Deltares Integrated Model Runner

DIMR

Deltares system

Design Document

Deltares

DIMR Design

Design document

Version: 1.2
SVN Revision: 00

17 March 2023

DIMR Design, Design document

Published and printed by:
Deltares
Boussinesqweg 1
2629 HV Delft
P.O. 177
2600 MH Delft
The Netherlands

telephone: +31 88 335 82 73
e-mail: info@deltares.nl
www: https://www.deltares.nl

For sales contact:
telephone: +31 88 335 81 88
e-mail: software@deltares.nl
www: https://www.deltares.nl/software

For support contact:
telephone: +31 88 335 81 00
e-mail: software.support@deltares.nl
www: https://www.deltares.nl/software

Contents

List of Tables

List of Figures

List of To Do's

1 DIMR design

1.1 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 1.2 describes the DIMR library used by the dimr application, section 1.3 describes the common BMI interface implemented by all D-HYDRO models, section 1.4 lists the basic steps executed during a dimr run and section 1.5 describes the Dimr::dimr_update function, which is responsible for executing and synchronizing the model runs. In section 1.6 some improvements are proposed.

1.2 The DIMR library (dimr_lib)

Figure 1.1 shows the simplified class diagram of dimr executable, 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:

1 Component: identifies a simulator (D-Flow FM, D-RTC, D-Water Quality, D-Waves 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.
2 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.
3 Coupler: identifies an entity where 2 components and their respective targets are coupled together (e.g. D-Flow FM + a water level and D-RTC + 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 1.1 and represent the BMI interface of dimr_lib, similarly to other components, as can be seen in Figure 1.1 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 1.2.

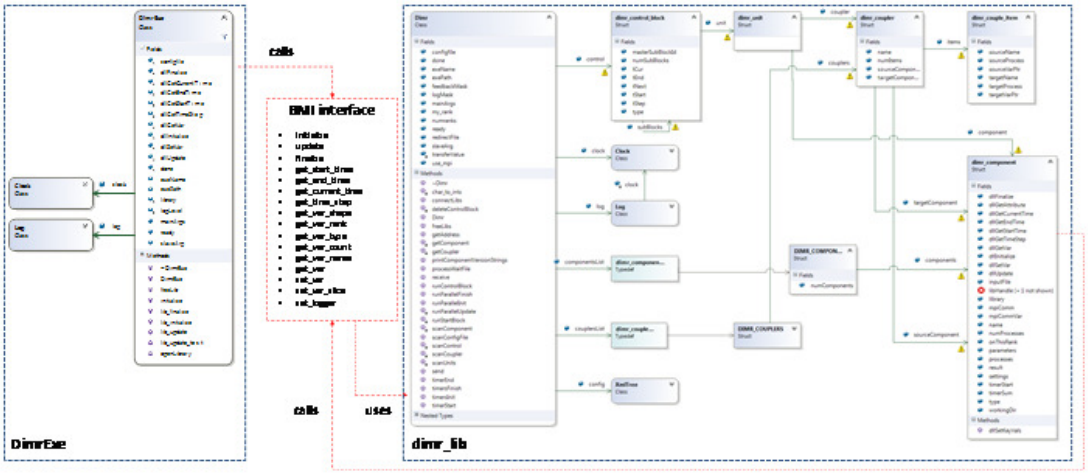


Figure 1.1: *The class diagram of DimrExe (left side) and dimr_lib (right side). DimrExe and DIMR components call the methods of D-HYDRO libraries using the BMI interface (red box in the central part of the figure delimited by a red dashed line)*

.

```
<control> #(level 0)
<parallel> #(level 1)
<startGroup> #(level 2)
<time>0.0 60.0 99999999.0</time>
<coupler name="flow_to_rtc"/>
<start name="rtc"/>
<coupler name="rtc_to_flow"/>
</startGroup>
<start name="dflowfm"/>
</parallel>
</control>
```

