MPI Mechanic

February 2004

Provided by ClusterWorld for
Jeff Squyres
cw.squyres.com

ClusterWorld™

REDEFINING HIGH PERFORMANCE COMPUTING

www.clusterworld.com

Copyright © 2004 ClusterWorld, All Rights Reserved

For individual private use only. Not to be reproduced or distributed without prior consent from ClusterWorld (info@clusterworld.com)

What *really* happens during MPI_INIT

In the first two columns, we covered the basics and fundamentals: what MPI is, some simple MPI example programs, and how to compile and run them. For this months column, we will look at the "ping-pong" example program in Listing One.

The ping-pong program starts up MPI (`MPI_INIT`) on line 8, gets the total number of peer MPI processes and finds its own identity (`MPI_COMM_SIZE` and `MPI_COMM_RANK` on lines 9 and 10), does some basic message passing (`MPI_SEND` and `MPI_RECV` on lines 15-22), and then finishes up (`MPI_FINALIZE`) on line 24. This simple example shows the use of six MPI functions. Surprisingly complex parallel applications can be written with just these six MPI functions.

Although there are hundreds of available MPI API functions, many MPI applications find that a relatively small subset is suitable for their needs. In that light, this month's column examines the six MPI functions used in the ping-pong example in detail. This analysis includes not only what the MPI standard specifies for each function's functionality, but also some of the more in-depth (and potentially implementation-specific) issues that frequently arise with MPI applications, especially when using one MPI application with multiple MPI implementations.

LISTING ONE
One MPI Ping-Pong Program

```
 1 #include <stdio.h>
 2 #include <mpi.h>
 3
 4 int main(int argc, char **argv) {
 5 int rank, size, mesg, tag = 123;
 6 MPI_Status status;
 7
 8 MPI_Init(&argv, &argv);
 9 MPI_Comm_size(MPI_COMM_WORLD, &size);
10 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11 if (size < 2) {
12 printf("Need at least 2 processes!\n");
13 } else if (rank == 0) {
14 mesg = 11;
15 MPI_Send(&mesg,1,MPI_INT,1,tag,MPI_COMM_WORLD);
16 MPI_Recv(&mesg,1,MPI_INT,1,tag,MPI_COMM_
   WORLD,&status);
17 printf("Rank 0 received "%d" from rank 1\
   n",mesg);
18 } else if (rank == 1) {
19 MPI_Recv(&mesg,1,MPI_INT,0,tag,MPI_COMM_
   WORLD,&status);
20 printf("Rank 1 received "%d" from rank 0/
   n",mesg);
21   mesg = 42;
22   MPI_Send(&mesg,1,MPI_INT,0,tag,MPI_COMM_WORLD);
23  }
24  MPI_Finalize();
25  return 0;
26 }
27 /* To compile and run the program:
28  *    mpicc pingpong.c -o pingpong
29  *    mpirun -np 2 pingpong
30  */
```

MPI_INIT: The Rest of the Story

Conceptually, `MPI_INIT` is very simple: start up the MPI communications layer — it is almost always the first MPI function invoked. There are, however, several functions that can be invoked before `MPI_INIT` — see the sidebar for more information. Most MPI implementations recommend that `MPI_INIT` be invoked as close to the beginning of `main()` as possible. This rule is as much as most MPI developers need to know. But it can be useful to know more — particularly when debugging. You may be debugging a parallel application or the parallel run-time environment (RTE) itself (e.g., when setting up a new cluster). In these cases — particularly when using closed-source MPI implementations — it may be helpful to understand what `MPI_INIT` is actually trying to do.

Every MPI implementation's `MPI_INIT` works differently.

Generally, its job is to create, initialize, and make available all aspects of the message passing layer. This may even include launching additional processes (typically peer MPI_COMM_WORLD processes). That is, some MPI implementations establish an MPI RTE *before* MPI_INIT (and MPI_INIT simply establishes a parallel application in that RTE) while others create the RTE *during* MPI_INIT. Although some parallel environments are a natural fit to one model or the other, since the MPI standard does not specify an MPI application's execution model, both are perfectly valid approaches.

