FD4 Manual

User Documentation of the *Four-Dimensional Distributed Dynamic Data structures*.

Version fd4-2014-11-05

Developed at ZIH, TU Dresden, Germany
http://www.tu-dresden.de/zih/clouds
This work was funded by the German Research Foundation (DFG).

Matthias Lieber (matthias.lieber@tu-dresden.de)

Contents

1 Introduction

The ***Four-Dimensional Distributed Dynamic Data structures*** (FD4) is a framework originally developed for the parallelization of spectral bin cloud models and their coupling to atmospheric models. Thus, the data structures are optimized for these kinds of model systems. To use FD4, models must basically meet the following requirements:

- Based on a 3D regular cartesian grid *without* local refinement (i.e. AMR)
- PDE calculations with data dependencies to a limited number of adjacent cells (stencil calculations)

Nevertheless, FD4 can be used for many other applications, especially if at least one of the following points applies:

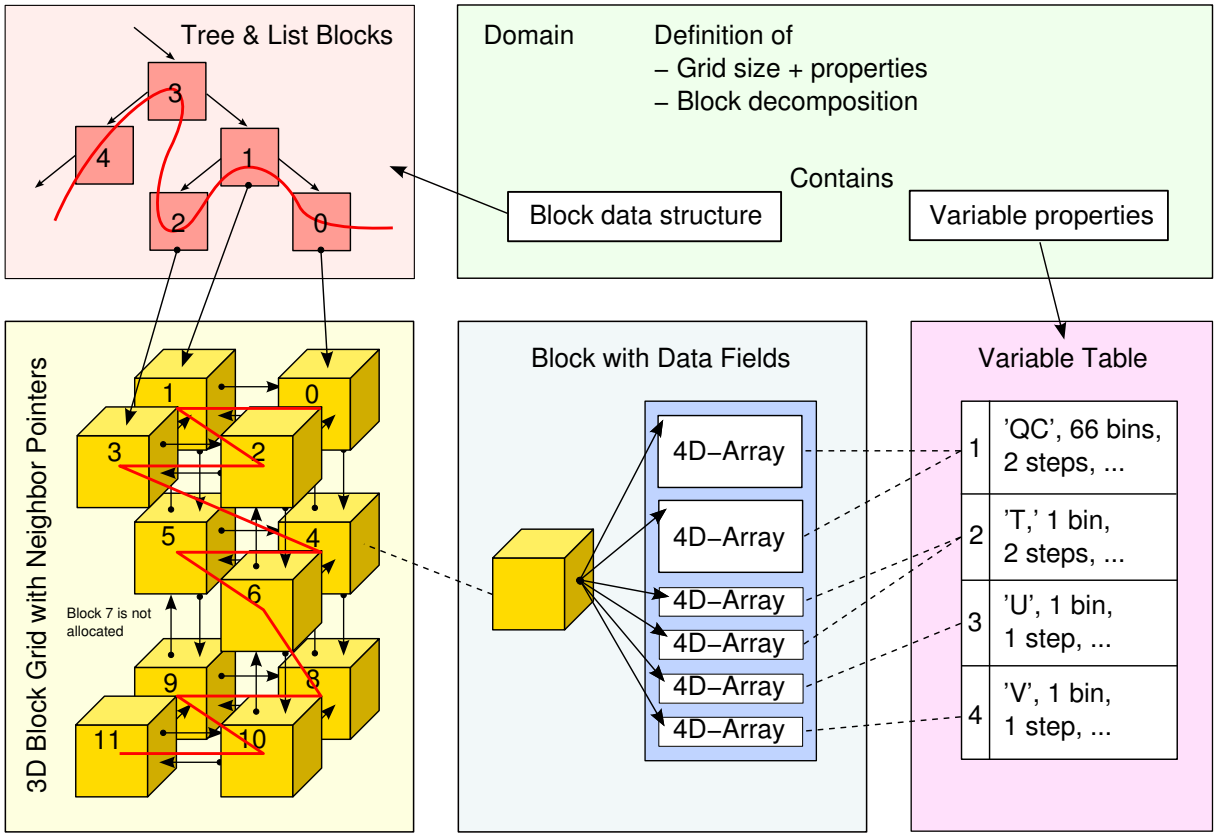
- Many variables per grid cell ($>$100)
- Varying workload per grid cell (varying in time as well as space) which demands dynamic load balancing
- Multiphase model: Additional computations for a limited spatial subset of the grid (drops, clouds, combustion processes, flame fronts, etc.)
- Model system: FD4-based Model coupled to other model(s)

The basic features of FD4 are:

- Open source software (GPL v3)
- Written in Fortran 95
- MPI parallelization (requires MPI-2)
- Block-based decomposition of a regular rectangular numerical grid
- Exchange of ghost cells (i.e. block boundaries, helo zones)
- Optimized for large number of variables per grid cell
- Dynamic load balancing with Hilbert space-filling curves and ParMETIS
- Dynamic adaption of grid allocation status according to spatial structures (multiphase models)
- Coupling interface
- Vis5D and (parallel) NetCDF output
- Scalability to 10 000s of cores

2 Basic Data Structure

FD4 consists of several Fortran 95 modules each providing different data structures and services. The basic data structure is constituted by the ***Variable Table***, the ***Block***, and the ***Domain***:



2.1 Variable Table

The ***Variable Table*** is a user-provided table of all variables that should be managed by FD4. It contains entries for several variable properties:

- The variable's name (character string)
- The discretization type (cell-centered or face-centered to any of the spatial dimensions)
- The number of time steps to allocate for this variable
- The size of a 4th (non-spatial) dimension called bin (originates from the size-resolving bin discretization for detailed cloud models)
- A default value ("null")
- An optional threshold value, to indicate separated phases in multiphase models and allow adaptive block allocation

The index of the variable in the table is called ***Variable Index*** and is used as identifier. All variables are floating point variables of the same kind (single or double precision). Integer or other types are not provided.

2.2 Block

Based on the *Variable Table*, FD4 allocates the arrays holding the variables in each ***Block***. The *Blocks* provide a 3D decomposition of the grid. *Blocks* are allowed to be of different size. The block decomposition is defined by one vector of block start indexes for each dimension, or, for convenience, by specifying a number of blocks for each dimension.

The *Blocks* are contained in two data structures:

- ***Block Tree***: A self-balancing binary tree (red-black tree) which provides logarithmic complexity for access to arbitrary *Blocks*. For fast iteration, this tree is combined with a linked list. The index of a *Block* in the *Block Tree* is derived from its position in the global grid by fast bit shifting operations.
- ***Neighbor Pointers***: To access ***Neighbor Blocks***, which is required for any kind of stencil computations, each *Block* contains pointers to its 6 *Neighbor Blocks*.

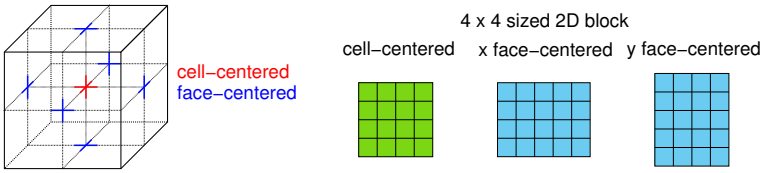
Note that not all *Blocks* may be present at a time: In a parallel run (which is the intended use of FD4!), the *Blocks* are distributed to the processes. For more details about parallelization, refer to Section 3. Additionally, when running in ***Adaptive Block Mode***, only a specific subset of the *Blocks* are present globally, refer to Section 2.7. Thus, a *Neighbor Block* may be: locally present, on a remote process, or not present on any process.

2.3 Domain and Iterator

The ***Domain*** is the central object in FD4. It contains all data to describe the numerical grid and the data structure of the allocated *Blocks*. The ***Iterator*** object allows iterating through the local list of *Blocks* associated with the *Domain* and offers subroutines to access ghost cells, see Section 2.6.

2.4 Cell-centered and Face-centered Variables

Cell-centered variables are located in the center of a 3D grid cell, whereas face-centered variables are centered on the grid cell's surfaces that correspond a specified spatial dimension. Thus, three types of face-centered variables are possible. Note, that the grid for face variables is extended by one in the face dimension - for the global domain as well as for each *Block*:



Consequently, two adjacent *Blocks* share copies of the same face variable at their boundary. This has consequences regarding consistency, see Section 5.14. The actual data arrays are allocated starting at index 1 for each dimension (block-local indexes).

One feature of FD4 is that the data arrays are allocated without ghost cells (helo zones), which saves memory when small *Blocks* are used.

2.5 Accessing Variable Arrays

The variables are allocated in the *Blocks* as one 4D array per discretization type (cell-centered, x-face, y-face, z-face). The variables, their time steps, and their bins are mapped on the first

dimension. The three other dimensions are used for the spatial indexes.

