# MPI Mechanic

March 2004

ClusterWorld™

REDEFINING HIGH PERFORMANCE COMPUTING

www.clusterworld.com

JEFF SQUYRES

# Zen and the Art of MPI Collectives

March — a time of renewal. Tulips and grass pushing through the snow. Hundreds and thousands of blades of grass, all growing at the same time. In parallel. Hmm. How do they know? Do they coordinate? Perhaps they have some kind of collective intelligence? Do they use MPI?

## The Story So Far

In previous editions of this column, we've talked about the six basic functions of MPI, how `MPI_INIT` and `MPI_FINALIZE` actually work, and discussed in agonizing detail the differences between MPI ranks, MPI processes, and CPU processors. Armed with this knowledge, you can write large, sophisticated parallel programs. So what's next?

Collective communication is a next logical step — MPI's native ability to involve a group of MPI processes together in a single communication, possibly involving some intermediate computation.

## MPI Collective Basic Concepts

Many parallel algorithms include the concept of a collective operation - an operation in which multiple processes participate in order to compute a result. A global sum is an easy example to discuss — each process contributes an integer that is summed in an atomic fashion and the final result is made available (perhaps just to a single "root" process, or perhaps made available to all participating processes).

A brief recap: MPI defines all point-to-point communications in terms of "communicators." Communicators are a fixed set of ordered processes with a unique context. Communication that occurs on a communicator is guaranteed to not collide with communications occurring on other communicators.

MPI also defines collective communication in terms of communicators. All collective operations explicitly involve every process in a communicator. Specifically: a collective will not be complete until all processes in the communicator have participated. Due to the nature of some of MPI's pre-defined collective operations (see the sidebar "Will That Collective Block?"), this may or may not imply blocking behavior. There is one exception to this rule: `MPI_BARRIER` is guaranteed not to return until all processes in the communicator have entered the barrier.

There are two main kinds of collectives defined in MPI: rooted and non-rooted. "Rooted" operations have a single process acting as the explicit originator or receiver of data. For example, `MPI_BCAST` broadcasts a buffer from a root process to all other processes in

### Why Not Use MPI_SEND and MPI_RECV?

An obvious question that arises is: why bother? Why not simple use a linear loop over `MPI_SEND` and `MPI_RECV` to effect the same kind of operations? In short: it's all about optimization. The MPI built-in collectives usually offer the following advantages:

- **Avoid `MPI_SEND` and `MPI_RECV`:** An MPI implementation is able to optimize each collective operation in many ways that are not available to user applications. For example, on SMP nodes, collective operations may occur directly in shared memory and avoid the entire `MPI_SEND` / `MPI_RECV` protocol stack.

- **Multiple algorithms:** There are typically many algorithms that can be used for implementing a collective operation (even when using `MPI_SEND` / `MPI_RECV`), each yielding different performance characteristics in a given run-time environment. Factors such as the size and configuration of the communicator as well as the size and shape of the data to be communicated may influence the specific algorithm that is used. Much research has been conducted in this area over the past 20 years; let the MPI implementors worry about it - not you.

- **Performance portability:** Collective algorithms that work well on a cluster may or may not work well on "big iron" parallel machines. Using the collective operations in the native MPI implementation usually means that you'll get algorithms that are tuned for the platform that your application is running on - one of the main goals of MPI.

The moral of the story: it is generally safer to trust your MPI implementation's collective algorithms than to implement your own. While no MPI implementation is perfect, most modern versions do a reasonable job of optimizing collective operations.

the communicator; MPI_GATHER gathers buffers from each process in the communicator to a single, combined buffer in the root process. "Non-rooted" operations are those where there is either no explicit originator/receiver or all processes are sending/receiving data. MPI_BARRIER, for example, has no explicit senders/receivers, but MPI_ALLGATHER both performs a gather operation from all processes in the processor and makes the result available to all processes.

## Barrier Synchronization

One of the simplest collective operations to describe is the barrier synchronization. MPI's function for this is MPI_BARRIER. It takes a single significant argument: a communicator.

One should note that, while the argument lists of the MPI C, C++, and Fortran bindings for a given function are typically similar in terms of "significant" arguments, there are some minor differences. One notable difference is that all MPI Fortran calls take a final "ierr" argument that the C and C++ bindings do not. The "ierr" argument is used for passing errors back to the caller (errors are handled differently in the C and C++ bindings).

As described above, MPI_BARRIER does not return until all processes in the communicator have entered the barrier. The seemingly-simple barrier operation is a good example illustrating that a variety of different algorithms that can be used:

- **linear:** the root receives from all processes followed by the root sending to all processes
- **logrithmic:** a binomial tree gather to the root followed by a binomial tree scatter from the root
- **2-level latency split algorithm:** a local gather, global gather, global scatter, and finally a local scatter
- **N-level latency split algorithm:** similar to the above, but for N levels, not 2
- **shared memory:** each process increments a shared counter; when the counter equals the number of processes, exit the barrier

The barrier operation has been researched for years (particularly in the area of shared memory algorithms; the shared memory algorithm listed above will typically provide dismal performance); many other algorithms are possible; the above list just a few possibilities.

