FD4 Manual

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

Version fd4-2010-07-01

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:

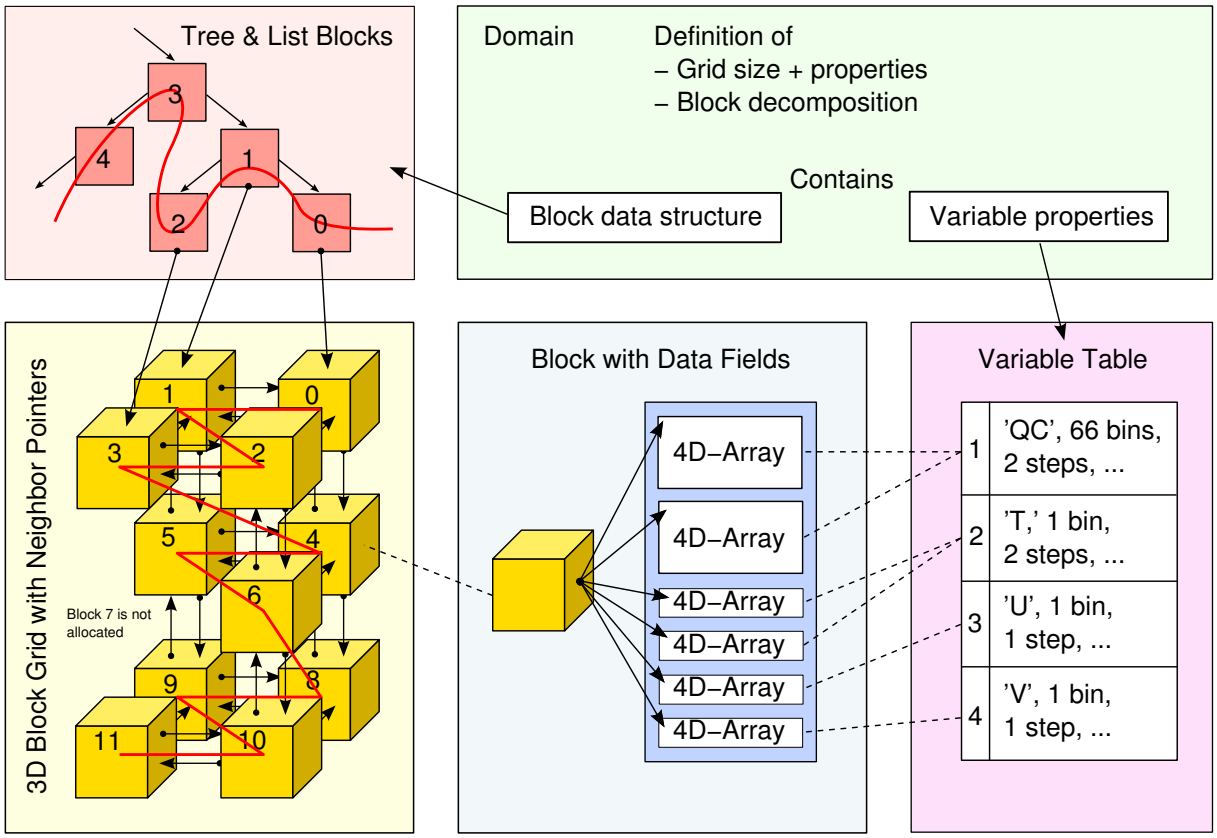
- Many variables per grid cell ($>$100)
- Varying workload per grid cell (varying in time as well as space) which demands dynamic load balancing
- Multiphase model: Additional computations for a limited spatial subset of the grid (drops, clouds, combustion processes, flame fronts, etc.)
- Model system: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

The basic features of FD4 are:

- 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
- Simple NetCDF and Vis5D 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 which 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 indentifier. 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 indices 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*.