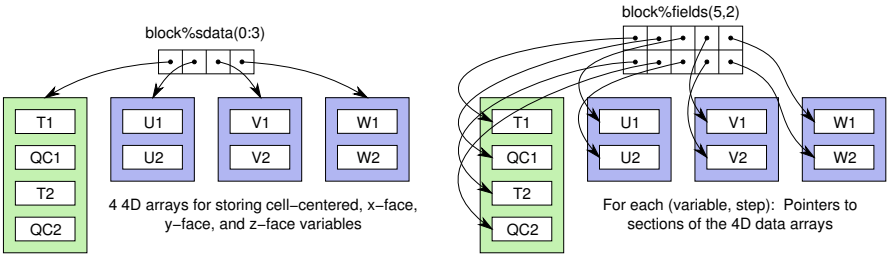
A specific variable item of one *Block* is accessed as `block%sdata(f)%l(b,x,y,z)` with

- the face variable indicator `f` (0 for cell-centered, 1–3 for face-centered in x, y, z respectively)
- the variable, time steps, and bins encoded to `b`
- the block-local spatial indexes `x`, `y`, `z`

Since this not straightforward, an array of pointers for variables and their time steps pointing to the corresponding sections in the actual data arrays is provided. The access is then via `block%fields(idx,st)%l(b,x,y,z)` with

- the variable index `idx` as defined by the *Variable Table*
- the time step index `st` (starting at 1)
- the bin `b` (1 for non-4D variables)
- the block-local spatial indexes `x`, `y`, `z`

This figure illustrates an example for the data structures:

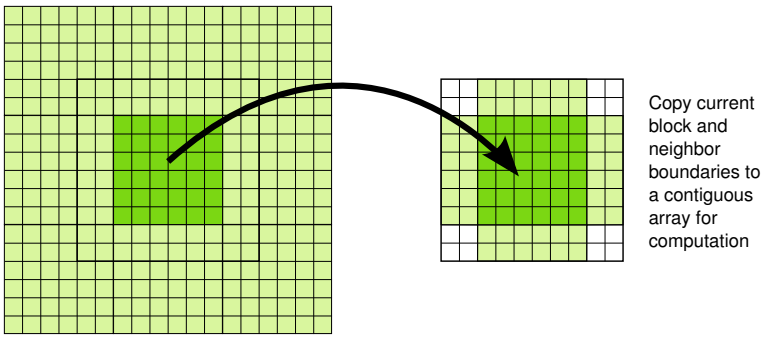


Using the `block%fields` array, only grid cells of the local block can be accessed, but not grid cells of *Neighbor Blocks* (ghost cells).

2.6 Accessing Variable Arrays with Ghosts

The ***Iterator*** object contains the subroutine `fd4_iter_get_ghost` to access variables of the current *Block* including the boundaries of the 6 *Neighbor Blocks* (ghost cells). The variables are copied to a buffer array, which can than be used for stencil computations. The number of ghost cell rows is defined for each dimension when creating the domain. It is the same for all cell-centered variables. Access to face-centered variables of *Neighbor Blocks* is not implemented.

This figure shows an example in 2D:



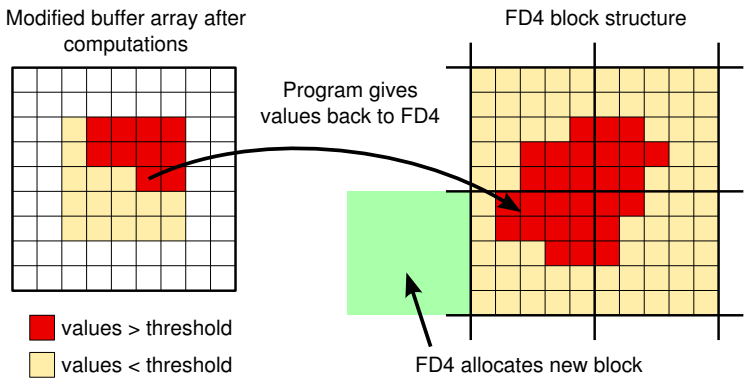
Note, that only data of the 6 direct *Neighbor Blocks* (in 3D) are copied, not the data of the diagonal *Neighbor Blocks*. The resulting values in the area of the ghost cells depend on the state of each *Neighbor Block*:

- Neighbor is locally present: Data are copied directly from the *Neighbor Block* to the buffer array.

- Neighbor is present on a remote process: Data are copied from the ***Ghost Block*** - a copy of the remote *Block*'s boundary - to the buffer array. See Section 3.1.
- Neighbor is not present on any process: The corresponding section of the buffer array is filled with the default value of the variable(s).

2.7 Adaptive Block Mode

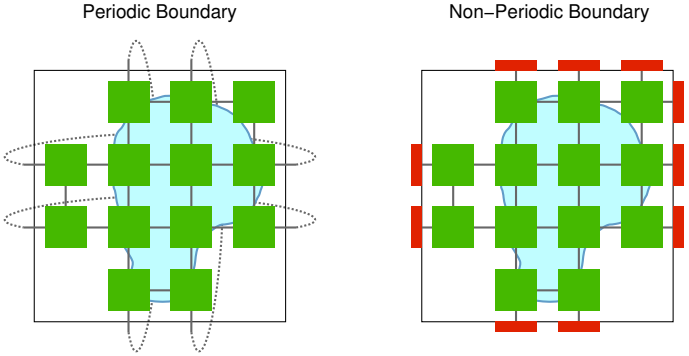
FD4 allows the dynamic adaption of the block allocation to spatial structures. It is useful for special multiphase applications when neither computations nor data are required for certain regions of the spatial grid. In this case, memory can be saved by not allocating the unused (***empty***) blocks. This mode, the so-called ***Adaptive Block Mode***, is only enabled if any of the variables in the *Variable Table* are threshold-variables, i.e. these variables have a threshold value. A *Block* is considered *empty* if in all its grid cells the values of all threshold-variables are less or equal than their corresponding threshold value. Based on this definition, FD4 decides which blocks to deallocate from the global block structure. Additionally, FD4 ensures that appropriate data are provided for the numerical stencil around non-*empty* cells. This mechanism also triggers the allocation of new blocks:



The actual block adaption (allocation of new *Blocks*, deallocation of unused *Blocks*) is carried out in the dynamic load balancing routine, see Section 3.2.

2.8 Boundary Conditions

Periodic boundary conditions are implemented straightforward in FD4 by periodic *Neighbor Pointers*. For non-periodic boundary conditions, ***Boundary Ghost Blocks*** are added for *Blocks* at the domain boundary. The *Boundary Ghost Blocks* have to be filled by the user, except for zero gradient boundary conditions, which are implemented in FD4. This figure shows the concept for periodic (left) and non-periodic (right) boundary conditions for an exemplary 2D domain (in *Adaptive Block Mode*):



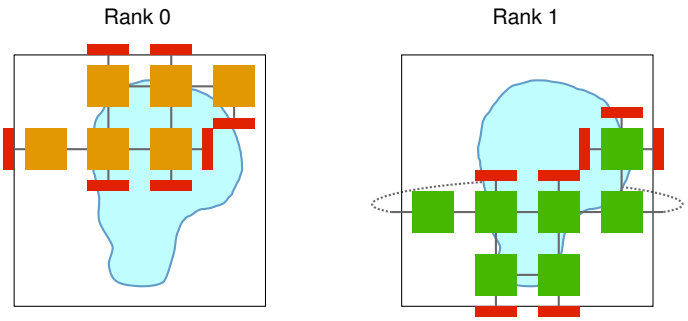
3 Parallelization and Coupling

Parallelization of the FD4 grid is achieved by distributing the *Blocks* to the processes. Consequently, the total number of *Blocks* should be greater or equal than the number of processes.

3.1 Ghost Communication

Before performing stencil computations in parallel runs (which require the boundary of *Neighbor Blocks*, see Section 2.6), the boundaries have to be transferred between the processes. So-called ***Communication Ghost Blocks*** are allocated at process borders in the block decomposition to store the boundary of remote *Neighbor Blocks*. The ***Ghost Communicator*** object handles the update of the *Communication Ghost Blocks*. The *Ghost Communicator* is created for a specified set of variables and respective time steps and can be executed whenever necessary. Optionally, it is possible to restrict the spatial dimensions of the ghost exchange to one or two specified dimensions. The number of ghost cells that are exchanged is fixed for the domain. The ghost communication is only possible for cell-centered variables and not for face-centered variables.

This figure shows an exemplary block decomposition for two processes and the *Communication Ghost Blocks*:



3.2 Dynamic Load Balancing

The dynamic load balancing in FD4 performs 3 major steps:

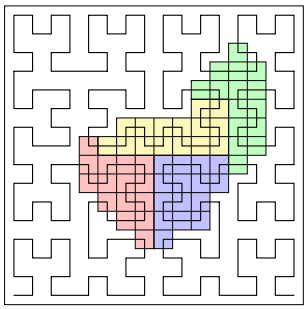
1. Determine if load balancing is necessary.
2. Calculate a new partitioning, i.e. mapping of *Blocks* to processes.
3. Migration and (De)allocation of *Blocks*.

Basically there are two situations for which load balancing is necessary: Firstly, when running in *Adaptive Block Mode*, *Blocks* may be added or removed from the global domain, which requires a new mapping of *Blocks* to processes. Secondly, if the workload of the *Blocks* changes non-uniformly, the load balance of the processes declines and more time is lost at synchronization points of the program. Of course, both reasons may also appear at the same time.

The workload of the *Blocks* is described by the ***Block Weight***. The default value is the number of grid cells of the *Block*. If the workload does not exclusively depend on the number of grid cells, the *Block Weight* should be set to the actual computation time for each *Block*. If no *Blocks* were added or removed from the global domain, the decision whether load balancing

is necessary or not depends on the load balance of the last time step (based on the *Block Weight*) and a specified load balance tolerance. Thus, it is possible to control how sensitive FD4 should react on emerging load imbalances. Instead of specifying a fixed tolerance, FD4 can also automatically decide whether load balancing is beneficial or not. FD4 weighs the time lost due to imbalance against the time required for load balancing. This ***Auto Mode*** requires that the *Block Weights* are set to the computation time in microseconds since the last call to the load balancing subroutine.

