PDAF Tutorial

Implementation of the analysis step in online mode with a serial model

http://pdaf.awi.de
We demonstrate the implementation of an online analysis step with PDAF with a model that is itself not parallelized using the template routines provided by PDAF.

The example code is part of the PDAF source code package downloadable at http://pdaf.awi.de.
Implementation Tutorial for PDAF online / serial model

This is just an example!

For the complete documentation of PDAF’s interface see the documentation at http://pdaf.awi.de
Overview


2 Parts

a) Global filter
b) Localized filter (and OpenMP-parallelization)

We recommend to first implement the global filter. The localized filter re-uses routines of the global filter.

In this tutorial we only cover the case of a serial model. The implementation with a parallelized model is described in a separate tutorial.
Contents

0a) Files for the tutorial
0b) The model without parallelization
0c) State vector and observation vector
0d) PDAF online mode
0e) Inserting subroutine calls
0f) Forecast phase

1a) Global filter
1b) Local filter without parallelization
   1b.1) Add OpenMP-parallelization to local filter without parallelization

2) Hints for adaptions for real models
0a) Files for the Tutorial
Tutorial implementation

Files are in the PDAF package

Directory:

/tutorial/online_2D_serialmodel

- Fully working implementations of user codes
- PDAF core files are in /src
  Makefile refers to it and compiles the PDAF library
- Only need to specify the compile settings (compiler, etc.) by environment variable PDAF_ARCH. Then compile with ‘make’. 
Template files for online mode

Directory: /templates/online

- Contains all required files
- Contains also command line parser (convenient but not required)

To generate your own implementation:

1. Copy content of directory e.g. into sub-directory of model source code
2. Add calls to interface routines to model code
3. Complete user-routines for your model
4. Adapt compilation (e.g. Makefile) and compile
5. Run with assimilation options
PDAF library

Directory: /src

• The PDAF library is not part of the template
• PDAF is compiled separately as a library and linked when the assimilation program is compiled
• Makefile includes a compile step for the PDAF library
• One can also cd to /src and run ‘make’ there (requires setting of PDAF_ARCH)

$PDAF_ARCH

• Environment variable to specify the compile specifications
• Definition files in /make.arch
• Define by, e.g.
  setenv PDAF_ARCH linux_gfortran (tcsh/csh)
  export PDAF_ARCH=linux_gfortran (bash)
0b) The model without parallelization
Simple assimilation problem

- 2-dimensional model domain
- One single field (like temperature)
- Direct measurements of the field
- Data gaps (i.e. data at selected grid points)
- Same error estimate for all observations
- Observation errors are not correlated (diagonal observation error covariance matrix)
- Simple time stepping:
  Shift field in vertical direction one grid point per time step
2D „Model“

- Simple 2-dimensional grid domain
- 36 x 18 grid points (longitude x latitude)
- True state: sine wave in diagonal direction (periodic for consistent time stepping)
- Simple time stepping:
  Shift field in vertical direction one grid point per time step
- Stored in text files (18 rows) – true_step*.txt
**Model: General program structure**

```plaintext
program main

initialize
    initialize model information:
    - set dimensions
    - allocate model field array
    - read initial field

integrate
    perform time stepping
    - shift model field
    - write new model field

end program

No parallelization!
```
Model: Shared variables

Shared variables are declared in Fortran module (mod_model.F90)

MODULE mod_model

... 

INTEGER :: nx, ny ! Size of 2D grid
INTEGER :: total_steps ! Total number of time steps
REAL, ALLOCATABLE :: field(:,:,:) ! Model field

END MODULE mod_model

• Included with ‘use’ in initialize and integrate
The model source code consists of the following files:

- mod_model.F90
- main.F90
- initialize.F90
- integrate.F90

For clarity, the implementation with PDAF is found in

- main_pdaf.F90
- integrate_pdaf.F90

It allows for easy comparison of the implementations.
Running the tutorial model

- cd to /tutorial/online_2D_serialmodel
- Set environment variable PDAF_ARCH or set it in Makefile (e.g. linux_gfortran)
- Compile by running ‘make model’
- Run the program with ./model

- Inputs are read in from /tutorial/inputs_online
- Outputs are written in /tutorial/online_2D_serialmodel
- Plot result, e.g with ‘octave’:
  
  Load true_step10.txt
  
  Pcolor(true_step10)
Observations

- Add random error to true state (standard deviation 0.5)
- Select a set of observations at 28 grid points
- File storage (in inputs_online):
  - text file, full 2D field, -999 marks ‘no data’ – obs_step*.txt
  - one file for each time step
Ensemble

- Prepared 9 ensemble state files
- Sine waves shifted along diagonal (truth not included)
- One text file per ensemble member – *ens_* .txt (in inputs_online)
Ensemble states at initial time

- Ensemble member 1
- Ensemble member 2
- Ensemble member 3
- Ensemble member 4
- Ensemble member 5
- Ensemble member 6
- Ensemble member 7
- Ensemble member 8
- Ensemble member 9
0c) state vector and observation vector
State vector – some terminology used later

