## Introduction

Dimr is the Deltares Integrated Model Runner. Its task is to execute models and exchange data between them using the BMI interface.

This document describes the design of dimr, the main implementation choices and some suggestions to improve the existing code. The document is organized as follow: section 2 describes the dimr library used by the dimr application, section 3 describes the common BMI interface implemented by all D-HYDRO models, section 4 lists the basic steps executed during a dimr run and section 5 describes the Dimr::dimr\_update function, which is responsible for executing and synchronizing the model runs. In section 6 some improvements are proposed.

## The dimr\_lib

Figure 1 shows the simplified class diagram of dimr exe, dimr\_lib and the BMI interface used by D-HYDRO models (some type definitions are omitted for simplicity). In order to understand the class diagram we introduce the following name conventions:

* Component: identifies a simulator (D-FlowFM, RTC-Tools, WAQ, WAVE or Delft3D-FLOW). Each component is available as a dynamic library (.dll or .so) with a BMI interface (Basic Model Interface) consisting of 11 functions responsible for executing the simulator, getting the computed values and setting the boundary conditions.
* Target: identifies an output quantity (e.g. the water levels) or a boundary condition (e.g. the level of a weir) for a specific component.
* Coupler: identifies an entity where 2 components and their respective targets are coupled together (e.g. D-FlowFM + a water level and RTC-Tools + a weir level).

As can be seen from the diagram, the dimr\_lib consists of a dimr class and heavily uses composition of structures (no inheritance).

The methods of the dimr class are used in library functions declared in the global scope and exposed to clients of the library using DllExport statements (e.g. DllExport int initialize(const char \* configfile)). These functions are listed in Table 1 and represent the BMI interface of dimr\_lib, similarly to other components, as can be seen in Figure 1 for the dimr\_component structure. The functions of the BMI interface of dimr\_lib have access to an instance of the dimr class through the static pointer “thisDimr” declared in the global scope. Only one instance of the dimr class is present at runtime (or more precisely one instance for each MPI rank), effectively implementing a singleton pattern.

The dimr\_control\_block structure contains all the information read from the <control> group in the xml configuration file. The subBlocks array of dimr\_control\_block store the functional elements of dimr, themselves of type dimr\_control\_block. This choice of a nested data structure seems to be motivated by the nested structure of the xml configuration file. To explain how the data is organized in the subBlocks array we need to introduce an example from a configuration file, graphically represented in Figure 2.

D:\carniato\LUCA\ENGINES\delft3d\DocumentingDIMR\complete_diagram.emf

**Figure 1.** The class diagram of DimrExe (left side) and dimr\_lib (right side). DimrExe and dimr components call the methods of D-HYDRO libraries using the BMI interface (red box in the central part of the figure delimited by a red dashed line).

<control> #(level 0)

<parallel> #(level 1)

<startGroup> #(level 2)

