Message-Passing and MPI Programming

Communicators etc.

N.M. Maclaren

nmm1@cam.ac.uk

July 2010

7.1 Basic Concepts

A group is a set of process identifiers; programs view them as integers in the range `0...(size-1)`, where `size` is the number of processes in the communicator.

A context is the communication environment, including information about the number of processes and their locations; separate contexts are entirely independent. Programs do not and cannot view contexts directly.

A communicator is a group plus a context, so separate communicators are independent, even if they have the same group of processes. Normally, we work solely on communicators.

There are several predefined communicators, and you should use these when appropriate:

- `MPI_COMM_WORLD` is all processes together.
- `MPI_COMM_SELF` is just the local process.
- `MPI_COMM_NULL` is an invalid communicator, and is used as an error result from several functions.

Most people use only `MPI_COMM_WORLD`, and we covered information calls in the first lecture, including `MPI_Comm_rank` and `MPI_Comm_size`. Why do we need to go beyond that?

- You need to use collectives on only some processes
- You need to do a task on only some processes
- You want to do several tasks in parallel

We can do those messily by using point-to-point, or by creating new, subset communicators. This lecture describes how to do the latter.

7.2 Use of Communicators

Despite their independence, you should avoid using two communicators that overlap, including using one together with a subset of itself. Clean up the activities on before starting the other. MPI will not get confused, but you and I will – and do not even think of trying to tune such a mess!

You should design your communicator use to be hierarchical, exactly like recursion in groups of processes. This is easier to show using figures than to describe.

The first figure shows a general set of communicators, which overlap in unstructured ways; this is not advised.

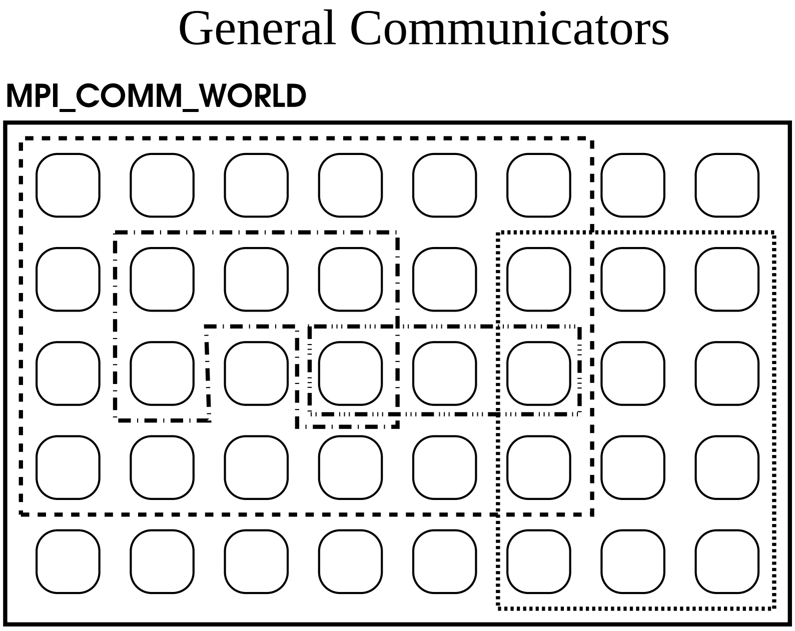

Figure 7.1

The second figure shows a hierarchical set of communicators, where each communicator is a subset of its parent, and two sibling communicators (i.e. children of the same parent) do not overlap.