- PDAF performs computations on state vectors
- **State vector**
  - Stores model fields in a single vector
  - Tutorial shows this for one 2-dimensional field
  - Multiple fields are just concatenated into the vector
  - All fields that should be modified by the assimilation have to be in the state vector
- **State dimension**
  - Is the length of the state vector
    (the sum of the sizes of the model fields in the vector)
- **Ensemble array**
  - Rank-2 array which stores state vectors in its columns
**Observation vector**

- **Observation vector**
  - Stores all observations in a single vector
  - Tutorial shows this for one 2-dimensional field
  - Multiple observed fields are just concatenated into the vector

- **Observation dimension**
  - Is the length of the observation vector
    (sum of the observations over all observed fields in the vector)

- **Observation operator**
  - Operation that computes the observed part of a state vector
  - Tutorial only selects observed grid points
  - The operation can involve interpolation or integration depending on type of observation
0d) PDAF online mode
Online mode

• Combine model with PDAF into single program
  • “model_pdaf”

• Add 3 subroutine calls:
  
  - init_parallel_pdaf - add parallelization
  - init_pdaf - initialize assimilation
  - assimilate_pdaf - perform assimilation

• Implement user-supplied routines, e.g. for
  • observation operator
  • initialization of observation vector
  • transfer between state vector and model fields
Program flow with model extended for data assimilation

Simulation Model

Start

- Initialize Model
  - generate mesh
  - Initialize fields

- Do i=1, nsteps
  - Time stepper
    - consider BC
  - Consider forcing

- Post-processing

Stop

Assimilation System

Start

- init_parallel_pdaf

- Initialize Model
  - generate mesh
  - Initialize fields

- init_pdaf

- Do i=1, nsteps
  - Time stepper
    - consider BC
  - Consider forcing

- assimilate_pdaf

- Post-processing

Stop

Legend

Model

Extension for data assimilation
Fully parallel configuration

• Tutorial shows implementation for a fully parallel case
  → Number of processes equals ensemble size!

• For a more flexible (and complicated) configuration see PDAF’s online guide
model_pdaf: General program structure

```fortran
program main_pdaf

  init_parallel_pdaf   - initialize parallelization
  initialize          - initialize model information
  init_pdaf           - initialize parameters for PDAF
                      and read ensemble
  integrate           - time stepping loop
    assimilate_pdaf   - compute analysis step
                        (called inside stepping loop)

end program
```

Note:
In the example code, we use different files main.F90 and main_pdaf.F90 to allow for easy comparison.
mod_assimilation.F90

Fortran module

- Declares the parameters used to configure PDAF
- Will be included (with ‘use’) in the user-written routines
- Additions to template necessary for observation handling
0e) Inserting subroutine calls
Where to insert subroutine calls?

**init_parallel_pdaf**

→ at the start of the program  
  (first operations to be executed)

**init_pdaf**

→ after the initialization of the model  
  i.e. directly before the time stepping loop

**assimilate_pdaf**

→ Last operation in the time stepping loop  
  i.e. just before the `END DO`

Note: One can add the routines one after the other:  
First insert `init_parallel_pdaf` and test the program, then  
add `init_pdaf`, etc.
init_parallel_pdaf.F90

- It is fully implemented template
- Parallelization variables are declared in Fortran module `mod_parallel_pdaf`
- Required adaption:
  - un-comment the second `use model_parallel_pdaf` (includes variables that are declared by the model if it’s parallelized)
  - remove local declaration of `mype_world` & `npes_world`
- Important variable:
  - `n_modeltasks`
    - Defines number of concurrent model integrations.
    - Has to be equal to ensemble size
    - In the example: Read as ‘dim_ens’ from command line (using subroutine ‘parse’)

PDAF tutorial – Analysis step in online mode with a serial model
init_pdaf.F90

Routine sets parameters for PDAF, calls \texttt{PDAF\_init} to initialize the data assimilation, and calls \texttt{PDAF\_get\_state} to prepare the ensemble integrations:

Template contains list of available parameters (declared in and used from \texttt{mod\_assimilation})

Independent of the filter algorithm:

- Include information on size of model fields from model
- Define dimension of state vector
  \[
  \text{dim\_state\_p} = \text{nx} \times \text{ny}
  \]

In call to \texttt{PDAF\_init}, the name of the user-supplied routine for ensemble initialization routine is specified:

\texttt{init\_ens\_pda}
In call to PDAF_get_state, the names of 3 user-supplied routines are specified:

- **next_observation_pdaf**
  - Set number of time steps in forecast phase

- **distribute_state_pdaf**
  - Initialize model fields from state vector

- **prepoststep_ens_pdaf**
  - Poststep routine (compute estimated errors, write state estimate, etc.)

Initially, one can just copy the template routines. One can adapt them later to the particular application.
Routine just calls a filter-specific routine like

\[ \text{PDAF\_assimilate\_estkf} \]

We don’t insert `PDAF\_assimilate\_estkf` directly into the model code

→ because, we need to declare all user-supplied routines as ‘EXTERNAL’. This could clutter the model code.

Filter-specific user routines are described next. Initially, one can just copy the template routines.

