# MPI Mechanic

April 2004

ClusterWorld™

REDEFINING HIGH PERFORMANCE COMPUTING

www.clusterworld.com

# Everything You Wanted to Know About Groups and Communicators

Actual e-mail spams seen recently:

- Banned MPI CD! The government doesn't want you to see this!
- Our licensed MPI programmers will prescribe parallel applications for free.
- Enlarge your parallel application performance by 5x with MPIagra!
- Nigerian bank director needs MPI developers to receive US$25M in offshore funding.

Am I the only one that gets these?

## The Story So Far

We've been progressively leading up to more complicated topics in this column — starting with the 6 basic functions of MPI, moving on to the differences between ranks, processors, and processes, detailing what `MPI_INIT` and `MPI_FINALIZE` really mean. Last month, we discussed collective communication. But wait — if you call in the next 10 minutes, there's even more! (Sorry — spam flashbacks.)

Groups and communicators can play a critical role in the selection of parallel algorithms that you use in your application. Although parallel applications can be implemented in different ways, MPI provides a rich set of process grouping features that are frequently under-utilized in user applications (particularly with respect to collective communications).

## MPI Groups and Communicators

I've made references to "communicators" in previous editions of this column and usually made cryptic statements about "fixed sets," "ordered processes," and "communication contexts." But I've never really explained what a communicator *is*. It's important to understand communicators and what they mean to your application because communicators are the basis for all MPI point-to-point and collective communication.

Communicators comprise two elements: a group and a unique communications context (actually, it may be *two* groups — more on that below). Let's discuss groups first.

## MPI Groups

An MPI group is a fixed, ordered set of unique MPI processes. The exact definition of an MPI process was discussed in the Jan 2003 edition of this column. Essentially, the MPI implementation is free to define what "MPI process" means. Examples include: a thread, a POSIX process, or a Windows process. Although most MPI implementations use the operating system's concept of a "process," some do define threads as an MPI process. A process can appear at most exactly once in a group — it is either in the group or not; a process is never in a group more than once. A process can be in multiple groups, however.

More specifically, an MPI group is a local representation of a set of MPI processes. MPI groups are represented by the opaque type `MPI_Group` in C applications. Hence, a process can contain local representations of many MPI groups — some of which may not include the process itself.

MPI defines a rich set of operations on groups; since a group is essentially an ordered set (in the algebraic sense of the word), an application can perform group unions, intersections, inclusions, exclusions, comparisons, and so on. These operations, while not commonly invoked in many user applications, form the backbone of communicator functionality and may be used by the MPI implementation itself.

As an example, one of the group operations provided by MPI is the comparison of two groups (`MPI_GROUP_COMPARE`), which can yield one of three results:

- `MPI_IDENT`: The two groups contain the same set of processes in the same order
- `MPI_SIMILAR`: The two groups contain the same set of processes, but in a different order
- `MPI_UNEQUAL`: The two groups do not contain the same set of processes.

While seemingly an unimportant operation, it provides insight into one of MPI's central philosophies: the membership in a group is fixed and strongly ordered. This feature is most apparent to users because *communicators* have fixed, ordered memberships. But this is only a byproduct of the fact that a communicator contains a group.

## MPI Communicators

Communicators are represented in MPI C programs by the type `MPI_Comm` (Fortran programs use integers). Although communicator is a local MPI object (i.e., it physically resides in the MPI process), it represents a process' membership in a larger process group. Specifically, even though MPI_Comm objects are local, they are always created collectively between all members in the group that the communicator contains. Hence, a process can only have an MPI_Comm handle for communicators of which it is a member.

The context of a communicator is effectively a guarantee that a message sent on one communicator will never be received on a different communicator. Consider the arguments of the MPI_SEND and MPI_RECV functions (C binding shown below):

```
int MPI_Send(
  void *buf,
  int count,
  MPI_Datatype dtype,
  int dest,
  int tag,
  MPI_Comm comm
);
int MPI_Recv(
  void *buf,
  int count,
  MPI_Datatype dtype,
  int src,
  int tag,
  MPI_Comm comm,
  MPI_Status *status
)
```

A sent message will only be delivered to a matching receive in the destination process. This means that the MPI_SEND has to use the triple (dest, tag, comm) that specifies a peer process in the communicator who has posted a receive with a corresponding (src, tag, comm) triple. The src and dest values must be equal, or the receiver can use the wildcard MPI_ANY_TAG; the tag values must be equal, or the receiver can use the wildcard MPI_ANY_TAG; the comm values must represent the same communicator.

Note the last part — the comm values must represent the same communicator; there is no wildcard communicator value. The communicator therefore functions similarly to the "tag" argument in MPI_SEND (and friends) — think of it as a system-level tag. Specifically, a message sent on a given (tag, communicator) tuple will only ever be received on the same (tag, communicator) tuple by the receiver (with the MPI_ANY_TAG exception).

### Communicators: What's the point?

So what's all this hoopla about communicators? Why bother? Why not just send and receive messages, filtering them via tags?

One answer is parallel libraries. Libraries that use message passing need to have a way to guarantee that the messages they send and receive will never be confused with messages sent and received by the user application. Communicators, with their unique (and private) communication context, allow this message passing safety.

Many parallel libraries, for example, use the MPI_COMM_DUP call at startup time to duplicate MPI_COMM_WORLD — the pre-defined communicator created after MPI_INIT that contains all processes that were started together. The new communicator will have exactly the same process group, but a different (unique) context than MPI_COMM_WORLD. The library can then use this communicator for all of its communications.

