requirement means that its structure must be defined in `mpi.h` — including any internal data members (so that pointer math in the application can be accurate).

Although the standard disallows MPI applications from using the internal data members, the fact that `MPI_Status` is accessed by value (and not through a handle) means that its size is a key aspect of the MPI implementation's interface to user applications.

User Threads

A fundamental decision that an MPI needs to make during the beginning of its development is whether to allow multiple user threads, and if so, whether to support concurrency within the MPI library. It is fundamentally easier for an MPI implementation to assume that there will only be one user thread in the library at a given time, either by only allowing single-threaded MPI applications or using a single, global mutex to protect all entry points to the library — effectively only allowing one thread into the library at a time.

When multiple, concurrent user threads are allowed, some form of locking must be used in the MPI library to protect internal data structures yet (assumedly) still allow fine-grained concurrency. For example, it is desirable to allow multiple threads executing `MPI_SEND` to progress more-or-less independently. Note that this may not be possible if both sends are going to the same destination (or otherwise must share the same network channel) or if the threads are running on the same CPU. But in general, the goal of allowing multiple user threads within the MPI library is to offer a high degree of concurrency wherever possible.

Unless this is considered during the initial design, it is difficult (if not impossible) to graft a fine-grained locking system onto the MPI implementation's internal progression engine(s). This issue is not really related to MPI, however, it is a design-for-threads issue.

Progress: Asynchronous or Polling?

Many MPI implementations only make progress on pending message passing operations when an MPI function is invoked. For example, even if an application started a non-blocking send with `MPI_ISEND`, the message may not be fully sent until `MPI_TEST` and/or `MPI_WAIT` is invoked. This procedure is common for single-threaded MPI implementations (although this is a different issue than allowing multiple simultaneous application-level threads in the MPI library).

Other MPI implementations offer true asynchronous progress, potentially utilizing specialized communication hardware or extra, hidden threads in the library that can make message passing progress regardless of what the application's threads are doing.

Designing for asynchronous progress really needs to be included from the start. Either specific hardware needs to be used or many of the same issues with multiple application threads need to be addressed. It is therefore difficult (if not impossible) to add true asynchronous support to a polling-only MPI implementation.

Binary [In]Compatibility

Several of the issues discussed

Resources

MPI Forum	www.mpi-forum.org
MPI — The Complete Reference: Volume 1, The MPI Core (2nd ed) (The MIT Press) by Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. ISBN 0-262-69215-5.	
MPI — The Complete Reference: Volume 2, The MPI Extensions (The MIT Press) by William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. ISBN 0-262-57123-4.	
NCSA MPI tutorial	webct.ncsa.uiuc.edu:8900/public/MPI

above (the types of MPI handles, the contents of `MPI_Status`, and the values of constants) can be simplified into a single phrase: have a common `mpi.h` and `mpif.h`. If all implementations used the same `mpi.h` and `mpif.h`, this would go a long way towards binary compatibility on a single platform.

However, as was recently pointed out to me, that's not really enough. Even though different `libmpi.so` instances could be used at run-time with a single executable, it is desirable to have a common `mpirun` as well (and other related MPI command line tools). This requirement means commonality between implementations of `MPI_INIT` — how to receive the list of processes in `MPI_COMM_WORLD`, their location, how to wait for or forcibly terminate a set of MPI processes, etc. It also has implications in the implementation of the MPI-2 dynamic process functions (`MPI_COMM_SPAWN` and friends). This situation translates to a unified run-time environment between MPI implementations.

Given the wide variety of run-time environments used by MPI implementations, this does not seem likely in the near future. Never say "never," of course, but the run-time environment comprises a good percentage of code in an MPI implementation — it is the back-end soul of the machine. More specifically: given that the MPI interface is standardized, there is at least a hope of someday specifying a common `mpi.h` and `mpif.h`. But the run-time environment in an MPI implementation is not specified in the MPI standard at all - there is little to no similarity between each implementation's run-time system. As such, merging them into a single, common system seems unlikely.

Where to Go From Here?

Yes, Virginia, MPI implementations are extremely complicated. Although binary compatibility is unlikely, source code compatibility has been and always will be available. This feature is part of the strength of MPI. The other is an unrelenting desire of developers to optimize the heck out of their MPI implementation. Take comfort that your code will not only run everywhere, it will likely run *well* everywhere.

Got any MPI questions you want answered? Wondering why one MPI does *this* and another does *that*? Send them to jsquyres@open-mpi.org.

Jeff Squyres is a post-doctoral research associate at Indiana University and is the one of the lead technical architects of the Open MPI project. You can reach Jeff at jsquyres@open-mpi.org