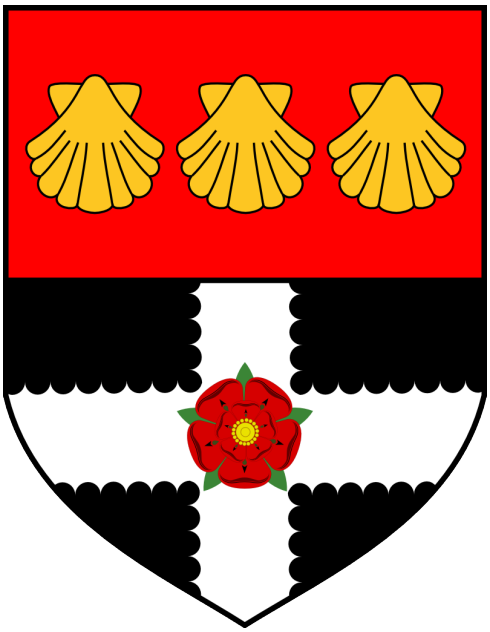
EMPIRE IMPLEMENTATION MANUAL

Philip A. Browne

University of Reading

February 17, 2015



Fortran 95 version.

Version 5.1

Contents

1 Compile and test the minimal programs

- Ensure that a Fortran compiler and mpi is installed on the machine. This has been tested with GNU Fortran (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3.
- Ensure that make is available on the system. This has been tested with GNU Make 3.81.
- Download fortran_empire.tar from:

  `http://www.met.reading.ac.uk/~darc/empire/fortran_empire.tar`.
- Uncompress the tar archive using: `tar -xvf fortran_empire.tar`
- In that directory, open Makefile and ensure that the variable `FC` is set appropriately for the Fortran compiler on the system.
- Set the corresponding compiler options in `FCOPTS`, though none should be strictly necessary.
- Exit the Makefile, and type `make` to generate the executable files.
- If the compilation works successfully, files minimal_model, minimal_pf and run.sh should be created.
- To run the coupled system, and example launching command is in run.sh. If there is no queueing system to be used, this can be executed with the command `./run.sh`.
- If there is a queueing system, the second line in file run.sh should be submitted appropriately. See the local system administrator for help if necessary.
- If the programs run successfully they should output files out.1.00 to out.1.22 containing the output from the models and the particle filter code.
- Experiment with changing the number between -np and minimal_model in `./run.sh` to run multiple ensemble members.

2 Set up communicator

Before any information can be sent we have to define an MPI communicator that links the model to the DA code.

- Find place in code to put initialisation statements. This should be somewhere before the initial model state is defined. If the model is already parallelised using MPI, this should be where mpi_init is called.
- In the program/subroutine corresponding to this place, (if it does not already exist) make the inclusion as follows (normally directly after implicit none):

```
include 'mpif.h'
```

- In this section of code, declare the following variables:

```
integer :: mpi_err,world_size,world_id
integer :: cpl_mpi_comm,couple_root,couple_colour
integer :: particle_id,nens, da, nda
integer :: mdl_num_proc,mdl_id,mdl_mpi_comm,mdlcolour
integer :: models,models_id,models_size
```

- If it does not already exist, add the following line to initilise MPI:

```
call mpi_init(mpi_err)
```

- After the call to mpi_init, we must define how many mpi processes refer to a single instance of the model. For instance, the model may be run on 256 nodes or some other number. Here we hard code the number into the code, but could easily be read from a file to change at runtime. Hard-coding the number into the code corresponds to writing the following line directly after the mpi_init command:

```
mdl_num_proc = 32
```

  Note that if the model is serial, then set mdl_num_proc = 1.

- After setting mdl_num_proc, add the following lines of code:

```
da = 0
call mpi_comm_rank (MPI_COMM_WORLD,world_id, mpi_err)
call mpi_comm_size (MPI_COMM_WORLD,world_size, mpi_err)
call mpi_comm_split(MPI_COMM_WORLD,da, world_id, models, mpi_err)
call mpi_comm_size (models, models_size,mpi_err)
call mpi_comm_rank (models, models_id, mpi_err)
mdlcolour = models_id/mdl_num_proc
call mpi_comm_split(models, mdlcolour, models_id,mdl_mpi_comm,mpi_err)
call mpi_comm_rank (mdl_mpi_comm, mdl_id, mpi_err)
if(mdl_id .eq. 0) then
```

```
   couple_colour = 9999
else
   couple_colour = MPI_UNDEFINED
end if
call mpi_comm_split(MPI_COMM_WORLD,couple_colour,mdlcolour,cpl_mpi_comm,mpi_err)
if(mdl_id .eq. 0) then
   call mpi_comm_size(cpl_mpi_comm,nens,mpi_err)
   call mpi_comm_rank(cpl_mpi_comm,particle_id,mpi_err)
   nda = world_size-models_size
   nens = nens - nda
   couple_root = floor ( real ( nda * particle_id )/ real ( nens ))+ nens
else
   couple_root = -1
end if
```

