Implementation of the analysis step in offline mode
We demonstrate the implementation of an offline analysis step with PDAF 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 offline

This is just an example!

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

Focus on Error Subspace Transform Kalman Filter  

4 Parts

1. Without parallelization  2. With MPI-parallelization
   a) Global filter  a) Global filter
   b) Localized 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.

We assume that 1a is implemented before 1b and 1a is implemented 
before 2a (1b before 2b).
Contents

0a) Files for the tutorial
0b) The model
0c) State vector and observation vector
0d) PDAF offline mode

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

2a) Parallelized global filter
2b) Parallelized local filter

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

Files are in the PDAF package

Directories:

/tutorial/offline_2D_serial  (without parallelization)
/tutorial/offline_2D.openmp  (with OpenMP-parallelization)
/tutorial/offline_2D.parallel  (with MPI parallelization)

• 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’.
Templates for offline mode

Directory: /templates/offline

- Contains all required files
- Contains also command line parser, memory counting, timers (convenient but not required)

To generate your own implementation:

1. Copy directory to a new name
2. Complete routines for your model
3. Set base directory (BASEDIR) in Makefile
4. Set $PDAF_ARCH
5. Compile
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
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)
- Perform a single analysis step using input files for the ensemble and observations (offline mode: No time stepping in the assimilation program)
2D „Model“

- Simple 2-dimensional grid domain
- 36 x 18 grid points (longitude x latitude)
- True state: sine wave in diagonal direction
- No dynamics for offline mode
- Stored in text file (18 rows) – true.txt
Observations

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

- Ensemble size 9
- Sine waves shifted along diagonal (truth not included)
- One text file per ensemble member – ens*.txt

Initial estimate (ensemble mean) vs True field
Ensemble states
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 offline mode
Offline mode

• Two separate programs
  • “Model” – performs ensemble integrations
  • “PDAF_offline” – perform analysis step

• Couple both programs through files
  1. “PDAF_offline” reads ensemble forecast files
  2. Performs analysis step
  3. Writes analysis ensemble files (restart files for “Model”)
  4. “Model” reads restart files and performs ensemble integration
Programs in offline mode

**Model**
- Start
- Initialize Model
  - generate mesh
  - Initialize fields
- Do i=1, nsteps
  - Time stepper
    - consider BC
    - Consider forcing
- Post-processing
- Stop

**Assimilation program**
- Start
  - init_parallel_pdaf
  - init_pdaf
  - assimilation_pdaf
- Stop

- Run for each ensemble member
- Write restart files
- Read restart files (ensemble)
- Compute analysis step
- Write new restart files

PDAF tutorial – Analysis step in offline mode
**PDAF**

**PDAF**

**offline**: General program structure

```plaintext
program main_offline
  init_parallel_pdaf
    initialize communicators
    (not relevant without parallelization)
  initialize
    initialize model information
  init_pdaf
    initialize parameters for PDAF and read ensemble
  assimilation_pdaf
    perform analysis
    (by call to PDAF_put_state_X)
end program
```

PDAF tutorial – Analysis step in offline mode
1a) Global filter without parallelization
Running the tutorial program

- cd to /tutorial/offline_2D_serial
- Set environment variable PDAF_ARCH or set it in Makefile (e.g. linux_gfortran)
- Compile by running ‘make’ (next slide will discuss possible compile issues)
- Run the program with ./PDAF_offline

- Inputs are read in from /tutorial/inputs_offline
- Outputs are written in /tutorial/offline_2D_serial
- Plot result, e.g with ‘octave’:
  - load state_ana.txt
  - pcolor(state_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 $PDAF_ARCH=linux_gfortran$ the specification is

  LINK_LIBS =-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: OPT = -O3 -fdefault-real-8
Files in the tutorial implementation

/tutorial/inputs_offline
- true.txt  true state
- state_ini.txt  initial estimate (ensemble mean)
- obs.txt  observations
- ens_X.txt ($X=1,\ldots,9$)  ensemble members