There's no good reason for a user application to include implementations for all of these algorithms; the MPI implementation should provide some form of an optimized barrier (which may be one or more of the above algorithms) so that the user application does not have to worry about such issues.

Be wary of overusing MPI_BARRIER. It is frequently tempting to insert barriers for ease of control and simplicity of code. However, barriers are usually unnecessary when writing MPI programs — MPI's tag/communicator matching rules for point-to-point communicator and "fence" operation for one-sided operations (to be described in a later column) typically obviate the need for barriers. Indeed, a barrier that is executed in every iteration of a repetitive code can introduce artificial performance limitations.

## Broadcast

Another simple collective operation to describe is the broadcast: data is sent from one process to all other processes in a communicator.

Its function prototype is similar to MPI_SEND; it takes a buffer, count, MPI datatype, and communicator — just like MPI_SEND. But,

**LISTING ONE**

### Simple Broadcast MPI Program

```
void simple_broadcast(void) {
 int rank, value;
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 if (rank == 0) {
  printf("Enter a value: ");
  scanf("%d", &value);
 }
 MPI_Bcast(&value,1,MPI_INT,0,MPI_COMM_
   WORLD);
 printf("Rank %d has value: %d\n",rank,
   value);
}
```

### Resources

**MPI Forum**
- www.mpi-forum.org

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

rather than requiring a destination rank and tag, MPI_BCAST accepts a root rank specifying which process contains the source buffer. Listing One shows a simple program using MPI_BCAST.

MPI_COMM_WORLD rank 0 will prompt for an integer and then broadcast it to all other processes. Note that all processes call MPI_BCAST in exactly the same way; the same parameters are used in each process.

At the root (MPI_COMM_WORLD rank 0), the value variable is used as an input buffer; value is used as an output buffer in all other processes. After MPI_BCAST returns, all processes have the same value in value.

## Reduction Operations

Another type of common collective operation is reductions. Predefined and user-defined operations can be applied to data as it is combined to form a single answer. A simple program showing a global sum is shown in Listing Two.

MPI_SUM is a predefined operation that computes the sum of the input buffers provided by all processes. The resulting sum is placed in the output buffer, sum.

Note that just like MPI_BCAST, all processes execute the same collective function - but only the root (MPI_COMM_WORLD rank 0) receives the resulting sum value. On all other processes, the value of the sum variable is unmodified by MPI.

The function MPI_ALLREDUCE operates in the same way as MPI_REDUCE except that all processes receive the answer, not just the root.

You can think of it as an MPI_REDUCE immediately followed by an MPI_BCAST (although, for optimization reasons, it may not be implemented that way).

MPI has several other pre-defined operations, including (but not limited to): maximum, minimum, product, logical and bit-wise AND, OR, and XOR, and maximum/minimum location (essentially for finding the process rank with the maximum/minimum value)

## Other Collective Operations

MPI has other collective operations that are worth investigating, such as: scatter, gather, all-to-all, and both internal and external scan.

Some of these operations have multiple variants; for example, there is both a rooted gather (where one process receives all the data) and an "allgather" (where all processes receive all the data).

These operations are described in detail in the MPI-1 and MPI-2 standards documents.

**LISTING TWO**

**Simple Reduction MPI Program**

```
void simple_reduction(void) {
  int rank, sum;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Reduce(&rank,&sum,1,MPI_INT,MPI_
    SUM,0,MPI_COMM_WORLD);
  if (rank == 0)
   printf("Sum of rank values: %d\n",sum);
}
```

### Will That Collective Block?

As mentioned earlier in the column, the only collective that guarantees to block is MPI_BARRIER. All other collectives are defined to block **only until their portion of the collective is complete.** In some cases — depending on how the particular collective algorithm is implemented — this may be immediately. In other cases, processes may block, but for varying amounts of time.

Consider MPI_GATHER — an operation where every process sends its buffer to the root. As soon as each process sends its buffer, it can return. In this scenario, the return from MPI_GATHER on non-root ranks does not imply anything about the completion of MPI_GATHER on any other process in the communicator. The only thing that is known is that the root process will be the last one to complete.

## Where To Go From Here?

The short version of the column is: MPI collectives are your friends. Use them.

Don't code up your own collective algorithms unless you really need to. If the collectives in your MPI implementation perform poorly, write to your Congressman.

Communicators were mentioned frequently this month; next month, we'll discuss them in detail along with their partner in crime: MPI groups.

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