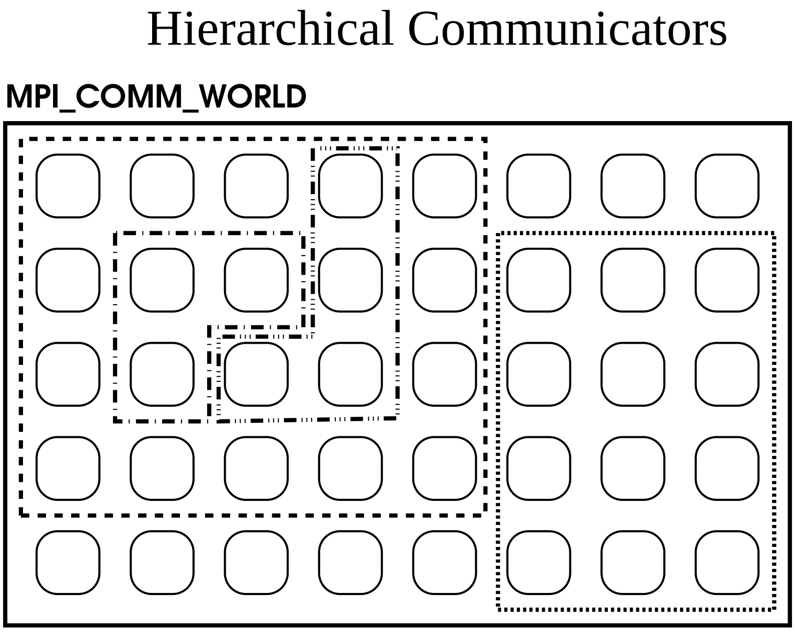

Figure 7.2

The third figure shows how hierarchical communicators can be used. Note that it alternates between the use of a parent communicator and the use of its children. The restriction against doing that is solely against **using** them in parallel – overlapping inactive communicators are not a problem.

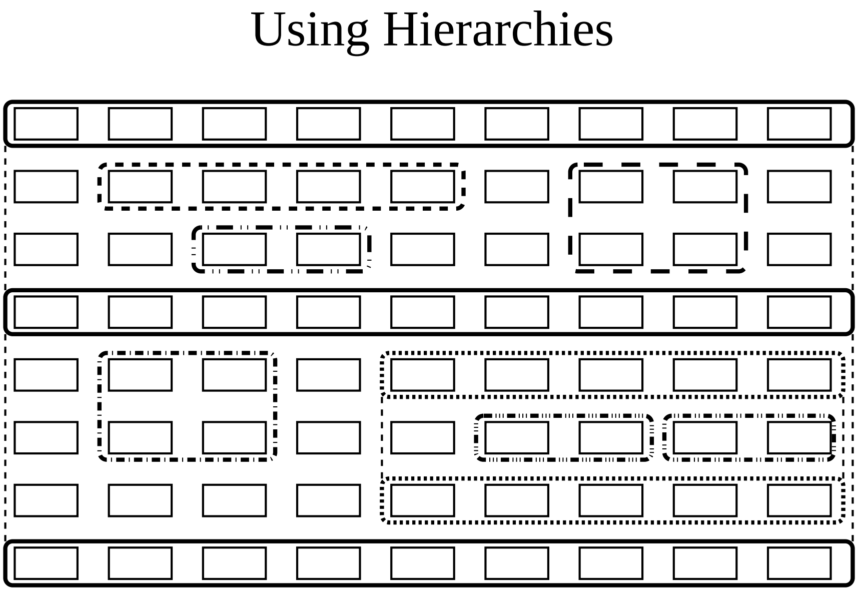

Figure 7.3

7.3 Splitting Communicators

You always start with an existing communicator and subdivide it to make one or more new ones. Doing this is a collective operation on the existing communicator.

In the call we shall use, each process specifies a non-negative integer; its value is commonly called the colour. Each new communicator corresponds to one colour; e.g. all processes that specify the integer 42 belong to the same child communicator. If two processes specify different colours, the call returns different communicators (i.e. the ones they belong to); in computer science terms, a communicator is a value not an identifier.

You can also specify `MPI_UNDEFINED` to opt out of any child communicator. That is an unspecified negative integer – note that zero is a valid colour, and therefore cannot be used for the undefined value. In this case, the call will return `MPI_COMM_NULL`, which is an invalid communicator, so do not use it.