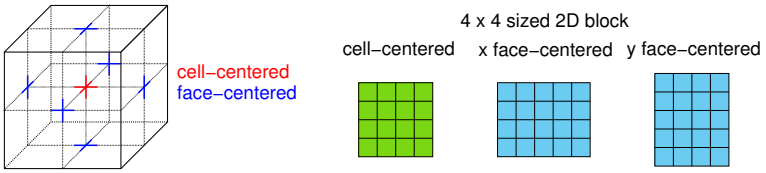
Note that not all *Blocks* may be present at a time: In a parallel run (which is the intended use of FD4!), the *Blocks* are distributed to the processes. For more details about parallelization refer to Section 3. Additionally, when running in ***Adaptive Block Mode***, only a specific subset of the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



As a consequence, two adjacent *Blocks* share copies of the same face variable at their boundary. This has consequences regarding consistency, see Section 5.6. 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.

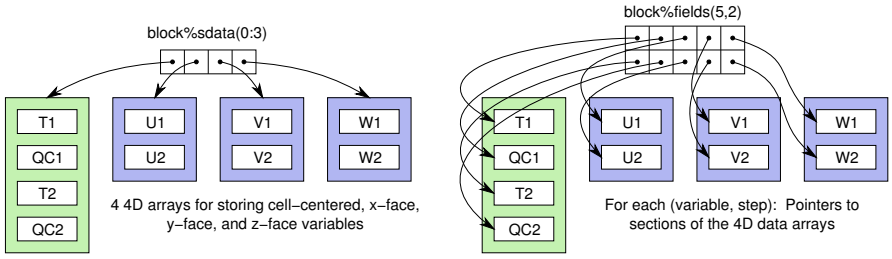
A specific variable item of one *Block* is accessed as `block%sdata(f)%l(b,x,y,z)` with

- 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:

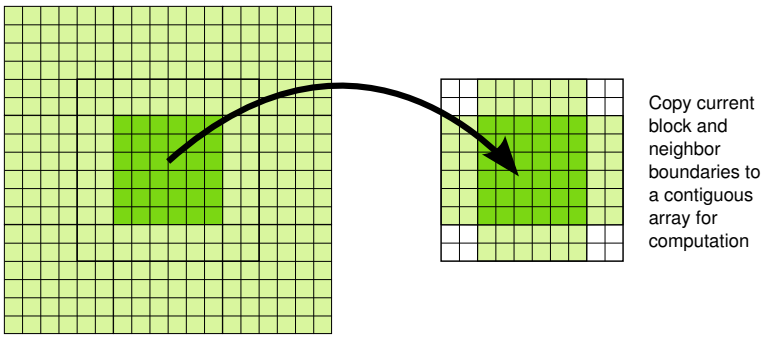


Using the `block%fields` array, only grid cells of the local block can be accessed, but not grid cells of *Neighbor Blocks* (ghost cells).

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:



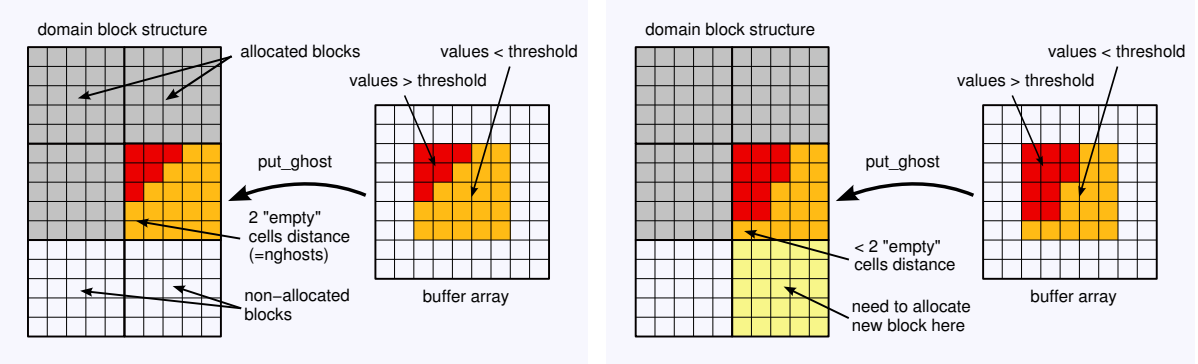
Note, that only data of the 6 direct *Neighbor Blocks* (in 3D) are copied, not the data of the diagonal *Neighbor Blocks*. The resulting values in the area of the ghost cells depend on the state of each *Neighbor Block*:

- 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

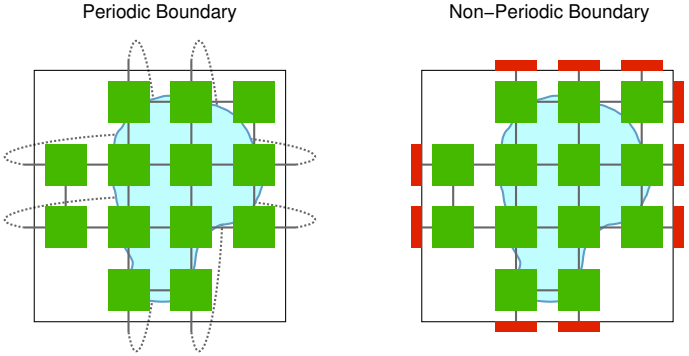
FD4 allows the dynamic adaption of the block allocation to spatial structures. It is useful for special multiphase applications when neither computations nor data are required for certain regions of the spatial grid. In this case, memory can be saved by not allocating the unused (***empty***) blocks. 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, as shown in these figures:



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*):



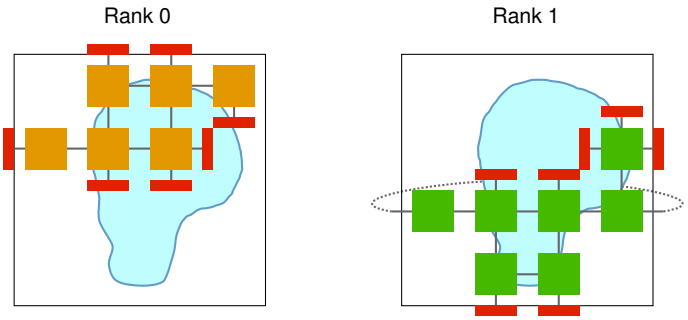
3 Parallelization and Coupling

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 can be executed whenever neccesary.

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:

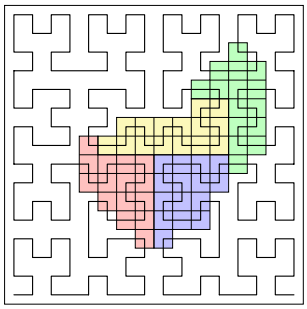
1. Determine if load balancing is necessary.
2. Calculate a new partitioning, i.e. mapping of *Blocks* to processes.
3. Migration and (De)allocation of *Blocks*.

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. This ***Auto Mode***

requires that the *Block Weight* are set to the computation time. FD4 weighs the time lost due to imbalance against the time required for load balancing.

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:

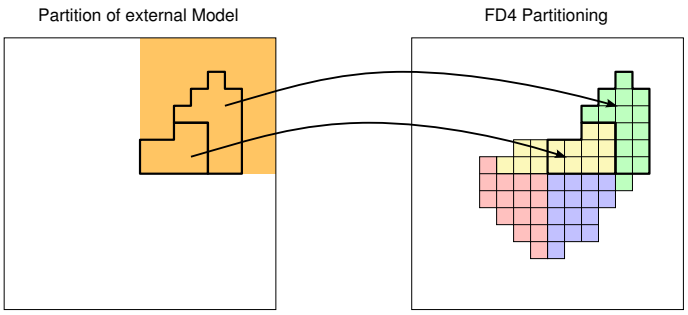


3.3 Coupling

FD4 allows to couple models based on FD4 to external models, i.e. transfer 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 than 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 to couple 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:

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

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. Note, that NetCDF output is currently not optimzed in FD4.
  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.
  Website: `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview/`

4.2 Configuration

Refer to the file `README` provided with the FD4 package.

5 User Interface

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 documentation of the routines can be found in `doc/index.html`.

5.1 Basics

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 `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 real 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.

5.2 Variable Table Definition

5.3 Domain Creation

5.4 Block Iteration

5.5 Block Data Access

5.6 Ghost Data Exchange

5.7 Boundary Conditions

5.8 Adaptive Block Allocation

5.9 Dynamic Load Balancing

5.10 Coupling Interface

5.11 Data Utilities

5.12 Vis5D Output

5.13 NetCDF Output