<time>0.0 60.0 99999999.0</time>

<coupler name="flow2rtc"/>

<start name="rtc"/>

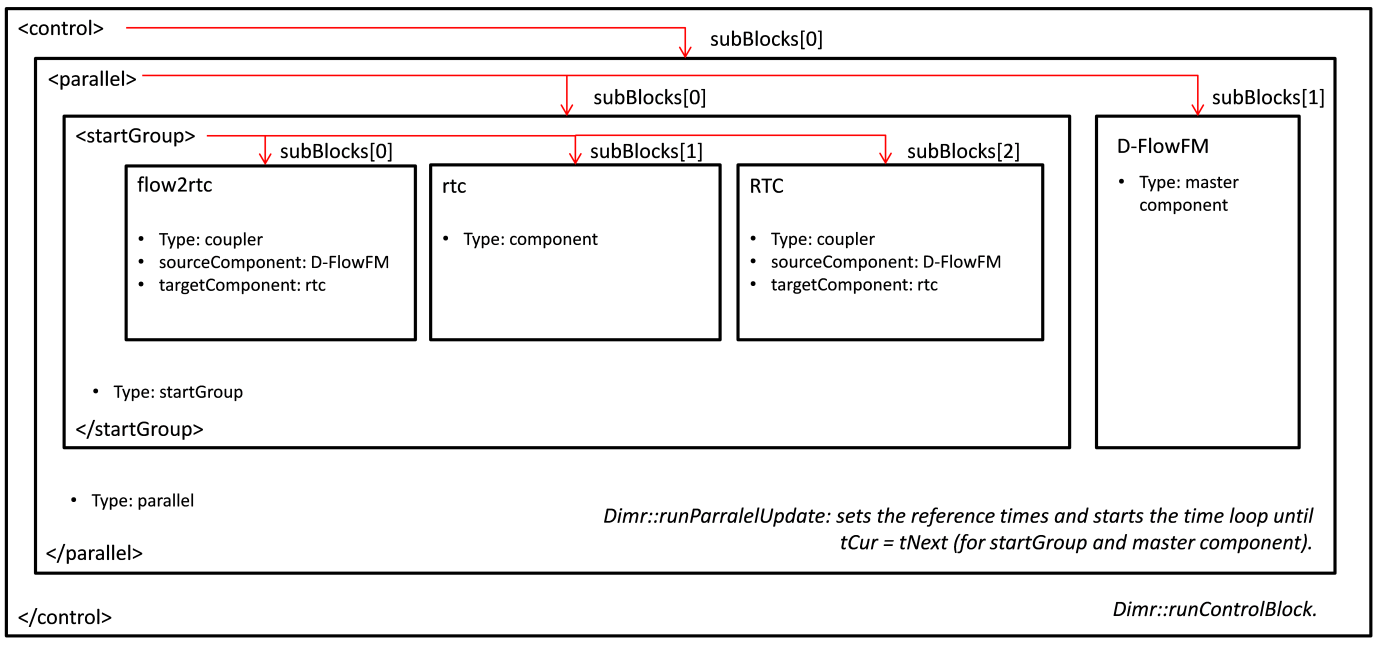
<coupler name="rtc2flow"/>

</startGroup>

<start name="D-FlowFM"/>

</parallel>

</control>



**Figure 2.** A graphical representation of the dimr configuration file snippet shown above.

At the level = 0 the dimr\_control\_block contains the information delimited by <control></control>, its type is sequential and the subBlock array has a size 1 (simply corresponding to the running instance of dimr). At level = 1 the control block is of type parallel and the subBlock array has size 2 with the master subBlock “D-FlowFM” and the child subBlock delimited by <startGroup></startGroup>. At level = 2 the control block is of type startGroup and the subBlock array has size 3, with two couplers (“flow2rtc” and “rtc2flow”) and one component (“rtc”). The most important function responsible to orchestrate the model runs is Dimr::update and relies on this nested structure of the “control” member variable. Dimr::update will be described in section 5.

## The BMI interface

As mentioned above, all components used by dimr implement a BMI-Interface, which consist of a series of global functions available to clients. Dimr\_lib itself implements a BMI interface used by the dimr executable and Delta Shell. The dimr executable does not implement the BMI interface because the executable is responsible for initializing/finalizing MPI (Message Passing Interface), and this can only be done once in the main thread.

In Table 1 the responsibility of each function of the BMI interface is documented. All components in D-HYDRO Suite (D-Flow 1D, D-Flow FM, D-Waves, etc.) implements a BMI interface (although not all of them conform to the BMI standards).

## The Dimr executable

Dimr application is implemented as a separate executable. The methods are encapsulated in the DimrExe class (Figure 3). Here we list the most important ones in order of execution:

1. DimrExe::initialize: is responsible for reading command line arguments and initializing the members of the DimrExe object, such as the exePath, the log file, the path to the configuration file, the clock starting time.
2. DimrExe::openLibrary: is responsible for loading the dimr\_lib library (dirm\_dll.dll), assigning a value to libHandle and to all pointers pointing to the dimr\_lib BMI interface. The types of these function pointers are defined in dimr.h, which is shared by the dimr executable and dimr\_lib.
3. DimrExe::lib\_initialize: passes configuration settings (related to MPI, debug level) to dimr\_lib and initializes dimr\_lib. In the dimr\_lib initialization the configuration file is read, the control blocks are created for the first “level 1” block inside <control> group (“level 0” block), the libraries of each component are loaded and all dimr\_component pointers to the BMI interfaces are assigned (see dimr\_component struct in Figure 1). tStart, tCurrent, tStep and tEnd are obtained from dimr\_lib.
4. DimrExe::lib\_update: performs an update of Dimr for the full simulation interval from tStart to tEnd. Here the core functionality of the Dimr application is implemented and will be detailed in more detail in the next section.
5. DimrExe::finalize: all components are finalized for the first “level 1” block. If there are more “level 1” blocks, they are all initialized, updated and finalized too.