Splitting Communicators

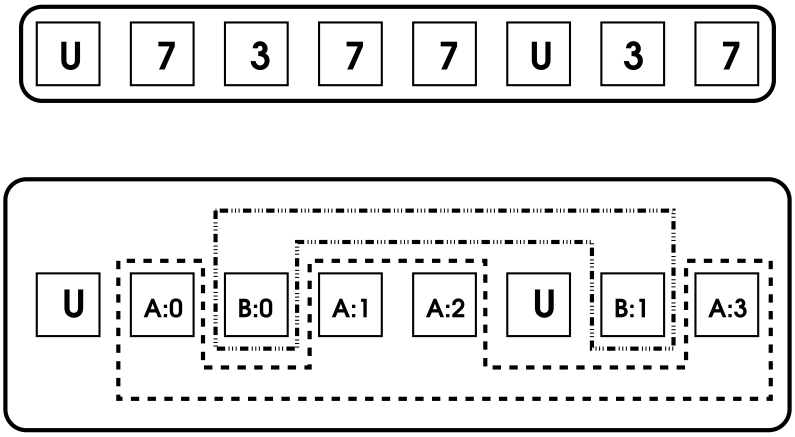

Figure 7.4

You can also set the rank in the new communicator. You provide a key argument that has an integer value. In this case, any values are allowed, even negative ones, and the processes have ranks in the new communicator in the same order as the key values. Setting all keys to zero says you do not care, and I recommend doing just that, because it is one less detail to worry about. Doing anything else with keys is advanced use, comparable to operating on groups directly (see later); the examples will use keys of zero.

When you have finished with a communicator, you should free (i.e. delete or destroy) it. That is a collective call on the new communicator, and will free any resources it uses. But you **must** tidy up all transfers first; some libraries and tools may check that has been done, but others may fail in horrible ways if there are any outstanding transfers. You need not free it if you only stop using it (i.e. when you are going to reuse it later).

Fortran:

```
INTEGER :: colour , newcomm , error
! 'colour' is set to an appropriate value

CALL MPI_Comm_split ( MPI_COMM_WORLD ,    &
    colour , 0 , newcomm , error )
IF ( newcomm /= MPI_COMM_NULL ) THEN
    CALL My_collective ( newcomm , ... )
    CALL MPI_Comm_free ( newcomm , error )
END IF
```

C:

```
int colour , error ;
/* 'colour' is set to an appropriate value */
MPI_Comm newcomm ;

error = MPI_Comm_split ( MPI_COMM_WORLD ,
    colour , 0 , & newcomm ) ;
if ( newcomm != MPI_COMM_NULL ) {
    My_collective ( newcomm , ... ) ;
    error = MPI_Comm_free ( newcomm ) ;
}
```

7.4 More Complex Uses

You can obviously do the above recursively. All you need is to change `MPI_COMM_WORLD` to `newcomm` and `newcomm` to `evennewercomm`. No example is given of doing this.

To repeat, do not **use** overlapping communicators; inactive communicators are not a problem. All you need to do is to tidy up all transfers before proceeding, though it is a good idea to use barriers for tuning reasons. This is a very simple C example of splitting collectives:

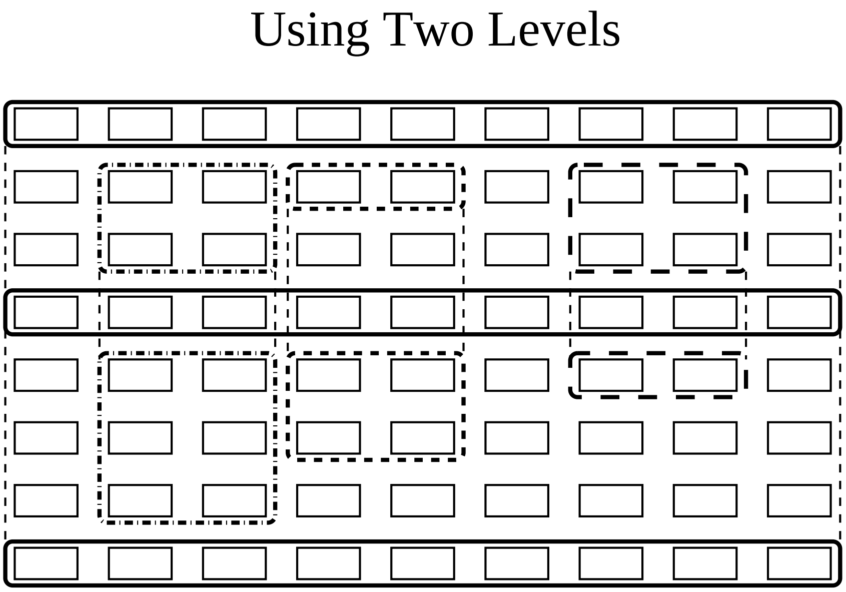

Figure 7.5

Note that `newcomm` is actually three communicators in the figure; they cannot overlap, so the above use is safe. Yes, that is parallel use of collectives.

```
errno = My_global_collective ( MPI_COMM_WORLD ) ;
errno = Split ( MPI_COMM_WORLD , colour , 0 , & newcomm ) ;
if ( newcomm != MPI_COMM_NULL )
    errno = My_split_collective ( newcomm , ... ) ;
errno = My_global_collective ( MPI_COMM_WORLD ) ;
if ( newcomm != MPI_COMM_NULL )
    errno = My_split_collective ( newcomm , ... ) ;
errno = My_global_collective ( MPI_COMM_WORLD ) ;
```

