ERDC/CHL TR-00-12

US Army Corps of Engineers®
Engineering Research and Development Center

Flood and Coastal Research Program

The Adaptive Hydraulics Software Framework

C. J. Trahan, L. Pettey and G. Savant

January 2016

Approved for public release; distribution is unlimited.

Engineer Research and Development Center

Flood and Coastal Research Program **ERDC/CHL TR-00-12**
January 2016

The Adaptive Hydraulics Software Frame-work

C. J. Trahan

Information Technology Laboratory
U.S. Army Engineer Research and Development Center
3909 Halls Ferry Road
Vicksburg, MS 39180-6199

L. Pettey

Engility
U.S. Army Engineer Research and Development Center
3909 Halls Ferry Road
Vicksburg, MS 39180-6199

G. Savant

Dynamic Solutions, LLC
6421 Deane Hill Dr. Suite 1.
Knoxville, TN 37919

Final Report

Approved for public release; distribution is unlimited.

Prepared for U.S. Army Corps of Engineers
Washington, DC 20314-1000

Abstract: The Adaptive Hydraulics (AdH) software suite, developed in collaboration between the Coastal and Hydraulics and Information Technology Laboratories, has been one of the flagship numerical tools for hydrodynamic modeling within the USACE-ERDC community for a number of years. AdH is a multi-physics hydraulic suite comprised of internal models for 2D and 3D shallow water, Navier-Stokes, groundwater and sediment/salt transport. Despite it's long years of modeling success, the legacy AdH framework has, for some applications, proven to be memory inefficient on HPC platforms, insufficient for rapid development cycles often required for mission critical projects and arduous to extend to the more complex model coupling needs of its user-community. This report documents a complete restructure of the legacy AdH suite to rectify these historical issues. The new AdH version 5 software provides a highly modular, portable and extendable framework, with special focus on HPC efficiency and facilitating future work in model-to-model coupling applications, a modern trend in numerical modeling.

Disclaimer: The contents of this report are not to be used for advertising, publication, or promotional purposes. Citation of trade names does not constitute an official endorsement or approval of the use of such commercial products. All product names and trademarks cited are the property of their respective owners. The findings of this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

DESTROY THIS REPORT WHEN NO LONGER NEEDED. DO NOT RETURN IT TO THE ORIGINATOR.

Contents

Figures and Tables

Figures

Tables

Preface

This report gives a detailed description of the new Adaptive Hydraulics software suite (AdH) version 5 framework. This framework required a complete restructuring of the legacy AdH software to a more efficient suite than can be straightforwardly applied to either split-operated or monolithic multi-model AdH internal coupling applications. This investigation was conducted from October 2014 through October 2016 at the U.S. Army Engineer Research and Development Center (ERDC) by Dr. Corey Trahan of the Information Technology Laboratory, Dr. Lucas Pettey of Engility and Dr. Gaurav Savant of Dynamic Solutions LLC.

This report is published as a product of the Flood and Coastal Storm Damage Reduction Program of the ERDC, Vicksburg, MS. Dr. Cary Talbot was the Program Manager, and William Curtis was the Technical Director.

At the time of publication of this report, Dr. Jeffery P. Holland was Director of ERDC, and LTC John T. Tucker III was Acting Commander.

Unit Conversion Factors

Multiply	By	To Obtain
degrees Celsius	1.8C+32	degrees Fahrenheit
meters	3.2808399	feet
cubic meters	35.31466247	cubic feet
square meters per second	1000000.	centistokes
cubic inches	1.6387064E-05	cubic meters
cubic yards	0.7645549	cubic meters
radians	57.29578779	degrees (angle)

1 Introduction

The Adaptive Hydraulics (AdH) software suite, developed in collaboration between the Coastal and Hydraulics and Information Technology Laboratories, has been one of the flagship numerical tools for hydrodynamic modeling within the USACE-ERDC community for a number of years now. AdH is a multi-physics suite comprised of 2D and 3D shallow water (SW) models, a 3D Navier-Stokes model, models for sediment and salt transport, etc. Each of the models is solved using a continuous Galerkin finite element engine with streamline upwind Petrov-Galerkin stabilization. The software supports triangular mesh discretization in 2D, tetrahedral meshes in 3D and implements up to second-order implicit time-stepping. As the name implies, AdH is both adaptive in space and time and supports a host of features necessary for typical engineering applications, such as hydraulic structures for flow control, friction and turbulence libraries, contaminant transport linkages, sediment process libraries, vessel propagation via pressure lids, etc. The internal finite element engine supports typical hydraulic boundary conditions and all external forces required for riverine, estuarine, lake and coastal applications. Lastly, AdH, being extremely portable, can be built serially on local machines or in parallel for either local or HPC applications.

The objective of any software framework is to improve the efficiency of both existing code enhancements and new feature additions by facilitating developer productivity while maintaining the software's quality, reliability and robustness. Developer productivity is improved by allowing developers to focus on the unique requirements of their application instead of spending time on application infrastructure. Specific advantages of a software framework include: (1) the ability to re-use code that has been pre-built and pre-tested, (2) an increase in reliability of the new application, subsequently reducing the programming effort, testing effort and time to market and (3) a recipe/guideline for better and more consistent future programming practices. Creating a framework, however, can be difficult and time-consuming, and the learning curve for a new developer can be steep for complex frameworks. That said, the benefits of a solid framework far outweigh these disadvantages for large commercial software suites with ongoing development plans. The AdH suite greatly benefits from a coherent framework due to its multi-model capabilities and its general user-trend towards more complex applications. At any given time, various teams with an array of expertise may be developing models within the AdH suite. Because of this, it is vital that a common development philosophy be maintained by following standard protocols when declaring variables, creating binary builds, etc.

Despite it's long years of modeling success, the legacy AdH framework has suffered from slow development cycles and has shown to be difficult to extend to the more complex modeling needs of its user-community. The original framework was established decades ago, and at that time, the extent of AdH's future usage was unknown. Today, the software is used nationwide and deeply engrained into ERDC's hydraulic modeling practices. That said, many mission critical, complex applications using the original AdH, particularly in 3D, were suffering from common problems associated with poorly built or outdated frameworks. Some of these problems, for example, included having to use twice as many HPC cores for sufficient application memory arising from development shortcuts, hacked bug fixes and feature additions and an overly difficult/complex front-end. Preventable issues such as these end up costing the ERDC funds that instead should be used to enhance existing software or create new software all-together. Additionally, as mentioned, AdH legacy framework issues were all but preventing the suite from being extended to more complex modeling-coupling scenarios. Modern trends in numerical modeling tend toward code-coupling as a way to capture multi-scale solutions across expansive domains. Since AdH internally includes a host a hydraulic models valid at different geographic scales, the suite has the potential to model a wide array of memory-coupled problems without linking to outside software. For example, a 3D SW model coupled to a 2D SW model would allow for moving boundaries within the 3D application domain. Another example might be a 3D SW model coupled to a 2D bed load transport model for 3D sediment applications. It is easy to see the utility of such coupling capabilities. This type of model-to-model coupling within AdH was attempted previously, only to find that the suites poor design (such as its global variable scoping, for example) prevented a clean coupling on a reasonable development schedule. It is for these reasons, and many more, that a complete re-design of the AdH framework was undertaken.

This technical report focuses on the new AdH software framework, AdH version 5 (AdHv5). The new framework was built with one eye towards its potential and future use and another on modern software practices, such as code and build modularity, minimal code redundancy, minimal variable scope and minimal dependencies of data types. The restructuring of AdH will reduce both application and development turn-around time for bug fixes, feature additions and engine enhancements, since as history shows, when software becomes bloated, redundant and inconsistent, inefficiencies in both model applications and development arise as developers look for short-cuts to minimize an overly arduous development and application cycle. This report is intended to be a road map for future development so that history does not repeat itself. Any active developer of the AdH suite will find this document a helpful guide to both the design philosophy and historical motivation underlying where and why specific design components are in place.

2 AdHv5 Design Strategy

The AdHv5 framework is built as a callable application with three distinct callable units; (1) model initialization, (2) model run and (3) model finalization. In this way, AdH can be classified to be model compliant with the Earth System Modeling Framework (ESMF). A call graph of the upper level AdH routines is given in Fig. [1].

The design of the new AdH framework focuses on the following items:

- Modularity
- Data Storage and Implementation
- Quality Assurance
- Debug-ability
- High-Performance Computer Efficiency and Scalability
- Front-end Applicability

Before embarking on the new design, these items were determined to be, either directly or indirectly, fundamental for the future success of the AdH suite. As mentioned, through its many years of rapid turn-around development, the legacy AdH framework had lost focus on most of these items, resulting in a bloated code that is both hard to debug and enhance. Our aim was to create a framework that optimized these focus areas in a straightforward way so that future development could easily do the same. The remainder of this report details why and how these items have been addressed in the AdHv5 framework.

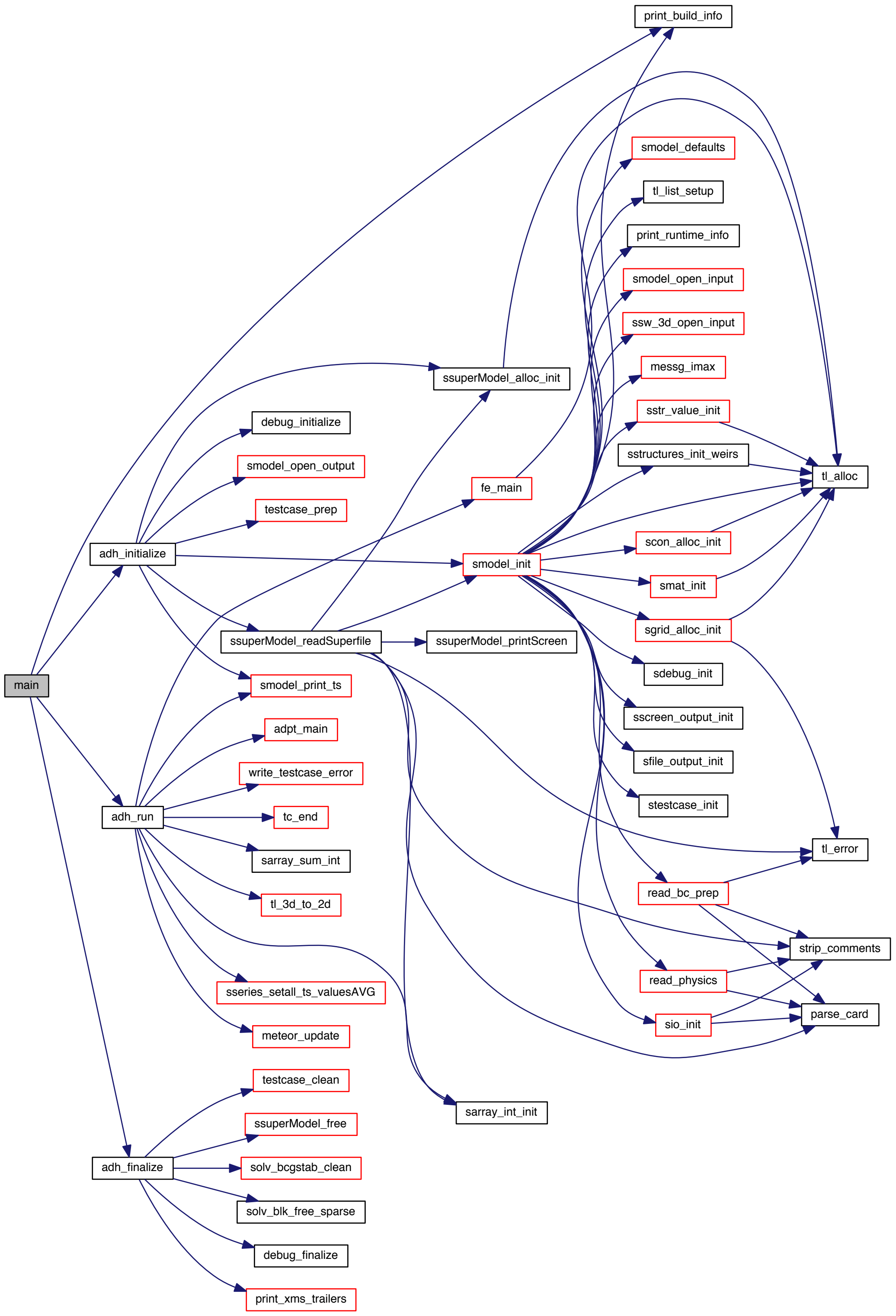


Figure 1. The AdHv5 function call graph.

3 AdHv5 Framework Modularity

A solid framework is designed with as much modularity as possible, particularly for multi-model/multi-physics software suites. This separation facilitates parallel development, plug-and-playability, quick application-dependent builds and general physics/model independence. Although an attempt at modularity in the legacy framework was made, a number of fundamental design issues bottlenecked its progress. The most notable of these were globally scoped/shared variables and non-modular fundamental data-types. These fundamental scoping/modularity issues are what inevitably prevented the legacy AdH suite from being efficiently used to couple internal models, for example. Additionally, due to the mostly non-modular nature of the legacy framework, AdH has a history of non-formal version splitting from one development team's fear of polluting other internal AdH models. This software splitting carries a number of detrimental effects, the most prominent being that development of new, generally useful features, enhancements, etc, on the alternate versions of the software are almost never added to the original software, and as support/funding for the rogue code dies, these costly enhancements are lost. This fear of polluting other models diminishes as the software becomes more modular, and any development which may be useful for other models is naturally inherited and henceforth supported/updated by the whole AdH team. In a nutshell, software modularity works to focus limited resources such as development, marketing and funding. (Note: At the time of this report, AdHv5 only officially supports SW 2D, SW 3D and transport models; however, the Navier-Stokes model port into AdHv5 is nearly complete, and future porting of the groundwater model is being discussed.