Note: Template contains calls for `PDAF\_assimilate\_estkf` and `PDAF\_assimilate\_lestkf`. Need to adapt for other filters.
Differences online and offline

- If you’ve studied the tutorial for offline mode

### Offline
- Separate programs for model and assimilation
- Needed to implement routine `initialize`
- Grid dimensions declared in `mod_assimilation`
- Ensemble information read from files
- `mod_assimilation` contains all field and assimilation variables

### Online
- Extend model program for assimilation
- Operations in `initialize` given by model; no changes for assimilation!
- Grid dimensions defined in model code (`mod_model`)
- Ensemble information provided by model fields
- `mod_assimilation` only contains variables for assimilation
Optional routine: finalize_pdaf.F90

Call to finalize_pdaf can be inserted at the end of the model

Routine contains two calls to PDAF_print info:

CALL PDAF_print_info(2)
       – display information on allocated memory inside PDAF

CALL PDAF_print_info(1)
       – display timing information
         (values 3 and 4 also possible for more detailed timers)

Note: finalize_pdaf only prints the information for mype_world==0

In addition there is

CALL PDAF_deallocate()

which deallocates internal arrays in PDAF
0f) Forecast phase
Files to be changed

Template contains all required files

- just need to be filled with functionality

init_pdaf.F90
init_ens.F90
next_observation_pdaf.F90
distribute_state_pdaf.F90
collect_state_pdaf.F90
init_dim_obs_pdaf.F90
obs_op_pdaf.F90
init_obs_pdaf.F90
prodrinva_pdaf.F90
prepoststep_ens_pdaf.F90

 initialization

ensemble

forecast

analysis step

post step
Routine sets parameters for PDAF and calls PDAF_init to initialize the data assimilation:

Template contains list of available parameters (declared in and used from mod_assimilation)

For the example set:

1. dim_ens = 9
2. rms_obs = sqrt(0.5)
3. filtertype = 6 (for ESTKF)
4. delt_obs = 2 (assimilate after each 2^{nd} time step)

In call to PDAF_init, the name of the ensemble initialization routine is specified:

\texttt{init\_ens\_pdaaf}
A call-back routine called by PDAF_init:

- Implemented by the user
- Its name is specified in the call to PDAF_init
- It is called by PDAF through a defined interface:

```fortran
SUBROUTINE init_ens_offline(filtertype, dim_p, dim_ens, state_p, Uinv, ens_p, flag)
```

Declarations in header of the routine shows “intent” (input, output):

```fortran
REAL, INTENT(out) :: ens_p(dim_p, dim_ens)
```

Note:
All call-back routines have a defined interface and show the intent of
the variables. Their header comment explains what is to be done in
the routine.
**init_ens_pdaf.F90 (2)**

Initialize ensemble matrix \( \text{ens}_p \) for the start time of the assimilation

1. Include \( \text{nx}, \text{ny} \) with use \text{mod_model}
2. Declare and allocate \text{real} :: \text{field}(\text{ny}, \text{nx})
3. Loop over ensemble files \((i=1,\text{dim}_\text{ens})\) for each file:
   - read ensemble state into \text{field}
   - store contents of \text{field} in column \( i \) of \text{ens}_p
4. Deallocate \text{field}

Note:
Columns of \text{ens}_p are state vectors.
Store following storage of field in memory (column-wise in Fortran)
The forecast phase

At this point the initialization of PDAF is complete:

• Initial Ensemble of model states is initialized
• Filter algorithm and its parameters are chosen

Next:

• Implement user-routines for forecast phase
• All are call-back routines:
  - User-written, but called by PDAF

Note:
Some variables end with \_p.
It means that the variable is specific for a process.
(Not relevant until we do parallelization in the analysis step)
next_observation_pdaf.F90

Routine to

- Set number of time steps in next forecast phase
- Set flag to control exit from forecasts (doexit)

Most simple setting:

```f90
include delt_obs from mod_assimilation
nsteps = delt_obs
doexit = 0
```

Note: The assimilation program stops when the maximum number of time steps of the model is reached, even if doexit=0.
More sophisticated setting:

- Utilize `stepnow` (current time step) and `total_steps` (total number of time steps given by model).

```fortran
IF (stepnow + nsteps <= total_steps) THEN
    nsteps = delt_obs ! Forecast length
    doexit = 0 ! Continue assimilation
ELSE
    nsteps = 0 ! No more steps
    doexit = 1 ! Exit assimilation
END IF
```

Note: In the example `doexit=1` is used only inside PDAF and avoids some screen output.
**distribute_state_pdaf.F90**

Routine to

- Initialize model fields from a state vector
- Routine is provided with the state vector `vector_p`

For the example:

1. Access `nx`, `ny` and `field` with use `mod_model`

2. Initialize model field from state vector:

```fortran
DO j = 1, nx
    field(1:ny, j) = state_p(1+(j-1)*ny : j*ny)
END DO
```
Prepoststep_ens_pdaf.F90

Post-step routine for the online mode:

Already there in the template:
1. Compute ensemble mean state `state_p`
2. Compute estimated variance vector `variance`
3. Compute estimated root mean square error `rmserror_est`

Possible extensions:
4. Write analysis state (ensemble mean, `state_step*_ana.txt`)
5. Write analysis ensemble into files
   (Analogous to reading in `init_ens_pdaf`)
6. Analogously one can write the forecast fields
Completion of forecast phase

At this point the implementation of the forecast phase is practically complete:

- Initial ensemble and PDAF’s parameters are set
- The ensemble forecast can be computed

One can now compile the program model_pdaf (make model_pdaf) to check whether it runs.

