Updating TOPLATS with Fortran 2003

Nathaniel Chaney
Colby Fisher
Amanda Siemann
Ruolan Xu
Wang Zhan

Department of Civil and Environmental Engineering

January 18, 2013

1 Introduction

Land surface hydrologic models are used to simulate the interaction between the atmosphere and land surface. Their main objective is to model the water and energy balances and also to accurately partition the surface fluxes. Their ability to simulate these fluxes from limited data is key for monitoring the hydrologic cycle at continental and global scales. One of the models used at a regional scale is TOPLATS which was developed at Princeton University during the 1990's. Time has shown that this model is an excellent bridge between coarse scale models and very fine resolution models, making it an excellent candidate to replace coarse scale models to study global hydrology. This is also due, in part, to the large increase in computational power over the past two decades.

The model was originally written in Fortran 77 and it falls along the lines of "spaghetti code". It was never developed with the intent of making it easy to switch components of the models. Some of the inherent deficiencies are the lack of proper decoupling between the I/O and the model through an appropriate interface and the fact that the parameterizations of physical processes (evaporation, infiltration, and runoff) were not appropriately modularized, making it practically impossible to quickly and efficiently update the model with new research. With a desire to now update this model and run it over much larger spatial scales than what it was originally intended for, we see this as a perfect opportunity to gravitate towards object-oriented programming in Fortran 2003.

The main goals of this project included: 1) Modularizing the code based on the physics so that it will be easy to swap physics; 2) Improve data structures by introduce derived data types; 3) Add parallel computing using OpenMP; 4) Make tests to ensure subroutines as well as the whole model produce same results as the original model.

The following table concludes the programming tools we have used during development:

2 Interface

(Nate)
Driver
Separate I/O from actual model
Simplified input files

Table 1: Tools used in the project

Purpose	Tools
Language	Fortran 2003
Compiler	gfortran 4.46+
Parallelizaton	OpenMP
Testing	FRUIT
Version control	Git and Github
Memory debugger	Valgrind
Profiler	gprof and gprof2dot
Editor	Vi, Kate, and Gedit
Operating system	Centos and Fedora
Documentation	Doxygen

3 Modularization

One of the main tasks in the project was modularization. To this end, we took the original 100+ files written in F77 and split their functions and subroutines according to their purpose. We set up 11 modules that are combined and used by the interface (TOPLATS_DRIVER.f90) to run the model. Each module is written in a separate file, allowing us to reduce the number of files significantly down to 11. The new TOPLATS model files are shown in Figure 1.

The model consists of three main parts: MODULE_CELL, MODULE_CATCHMENT, and MODULE_REGIONAL. MODULE_CELL uses MODULE_LAND, MODULE_ATMOS, MODULE_CANOPY and MODULE_SNOW. The program's IO was placed in MODULE_IO and variables and data structures are defined in MODULE_VARIABLES. With modularization, swapping physics in the model becomes much easier since we are able to replace the subroutine or module instead of making changes to every related variables and functions throughout the model. It also allows us to take an object-oriented programming approach and choose at runtime what physics we want to use.

Within each module, as shown in Figure 2, we first list shared variables and structures from other modules, defined structures being shared by subroutines in the module if there are any, and then list the subroutines that are related to the module.

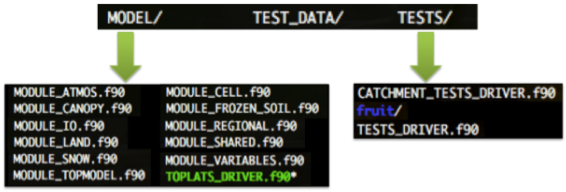


Figure 1: New TOPLATS model

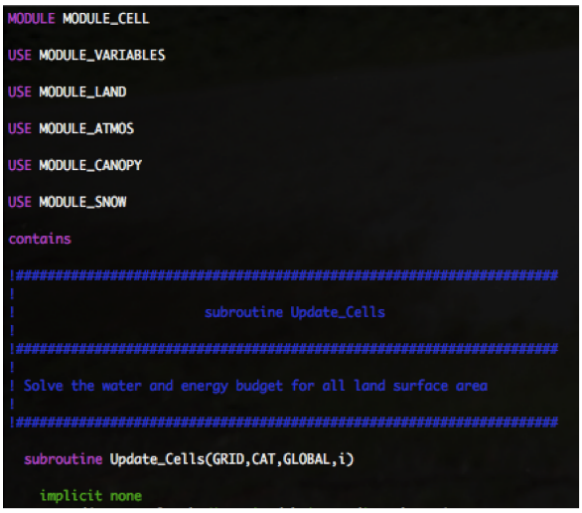


Figure 2: Snapshot of MODULE_CELL.f90

4 Variables

In the original version of TOPLATS, the variables were stored in one lengthy list. Hundreds of variables were passed into subroutines individually, causing extremely large subroutine calls. For the updated version of TOPLATS developed in this project, these hundreds of variables were organized into derived data types (structures) based upon function or purpose. In this way, structures could be passed into the modules and subroutines in relevant groups, thereby cleaning up the otherwise unmanageable code. In each larger subroutine, such as Atmos, the variables used in the code were replaced with the corresponding member of the structure. For each smaller subroutine, the members of the structures were passed into the subroutine individually to avoid passing in more variables than necessary.

Not only was the original version of TOPLATS crippled by the massive subroutine calls, but it was also impaired by the unreasonable number of help files storing the initialized variables with their corresponding types. Each

subroutine was associated with a help file which initialized new variables in each subroutine but which also re-initialized all variables which were passed into the subroutine. This process of re-initializing variables over and over was cumbersome and unnecessary. In the updated version of TOPLATS, the variables in structures are initialized within the structure, and the rest of the variables are only initialized within the larger subroutine in which they are used. In this way, hundreds of files were removed.

Another limitation of the original version of TOPLATS was the use of static memory allocation. Using this method, the size of all arrays needed to be provided at compile time, meaning that the users needed to alter the source files every time they used the model for a different domain. In order to improve upon this, the dynamic memory allocation features implemented in Fortran 90 were used. By dynamically allocating the arrays, the user is able to specify dimensions at run time, allowing for the model to be used across multiple domains without maintaining and compiling multiple versions of the model code.

5 Tests

The FORTRAN Unit Test Framework (FRUIT) is used for testing. We have include two test programs: CATCHMENT_TESTS_DRIVER and TESTS_DRIVER, each serving a different purpose.

The CATCHMENT_TESTS_DRIVER runs the model and compares all the output variables with the original model over a test catchment. Although written in a messy way, the original model's results have been validated so starting from scratch seemed unnecessary. To this end, by using the validated output from a sample input data set, all we need to ensure in the catchment tests is that our our modified model reproduces the output at every time step. These tests were the first thing we wrote, which we ran every time before we made a commit to ensure we did not change any essential physics or algorithms.

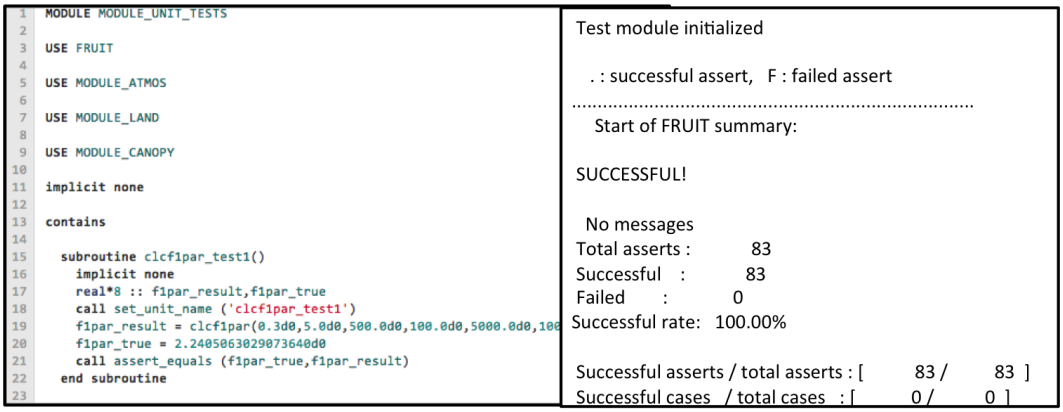
```
41  ! ------------------------------------------------------------------
42  ! Actual energy fluxes.
43  ! ------------------------------------------------------------------
44
45    call set_unit_name (i_str//': '//i_str//': '//'lswb.f90: rnsum')
46    call assert_equals (REG_OLD%rnsum,REG_NEW%rnsum)
47    call set_unit_name (i_str//': '//'lswb.f90: xlesum')
48    call assert_equals (REG_OLD%xlesum,REG_NEW%xlesum)
49    call set_unit_name (i_str//': '//'lswb.f90: hsum')
50    call assert_equals (REG_OLD%hsum,REG_NEW%hsum)
51    call set_unit_name (i_str//': '//'lswb.f90: gsum')
52    call assert_equals (REG_OLD%gsum,REG_NEW%gsum)
53    call set_unit_name (i_str//': '//'lswb.f90: tksum')
54    call assert_equals (REG_OLD%tksum,REG_NEW%tksum)
55    call set_unit_name (i_str//': '//'lswb.f90: tkmidsum')
56    call assert_equals (REG_OLD%tkmidsum,REG_NEW%tkmidsum)
57    call set_unit_name (i_str//': '//'lswb.f90: tkdeepsum')
58    call assert_equals (REG_OLD%tkdeepsum,REG_NEW%tkdeepsum)
59
60  ! ------------------------------------------------------------------
61  ! Canopy water balance.
62  ! ------------------------------------------------------------------
63
64    call set_unit_name (i_str//': '//'lswb.f90: wcip1sum')
65    call assert_equals (REG_OLD%wcip1sum,REG_NEW%wcip1sum)
```

```
SUCCESSFUL!


No messages
Total asserts :       16250
Successful    :       16250
Failed        :       0
Successful rate:  100.00%


Successful asserts / total asserts : [      16250 /
16250  ]


Successful cases   / total cases   : [        0 /        0  ]
```

Figure 3: Catchment tests example code (left) and its corresponding assertions output (right)

One of the major downfalls of testing the model against old output is the inability to then swap physics. To this end, a set of unit tests were devloped in TESTS_DRIVER.90 to run tests on the individual subroutines by using random inpt values. Considering the amount of subroutines and functions and that the model was previously validated, we only tested about 50 of them. Random values were given to the subroutine or function being tested, and the result value was compared with the true value. The true value was previously calculated by the original subroutine or function. As we move to the next step and we implement new subroutines, we need to ensure that the unit tests capture the same errors as the CATCHMENT_TESTS.f90 does. This will allow to eventually completely replace those tests with the subroutine

unit tests.



```
MODULE MODULE_UNIT_TESTS

USE FRUIT

USE MODULE_ATMOS

USE MODULE_LAND

USE MODULE_CANOPY

implicit none

contains

  subroutine clcf1par_test1()
    implicit none
    real*8 :: f1par_result,f1par_true
    call set_unit_name ('clcf1par_test1')
    f1par_result = clcf1par(0.3d0,5.0d0,500.0d0,100.0d0,5000.0d0,100
    f1par_true = 2.2405063029073640d0
    call assert_equals (f1par_true,f1par_result)
  end subroutine
```

```
Test module initialized

  . : successful assert,   F : failed assert
.....................................................................
   Start of FRUIT summary:

SUCCESSFUL!

 No messages
Total asserts :         83
Successful   :         83
Failed       :          0
Successful rate:  100.00%

Successful asserts / total asserts : [        83 /        83  ]
Successful cases   / total cases   : [         0 /         0  ]
```

Figure 4: Subroutines tests example code (left) and its corresponding assertions output (right)

6 Parallelization

Parallelization is currently implemented using OpenMP. The user can define at runtime how many cores to use. In order to do this, for each time step, the domain is split up into equal sized blocks and then is distributed among the threads. Calculations are then performed for each cell before the domain is reassembled and the regional processes are calculated. For the domain size that we are using the parallelization does not have a significant impact on the calculation time. It should be noted though that for future work, as the domain size increases, we believe that we will find that parallelization becomes increasingly important. In order to run the model on a global scale, we plan to implement parallelization through MPI in order to maximize the computational power of each node.

7 Variable Initialization

One of the challenges of the original code was the improper initialization of variables. Many times, uninitialized variables were passed into a subroutine. This was not an issue if the variable was reassigned. But many times, the variables were used as sums. If they were not initialized to zero, there was no assurance that it would be zero. This caused errors in the output.

To deal with this problem, we initially used the -finit-local-zero option available in gfortran. This set all uninitialized values to 0 at runtime and temporarily fixing the problem. However, if we want to port this model to other compilers, we can't rely on that option. Plus it seems like a poor coding practice.

For these reasons, we spent time towards the end with valgrind. This tool provides information on uninitialized variables. It's not that simple as it only provides the line but doesn't say exactly what variable is causing issues but nonetheless it goes a long way in providing reliable information. It took about 6 hours to find around 100 uninitialized variables in the code. After solving this issue, we were successfully able to remove the -finit-local-zero option.

8 Profiling

Another point of interest was determining where the program is spending most of its time when running. To this end, we used gprof to determine the run time statistics. We then used gprof2dot to convert the ascii output into a call graph visualization as shown in Figure 5.