- Finally, the communicator which the model uses as its global communicator should be changed to mdl_mpi_comm. This can be done with a find and replace command. The process identifier for the model is given by the variable mdl_id. This is not necessary for serial models.

3 Initial send a receive calls

- Find the place in the code where all the model prognostic variables are initialised.
- In this corresponding program/subroutine, declare the following:

```
integer :: state_dim,tag,mpi_err
integer :: mpi_status(MPI_STATUS_SIZE)
real(kind=kind(1.0D0)), allocatable, dimension(:) :: state_vector
```

- If it was not included in this program/subroutine already, make sure that

```
include 'mpif.h'
```

  appears before the declarations.

- Count the total number of prognostic variables. Save this number into the variable state_dim

```
state_dim = ######
```

- Allocate state_vector as

```
if(mdl_id .eq. 0) then
allocate(state_vector(state_dim),mpi_err)
if(mpi_err .ne. 0) write(*,*) 'Could not allocate state_vector'
end if
```

- Now we have to pack all the prognostic variables into state_vector. This step will need to be documented and made available to the users of the data assimilation system, to understand the meaning of the data they will receive. Normally this may be done by some form of loop so we will illustrate this by an example:

```
if(mdl_id .eq. 0) then
       DO I = 1,6032
              STATE_VECTOR(I) = wind(I)
              STATE_VECTOR(I+6032) = height(I)
              STATE_VECTOR(I+2*6032) = density(I)
       END DO
end if
```

- Now immediately write the reverse of this process to unpack the variables.

```
if(mdl_id .eq. 0) then
       DO I = 1,6032
              wind(I) = STATE_VECTOR(I)
              height(I) STATE_VECTOR(I+6032)
              density(I) = STATE_VECTOR(I+2*6032)
       END DO
end if
```

- NOTE FOR PARALLEL PROGRAMS: The prognostic variables may be stored across different processors and so a gather and scatter of this data onto processor mdl_id = 0 may have to be inserted.
- Between the packing and unpacking of the variables, add the following code fragment:

```
if(mdl_id .eq. 0) then
tag = 1
call mpi_send(state_vector,state_dim,MPI_DOUBLE_PRECISION,&
              couple_root,tag,cpl_mpi_comm,mpi_err)
call mpi_recv(state_vector,state_dim,MPI_DOUBLE_PRECISION,&
              couple_root,MPI_ANY_TAG,cpl_mpi_comm,mpi_status,mpi_err)
end if
```

4 Timestepping send and receive calls

- Find the timestepping loop of the code.
- Go to the end of the timestepping loop, after the prognostic variables have been updated to their new values.
- Copy and paste the entire section from the initial calls to mpi_send and mpi_recv to the end of the timestepping loop:

```
if(mdl_id .eq. 0) then
       DO I = 1,6032
              STATE_VECTOR(I) = wind(I)
              STATE_VECTOR(I+6032) = height(I)
              STATE_VECTOR(I+2*6032) = density(I)
       END DO
end if
if(mdl_id .eq. 0) then
tag = 1
call mpi_send(state_vector,state_dim,MPI_DOUBLE_PRECISION,&
              couple_root,tag,cpl_mpi_comm,mpi_err)
call mpi_recv(state_vector,state_dim,MPI_DOUBLE_PRECISION,&
              couple_root,MPI_ANY_TAG,cpl_mpi_comm,mpi_status,mpi_err)
end if
if(mdl_id .eq. 0) then
       DO I = 1,6032
              wind(I) = STATE_VECTOR(I)
              height(I) STATE_VECTOR(I+6032)
              density(I) = STATE_VECTOR(I+2*6032)
       END DO
end if
```

5 Cleaning up

- After the timestepping loop, add the following to deallocate the state_vector

```
if(mdl_id .eq. 0) deallocate(state_vector)
```

- If the code was SERIAL, add the following line:

```
call mpi_finalize(mpi_err)
```

6 Testing

1. Edit the minimal code.
   - Change the state dimension to equal the size of state_vector in the model.
   - Change the number of timesteps to equal the number of timesteps in the model.
2. Recompile the minimal particle filter.
3. Run the code with, say,

   ```
   mpirun -np 1 :  YOUR_EXECUTABLE : -np 1 pf_minimal
   ```

7 Common issues and solutions

- The real kind of `state_vector` was not accounted for.