Note: It is recommended to compile PDAF with –DPDAF_NO_UPDATE at this point as the routine for the analysis step are not yet implemented.

Note: For now, prepoststep_ens_pdaf only lets you test the initial ensemble. Testing the forecast fields need implementation of routine collect_state_pdaf
1a) Global filter
Running the tutorial program

- cd to /tutorial/online_2D_serialmodel
- Set environment variable PDAF_ARCH or set it in Makefile (e.g. linux_gfortran_openmpi)
- Compile by running ‘make model_pdaf’ (next slide will discuss possible compile issues)
- Run the program with
  mpirun –np 9 ./model_pdaf –dim_ens 9
- Inputs are read in from /tutorial/inputs_online
- Outputs are written in /tutorial/online_2D_serialmodel
- Plot result, e.g with ‘octave’:
  load state_step10_ana.txt
  pcolor(state_step10_ana)
Requirements for compiling PDAF

PDAF requires libraries for BLAS and LAPACK

- Libraries to be linked are specified in the include file for make in /make.arch (file according to PDAF_ARCH)

- For $PDAF_ARCH=\text{linux}_gfortran_openmpi$ the specification is
  \text{LINK_LIBS} = \text{-L/usr/lib -llapack -lblas -lm}

- If the libraries are at another non-default location, one has to change the directory name (/usr/lib)

- Some systems or compilers have special libraries (e.g. MKL for ifort compiler, or ESSL on IBM/AIX)

PDAF needs to be compiled for double precision

- Needs to be set at compiler time in the include file for make:

  - For gfortran: \text{OPT = -O3 -fdefault-real-8}
Files in the tutorial implementation

/tutorial/inputs_online

- true_stepY.txt: true state
- state_ini.txt: initial estimate (ensemble mean)
- obs_stepY.txt: observations
- ens_X.txt: initial ensemble members

/tutorial/online_2D_serialmodel (after running model_pdaf)

- state_stepY_ana.txt: analysis state estimate
- ens_X_stepY_ana.txt: analysis ensemble members

X=1,…,9: ensemble member index
Y=1,…,18: time step index

Note: Files *_for.txt contain forecast fields
Result of the global assimilation

For example, at step 10

- The analysis state (center) is closer to the true field than without assimilation (left)
- Truth and analysis are nearly identical (right)
The analysis step

Next: Implement user-routines for the analysis step

The analysis step needs several user-supplied routines for operations like

• write forecast model fields into state vector
• determine number of available observations
• observation operator acting on a state vector
• initialization of the vector of observations
collect_state_pdaf.F90

Routine to

- Fill state vector with forecasted model fields
- Routine is provided with the state vector \texttt{vector}_p

For the example:

1. Access \texttt{nx}, \texttt{ny} and \texttt{field} with \texttt{use mod_model}

2. Initialize state vector from model field:

   \begin{verbatim}
   DO j = 1, nx
       state_p(1+(j-1)*ny : j*ny) = field(1:ny, j)
   END DO
   \end{verbatim}

\textbf{Note:} The routine independent of the filter!
init_dim_obs_pdaf.F90

Routine to

• read observation file
• count number of available observations
  (direct output to PDAF: \texttt{dim_obs_p})

Optional, also

• initialize array holding available observations
• initialize index array telling index of observation point
  in full state vector

The most complicated routine in the example!
(but less than 100 lines)
Preparations and reading of observation file:

1. Include \( nx, ny \) with use mod_model
2. declare and allocate real array \( \text{obs\_field}(ny, nx) \)
3. read observation file for current time step:

Initialize string ‘stepstr’ for time step

OPEN (12, &
    file='inputs\_online/obs'//stepstr//''.txt', &
    status='old')
DO i = 1, ny
    READ (12, *) \( \text{obs\_field}(i, :) \)
END DO
CLOSE (12)
Count available observations (\texttt{dim\_obs\_p}):

1. Declare integer :: \texttt{cnt}, \texttt{cnt0}
2. Now count

\begin{verbatim}
cnt = 0
DO j = 1, nx
  DO i= 1, ny
    IF (\texttt{obs\_field}(i, j) > -999.0) \texttt{cnt} = \texttt{cnt} + 1
  END DO
END DO
\end{verbatim}
\texttt{dim\_obs\_p} = \texttt{cnt}
**init_dim_obs_pdaf.F90 (4)**

Initialize observation vector (`obs`) and index array (`obs_index`):

1. Include `obs_p` and `obs_index_p` with `use mod_assimilation`

2. Allocate
   `obs_p(dim_obs_p), obs_index_p(dim_obs_p)`
   (If already allocated, deallocate first)

3. Now initialize …

**Note:**
The arrays only contain information about valid observations; one could store observations already in files in this way.
3. Now initialize

```fortran
  cnt0 = 0  ! Count grid points
  cnt = 0   ! Count observations
  DO j = 1, nx
    DO i = 1, ny
      cnt0 = cnt0 + 1
      IF (obs_field(i, j) > -999.0) THEN
        cnt = cnt + 1
        obs_index_p(cnt) = cnt0  ! Index
        obs_p(cnt) = obs_field(i, j)  ! observations
      END IF
    END DO
  END DO
END DO
```
obs_op_pdaf.F90

