MPIFX

Release 1.4.0

B. Aradi

Jun 30, 2023

Contents

Chapter 1

About MPIFX

MPIFX is a library containing modern Fortran (Fortran 2003) wrappers around MPI routines. The goal is to make the use of MPI as simple as possible in Fortran.

Consider for example a simple MPI broadcast. In order to broadcast an integer array with 25 elements using the legacy MPI routine, you have to issue:

```
call mpi_bcast(myarray, 25, MPI_INTEGER, 0, MPI_COMM_WORLD, error)
```

Additional to the object to be broadcasted and the communicator, you also *must* specify following arguments:

- type of the array (which is redundant, as it is *known* at compile-time)
- size of the array (which is redundant, as it is *known* at run-time)
- root node of the broadcast (setting it to the lead node as default would be a definitely safe choice)
- error flag (one could per default just omit it and rely on the program to stop if a problem arised, similar as done in Fortran for allocations)

Using MPIFX the call above is as simple as:

```
call mpifx_bcast(comm, myarray)
```

No redundant arguments, sensible defaults. Nevertheless the full functionality is still available via optional parameters if needed. E.g. if you wanted to handle the error flag yourself (making sure an error won't stop your code), you could call:

```
call mpifx_bcast(comm, myarray, error=ierr)
```

A few essential communication routines are already covered (see *List of routines* (page 7)). If your desired MPI-routine is not among them yet, you are cordially invited to extend MPIFX and to share it in order to let others profit from your work (MPIFX is licensed under the simplified BSD license). For more details see the project page.

Chapter 2

Compiling and installing MPIFX

In order to compile MPIFX, you need following prerequisites:

- Fortran 2003 compiler,
- Python (2.6, 2.7 or any 3.x release)
- GNU Make.

There are basically two different ways of using the library in your project:

- *Precompiling the library* (page 3) and linking it later to your project.
- *Compiling the library during your build process* (page 3).

Both are described below.

2.1 Precompiling the library

In order to create a precompiled library

1. Copy the file *make.arch.template* to *make.arch* in the root directory of the source and customize the settings for the compilers and the linker according to your system.
2. Issue *make* to build the library.
3. Issue *make install* to copy the library and the module files to the installation destination.

During the build process of your project, you may link the library with the *-lmpifx* option. Eventually, you may need to specify options for your compiler and your linker to specify the location of those directories. Assuming you've put the module files in the directory *<MODFILEDIR>* and the library file in *<LIBRARYDIR>*, you would typically invoke your compiler for the source files using the *libmpifx_module* as:

```
F2003_COMPILER -I<MODFILEDIR> -c somesource.f90
```

and link your object files at the end with:

```
LINKER -I<LIBRARYDIR> somesource.o ... -L<LIBRARYDIR> -lmpifx
```

2.2 Compiling the library during your build process

In order to build the library during the build process of your project:

1. Copy the content of the *lib/* folder into a *separate* folder within your project.

2. During the make process of your project, invoke the library makefile (*make.build*) to build the module files and the library in the folder where you've put the library sources.

   You must pass the compiler and linker options via variable defintions at the make command line. Assuming that the variables *$(FXX)*, *$(FXXOPT)*, *$(LN)* and *$(LNOPT)*, *$(FYPP)* and *$(FYPPOPT)* contain the Fortran compiler, the Fortran compiler options, the linker, the linker options, the Fypp preprocessor and its options, respectively, you would have something like:

   ```
   libmpifx.a:
           $(MAKE) -C $(MPIFX_BUILDDIR) \
               FXX="$(FXX)" FXXOPT="$(FXXOPT)" \
               LN="$(LN)" LNOPT="$(LNOPT)" \
               FYPP="$(FYPP)" FYPPOPT="$(FYPPOPT)" \
               -f $(MPIFX_SRCDIR)/make.build
   ```

   in the makefile of your project with *$(MPIFX_SRCDIR)* being the directory where you've put the source of MPIFX and *$(MPIFX_BUILDDIR)* where the build of the library should be done.

You should also have a look at the *Umakefile* in the root folder of MPIFX, which uses exactly the same technique to compile the library.

Chapter 3

Using MPIFX

Before you can use the MPIFX routines you need the following steps:

1. Use the module *libmpifx_module* in your routines.
2. Initialize the MPI framework via the *mpifx_init()* routine. (If you already initialized it via the legacy *mpi_init()* call, you should omit this step.
3. Initialize a communicator of *type(mpifx_comm)*.

Below you find a self containing example for reduction on all processes using a wrapper around *mpi_allreduce()*:

```
program test_allreduce
  use libmpifx_module
  implicit none

  integer, parameter :: dp = kind(1.0d0)

  type(mpifx_comm) :: mycomm
  integer :: vali0, resvali0
  real(dp) :: valr(3), resvalr(3)

  call mpifx_init()
  call mycomm%init()

  ! Reduce scalar value
  vali0 = mycomm%rank * 2  ! Some arbitrary number
  write(*, "(I2.2,'-',I3.3,'|',1X,A,I0)") 1, mycomm%rank, &
      & "Value to be operated on:", vali0
  call mpifx_allreduce(mycomm, vali0, resvali0, MPI_SUM)
  write(*, "(I2.2,'-',I3.3,'|',1X,A,I0)") 2, mycomm%rank, &
      & "Obtained result (sum):", resvali0

  ! Reduce vector
  valr(:) = [ real(mycomm%rank + 1, dp) * 1.2, &
      & real(mycomm%rank + 1, dp) * 4.3, real(mycomm%rank + 1, dp) * 3.8 ]
  write(*, "(I2.2,'-',I3.3,'|',1X,A,3F8.2)") 3, mycomm%rank, &
      & "Value to be operated on:", valr(:)
  call mpifx_allreduce(mycomm, valr, resvalr, MPI_PROD)
  write(*, "(I2.2,'-',I3.3,'|',1X,A,3F8.2)") 4, mycomm%rank, &
      & "Obtained result (prod):", resvalr(:)
  call mpifx_finalize()

end program test_allreduce
```

When running on 4 processors:

```
mpirun -n 4 test_allreduce | sort
```

you should obtain the following output:

```
01-000| Value to be operated on:0
01-001| Value to be operated on:2
01-002| Value to be operated on:4
01-003| Value to be operated on:6
02-000| Obtained result (sum):12
02-001| Obtained result (sum):12
02-002| Obtained result (sum):12
02-003| Obtained result (sum):12
03-000| Value to be operated on:    1.20    4.30    3.80
03-001| Value to be operated on:    2.40    8.60    7.60
03-002| Value to be operated on:    3.60   12.90   11.40
03-003| Value to be operated on:    4.80   17.20   15.20
04-000| Obtained result (prod):   49.77 8205.12 5004.33
04-001| Obtained result (prod):   49.77 8205.12 5004.33
04-002| Obtained result (prod):   49.77 8205.12 5004.33
04-003| Obtained result (prod):   49.77 8205.12 5004.33
```

Have a look at the test folder in the source tree for further examples.

Chapter 4

List of routines

You can generate the list and the description of the MPIFX routines via doxygen (see folder *doc/doxygen/* in the source tree).

Chapter 5

License

MPIFX is licensed under the simplified BSD license: