Coordination of a Heterogeneous Coastal Hydrodynamics Application in Manifold

C.L. Blom, F.Arbab, S. Hummel, I.J.P. Elshof

Software Engineering (SEN)

SEN-R9833 December 1998

Report SEN-R9833
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Coordination of a Heterogeneous Coastal Hydrodynamics Application in Manifold

C.L. Blom

F.Arbab
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

S. Hummel

I.J.P. Elshof
WL | delft hydraulics
P.O. Box 177, 2600 MH Delft, The Netherlands

ABSTRACT

In this paper we show how the coordination language Manifold can be used to control the interactions of multiple heterogeneous application programs. We use a concrete example from Delft Hydaulics, a consulting and research company which develops models of natural hydraulic systems (e.g., river flows, tidal currents, wave penetrations in harbours, etc.). These simulation programs accurately model water flow phenomena and are used for many places in the world. Often, however, a number of these programs need to be used in conjunction with each other to address more comprehensive problems. For example, the water level in the Rotterdam harbour is determined both by the behaviour of the North Sea and by the discharge of the rivers Rhine and Meuse. Instead of creating Unix shell scripts for each particular configuration of application programs, an executive program has been developed and implemented in Manifold that reads a configuration file and subsequently starts, interconnects and controls all relevant component programs.

1991 Mathematics Subject Classification: 65Cxx
1991 Computing Reviews Classification System: C.2.4, D.1.3, G.1.8, I.6.3
Keywords and Phrases: coordination, reusability, parallellism, heterogeneous applications, distributed computation, hydraulic modelling, industrial applications

1. Introduction

Shallow water hydrodynamics is a computation intensive application where a combination of various modeling techniques and algorithms are used to study the real life consequences of a number of natural phenomena. Such studies are of vital importance in various planning and decision making processes in many parts of the world and are required by the regulatory and policy making agencies in many countries. For example, consider the problem of deciding whether or not a factory should be allowed to drain its waste water in a river. Adding the factory outlet to the river has consequences for the river: thermal, chemical, morphological, biological, hydrodynamical, etc. There exist various models for the study of the different aspects of the consequent natural phenomena. However, these different aspects are not independent and they influence each other: thermal changes affect viscosity and thus the nature of the flow; the flow changes the morphology of the river bed by displacing sediments; changes in chemistry and morphology affect (plant and animal) biology of the river; biological changes in turn affect chemistry, morphology, and thermal properties of the river; etc.

An application of this type is heterogeneous in the sense that it employs a number of inherently different mathematical models and algorithms. Furthermore, in principle, the decision on the applicability of a particular model or algorithm may depend on the fluctuations of certain computed values, and may have to be made dynamically. In and near coastal areas, the effects of coastal waves, tides, and their seasonal changes must also be accounted for. This adds another form of heterogeneity to the application in the domain of time: the scale of relevant phenomena (i.e., the time step in modeling) ranges from fractions of a second (e.g., for flow, thermal, and chemical transport) to decades (e.g., the long term effects of tidal waves on the coastal and the river beds).

Founded in 1927 as an independent research and consulting organization, Delft Hydraulics has become well known for its experimental modelling facilities. Throughout the decades Delft Hydraulics has developed and and otherwise acquired a large number of numerical hydraulic modelling programs for various aspects of flow modelling. However, many of these programs have their own input specifications and internal (sometimes hidden) assumptions on their input data consistency.

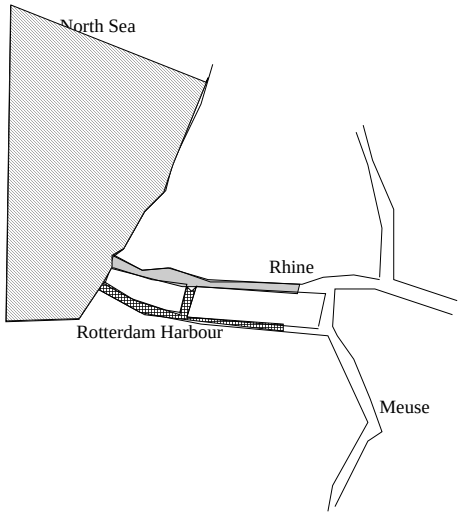

Figure 1: Domain Decomposition of the Rhine-Meuse Estuary.