The following is the first half of the above example, with some barriers added. There is no difference in behaviour, but this is probably easier to tune, and possibly faster. You should note which communicator they are used with!

```
errno = My_global_collective ( MPI_COMM_WORLD ) ;
errno = Split ( MPI_COMM_WORLD , colour , 0 , & newcomm ) ;
if ( newcomm != MPI_COMM_NULL ) {
    errno = My_split_collective ( newcomm , ... ) ;
    errno = Barrier ( newcomm ) ;
}
errno = Barrier ( MPI_COMM_WORLD ) ;
errno = My_global_collective ( MPI_COMM_WORLD ) ;
```

The error handler is inherited. You can change that subsequently, but it is difficult to imagine many people wanting to. If you want to set the error handler simply, you should set it before creating any sub-communicators (obviously on `MPI_COMM_WORLD`). It will then be inherited to all of the communicators you create.

You can make an exact copy of a communicator, which is then completely independent of the first one; the function is `MPI_Comm_dup`. This could be useful to bypass implementation bugs though I shall mention another possible use later, under I/O. But, in general, very few people will want to do this. *FFTW* and *SPOOLES* do use it, but it looks as if they may have misunderstood the MPI specification, though it could have been to fix up some broken implementation.

Topologies are how the processes are connected. MPI has some virtual topology facilities that map the program's structure, which are independent of the actual hardware network. There is another lecture on Cartesian topologies, which may clarify code that uses an N-dimensional grid (see later under "*Problem Decomposition*"); that use is simple but is omitted for brevity. However, topologies are almost essential if:

- you are writing structure-generic libraries, or
- your program has a variable graph structure.

That is more-or-less all you need to know. You can add names to communicators, which might improve your diagnostics considerably (or might not); look up `MPI_Comm_get_name` and `MPI_Comm_set_name` if you want to do that. And there is one other function, for comparing communicators, useful for advanced use only: `MPI_Comm_compare`.

7.5 Groups

There are facilities for operating on groups, which are not often used, though I have and *CPMD* does. Because of that, here is just a very brief summary in case you have a program that uses them.

Operations on groups are entirely local; they are just operating on sets of integers, after all. For cleanliness, MPI hides them behind a handle, which is the opaque type `MPI_Group` in C, and you should use only the facilities that MPI provides. Groups take effect only when you create a communicator. There is an alternative way of creating subset communicators by greating a group and then creating a communicator using that group:

`MPI_Comm_group` gets the current group; i.e. it extracts it from the communicator.

`MPI_Group_incl` creates a subset group ; you pass it the ranks you want to keep.

`MPI_Comm_create` makes a new communicator using the new subset group.

`MPI_Group_free` releases the groups (i.e. the opaque type); using this is highly desirable to avoid resource leaks.

`MPI_Comm_free` is used to free the communicator (as before).

You are **strongly** advised to program those collectively – i.e. to do identical group calculations on all processes. This is not because MPI needs that (it does not), but to avoid errors. There are only two actual collectives, `MPI_Comm_create` and `MPI_Comm_free`, but group membership in all processes must match when you call the former. You may find using groups easier than using `MPI_Comm_split`; they are functionally equivalent.

Other group functions include `MPI_Group_compare`, `MPI_Group_difference`, `MPI_Group_excl`, `MPI_Group_intersection`, `MPI_Group_range_excl`, `MPI_Group_range_incl`, `MPI_Group_rank`, `MPI_Group_size`, `MPI_Group_translate_ranks` and `MPI_Group_union`. Many of them are alternatives to `MPI_Group_incl`, and you probably will never want to use the ones that are not.

7.6 Epilogue

You now know what you can do with communicators, though most of you will use only `MPI_COMM_WORLD`. There is one simple exercise using `MPI_Comm_split`.