Implementation of observation operator 
acting on some state vector

Input: state vector state_p
Output: observed state vector m_state_p

1. Include obs_index_p with use mod_assimilation
2. Select observed grid points from state vector:

\[
\text{DO } i = 1, \text{dim}_\text{obs}_p \\
\text{m}_\text{state}_p(i) = \text{state}_p(\text{obs}_\text{index}_p(i)) \\
\text{END } \text{DO}
\]

Note:
dim_obs_p is an input argument of the routine
init_obs_pdaf.F90

Fill PDAF’s observation vector

Output: vector of observations \texttt{observation\_p}

1. Include \texttt{obs\_p} by use \texttt{mod\_assimilation}
2. Initialize \texttt{observation\_p}:

   \[
   \texttt{observation\_p} = \texttt{obs\_p}
   \]

Note:
This is trivial, because of the preparations in \texttt{init\_dim\_obs\_pdaf}!
(However, the operations needed to be separate, because PDAF
allocates \texttt{observations\_p} after the call to \texttt{init\_dim\_obs\_pdaf})
Compute the product of the inverse observation error covariance matrix with some other matrix

- Input: Matrix $A_p(dim_{obs\_p}, \text{rank})$
- Output: Product matrix $C_p(dim_{obs\_p}, \text{rank})$
  
  (rank is typically $dim_{ens}-1$)

1. Declare and initialize inverse observation error variance
   
   $ivariance_{obs} = 1.0 / rms_{obs}^{**2}$

2. Compute product:

   DO $j = 1, \text{rank}$
   
   DO $i = 1, dim_{obs\_p}$
   
   $C_p(i, j) = ivariance_{obs} * A_p(i, j)$
   
   END DO
   
   END DO
The analysis step in online mode with the serial (non-parallelized) model is fully implemented now.

The implementation allows you now to use the global filter ESTKF (ETKF and SEIK are usable by adding a call to the corresponding routines PDAF_assimilate_X in assimilate_pdaf)

Not usable are EnKF and SEEK (The EnKF needs some other user files und SEEK a different ensemble initialization)
A complete analysis step

We now have a fully functional analysis step
- if no localization is required!

Possible extensions for a real application:

Adapt routines for

- Multiple model fields
  → Store full fields consecutively in state vector

- Third dimension
  → Extend state vector

- Different observation types
  → Store different types consecutively in observation vector

- Other file type (e.g. binary or NetCDF)
  → Adapt reading/writing routines
Differences between online and offline modes

For the analysis step in online mode:

- `collect_state_pdaf` - additional routine for online mode
- `init_dim_obs_pdaf` - read from file for current time step; include \( nx, ny \) from `mod_model` instead of `mod_assimilate`
- `obs_op_pdaf` - identical in online and offline modes
- `init_obs_pdaf` - identical in online and offline modes
- `prodrinva_pdaf` - identical in online and offline modes
1b) Local filter without parallelization
Localization

Localization is usually required for high-dimensional systems

- Update small regions \((S)\)
  (e.g. single grid points, single vertical columns)
- Consider only observations within cut-off distance \((D)\)
- Weight observations according to distance from \(S\)
The FULL observation vector

- A single local analysis at S (single grid point) need observations from domain $D$
- A loop of local analyses over all S needs all observations
  - This defines the *full* observation vector
- Why distinguish *full* and *all* observations?
  - They can be different in case of parallelization!
- Example:
  - Split domain in left and right halves
  - Some of the analyses in left half need observations from the right side.
  - Depending on localization radius not all observations from the right side might be needed for the left side analyses
Running the tutorial program

- Compile as for the global filter
- Run the program with
  ```bash
  mpirun -np 9 ./model_pdaf --dim_ens 9 OPTIONS
  ```
- OPTIONS are always of type  _KEYWORD VALUE_
- Possible OPTIONS are
  - `filtertype 7`  (select LESTKF if not set in init_pdaf)
  - `local_range 5.0`  (set localization radius, 0.0 by default, any positive value should work)
  - `locweight 2`  (set weight function for localization, default=0 for constant weight of 1; possible are integer values 0 to 4; see init_pdaf)
Result of the local assimilation

```bash
mpirun -np 9./model_pdaf --dim_ens 9 -filtertype 7
```

- Default: zero localization radius (local_range=0.0)
- Change only at observation locations
Result of the local assimilation (2)

\[ \text{... -filtertype 7 -local\_range 10.0} \]

- All local analysis domains are influenced (all see observations)
- Up to 16 observations in a single local analysis (average 9.6)

Note: The set up of the experiment favors the global filter because of the shape of the ensemble members
Result of the local assimilation (2)

... -filertype 7 -local_range 10.0 -locweight 2

- Observation weighting by 5th-order polynomial
- Analysis field is smoother than before (because of weighting)
Result of the local assimilation (3)

... -filtertype 7 -local_range 40.0

- Large radius: All local analysis domains see all observations
- Result identical to global filter
Local filter LESTKF