After the library finalization the instance of the DimrExe class is destroyed and MPI is finalized.

## The dimr lib\_update method

The DimrExe::lib\_update is the function executing the model runs, getting the results from source components and setting them to the target components. As explained in the user manual, dimr enables sequential and parallel simulations in three ways:

1. Sequential simulations: component 1 is executed for its full simulation period (first “level 1” block), output is produced, then component 2 is executed (second “level 1” block), optionally using the output produced by simulation component 1. Component 2 cannot influence component 1.
2. Parallel simulations: components 1 and 2 are both started (one “level 1” block), component 1 simulates a time period, exchanges data with component 2, component 2 is executed and exchanges data with component 1. This is repeated until the full simulation period is handled.
3. Parallel simulations using a parallel components: in the examples above, components can run in parallel, using several partitions to perform flow simulation, exchanging data directly via MPI.

Here the cases 2 and 3 are described. For case 2 we refer to the snippet of the configuration file shown below Figure 2.

**Table 1.** The BMI Interface and matching between the BMI interface and the DimrExe function pointers.

|  |  |  |
| --- | --- | --- |
| **BMI interface (Deltares components)** | **DimrExe function pointers to dimr\_lib** | **Function responsibility (for all components)** |
| int initialize(const char \*config\_file) | dllInitialize | Initialize and load the library |
| int update(double dt) | dllUpdate | Advance the component for specified time interval |
| int finalize() | dllFinalize | Shutdown the library and clean up the model |
| void get\_start\_time(double \*t) | dllGetStartTime | Gets the start time |
| void get\_end\_time(double \*t) | dllGetEndTime | Gets the end time |
| void get\_current\_time(double \*t) | dllGetCurrentTime | Gets the current simulation time |
| void get\_time\_step(double \*dt) | dllGetTimeStep | Gets the time step |
| void get\_var\_shape(const char \*name, int shape[MAXDIMS]) | - | Gets the shape of the array (lengths along each dimension) |
| void get\_var\_rank(const char \*name, int \*rank) | - | Gets the rank of the array (e.g. 1 for a vector) |
| void get\_var\_type(const char \*name, char \*type) | - | To be detailed |
| void get\_var\_count(int \*count) | - | To be detailed |
| void get\_var\_name(int index, char \*name) | - | Gets the name of the variable |
| void get\_var(const char \*name, void \*\*ptr) | dllGetVar | Gets a pointer to the variable “name” |
| void set\_var(const char \*name, const void \*ptr) | dllSetVar | Sets the value of the variable “name” |
| void set\_var\_slice(const char \*name, const int \*start, const int \*count, const void \*ptr) | - | Set a slice of the variable from contiguous memory using start / count multi-dimensional indices |
| void set\_logger(Logger logger) | set\_logger\_entry | Sets the logger for the component |

### 5.1 Parallel simulation

The DimrExe::lib\_update of the dimr executable calls the update function of the dimr\_lib (through the BMI interface), which starts processing the control block at level = 1. If this control block is of type “parallel”, it calls Dimr::runParallelUpdate where it identifies a master control block (“D-FlowFM”) and sets the reference time and the time steps of all other control blocks relative to the master (tCur = 0, tNext = end time of the current update, tStep = time step of the master component). After this initialization stage the main time loops begins, with a nested loop over the two entries of the subBlock array at level 1 (see Figure 2). In this loop if the entry index is not masterSubBlockId a loop over the nested entries starts (level 2), in this case over the two couplers and one component. Note that all units (components and couplers) must be started in exactly the same order as they are located in the <control> block.

For the first coupler “flow2rtc” the quantity to be communicated is the water level at a certain location, the source component is “D-FlowFM” and the target component is “rtc”. The task of getting and setting the water levels is accomplished by using the function pointers ddlGetVar/dllSetVar of each component specified in the coupler and in most cases is a three step process (exceptions are D-Flow FM and Wanda, see section 5.3). First the address is retrieved from the source component (Dimr::getAdress), second the value is read (Dimr::send, but could be called differently, e.g. Dimr::getValue) and then the value is set into the target component (Dimr::receive). After these steps on the first coupler, the next control block is processed (“rtc”). In case the subBlock is of type component and dimr simply runs RTC Tools with the updated water level. Once the update is completed the third subBlock “rtc2flow” of type coupler is processed. Here the same steps described for the first coupler are executed, but with the weir level as the quantity to be communicated (the calculated weir level is read from RTC Tools and set into D-FlowFM).