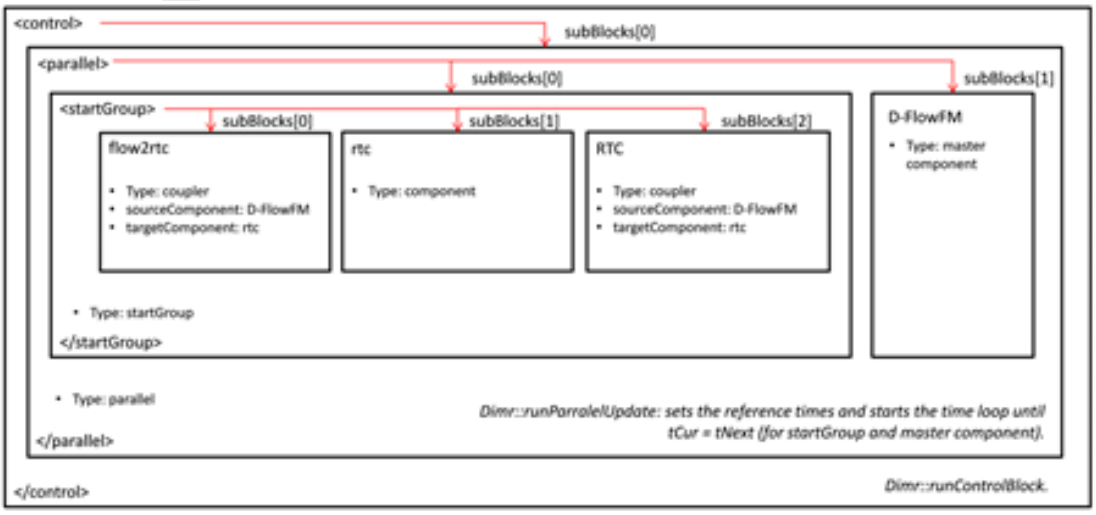


Figure 1.2: *A graphical representation of the DIMR configuration file snippet shown above.*

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-Flow FM” 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 (“flow_to_rtc” and “rtc_to_flow”) 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 1.5.

1.3 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 Delft3D FM and D-HYDRO framework. 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 1.1 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).

1.4 The DIMR executable (DimrExe)

DIMR application is implemented as a separate executable. The methods are encapsulated in the DimrExe class (Figure 1.3). 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 1.1). 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.

1.5 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 1.2.

TODO ***TODO 1.1: Adri Mourits****This should be improved in the current version (revision7501 or higher)*

TODO ***TODO 1.2: Luca Carniato*** *When reading http://bmi-spec.readthedocs.io/en/latest/bmi.control_funcs.html I see not all functions have a BMI compliant signature*

Table 1.1: *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

1.5.1 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-Flow FM”) 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 1.2). 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-Flow FM” 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 1.5.3). 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 D-RTC with the updated water level. Once the update is completed the third subBlock “rtc_to_flow” 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 D-RTC and set into D-Flow FM).

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-Flow FM).

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

1.5.2 5.2 Parallel simulation using a parallel

TODO ***TODO 1.3: Luca Carniato*** *I think I have found a bug for WANDA component, line 1110*

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 1.3. It should also be verified if the reduction process of the transfer array is always necessary, because for some components (such as D-Flow FM) the target values are already reduced in rank 0.

```
<component name=" \dflowfm">
<library>dflowfm</library>
<process>0 1</process>
<mpiCommunicator>DFM\_COMM\_DFMWORLD</mpiCommunicator>
<workingDir>dflowfm</workingDir>
<inputFile>weirtimeseries.mdu</inputFile>
</component>
<component name="rtc">
<library>RTCTools\_BMI</library>
<process>0</process>
<workingDir>drtc</workingDir>
<!-- component specific -->
<inputFile>.</inputFile>
</component>
```

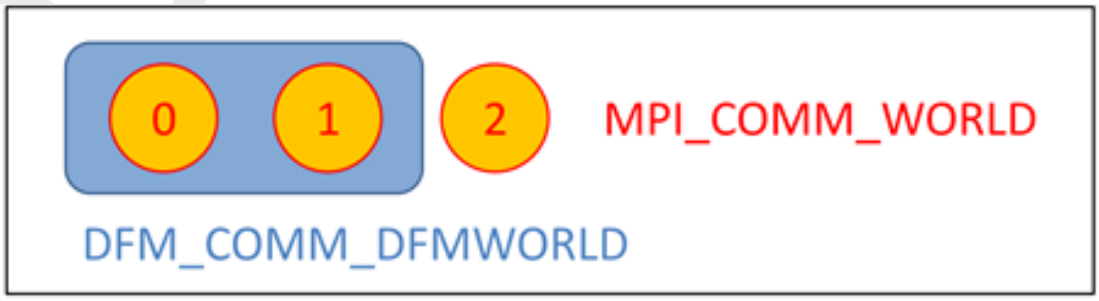


Figure 1.3: *In a parallel run using a parallel communicator, DIMR initializes MPI and creates a MPI_COMM_WORLD communicator in DimrExe::initialize_parallel. In Dimr::runParallelInit 2 processes are assigned to “dflowfm”, as specified in the configuration file snippet shown above. Note that DFM_COMM_DFMWORLD is part of MPI_COMM_WORLD.*

1.5.3 5.3 Components with dedicated handling

In Dimr::getAdress, Dimr::send and Dimr::receive the components type WANDA, D-RTC and D-Flow 1D do not use pointers to access the actual variables.

TODO 1.4: Luca Carniato *If pointers are not used, how the variable is read from memory?* ***TODO***

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 hard-coded 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.