- Localized filters are a variant of the global filters
- User written files for global filter can be widely re-used
- Additional user-written files to handle local part

- No changes to:
  ```
  initialize.F90
  init_ens_pdaf.F90
  prepoststep_ens_pdaf.F90
  ```

- Change in `init_pdaf.F90`:
  ```
  Set filtertype = 7
  ```
  (You can also set it later on command line)
Local filter LESTKF (2)

Adapt files from global analysis

- `init_dim_obs_pdaf.F90` → `init_dim_obs_f_pdaf.F90`
- `obs_op_pdaf.F90` → `obs_op_f_pdaf.F90`
- `init_obs_pdaf.F90` → `init_obs_f_pdaf.F90`
- `prodrinva_pdaf.F90` → `prodrinva_l_pdaf`

Naming scheme:

- `_f_` “full”: operate on all required observations
  (without parallelization these are all observations)
- `_l_` “local”: operation in local analysis domain or corresponding
  local observation domain
Local filter LESTKF (3)

Additional files for local analysis step

```
init_n_domains_pdaf.F90
init_dim_l_pdaf.F90
init_dim_obs_l_pdaf.F90
g2l_state_pdaf.F90
g2l_obs_pdaf.F90
init_obs_l_pdaf.F90
l2g_state_pdaf.F90
```

Discuss now the files in the order they are called
init_n_domains_pdaf.F90

Routine to set the number of local analysis domains

Output: n_domains_p
For the example: number of grid points (nx * ny)

To do:
1. Include nx, ny with use mod_model
2. Set
   \[ n\_domains\_p = nx * ny \]
init_dim_obs_f_pdaf.F90

Initialize dimension of full observation vector

For the local filter:

1. Copy functionality from init_dim_obs_pdaf.F90
2. Rename dim_obs_p to dim_obs_f
3. Add storage of observation coordinates
   1. include coords_obs_f with use mod_assimilation
   2. Where obs_index_p is allocated in the routine:
      Allocate also coords_obs_f(2,cnt)
   3. In the loop where obs_index_p is initialized add:
      coords_obs_f(1,cnt)=j
      coords_obs_f(2,cnt)=i
obs_op_f_pdaf.F90

Implementation of observation operator
for full observation domain

1. Copy functionality from obs_op_pdaf.F90

2. Rename
   
   • dim_obs_p to dim_obs_f
   
   • m_state_p to m_state_f

Note:

The renaming is just for consistency. Quantities referring to the full observations should be recognizable by _f
init_obs_f_pdaf.F90

Fill PDAF’s full observation vector

1. Copy functionality from init_obs_pdaf.F90

2. Rename
   - `dim_obs_p` to `dim_obs_f`
   - `observation_p` to `observation_f`

Note:

The renaming is just for consistency. Quantities referring to the full observations should be recognizable by `_f`
**init_dim_l_pdaf.F90**

Set the size of the local analysis domain

Each single grid point is a local analysis domain in the example

1. Set `dim_l = 1`
init_dim_obs_l_pdaf.F90

Set the size of the observation vector for the local analysis domain

As for the global filter, this is the longest routine (~108 lines)

Only direct output: \texttt{dim\_obs\_l}

Operations:

1. Determine coordinates of local analysis domain
2. Determine coordinate range for observations
3. Count observations within prescribed localization radius
4. Set index array for local observations and array of distances of local observations (optional)

\textbf{Note:} The index array in step 4 is re-used for an efficient implementation of \texttt{g2l\_obs\_pdaf}. The local distance array initialized in step 4 is re-used in \texttt{prodrinva\_l\_pdaf} avoiding to recompute distances.
1. Determine coordinates of local analysis domain

   1. Declare
      
      ```
      real :: coords_l(2)
      ```

   2. Include nx, ny with use mod_model

   3. Compute coords_l from nx, ny:

      ```
      coords_l(1) = real(ceiling(real(domain_p)/real(ny)))
      coords_l(2) = real(domain_p) - (coords_l(1)-1)*ny
      ```

**Note:** The computation of coords_l relies on the order of elements in the state vector (full columns of the field are stored one after the other in the state vector)
init_dim_obs_l_pdaf.F90 (3)

2. Determine coordinate range for local observations
   1. Declare real :: limits_x(2), limits_y(2)
   2. Include local_range with use mod_assimilation
   3. Set lower and upper limits. E.g. for x-direction

   \[
   \text{limits}_x(1) = \text{coords}_l(1) - \text{local_range} \\
   \text{if } (\text{limits}_x(1) < 1.0) \text{ limits}_x(1) = 1.0 \\
   \text{limits}_x(2) = \text{coords}_l(1) + \text{local_range} \\
   \text{if } (\text{limits}_x(2) > \text{real}(\text{nx})) \text{ limits}_x(2) = \text{real}(\text{nx})
   \]

   (analogous for y-direction)

**Note:** Using \text{limits}_x, \text{limits}_y is not strictly required, but it makes the search for local observations more efficient.