## Communicator Properties

Remember that a communicator *contains* a group, and a group is a strongly ordered set of processes. Therefore, communicators are also strongly ordered sets of processes. More importantly, the order is guaranteed to be the same on all processes in the communicator (group). Hence, the process referred to by (index, communicator) is guaranteed to be the same on all processes in the communicator.

The "index" value ranges from zero to the number of processes in the communicator minus 1. This index value is called the process' "rank" in the communicator. Hence, MPI point-to-point communication routines (e.g., MPI_SEND and MPI_RECV) are expressed in terms of ranks and communicators — the source or destination of the message.

Don't get carried away with the term "rank," however. A rank refers to a specific process *in 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."

There are actually two kinds of communicators: *intra*communicators (those that only contain one group of processes) and *inter*communicators (those that contain two groups of processes). Let's talk about the most common kind first, *intra*communicators (one group).

## Intracommunicators

The name "intracommunicator" specifically refers to communication within a single group. MPI_COMM_WORLD is perhaps the most famous of intracommunicators. It is defined in the MPI-1 standard as "all processes the local process can communicate with after initialization (including itself), and is defined once MPI_INIT has been called." Although the specific meaning of this statement varies between different MPI implementations, it generally means that all MPI processes started via `mpirun` will be included in `MPI_COMM_WORLD` together.

Another, lesser-known pre-defined intracommunicator is `MPI_COMM_SELF`, which is defined to only include the local process. This communicator can be useful for loopback kinds of communicators, depending on the application's algorithms.

## Intercommunicators

Intercommunicators refers to communication between two groups in a single communicator — a local group and a remote group. When using an intercommunicator, the initiating process is defined to be in the local group and the target process is defined to be in the remote group. For example, a process invoking `MPI_SEND` on an intercommunicator is in the `comm`'s local group, but the `dest` rank argument is relative to the remote group. Similarly, a process invoking `MPI_RECV` on an intercommunicator is in the comm's local group, but the `src` rank argument is relative to the remote group.

Intercommunicators are somewhat of an advanced topic; I'll return to them a few months from now.

## Topologies

MPI also contains a full set of topology-based communicators utilizing on N-dimensional Cartesian shapes as well as arbitrary graphs. The rationale is that if the communicator (and therefore the MPI implementation) is aware of the underlying network topology, the user application can effectively "Send this message to the peer on my right," and the MPI application can both figure out who the peer "on my right" is as well as potentially optimize the message routing.

Essentially, the MPI implementation is free to define what "MPI process" means

My own opinion is that topologies are the best kept secret in MPI. They are rarely utilized by real user applications for two reasons:

1. The setup function calls are somewhat bulky and inconvenient
2. Since no users use them, few MPI implementations have optimized them, and there is little performance gain realized

It's a vicious circle — no one uses them because they aren't optimized, and most MPI implementors don't optimize them because no one uses them.

## Operations on Communicators

Now that you know what communicators are, what can you do with them?

Perhaps the most common two operations invoked on communicators are functions that have been mentioned in previous editions of this column: `MPI_COMM_RANK` and `MPI_COMM_SIZE`. The former returns the rank of the calling process in the communicator; the latter returns the total size of the local group in the communicator. These two functions are critical for identity-based algorithms.

`MPI_COMM_DUP` (mentioned in the "Communicators - what's the point?" sidebar) takes a single communicator as input and creates a new communicator as output. The new communicator has exactly the same process membership and order (i.e., the groups in the two communicators are `MPI_IDENT`), but they have different communication contexts. `MPI_COMM_DUP` is a collective call — all processes in the input communicator must call `MPI_COMM_DUP` before it will return.

Another common communicator operation is `MPI_COMM_SPLIT` (also a collective call). This operation takes an input communicator and splits it into sub-communicators:

```
int MPI_Comm_split(
  MPI_Comm comm,
  int color,
  int key,
  MPI_Comm *newcomm
);
```

Each calling process will provide the same input `comm` argument. Processes that use the same `color` will end up in a new communicator together (`newcomm`). The special color `MPI_UNDEFINED` can also be used, in which case the calling process will receive `MPI_COMM_NULL` in `newcomm` (i.e., it won't be part of any new communicators). The `key` argument is used for relative ordering in each process' respective `newcomm`.

`MPI_COMM_SPLIT` is useful to partition groups of processes into sub-communicators for specific purposes. Here's a simple example:

```
1 int rank;
2 MPI_Comm row, col;
3 MPI_Comm_rank(
    MPI_COMM_WORLD,
    &rank
  );
4 MPI_Comm_split(
    MPI_COMM_WORLD,
    rank / 4,
    0,
    &row
```

*See MPI, page xx*

*MPI, from page xx*

```
   );
   MPI_Comm_split(
     MPI_COMM_WORLD,
     rank % 4,
     0,
     &col
   );
```

Assuming that this program was invoked with 16 processes (i.e., a two-dimensional 4x4 grid), the calling process will end up being in two new communicators: `row`, which contains all processes in the same row as this process, and `col`, which contains all the processes in the same column as this process.

## Where To Go From Here?

As with MPI collectives, MPI communicators are your friends. Use `MPI_DUP` to create safe communication contexts in different parts of

My own opinion is that topologies are the best kept secret in MPI. They are rarely utilized by real user applications

your application, and use the subsetting capabilities of `MPI_COMM_SPLIT` instead of creating arbitrarily complicated grouping schemes with additional arrays, pointers, or lookup tables.

Next month: datatypes! We've talked about sending messages and shown simple examples of involving a single integer or character - real application need to send much more complex data.

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

## Resources

**MPI Forum**

- http://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