When these programs are to be used in conjunction with one another in a single application, we can recognize a coordination problem, compounded by their inherent heterogeneity: sometimes the output of one program cannot easily be used as input for another. Another more serious form of this heterogeneity is the grid-boundary consistency problem. Programs that use domain decomposition models often use different grid sizes, orientations, and/or origins. At the common boundaries of their respective regions, then, we observe a mismatch of their grids. In Figure 1, for instance, a possible decomposition for the modelling of the flow phenomena in the Rhine-Meuse estuary is sketched. A promising approach to overcome such heterogeneity is to use special mapper modules to convert from one domain to the other over their common boundary. In the case of the grid inconsistency problem, for example, such mappers ensure that certain parameters such as water levels and discharges (of heat, chemicals, sediments, etc.) remain consistent across the common boundary of two domains, according to the preservation of matter and energy equations.

To facilitate coupling across different domains, Delft Hydaulics has constructed a set of *Mappers* whose task it is to map the output of one program into suitable input of another [3, 5, 4]. Because each mapper is bidirectional, for n programs there are be at most $\frac{1}{2}n(n-1)$ mapper programs, although not all combinations are sufficiently meaningfull to be supported by their own individual mappers each. The number of participating programs and their respective mappers is still increasing. Furthermore, such programs are generally developed as separate "black box" pieces of code, perhaps by independent

groups, with no regards for their potential coupling in some future application. Thus, the concept of *coordination from outside* becomes relevant in practice here.

The IWIM model [1], which offers a paradigm for exogenous control-oriented process coordination, seemes ideally suited for the situation described above. In IWIM, a clear distinction is made between two classes of processes: *workers* and *managers*. *Worker processes* are considered as *black boxes* with regulated openings called *ports* through which *information units* can flow in and out. Furthermore, every process has a virtual antenna through which it can broadcast (or receive) occurrences of *events* to be received (or broadcast) by other processes. Worker processes are supposed to do the job for which they are designed: e.g., do numerical computation, sort or maintain large datasets, etc.

Manager processes on the other hand are supposed to coordinate and control pools of worker processes. Beside the facilities that worker processes possess to communicate with other processes, they also have facilities to create new processes and set-up communication channels among them. In this way, they can set up and break down arbitrary complex networks of communicating processes.

MANIFOLD is a pure coordination programming language that is a realization of the IWIM model. All processes described above (hydraulic modelling processes as well as data mapper processes) can conveniently be regarded as black boxes, and thus used as *worker* processes in IWIM and **MANIFOLD**. A new executive program called HYMAN (HYdra in MANinfold) was written in **MANIFOLD**. The task of this program is to start all necessary processes (for computation and data-mapping), and to interconnect and coordinate them for a particular application run.

In the rest of this paper we describe the most notable elements of this executive program, using excerpts of the **MANIFOLD** source code. Since an introduction to **MANIFOLD** is presented elsewhere (see, e.g., reference [2]), only some additional language elements will be informally explained in this paper.

The remainder of this paper is organized as follows. First, we give an overview of the executive program HYMAN describing the layout of the process table, how this table is constructed, and how it is used during an application run. Then we give an overview of the methods used to hook on the various existing Atomic Processes with minimal effort. Finally, section 4 is the conclusion of the paper.

2. Overview of the executive program **HYMAN**

A typical hydraulic modelling example is depicted in Figure 2.

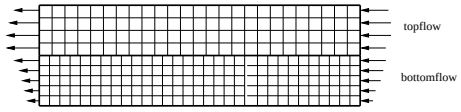

Figure 2: Cross section of a river model with 2 different flow layers.

Here, the flow of a river is modelled using 2 layers (called `topflow` and `bottomflow`) where each is represented by a different finite element grid. For each small element, a number of physical quantities (e.g., velocity, concentrations of chemicals, temperature) can be computed by solving partial differential equations subject to some initial conditions. The results of these computations are valid for some limited time period and will determine the initial conditions for the next time-step. By repeating this procedure, one obtains a series of values for these physical quantities, representing the flow of a river.