  EMPIRE will only accept a double as the entries of the state vector. Thus you have to ensure that the variables which you put into it are doubles, and when you extract the variables from the vector that you ensure their kind is the same as before.

  Typically a compiler flag will ensure that all reals are treated as doubles (such as -r8 for the Cray Fortran compiler.)

- Not enough processors were requested to run the job.

  Remember that the number of processors required will include both the requirements of the model (multiple models in an ensemble scenario) and the particle filter code.

  For example in a HecTOR/Archer jobscript, `mppnppn` will have to be large enough to accomodate the job.

8 The Coupling Communicator Explained

Let us describe, line-by-line, what the coupling communicator does. Understanding it should be unnecessary for the user, but may be of interest.

- `da = 0`

  This is a model, not a data assimilation process, so set da to zero.

- `call mpi_comm_rank (MPI_COMM_WORLD,world_id, mpi_err)`

  Get the global identifier world_id of this process.

- `call mpi_comm_size (MPI_COMM_WORLD,world_size, mpi_err)`

  Get the total number of processes launched, store it in world_size.

- `call mpi_comm_split(MPI_COMM_WORLD,da, world_id, models, mpi_err)`

  Separate the global communicator into models and data assimilation processes. All the model processes now share the models communicator.

- `call mpi_comm_size (models, models_size,mpi_err)`

  Compute how many model processes there are, store this as models_size.

- `call mpi_comm_rank (models, models_id, mpi_err)`

  Compute the process rank amongst the models, and store as models_id.

- `mdlcolour = models_id/mdl_num_proc`

  Block the instances of the model into colours. The first mdl_num_proc instances are given mdlcolour = 0, the second mdl_num_proc instances are given mdlcolour = 1 and so on. Note this uses fortran integer division.

- `call mpi_comm_split(models, mdlcolour, models_id,mdl_mpi_comm,mpi_err)`

  Split the global communicator into a communicator for each grouping of the model instances. Those with the same colour are part of the same communicator and this is given in the variable mdl_mpi_comm.

- `call mpi_comm_rank (mdl_mpi_comm, mdl_id, mpi_err)`

  Get the processor identifier within the model communicator. This is stored in the variable mdl_id.

- `if(mdl_id .eq. 0) then`

  `couple_colour = 9999`

  For each ensemble member, we set a coupling colour 9999 only on the master processor. The master processor is given by mdl_id = 0.

- `else`

  `couple_colour = MPI_UNDEFINED`

  `end if` Those processors which are not the master processor of each ensemble member are not given the same coupling colour.

- `call mpi_comm_split(MPI_COMM_WORLD,couple_colour,mdlcolour,cpl_mpi_comm,mpi_err)`

  This split groups the master processor of each ensemble member with each of the instances of the ensemble data assimilation code. It creates the cpl_mpi_comm communicator.

- `if(mdl_id .eq. 0) then`

  ⋮

  `else`

  `couple_root = -1`

  `end if`

  This section is only done by the master processor of each ensemble member. In the other cases, couple_root is set negative to aide in debugging, though it should never be used.

- `call mpi_comm_size(cpl_mpi_comm,nens,mpi_err)`

  This finds the total number of processes in the communicator cpl_mpi_comm. It is temporarily stored in the variable nens.

- `call mpi_comm_rank(cpl_mpi_comm,particle_id,mpi_err)`

  We find the ensemble member number and store it in the variable particle_id.

- `nda = world_size-models_size`

  Compute the number of data assimilation processes as they are those processes which are not on the model communicator.

- `nens = nens - nda`

  Here we use the previously stored variable nens, that contained the combined number of ensemble members and data assimilation processes, to calculate the number of ensemble members.

- `couple_root = floor ( real ( nda * particle_id )/ real ( nens ))+ nens`

  This selects the appropriate identifier of the data assimilation code on the communicator cpl_mpi_comm which the ensemble member will link to.