/tutorial/offline_2D_serial  (after running PDAF_offline)
- state_ana.txt  analysis state estimate
- ens_X_ana.txt ($X=1,\ldots,9$)  analysis ensemble members
Result of the global assimilation

- The analysis state is closer to the true field than the initial estimate
- Truth and analysis are not identical (the ensemble does not allow it)
Files to be changed

Template contains all required files

➢ just need to be filled with functionality

- mod_assimilation.F90
- initialize.F90
- init_pdaf_offline.F90
- init_ens_offline.F90
- init_dim_obs_pdaf.F90
- obs_op_pdaf.F90
- init_obs_pdaf.F90
- prodrinva_pdaf.F90
- prepoststep_ens_offline.F90

Fortran module

- initialization
- analysis step
- post step

PDAF tutorial – Analysis step in offline mode
mod_assimilation.F90

Fortran module

- Declares the parameters used to configure PDAF
- Add model-specific variables here (see next slides)
- Will be included (with ‘use’) in the user-written routines
initialize.F90

Routine initializes the model information

1. Define 2D mesh in mod_assimilation.F90
   
   ```
   integer :: nx, ny
   ```

2. In initialize.F90 include `nx`, `ny`, and `dim_state_p`
   with `use mod_assimilation`

3. Define mesh size in initialize.F90
   
   ```
   nx = 36
   ny = 18
   ```

4. Define state dimension in initialize.F90
   
   ```
   dim_state_p = nx * ny
   ```

**Note:** Some variables end with `_p`.
It means that the variable is specific for a process.
(Not relevant until we do parallelization)
Routine sets parameters for PDAF and calls \texttt{PDAF\_init} to initialize the data assimilation:

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

For the example set:

1. \texttt{dim\_ens} = 9
2. \texttt{rms\_obs} = sqrt(0.5)
3. \texttt{filtertype} = 6 (for ESTKF)

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

\texttt{init\_ens\_offline}
init_envelope_offline.F90

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_envelope_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.
Initialize ensemble matrix \texttt{ens\_p}

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

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

At this point the initialization of PDAF is complete:

- Forecast ensemble is initialized
- Filter algorithm and its parameters are chosen

Next:

- Implement user-routines for analysis step
- 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)
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 \texttt{nx}, \texttt{ny} with \texttt{use mod_assimilation}
2. declare and allocate real array \texttt{obs\_field(ny, nx)}
3. read observation file:

\begin{verbatim}
OPEN (12, file='inputs_offline/obs.txt', &
     status='old')
DO i = 1, ny
    READ (12, *) \texttt{obs\_field(i, :)}
END DO
CLOSE (12)
\end{verbatim}
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 (obs_field(i,j) > -999.0) cnt = cnt + 1
  END DO
END DO
\end{verbatim}

\texttt{dim_obs_p} = cnt
Initialize observation vector (obs) and index array (obs_index):

1. In `mod_assimilation` it is declared
   
   ```f90
   real, allocatable :: obs_p(:), obs_index_p(:)
   ```
   Include these variable with `use mod_assimilation`

