MPI Mechanic

December 2003

Provided by ClusterWorld for
Jeff Squyres
cw.squyres.com

www.clusterworld.com

JEFF SQUYRES

Definitions and Fundamentals: The Message Passing Interface (MPI)

The term "MPI" is often bandied about in high-performance computing discussions — sometimes with love, sometimes with disdain, and sometimes as a buzzword that no one really understands. This month's column attempts to provide both overview information about what MPI is for managers as well as basic technical information about MPI for developers.

What is MPI?

I am often asked to provide a quick summary of exactly what MPI *is*. I typically emphasize the following points:

- MPI stands for the Message Passing Interface.
- MPI is a standard defined by a large committee of experts from industry and academia.
- The design of MPI was heavily influenced by decades of "best practices" in parallel computing.
- Although typically collectively referred to as the "MPI standard," there are actually two documents (MPI-1 and MPI-2)
- Implementations of the MPI standard provide message passing (and related) services for parallel applications.
- There are many implementations of the MPI standard.

Essentially: the MPI standard defines a set of functions that can be used by applications to pass messages from one MPI process to another. MPI actually defines a lot more services than just message passing — but the heart and soul of MPI is (as its name implies) passing messages between MPI processes.

As noted, there are actually two documents that comprise the MPI standard: MPI-1 and MPI-2. MPI-1 is the "core" set of MPI services for message passing. It provides abstractions and mechanisms for basic message passing between MPI processes (as well as some additional features that are helpful for general parallel computing). MPI-2 is a set of extensions and functionality beyond what is defined in MPI-1 such as dynamic process control, one-sided message passing, parallel I/O, etc.

Since MPI is simply a specification, it is not tied to any particular computing hardware or software environment. As such, implementations of the MPI specification are available on a wide variety of platforms. For example, most major parallel computing vendors provide an MPI implementation that is highly tuned for their hardware. Several ISVs (Independent Software Vendors) also provide MPI implementations targeted for certain families of platforms. Open source/freeware MPI implementations are available from researchers on a variety of different platforms; some are targeted at production-quality usage while others are solely intended as research-quality projects.

MPI Implementations

As noted above, most parallel hardware vendors have their own version of MPI that is highly tuned for their hardware. Specifically, vendor-provided MPI implementations provide the software necessary for applications to utilize high bandwidth, low latency message passing in customized software and hardware environments. Most clusters built with such hardware should use these vendor-provided MPI implementations to extract the maxi-

Should you parallelize your application?

Before parallelizing your application, it is advisable to perform some analysis to see if it will benefit from parallelization. Generally speaking, the point of parallelizing an application is to make it run faster. Hence, it only makes sense to parallelize an application if:

- The amount of work performed is large enough (i.e., the application takes a long time to run in serial), or
- The amount of data to be processed is too large for one node (i.e., there is too much data to fit in RAM).

If neither of the above conditions are met, the overhead added by parallelization may actually cause the application to run *slower*. For example, there is no point in parallelizing an application that only takes a few seconds to run. However, an application that takes several hours to run in serial and can easily have its work divided into multiple, semi-independent parts is probably a good candidate for parallelization.

MPI function notation

The MPI standard defines all functions in three languages: C, C++, and Fortran. Every function defined by the MPI standard will necessarily have a different binding in each language. For example, the bindings for the MPI initialization function are listed in the "MPI Initialization Function Language Bindings" table.

LANGUAGE	BINDING
C	int MPI_Init(int *argc, char ***argv);
C++	int MPI::Init(int& argc, char**& argv);
Fortran	MPI_INIT(IERROR) INTEGER IERROR

To refer to the MPI function without referring to a specific language binding, the MPI standard uses all capital letters: `MPI_INIT`.

mum amount of performance in their applications.

Clusters built with commodity components tend to have more of a choice of which MPI implementation to use — many cluster administrators choose to install multiple different implementations and use whichever one provides the best performance on an application-by-application basis.

Two notable open source MPI implementations are LAM/MPI (from Indiana University) and MPICH (from Argonne National Labs). Both have full MPI-1 support as well as support for some portions of MPI-2. These implementations are both freely available on the web (see the "Resources" sidebar) and can run in a wide variety of parallel and cluster environments.

The examples provided in this column will tend to use the LAM/MPI implementation (reflecting my obvious bias as one of the LAM/MPI developers) simply for consistency of command syntax, run-time semantics, and output. Note, however, that since MPI is a standard, example code provided in this column should generally be portable to other MPI implementations. Normally the only real difference the users should encounter between MPI versions is how MPI programs are started.

Who Uses MPI?

Two main groups of people use MPI: parallel application developers and end users.

Parallel application developers will actively use MPI function calls in their code to effect parallelism and message passing between MPI processes. They compile, debug, and run their applications in an MPI implementation's run-time environment. Developers are therefore knowledgeable in the API defined by the MPI standard as well as the semantics of their MPI implementation's run-time environment, commands, and MPI functionality.

Depending on the application, end users may run an application without even being aware that it uses MPI (or that it is parallel). As such, end users may *use* MPI, but have varying levels of familiarity with their particular MPI implementation's commands and run-time environments.

Basic MPI Definitions

Many aspects of MPI are described in terms of interactions between "MPI processes." MPI processes are usually independent processes but may also be threads.

The MPI standard specifically does not define an execution model; it is therefore left up to the implementation to define how parallel applications are run and exactly what an MPI process is. A collection of MPI processes that are launched together are referred to as an MPI application.

Messages are sent from (and received into) typed buffers in MPI processes. You can loosely think of using MPI as: "take this array of integers and send its contents to process X."

Using typed buffers allows a heterogeneous-capable MPI implementation to ensure that endian ordering is preserved.

For example, when sending the integer value of "7" from a Sun workstation, the MPI implementation will automatically convert (if necessary) the data such that the value of "7" is received at the target process — even if the target process is running on an Intel workstation.

Note that the MPI application programmer interface (API) is specified in three different languages: C, C++, and Fortran. Hence, it is possible to write applications in any of these languages that utilize the same underlying MPI functionality.

Indeed, it is even permissible to mix languages within the same MPI application, if desired (although certain data structures and handles need to be "translated" between the different languages). A common example of this scenario is a C-based MPI application that uses a Fortran MPI numerical library.

"Hello, World" MPI Code Example

The following is a simple "hello world" program that shows the basics of the MPI C API.

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char
 **argv) {
int rank, size;

MPI_Init(&argv, &argv);
MPI_Comm_size(MPI_COMM_
  WORLD, &size);
MPI_Comm_rank(MPI_COMM_
  WORLD, &rank);
printf("Hello, world.
  I am %d of %d.\n", rank,
  size);
MPI_Finalize();
return 0;
}
```

The first use of MPI is on line 2: `#include <mpi.h>`. The header file `<mpi.h>` is defined by the MPI standard to be available in all implementations.

It must provide all necessary prototypes and external variable declarations that may be required for user MPI programs. Hence, it should always be included in any compilation unit that will utilize MPI functionality.

Line 7 is the first use of an MPI function: `MPI_INIT`. As its name implies, `MPI_INIT`'s main purpose is to initialize the MPI communications layer. `MPI_INIT` must be invoked before any other MPI function.

Lines 8 and 9 call `MPI_COMM_SIZE` and `MPI_COMM_RANK`, respectively. In this example, the first argument to both of these functions is `MPI_COMM_WORLD`. `MPI_COMM_WORLD` is a pre-defined MPI *communicator* — an ordered set of MPI processes and a unique communications context. `MPI_COMM_WORLD` is meaningful after `MPI_INIT` returns; it implicitly contains the set of MPI processes in the MPI application.

`MPI_COMM_SIZE` and `MPI_COMM_RANK` query `MPI_COMM_WORLD` to obtain the total number of MPI processes in the application and a unique identification for *this* MPI process, respectively. The results of these functions are stored in the variables `size` and `rank`.

Hence, after executing line 9, the MPI process knows how many peer processes it has as well as its own unique identity in `MPI_COMM_WORLD` (note that ranks are expressed in the range of [0, size-1] for a given communicator).

Finally, `MPI_FINALIZE` performs any necessary cleanup and shuts down the MPI layer within the MPI process. It is not legal to invoke any other MPI functions after `MPI_FINALIZE` returns.

MPI processes vs. ranks

Many MPI programmers tend to refer to MPI processes as "ranks." This is not technically correct.

- An MPI process is a unique entity.
- A rank value is is only unique in the context of a specific communicator.

A single rank value may therefore refer to multiple different MPI processes. For example, it is not correct to say "send to rank 0." It is more correct to say "send to `MPI_COMM_WORLD` rank 0."

Unfortunately, even [communicator, rank] pairs are not always unique. It is easy to imagine cases where multiple communicators containing disjoint sets of processes are referred to by the same variable. Consider a communicator referred to by a variable named `row`: "`row` rank 0" therefore does not necessarily refer to a unique MPI process. In this case, it is typically safer to refer to the MPI process through its `MPI_COMM_WORLD` rank.

But the situation becomes even more complicated when we introduce the concept of MPI-2 dynamic processes — where it is possible to have multiple, simultaneous instances of `MPI_COMM_WORLD` with disjoint sets of processes. Dynamic MPI processes — and the issues surrounding them — will be explored in a future edition of this column.

Compiling the Program

Each MPI implementation provides a different mechanism for compiling MPI programs. Some implementations provide "wrapper" compilers that add command-line flags such as `-I`, `-L`, and `-l` before invoking an underlying compiler (e.g., `cc`, `gcc`, `icc`, etc.). Since the "wrapper" compilers simply manipulate command line options, they can be treated just like the underlying compiler. Specifically, any command-line options that are given to the wrapper compiler will simply be passed on to the back-end compiler.

LAM/MPI provides `mpicc`, `mpiCC` (or `mpic++` on case-insensitive filesystems), and `mpif77` wrapper compilers for C, C++, and Fortran, respectively. For example, to compile the sample "hello world"

MPI program with LAM/MPI:

```
$ mpicc hello.c -o hello
```

Users are *strongly* encouraged to use the wrapper compilers to compile all MPI applications instead of directly invoking the underlying compiler (and attempting to provide the implementation's `-I`, `-L`, `-l`, etc. flags). It is also advisable to check that execution path and environment variables are set correctly for your MPI implementation. The same program can be compiled with MPICH using the same command.

Running the MPI Application with LAM/MPI

Before parallel MPI programs can be run, the LAM/MPI run-time environment must be started (or "booted") with the `lamboot` command.

For simplicity's sake, this example assumes running on a traditional `rsh`/`ssh`, Beowulf-style cluster where the user can login to all nodes without needing to interactively provide a password or passphrase. If you are required to enter a password, then you will have difficulty with the following steps.

`lamboot` expects an argument specifying the name of a boot schema file (or "hostfile") indicating on which nodes to launch the run-time environment. The simplest boot schema file is a text file with one hostname (or IP address) per line. Consider the following boot schema file (named "`myhosts`"):

```
node1.example.com
node2.example.com
node3.example.com
node4.example.com
```

Run the `lamboot` command with `myhosts` as an argument:

MPI Resources

- MPI Forum: http://www.mpi-forum.org/
- LAM/MPI: http://www.lam-mpi.org/
- MPICH: http://www.mcs.anl.gov/mpi/mpich/
- 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: http://webct.ncsa.uiuc.edu:8900/public/MPI

```
$ lamboot myhosts
```

When `lamboot` completes successfully, the LAM run-time environment has been booted on all the nodes listed in `myhosts` and is available to run MPI programs. The `mpirun` command is used to launch MPI applications.

The `C` switch is used to tell LAM to launch one MPI process per "CPU" (as indicated in the boot schema file; if no CPU count is indicated — as in this example — one CPU per node is assumed) in the run-time environment. For example, the following launches four MPI "hello" processes:

```
$ mpirun C hello
Hello, world.  I am 0 of 4.
Hello, world.  I am 1 of 4.
Hello, world.  I am 2 of 4.
Hello, world.  I am 3 of 4.
```

Both `mpirun` and `lamboot` support many more command line features, options, and modes of operation; be sure to see their respective manual pages for more details.

Running the MPI Application with MPICH

Running a program with MPICH is not quite as involved as LAM/MPI. Check that your execution path and environment variables are set correctly for your version of MPICH. To run the program under MPICH, you will need a machine file which looks strikingly familiar to the LAM schema file:

```
node1.example.com
node2.example.com
node3.example.com
node4.example.com
```

Again, you must be able to login to the nodes in your machine file without entering a password. To run the MPICH complied program, simply create a machine file called `machines` with the names of the machines in your cluster then execute the following.

```
mpirun -np 2 -machinefile
  machines hello
```

You should see a similar, but not necessarily identical output order as the LAM example. You may also see something *funny* with the output in any case. We will talk about this and some other runtime issues next month.

Jeff Squyres is a research associate at Indiana University and is the lead developer for the LAM implementation of MPI. jsquyres@lam-mpi.org