All software modularity first depends on the fundamental manner in which variables are scoped. Once this is in place, file-modularity should fall into place, with binary build modularity subsequently following. The AdHv5 framework was built with each of these types of modularity in mind. Implementation of each of these in AdHv5 is now discussed.

Variable Modularity

Previously, all AdH variables had global scoping, causing a host of development problems, particularly when coupling internal models. Generally speaking, even though they are easy to implement, global variables should be avoided at all cost for the following reasons:

- **Read Transparency** – Source code is easiest to understand when the scope of its individual elements are limited. Global variables can be read or modified by any part of the program, making it difficult to ascertain their use.

- **No Access Control or Constraint Checking** – A global variable can be accessed or set by any part of the program, and any rules regarding its use can be easily broken or forgotten.
- **Implicit coupling** – A program with many global variables often has tight couplings between some of those variables and functions. Grouping coupled items into cohesive units usually leads to better programs.
- **Memory allocation issues** – Some environments have memory allocation schemes that make allocation of global variables tricky. This is especially true in languages where ”constructors” have side-effects other than allocation (because, in that case, you can express unsafe situations where two globals mutually depend on one another). Also, when dynamically linking modules, it can be unclear whether different libraries have their own instances of global variables or whether the globals are shared.
- **Testing and Confinement** - Software with global variables are somewhat more difficult to test because one cannot readily set up a ‘clean’ environment between runs. More generally, source that utilizes global services of any sort that aren’t explicitly provided to that source is difficult to test for the same reason.

C-structures/datatypes are at the core of the new framework. These structures have been modularly crafted to store appropriate variables for a given type of physics, grid or finite-element engine component. Nearly all globally scoped variables were removed in the new framework and stored into these structures. For example, all SW 2D variables, such as water depth, velocities, etc. are stored in a shallow water 2D structure. In fact, physics structs in the new framework are cascading, in the sense that both the 2D and 3D shallow water struct are variables within the over-arching shallow water struct. This way, both 2D and 3D can share similar variables/methods to minimize redundancy when need be. Details on the most fundamental AdHv5 structs used are discussed in Chapter 4 of this report.

File Modularity

Once variables are appropriately scoped, the next level of software modularity is in file/source design. Legacy AdH has a history of “cross-contaminating” physics in files, all the way from the front-end to the model engine (though some effort was made to avoid this in the engine). In the legacy front-end, for example, input for all models/physics was read by one file. An AdH Navier-Stokes developer would need to touch this file for any front-end changes with respect to their model, which would have global affect in all models if a bug was introduced. Generally speaking, there is little file-modularity in the legacy framework, which as discussed, can lead to software “stemming” to rogue versions. AdHv5 goes to great lengths to keep all physics as file-modular as possible so that parallel development on different models within the suite can be done with minimal interference. A great example of this is the sediment transport library within AdH. Previously, all sediment input and variables were tightly wound to the SW 2D/3D hydraulic portions of the code. AdHv5, however, is modularized so that sediment is completely dissociated from other models. For example, all sediment inputs are read via their own AdH input file, stored in their own struct, all calls to sediment routines are wrapped in preprocessor #ifdefs and all finite element calls specific to sediment are stand-alone.

It is worth mentioning that although great lengths have been taken to modularize AdH, it can be advantageous to have some file/source overlap to minimize code redundancy. Any cross-contamination in the new framework has been carefully scrutinized and constructed to maintain compilation modularity (mostly through preprocessor wrappings). Developers of AdH should weight heavily in favor of file-modularity when making such a decision.

Build Modularity

Some effort was made in the legacy framework to modularize AdH by wrapping physics-specific sections of the code in C-preprocessor ”#ifdefs”, which are only included in the executable by turning on the appropriate macros in the GNU makefile. AdHv5 also uses the same macro-usage, however, CMake has been adopted in AdHv5 for building the executable. CMake is cross-platform free and open-source software for managing the build process of software using a compiler-independent method. It is designed to support directory hierarchies and applications that depend on multiple libraries. It is used in conjunction with native build environments such as GNU Make, Apple’s Xcode, and Microsoft Visual Studio. It has minimal dependencies, requiring only a C++ compiler on its own build system.

CMake has many advantages over GNU Make, some being

- Cross platform discovery of system libraries.
- Automatic discovery and configuration of the toolchain.
- Easier to compile your files into a shared library in a platform agnostic way, and in general easier to use

In addition to these strengths, CMake provides a simple and elegant terminal interface where macros for C-preprocessors can be easily switched on/off and multiple builds can be constructed. It should be noted that this type of build modularity can be used to create binaries of AdH with only selected physics/functionality when need be.

In order to build AdH, some external software may be required. A detailed description of what libraries are needed and how to link them is given in Appendix

[A].

4 AdHv5 Data Storage and Implementation

At the heart of the new AdHv5 framework are C-structures ("structs") which organize/modularize data storage. These structs are created and stored according to the following rules:

- both structure headers and c-filenames begin with the letter "s"
- all AdH structure files are located in the /structs AdH directory
- structure header files are be wrapped in #ifndef H_S[filename]_ so as to only define it once during compilation
- when possible, structure header files include the struct type-definition and any function calls (methods) which only depend on the struct and/or the AdH fundamental types. When not possible due to complex type-dependencies, the function prototypes should be placed in fnctn_structs.h.
- the structure c-file should contain all the function calls associated with the struct
- All new structures must be added to /include/type.h, placed in the appropriate dependency-location with a list of their dependencies.

With regards to "AdH fundamental types"; these are structure type definitions with minimal dependencies. At the lowest level, all C-fundamental types are included in the definition. Additionally, AdH types that only depend on these C-fundamental types are also included. These AdH fundamental types are listed at the top of the type.h file. It is necessary to distinguish these due to the cascading dependency of the structures during the build. AdH types with complex dependencies on many other AdH types should be built last and are not easily ordered or organized. It is for this reason that structure-function calls should be designed with (1) as minimal inputs as possible and (2) with as few AdH-types as possible (except, of course, the actual structure).

An AdHv5 struct collaboration graph can be seen in Fig. [2]. Collaboration graphs such as these show (1) the inheritance relations with base structs and (2) the usage relations with other structs (e.g. struct A has a member variable m_a of type struct B, then A has an arrow to B with m_a as label). Collaboration graphs will be used regularly throughout this report.

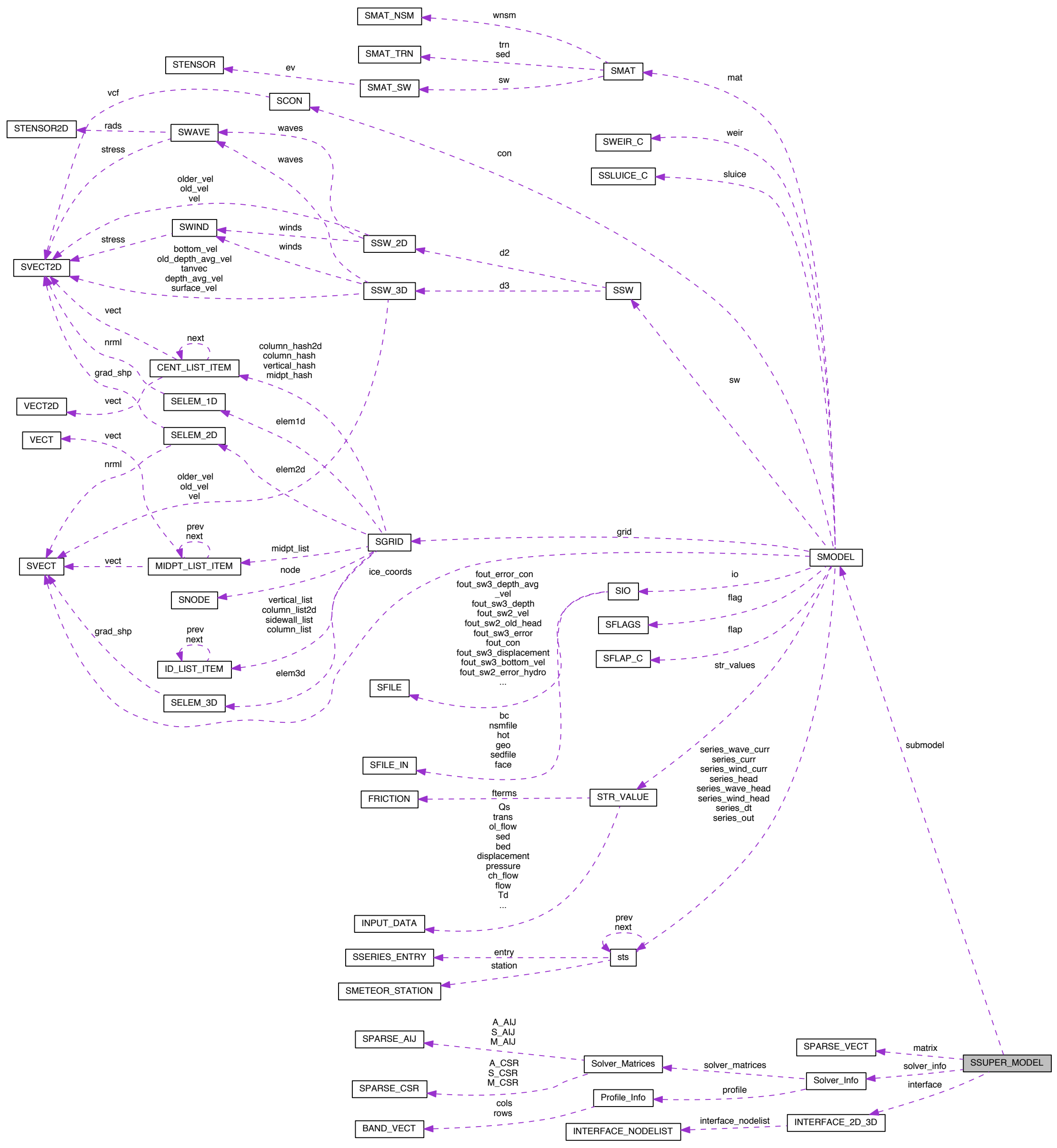


Figure 2. The AdHv5 struct collaboration graph.

As can be seen from this graph, there are many structures that comprise the AdHv5 framework. However, there are a few that lay at the heart of the suite and are worth discussing in detail. We focus on these for the remainder of this chapter. We start with from the top and work down the collaboration hierarchy.

The “*ssuper_model*” struct

At the top of the hierarchical graph given in Fig. [2] is the *ssuper_model* struct. This structure was designed to facilitate model coupling within AdH. In general, there are two methods of internal (memory) model-to-model coupling; (1) models can be coupled via passing fluxes across a shared boundary or interface and (2) models can be coupled by solving all domains together as one large, monolithic linear system. Some advantages to the former include; (1) the ability to solve smaller, independent model matrices instead of one large system, (2) each model can progress with its own time-step and (3) algorithmic simplicity. However, these advantages come at a cost, as flux-exchange coupling methods can suffer from significant split-operator errors in some applications. These errors are eliminated when solving all the models as one monolithic system at the cost of what could potentially be a significant increase in the computational overhead of the application.

Given the advantages/disadvantages of both of these coupling methods, the new framework has been set up to allow both coupling types through a super/sub-model arrangement. The over-arching *ssuper_model* struct stores an array of *smodel* structures (discussed in the next section). The *smodel* instances within a *ssuper_model* instance can be thought of as submodels for a given *ssuper_model*. In some applications, more than one *ssuper_model* struct may be allocated, each containing an array with some number of *smodel* instances. All submodels within a supermodel are solved monolithically, while coupling between supermodels is achieved by flux exchange. A pictorial example of a hypothetical set-up with N supermodels, each containing n_N submodels is shown in Fig. [3].

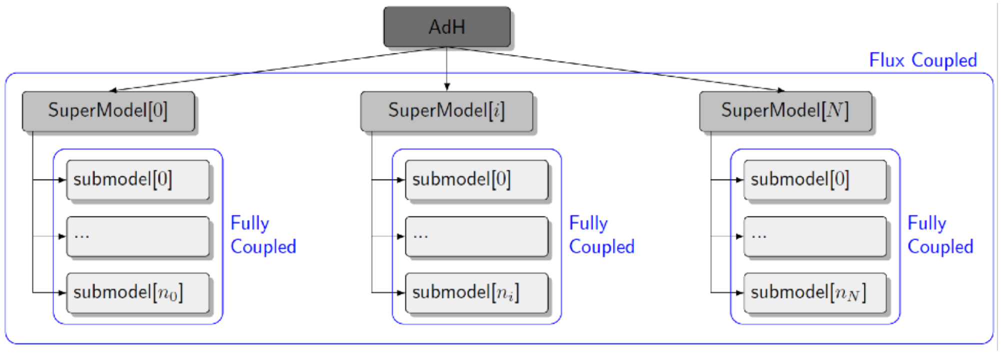


Figure 3. A hypothetical set-up N supermodels, each containing n_N submodels.

Since a given instance of a *ssuper_model* stores all models which are to be solved as one system, this struct must contain the individual models, the nonlinear solver information, storage components for the linear system and the model-to-model interface data for reducing the linear system. A level one collaboration graph of this structure is given in Fig. [4]. Note that the graph level indicates how many connections are displayed. In this case, level 1 means only one connection is shown. Also, red borders indicate the structure is connected to additional structures (higher levels) not shown in the graph.

Every AdHv5 application has at least one supermodel allocated which contains at least one submodel. This is true even when no coupling is used, at which point the user runs AdH as normal, by typing the binary with the root project name argument.

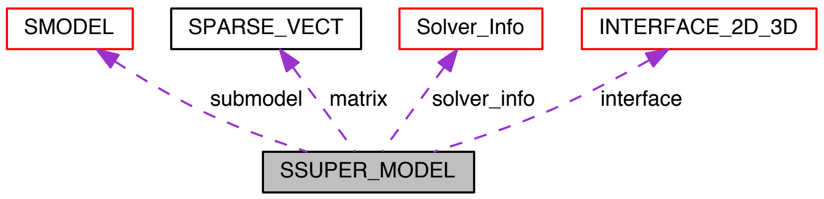


Figure 4. The level 1 *ssuper_model* collaboration graph. The graph level indicates how many connections are displayed. In this case, level 1 means only one connection is shown. Red borders indicate the structure is connected to additional structures (higher levels) not shown here.

In coupling applications, however, the user must specify a “superfile”, which details all models to be ran in the simulation, how they are coupled, etc. More details on the superfile will be given later in this report.

The “*smodel*” struct

At the heart of every AdHv5 model simulation is at least one instance of the *smodel* struct (or submodel within a supermodel). Currently, this struct is designed to hold all information for one particular model within AdH, including the model parameters, grid, physics, etc. An array of these structs can be simultaneously created and executed, each model instance having its own input files as designated by a “superfile”, which details how many models will be executed, their input file names and locations, which supermodels they belong to, etc. All non-globally scoped variables dealing with a specific model are located within structs nested into this *smodel* struct. This provides a good guideline to when one can actually define a variable with global scope in the new framework: if the variable is not model specific, then it is probably acceptable for global scoping. Please note though, global scoping is still discouraged and should be used only when completely necessary.

A level one collaboration graph for the *smodel* struct is given in Fig. [5]. It can

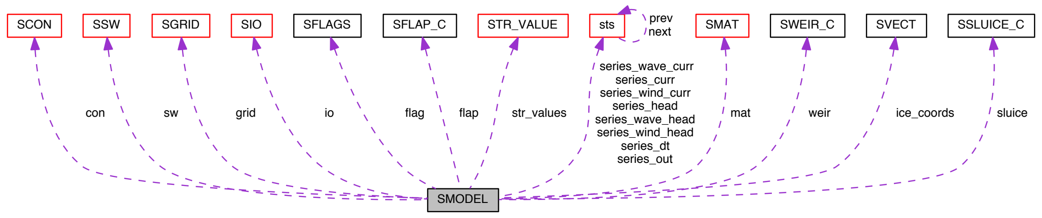


Figure 5. Level 1 *smodel* collaboration graph.

be seen from this graph that the *smodel* struct serves as a catch-all for all potential physics and any number of grids. Which physics and grid structs become allocated depend on user input for that model. A hypothetical layout of the *smodel* struct, for example, can be seen in Fig. [6]. This figure shows an *smodel* instance that describes a 2D and 3D shallow water simulation that transports both one regular constituent and one sediment constituent. Two grids are allocated for this instance, one for the 2D shallow water domain and one for the 3D.

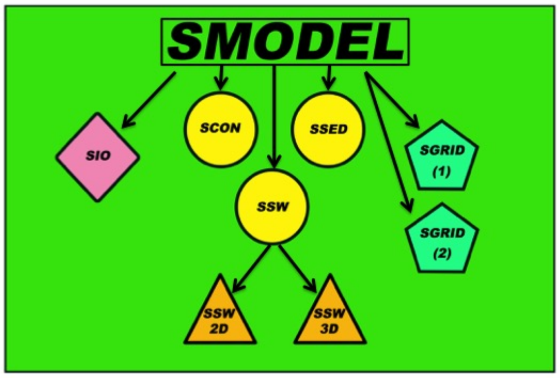


Figure 6. Diagram of the possible datatypes allocated within the AdHv5 *smodel* structure.

Figure [5] shows a number of structures nested within the *smodel* struct, and this number will likely grow in time, however, only a few are commonly used in most AdH applications and will be discussed herein. These are the *sgrid*, *ssw* and *scon* structures.

Of particular note before proceeding to deeper AdH structures is model parameter initialization. Initializing parameters was a scattered process in the AdH legacy framework, subsequently leading to many bugs over the years. In AdHv5, all model parameters are collectively initialized in the structs/smodel_defaults.c file. The following rules are in place for this file when initialize variables of the *smodel* structure:

- All arrays should initially be set to NULL
- the “OFF” and “FALSE” macros are defined as integer 0, whereas the “NO” macro is equal to integer -3. The later can be used as a flag for not-defined.
- the “UNSET_INT” macro is set to integer -3 and can be used to flag the variable as undefined
- the “UNSET_FLT” macro is set to float -3.0 and can be used to flag the variable as undefined

The “*sgrid*” struct

In the legacy AdH, variables related to the grid were un-contained and globally shared. Because of this, it was, for example, impossible to instantiate more than one set of grid variables at a time, as needed for multi-model applications. To rectify this, AdHv5 was framed so that all grid variables for a given model are stored in the *sgrid* struct. Many formats for this structure were considered. While using separate storage types for 2D and 3D grids can be useful, it was determined that a more general grid container was best to generalize the C-function grid-related calls. The *sgrid* struct was therefore created as a container for all 2D and 3D AdH grid/geometry related variables. This includes anything not related to model physics, such as element connectivities, 3D column information, 2D/3D maps, 1D/2D/3D elements, nodes, material IDs, etc. Also included within the *sgrid* struct are all MPI communication variables. The general structure of the *sgrid* struct can be seen in the level 1 collaboration graph shown in Fig. [7].

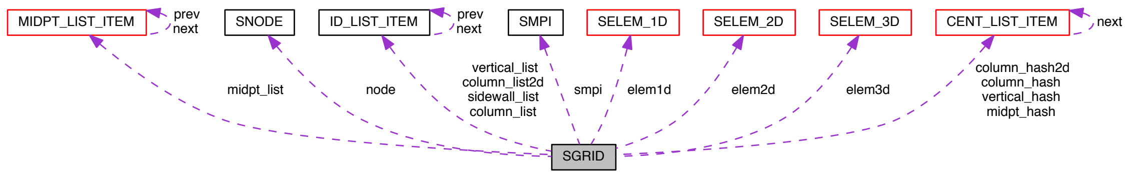


Figure 7. Level 1 *sgrid* collaboration graph.

All *sgrid* allocation and initiation occurs in the *sgrid_alloc_init* routine of the sgrid.c file. A call graph of this routine can be seen in Fig. [8]. This figure shows that the

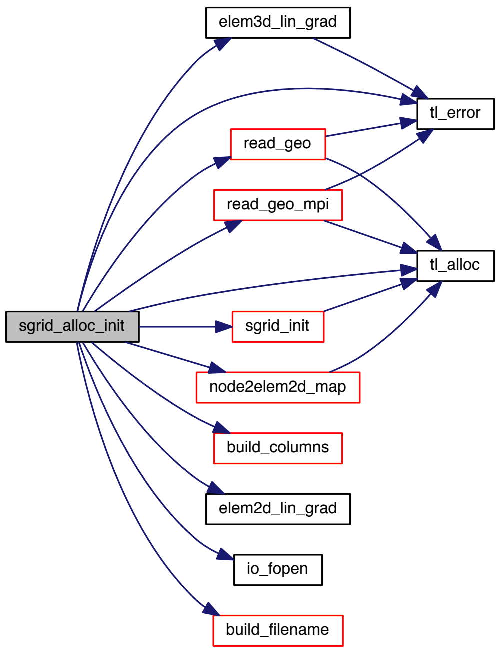


Figure 8. The *sgrid_alloc_init* routine call graph.

sgrid_alloc_init routine not only initializes and allocates the *sgrid* struct (*sgrid_init*), but also is responsible for reading the *3dm* geo file for storing grid data (*read_geo*). The routine also calculates elemental Jacobians (*elem2d/3d_lin_grad*) and builds the 3D columns when necessary.

The "*ssw*" struct

All model physics variables related to both 2D and 3D shallow water dynamics are stored in the *ssw* struct (collaboration graph given in Fig. [9]). This is a simple

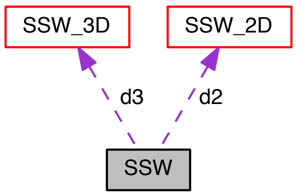


Figure 9. Level 1 *ssw* collaboration graph.

container which holds pointers to one or more *ssw_2d* or *ssw_3d* structures. We now discuss these two sub-structures.

The "*ssw_2d*" struct

This structure holds all solution variables for the 2D shallow water model (see Fig. [10]), including 2d arrays for heads, velocities and wind/wave data. Initialization and allocation of these structure occurs in the *ssw_2d_alloc_init* routine. A call graph of this routine is given in Fig. [11]

The "*ssw_3d*" struct

This structure holds all solution variables for the 3D shallow water model (see Fig. [12]), including 2d arrays for heads and wind/wave data and 3d arrays for velocities. Initialization and allocation of these structure occurs in the *ssw_3d_alloc_init* routine. A call graph of this routine is given in Fig. [13]

The "*scon*" struct

The *scon* structure is designed to contain all variables related to constituent transport in AdH. Figure [14] displays the level 1 collaboration graph for this struct. Initialization and allocation of these structure occurs in the *scon_alloc_init* routine. A call graph of this routine is given in Fig. [15]

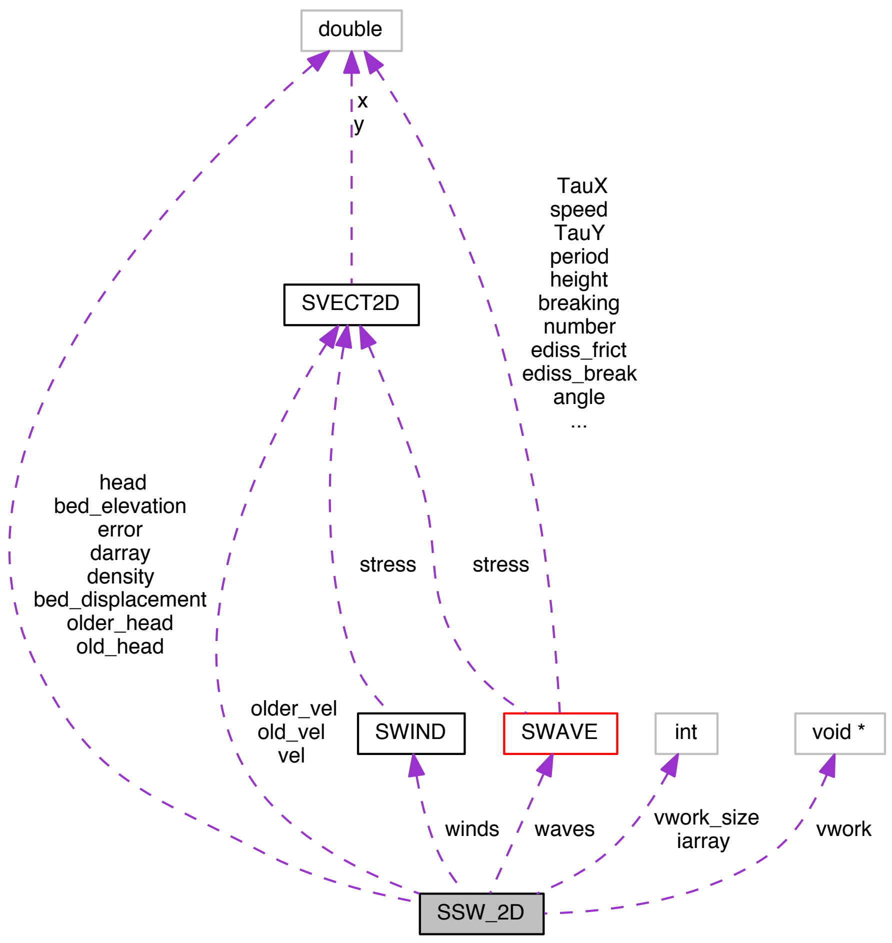


Figure 10. Level 1 *ssw_2d* collaboration graph.

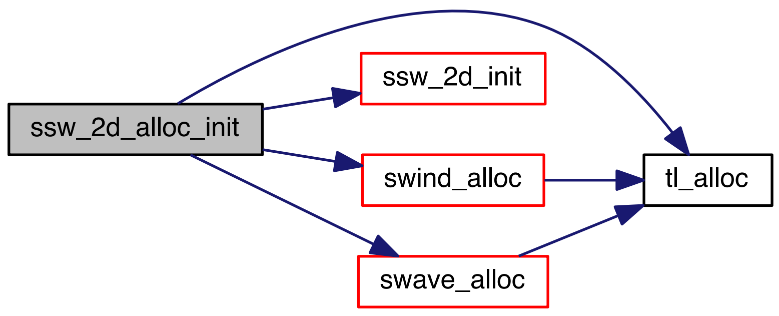


Figure 11. *ssw_2d_alloc_init* routine call graph.

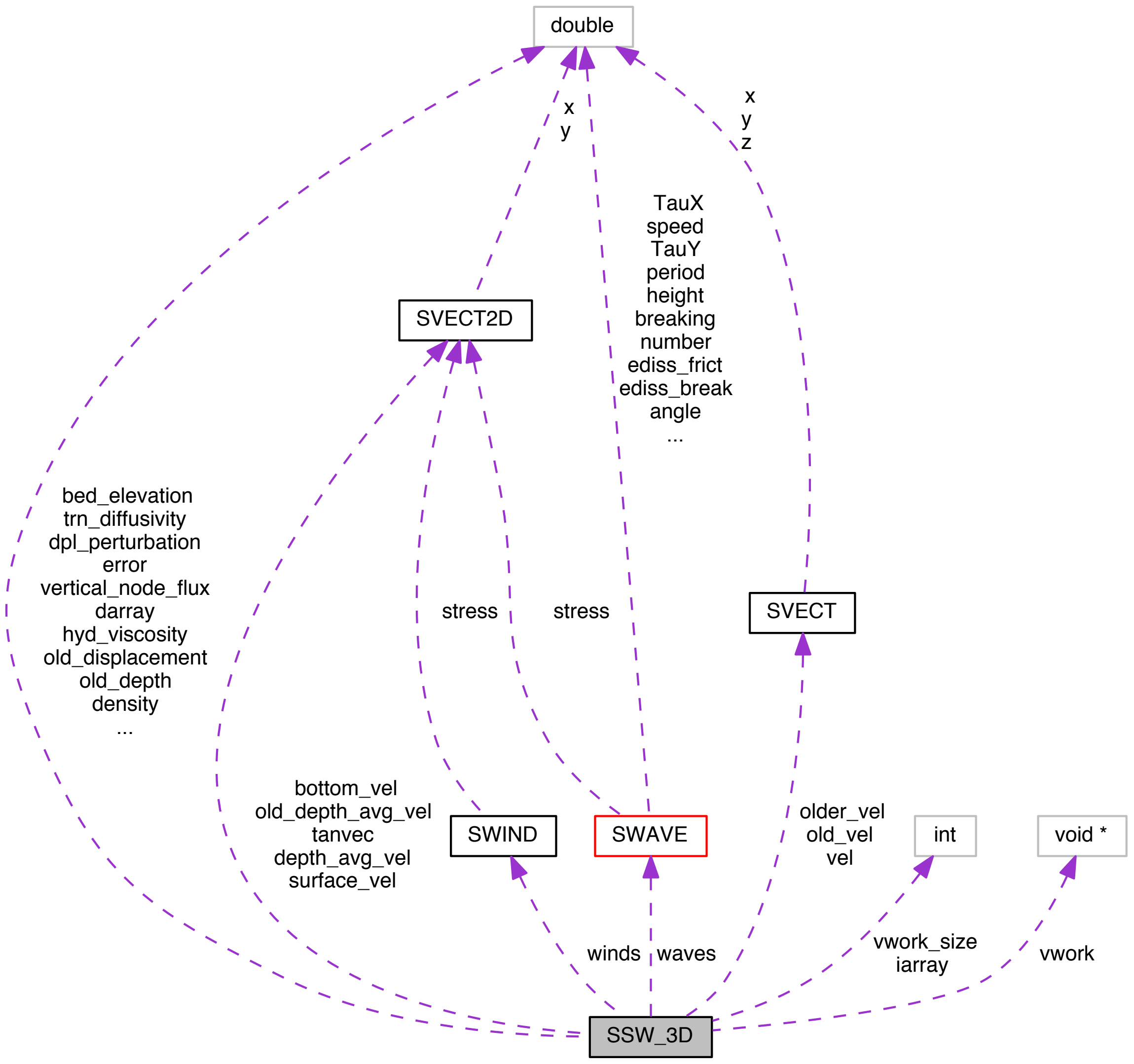


Figure 12. Level 1 *ssw_3d* collaboration graph.

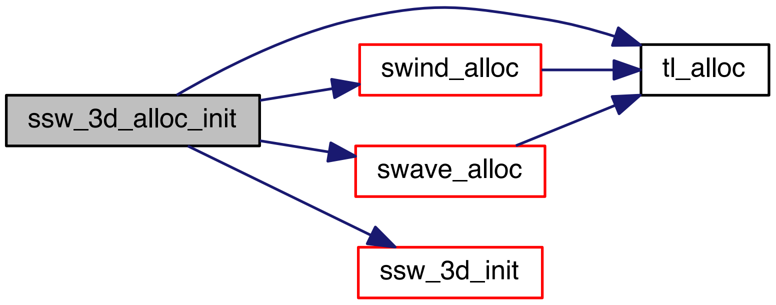


Figure 13. *ssw_3d_alloc_init* routine call graph.

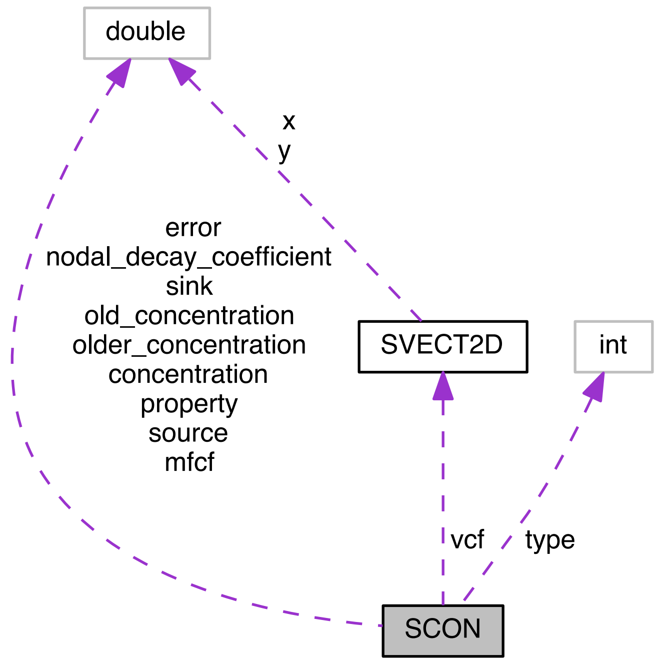


Figure 14. Level 1 *scon* collaboration graph.

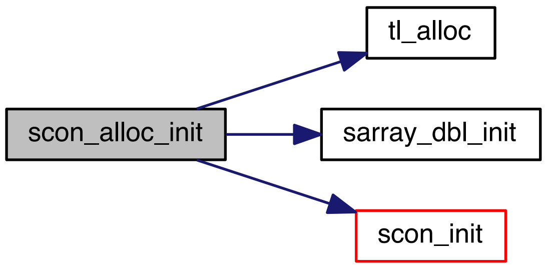


Figure 15. *scon_alloc_init* routine call graph.

5 AdHv5 Quality Assurance

Quality assurance is vital to any production model with continued development. For numerical models, this comes in the form of routine verification and validation. A number of AdH enhancements were made to the new framework to facilitate verification. This chapter details those additions.

Verification

While a host of 2D/3D hydrodynamic and transport verification test cases exist for the legacy framework, these cases were not previously implemented in a consistent nor accurate way. Regarding the latter, for example, no error report mechanism was in place, despite verification test cases having analytic solutions for which errors can be calculated. Instead, model output files for verification test cases were compared between old and new runs, and if a difference existed, a flag was thrown. It was then left up to the user to visually inspect the solutions to make sure the difference was a positive one. Not only is this a qualitative, time-consuming process, but it also intractable for small differences. Additionally, a change to the model test case input means having to again visualize the results to ensure the new solutions are more accurate.

A large effort was made in the new framework to streamline and solidify verification. Embedded in the new AdHv5 framework are functions which calculate the analytic solutions to the shallow water and transport equations for all verification test cases. These functions are used to calculate and write to a file the typical error-norms for each case. The error files for the last run are included in a verification software repository and any subsequent errors files are compared to the previous. If the analytic errors are equal to the previous, then the source changes pass. If they are better, then the changes pass and the versioned error file is updated with the current one. If the errors are worse, then the results are scrutinized by an expert user.

To invoke analytic error file output, the TEST control card is used. The form of this card is:

```
TEST [specific test case card] [specific test case parameters]
```

where the [specific test case parameters] depend on the test case. For the slosh/seiche test case, for example, the following is used:

TEST SLOSH 737 0.1 40000 8000 12

where the parameters following the slosh card are defined as:

- 737 - node number at which errors will be calculated
- 0.1 - the initial amplitude of the free surface disturbance
- 40000 - the length of the grid
- 8000 - the width of the grid
- 12 - the total initial water depth

For some test cases, like this one, the model is internally initialized depending on the parameters supplied. In this case, the hot-start conditions are over-written and not used. This flexibility allows users to easily change the test case without having to recreate the input files.

Please note that the AdHv5 verification test case repository should only host those cases with analytic solutions. An additional repository will be used for validation. Also, wall-clock times for verification test cases should not exceed 10 minutes when possible, as these cases are meant to be executed nightly, one after another, using CMake/CDash.

As previously mentioned, there are a number of advantages to using CMake within the new AdH framework. One such advantage is the ability to use CDash. CDash is an open source, web-based software testing server. CDash aggregates, analyzes and displays the results of software testing processes submitted from clients located around the world. Developers depend on CDash to convey the state of a software system, and to continually improve its quality. CDash is a part of a larger software process that integrates Kitware's CMake, CTest, and CPack tools, as well as other external packages used to design, manage and maintain large-scale software systems. This powerful integration will ensure quality assurance by automatically calculating AdH verification errors for all test cases on a nightly basis and reporting when a test case fails. Currently, servers within the ERDC firewall are being set-up for this purpose.

In order to add new verification test cases to the AdH internal error calculation suite, the following steps must be taken:

- A new *TEST* card must be created to tell AdH errors are to be calculated and reported. This card is applied in the *bc* file.

- If the test case is to be prepared in any way internal to AdH (internally hot started, etc.), a new *testcase_prep_* routine must be created in the \initio\testcases.c file. To use this new routine, the new line condition must be added to the *testcase_prep* routine at the top of file. Figure [16] displays a call graph of the *testcase_prep* routine.

- Once the test case has been prepared, a new error routine, *write_testcase_error_*, must be created and called from the *write_testcase_error* over-arching routine, whose call graph displayed in Fig. [17].

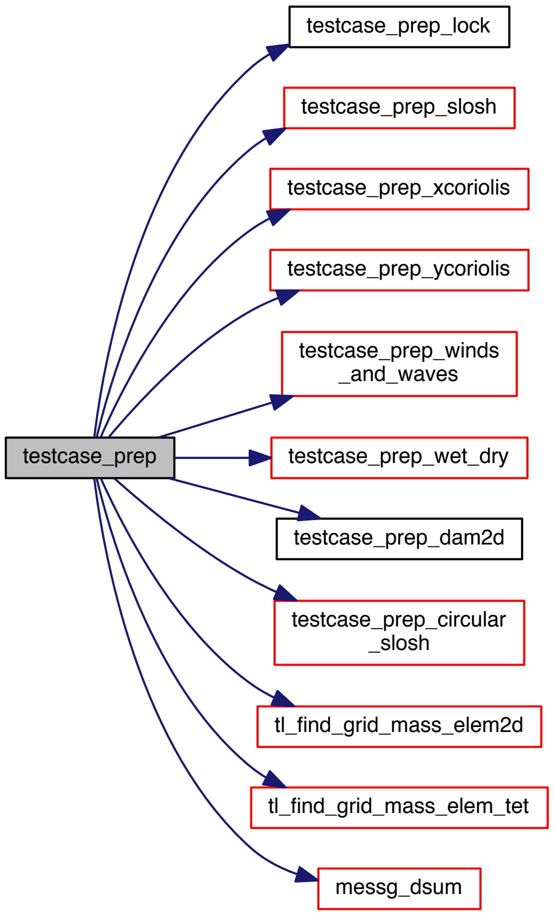


Figure 16. *testcase_prep* routine call graph.

Validation

Most often, analytic solutions are not available, especially for complex simulations that implement a number of forces/features. Also, it is important to know if the physics implemented adequately describes the physical system being simulated, which cannot be assessed through verification alone. Validation is vital for the QA of such problems. Oftentimes, validation runs have large wall-clock times and/or

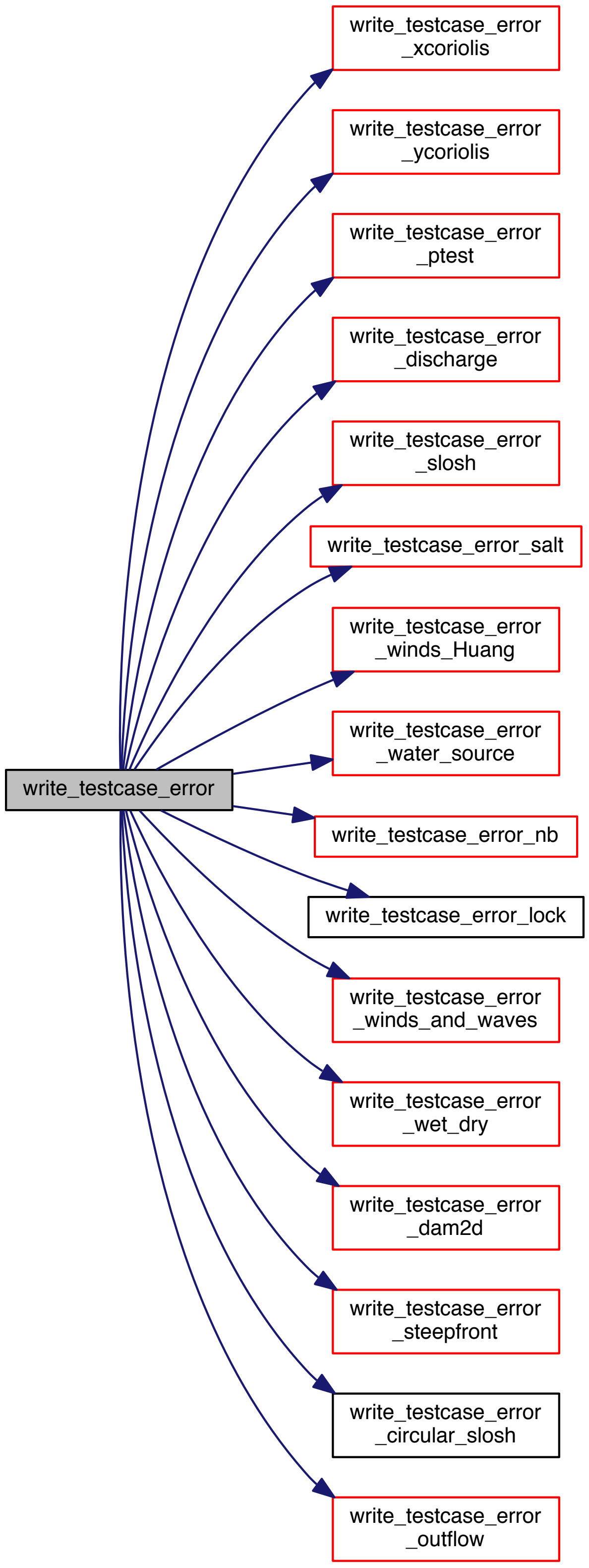

write_testcase_error
write_testcase_error_xcoriolis
write_testcase_error_ycoriolis
write_testcase_error_ptest
write_testcase_error_discharge
write_testcase_error_slosh
write_testcase_error_salt
write_testcase_error_winds_Huang
write_testcase_error_water_source
write_testcase_error_nb
write_testcase_error_lock
write_testcase_error_winds_and_waves
write_testcase_error_wet_dry
write_testcase_error_dam2d
write_testcase_error_steepfront
write_testcase_error_circular_slosh
write_testcase_error_outflow

Figure 17. *testcase_prep* routine call graph.

must be calculated using high performance computers. Solution errors are calculated using field data given at specific locations within the calculation domain. Validation has always been an integral part of QA for AdH, however, previous to AdHv5, it was poorly versioned/controlled. All validation cases for the new framework are now located in a repository dedicated to validation test cases. Because these applications can have large wall-clock times, it is not advised that they are included in the nightly runs, however, they should be re-executed each time significant source changes are made.

6 AdHv5 Enhanced Debugging

Essential to minimizing development and application turn-around time are easy to use internal debugging tools. To this end, the legacy AdH framework was equipped with debug routines that can be enabled using a macro flag during compilation. These routines hijack the heap memory allocations and extend the allocations (“pickets”) for monitoring purposes. It is then left to the developer to add memory checks when debugging. This extremely useful debug feature has not only been ported over to the new framework, but also created into a library for further use in other software, such as the restructure version of SedLib - where it has already been implemented.

In addition to memory monitoring, a whole host of new debugging options are available in AdHv5. First, a new control card was added that will turn-off physics during the simulation. This is implemented using the NOTERM card. The format of this card is as follows:

NOTERM [specific equation term card] [parameters]

Currently, the following term cards are supported (these cards have no parameters):

- DIFF - the diffusion term
- CONV - the convection term
- SUPG - the streamline upwind Petrov-Galerkin stabilization term
- FRIC - the friction term
- TIME - the temporal term
- PRESS - the pressure term
- COR - the coriolis term

Also included is a NOTERM HYDRO option that will remove all hydrodynamic calculations and, instead, use user-supplied and constant water displacements and velocities via the following:

NOTERM HYDRO [u] [v] [w] [dpl]

where u, v, w are the hydrodynamic velocities and dpl are the water surface displacements.

Oftentimes, it is necessary to scrutinize the finite-element matrix contributions from each of these terms. While options to write out these contributions were somewhat available in the SW 3D model of the old framework, they were not available for other models nor robust or controllable from the front-end. In an effort to extend its use, a new DEBUG control card has been implemented in AdHv5 that will write out specific matrix contributions. The format of this new card is as follows:

DEBUG [specific equation term card]

This card will print to screen all elemental contributions from the term as well some parameters used in the elemental calculations. Options for the terms are as follows:

- READBC - writes out boundary condition file input
- RHS - writes SW 2D right hand side contributions from all terms
- RHSHV - writes SW 3D hvel right hand side contributions from all terms
- RHSWV - writes SW 3D wvel right hand side contributions from all terms
- MAT - writes the SW 2D matrix
- MATHV - writes the SW 3D hvel matrix
- MATWV - writes the SW 3D wvel matrix
- NEWTON - writes out information during the nonlinear newton solve

7 AdHv5 High-Performance Computer Efficiency and Scalability

A primary motivation for restructuring the AdH framework was to improve the software's HPC performance. In previous versions of AdH, for example, all processors allocated memory for all variables on the full computational grid before domain decomposition occurs, maxing out the computation node memory before the simulation had even begun. This was prohibitive for large problems, such as 3D estuaries or sediment transport models with large numbers of constituents, unless more cores were reserved for memory alone. Additionally, all communication related variables were again globally scoped in the legacy AdH, preventing multi-model/grid simulations. The parallelization scheme of the new framework was designed to not only mitigate these two issues but also with an eye towards future HPC development.

The "*smpi*" struct

As previously mentioned, each grid created within AdHv5 has its own set of HPC MPI communication variables. The new framework stores all these variables in a "*smpi*" structure, which is a member of the *sgrid* structure (see Fig. [7]). By using this containment strategy, AdHv5 has the ability to use multiple grids in one or more *smodel* structures. Each of these grids is capable of running on multiple processors with its own set of private communication variables. Since all of the communication subroutines have been written with this strategy in mind, a developer working on multi-grid systems only needs to keep track of their *sgrid* structures. All of the appropriate MPI functions will naturally cascade down from the *sgrid* to *smpi* structs and routines. By default, all of the *smpi* struct members are initialized to NULL, except for the number of processors and the processor rank, which are set to values returned from MPI_Comm_Size and MPI_Comm_Rank respectively. If AdHv5 is built without MPI support, the *smpi* struct only contains the number of processors (1) and processor ranks (0) as members. These are included in non-HPC builds because the debug routines in AdHv5 were enhanced to report processor ID, requiring this structure. A level 1 collaboration graph of the *smpi* struct is given in Fig. [18].

Building AdH with MPI parallelization

It should be noted that the previous parallelization of AdH was done at a time when message passing protocols other than MPI were being developed by the HPC community. An attempt was made to separate AdH communication functions from the protocol being used so that future methods might be adopted. This led to two

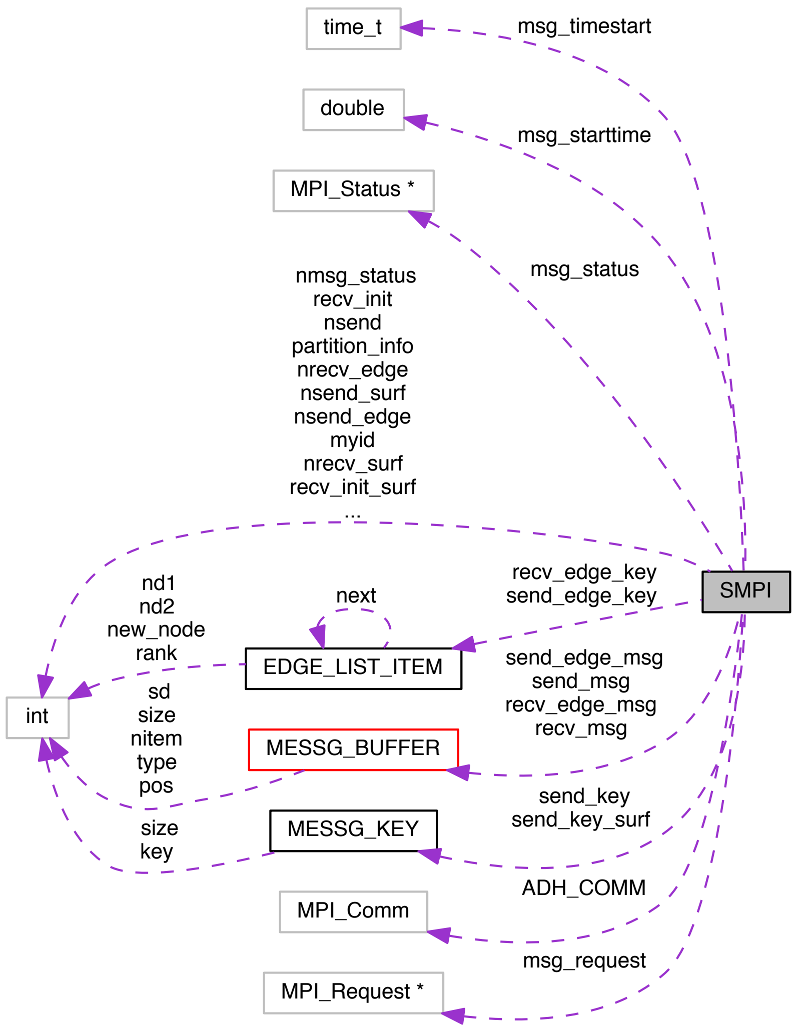


Figure 18. Level 1 *smpi* collaboration graph.

different preprocessor macros, “D_MESSG” and ‘D_MPI”. Through the years, however, these macros were erroneously used interchangeably, and their original intentions confused. Since MPI has become the dominant communication protocol on HPC platforms, AdHv5 has simplified the communication routines to use one “D_MESSG” preprocessor macro. To build AdH with parallel capabilities, the “USE_MPI” flag to “ON” in CMake. CMake will then detect the appropriate system MPI libraries and compilers as well as add the “D_MESSG” macro to the Makefile. The default settings will link to a AdH super library that builds AdH with the ParMETIS grid partitioning routines. This can disabled by turning off the “USE_SUPER_LIBRARY” option within CMake. For more information on ParMETIS, please see http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

Communicating AdH variables with MPI

Previous versions of AdH made the MPI communication calls appear simple to a non-MPI expert. This functionality was retained in version 5. To update a variable with information from other processors, a call is made to

comm_update_XXX($*variable$, $*smpi$)

where XXX is the type of variable to be updated (integer, double, etc.), $*variable$ is a pointer to the communicated variable and $*smpi$ is the *smpi* struct containing a specific grid's MPI information. For example, to update velocities in a SW 3D simulation, comm_update_VECT(vel, smpi) would be used. This is exactly the same paradigm used in the legacy AdH framework. Developers of AdH need only know when ghost node information is needed from other processors.

The MPI routines themselves are currently structured in a very similar manner to previous versions of AdH. An advantage of AdHv5's modularity is that the MPI routines can be profiled and modifications tested without disturbing the rest of the AdH software. This will facilitate future additions/enhancements to the HPC protocols within the software. One important change in AdHv5 is that processors only store in memory data from their subdomain, as opposed to all processors allocating memory for the entire computational mesh as is done in the legacy software. This greatly reduces the memory overhead and eliminates the need to allocate extra HPC resources for memory. To do this, it was necessary to change the way AdHv5 reads the initial grid file. Upon execution in parallel, each AdH processor first scans the 3dm file to determine the mesh dimension, the number of elements and the number of nodes. If the mesh is 3D, then the number of surface nodes is also determined. Once this initial scan is done, the mesh is then partitioned by splitting the number of nodes in 2D, or the number of surface nodes in 3D, evenly among the processors. The 3dm file is then re-read and the nodal information (including ghost nodes), elements and connectivity are stored in an *sgrid* struct. Importantly, memory allocation of *sgrid* is only as large as required to contain the subdomain data for a given processor. If AdHv5 is built with ParMETIS (default for MPI builds), then after all processors have read their subdomain, the mesh undergoes a re-partitioning using the ParMETIS library. This re-partitioning seeks to reorganize the nodes so that communication between processors is minimized.

Overall, the mesh input, partitioning and communication of variables was scrutinized and optimized in the restructuring of AdH into version 5. A great deal of unnecessary legacy communication calls and barriers were removed to improve scalability. Memory usage was made as efficient as possible to eliminate the need for extra HPC memory resources. The only area which was not significantly altered was output. Currently AdHv5 outputs variables exactly as previous versions, where all written data is communicated in lock-step to the root processor. Once all the information for a given array is received, the root processor writes it to an ASCII file. This method is sufficient for now and allows users to take advantage of legacy post-processing tools such as SMS. Eventually, problem sizes will grow to the point that this method of output will effect scalability. When this issue arrises, the modularity of AdHv5 will allow new, scalable I/O strategies to be incorporated rather easily.

Scalability and memory utilization of AdHv5

All additions/enhancements made within AdHv5 have been designed to optimize the HPC memory management and interprocessor communications for large scale applications. Strong scaling results and efficiencies for both the old and new AdH are given in Figures [19]-[20] and Figures [21]-[22] for the SW 2D Anchorage and SW 3D Galveston Bay applications, respectively. In these figures, the 2D Anchorage grid had a total of 70861 nodes, 138132 elements and the simulation length was about 9 hours. The Galveston Bay grid had a total of 64328 nodes, 269543 elements and the simulation length was about 125 hours.

Figures [19]-[20] show the 2D application scalability and efficiencies of the two codes being similar and as expected. These figures also show a slight speed-up in wall-clock time of AdHv5 over AdHv4.5 for the 2D run. Though algorithm efficiency was not optimized at the time of this report, this shows promise for the AdHv5. Figures [21]-[22] again show similar scaling and efficiencies between the two versions. The AdHv5 3D SW application, however, had generally better HPC efficiencies, though slightly slower wall-clock timings. The efficiency increase is likely a result of carefully removing unnecessary MPI collective calls in AdHv5. The longer simulation times are expected in AdHv5 due to the collection of redundant code into a large number of small functions. AdHv5 will soon be optimized for speed by identifying bottlenecks via HPC profiling.

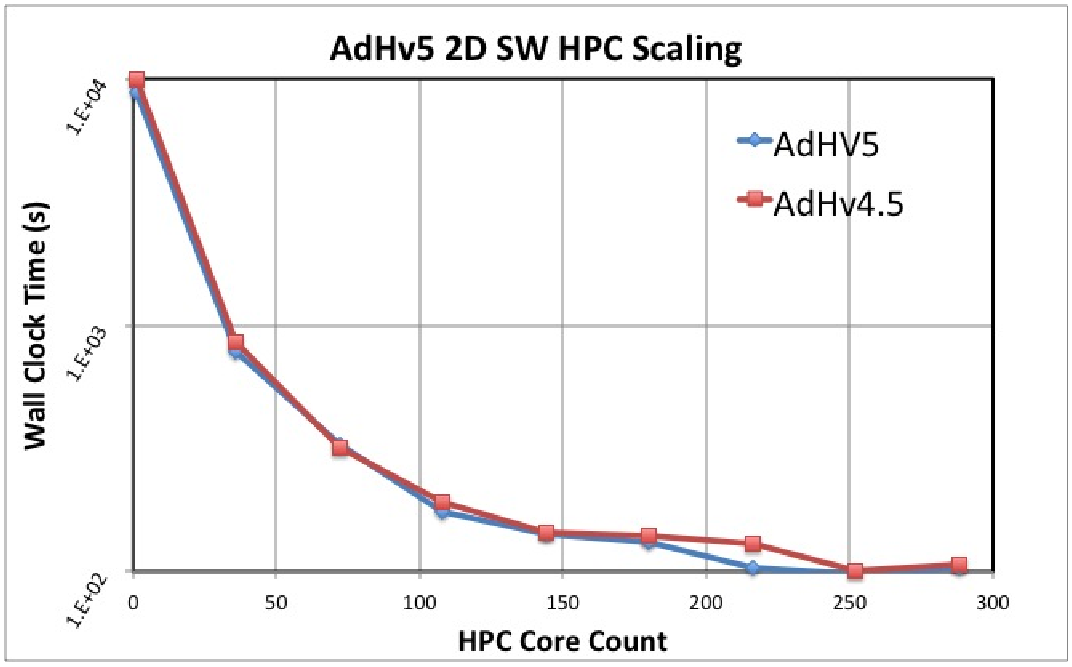


Figure 19. Anchorage HPC strong-scaling results for the 2D AdHv5 and AdHv4.5 shallow water models.

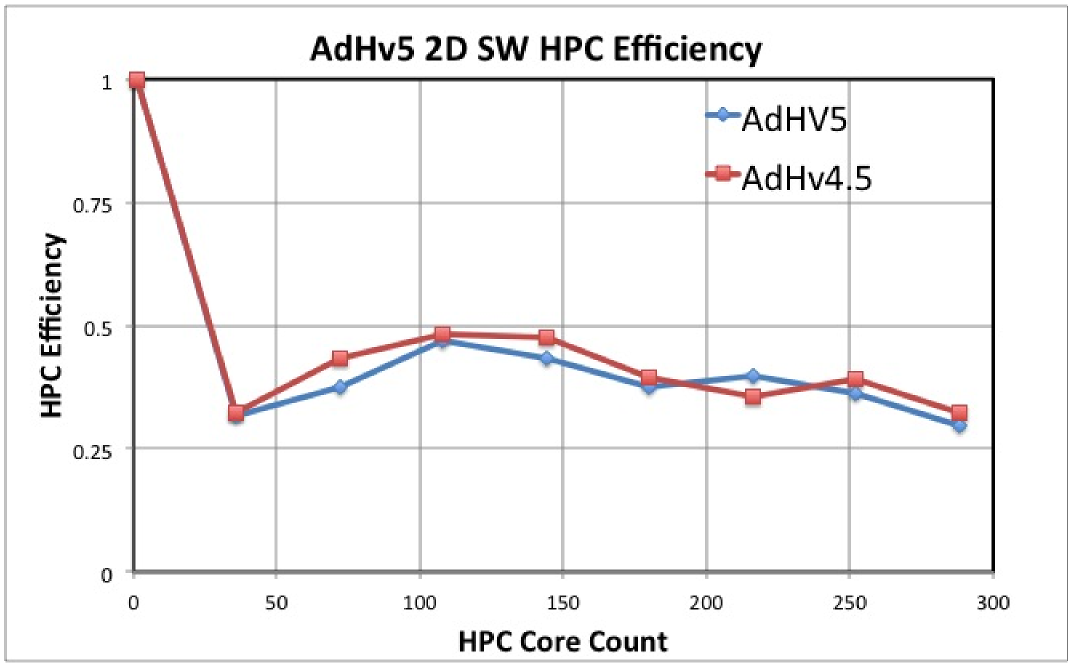


Figure 20. Anchorage HPC strong efficiency for the 2D AdHv5 and AdHv4.5 shallow water models.

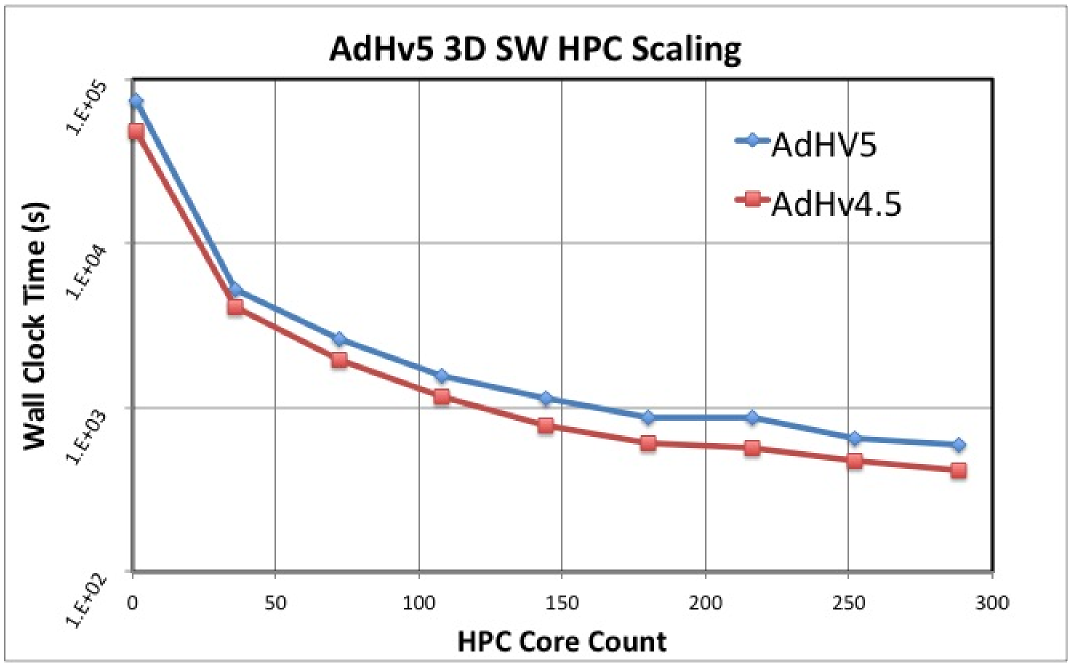


Figure 21. Galveston Bay HPC strong-scaling results for the 3D AdHv5 and AdHv4.5 shallow water models.

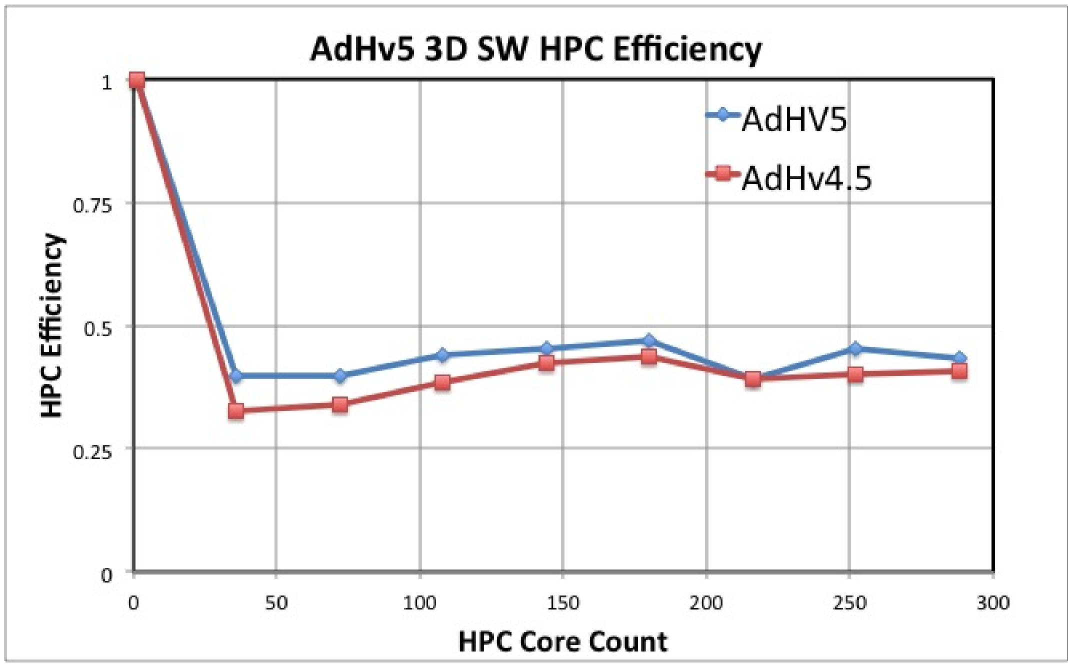


Figure 22. Galveston Bay HPC strong efficiency for the 3D AdHv5 and AdHv4.5 shallow water models.

As mentioned, one of the primary motivations for the new framework was better memory management. Previous versions of AdH required that users allocate as much as twice the HPC core count strictly for memory purposes for large applications. The new framework in AdHv5 was designed to rectify this HPC overuse. Figures (23) and (24) display the max core memory usage for the SW 2D Anchorage and SW 3D Galveston Bay simulations. This figure shows the 2D Anchorage AdHv5 application using less than half the total core memory than AdHv4.5 for multi-core runs. This difference is more pronounced in the 3D Galveston Bay simulation, as the figure shows AdHv5 requires nearly one-third the memory per core of AdHv4.5.

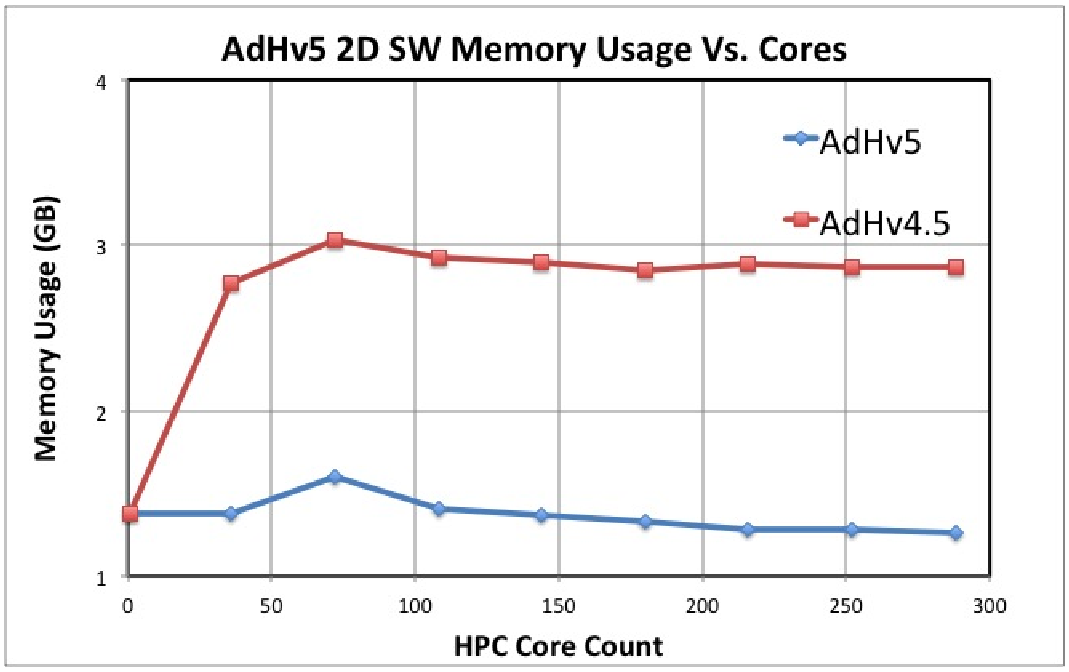


Figure 23. Anchorage HPC memory usage for the 2D AdHv5 and AdHv4.5 shallow water models.

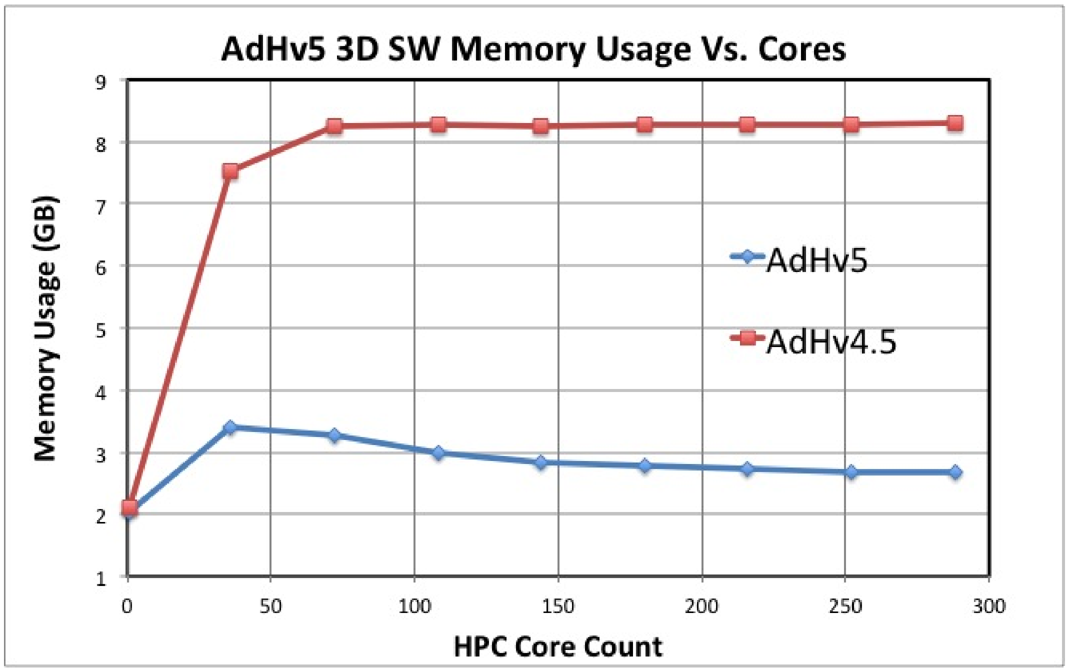


Figure 24. Galveston Bay HPC memory usage for the 3D AdHv5 and AdHv4.5 shallow water models.

8 Front-end Enhancements

To remain competitive in the hydrodynamic and transport modeling market, it is vital that the AdH software suite be easy to use and contain a wide array of output options for engineering analytics. This chapter details all AdHv5 front-end enhancements and additions to the legacy software.

Removal of pre_adh

To execute versions of AdH prior to version 5, it was necessary to first create a binary input file by running "pre_adh". The AdH executable would then read this binary to initialize the model. The pre_adh binary was meant to reduce the AdH total simulation time by reading the binary instead of the ASCII input files each time the application is ran. This step has been abandoned in the new framework for a couple of reasons. First, and most importantly, for large problems - particularly in 3D - the pre_adh binary creation was prohibitive, as this is a serial execution and the whole grid, for example, needed to be stored in memory. A benefit of the new AdH framework is that it never reads/stores the entire grid when running multi-core applications, greatly reducing the amount of memory needed for HPC applications. Because each core only reads its subdomain data, the time required to read the input files is minimal. Another idealized advantage of the the pre_adh binary was that it could be moved between machines instead of all the input files, however, this is almost never done, since it is often necessary to see that input data to make changes.

One real advantage of the pre_adh binary, however, is input file integrity checks. Having recognized the utility of this, a new executable flag has been introduced in the AdHv5 framework which stops/exits the executable after model initialization has taken place (see level 1 of Fig. [1]). This can be implemented by the following:

$$./adh[project]-s \tag{1}$$

where the "-s" flags for AdH to stop after file-integrity checking. Note that file integrity assertion happens every time AdH is ran, and that this binary flag only stops the application after this is done so as not to waste calculation time/cores due to incorrect user input.

New card defaults

The following cards are no longer needed as they will default to standard choices based on physics implemented: OP PRE (1), OP BLK (1), OP INC (40) and OP TRN (0). Of course, the user is still allowed to utilize these cards to change the defaults.

Replacement of the XYSERIES control card

In AdH, time-dependent data applied at or along a boundary string is given as a time series. The series may be used to define how the flow changes with time, the change in the boundary depth over time, the change in the time-step as the model runs, wind and wave surface stresses and even the time at which data is output for analytics. Previously, XY1/XY2 cards were used, but for some cases, these cards required additional cards and cross-referencing (such as requiring the TC DT or XYC cards for time-stepping or wind boundary conditions). The SERIES control card streamlines this without the need for additional cards by allowing for different arguments depending on which sub-card is used. All time series cards, regardless of type, follow a similar syntax. They begin with SERIES, followed by the type of series, the series number, the number of points in the time series and the unit flag for the time values in the series. The unit specifications are as follows: 0 = seconds, 1 = minutes, 2 = hours, 3 = days, and 4 = weeks. These unit specifications only refer to the time column, and they do not impact the values in the additional data columns, which always reference time in seconds. Below the SERIES line are the time and data points making up the series. The series must be sequential in time, and the data values will be linearly interpolated for times falling between two specified values. The following are time-series options:

- SERIES BC :: supplies AdH with time-dependent boundary conditions
- SERIES DT :: supplies AdH with a time-dependent time-step
- SERIES OUTPUT :: supplies AdH with manual time-dependent file output
- SERIES AWRITE :: supplies AdH with automatic time-dependent file output
- SERIES WINDS :: supplies AdH with time-dependent wind stresses
- SERIES WAVES :: supplies AdH with time-dependent wave stresses

The SEDLIB control card

As mentioned, former versions of AdH offered little modularity with respect to physics and executable builds. This was particularly true with the sediment transport library, SedLib. SedLib is a sediment sourcing library that can handle multiple bed layers and multiple grain classes. The library provides the AdH mother transport model with vertical sources/sinks by splitting off the vertical from horizontal terms of the equation. As the name implies, it is meant to be a portable library that can be used with any hydrodynamic/transport software. However, it's actual portability became restricted to AdH through the years as variable and function pollution grew. Maintaining portability to other hydraulic codes opens the market for the library and invites users/developers to use the software. For these reasons, substantial changes to the AdHv5 framework and SedLib were made.

The first, and probably most important SedLib change was made to the AdH front-end. Previously, all SedLib boundary conditions were read into AdH through the AdH bc file, located in the AdH bc read file contained within the AdH source code. The new framework keeps all SedLib input/output completely separate from that of AdH. In AdHv5, all sediment applications must include a new [project_name].sedlib file which stores all sediment boundary condition information. This data is read outside of the typical AdH boundary condition and is disabled if SedLib is not included in the CMake macro list upon compilation. By doing this, all SedLib input, both algorithmically and file I/O are completely independent. Importantly, in prior AdH versions, any sediment constituents were counted as part of the OP TRN total; however, this total now only counts non-sediment transport constituents.

In an effort towards greater source modularity, the new framework does not include a global scope for SedLib variables as the previous AdH did. Instead, all sediment variables are stored in two structs, one needed by the hydraulic suite (in this case AdH) and the other used within SedLib library itself. The former can be found in the structs/ssediment.c file. This file contains one over-arching structure, *ssed*, which contains pointers to structures for suspended and bed loads, grain classes, etc. A level 1 collaboration diagram of this structure is given in Fig. [25] and a hypothetical schematic can be seen in Fig. [26]. The SedLib struct is very similar to the AdH storage structure.

Outside of the AdH framework, SedLib itself was also updated to be more consistent with the new framework. First, AdH-style debug routines were added to the new version of SedLib for easy memory trapping and segfault locating. Additionally, all globally-scoped variables were moved into a *ssedlib* structure, which can be found in the /SedLib-core folder within the SedLib source.

The 3D faces File

In former versions of AdH, no clear distinction between geometry and physics was made, particularly in 3D. For example, in order for the *sgrid* struct to know a 3D element face was a boundary, it had to read the boundary condition file. This inter-dependency cross-polluted structures and routines and generally added to the non-modularity of the legacy code. More importantly, it prevented the grid structure and related functions from being used as a static library for linking with other utilities or software. Creating these static libraries for pre/post processing utilities (such as the 2D/3D extrusion code, for example), allows these utilities to stay current as AdH changes are made. This is opposed to continually and manually updating these utilities when AdH data-types or methods of the data-types are updated. To avoid this inter-dependency of geometry and physics, two front-end changes we're made; (1)

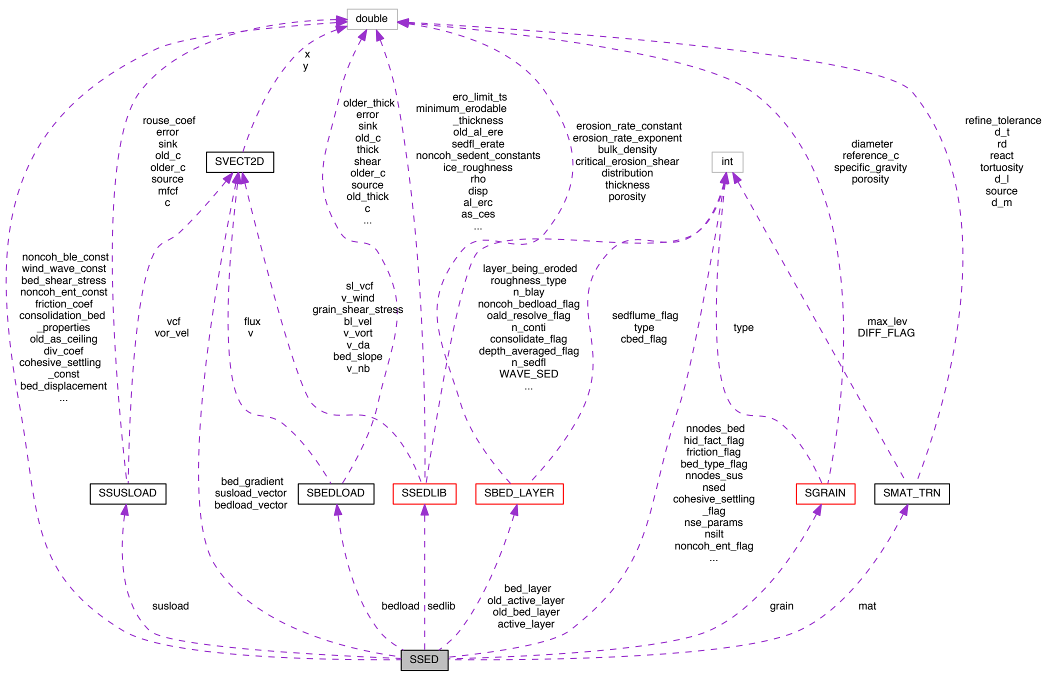

double
x
y
rouse_coef error sink old_c older_c source mfcf c
SVECT2D
older_thick error sink old_c thick shear older_c source old_thick c ...
ero_limit_ts minimum_erodable _thickness old_al_ere sedfl_erate noncoh_sedent_constants ice_roughness rho disp al_erc as_ces ...
erosion_rate_constant erosion_rate_exponent bulk_density critical_erosion_shear distribution thickness porosity
int
diameter reference_c specific_gravity porosity
refine_tolerance d_t rd react tortuosity d_l source d_m
noncoh_ble_const wind_wave_const bed_shear_stress noncoh_ent_const friction_coef consolidation_bed _properties old_as_ceiling div_coef cohesive_settling _const bed_displacement ...
vcf vor_vel
flux v
sl_vcf v_wind grain_shear_stress bl_vel v_vort v_da bed_slope v_nb
layer_being_eroded roughness_type n_blay noncoh_bedload_flag oald_resolve_flag n_conti consolidate_flag depth_averaged_flag n_sedfl WAVE_SED ...
sedflume_flag type cbed_flag
type
max_lev DIFF_FLAG
nnodes_bed hid_fact_flag friction_flag bed_type_flag nnodes_sus nsed cohesive_settling _flag nse_params nsilt noncoh_ent_flag ...
SSUSLOAD
bed_gradient susload_vector bedload_vector
SBEDLOAD
SSEDLIB
SBED_LAYER
SGRAIN
SMAT_TRN
susload
bedload
sedlib
bed_layer old_active_layer old_bed_layer active_layer
grain
mat
SSED

Figure 25. Level 1 *ssed* struct collaboration graph.

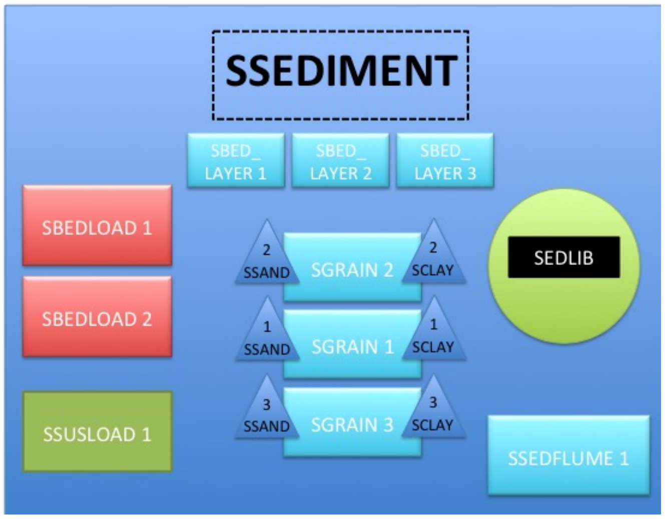


Figure 26. Diagram of one possible allocation of the AdHv5 *ssediment* structure. This scenario shows 3 grains, 2 bedloads, 1 suspended load and sedflume and three bed layers. Note that every allocation to the *ssediment* struct allocates one SSedLib struct for linking to the SedLib libarary.

the 3D element face listings were moved from the boundary condition file to their own [project_name].faces file, and (2) a new integer was added to the face listing that indicates where the face is on the boundary. Options for this parameter are as follows:

- flag=0 :: surface face
- flag=1 :: bed face
- flag=2 :: sidewall face

By including this new flag in the face listing, only the .3dm and .faces are required for all grid information. Moving the faces to their own file also makes the boundary condition file much more straightforward, as the list of faces can be lengthy and cumbersome. Note that this is not required for edges in a 2D application, mostly because these boundaries can be defined internal to AdH as default no-flow when no conditions are supplied by the bc file. The precise format of the .faces file is a listing of 3D element boundary face strings using the following control card:

FCS [element id] [local face id] [boundary flag] [string id]

In AdH, both 2D and 3D grid files are named [project_name].3dm. Consideration was given towards changing the 2D input file to a 2dm extension, however, it was not pursued due to the large number of 2D SW users who require backwards compatibility. The 3dm file format for both 2D and 3D applications remains the same. Currently, the new framework supports tetrahedral elements only in 3D. These element are given with the “E4T” control card. However, current efforts are underway to extend the AdH framework to support triangular prisms, which will add a ”E6P” card to this 3dm file.

