

`MPI::Comm::Clone()` The C++ language interface for MPI includes a new function `Clone()`. `MPI::Comm::Clone()` is a pure virtual function. For the derived communicator classes, `Clone()` behaves like `Dup()` except that it returns a new object by reference. The `Clone()` functions are prototyped as follows:

```
Comm& Comm::Clone() const = 0
Intracomm& Intracomm::Clone() const
Intercomm& Intercomm::Clone() const
Cartcomm& Cartcomm::Clone() const
Graphcomm& Graphcomm::Clone() const
```

Rationale. `Clone()` provides the “virtual dup” functionality that is expected by C++ programmers and library writers. Since `Clone()` returns a new object by reference, users are responsible for eventually deleting the object. A new name is introduced rather than changing the functionality of `Dup()`. (*End of rationale.*)

Advice to implementors. Within their class declarations, prototypes for `Clone()` and `Dup()` would look like the following:

```
namespace MPI {
  class Comm {
    virtual Comm& Clone() const = 0;
  };
  class Intracomm : public Comm {
    Intracomm Dup() const { ... };
    virtual Intracomm& Clone() const { ... };
  };
  class Intercomm : public Comm {
    Intercomm Dup() const { ... };
    virtual Intercomm& Clone() const { ... };
  };
  // Cartcomm and Graphcomm are similarly defined
};
```

(*End of advice to implementors.*)

16.1.8 Exceptions

The C++ language interface for MPI includes the predefined error handler `MPI::ERRORS_THROW_EXCEPTIONS` for use with the `Set_errhandler()` member functions. `MPI::ERRORS_THROW_EXCEPTIONS` can only be set or retrieved by C++ functions. If a non-C++ program causes an error that invokes the `MPI::ERRORS_THROW_EXCEPTIONS` error handler, the exception will pass up the calling stack until C++ code can catch it. If there is no C++ code to catch it, the behavior is undefined. In a multi-threaded environment or if a nonblocking MPI call throws an exception while making progress in the background, the behavior is implementation dependent.

location of data by providing memory addresses and offsets. In the C language, sequence association rules plus pointers provide all the necessary low-level structure.

In Fortran 90, user data is not necessarily stored contiguously. For example, the array section `A(1:N:2)` involves only the elements of `A` with indices 1, 3, 5, The same is true for a pointer array whose target is such a section. Most compilers ensure that an array that is a dummy argument is held in contiguous memory if it is declared with an explicit shape (e.g., `B(N)`) or is of assumed size (e.g., `B(*)`). If necessary, they do this by making a copy of the array into contiguous memory. Both Fortran 77 and Fortran 90 are carefully worded to allow such copying to occur, but few Fortran 77 compilers do it.¹

Because MPI dummy buffer arguments are assumed-size arrays, this leads to a serious problem for a nonblocking call: the compiler copies the temporary array back on return but MPI continues to copy data to the memory that held it. For example, consider the following code fragment:

```
real a(100)
call MPI_IRECV(a(1:100:2), MPI_REAL, 50, ...)
```

Since the first dummy argument to `MPI_IRECV` is an assumed-size array (`<type> buf(*)`), the array section `a(1:100:2)` is copied to a temporary before being passed to `MPI_IRECV`, so that it is contiguous in memory. `MPI_IRECV` returns immediately, and data is copied from the temporary back into the array `a`. Sometime later, MPI may write to the address of the deallocated temporary. Copying is also a problem for `MPI_ISEND` since the temporary array may be deallocated before the data has all been sent from it.

Most Fortran 90 compilers do not make a copy if the actual argument is the whole of an explicit-shape or assumed-size array or is a ‘simple’ section such as `A(1:N)` of such an array. (We define ‘simple’ more fully in the next paragraph.) Also, many compilers treat allocatable arrays the same as they treat explicit-shape arrays in this regard (though we know of one that does not). However, the same is not true for assumed-shape and pointer arrays; since they may be discontinuous, copying is often done. It is this copying that causes problems for MPI as described in the previous paragraph.

Our formal definition of a ‘simple’ array section is

```
name ( [:,]... [<subscript>]:<subscript> [,<subscript>]... )
```

That is, there are zero or more dimensions that are selected in full, then one dimension selected without a stride, then zero or more dimensions that are selected with a simple subscript. Examples are

```
A(1:N), A(:,N), A(:,1:N,1), A(1:6,N), A(:, :, 1:N)
```

Because of Fortran’s column-major ordering, where the first index varies fastest, a simple section of a contiguous array will also be contiguous.²

The same problem can occur with a scalar argument. Some compilers, even for Fortran 77, make a copy of some scalar dummy arguments within a called procedure. That this can cause a problem is illustrated by the example

¹Technically, the Fortran standards are worded to allow non-contiguous storage of any array data.