Two different methods for the calculation of the new partitioning are implemented in FD4: A graph-based approach using the ParMETIS library and a geometric approach using the Hilbert space-filling curve (SFC). Both methods are incremental, which means that the difference of successive partitionings is low to reduce migration costs. SFC partitioning is preferred since it executes much faster compared to ParMETIS. This figure shows a 2D Hilbert SFC and an exemplary partitioning derived from the curve:

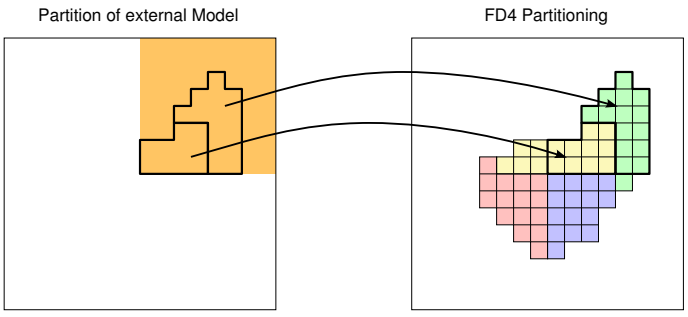


3.3 Coupling

FD4 allows coupling models based on FD4 to external models, i.e. transferring variables between these models. The coupling interface has the following assumptions:

- Sequential coupling: Both models (FD4-based and external) work on the same set of processes and all processes perform computations for these models alternately.
- Same grid structure: Both models have the same grid structure, or at least the external model provides its coupling data matching the grid used in FD4.
- Block-based partitioning: The partitioning of the external model is based on rectangular blocks, but may be different from the partitioning in FD4.

The ***Couple Context*** is the description of the ***Couple Arrays***, the data fields of the external model. Among other specifications, the position of each *Couple Array* in the global grid, the process owning this array, and the matching FD4 variable must be provided. Based on this description, FD4 computes the overlaps of each provided *Couple Array* with the *Blocks* and transmits the variables directly between the processes. FD4 is able to communicate coupling data in both directions: The ***Put*** operation sends variables from the external model to FD4 whereas ***Get*** sends variables from FD4 to the external model. This figure shows a *Put* operation from one single partition of an external model to the matching FD4 *Blocks*:



In this example, two messages are sent, if none of the two receiving FD4 partitions belongs to the sender process of the external model. If the sender owns a receiving partition in FD4, the corresponding data is copied locally without sending a message.

The *Couple Context* concept allows coupling multiple external models to multiple FD4-based models. However, the direct coupling between two models based on FD4 is not implemented.

4 Building the FD4 Library

4.1 Prerequisites

Compiling and running FD4 requires:

- Unix or Linux system
- GNU make
- C and Fortran 95 compilers
- An MPI-2 implementation (for example Open MPI or MPICH2)

FD4 has been tested with the following compilers: GCC/GNU Fortran, GCC/G95, Intel, IBM, PathScale, PGI, Solaris Studio, GCC/NAG.

Optional features of FD4 require additional external packages:

- The NetCDF library is required for NetCDF output. Parallel output is available with NetCDF4 only (if compiled with parallel HDF5). Serial output is possible with both NetCDF3 and NetCDF4.
  Website: `http://www.unidata.ucar.edu/software/netcdf/`
- Compiled sources of Vis5D+ are required to write output to Vis5D files.
  Website: `http://vis5d.sourceforge.net`
- ParMETIS is required for graph-based dynamic load balancing. The built-in SFC load balancing has proven to be much more scalable, so there is actually no need to build FD4 with ParMETIS support.
  Website: `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview/`

To visualize data from NetCDF files you can use tools such as Visit (`https://wci.llnl.gov/codes/visit/`) or Ncview (`http://meteora.ucsd.edu/~pierce/ncview_home_page.html`).

4.2 Configuration

- Create a configuration file for your system in the directory `config/`.
- You can use conf.default as a starting point or the other config files specific to some compilers.
- Name the config file `conf.<NAME>` or just overwrite `conf.default`.
- Optionally edit `config/fd4flags.in` to set some configuration flags for FD4, most notably `FD4_VERBOSE_LEVEL` (level 3 enables expensive runtime checks and has performance impact!).

4.3 Compiling FD4

- Type `make conf=<NAME>`.
  - You need GNU make; this may require calling `gmake` instead of `make` on certain systems.
  - If you name your config file `conf.default`, you can just type `make`.

- You can use parallel make to speed up the build process by calling, e.g., `make -j 4 conf=<NAME>`.
- This should build the library `libfd4.a`.
- To create multiple builds of FD4, use `./mkbuilddir.sh <DIR>` to create a new build directory (with its own configuration) and call make from there.

5 An FD4 Tutorial

This chapter shows the user interface subroutines of FD4 by means of small examples. The example programs are contained in the FD4 package in the directory `tutorial`. They are numbered in the same order as the following sections. The complete FD4 API documentation of the user routines can be found in the FD4 package in `doc/index.html`.

5.1 Basics: 01_basics.F90

Include the module `fd4_mod` to your Fortran 95 source to make the FD4 interface available. FD4 defines kind type parameters for integer and real variables in `util/kinds.F90`:

Name	Data type	Remarks
`i4k`	4 byte integer	
`i8k`	8 byte integer	
`i_k`	4 byte integer	default integer type in FD4
`r4k`	4 byte real	
`r8k`	8 byte real	
`r_k`	8 byte real	type for grid variables, can be changed to `r4k`

One of the basic utility functions is `gettime`, which returns the microseconds since 1970 as an 8 byte integer. It can be used to clock parts of the program.

```
program fd4_demo_basics

  use fd4_mod
  implicit none

  integer(i8k) :: time_now_us, t0, t1
  real(r8k)    :: time_now_s

  ! initialize the timing routines
  ! (normally, this routine is called when creating an FD4 domain)
  call gettime_init()

  ! gettime is part of the FD4 utilities, calls C system function gettimeofday
  call gettime(time_now_us)
  time_now_s = real(time_now_us,r8k) / 1.e6_r8k

  write(*,'(A,F16.4)') 'seconds since 1970: ', time_now_s

  call gettime(t0)
  call gettime(t1)
  do while(t1<=t0)
    call gettime(t1)
  end do
  write(*,'(A,I7,A)') '       gettime resolution: ',t1-t0,' us'

  ! FD4 also supports using getrusage (ru_utime) as timer
  ! (this is the time the process has run in user mode)
  call gettime_rusage(t0)
  call gettime_rusage(t1)
  do while(t1<=t0)
    call gettime_rusage(t1)
  end do
  write(*,'(A,I7,A)') 'gettime_rusage resolution: ',t1-t0,' us'

  ! If compiled with PAPI, FD4 can make use of PAPIF_get_virt_cyc
  ! (this is the time the process has run in user mode)
```

```
  call gettime_papi(t0)
  if(t0/=TIMING_MOD_NO_PAPI .and. t0>0) then
    call gettime_papi(t0)
    call gettime_papi(t1)
    do while(t1<=t0)
      call gettime_papi(t1)
    end do
    write(*,'(A,I7,A)') '  gettime_papi resolution: ',t1-t0,' us'
  end if

end program fd4_demo_basics
```

5.2 Variable Table Definition: 02_vartab.F90

Each variable that should be managed by FD4 is defined by an entry in the *Variable Table*, an array of type `fd4_vartab`. See Section 2.1 for a description.

```
program fd4_demo_vartab

  use fd4_mod
  implicit none

  ! these paramters are the indexes of the variables in the variable table
  integer, parameter :: varTmp = 1, varRho = 2, varQC = 3, varU = 4, varV = 5
  integer, parameter :: number_of_variables = 5
  ! this is the variable table
  type(fd4_vartab) :: vartab(number_of_variables)

  ! fill the variable table for varTemp
  vartab(varTmp)%name     = 'Temperature' ! name of the variable, at most 64 characters
  vartab(varTmp)%nbins    = 1             ! number of bins = size of the 4th dimension
  vartab(varTmp)%nsteps   = 2             ! number of time steps to allocate
  vartab(varTmp)%dynamic  = .false.       ! currently unused
  vartab(varTmp)%vnull    = 0.0_r8k       ! initial/default value
  vartab(varTmp)%vthres   = FD4_NOTHRES   ! threshold value or FD4_NOTHRES
  vartab(varTmp)%facevar  = FD4_CELLC     ! discretization type

  ! since the type fd4_vartab has default values for all components but the name,
  ! you can left out some definitions
  vartab(varRho)%name     = 'Densitiy'
  vartab(varRho)%nsteps   = 2

  ! but the clearest method is to use the derived type constructors and
  ! arrange them as table with one variable per row
  !                                name, nb, st,  unused, ini,      vthres,  discret.
  vartab(varQC) = fd4_vartab('Droplets', 12,  2, .false., 0.0,         0.0, FD4_CELLC )
  vartab(varU)  = fd4_vartab(  'u Wind',  1,  1, .false., 0.0, FD4_NOTHRES, FD4_FACEX )
  vartab(varV)  = fd4_vartab(  'v Wind',  1,  1, .false., 0.0, FD4_NOTHRES, FD4_FACEY )

  write(*,'(5(A24,I4,I4,L3,E11.3,E11.3,I3,/))') vartab

end program fd4_demo_vartab
```

5.3 Domain Creation: 03_domain.F90

The *Domain* is described by the derived type `fd4_domain`. Note, that the type `fd4_domain` *must* be declared with the `target` attribute, though compilers will also accept code without the attribute. A *Domain* is created by calling `fd4_domain_create`, which needs the following inputs:

- Number of *Blocks* in x, y, z
- Lower and upper bounds of the grid in x, y, z
- *Variable Table*
- Number of ghost cells in x, y, z

- Periodic boundary conditions in x, y, z
- MPI communicator

Creating a *Domain* does not allocate any *Blocks*. Use `fd4_util_allocate_all_blocks` to allocate all *Blocks* balanced over all processes. The subroutine `fd4_domain_delete` removes all *Blocks* and frees all memory associated with the *Domain*. FD4 collects some internal statistics that are printed to stdout when calling `fd4_domain_delete`. You can also use `fd4_domain_dump_stats` to print the statistics at any place in the program. The statistics will contain more entries when you use ghost communication and coupling.

```
program fd4_demo_domain

  use fd4_mod
  implicit none
  include 'mpif.h'

  ! FD4 variable table
  integer, parameter :: varTmp = 1, varRho = 2, varQC = 3, varU = 4, varV = 5
  integer, parameter :: number_of_variables = 5
  type(fd4_vartab) :: vartab(number_of_variables)
  ! FD4 domain
  integer :: dsize(3,2), bnum(3), nghosts(3)
  logical :: periodic(3)
  type(fd4_domain), target :: domain
  ! misc
  integer :: rank, err

  !! Create the variable table
  !                                    name, nb, st,  unused, ini,      vthres,  discret.
  vartab(varTmp) = fd4_vartab('Temperature',  1,  2, .false., 0.0, FD4_NOTHRES, FD4_CELLC )
  vartab(varRho) = fd4_vartab(   'Densitiy',  1,  2, .false., 0.0, FD4_NOTHRES, FD4_CELLC )
  vartab(varQC)  = fd4_vartab(   'Droplets', 12,  2, .false., 0.0, FD4_NOTHRES, FD4_CELLC )
  vartab(varU)   = fd4_vartab(     'u Wind',  1,  1, .false., 0.0, FD4_NOTHRES, FD4_FACEX )
  vartab(varV)   = fd4_vartab(     'v Wind',  1,  1, .false., 0.0, FD4_NOTHRES, FD4_FACEY )

  !! MPI Initialization
  call MPI_Init(err)
  call MPI_Comm_rank(MPI_COMM_WORLD,rank ,err)

  !! Create the FD4 domain
  dsize(1:3,1) = (/  1,  1,  1/)     ! grid start indices
  dsize(1:3,2) = (/ 16, 16, 16/)     ! grid end indices
  bnum(1:3) =    (/  4,  4,  4/)     ! number of blocks in each dimension
  nghosts(1:3) = (/  2,  2,  2/)     ! number of ghost cells in each dimension
  periodic(1:3) = .true.             ! periodic boundaries
  call fd4_domain_create(domain, bnum, dsize, vartab, nghosts, periodic, MPI_COMM_WORLD, err)
  if(err/=0) then
    write(*,*) 'fd4_domain_create failed'
    call MPI_Abort(MPI_COMM_WORLD, 1, err)
  end if

  if(rank==0) write(*,'(A,I5)') 'number of allocated blocks: ', domain%blockcount

  !! Allocate the blocks
  call fd4_util_allocate_all_blocks(domain, err)

  if(rank==0) write(*,'(A,I5)') 'number of allocated blocks: ', domain%blockcount

  !! Delete the domain and finalize MPI
  call fd4_domain_delete(domain)
  call MPI_Finalize(err)

end program fd4_demo_domain
```

If you have compiled FD4 with `FD4_VERBOSE_LEVEL` 2 or higher, FD4 should print something like this:

```
[FD4:0000] created new fd4_domain:
[FD4:0000] dim   start     end  blocks   blksz  ghosts  per.bd
[FD4:0000]   x       1      16       4    4.00       2       T
[FD4:0000]   y       1      16       4    4.00       2       T
[FD4:0000]   z       1      16       4    4.00       2       T
[FD4:0000] max. number of blocks:   64
```

```
[FD4:0000] Hilbert SFC level:       2
[FD4:0000] number of MPI processes: 2
[FD4:0000] block pool lists:        4
[FD4:0000] block pool max. size:    152
[FD4:0000] adaptive block mode:     F
[FD4:0000] variable table:
[FD4:0000]  id  nbins  nsteps  dyn     vnull  threshld  face  name
[FD4:0000]   1      1       2    F   0.0E+00         -     -  Temperature
[FD4:0000]   2      1       2    F   0.0E+00         -     -  Densitiy
[FD4:0000]   3     12       2    F   0.0E+00         -     -  Droplets
[FD4:0000]   4      1       1    F   0.0E+00         -     x  u Wind
[FD4:0000]   5      1       1    F   0.0E+00         -     y  v Wind
number of allocated blocks:     0
number of allocated blocks:    64
[FD4:0000] FD4 Statistics               min          max          avg                sum
[FD4:0000] Balance check us              66          606          336                672
[FD4:0000] Balance part  us             143          143          143                286
[FD4:0000] Balance mig   us            2838         2946         2892               5784
[FD4:0000] Balance recv Blocks            0            0            0                  0
[FD4:0000] Balance recv Bytes             0            0            0                  0
[FD4:0000] Domain block alloc            32           32           32                 64
[FD4:0000] Domain block free              0            0            0                  0
[FD4:0000] Domain ghost alloc            32           32           32                 64
[FD4:0000] Domain ghost free              0            0            0                  0
```

5.4 Block Iteration: 04_iterator.F90

To access the *Blocks* a process owns, FD4 provides an *Iterator*, which iterates over all *Blocks* of the *Domain* in unspecified order. To read or write the data fields of a *Block*, use the `block%fields(idx,st)%l(b,x,y,z)` approach as described in Section 2.5. Note that you cannot access the ghost cells in this way, see Section 5.5. A *Block* iteration loop looks as follows:

```
  !! Loop over all blocks of the domain and initialize the temperature
  call fd4_iter_init(domain, iter)
  do while(associated(iter%cur))
    write(*,'(A,I4,A,3(I3))') 'rank ',rank,' iterates to block at (x, y, z) ',iter%cur%pos
    ! get offset from domain indexes to block-local indexes
    call fd4_iter_offset(iter, offset)
    ! loop over block's grid cells
    do z=1,iter%cur%ext(3)
      do y=1,iter%cur%ext(2)
        do x=1,iter%cur%ext(1)
          ! get z coordinate of this grid cell in global coordinates
          gz = offset(3) + z
          ! set temperature depending on global z coordinate
          iter%cur%fields(varTmp,1)%l(1,x,y,z) = 295.0 + f * REAL(gz)
        end do
      end do
    end do
    ! go to next block
    call fd4_iter_next(iter)
  end do
```

5.5 Ghost Cells: 05_ghosts.F90

To get variables from a *Block* with ghost cells from the six *Neighbor Blocks*, use the subroutine `fd4_iter_get_ghost`. It copies the current *Block*'s data and the boundary of *Neighbor Blocks* to a 4D buffer array. See Section 2.6 for more details about accessing ghost cells. The buffer array must be large enough to hold the spatial bounds of the *Block* and the 4th dimension of the variables. Note that the 4th dimension is in fact the 0th dimension: it comes first. To get the bounds of the largest *Block* in the *Domain*, use the subroutine `fd4_domain_max_bext`. This

example shows how to allocate a buffer array for a variable with a 4th dimension and how to read the ghost cells:

```
  !! Allocate the buffer array for a single block with ghost cells
  ! get the max block extent (bext) including ghost cells
  call fd4_domain_max_bext(domain, bext(1:3), .true.)
  bext(0) = vartab(varQC)%nbins ! 4th dimension
  ! allocate the buffer array with interior grid cells starting at 1
  allocate( buffer(bext(0),-1:bext(1)-2,-1:bext(2)-2,-1:bext(3)-2) )
  buffer = 0.0_r_k

  !! Initialize time step indicators
  now = 1
  new = 2

  !! Loop over all blocks of the domain and do some sort of computations
  call fd4_iter_init(domain, iter)
  do while(associated(iter%cur))
    ! get droplets with ghost cells from current block at time step 'now'
    call fd4_iter_get_ghost(iter, varQC, now, bext, buffer)
    ! loop over block's grid cells
    do z=1,iter%cur%ext(3)
      do y=1,iter%cur%ext(2)
        do x=1,iter%cur%ext(1)
          ! do some stencil computations, update 'new' values
          iter%cur%fields(varQC,new)%l(:,x,y,z) = buffer(:,x,y,z) + &
            ( f(-2) * buffer(:,x-2,y,z) + f(-1) * buffer(:,x-1,y,z) + f(0) * buffer(:,x,y,z) &
            + f( 1) * buffer(:,x+1,y,z) + f( 2) * buffer(:,x+1,y,z) ) * dt
          ! ...
        end do
      end do
    end do
    call fd4_iter_next(iter)
  end do
```

5.6 Ghost Data Exchange: 06_heat.F90

Three functions are required to perform ghost communication (3.1): `fd4_ghostcomm_create`, `fd4_ghostcomm_exch`, and `fd4_ghostcomm_delete`.

Here is a complete demo application. It solves the heat conduction equation in 3D.

```
program fd4_demo_heat

  use fd4_mod
  implicit none
  include 'mpif.h'

  !! Setup parameters
  real, parameter :: radius = 0.5            ! rel. radius of initial heat bubble
  integer, parameter :: grid(3) = 32         ! number of grid cells for x, y, z
  real(r_k), parameter :: ds(3) = 1.0        ! grid cell size for x, y, z
  real(r_k), parameter :: dt = 0.1           ! time step size
  integer, parameter :: nsteps = 1000        ! number of time steps to compute

  ! FD4 variable table
  type(fd4_vartab) :: vartab(1)
  integer, parameter :: THETA = 1
  ! FD4 domain
  type(fd4_domain), target :: domain
  integer :: dsize(3,2), bnum(3), nghosts(3)
  logical :: periodic(3)
  ! FD4 iterator
  type(fd4_iter) :: iter
  ! FD4 ghost communication
  type(fd4_ghostcomm) :: ghostcomm(2)
  ! misc
  integer :: rank, err, bext(0:3)
  integer :: offset(3), x, y, z, now, new, step
  real(r_k), allocatable :: buf(:,:,:,:)
  real(r_k) :: dtheta
  real :: global_pos(3), cr
```

```
!! MPI Initialization
call MPI_Init(err)
call MPI_Comm_rank(MPI_COMM_WORLD,rank ,err)

!! Create the FD4 variable table
!! (only one cell-centered variable 'theta' with 2 time steps)
!                            name, nb, st,  unused, ini,      vthres,  discret.
vartab(THETA) = fd4_vartab('theta',  1,  2, .false., 0.0, FD4_NOTHRES, FD4_CELLC )

!! Create the FD4 domain
dsize(1:3,1) = (/1, 1, 1/)     ! grid start indices
dsize(1:3,2) = grid(1:3)       ! grid end   indices
bnum(1:3) = grid(1:3) / 4      ! number of blocks in each dimension
nghosts(1:3) = (/1, 1, 1/)     ! number of ghost cells in each dimension
periodic(1:3) = .true.         ! periodic boundaries
call fd4_domain_create(domain, bnum, dsize, vartab, nghosts, periodic, MPI_COMM_WORLD, err)
if(err/=0) then
  write(*,*) rank, ': fd4_domain_create failed'
  call MPI_Abort(MPI_COMM_WORLD, 1, err)
end if

!! Allocate the blocks of the domain
call fd4_util_allocate_all_blocks(domain, err)

!! Allocate the buffer array for a single block with ghost cells
call fd4_domain_max_bext(domain, bext(1:3), .true.)
bext(0) = 1 ! 4th dimension not used here
allocate( buf(bext(0),0:bext(1)-1,0:bext(2)-1,0:bext(3)-1) )
buf = 0.0_r_k

!! Initialize time step indicators
now = 1
new = 2

!! Initialize theta with a spherical heat bubble
call fd4_iter_init(domain, iter)
do while(associated(iter%cur))
  ! offset from domain indexes to block-local indexes
  call fd4_iter_offset(iter, offset)
  ! loop over block's grid cells
  do z=1,iter%cur%ext(3)
    do y=1,iter%cur%ext(2)
      do x=1,iter%cur%ext(1)
        ! get global coordinates of this grid cell and scale to [0,1]
        global_pos(1:3) = REAL( offset(1:3) + (/x,y,z/) - 1 ) / REAL(dsize(1:3,2) - 1)
        ! distance from domain center to current grid cell
        cr = sqrt( (global_pos(1)-0.5)**2+(global_pos(2)-0.5)**2+(global_pos(3)-0.5)**2 )
        if(cr < radius) then
          iter%cur%fields(THETA,now)%l(1,x,y,z) = 2.0_r_k * cos(3.14159*cr/(2*radius))
        end if
      end do
    end do
  end do
  call fd4_iter_next(iter)
end do

!! Create ghost communicator for variable THETA (one for each time level)
call fd4_ghostcomm_create(ghostcomm(1), domain, 1, (/THETA/), (/1/), err)
call fd4_ghostcomm_create(ghostcomm(2), domain, 1, (/THETA/), (/2/), err)

!! Time stepping loop
do step=1,nsteps

  ! exchange ghost cells for time level 'now'
  call fd4_ghostcomm_exch(ghostcomm(now), err)

  ! iterate over all local blocks
  call fd4_iter_init(domain, iter)
  do while(associated(iter%cur))
    ! get theta with ghost cells from current block
    call fd4_iter_get_ghost(iter, THETA, now, bext, buf)
    ! loop over block's grid cells
    do z=1,iter%cur%ext(3)
      do y=1,iter%cur%ext(2)
        do x=1,iter%cur%ext(1)
```

```
            !                              ( d2T   d2T   d2T )
            !  theta_new =  theta_now + ( --- + --- + --- ) * dt
            !                              ( dx2   dy2   dz2 )
            ! calculate dtheta
            dtheta = ( buf(1,x-1,y,z) + buf(1,x+1,y,z) - 2*buf(1,x,y,z) ) / (ds(1) * ds(1)) &
                   + ( buf(1,x,y-1,z) + buf(1,x,y+1,z) - 2*buf(1,x,y,z) ) / (ds(2) * ds(2)) &
                   + ( buf(1,x,y,z-1) + buf(1,x,y,z+1) - 2*buf(1,x,y,z) ) / (ds(3) * ds(3))
            ! set updated theta value
            iter%cur%fields(THETA,new)%l(1,x,y,z) = buf(1,x,y,z) + dtheta * dt
          end do
        end do
      end do
      call fd4_iter_next(iter)
    end do

    if(rank==0 .and. mod(step,100)==0) write(*,'(A,I5)') 'step ',step

    ! swap time step indicators
    now = 3 - now
    new = 3 - new

  end do

  !! Delete the ghost communicator and the domain, finalize MPI
  call fd4_ghostcomm_delete(ghostcomm(1))
  call fd4_ghostcomm_delete(ghostcomm(2))
  call fd4_domain_delete(domain)
  call MPI_Finalize(err)

end program fd4_demo_heat
```

5.7 Vis5D Output: 07_heat_v5d.F90

Simple output of grid data to Vis5D files is supported in FD4. The work sequence is quiet simple: *open* - *write* - *write* - ... - *close* where each write call writes data from a different time step of the simulation. In comparison to FD4's NetCDF output, there are two limitations which originate from Vis5D's simple interface: Only one of such work sequences can be active at any time during the whole program run and when opening the Vis5D file you must already know how many write calls you will issue.

To add Vis5D output to the demo application in 5.6, three code snippets need to be added to the code. The first snippet defines the output file's name, the number of write calls, and the variables with their corresponding time steps. It must be called once before the time stepping loop. `fd4_vis5d_open` has additional optional parameters to define grid cell size and map projection, see the API documentation. After this, the first write call writes initial data to the file.

```
  !! Initialize Vis5D output and write initial data
  call fd4_vis5d_open(domain, 'out.v5d', nsteps/100+1, 1, (/THETA/), (/now/), err)
  call fd4_vis5d_write(err)
```

The following snippet writes data during time stepping to the Vis5D file (every 100th step only). The optional parameter `st_opt` is used to tell FD4 to write the data of the current step.

```
    !! Write Vis5D output
    if(mod(step,100)==0) then
      if(rank==0) write(*,'(A,I5)') 'step ',step
      call fd4_vis5d_write(err, st_opt=(/new/))
    end if
```

And finally the Vis5D file needs to be closed before terminating the program:

```
  !! Close Vis5D
  call fd4_vis5d_close(err)
```

5.8 NetCDF Output: 08_heat_netcdf.F90

NetCDF output is working quiet similar to Vis5D output. But here, multiple ***NetCDF communicators*** can be defined to create independent output contexts. Thus, a new variable needs to be defined:

```
! FD4 netcdf communicator
type(fd4_netcdf4_comm) :: nfcomm
```

When opening a NetCDF file, the *NetCDF communicator* is initialized. This handle is parameter of all NetCDF output routines. Thus, opening the file and writing the initial data before the time stepping loop looks like this:

```
!! Initialize NetCDF output and write initial data
call fd4_netcdf4_open(nfcomm, domain, 'out.nc', 1, (/THETA/), (/now/), err)
call fd4_netcdf4_write(nfcomm,err)
```

Writing to NetCDF during the time stepping looks quiet similar to the Vis5D version. There is also an optional parameter `st_opt` to set change the time step to write for all variables.

```
  !! Write NetCDF output
  if(mod(step,100)==0) then
    if(rank==0) write(*,'(A,I5)') 'step ',step
    call fd4_netcdf4_write(nfcomm, err, st_opt=(/new/))
  end if
```

And this is how an NetCDF file is closed. The *NetCDF communicator* can be re-used (by calling `fd4_netcdf4_open`) after this call:

```
!! Close NetCDF
call fd4_netcdf4_close(nfcomm, err)
```

Note, that there are two ways of integrating NetCDF in FD4:

- Serial NetCDF: This is the standard way. You can use NetCDF version 3 or 4 in this case. The output is performed completely serial, that means rank 0 collects and writes all data.
- Parallel NetCDF4 based on HDF5: This requires an installation of parallel HDF5 and NetCDF4 based on HDF5. In this case, the data is written in parallel. The resulting file is actually a HDF5 file, but it can be read by all tools that are based on the NetCDF4/HDF5 installation. However, if you experience problems in post-processing or visualizing the output file, you can convert it to real NetCDF format using the `nccopy` utility of the NetCDF4/HDF5 installation:
  `nccopy -k2 <input NetCDF4 file> <output NetCDF file>`

FD4 can only use exclusively one of these two ways. The method used is determined when building the FD4 library, which is described in Chapter 4.

5.9 Boundary Conditions: 09_heat_boundary.F90

FD4 supports three types of boundary conditions, see 2.8 for more information

- Periodic boundary conditions: Can be enabled for each dimension with the argument `periodic` of `fd4_domain_create`.
- Zero-gradient: Must be set in each iteration with the routines `fd4_boundary_zerograd` and `fd4_boundary_zerograd_block`. They set cell-centered variables in the *Boundary Ghost Blocks* to the same value as in the cells at domain boundary and face variables at domain boundary (within domain) to `vnull`.
- Specific boundary conditions: User-defined values in the *Boundary Ghost Blocks* can be set with the routines `fd4_boundary_spec` and `fd4_boundary_spec_block`.