If the localization is only based on grid point indices, the coordinates could be handled as integer values.
3. Count local observations (within distance local_range)

\[
\text{dim\_obs\_l} = 0 \\
\text{DO } i = 1, \text{dim\_obs\_f} \\
\quad \text{IF ("coords\_obs(:,i) within coordinate limits") THEN} \\
\quad \quad \text{Compute distance between coords\_obs and coords\_l} \\
\quad \quad \text{IF (distance } \leq \text{ local\_range) } \& \\
\quad \quad \quad \text{dim\_obs\_l} = \text{dim\_obs\_l} + 1 \\
\quad \text{END IF} \\
\text{END DO}
\]

**Note:**
For efficiency, we only compute distance for observations within coordinate limits limits_x, limits_y. Valid local observations reside within circle of radius local_range.
4. Set index array for local observations
   - Index of a local observation in the full observation vector
     1. Include `obs_index_l` and `distance_l` with `use mod_assimilation`
     2. Allocate `obs_index_l(dim_obs_l)`
     3. Fill index array:
        ```
cnt = 0
DO i = 1, dim_obs_f
   IF ("coords_obs(:,i) within coordinate limits") THEN
      Compute distance between coords_obs and coords_l
      IF (distance <= local_range) THEN
         cnt = cnt + 1
      obs_index_l(cnt) = i
      distance_l(cnt) = distance
      END ...
```
g2l_state_pdaf.F90

Initialize state vector for local analysis domain from global state vector

➢ Here the local state is just one element of the global state vector

Input: \texttt{state\_p(1:dim\_p)}
Output: \texttt{state\_l(1:dim\_l)}

1. Set
   \[
   \texttt{state\_l} = \texttt{state\_p(domain\_p)}
   \]

Note:
\[\texttt{dim\_l} = 1\] in the example
Initialize local observed state vector from full observed vector

- We use the index array $\text{obs\_index\_l}$ for this.

Input: $\text{mstate\_f}(1:\text{dim\_obs\_f})$
Output: $\text{mstate\_l}(1:\text{dim\_obs\_l})$

1. Include $\text{obs\_index\_l}$ with use \texttt{mod\_assimilation}
2. Initialize $\text{mstate\_l}$:

   \[
   \text{DO } i = 1, \text{ dim\_obs\_l} \\
   \quad \text{mstate\_l}(i) = \text{mstate\_f}(\text{obs\_index\_l}(i)) \\
   \text{END DO}
   \]
Initialize local vector of observations.

- Again, we use the index array `obs_index_l` for this.

Output: `observation_l(1:dim_obs_l)`

1. Include `obs_f` and `obs_index_l` with `use mod_assimilation`

2. Initialize local observation vector

   ```fortran
   DO i = 1, dim_obs_l
       observation_l(i) = obs_f(obs_index_l(i))
   END DO
   ```
prodrinva_l_pdaf.F90

Compute the product of the inverse observation error covariance matrix with some other matrix
+ apply observation localization (weighting)

➢ The weighting and the product are fully implemented for a diagonal observation error covariance matrix with constant variance

When we re-use the array distance_l initialized in init_dim_obs_l_pdaf, no changes are required here.
l2g_state_pdaf.F90

Initialize global state vector from state vector for local analysis domain

➤ Here the local state is just one element of the global state vector

Input: \texttt{state\_l(1:dim\_l)}
Output: \texttt{state\_p(1:dim\_p)}

1. Implement inverse operation to that in \texttt{g2l_state_pdaf.F90}

\[
\texttt{state\_p(domain\_p)} = \texttt{state\_l}
\]

Note:
The implementation utilizes that \texttt{dim\_l = 1}
Now, the analysis step for local ESKTF in offline mode is fully implemented. The implementation allows you now to use the local filter LESTKF (LETKF, LSEIK can be used after adding calls to PDAF_assimilate_X)

Not usable are EnKF and SEEK (PDAF does not have localization for these filters)

For testing one can vary localization parameters:

\begin{align*}
\text{local\_range} & \quad \text{the localization radius} \\
\text{locweight} & \quad \text{the weighting method}
\end{align*}

Default are $\text{local\_range}=0.0$ (observation at single grid point) and $\text{locweight}=1$ (uniform weight)
A complete local analysis step

We now have a fully functional analysis step including localization

- It can be adapted to multiple model fields, 3 dimensions, different observations, etc.
- It can be used even with big models
  - if computing time is no concern
  - and if the computer has sufficient memory
    (e.g. ensemble array with dimension $10^7$ and 20 members requires about 1.6 GB)

- Parallelization of the analysis step
  - is required if the problem is too big for a single process
  - is recommended if you used a parallelized model
1b.1) Add OpenMP-parallelization to local filter without parallelization
Running the tutorial program

Run analogously to local files without parallelization

- `cd` to `/tutorial/online_2D_serialmodel_openMP`
- Set environment variable `PDAF_ARCH` or set it in Makefile (e.g. `linux_gfortran_openmpi`)
- Check and edit the make include file to activate OpenMP
  - for `gfortran`: `OPT = ... -fopenmp`
  - for Intel compiler: `OPT = ... -openmp`
- Compile by running ‘`make`’
- Set the number of OpenMP threads as environment variable, e.g.
  - for bash: `export OMP_NUM_THREADS=2`
  - for tcsh: `setenv OMP_NUM_THREADS 2`
- Run the program as in the case without OpenMP-parallelization
Results from running with OpenMP parallelization

The results should be *identical* to those without parallelization.

If the program is compiled with activated OpenMP-parallelization, you will see in the output of the analysis step the line

--- Use OpenMP parallelization with 2 threads
OpenMP

• OpenMP is so-called *shared-memory parallelization*

• Support for OpenMP is built into current compilers (needs to be activated by compiler-flag)

• Define OpenMP in the code by compiler directives: `!$OMP ...`

• Shared-memory parallelization:
  • Run several OpenMP “threads” (like processes in MPI)
  • All threads can access the same array in memory, but perform different operations
  • Typical is loop-parallelization: Each thread executes some part of a loop. For example, a fraction of a vector:

    ```
    !$OMP parallel do
    DO i = 1, 1000
       a(i) = b(i) + c(i)
    ENDDO
    ```

    With 2 threads, typically:
    • thread 1 runs i=1 to 500
    • thread 2 runs i=501 to 1000
OpenMP – what’s relevant for PDAF

The local filters (LESTKF, LETKF, LSEIK) are parallelized with OpenMP

- The loop over local analysis domains is distributed over threads

To make this work:

- Take into account, whether a variable is
  - shared (all threads see the same) or
  - private (each thread has its own copy)

- Variables referring to a local analysis domain (e.g. coords_l) have to be private

- Variables with Fortran ‘save’ attribute are shared by default!

Change in user-supplied routines:

- Ensure that variables for local analysis domain are private
Adding OpenMP to local filter LESTKF

In the tutorial implementation only two files need changes:

```
prodrinva_l_pdaf.F90
mod_assimilation.F90
```

One can easily see the changes by using ‘diff’ to compare the files in
```
/tutorial/online_2D_serialmodel_openmp
```
with those in
```
/tutorial/online_2D_serialmodel     (no parallelization)
```
prodrinva_I_pdaf.F90

One variable has attribute ‘save’:

    domain_save

Set it private to the thread by

    !$OMP THREADPRIVATE(domain_save)

The other changes are just there for ‘nice’ screen output
(see following slides)
Each OpenMP thread will write screen output

- Change output, so that only thread 0 writes
- Determine thread number with omp_get_thread_num()

Necessary changes:

1. Include omp_get_thread_num from module omp_lib
   (OpenMP provides the module)
   
   ```
   #if defined (_OPENMP)
     USE omp_lib, ONLY: omp_get_thread_num
   #endif
   ```

2. Declare the thread variable mythread and set it private

   ```
   INTEGER, SAVE :: mythread
   !$OMP THREADPRIVATE(mythread, domain_save)
   ```
3. Now, initialize `mythread`

   ```
   #if defined (_OPENMP)
   mythread = omp_get_thread_num()
   #else
   mythread = 0
   #endif
   
   (The construct is required to ensure that `mythread` is also initialized without openmp)
   
   4. Finally, use `mythread` to set the verbosity flag

   IF (mythread>0) verbose = 0
Several of the user-supplied routines use the variables

    obs_index_l and distance_l

- These variables are specific for each local analysis domain
- The variables are declared in mod_assimilation.F90
- to ensure that these variables are ‘private’ add after the declaration of the variables

    !$OMP THREADPRIVATE (obs_index_l, distance_l)
2) Hints for adaptations for real models
Implementations for real models

- Tutorial demonstrates implementation for simple model
- You can base your own implementation on the tutorial implementation or the templates provided with PDAF
- Need to adapt most routines, e.g.
  - Specify model-specific state vector and its dimension
  - Adapt `distribute_state` and `collect_state`
  - Adapt routines handling observations
- Further required changes
  - Adapt file output (usually only want to write ensemble mean state in `prepoststep_pdaf`; sometimes possible to use output routines from model)
Multiple fields in state vector

- Tutorial uses a single 2-dimensional field
- All fields that should be updated by the assimilation have to be part of the state vector
- For more fields:
  - concatenate them in the state vector
  - adapt state dimension in `init_pdaf`
  - adapt `init_ens_pdaf`, `collect_state_pdaf`, `distribute_state_pdaf`, `prepoststep_pdaf`
  - For local filters: Adapt full (_f_) and local (_l_) routines and `g2l_state_pdaf`, `l2g_state_pdaf`, `g2l_obs_pdaf`
- Note
  - It can be useful to define a vector storing the offset (position) of each field in the state vector
Multiple observed fields

- In tutorial: observed one field at some grid points
- For several observed fields adapt observation routines:
  - concatenate observed fields in observation vector
  - adapt all observation-handling routines
- Note
  - The observation errors can be set differently for each observed field (e.g. using an array `rms_obs`)
  - The localization radius can be set specific for each observed field (observation search in `init_dim_obs_l_pdaf` would use different `local_range` for different fields)
  - One can use spatially varying observation errors using an array `rms_obs` in `prodrinva(_l)_pdaf`
The End!

Tutorial described example implementations

• Online mode of PDAF parallelized over ensemble members
• Simple 2D model without parallelization and with OpenMP parallelization
• Square root filter ESTKF
  • global and with localization
• Extension to more realistic cases possible with limited coding
• Applicable also for large-scale problems

For full documentation of PDAF and the user-implemented routines see http://pdaf.awi.de