1.6 Possible improvements

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

1.6.1 Reduce code complexity

1 Dimr

TODO 1.5: Adri Mourits *All components need to implement this retrievement, a default must be defined for components from outside Deltares. Big job.* ***TODO***

TODO 1.6: Luca Carniato *We can set a default behaviour in Dimr in case asking the access returns a null pointer.* ***TODO***

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 1.5.3.

2 Rename

TODO 1.7: Adri Mourits *I was afraid that these names were confusing too, but they are much better than send/receive. Do it. Small job* ***TODO***

Dimr::send to Dimr::getValue and Dimr::receive to Dimr::setValue.

1.6.2 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

TODO ***TODO 1.8: Adri Mourits*** *There is a lot going on related to logging. I prefer to keep it as simple as possible.*

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). 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

TODO ***TODO 1.9: Adri Mourits*** *Yes. This is also needed when arrays are going to be exchanged between components.* ***Low priority****.*

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

TODO ***TODO 1.10: Adri Mourits*** *Issue DELFT3DFM-604 This is possibly a bug: high priority*

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

TODO ***TODO 1.11: Adri Mourits*** *OK, big job*

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

TODO ***TODO 1.12: Adri Mourits****OK, big job*

xml files are parsed by xmltree.cpp, an in house C++ class. A better parser such as Xerces-C++ (available here) 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

TODO 1.13: Adri Mourits** OK, smalljob* ***TODO

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.

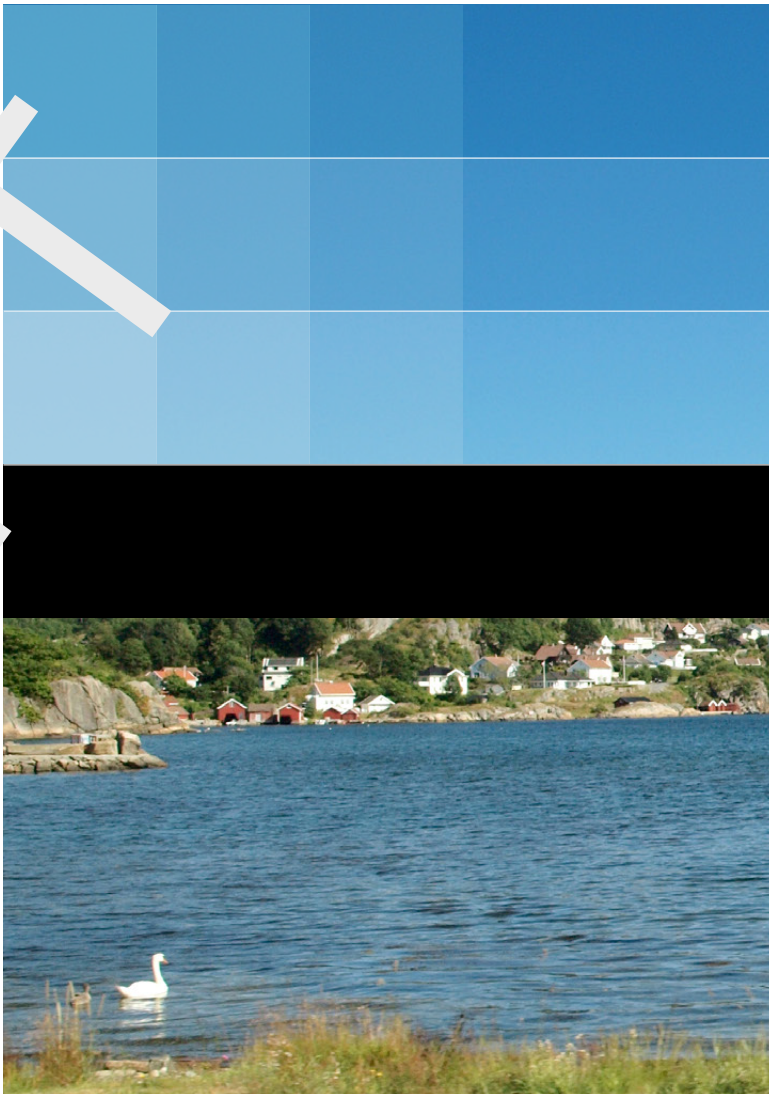
13 It should be possible to log the values communicated between components during a simulation (e.g. water/weir levels) for debugging purposes, for example in a separate file (the file format needs to be decided, probably will be a NetCDF file).

1.6.3 Possible bugs

1 The

TODO 1.14: Adri Mourits** OK, smalljob* ***TODO

. 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.



Deltares systems

PO Box 177
2600 MH Delft
Boussinesqweg 1
2629 HV Delft
The Netherlands

+31 (0)88 335 81 88
software@deltares.nl
www.deltares.nl/software