The following snippets extend the heat equation example by non-periodic boundary conditions. Firstly, specific, fixed boundary conditions for the lower z boundary of the domain are set before the beginning of the iteration. The values are set to 0 for all *Boundary Ghost Blocks* except for a rectangular region, in which the values are set to 2:

```
! set (fixed) boundary conditions for lower z
call fd4_boundary_spec (domain, (/THETA,THETA/), (/now,new/), 3, 1, (/0.0_r_k/))
! iterate over all local blocks to add some position-dependent boundary conditions
call fd4_iter_init(domain, iter)
do while(associated(iter%cur))
  ! block is at lower boundary in z dimension
  if(iter%cur%pos(3)==1) then
    ! get offset from block-local to global coordinates
    call fd4_iter_offset(iter, offset)
    ! loop over grid cells in x and y
    do y=1,iter%cur%ext(2)
      do x=1,iter%cur%ext(1)
        ! set boundary conditions for specific grid cells
        if(x+offset(1)>grid(1)/4 .and. x+offset(1)<3*grid(1)/4 .and. &
           y+offset(2)>grid(1)/4 .and. y+offset(2)<3*grid(1)/4) then
          iter%cur%neigh(3,1)%l%fields(THETA,now)%l(:,x,y,1) = 2.0_r_k
          iter%cur%neigh(3,1)%l%fields(THETA,new)%l(:,x,y,1) = 2.0_r_k
        end if
      end do
    end do
  end if
  call fd4_iter_next(iter)
end do
```

The other boundaries (x, y, upper z) are initialized with zero-gradient boundary conditions at the beginning of each iteration for each single *Block*:

```
    ! set zero-gradient boundary conditions for x, y, and upper z for this block
    call fd4_boundary_zerograd_block(domain, iter%cur, (/THETA/), (/now/), FD4_XY)
    call fd4_boundary_zerograd_block(domain, iter%cur, (/THETA/), (/now/), FD4_Z, opt_dir=2)
```

The example demonstrates how the heat expands from the lower z boundary into the domain.

5.10 Coupling Interface I: 10_heat_couple.F90

This example extends `08_heat_netcdf.F90` by using the coupling interface of FD4 (3.3) to initialize the grid variable theta. Seven FD4 routines are required to achieve this:

- `fd4_couple_create`: Creates an empty *Couple Context*.
- `fd4_couple_add_partition`: Adds metadata of a rectangular 3D partition: Position in the grid and owner rank. Must be called for all partitions (local and non-local). For each rank an arbitrary number of partitions can be defined, but the tutorial programs only use one partition per rank.
- `fd4_couple_add_var`: Adds metadata of a variable for coupling: Variable index and step index. The coupling takes place for each partition/variable combination. One such combination is called *Couple Array*. This subroutine returns an *Couple Array Index* which must be used in `fd4_couple_set_local_3D_array` to identify the local *Couple Array*. After the first call to this routine, you cannot add further partitions.
- `fd4_couple_set_local_3D_array`: Assign the pointer and offset to relevant data of the local *Couple Array*. The argument must be a pointer. Use an additional pointer to point to an allocatable where required (contiguous subsections are allowed). After the first call to this routine, you cannot add further variables. Note, that for face variables the *Couple Array* must be extended in face direction by one cell plane.
- `fd4_couple_commit`: Commit the *Couple Context*, FD4 now checks the *Couple Arrays* and prepares MPI data types. From now on, you cannot add further arrays to the *Couple Context*.

- `fd4_couple_put`: This routine actually performs the coupling data transfer. It puts data from the distributed coupling arrays to FD4's data structures.
- `fd4_couple_delete`: This routine frees all memory associated with the coupling context, which can be very much when running with 1000s of processes.

In the example program, the array with initial data of theta is distributed over all ranks in a simple 1D partitioning. Each rank adds a local coupling array to the *Couple Context*.

```
!! Create couple context to put initial data to the FD4 domain
call fd4_couple_create(couple_init, domain, err, opt_cpldir=FD4_CPL_PUT)
! add one couple array per rank for theta
do irank = 0, nproc-1
  ! Set cabnd(:,:) to the bounds of the coupling arrays of current rank.
  ! (here we simply calculate bounds for an 1D-partitioning)
  cabnd(1:3,1) = (/   1+(irank*dsize(1,2))/nproc,          1,          1 /)
  cabnd(1:3,2) = (/ ((irank+1)*dsize(1,2))/nproc, dsize(2,2), dsize(3,2) /)
  ! Add a couple array of current rank with given bounds.
  call fd4_couple_add_partition(couple_init, irank, cabnd, err)
end do
! Add the variable THETA to the couple context.
! FD4 returns the identifier caidxTheta for the local couple array.
call fd4_couple_add_var(couple_init, THETA, now, err, caidxTheta)
! allocate the local couple array and add it to the couple context
allocate(iniTheta(cabnd(1,1):cabnd(1,2),cabnd(2,1):cabnd(2,2),cabnd(3,1):cabnd(3,2)))
iniTheta = 0.0_r_k
arraypointer => iniTheta(:,:,:)
call fd4_couple_set_local_3D_array(couple_init, caidxTheta, arraypointer, (/1,1,1/), err)
! commit the couple context, FD4 now checks the couple arrays and prepares MPI data types
call fd4_couple_commit(couple_init, err)
! fill the couple array with initial data
do z=cabnd(3,1),cabnd(3,2)
  do y=cabnd(2,1),cabnd(2,2)
    do x=cabnd(1,1),cabnd(1,2)
      ! get global coordinates of this grid cell and scale to [0,1]
      global_pos(1:3) = REAL( (/x,y,z/) - 1 ) / REAL(dsize(1:3,2) - 1)
      ! distance from domain center to current grid cell
      cr = sqrt( (global_pos(1)-0.5)**2+(global_pos(2)-0.5)**2+(global_pos(3)-0.5)**2 )
      if(cr < radius) then
        iniTheta(x,y,z) = 2.0_r_k * cos(3.14159*cr/(2*radius))
      end if
    end do
  end do
end do
! finally put data to FD4's data structures
call fd4_couple_put(couple_init, err)
! delete couple context and deallocate couple array
call fd4_couple_delete(couple_init, err)
deallocate(iniTheta)
```

The reverse action of `fd4_couple_put` is performed by `fd4_couple_get`: It transfers data from the FD4 data structures to distributed coupling arrays. You can also add 4D *Couple Arrays* (for variables with bins) to the *Couple Context* using `fd4_couple_set_local_4D_array`. It is allowed to change the coupling arrays of a committed *Couple Context* (as long as the size of the array fits) using `fd4_couple_chg_local_3D_array` and `fd4_couple_chg_local_4D_array`.

5.11 Coupling Interface II: 11_advection_couple.F90

This is a completely new example program to demonstrate how to use the coupling interface not only for initialization, but for its main purpose: to exchange data between the FD4 data structures and differently distributed data regularly during iteration.

The program computes the concentration of some tracer `c` under a changing wind field by a very simple advection scheme. The horizontal winds `u` and `v` and the vertical wind `w` are 'computed' externally (which stands for some external model coupled to the tracer model) and transferred to the FD4 data structures at the beginning of every time step using the coupling interface.

Here is the main program with the time stepping loop. Note the call to `advection_set_vortex` at program initialization and during time stepping, which sets the wind fields in the *Couple Arrays*. The actual coupling is carried out by the following call to `fd4_couple_put`.

```
program fd4_demo_advection
  use fd4_mod
  use advection
  implicit none
  include 'mpif.h'
  !! Setup parameters
  integer, parameter :: grid(3) = 32          ! number of grid cells for x, y, z
  real(r_k), parameter :: dt = 0.1            ! time step size
  integer, parameter :: nsteps = 2000         ! number of time steps to compute
  integer, parameter :: omod = 100            ! number of steps between output
  ! FD4 variable table
  type(fd4_vartab) :: vartab(4)
  ! FD4 domain
  type(fd4_domain), target :: domain
  integer :: dsize(3,2), bnum(3), nghosts(3)
  logical :: periodic(3)
  ! FD4 iterator
  type(fd4_iter) :: iter
  ! FD4 ghost communication
  type(fd4_ghostcomm) :: ghostcomm(2)
  ! FD4 netcdf communicator
  type(fd4_netcdf4_comm) :: nfcomm
  ! FD4 coupling context
  type(fd4_couple) :: couple
  ! misc
  integer :: rank, err, now, new, step
  !! MPI Initialization
  call MPI_Init(err)
  call MPI_Comm_rank(MPI_COMM_WORLD,rank ,err)
  !! Create the FD4 variable table
  !                              name, nb, st,  unused, ini,      vthres,  discret.
  vartab(varU) = fd4_vartab(     'u',  1,  1, .false., .0 , FD4_NOTHRES, FD4_FACEX )
  vartab(varV) = fd4_vartab(     'v',  1,  1, .false., .0 , FD4_NOTHRES, FD4_FACEY )
  vartab(varW) = fd4_vartab(     'w',  1,  1, .false., .0 , FD4_NOTHRES, FD4_FACEZ )
  vartab(varC) = fd4_vartab(     'c',  1,  2, .false., .0 , FD4_NOTHRES, FD4_CELLC )
  !! Create the FD4 domain
  dsize(1:3,1) = (/1, 1, 1/)     ! grid start indices
  dsize(1:3,2) = grid(1:3)       ! grid end   indices
  bnum(1:3) = grid(1:3) / 8      ! number of blocks in each dimension
  nghosts(1:3) = (/1, 1, 1/)     ! number of ghost cells in each dimension
  periodic(1:3) = .true.         ! periodic boundaries
  call fd4_domain_create(domain, bnum, dsize, vartab, nghosts, periodic, MPI_COMM_WORLD, err)
  if(err/=0) then
    write(*,*) rank, ': fd4_domain_create failed'
    call MPI_Abort(MPI_COMM_WORLD, 1, err)
  end if
  !! Allocate the blocks of the domain
  call fd4_util_allocate_all_blocks(domain, err)
  !! Initialize time step indicators
  now = 1
  new = 2
  !! Allocate buffer array and initialize the concentration field varC
  call advection_init(domain, now)
  !! Initialize coupling with wind fields
  call advection_init_coupling(domain, couple)
  !! Set wind fields and perform first coupling
  call advection_set_vortex(0.0)
  call fd4_couple_put(couple, err)
  !! Create ghost communicator for variable varC (one for each time level of varC)
  call fd4_ghostcomm_create(ghostcomm(1), domain, 1, (/varC/), (/1/), err)
  call fd4_ghostcomm_create(ghostcomm(2), domain, 1, (/varC/), (/2/), err)
  !! Initialize output and write initial data
  call fd4_netcdf4_open(nfcomm, domain, 'out.nc', 4, (/varU,varV,varW,varC/), &
                        (/1,1,1,now/), err)
  call fd4_netcdf4_write(nfcomm,err)
  call fd4_vis5d_open(domain, 'out.v5d', nsteps/omod+1, 4, (/varU,varV,varW,varC/), &
                        (/1,1,1,now/), err)
  call fd4_vis5d_write(err)
  !! Time stepping loop
  do step=1,nsteps
    !! Set wind fields in the couple arrays and perform coupling
    call advection_set_vortex((step-1.0)/nsteps)
```

```
    call fd4_couple_put(couple, err)
    ! exchange ghost cells for time level 'now'
    call fd4_ghostcomm_exch(ghostcomm(now), err)
    ! iterate over all local blocks
    call fd4_iter_init(domain, iter)
    do while(associated(iter%cur))
      ! compute simplistic advection scheme on current block
      call advection_compute(iter, now, new, dt)
      call fd4_iter_next(iter)
    end do
    ! write output
    if(mod(step,omod)==0) then
      if(rank==0) write(*,'(A,I5)') 'step ',step
      call fd4_netcdf4_write(nfcomm, err, st_opt=(/1,1,1,new/))
      call fd4_vis5d_write(err, st_opt=(/1,1,1,new/))
    end if
    ! swap time step indicators
    now = 3 - now
    new = 3 - new
  end do
  !! Close output files
  call fd4_netcdf4_close(nfcomm, err)
  call fd4_vis5d_close(err)
  !! Delete couple context, ghost communicator, and domain, finalize MPI
  call fd4_couple_delete(couple, err)
  call fd4_ghostcomm_delete(ghostcomm(1))
  call fd4_ghostcomm_delete(ghostcomm(2))
  call fd4_domain_delete(domain)
  call MPI_Finalize(err)
end program fd4_demo_advection
```