2. Allocate
   
   ```f90
   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

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
Implementation of observation operator acting on some state vector

Input: state vector  \texttt{state\_p}

Output: observed state vector  \texttt{m\_state\_p}

1. Include \texttt{obs\_index\_p} by use \texttt{mod\_assimilation}

2. Select observed grid points from state vector:

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

Note:
\texttt{dim\_obs\_p} is an input argument of the routine
Fill PDAF’s observation vector

Output: vector of observations observation_p

1. Include obs by use mod_assimilation
2. Initialize observation_p:

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

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

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

1. Declare and initialize inverse observation error variance
   
   $\text{ivariance\_obs} = 1.0 / \text{rms\_obs}**2$

2. Compute product:

   ```
   DO j = 1, rank
     DO i = 1, dim_{obs}\_p
       c_p(i, j) = ivariance\_obs * A_p(i, j)
     END DO
   END DO
   ```
Post-step routine for the offline mode:

Already there in the template:
1. Compute ensemble mean state \texttt{state\_p}
2. Compute estimated variance vector \texttt{variance}
3. Compute estimated root mean square error \texttt{rmserror\_est}

Required extension:
4. Write analysis ensemble into files used for model restart
   (Analogous to reading in \texttt{init\_ens\_pdaf\_offline})

Possible (useful) extension:
5. Write analysis state (ensemble mean, \texttt{state\_ana.txt})
Done!

The analysis step in offline mode is fully implemented now.

The implementation allows you now to use the global filters ESTKF, ETKF, and SEIK.

Not usable are EnKF and SEEK (The EnKF needs some other user files and 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
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 ./PDAF_offline OPTIONS
• OPTIONS are always of type –KEYWORD VALUE
• Possible OPTIONS are
  • –filtertype 7 (select LESTKF if not set in init_pdaf_offline)
  • –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_offline)
Result of the local assimilation

```
./PDAF_offline -filtertype 7
```

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

./PDAF_offline -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)

```
./PDAF_offline -filtertype 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)

./PDAF_offline -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_offline.F90`
  - `prepoststep_ens_offline.F90`

- Change in `init_pdaf_offline.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_assimilation
2. Set
   \[
   n\_domains\_p = nx \times ny
   \]
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 and obs_p to obs_f

3. Add storage of observation coordinates
   a) Include coords_obs_f with use mod_assimilation
   b) Where obs_index_p is allocated in the routine: Allocate also coords_obs_f(2,cnt)
   c) In the loop where obs_index_p is initialized add:
      
      coords_obs_f(1,cnt)=REAL(j)
      coords_obs_f(2,cnt)=REAL(i)

Note: We treat all coordinates as REAL variables even we use grid point indices here
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
Set the size of the local analysis domain

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

1. Set \texttt{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 (~105 lines)
Only direct output: **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)

**Note:** The index array in step 4 is re-used for an efficient implementation of g2l_obs_pdaf. The local coordinate array initialized in step 4 is re-used in prodrinva_l_pdaf avoiding to recompute coordinates.
1. Determine coordinates of local analysis domain
   1. Declare
      ```fortran
      real :: coords_l(2)
      ```
   2. Include $nx$, $ny$ with use mod_assimilation
   3. Compute $coords_l$ from $nx$, $ny$:
      ```fortran
      coords_l(1) = real(ceiling(real(domain_p)/real(ny)))
      coords_l(2) = real(domain_p) - (coords_l(1)-1)*real(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)
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

      limits_x(1) = coords_l(1) - local_range
      if (limits_x(1) < 1.0) limits_x(1) = 1.0
      limits_x(2) = coords_l(1) + local_range
      if (limits_x(2) > real(nx)) limits_x(2) = real(nx)

      (analogous for y-direction)

**Note:** Using limits_x, limits_y is not strictly required, but it can make 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`)

\[
\text{dim\_obs\_l} = 0
\]

\[
\text{DO } i = 1, \text{ dim\_obs\_f }
\]

\[
\text{IF ("coords\_obs(:,i) within coordinate limits") THEN}
\]

\[
\text{Compute distance between coords\_obs and coords\_l}
\]

\[
\text{IF (distance } \leq \text{ local\_range) } \&
\]

\[
\text{dim\_obs\_l} = \text{dim\_obs\_l} + 1
\]

\[
\text{END IF}
\]

\[
\text{END DO}
\]

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` which is checked with distance.
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:

   ```fortran
   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...
   END ...
   ```
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: state_p(1:dim_p)
Output: state_l(1:dim_l)

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

Note:
\[ \text{dim}_l = 1 \] in the example, so there is not loop required here.
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 \text{mod\_assimilation}

2. Initialize \( \text{mstate\_l} \):

   ```
   \text{DO } i = 1, \text{dim\_obs\_l}
   \qquad \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

   ```
   DO i = 1, dim_obs_l
       observation_l(i) = obs_f(obs_index_l(i))
   END DO
   ```
Compute the product of the inverse observation error covariance matrix with some other matrix
+ apply observation localization (weighting)

- In the template 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 of the template are required here.
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: `state_l(1:dim_l)`
Output: `state_p(1:dim_p)`

1. Implement inverse operation to that in `g2l_state_pdaf.F90`
   
   `state_p(domain_p) = state_l`

Note:
The implementation utilizes that `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 filters LESTKF, LETKF, and LSEIK.

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

For testing one can vary localization parameters:

- `local_range` – the localization radius
- `locweight` – the weighting method

Default are `local_range=0.0` (observation at single grid point) and `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 is required
  - if the problem is too big for a single process
1b.1) Add OpenMP-parallelization to local filter without parallelization
Running the tutorial program

Run analogously to local files without parallelization

- cd to /tutorial/offline_2D_openMP
- Set environment variable PDAF_ARCH or set it in Makefile (e.g. linux_gfortran)
- 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 it’s 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)
```
One variable has attribute ‘save’:

\[ \text{domain}\_\text{save} \]

Set it private to the thread by

\[ !\text{OMP THREADPRIVATE}(\text{domain}\_\text{save}) \]

The other changes are just there for ‘nice’ screen output (see following slides)
prodrinva_l_pdaf.F90 – nice screen output with OpenMP

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

```c
#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

```c
IF (mythread>0) verbose = 0
```
mod_assimilation.F90

Several of the user-supplied routines use the variables

\[ \text{obs\_index\_l and distance\_l} \]

- 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 variable declarations

\[ \text{!$OMP THREADPRIVATE (obs\_index\_l, distance\_l)} \]
2a) Parallelized global filter
Parallelize the analysis step

Implementation Strategy:
Take files from global analysis without parallelization and add the parallelization

Parallelization:
• Perform analysis step using multiple processors
• Split the state vector into equal parts to distribute the work

Notation for parallelization:
• Suffix _p marks variables with process-specific values
• Parallelization variables are declared in the module mod_parallel
Decomposition of model field

Want to distribute the state vector over the processes

→ Split state vector into approximately equal continuous parts

→ Corresponds to distribution along second index of model field (the first one in continuous in memory)

For 36 grid points we have uniform distributions for 2, 3, 4, 6, or 9 processes (other numbers are possible)
Running the parallel tutorial program

• cd to /tutorial/offline_2D_parallel
• Set environment variable PDAF_ARCH or set it in Makefile (e.g. linux_gfortran_openmpi)
• Clean existing files with ‘make cleanall’ (This also removes the compiled PDAF library from previous tests)
• Compile by running ‘make’ (this also builds the PDAF library again; now with parallelization)
• Run the program with
  
  mpirun -np X ./PDAF_offline

  (X>0; optimal are x=1,2,3,4,6 because then ny=36 is dividable by X)
Impact of the parallelization

- Ensemble array is distributed $\rightarrow$ less memory per process (visible in the memory display at the end of the screen output):

$\texttt{mpirun -np 1 ./PDAF_offline}$

PDAF Memory overview
---------------------------------------------
Allocated memory  (MB)
state and A:  0.00543 MB (persistent)
ensemble array:  0.04449 MB (persistent)
analysis step:  0.02684 MB (temporary)

$\texttt{mpirun -np 4 ./PDAF_offline}$

Allocated memory  (MB)
state and A:  0.00172 MB (persistent)
ensemble array:  0.01112 MB (persistent)
analysis step:  0.01929 MB (temporary)
Impact of the parallelization (2)

Screen output shows some influence of the parallelization

Parallelization – Filter on model PEs:

  Total number of PEs: 4
  Number of parallel model tasks: 1
  PEs for Filter: 4

# PEs per ensemble task and local ensemble sizes:

  Task 1
  #PEs 4
  N 9

At analysis step:

  --- PE-domain 1 dimension of observation vector 8
  --- PE-domain 2 dimension of observation vector 8
  --- PE-domain 3 dimension of observation vector 8
  --- PE-domain 4 dimension of observation vector 4

Note: The output lines might be unordered
Global ESTKF: Files to be changed for parallelization

mod_assimilation.F90
initialize.F90
init_pdaf_offline.F90
init_ens_offline.F90
init_dim_obs_pdaf.F90
obs_op_pdaf.F90
init_obs_pdaf.F90
prodrinva_pdaf.F90
prepoststep_ens_offline.F90

Fortran module
initialization
analysis step
post step
Initialize the model information – we have: \( nx, \ ny, \ dim\_state\_p \)

1. Use additional dimensions from `mod_assimilation`:
   ```fortran
   integer :: dim_state
   integer, allocatable :: local_dims(:)
   ```

2. Rename `dim_state_p` to `dim_state` (global dimension)

3. Allocate `local_dims(npes_model)`

4. Set `dim_state_p` and `local_dims(:)`
   - distribute `dim_state` over number of processes

   ```fortran
   local_dims = FLOOR(REAL(dim_state) / REAL(npes_model))
   DO i = 1, (dim_state - npes_model * local_dims(1))
       local_dims(i) = local_dims(i) + 1
   END DO
   ```

   ```fortran
   dim_state_p = local_dims(mype_model+1)
   ```
Initialize ensemble matrix \texttt{ens\_p}

Simple parallel variant:

1. Initialize global ensemble array (only one process)
2. Distribute sub-states of ensemble array (from the process doing step 1 to all others)

1. Required steps – only for \texttt{mype\_filter==0}
   1. Declare array \texttt{ens} and allocate \texttt{ens} (\texttt{dim\_state, dim\_ens})
   2. Use serial implementation for initialize \texttt{ens} (replace \texttt{ens\_p} by \texttt{ens})