²To keep the definition of ‘simple’ simple, we have chosen to require all but one of the section subscripts to be without bounds. A colon without bounds makes it obvious both to the compiler and to the reader that the whole of the dimension is selected. It would have been possible to allow cases where the whole dimension is selected with one or two bounds, but this means for the reader that the array declaration or most recent allocation has to be consulted and for the compiler that a run-time check may be required.

```

call user1(a,rq)
call MPI_WAIT(rq,status,ierr)
write (*,*) a

subroutine user1(buf,request)
call MPI_Irecv(buf,...,request,...)
end

```

If `a` is copied, `MPI_Irecv` will alter the copy when it completes the communication and will not alter `a` itself.

Note that copying will almost certainly occur for an argument that is a non-trivial expression (one with at least one operator or function call), a section that does not select a contiguous part of its parent (e.g., `A(1:n:2)`), a pointer whose target is such a section, or an assumed-shape array that is (directly or indirectly) associated with such a section.

If there is a compiler option that inhibits copying of arguments, in either the calling or called procedure, this should be employed.

If a compiler makes copies in the calling procedure of arguments that are explicit-shape or assumed-size arrays, simple array sections of such arrays, or scalars, and if there is no compiler option to inhibit this, then the compiler cannot be used for applications that use `MPI_GET_ADDRESS`, or any nonblocking MPI routine. If a compiler copies scalar arguments in the called procedure and there is no compiler option to inhibit this, then this compiler cannot be used for applications that use memory references across subroutine calls as in the example above.

Special Constants

MPI requires a number of special “constants” that cannot be implemented as normal Fortran constants, e.g., `MPI_BOTTOM`. The complete list can be found in Section 2.5.4 on page 14. In C, these are implemented as constant pointers, usually as `NULL` and are used where the function prototype calls for a pointer to a variable, not the variable itself.

In Fortran the implementation of these special constants may require the use of language constructs that are outside the Fortran standard. Using special values for the constants (e.g., by defining them through `parameter` statements) is not possible because an implementation cannot distinguish these values from legal data. Typically these constants are implemented as predefined static variables (e.g., a variable in an MPI-declared `COMMON` block), relying on the fact that the target compiler passes data by address. Inside the subroutine, this address can be extracted by some mechanism outside the Fortran standard (e.g., by Fortran extensions or by implementing the function in C).

Fortran 90 Derived Types

MPI does not explicitly support passing Fortran 90 derived types to choice dummy arguments. Indeed, for MPI implementations that provide explicit interfaces through the `mpi` module a compiler will reject derived type actual arguments at compile time. Even when no explicit interfaces are given, users should be aware that Fortran 90 provides no guarantee of sequence association for derived types or arrays of derived types. For instance, an array of a derived type consisting of two elements may be implemented as an array of the first elements followed by an array of the second. Use of the `SEQUENCE` attribute may help here, somewhat.

- The compiler may be prevented from moving a reference to a buffer across a call to an MPI subroutine by surrounding the call by calls to an external subroutine with the buffer as an actual argument. Note that if the intent is declared in the external subroutine, it must be `OUT` or `INOUT`. The subroutine itself may have an empty body, but the compiler does not know this and has to assume that the buffer may be altered. For example, the above call of `MPI_RECV` might be replaced by

```

      call DD(buf)
      call MPI_RECV(MPI_BOTTOM,...)
      call DD(buf)

```

with the separately compiled

```

      subroutine DD(buf)
         integer buf
      end

```

(assuming that `buf` has type `INTEGER`). The compiler may be similarly prevented from moving a reference to a variable across a call to an MPI subroutine.

In the case of a nonblocking call, as in the above call of `MPI_WAIT`, no reference to the buffer is permitted until it has been verified that the transfer has been completed. Therefore, in this case, the extra call ahead of the MPI call is not necessary, i.e., the call of `MPI_WAIT` in the example might be replaced by

```

      call MPI_WAIT(req,...)
      call DD(buf)

```

- An alternative is to put the buffer or variable into a module or a common block and access it through a `USE` or `COMMON` statement in each scope where it is referenced, defined or appears as an actual argument in a call to an MPI routine. The compiler will then have to assume that the MPI procedure (`MPI_RECV` in the above example) may alter the buffer or variable, provided that the compiler cannot analyze that the MPI procedure does not reference the module or common block.

The **VOLATILE** attribute, available in later versions of Fortran, gives the buffer or variable the properties needed, but it may inhibit optimization of any code containing the buffer or variable.

In C, subroutines which modify variables that are not in the argument list will not cause register optimization problems. This is because taking pointers to storage objects by using the `&` operator and later referencing the objects by way of the pointer is an integral part of the language. A C compiler understands the implications, so that the problem should not occur, in general. However, some compilers do offer optional aggressive optimization levels which may not be safe.