The subroutine `advection_init_coupling` allocates the *Couple Arrays* `cplU`, `cplV`, and `cplW` and initializes the *Couple Context*. The *Couple Arrays* are distributed in the same way as in the previous example, but since the winds are defined as face variables, the size of the arrays in extended by one row in the dimension of the face.

```
  subroutine advection_init_coupling(domain, couple)
    type(fd4_domain), intent(inout), target :: domain
    type(fd4_couple), intent(inout) :: couple
    integer :: irank, err, caidxU, caidxV, caidxW
    real(r_k), pointer :: arraypointer(:,:,:)
    !! Create couple context to put initial data to the FD4 domain
    call fd4_couple_create(couple, domain, err, opt_cpldir=FD4_CPL_PUT)
    !! Add couple arrays of all processes to the couple context
    do irank = 0, nproc-1
      ! Set cabnd(:,:) to the bounds of the coupling arrays of current rank
      cabnd(1:3,1) = (/   1+(irank*dsize(1,2))/nproc,          1,          1 /)
      cabnd(1:3,2) = (/ ((irank+1)*dsize(1,2))/nproc, dsize(2,2), dsize(3,2) /)
      ! Add a couple array of current rank with given bounds.
      call fd4_couple_add_partition(couple, irank, cabnd, err)
    end do
    ! Add the variables varU, varV, and varW to the couple context.
    ! FD4 returns the identifiers for the local couple array.
    call fd4_couple_add_var(couple, varU, 1, err, caidxU)
    call fd4_couple_add_var(couple, varV, 1, err, caidxV)
    call fd4_couple_add_var(couple, varW, 1, err, caidxW)
    ! set this process' couple array bounds
    cabnd(1:3,1) = (/   1+(rank*dsize(1,2))/nproc,          1,          1 /)
    cabnd(1:3,2) = (/ ((rank+1)*dsize(1,2))/nproc, dsize(2,2), dsize(3,2) /)
    ! x-face variable cplU has one additional cell row in x dimension
    allocate(cplU(cabnd(1,1):cabnd(1,2)+1,cabnd(2,1):cabnd(2,2),cabnd(3,1):cabnd(3,2)))
    cplU = 0.0_r_k
    arraypointer => cplU(:,:,:)
    call fd4_couple_set_local_3D_array(couple, caidxU, arraypointer, (/1,1,1/), err)
    ! y-face variable cplV has one additional cell row in y dimension
    allocate(cplV(cabnd(1,1):cabnd(1,2),cabnd(2,1):cabnd(2,2)+1,cabnd(3,1):cabnd(3,2)))
    cplV = 0.0_r_k
    arraypointer => cplV(:,:,:)
    call fd4_couple_set_local_3D_array(couple, caidxV, arraypointer, (/1,1,1/), err)
    ! z-face variable cplW has one additional cell row in z dimension
    allocate(cplW(cabnd(1,1):cabnd(1,2),cabnd(2,1):cabnd(2,2),cabnd(3,1):cabnd(3,2)+1))
    cplW = 0.0_r_k
    arraypointer => cplW(:,:,:)
    call fd4_couple_set_local_3D_array(couple, caidxW, arraypointer, (/1,1,1/), err)
    ! commit the couple context, FD4 now checks the couple arrays and prepares MPI data types
```

```
    call fd4_couple_commit(couple, err)
  end subroutine advection_init_coupling
```

5.12 Dynamic Load Balancing: 12_advection_balance.F90

The advection scheme in `11_advection_couple.F90` causes some small load imbalances, since it requires less computation time in grid cells where the concentration is zero. These code snippets add dynamic load balancing (3.2) to the previous example. Firstly, in the initialization part of the program after creating the *Domain*, we can set some load balancing parameters:

```
  !! Set some load balancing parameters
  call fd4_balance_params (domain, method=FD4_BALANCE_HSFC2_PARALLEL, opt_lbtol=0.95)
```

Setting parameters is not required, but it may be useful to override the defaults. These parameters choose space-filling curve load balancing with an threshold of 0.95. This means that load balancing is only invoked, if the balance falls below this value, where load balance is defined as the average partition load divided by the maximum partition load. Thus, a value of 1.0 for `opt_lbtol` is the strictest and 0.0 disables load balancing. There are more parameters, refer to the API documentation.

The following modifications are necessary at the time stepping loop for dynamic load balancing:

```
  do step=1,nsteps
    !! Set wind fields in the couple arrays and perform coupling
    call advection_set_vortex((step-1.0)/nsteps)
    call fd4_couple_put(couple, err)
    ! exchange ghost cells for time level 'now'
    call fd4_ghostcomm_exch(ghostcomm(now), err)
    ! iterate over all local blocks
    call fd4_iter_init(domain, iter)
    do while(associated(iter%cur))
      ! time the start of computations on current block
      call fd4_iter_start_clock(iter)
      ! compute simplistic advection scheme on current block
      call advection_compute(iter, now, new, dt)
      ! time the end of computations on current block, set block weight
      call fd4_iter_stop_clock(iter, 0.0)
      call fd4_iter_next(iter)
    end do
    ! rebalance the workload based on the measured block weights
    call fd4_balance_readjust(domain, err, opt_stats=stats)
    ! write output
    if(mod(step,omod)==0) then
      if(rank==0) write(*,'(A,I5,A,F6.3)') 'step ',step,'   bal:',stats%last_measured_balance
      call fd4_netcdf4_write(nfcomm, err, st_opt=(/1,1,1,new/))
      call fd4_vis5d_write(err, st_opt=(/1,1,1,new/))
    end if
    ! swap time step indicators
    now = 3 - now
    new = 3 - new
  end do
```

The load is measured with the routines `fd4_iter_start_clock` and `fd4_iter_stop_clock` that time the computation of one *Block*. This measurement is used to determine the load balance and acts as weight for rebalancing the load. Every computation on a *Block* should be enclosed by these calls. The actual load balancing is invoked by `fd4_balance_readjust`. The optional parameter `opt_stats` returns a type that contains some statistical information about the load balancing such as the number of migrated *Blocks*, measured load balance and so on – see API documentation.