For each of these layers, the numerical computations are routinely done by executing existing FORTRAN programs from UNIX shell scripts. Typically, these programs read and write a number of disc files for each time-step.

Although modelling a river by dividing it in layers is a reasonable first approximation, these layers are not completely independent of each other. Therefore, after each step, the input for one layer must be updated with the output of the other by another program called *mapper* which is responsible to ensure that data from one program can be used as suitable for another. For example, it must take into account the difference in grid sizes so that a downwards movement of salt in a grid element of

`topflow` will be proportionally divided over the adjacent grid elements of `bottomflow`.

Further, with the advent of parallel computers, it is advantageous to let these programs run simultaneously as communicating parallel processes. To realize this, synchronization points were added in the exiting FORTRAN programs so that these processes can exchange control messages and data sets.

Next an executive program called hyman was written in the **MANIFOLD** coordination language to set up and control a static network of interconnected processes. We now focus on the structure of this executive program. To realize the example above, a configuration of interconnected processes must be set up as shown in Fig. 3.

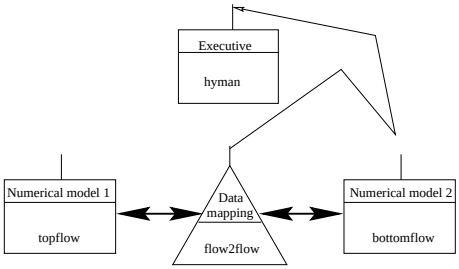

Figure 3: Process Configuration for a 2-layer numerical modelling of flow in a river.

The executive reads the desired process configuration from a file, which contains all specifications for the numerical processes (process names, class names and start-up arguments), followed by all mapper process specifications, and by a list of all bi-directional connections among these processes (called *joins*), e.g:

```
# Example hydra topology file

process topflow flow top.mdf 1 2000
process bottomflow flow bot.mdf 34 500

mapper  topbot flow2flow top.mdf bot.mdf

join topflow topbot
join topbot bottomflow
```

In this example, two computation process instances are defined, with instance names `topflow` and `bottomflow`, both of class `flow`. Furher, one mapper process is defined with the instance name `topbot` of class `flow2flow`. Both computation processes are to be connected with this mapper process. Having read this file, the executive now proceeds to built up the specified network in a number of steps.

First, the computation processes are created and activated while the executive waits for all computation processes to complete their own initialization procedures using barrier synchronization. Subsequently, all mapper processes are created and activated; each of them sends a message to its neighbors (computation processes) and waits for replies. After collecting all replies the mapper checks for convergence and when this happens, it reports that fact to the executive program by calling the barrier synchronization function. Otherwise, it converts datasets from one neighbor to equivalent datasets suitable for another neighbor, while the neighbors in turn wait for their new input.

Meanwhile, the executive program waits for the convergence results of all mappers, again using barrier synchronization. When this happens, final results are reported and all processes are terminated.

We now look more closely at some details of the implementation of the executive program HYMAN. When parsing the process configuration file, a process table is constructed that contains an entry for each process with information fields for process name, status, type, class names, arguments, run-time references, joins and mappings (which joins are connected to which processes).

With this information, the **MANIFOLD** `Main` program can now be understood:

```
467 manifold Main(process args) {
468
469   auto process i is variable().
470   auto process err is integer(noevent,0).
471   auto process processtable is variable(0)[MAX_PROCESSES+1].
472   auto process joins is variable(0)[MAX_JOINS+1].
473   event setup_complete, error_occurred, wait.
474    begin:
475     Message ("Hyman: start reading process configuration file\n");
476     err = ReadConfigurationfile(tuplepick(args,2),processtable,joins);
477     if err < 0
478     then post(error_occurred);
479     CreateProcesses(processtable);
480     CreateJoins(joins,processtable);
481     post(setup_complete).
482   setup_complete: {
483     process numericprocesses homonym
484         ConstructProcessset (processtable, "process").
485     process mappers  homonym
486         ConstructProcessset (processtable, "mapper").
487     begin:
488       ActivateProcessset (numericprocesses);
489       WaitProcessset (initdone, numericprocesses);
490       ActivateProcessset (mappers);
491       post(wait).
492     wait: Message ("Hyman: waiting for barrier.mappers\n");
493       WaitProcessset (barrier, mappers);
494       Message("All mappers reached barrier, checking votes...\n");
495       raise (barrier);    /* request mappers to vote */
496       if CheckMappervotes (mappers) == 0 then {
497 begin: Message("Hyman: at least one mapper voted NO");
498   post(wait).
499       };
500       Message("Hyman: all mappers vote YES, terminating...\n");
501       DeactivateProcessset (numericprocesses);
502       DeactivateProcessset (mappers);
503       deactivate(numericprocesses,mappers);
504       Message ("Hyman: all processes are deactivated now.\n");
505       void.
506     }.
507 end: Message ("Hyman: end.").
508
509 error_occurred:
510   writestr ("Hyman: error %d occurred.\n",err) -> stderr;
511   cancel.
512 }
```

