Source: http://beige.ucs.indiana.edu/I590/node85.html by Zdzislaw Meglicki

Handling MPI Errors

An MPI *communicator* is more than just a group of processes that belong to it. The latter is simply a group. But communications do not take place within the group. They take place within the communicator, because one needs more for a communication than just a list of participating processes. Amongst the items that the communicator hides inside its bulbous body is an *error handler*. The error handler is called every time an MPI error is detected within the communicator.

The predefined default error handler, which is called `MPI_ERRORS_ARE_FATAL`, for a newly created communicator or for `MPI_COMM_WORLD` is to *abort the whole parallel program* as soon as any MPI error is detected. Whether an error message is printed or not, and what the error message is, depends on the implementation.

There is another predefined error handler, which is called `MPI_ERRORS_RETURN`. The default error handler can be replaced with this one by calling function MPI_Errhandler_set, for example:

```
MPI_Errhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
```

Once you've done this in your MPI code, the program will not longer abort on having detected an MPI error, instead the error will be returned and you will have to handle it.

The returned error code is implementation specific. The only error code that MPI standard itself defines is `MPI_SUCCESS`, i.e., no error. But the meaning of an error code can be extracted by calling function MPI_Error_string.

For example, consider the following code fragment:

```
MPI_Errhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
error_code = MPI_Send(send_buffer, strlen(send_buffer) + 1, MPI_CHAR,
                      addressee, tag, MPI_COMM_WORLD);
if (error_code != MPI_SUCCESS) {

   char error_string[BUFSIZ];
   int length_of_error_string;

   MPI_Error_string(error_code, error_string, &length_of_error_string);
   fprintf(stderr, "%3d: %s\n", my_rank, error_string);
   send_error = TRUE;
}
```

On top of the above MPI standard defines the so called *error classes*. Every error code, even the one that is implementation specific, which is every one with the exception of `MPI_SUCCESS`, must belong to some error class, and the error class for a given error code can be obtained by calling function MPI_Error_class. Error classes can be converted to comprehensible error messages by calling the same function that does it for error codes, i.e., `MPI_Error_string`. The reason for this is that error classes are implemented as a subset of error codes. Here is the example:

```
MPI_Errhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
error_code = MPI_Send(send_buffer, strlen(send_buffer) + 1, MPI_CHAR,
                      addressee, tag, MPI_COMM_WORLD);
if (error_code != MPI_SUCCESS) {

   char error_string[BUFSIZ];
   int length_of_error_string, error_class;

   MPI_Error_class(error_code, &error_class);
   MPI_Error_string(error_class, error_string, &length_of_error_string);
   fprintf(stderr, "%3d: %s\n", my_rank, error_string);
   MPI_Error_string(error_code, error_string, &length_of_error_string);
   fprintf(stderr, "%3d: %s\n", my_rank, error_string);
   send_error = TRUE;
}
```

The idea here is that the error class should give you a general description of the problem, yet it should be precise enough for most debugging purposes, and the error code can then give you an even more precise, implementation specific, diagnostic.

If you have found an MPI error like this in your code, it may be very difficult to recover gracefully. Other than printing the message on standard error and then exiting, or, at best, going right to `MPI_Finalize`, there isn't much that you can do. Sometimes if the problem is, e.g., a receive buffer that is too small, you may be able to allocate a larger buffer dynamically. Your program has to anticipate such events though, and if it does, there are other means of finding how large a buffer you need and avoiding the error altogether.

Perhaps the best use of activating the non-aborting error handler is when you debug the program and try to find where exactly it fails.

Once you have detected the error and are desperate to exit in a controllable way, you can call MPI function MPI_Abort, for example:

```
MPI_Errhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
error_code = MPI_Send(send_buffer, strlen(send_buffer) + 1, MPI_CHAR,
                      addressee, tag, MPI_COMM_WORLD);
if (error_code != MPI_SUCCESS) {

   char error_string[BUFSIZ];
   int length_of_error_string, error_class;

   MPI_Error_class(error_code, &error_class);
   MPI_Error_string(error_class, error_string, &length_of_error_string);
   fprintf(stderr, "%3d: %s\n", my_rank, error_string);
   MPI_Error_string(error_code, error_string, &length_of_error_string);
   fprintf(stderr, "%3d: %s\n", my_rank, error_string);
   MPI_Abort(MPI_COMM_WORLD, error_code);
}
```

Each MPI file, which is always associated with a communicator and about which we are going to learn in the next section, has its own separate file handler, which can be altered with the call to function MPI_File_set_errhandler. The predefined values for an MPI file error handler are the same as the values for an MPI communicator error handler, i.e., `MPI_ERRORS_ARE_FATAL` and `MPI_ERRORS_RETURN`.

However, since file manipulation errors are very common, in this case `MPI_ERRORS_RETURN` is the default.

Apart from communicators and files MPI also supports the so called *windows*. These are *windows of existing memory* that each process *exposes* to direct memory accesses by processes within the communicator. Like MPI files, MPI windows are also associated with MPI communicators. Each MPI window has its own error handler associated with it too and these can be altered by calling function MPI_Win_set_errhandler. The predefined values for the windows error handlers are the same as for communicators and files.