2. Distribute sub-states of ensemble array

For \texttt{mype\_filter}=0

a) Initialize local part of \texttt{ens\_p} directly:

\[
\texttt{ens\_p}(1:dim\_p,1:dim\_ens) = \texttt{ens}(1:dim\_p,1:dim\_ens)
\]

b) Distribute other sub ensembles

\[
\text{DO domain=2, npes\_filter}
\]

allocate \texttt{ens\_p\_tmp}(local\_dims(domain), dim\_ens)

fill \texttt{ens\_p\_tmp} with part of \texttt{ens} for domain

\textit{MPI\_Send} \texttt{ens\_p\_tmp} from process 0 to process ‘domain-1’

deallocate \texttt{ens\_p\_tmp}
2. Distribute sub-states of ensemble array
For all processes with `mype_filter > 0`:

\[ MPI_{\text{Recv}} \text{ens}_p_{\text{tmp}} \text{ into } \text{ens}_p \]

Notes:

- “Classical” MPI communication: `MPI_Send/MPI_Recv`
- See tutorial code for MPI function calls
- Offset in state vector for `mype_filter=k` is
  \[ \text{sum of local_dims}(i) \text{ from } i=1 \text{ to } k \]
- Size of state vector part is `local_dims(k)`
- The example code is not the most efficient possibility:
  Each process could read its own local part of `ens_p`
Operations in case of parallelization:

• Read observation file

• Count number of observations for process-local part of state vector (\texttt{dim\_obs\_p})

• Initialize array \texttt{obs\_p} holding process-local available observations

• Initialize index array telling index of observation point in process-local state vector

Adapt serial implementation for these operations
Count available process-local observations (**dim_obs_p**):

1. Get offset of local part in global state vector

   \[ \text{off}_p = \text{Sum over } \text{local_dims}(i) \text{ up to } i=\text{mype_filter} \]

2. Now count

   \begin{align*}
   \text{cnt} &= 0 \\
   \text{cnt0} &= 0 \\
   \text{DO } j &= 1, \text{nx} \\
   &\quad \text{DO } i=1, \text{ny} \\
   &\quad \quad \text{cnt0} = \text{cnt0} + 1 \\
   &\quad \quad \text{IF } (\text{cnt0}>\text{off}_p \text{ .AND.} \\
   &\quad \quad \quad \text{cnt0}<\text{off}_p+\text{local_dims}(\text{mype_filter}+1)) \text{ THEN} \\
   &\quad \quad \quad \quad \text{IF } (\text{obs_field}(i,j) > -999.0) \text{ cnt} = \text{cnt} + 1 \\
   &\quad \quad \quad \text{END IF; END DO; END DO} \\
   \text{dim_obs}_p &= \text{cnt}
   \end{align*}
init_dim_obs_pdaf.F90 – parallelization (3)

Initialize obs_p and obs_index_p (now process-local parts)

cnt0 = cnt_p = cnt0_p = 0 ! Count grid points
DO j = 1, nx
    DO i= 1, ny
        cnt0 = cnt0 + 1
        IF (cnt0>off_p .AND. &
            cnt0<=off_p+local_dims(mype_filter+1)) THEN
            cnt0_p = cnt0_p + 1
        IF (obs_field(i,j) > -999.0) THEN
            cnt_p = cnt_p + 1
            obs_index_p(cnt_p) = cnt0_p ! Index
            obs_p(cnt_p) = obs_field(i, j) ! observations
        END IF; END IF
    END DO
END DO

PDAF tutorial – Analysis step in offline mode
prepoststep_ens_offline.F90 – parallelization

Post-step routine for the offline mode

Adapt writing of output files for parallelism
    ensemble array ens_p is distributed

To do – inverse operations to init_ens_offline
    • Use temporary array ens_p_tmp
    • For mype_filter>0:
        • MPI_Send ens_p to mype_filter=0
    • For mype_filter=0:
        • Do domain=2, npes_filter
        • MPI_Recv into ens_p_tmp
        • Initialize part of global array ens with ens_p_tmp
        • Write ens into files
Also in the tutorial implementation

- Collect local mean states \((\text{state}_p)\) into a global analysis state and write to file.

- Collect vector of estimated variance \((\text{variance}_p)\) into a global variance vector. Compute estimated RMS error from it.
The analysis step in offline mode with parallelization is fully implemented now.

The implementation allows you now to use the global filters ESTKF, ETKF, and SEIK.

- The parallel implementation can be compiled as before (without an MPI library) and run using 1 process.
- To use the parallelization we have to compile with MPI library (see next slide).
2b) Parallelized local filter
Impact of the parallelization

- Ensemble array is distributed → less memory per process (visible in the memory display at the end of the screen output):