First, some auxiliary processes are instantiated and `auto`-matically activated for internal usage. In particular, on line 471, the `processtable` and on line 472 the join table (`joins`) are created as instances of `variable`. Both tables will be partially filled with information from the process configuration file (line 476).

Next (line 479), all processes specified are instantiated (but not activated) and the table `processtable` is updated: the run-time process references are now known and must be remembered, and for the numerical computation processes shared-memory data segments are created to allow fast exchange of voluminous data. Thereafter (line 480) the joins are created and the join table and the process table are updated with the appropriate run-time information. All joins are created as streams of type KK, and the processes that are their sources and sinks are made permanent event sources for each other and for the executive HYMAN.

Then, the event `setup_complete` is posted and received (481-482) to enter a new block where two new auxiliary processes are created: `numericprocesses` and `mappers`, which are the values returned by the manner `constructprocessset`. This manner scans the `processtable` and delivers a set of

process references of the type specified as its second argument. These sets are then used to startup the whole application in an orderly fashion: first the `numericprocesses`, allowing them to initilize, then wait for the event `initdone` from everyone of them, and then activate the `mappers`. Since the joins where already made, information can now flow between all of these processes and the application is now running.

The coordinating `Main` process now waits for all `mappers` to raise the event `barrier`. If all `mappers` agree on having reached convergence, the application will terminate; otherwise the computations must be resumed, implying for `Main` resuming its `wait` state.

It is illustrative to show how this `manner waitprocessset` is implemented:

```
399 manner waitprocessset (event e, port in prefs) {
400   save *.
401
402   auto process i is variable (0).

403 begin:
404 if dimension(prefs) <= 0
405 then return;

406 while i < dimension(prefs) step INCR(i) do {
407   process p deref prefs[i].
408   begin: terminated(p).
409   e.p: void.
410 };
411 return.
412 }
```

The keyword `dimension` returns the actual number of "elements" in a port array. Since the argument `prefs` is the `output` port array of a variable containing process references, we may dereference each element to obtain a process identfication `p` and wait for the specified event `e` from that process to occur (line 408: the keyword `terminated` waits for the termination of the process specified or any other preemptable event). When this loop terminates, the event `e` of all processes whose references are contained in `prefs` must have been received.

3. Overview of the Application Program Interface

To facilitate the communication between the mapper processes and the numerical modelling processes an Application Programmers Interface had been defined consisting of the following "C" interface function specifications:

```
 1  #include "datatypes.h"
 2  /*-------------------------------------------------------------
 3   *  Hydra Application Programmers Interface functions
 4   */
 5  int Hy_Resume (
 6      int     neighborStep,
 7      int     numout,            /* # of outgoing messages */
 8      HyMesg* outMesg [],        /* outgoing messages */
 9      int *   numin,             /* max/actual # of incoming msgs */
10      HyMesg *inMesg []          /* incoming messages */
11      );

12  int Hy_BarrierMinimum (        /* for mapper processes only */
13      int        value           /* vote (0 => no convergence) */
14      );
15  Hy_StartProcessMain (
16     int         objID,          /* object identifier */
17     char *      objName,        /* object name */
18     char *      configString,   /* configuration string */
19     int         contextID       /* shared memory context ID */
20     );
```

```
21  void Hy_StartMapperMain (
22      int         objID,          /* object identifier */
23      char *      objName,        /* object name */
24      char *      configString,   /* configuration string */
25      int         numNeighbors,   /* number of neighbor objects */
26      NeighborInfo neighbors[]    /* array of neighbor objects */
27      );
```

There is an initialization function that provides an application process (numeric or mapper) with its own process identification (`objID`), process name and startup arguments as specified at instantiation by the MANIFOLD coordinating program (last two arguments were originally derived from the configuration file, as descibed above). In addition, a mapper process gets a description of all other processes to which it is connected (`neighbors`) and for which it must be prepared to receive requests from and send replies to.