5.13 Adaptive Block Mode: 13_advection_adaptive.F90

Now we can apply the *Adaptive Block Mode* (2.7) to the previous example. Only small modifications are necessary. Firstly, a threshold value (`vthres`) must be defined for the concentration field variable `c` in the *Variable Table*. FD4 will remove all *Blocks* where `c` is smaller than this threshold.

```
!! Create the FD4 variable table
!                              name, nb, st,  unused, ini,      vthres,  discret.
vartab(varU) = fd4_vartab(     'u',  1,  1, .false., .0 , FD4_NOTHRES, FD4_FACEX )
vartab(varV) = fd4_vartab(     'v',  1,  1, .false., .0 , FD4_NOTHRES, FD4_FACEY )
vartab(varW) = fd4_vartab(     'w',  1,  1, .false., .0 , FD4_NOTHRES, FD4_FACEZ )
vartab(varC) = fd4_vartab(     'c',  1,  2, .false., .0 ,      1.d-10, FD4_CELLC )
```

Then, the call to `fd4_util_allocate_all_blocks` must be removed in the initialization part of the main program. Instead of allocating all *Blocks*, only those *Blocks* where the initial values of `c` are greater than the threshold should be allocated. This is achieved by a 2-step procedure in subroutine `advection_init` prior to calling `fd4_couple_put` to transfer the initial data to FD4 (which has not allocated any *Blocks* yet):

1. `fd4_couple_mark_blocks` marks all *Blocks* as 'required' which contain `c` greater than the threshold in the *Couple Arrays* of the *Couple Context*.
2. `fd4_balance_readjust` allocates these marked *Blocks* in a load-balanced way.

```
! mark all FD4 blocks which contain varC > threshold
call fd4_couple_mark_blocks(couple_init, err)
! this actually allocates these blocks in a load-balanced way
call fd4_balance_readjust(domain, err)
! finally put data to FD4's data structures
call fd4_couple_put(couple_init, err)
```

Finally, an additional call to FD4 is required in the *Block* loop during time stepping. After the computations of a *Block* are done, `fd4_iter_empty` must be called so that FD4 checks if the *Block* is now empty or not.

```
! iterate over all local blocks
call fd4_iter_init(domain, iter)
do while(associated(iter%cur))
  ! time the start of computations on current block
  call fd4_iter_start_clock(iter)
  ! compute simplistic advection scheme on current block
  call advection_compute(iter, now, new, dt)
  ! check if the block is empty
  call fd4_iter_empty(iter, varC, new)
  ! time the end of computations on current block, set block weight
  call fd4_iter_stop_clock(iter, 0.0)
  call fd4_iter_next(iter)
end do
```

This information is used in `fd4_balance_readjust` to find not any longer required *Blocks* and to identify *Blocks* that need to be allocated because `c` moves near the boundary of an existing *Neighbor Block*.

5.14 Utilities: 14_advection_utils.F90

These utility subroutines are not optimized for performance and should be used during development and for debug purpuses only, but not for large-scale runs.

Face variable utilities: As described in 2.4 the *Blocks* share the same copy of face variables at the boundaries in the face variable's direction. To check the consistency of face variables between *Neighbor Blocks* for a given variable, FD4 offers `fd4_util_check_facevar`. Errors are reported on stdout. The implementation is very simple: All *Blocks* are migrated to rank 0

which performs the checks, than the *Blocks* are distributed again over all ranks. Therefore, you should use `fd4_util_check_facevar` only for small debug runs!

The routine `fd4_util_propagate_facevar` enforces the consistency of the given face variable in all *Blocks*. It copies face values at the boundary to the corresponding *Neighbor Block*.

```
  ! generate an inconsistency in varV
  call fd4_iter_init(domain, iter)
  if(rank==0 .and. associated(iter%cur)) iter%cur%fields(varV,1)%l = 0.1701_r_k
  ! check the 3 wind vectors for consistency between neighbor blocks
  if(rank==0) write(*,'(A)') 'Checking varU for consistency'
  call fd4_util_check_facevar(domain, varU, 1, err)
  if(rank==0) write(*,'(A)') 'Checking varV for consistency'
  call fd4_util_check_facevar(domain, varV, 1, err)
  if(rank==0) write(*,'(A)') 'Checking varW for consistency'
  call fd4_util_check_facevar(domain, varW, 1, err)
  ! make face variables consistent
  call fd4_util_propagate_facevar(domain, (/varV/), (/1/), .false., err)
  ! check varV again
  if(rank==0) write(*,'(A)') 'Checking varV for consistency'
  call fd4_util_check_facevar(domain, varV, 1, err)  ! should not report a problem
```

Data utilities: To quickly access the data at a certain point in the grid, you can use the subroutine `fd4_util_get_value`. The subroutine `fd4_util_get_array` gathers the grid for a single variable on a given rank. The array passed to this routine must match the size of the domain.

```
  ! print value at given position
  value = fd4_util_get_value(domain, varC, now, (/1,16,16,8/), 0)
  if(rank==0) write(*,'(A,F8.3)') 'varC at (16,16,8) = ',value
  ! let rank 0 collect the complete varC data
  abnd(1:3,1:2) = domain%dbnd
  abnd(0,1:2) = 1
  allocate(array(1,abnd(1,1):abnd(1,2),abnd(2,1):abnd(2,2),abnd(3,1):abnd(3,2)))
  call fd4_util_get_array(domain, varC, now, abnd, array, err, 0)
  ! print sum over each k-column in ascii-art
  if(rank==0) then
    do j=abnd(2,2),abnd(2,1),-1
      do i=abnd(1,1),abnd(1,2)
        if(SUM(array(1,i,j,:))>1.0_r_k) then
          write(*,'(A2)',advance='no') '#'
        elseif(SUM(array(1,i,j,:))>0.0_r_k) then
          write(*,'(A2)',advance='no') '+'
        else
          write(*,'(A2)',advance='no') ' '
        end if
      end do
      write(*,*)
    end do
  end if
```

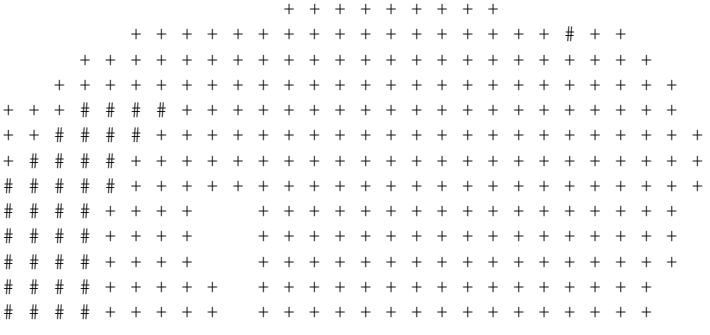
The output of the demo program `14_advection_utils` should look like this when running on 2 processes:

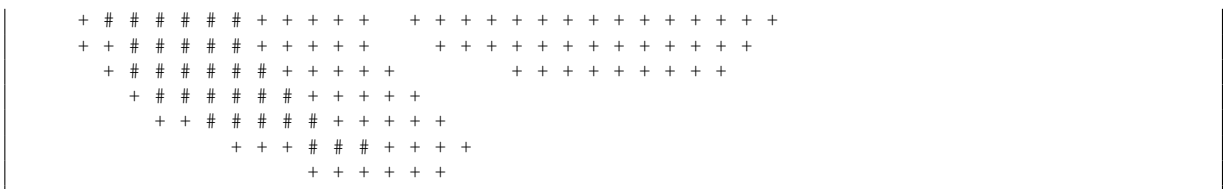
```
[FD4:0000] created new fd4_domain:
[FD4:0000] dim   start     end  blocks   blksz  ghosts  per.bd
[FD4:0000]   x       1      32       8    4.00       1       T
[FD4:0000]   y       1      32       8    4.00       1       T
[FD4:0000]   z       1      32       8    4.00       1       T
[FD4:0000] max. number of blocks:   512
[FD4:0000] Hilbert SFC level:       3
[FD4:0000] number of MPI processes: 2
[FD4:0000] block pool lists:        4
[FD4:0000] block pool max. size:    739
[FD4:0000] adaptive block mode:     T
[FD4:0000] variable table:
[FD4:0000]  id  nbins  nsteps  dyn     vnull  threshld  face  name
[FD4:0000]   1      1       1    F   0.0E+00         -     x  u
[FD4:0000]   2      1       1    F   0.0E+00         -     y  v
[FD4:0000]   3      1       1    F   0.0E+00         -     z  w
[FD4:0000]   4      1       2    F   0.0E+00   1.0E-10     -  c
Checking varU for consistency
Checking varV for consistency
```

```
[FD4:0000]  face 2 of block at   4   4   2 is not consistent!
Checking varW for consistency
Checking varV for consistency
varC at (16,16,8) =    1.469
```



```
step   100   bal: 0.953   blocks:   14
step   200   bal: 0.906   blocks:   20
step   300   bal: 0.981   blocks:   33
step   400   bal: 0.957   blocks:   42
step   500   bal: 0.989   blocks:   54
step   600   bal: 0.977   blocks:   66
step   700   bal: 0.983   blocks:   74
step   800   bal: 0.650   blocks:   77
step   900   bal: 0.982   blocks:   87
step  1000   bal: 0.938   blocks:   92
step  1100   bal: 0.969   blocks:  102
step  1200   bal: 0.954   blocks:  112
step  1300   bal:-1.000   blocks:  121
step  1400   bal: 0.966   blocks:  131
step  1500   bal: 0.985   blocks:  143
step  1600   bal: 0.986   blocks:  153
step  1700   bal: 0.972   blocks:  162
step  1800   bal: 0.800   blocks:  167
step  1900   bal: 0.981   blocks:  175
step  2000   bal: 0.976   blocks:  186
varC at (16,16,8) =    0.000
```





```
[FD4:0000] FD4 Statistics                min          max          avg              sum
[FD4:0000] Balance check us           168288       262888       215588           431176
[FD4:0000] Balance part  us            27410        28679        28044            56089
[FD4:0000] Balance mig   us            86703        87278        86990           173981
[FD4:0000] Balance recv Blocks         2769         2793         2781             5562
[FD4:0000] Balance recv Bytes       8251620      8323140      8287380         16574760
[FD4:0000] Domain block alloc           108          143          125              251
[FD4:0000] Domain block free              0            0            0                0
[FD4:0000] Domain ghost alloc            66           66           66              132
[FD4:0000] Domain ghost free              0            0            0                0
[FD4:0000] Couple put Bytes        47999104     99516544     73757824        147515648
[FD4:0000] Couple get Bytes               0            0            0                0
[FD4:0000] Ghostcomm Bytes          7142528      7142528      7142528         14285056
```