MPI_INIT also typically allocates resources such as shared memory, local interprocess communication channels, network communication channels (TCP sockets, Myrinet ports, and/or other network-specific resources), and/or "special" memory used for communicating with specialized networking hardware. Failure to obtain any of these resources will likely cause the entire parallel application to abort, or, even worse, "hang." Most modern MPI implementations display friendly messages when this kind of error occurs.

The important thing to realize is that MPI_INIT is likely to involve communication and/or coordination between at least some subsets of processes in the MPI application. As such, although some MPI implementations take pains to optimize the process, MPI_INIT is typically treated as a "catch-all" function. Implementations tend to gather as much setup and initialization in MPI_INIT as possible in order to optimize the run of the program itself. For example, MPI_INIT may coordinate between MPI processes to probe and discover the application's topology, potentially allowing underlying communication optimization later in the application. Hence, even though MPI_INIT may be "long," its cost can be amortized over the duration of the parallel application's run.

To Block or Not To Block

MPI_SEND and MPI_RECV are called "blocking" by the MPI-1 standard, but they may or may not actually block. Whether or not an unmatched send will block typically depends on how much buffering the implementation provides. For example, short messages are usually sent "eagerly" — regardless of whether a matching receive has been posted or not. Long messages may be sent with a rendezvous protocol, meaning that it will not actually complete until the target has initiated a matching receive. This behavior is legal because the semantics of MPI_SEND do not actually define whether a message has been sent when it returns. The only guarantee that MPI makes is that the buffer is able to be re-used when MPI_SEND returns.

Receives, by their definition, will not return until a matching message has actually been received. This situation may be "immediate," for example, if a matching short message was previously eagerly sent.

This is called an "unexpected" message, and MPI implementations typically provide some level of implicit buffering for this condition: eagerly-sent, unmatched messages are simply stored in internal buffering at the target until a matching receive is posted by the application. A local memory copy is all that is necessary to complete the receive.

Note that it is also legal for an MPI implementation to provide zero buffering — to effectively disallow unexpected messages and block MPI_SEND until a matching receive is posted (regardless of the size of the message). MPI applications that assume at least some level of underlying buffering are not conformant, and may run to completion under one MPI implementation but block in another.

No MPI_INIT Needed

Unknown to most MPI users, there are some functions that can legally called before MPI_INIT. These are only are MPI_INITIALIZED, MPI_FINALIZED, and MPI_GET_VERSION, which check to see if MPI_INIT and MPI_FINALIZE have been invoked, and return the version of the MPI standard that is provided by the MPI implementation, respectively. In addition, these functions can also be called after an MPI_FINALIZE has been issued.

MPI_COMM_SIZE and MPI_COMM_RANK: Query Functions

The MPI_COMM_SIZE and MPI_COMM_RANK functions are local functions; they rarely require communication and coordination outside of the local MPI process. The MPI implementation almost always determines the size and ordering of processes in a communicator when that communicator is created (e.g., when MPI_COMM_WORLD is created during MPI_INIT). Hence, MPI_COMM_SIZE and MPI_COMM_RANK are typically local (and immediate) lookups.

MPI_SEND and MPI_RECV: Basic Sending and Receiving

MPI provides several different flavors of sending and receiving mes-

sages. MPI_SEND and MPI_RECV are called "blocking," meaning that they will not return until the message buffer is available for use.

Notice that this indicates nothing about the status of the message being sent or received — it *only* guarantees that the application is able to re-use the message buffer.

Many of the arguments to MPI_SEND and MPI_RECV (and the other point-to-point send and receive functions in MPI) are the same.

- (buffer, count, datatype): This triple identifies the beginning of a buffer, the type of data, and how many items of that datatype will be sent/received. This data directly implies the size (and shape) of the buffer. Providing type information for the buffer allows the MPI implementation to perform endian-swapping if the source and target processes have different endian biases.