$ mpirun –np 1 ./PDAF_offline –filtertype 7$

Allocated memory (MB)
state and A: 0.01038 MB (persistent)
ensemble array: 0.04449 MB (persistent)
analysis step: 0.01922 MB (temporary)

$ mpirun –np 4 ./PDAF_offline –filtertype 7$

Allocated memory (MB)
state and A: 0.00296 MB (persistent)
ensemble array: 0.01112 MB (persistent)
analysis step: 0.01922 MB (temporary)

Note: Memory for analysis step is not changed!
Impact of the parallelization (2)

Screen output shows some influence of the parallelization

Parallelization - Filter on model PEs:
  Total number of PEs: 4

... 

At analysis step:

--- PE-domain: 0 number of analysis domains: 162
--- PE-domain: 1 number of analysis domains: 162
--- PE-domain: 2 number of analysis domains: 162
--- PE-domain: 3 number of analysis domains: 162

--- PE-Domain: 2 dimension of PE-local full obs. vector 28
--- PE-Domain: 3 dimension of PE-local full obs. vector 28
--- PE-Domain: 0 dimension of PE-local full obs. vector 28
--- PE-Domain: 1 dimension of PE-local full obs. vector 28

Note: The output lines might be unordered

PDAF tutorial – Analysis step in offline mode
Parallelize the local analysis step

Take files from global analysis without parallelization and add the parallelization

Parallelization:

• Perform analysis step using multiple processors
• Split the state vector into equal parts to distribute the work
• Particular for localization:
  
  Take care for local observation regions
  (they can reach into state vector parts of other processes)

Notation for parallelization:

• Suffix \(_p\) marks variables with process-specific values
Local filter LESTKF – parallelization

Files to be parallelized

init_n_domains_pdaf.F90
init_dim_obs_f_pdaf.F90
obs_op_f_pdaf.F90
   init_obs_f_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
prodrinva_l_pdaf
   l2g_state_pdaf.F90

No Changes

Discuss now the files in the order they are called
Routines for initialization!

initialize.F90
init_pdaf_offline.F90
init_ens_offline.F90

These routines are identical for the global and local filters! (Required changes are explained in the part about the parallelization of the global filter)

Only possible difference
• filtertype = 7
in init_pdaf_offline.F90
Routine to set the number of local analysis domains

\texttt{n\_domains\_p}: now the number of local analysis domains for the particular process (according to part of state vector)

To do:
1. Include \texttt{local\_dims} with \texttt{use mod\_assimilation}
2. Set
   \begin{equation}
   n\_domains\_p = \texttt{local\_dims(mype\_filter+1)}
   \end{equation}
init_dim_obs_f_pdaf.F90 – parallelization

Operations in case of parallelization:

- Read observation file
- Count number of observations for process-local part of state vector (dim_obs_p)
- Initialize arrays holding process-local available observations (obs_p) and their coordinates (obs_coords_p)
- Initialize index array (obs_index_p) telling index of a process-local observation in process-local state vector
- Initialize full number of observations (dim_obs_f), vector of observations (obs_f), and coordinates (coords_obs_f)

“FULL” observation vector:
All observations required for all local analyses in process-local part of state vector (Here: Full=All observations for simplicity)

Adapt serial implementation ...
Count process-local observations (dim_obs_p):

1. Include \texttt{dim_obs\_p} with use \texttt{mod\_assimilation}

2. Get offset \texttt{off} of local part in global state vector (see global filter)

3. Now count

\begin{verbatim}
cnt0 = 0; cnt_p = 0
DO j = 1, nx; DO i= 1, ny
    cnt0 = cnt0 + 1
    IF (cnt0>=off_p+1 .AND. cnt0<=off_p+local_dims(mype_filter+1)) THEN
        IF (obs_field(i,j) > -999.0) cnt_p = cnt_p + 1
    END IF; END DO; END DO
\end{verbatim}

\texttt{dim_obs\_p} = \texttt{cnt_p}
Initialize obs_p, obs_index_p, and coords_obs_p

1. Include obs_index_p, coords_obs_f and obs_f with use mod_assimilation
2. Add local arrays for obs_p(:) and coords_obs_p(:, :)
3. Adapt allocates to changed names and size dim_obs_p
4. In the loops rename the variables from _f to _p
5. Adapt the loop initializing the array by adding the check for the index range as for the counting loop