Once the loop over the child control block (level 2) is completed, the reference time and the time steps of the child control blocks are updated (tCur = tCur + tStep) and the next level 1 subBlock is executed (the master component D-FlowFM).

This loop over level = 1 control blocks is repeated until the current time step is equal to the ending time of the current update.

### 5.2 Parallel simulation using a parallel component

When processing one subBlock of type coupler a “transfer array” of size equal to the MPI communicator size is created. This is necessary because the model domain is decomposed in partitions and the address returned in Dimr::getAdress contains a valid value only for the partition where the quantity to be communicated is defined. The entries of the transfer array are set in Dimr::send to a large negative value except for the current MPI rank, where the value to be communicated is set. Then, a reduction operation is performed to maximize each entry of the array and to communicate identical copies of the maximized array to all MPI ranks. The value of the quantity to be communicated is retrieved by finding the maximum value of the transfer array.

The process described above assumes that dimr is also run in parallel with a larger or equal number of MPI processes (MPI\_intialize can be called once by the main thread, in DimrExe::initialize\_parallel). Indeed the MPI communicator initialized in dimr is divided in groups which are assigned to each parallel component. The creation of new MPI groups takes place in Dimr::runParallelInit and is illustrated in Figure 3. It should also be verified if the reduction process of the transfer array is always necessary, because for some components (such as D-FlowFM) the target values are already reduced in rank 0.

<component name=" D-FlowFM">

<library>dflowfm</library>

<process>0 1</process>

<mpiCommunicator>DFM\_COMM\_DFMWORLD</mpiCommunicator>

<workingDir>fm</workingDir>

<inputFile>weirtimeseries.mdu</inputFile>

</component>

<component name="rtc">

<library>RTCTools\_BMI</library>

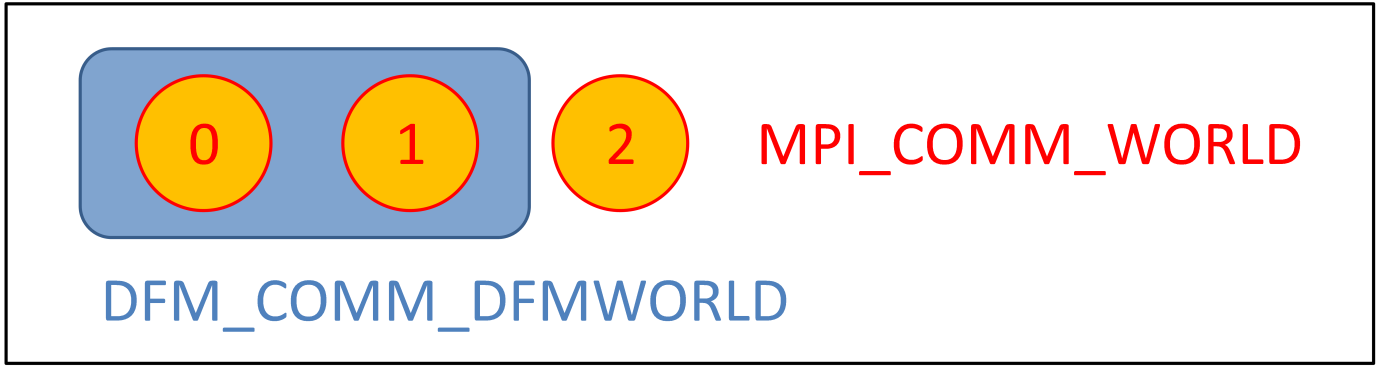
<process>0</process>

<workingDir>rtc</workingDir>

<!-- component specific -->

<inputFile>.</inputFile>

</component>



**Figure 3.** In a parallel run using a parallel communicator, dimr initializes MPI and creates a MPI\_COMM\_WORLD communicator in DimrExe::initialize\_parallel. In Dimr::runParallelInit 2 processes are assigned to “D-FlowFM”, as specified in the configuration file snippet shown above. Note that DFM\_COMM\_DFMWORLD is part of MPI\_COMM\_WORLD.

