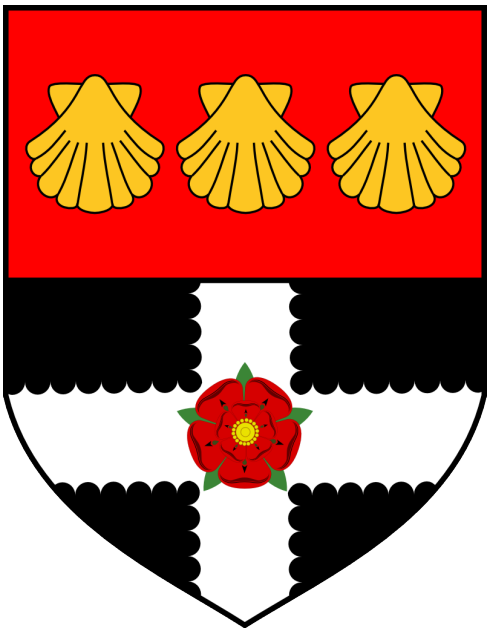
EMPIRE IMPLEMENTATION MANUAL

Philip A. Browne

University of Reading

February 17, 2015



Python 2.7 version.

Version 5.1

Contents

1 Install and test the minimal programs

- Ensure that python and mpi4py is installed on the machine. This has been tested with Python 2.7.6 (default, Mar 22 2014, 22:59:56) [GCC 4.8.2] on linux2
- Download python_empire.tar from:

  http://www.met.reading.ac.uk/~darc/empire/python_empire.tar.
- Uncompress the tar archive using: `tar -xvf python_empire.tar`
- If the compilation works successfully, files model_minimal.py, pf_minimal.py 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 model_minimal.py 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.
- In the program/subroutine corresponding to this place, firstly add the following lines of code:

```
def initialise_mpi():
    global cpl_root,mdl_mpi_comm,cpl_mpi_comm,cpl_id,world_id,mdl_id,ptcl_id


    mdl_num_proc = 4
    da = np.array(0, dtype='i')
    mpi_comm_world = MPI.COMM_WORLD
    world_id = mpi_comm_world.Get_rank()
    world_size = mpi_comm_world.Get_size()

    models=mpi_comm_world.Split(da,world_id)
    models_size= models.Get_size()
    models_id = models.Get_rank()
    mdlcolour = models_id / mdl_num_proc
    mdl_mpi_comm = models.Split(mdlcolour,models_id)
    mdl_id = mdl_mpi_comm.Get_rank()
    if ( mdl_id == 0):
        couple_colour = 9999
    else:
        couple_colour = MPI.UNDEFINED
    cpl_mpi_comm = mpi_comm_world.Split(couple_colour,mdlcolour)
    if ( mdl_id == 0):
        nens = cpl_mpi_comm.Get_size()
        ptcl_id = cpl_mpi_comm.Get_rank()
        nda = world_size-models_size
        nens = nens-nda
        for da in range(1,nda+1):
            if ( ptcl_id < np.float64((nens*(da)))/np.float64(nda)):
                cpl_root = da-1 + nens
                break
            else:
                cpl_root = -1
```

- The variables cpl_root, mdl_id and cpl_mpi_comm must be made available to the latter calls to mpi_send and mpi_recv.

3 Initial send a receive calls

- Find the place in the code where all the model prognostic variables are initialised.
- Count the total number of prognostic variables. Save this number into the variable state_dim
- In this corresponding program/subroutine, declare the following:

```
state_vector = np.zeros(state_dim, dtype=np.float64)
```

- 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 == 0):
       for i in range(6032)
              state_vector[i] = wind[i]
              state_vector[i+6032] = height[i]
              state_vector[i+2*6032] = density[i]
```

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

```
if(mdl_id == 0):
       for i in range(6032):
              wind[i] = state_vector[i]
              height[i] state_vector[i+6032]
              density[i] = state_vector[i+2*6032]
```

- 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 == 0):
       cpl_mpi_comm.Send([state_vector,len(state_vector),MPI.DOUBLE_PRECISION], dest=cpl_root,tag=1)
       cpl_mpi_comm.Recv([state_vector,len(state_vector),MPI.DOUBLE_PRECISION], source=cpl_root,tag=MPI.ANY_TAG,s
```

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 == 0):
       for i in range(6032)
              state_vector[i] = wind[i]
              state_vector[i+6032] = height[i]
              state_vector[i+2*6032] = density[i]

if(mdl_id == 0):
       cpl_mpi_comm.Send([state_vector,len(state_vector),MPI.DOUBLE_PRECISION], dest=cpl_root,tag=1)
       cpl_mpi_comm.Recv([state_vector,len(state_vector),MPI.DOUBLE_PRECISION], source=cpl_root,tag=MPI.ANY_TAG,s

if(mdl_id == 0):
       for i in range(6032):
              wind[i] = state_vector[i]
              height[i] state_vector[i+6032]
              density[i] = state_vector[i+2*6032]
```

5 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. Run the code with, say,

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

6 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.

- 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.

7 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 = np.array(0, dtype='i')`

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

- `mpi_comm_world = MPI.COMM_WORLD`

  Store MPI_COMM_WORLD as a python variable.

- `world_id = mpi_comm_world.Get_rank()`

  Get the global rank world_id of this process.

- `world_size = mpi_comm_world.Get_size()`

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

- `models=mpi_comm_world.Split(da,world_id)`

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

- `models_size= models.Get_size()`

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

- `models_id = models.Get_rank()`

  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 is equivalent to fortran integer division.

- `mdl_mpi_comm = models.Split(mdlcolour,models_id)`

  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.

- `mdl_id = mdl_mpi_comm.Get_rank()`

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

- ```
  if ( mdl_id == 0):
  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`

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

- `cpl_mpi_comm = mpi_comm_world.Split(couple_colour,mdlcolour)`

  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 == 0):`

  This section is only done by the master processor of each ensemble member.

- `nens = cpl_mpi_comm.Get_size()`

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

- `ptcl_id = cpl_mpi_comm.Get_rank()`

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

- ```
        for da in range(1,nda+1):
            if ( ptcl_id < np.float64((nens*(da)))/np.float64(nda)):
                cpl_root = da-1 + nens
                break
            else:
                cpl_root = -1
  ```

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