```
cnt0 = 0; cnt_p = 0; cnt0_p = 0
DO j = 1, nx; DO i = 1, ny
  cnt0 = cnt0 + 1
  IF (cnt0>=off_p+1 .AND. cnt0<=off_p+local_dims(mype_filter+1)) THEN
    cnt0_p = cnt0_p + 1
    IF (obs_field(i,j) > -999.0) THEN ... END IF
  END IF; END DO; END DO```

PDAF tutorial – Analysis step in offline mode
Initialize full quantities (dim_obs_f, obs_f, coords_obs_f)

1. Obtain dim_obs_f by calling PDAF_gather_dim_obs_f
2. Allocate obs_f and coords_obs_f
   (deallocate first if already allocated)
3. Obtain obs_f by calling PDAF_gather_obs_f
4. Obtain coords_obs_f by calling PDAF_gather_obs_f
5. Add DEALLOCATE for obs_p and coords_obs_p

**Note:** It is mandatory to call PDAF_gather_dim_obs_f once before using the two other functions because it stores dimension information.

**Note:** The three PDAF functions used here have been added with PDAF Version 1.13 to avoid that the user implementation needs calls to MPI functions.

**Note:** coords_obs_f has to be a REAL array
Implementation of observation operator for full observation domain

Difficulty:

- The state vector \texttt{state\_p} is local to each process
- Full observed vector goes beyond process boundary

Adapt serial version:

1. Initialize process-local observed state
2. Get full observed state vector using \texttt{PDAF\_gather\_obs\_f}
1. Initialize process-local observed state $m_{\text{state\_p}}$
   
a) Include $\text{dim\_obs\_p}$ and $\text{obs\_index\_p}$ with use mod_assimilation

   b) Declare real allocatable array $m_{\text{state\_p}}(:)$

   c) Allocate $m_{\text{state\_p}}(\text{dim\_obs\_p})$

   d) Fill the array

       \[
       \text{DO}\ i = 1, \text{dim\_obs\_p} \\
       \quad m_{\text{state\_p}}(i) = \text{state\_p}(\text{obs\_index\_p}(i)) \\
       \text{END DO}
       \]

   \textbf{Note:}
   In the serial version the upper bound of the loop was $\text{dim\_obs\_f}$ and we filled $m_{\text{state\_f}}$ directly

PDAF tutorial – Analysis step in offline mode
2. Get full observed state vector
   a) Add variable INTEGER :: status
   b) Add call to PDAF_gather_obs_f:

   
   CALL PDAF_gather_obs_f(m_state_p, m_state_f, status)

   c) Deallocate m_state_p

   **Note:** It is mandatory to call PDAF_gather_dim_obs_f once before using the two other functions because it stores dimension information. Usually this was already done in init_dim_obs_f_pdaf
Set the size of the observation vector for the local analysis domain

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

Using the serial implementation, the only change is needed in 1
1. Determine coordinates of local analysis domain
   1. Compute offset:
      \[ \text{off}_\text{p} = \text{Sum over local\_dims(i) up to i=mype\_filter} \]
   2. Compute \text{coords\_l} from \text{nx, ny}:
      \[
      \begin{align*}
      \text{coords\_l}(1) & = \text{real\(\text{ceiling}(\text{real}(\text{domain\_p + off\_p})/\text{real}(\text{ny}))\))} \\
      \text{coords\_l}(2) & = \text{real(\text{domain\_p+off\_p})} - (\text{coords\_l}(1)-1)*\text{real(\text{ny})}
      \end{align*}
      \]

**Note:**
With parallelization the domain numbering begins with 1 for each process. For the coordinates we also need to count the domains from processes with lower process rank.
Compute the product of the inverse observation error covariance matrix with some other matrix
+ apply observation localization (weighting)

Only change:
1. Add `mype_filter==0` to IF-clause in which `verbose=1` is set
   (Only one process should print the information to the screen)
Now, the analysis step for local ESKTF with parallelization in offline mode is fully implemented.

The implementation allows you now to use the local filters LESTKF, LETKF, and LSEIK

Not usable are EnKF and SEEK (PDAF doesn’t have localization for SEEK and a different localization scheme for EnKF)
3) 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 routines handling observations
- Adapt file output:
  - need to read and write restart files from specific model
  - adapt writing of ensemble mean state in `prepoststep_pdaf`
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

- Offline mode of PDAF
- Simple 2D example
- Square root filter ESTKF
  - global and with localization
  - without and with parallelization
- 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