- tag: This value is a logical separator between message types; it is typically used to distinguish between different messages sent on the same communicator.

- (rank, communicator): This pair uniquely identifies the target peer process and a communication scope. For sends, the target is the destination process; for receives, it is the source process. Remember that each communicator contains a unique communication context; MPI guarantees that a message sent on one communicator can only be received on that same communicator.

A sent message will only be received by a "matching" receive. A send and receive "match" when the sender and receiver use corresponding (tag, rank, communicator) triples:

- The tag arguments must correspond; the sender must use the same tag as the receiver, or the receiver must use the special MPI_ANY_TAG constant.

- The target rank arguments must correspond; the sender must use the receiver's rank and the receiver must use either the sender's rank or the special MPI_ANY_SOURCE constant.

- The communicator arguments must refer to the same underlying communicator.

MPI_FINALIZE: Shutting Down

Analogous to MPI_INIT, MPI_FINALIZE is conceptually simple: it shuts down the MPI communications layer. It is almost always the last MPI function invoked. Most MPI implementations recommend that MPI_

What's in a Rank?

A criticism of MPI-1 is that the communicator concept was modeled around fixed groups of processes; processes could not arbitrarily join and leave an MPI application. Although this issue was fully addressed in MPI-2 when dynamic processes were introduced into the standard (although still modeled on fixed process sets), MPI-1 was deliberately designed around the concept of static process groups. Some of the reasons behind this decision include:

1 Using static groups and fixed process ordering allows the user application to know how many peer processes it has. This information can affect setup decisions such as array sizes, data splitting, and communication patterns. Because this size will never change, the application developer is relieved of the burden of potentially having to reshape data structures in the middle of a run.

2 Imposing fixed ordering allows the application developer to easily use identification-based decisions. Since the MPI implementation is responsible for creating a consistent global ordering, the MPI developer is guaranteed that a rank number in a given communicator refers to the same MPI process by all of its peers. For example, a common identification-based decision is to use rank 0 in MPI_COMM_WORLD to be a "manager," and have all other ranks be "workers."

3 The landmark paper entitled "Impossibility of Distributed Consensus with One Faulty Process" by M. J. Fischer, N. A. Lynch, and M. S. Paterson published in the Journal of the ACM in April 1985 proves that group membership in an asynchronous environment (such as a cluster) is impossible to determine absolutely. Although in practice, dynamic group membership can be determined for relatively small groups, as modern clusters grow in size, the "FLP Impossibility Result" would have created theoretical (and therefore practical) problems for large-scale asynchronous parallel applications utilizing dynamic group membership.

See MPI, page xx

Hence, even though MPI-1 was criticized for being based on fixed, ordered process groups, it has proved to be a sound decision. Although an MPI implementation must be designed for scalability in order to handle arbitrarily large parallel applications, by point #3, the possibility of such an implementation is not limited by theoretical constraints.

MPI, from page xx

`FINALIZE` be invoked as close to exiting the MPI process as possible. `MPI_FINALIZE` must be called to finish all correct MPI programs.

Note that it is possible for `MPI_FINALIZE` to block. It is `MPI_FINALIZE`'s responsibility to shut down all network connections, releases all resources, and generally cleans up the process space. This may require completing unfinished message passing. Although this will usually not occur with the simple MPI API calls discussed so far in this column, it can happen and is legal behavior for an MPI implementation.

Where To Go From Here?

Now you know more about the basics of MPI than you ever thought you would. The intent of this month's column is to provide insight when debugging parallel applications by describing some aspects of how an MPI implementation works. Now you know *why* your parallel application seems to block for no reason, or *why* your application seems to go slowly at first and then speed up in later iterations.

> The semantics of `MPI_SEND` do not actually define whether a message has been sent when it returns

Stay tuned: next month, we'll discuss MPI collective communication capabilities.

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

Resources

MPI Forum
- www.mpi-forum.org

NCSA MPI tutorial
- webct.ncsa.uiuc.edu:8900public/MPI/

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.

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.