Enhanced meteorological input

Previous to AdHv5, the only way to apply time-dependent wind stresses and atmospheric pressures in the shallow water models was through a list of station locations within the boundary condition file. Because AdH was initially designed as a river and estuary model, this was sufficient. However, as the domain/scale of AdH applications grow, so does the need for more spatially varying winds over larger grids. It was recognized that AdH needed the ability to read in a separate, albeit coarse wind grid similar to what is done in the ADvanced CIRCulation model, ADCIRC. ADCIRC not only allows for efficient interpolation between grids, but it also supports a host of wind and atmospheric formats such as the US Navy Fleet Numeric format, Planetary Boundary Layer Hurricane Model format, etc. The meteorological library, MetLib, was created by leveraging the pertinent ADCIRC routines to create a portable, static FORTRAN library that may be linked to any hydraulic model. The library was created with the assistance of the DoD user Productivity Enhancement, Technology Transfer, and Training (PETTT) program in collaboration between the ERDC Information Technology and Coastal and Hydraulics Laboratories and the University of Texas in Austin. It is meant to be a centralized, focal point for all future meteorological development.

Wind library source and build

All source files related to the ADCIRC meteorological library are located within the AdHv5 metlib/ directory. To link this source to the executable when building, the USE_MET_LIB macro must be turned on inside CMake. If this is not done, the AdH executable will only support station-dependent winds as before. All MetLib source files are FORTRAN, so a suitable FORTRAN compiler is required to link to AdHv5. GNU FORTRAN is sufficient and is available for free distribution here: https://gcc.gnu.org/fortran

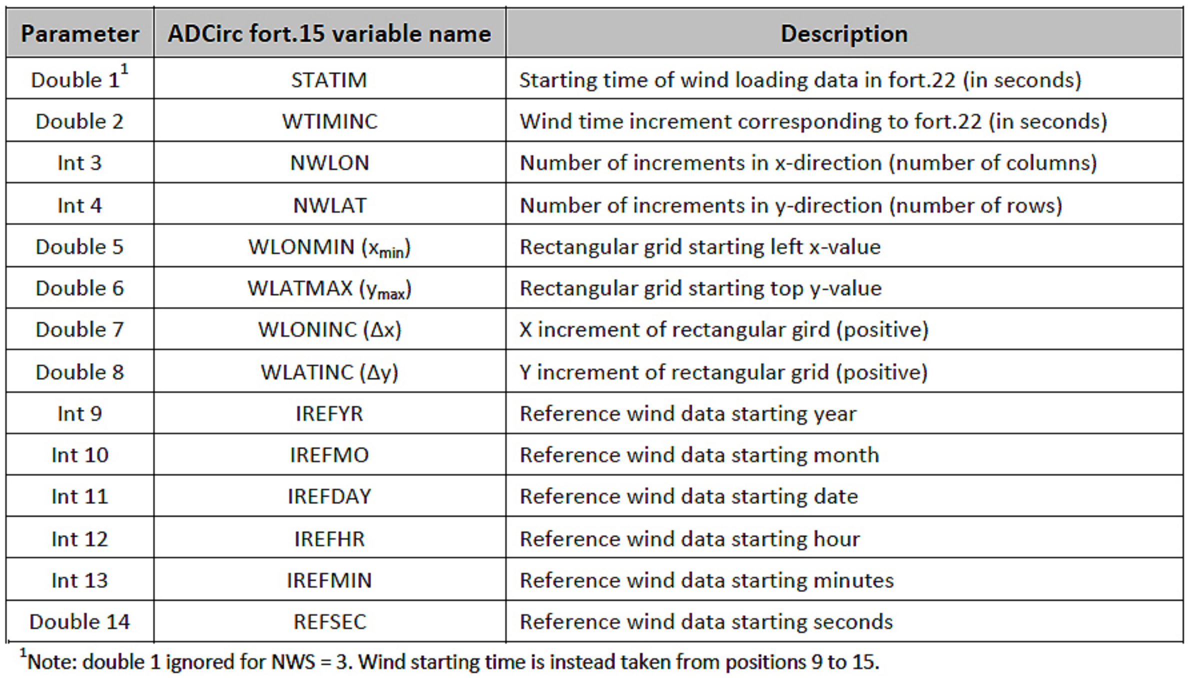
MetLib usage and options

The user indicates the use of the wind library by providing the operation card "OP METLIB" in the boundary condition input file. Additionally, the "OP WND" card must also be added to use wind library. Depending on the chosen NWS parameter, the format of the METLIB operation card must be chosen as per the following:

- OP WNDLIB 1
- OP WNDLIB 2 [double 1] [double 2]
- OP WNDLIB 3 [double 1] [double 2] [int 3] [int 4] [double 5] [double 6] [double 7] [double 8] [int 9] [int 10] [int 11] [int 12] [int 13] [double 14]
- OP WNDLIB 4 [double 1] [double 2]
- OP WNDLIB 5 [double 1] [double 2]
- OP WNDLIB 6 [double 1] [double 2] [int 3] [int 4] [double 5] [double 6] [double 7] [double 8]
- OP WNDLIB 7 [double 1] [double 2] [int 3] [int 4] [double 5] [double 6] [double 7] [double 8]
- The parameters in [ ] are defined in Table 27.

Depending on the NWS parameter chosen, the user must also include the appropriate fort.22 file to the same directory as root AdHv5 input files (.3dm, .bc, .hot). This file contains the spatial and temporal information of the wind data. A description of the format of fort.22 is available online here:

http://adcirc.org/home/documentation/users-manual-v51/input-file-descriptions/single-file-meteorological-forcing-input-fort-22



Parameter	ADCirc fort.15 variable name	Description
Double 1[1]	STATIM	Starting time of wind loading data in fort.22 (in seconds)
Double 2	WTIMINC	Wind time increment corresponding to fort.22 (in seconds)
Int 3	NWLON	Number of increments in x-direction (number of columns)
Int 4	NWLAT	Number of increments in y-direction (number of rows)
Double 5	WLONMIN (x_{min})	Rectangular grid starting left x-value
Double 6	WLATMAX (y_{max})	Rectangular grid starting top y-value
Double 7	WLONINC (Δx)	X increment of rectangular gird (positive)
Double 8	WLATINC (Δy)	Y increment of rectangular grid (positive)
Int 9	IREFYR	Reference wind data starting year
Int 10	IREFMO	Reference wind data starting month
Int 11	IREFDAY	Reference wind data starting date
Int 12	IREFHR	Reference wind data starting hour
Int 13	IREFMIN	Reference wind data starting minutes
Double 14	REFSEC	Reference wind data starting seconds

[1]Note: double 1 ignored for NWS = 3. Wind starting time is instead taken from positions 9 to 15.

Figure 27. OP WNDLIB card input description.

Though MetLib has the capability for various NWS parameters, AdH currently supports only NWS parameters 1 to 7. The following sub-sections briefly describe each of these options and any applications limitations they may currently have.

- NWS Type 1 :: wind stresses are read in from file fort.22 at mesh-specific node numbers at AdH-specific times, without either spatial, or temporal interpolation. Hence, for this case, runs with either temporal or spatial adaptivity are not possible. This case currently cannot be run in parallel since the input is mesh specific and requires extensive additions to the AdH code that were beyond the scope of this project.

- NWS Type 2 :: wind stresses are read in at mesh-specific node numbers without spatial interpolation, at fixed wind time intervals. This allows (linear) temporal interpolation of wind stresses. Hence, for this case, runs with temporal adaptivity are possible, but those with mesh adaptivity are not. This case currently cannot be run in parallel since it requires extensive additions to the AdH code that were beyond the scope of this project.

- NWS Type 3 :: requires wind velocities over a rectangular wind grid at variable wind times specified in fort.22 itself. The fort.22, in this case, is the U.S. Navy Fleet numeric wind file. This allows (linear) spatiotemporal interpolation. For this case, all possible run combinations are expected to work. This NWS case can run in parallel as well.

- NWS Type 4 :: wind velocities are read in over mesh-specific nodes, at fixed wind time intervals. The fort.22 in this case is in the PBL/JAG model format. This case is similar to NWS 2, the difference being that the input in this case are wind velocities instead of wind stresses.
- NWS Type 5 :: identical to NWS type 2 except that wind velocities are used instead of wind stresses.
- NWS Type 6 :: wind velocities are read in from fort.22 over a rectangular wind grid, at fixed wind time intervals. (Linear) spatiotemporal interpolation is used to apply the loading from the wind grid onto the mesh. Hence, for this case, all possible run combinations including parallel runs are expected to work.
- NWS Type 7 :: identical to NWS type 6 except that NWS type 7 uses wind stresses in place of wind velocities. This case can used in any combination including parallel runs

Note that because NWS = 1, 2, 4, 5 are mesh-specific wind inputs, at this time adaptive meshing cannot be used. The only possible NWS options that may be used in conjunction with adaptive meshing are NWS = 3, 6, 7, which provide wind loading on a rectangular grid.

MetLib verification

Each of the supported NWS AdHv5 wind case options were verified against analytic solutions to the SW 2D and 3D equations. The array of test cases can be found within the AdH verification test case repository, currently located here:

https://adh.usace.army.mil/svn/adh/adh_restructure/testcases

Ouput Enhancements

A series of output enhancements and additions were made to the AdHv5 software framework to facilitate model analytics. Each of these additions is discussed now.

Screen output

When executing the AdH binary, run-time information is written to screen. Previously, the user had something related to an "all or nothing" option when determining what information was written. The new framework allows the user to choose the output a la carte instead by implementing an "SOUT" card. Minimal information regarding the current time of the simulation, completion percentage and current physics are always displayed. Additional screen output options can be displayed using the following:

- SOUT RESID :: writes both NTL and ITL residuals to screen
- SOUT NLNODE :: write the worse nonlinear residual node offender to screen
- SOUT LNODE :: write the worse linear residual node offender to screen
- SOUT MERROR :: write the total mass error to screen

File output

In an effort for more rapid analytics and subsequently rapid project turn-around time, additional outputs options have been added and can be turned on with the FOUT control card. This include the following:

- BEDVEL :: outputs bed velocity for 3D applications
- SURVEL :: outputs surface velocities for 3D applications
- AVGVEL :: outputs average velocities for 3D applications
- PRS :: outputs the pressure filed for 3D applications
- WIND :: outputs wind fields
- VIS :: outputs the hydrodynamic viscosity calculated by AdH
- DIF :: outputs the transport diffusion calculated by AdH

Additionally, for 3D problems, AdH will output a 2D grid for visualization of 2D output (above) with SMS.

In addition to these new output options for AdH, some SedLib outputs have been changed/re-arranged in a more intuitive manner. AdHv5 SedLib outputs are as follows:

- sediment bed file :: contains all bed properties
- sediment bed load flux file :: contains the bed load flux
- sediment suspended load flux file :: contains the suspended load flux
- sediment active layer file :: contains all active layer properties
- sediment bed layer file :: contains all bed layer properties (one file for each layer)

- sediment suspended grain file :: contains all suspended grain properties (one file for each grain class)
- sediment bed grain file :: contains all bed grain properties (one file for each grain class)

9 Summary

This technical report documents all changes made to a dramatic revision of the AdH software suite, version 5, and establishes AdH coding guidelines for future development. Herein, we have detailed a new underlying framework which provides the suite with the following: (1) greater modularity, (2) a more organized, non-global variable containment strategy, (3) enhanced debugging abilities, (4) a more efficient HPC wrapper, (5) a more elegant user interface and lastly, (6) a greater number of file output options. Each of these enhancements/additions have been documented in detail throughout this report, which serves as a historical reminder of how costly poorly written software frameworks can be and why a discrete jump was made in the version history of AdH.

References

Appendix A: AdHv5 External Library Dependencies

Depending on the application, the AdHv5 suite may rely on some minimal number of external libraries. A listing of these dependencies and when they are needed are as follows:

- **SuiteSparce** :: This library provides AdHv5 with the latest version of the sparse linear solver Umfpack. Though AdH includes the SuperLU software as a general purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations, it is highly recommended that Umfpack be used instead for all applications. Information on this library, including its source and build instructions can be found at:

  http://faculty.cse.tamu.edu/davis/suitesparse.html

- **ParMETIS** :: ParMETIS is an MPI-based parallel library that implements a variety of algorithms for partitioning unstructured graphs, meshes, and for computing fill-reducing orderings of sparse matrices. ParMETIS extends the functionality provided by METIS and includes routines that are especially suited for parallel AMR computations and large scale numerical simulations. This library is required for all AdHv5 HPC applications. Information on this library, including it's source and build instructions can be found at:

  http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview

- **SedLib** :: This ERDC developed sediment transport process library is required by AdHv5 for all sediment applications (note: sediment must be turned "ON" in the CMAKE dialogue box). The AdHv5 compatible version of this library can be checked out from an ERDC repository located at:

  http://adh.usace.army.mil/svn/adh/sedlib/branches/restructure/

The above libraries can each be freely downloaded and built on any machine. Once independently built, they can be linked to AdHv5 by including their local respective paths within the CMAKE dialogue box. If the libraries cannot be found, CMAKE will refuse to move forward with the build.

Since these three libraries are commonly used for ERDC AdH applications, they have been pre-built for many standard ERDC machines, such as the following: Diamond (dmd), Garnet (gar), Lightning (lightning), Macintosh El Capitan (mac), PC with Portand Group compilers (pc_pg), Topaz (topaz), etc. These libraries can be check out independently from an ERDC subversion repository located at:

http://adh.usace.army.mil/svn/adh/adh-libraries/

In order to streamline linking using any of these machines, the libraries have also been consolidated into one static library. To use this, simply set

USE_SUPER_LIBRARY = ON

in the CMAKE dialogue box, and set the appropriate path in the

ADH_SUPER_LIBRARY [path to super library]

box. All "super library" files are located on the above library server with filenames structured as libadh_[machine].a. These super libraries were created by Lucas Pettey, HPCMP PETTT EQM CTA Lead (second author of this report). For more information on the AdHv5 super library, including how to create one for your machine, please contact: *Lucas.R.Pettey@erdc.dren.mil*.