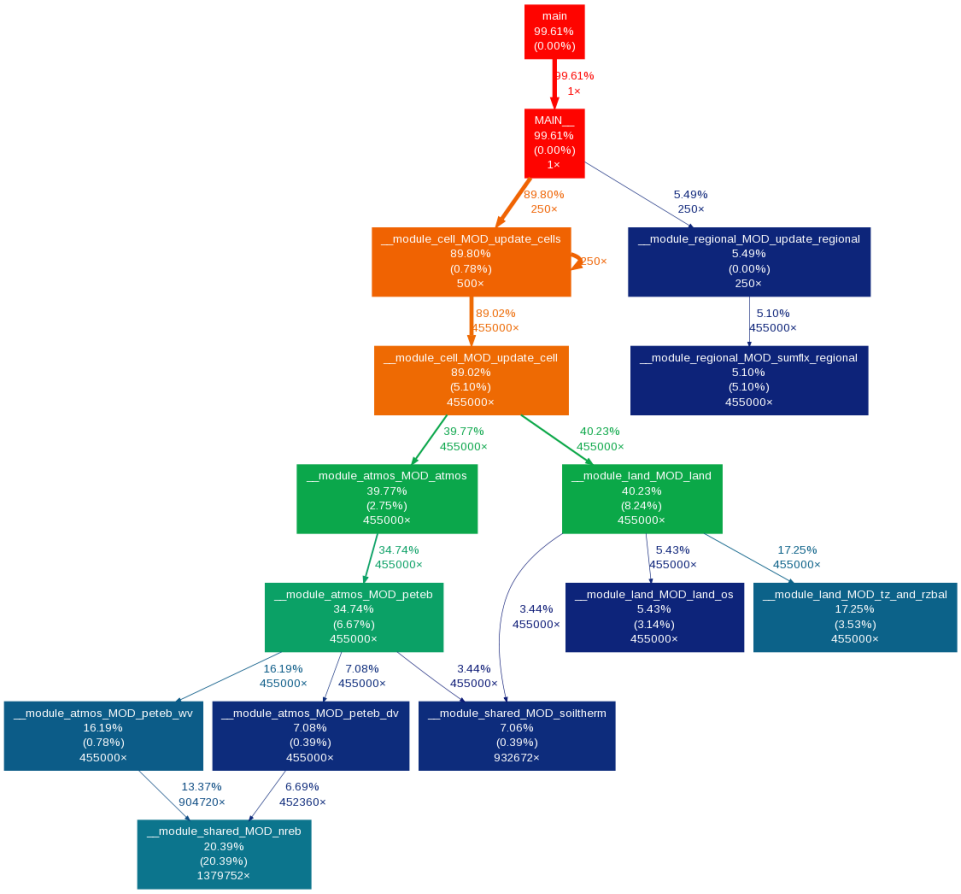


Figure 5: Schematic of the call-graph of TOPLATS. We define a threshold of 5%, below which all subroutines are discarded. This offers a detailed view on where the model is spending most of its time.

As expected, the model spends a very large amount of its time in the Update_Cell module (90%). This is mainly due to the need to ensure energy balance closure. This means that the surface energy balance equation for each grid cell at each time step needs to be minimized. Any future improvement in performance (excluding parallelization) will need to address the nreb subroutine and use more efficient algorithms. That being said, ,we are happy

with the performance of the model. It's current runtime for 250 time steps over a 60X66 domain is around 5 seconds. For reference, this is much faster than most other hydrologic models that thoroughly solve the groundwater interactions between cells.

9 Conclusion

We have accomplished most of our goals. Derived data types were implemented (although not completely); modules were set up to simplify the code and make it simple to swap physics; the I/O components were separated from other routines and enhanced; parallelization using OpenMP was added; and the main program was replaced with a much cleaner interface.

The huge changes, will enable us to use this outdated model for future hydrologic studies. Current work is looking at adding the MPI capability for massive parallelization. Nathaniel Chaney is currently developing a global model run at a 500 meter spatial resolution on the Blue Waters supercomputer in Illinois. This will be groundbreaking work that looks to redefine how the global water balance is modeled.

This project and the course has been beneficial to everyone in the project. We have learned how to effectively collaborate on the same program with great help form github. Testing has become an important part of our programming practices, which was almost absent before taking this course. Learning how to use valgrind has been extremely useful as it also serves as a debugger. In conclusion, all the tools we have used, github, Doxygen, Valgrind , gprof, etc... will continue to serve as our programming companions in the future.