### 5.3 Components with dedicated handling

In Dimr::getAdress, Dimr::send and Dimr::receive the components type Wanda, RTC and Flow1D do not use pointers to access the actual variables. This dedicated handling increases the complexity of the code by introducing additional if statements inside loops and a series of integer comparisons. This decreases the maintainability of the code, which needs to be modified every time a new component is added. To avoid hardcoded checking for each component, the information about the access type can be stored in the component libraries and retrieved using a get\_var call (e.g. get\_var with name equal to “accessType”).

Note that this solution can be also extended to other parts of the code where the control flow differs based on the component type.

## Possible improvements

In this section we suggest a list of improvement that could be implemented to improve Dimr:

Reduce code complexity

1. Dimr code can be simplified by “asking” the components for the properties (e.g. the type of variable access) responsible for the program control flow, as described in section 5.3.
2. Rename Dimr::send to Dimr::getValue and Dimr::receive to Dimr::setValue.

Correct use of the language/improvements

1. Exception to be thrown should not be allocated using operator new and they should be caught by reference (every memory allocation with the operator new that is not deleted is a memory leak).
2. Even if extra typing, it would be useful to write the dummy arguments in the definition of the methods in the header file (see for example dimr.h). By doing so we, the meaning of the input arguments is clearer.
3. There should be consistency in the procedures used for memory allocation/deletion. Now is a mixture of new/malloc and delete/free.
4. NULL pointers could be compared using nullptr (a real null pointer), because in most cases NULL is an integer literal with value of 0 (a macro such as “#define NULL nullptr” could be used to substitute all instances of NULL with nullptr).
5. The integer returned by BMI functions initialize, update, finalized should be 0 for success and a negative integer for runtime errors. This can be implemented by throwing exceptions with an error code and returning the error code of the caught exception in the BMI function. The error codes can be :
   * ERR\_UNKNOWN = -1
   * ERR\_OS = -2
   * ERR\_METHOD\_NOT\_IMPLEMENTED = -3
   * ERR\_INVALID\_INPUT = -4
   * ERR\_MPI = -5
   * ERR\_XML\_PARSING = = -6
   * ERR\_PTHREADS =-7
6. It should be possible to supress the logging to file when using dimr\_lib because it can affect the runtime. The logging level should also comply with the standards (levels should be Off, Debug, Info, Error, Fatal, see BMI standard [here](https://github.com/openearth/bmi-csharp/blob/master/src/BasicModelInterface/Logger.cs)). Experiments show that a flow simulation managed by dimr can be substantially slower than the same simulation executed from command line (up to 75%). The logging level of dimr logger can be set using set\_var. Similarly we should verify that the logging level can be set for all components.
7. In order to monitor the computed values during a simulation, the get\_var function of dimr\_dll should return a pointer to an array and not to single scalar. This requires some changes of the MPI calls inside Dimr::Send.
8. The starting time of all components should be consistent and standardized as required by BMI. Now for some components the starting time is 0 while for others the start time can be larger than zero.
9. The interface of some components is not fully standardized (e.g. RTC Tools). An effort to standardize all components interfaces accordingly to BMI specifications should be made.
10. The xml files are parsed by xmltree.cpp, an in house C++ class. A better parser such as Xerces-C++ (available [here](https://xerces.apache.org/xerces-c/)) could be used instead and be included in third party open folder. Note that using Xerces does not necessary mean that the xmltree.h/ xmltree.cpp code will be simpler or shorter.
11. In dimr.cpp lines 1413 to 1432 the names of the libraries of each component is hardcoded. This is not necessary because in the xml file a field is already reserved for the name of the component (could be the library name). We can get rid of the hardcoded library names, so when new libraries are created, dimr does not need to be modified.
12. The methods of the Dimr class can be grouped in two classes: the first class handling the reading of the xml file and the second class containing the BMI interface of Dimr.
13. It should be possible to log the values communicated between components during a simulation (e.g. water/weir levels) for debugging purposes, for example in a separate file (the file format needs to be decided, probably will be a NetCDF file).

Possible bugs:

1. The use of the transfer array seems to be unmotivated for the parallel simulation using a parallel component. Because every variable is replicated in each MPI rank, it will be simpler to use only a scalar in place of the transfer array and then call MPI\_Allreduce over all copies of the same scalar.