Initially, as part of the startup protocol, each numeric process must call `Hy_InitializeProcess`, initialize itself, and then call `Hy_Resume` with argument `neighborStep` set to zero and no output messages. This is interpreted by `Hy_Resume` to raise the event `initdone`, which will be picked up by the MANIFOLD coordinating program. As described in the previous section, the coordinating program activates the mapper processes when all numeric processes have reached this stage. The numeric processes now wait in `Hy_Resume` until incoming messages become available from the mapper to which they are connected (barrier synchronization).

Then, the mapper processes initialize themselves, and in a loop, call `Hy_Resume` to exchange messages with their neighbors. First, outgoing messages are send to all neighbors, so that the numeric processes wake up from their wait-states in `Hy_Resume`. Then, the mapper processes in turn wait in `Hy_Resume` until messages from all neighbors have arrived. The mapper processes have one other essential task: based on the data that is received from all neighbors they must decide whether or not a convergence criterion has been reached. If so, a mapper process must call the interface function `Hy_BarrierMinimum` with its argument set to 1. If it has been established that such a convergence criterion can never been reached with the current data sets, it must call the same function with its argument set to 0, indicating that the computation needs to be restarted on a slightly modified data set with a better chance of reaching convergence.

This function will generate the event `barrier` to be picked up by the MANIFOLD coordinating program, which will then take the appropriate action: either terminate the application or restart it with adapted data.

In the MANIFOLD environment, these functions are implemented using the MANIFOLD Atomic Process Interface functions where the messages and other relevant information is packed in *units* that are sent through the output ports on which the executive has made the appropriate connections. These units can be picked up through the input ports at the other end of their respective joins.

4. Conclusions.

Using the MANIFOLD coordination language, a *process control* application has been realized consisting of a simple static process network. This can serve as a model for other, more complex process control applications where, e.g., dynamic configuration is required.

The amount of modifications to the existing programs is minimal, partly due to the introduction of the mapper processes. In the numerical computation processes modifications are needed only during initialization and whenever data is needed or is produced by calling `Hy_Resume`.

Having realized the application in MANIFOLD, it can now run in parallel, since each process in MANIFOLD may be run on another machine, or in the case of a multiprocessor machine, as a thread on a separate processor.

Using MANIFOLD, threaded applications are easier to write and to maintain, because threading is not to be realized by calling low-level functions. Instead, it is logically implied by the MANIFOLD language, and is effectively realized by the MANIFOLD compiler and its run-time system.

References

1. F. Arbab. The IWIM model for coordination of concurrent activities. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 34–56. Springer-Verlag, April 1996.
2. F. Arbab, C.L. Blom, F.J. Burger, and C.T.H. Everaars. Reusable coordinator modules for massively concurrent applications. *Software: Practice and Experience*, 28(7):703–735, June 1998. Extended version.
3. I.J.P. Elshoff, K.H. Tan, S. Hummel, and M.J.A. Borsboom. Delft-hydra, an architecture for coupling concurrent simulators. In *Parallel Computational Fluid Dynamics, Recent Developments and Advances using Parallel Computers*, Amsterdam, 1998. Elsevier Science.
4. A.E. Mynett, S.Hummel, I.J.P. Elshoff, and H.H. ten Cate. Distributed computing systems in environmental hydroinformatics: Applications in engineering and in education. In *In Proc. of the 3th Int. Conf. on Hydroinformatics*, Rotterdam, 24-26 August 1998. Balkema.
5. S.Hummel, I.J.P. Elshoff, and A.E. Mynett. Distributed engineering systems in coastal zone management. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes in Computer Science*, pages 133–140. Springer-Verlag, April 1998.