# Improved MESH efficiency via parallelization and code optimization

Chris B. Marsh[1,2], Bruce Davison[3], and Raymond J. Spiteri[2]

[1] *Centre for Hydrology, Department of Geography, University of Saskatchewan*
[2] *Simulation Research Laboratory, Department of Computer Science, University of Saskatchewan*
[3] *National Hydrology Research Centre, Saskatoon, Saskatchewan*

October 2009

## Contents

**Abstract**

Currently, advances in hydrological science are based on field data collection, computer modeling, and analysis of these types of data in order to draw scientific conclusions about physical processes and to perform "what-if" scenarios to aid in scientific prediction. However, as the modeling domains become larger or the model resolution becomes smaller, ever greater increases in computational effort are required. In addition to the limits on computer processing power currently available, other issues, such as round off errors, can present themselves when solving equations numerically. Although current computing power continues to slowly increase, much greater performance gains can be obtained by taking advantage of multi-core, multi-processor computer architectures. Completely utilizing modern computational methods can help to further scientific advancement. Code optimizations are important for future work that incorporates more computationally expensive physics into the model. By decreasing the run time, model users can more quickly iterate over calibration parameters, allowing more time to be spent on the science. In this work, Environment Canada's model Modélisation Environmentale Communautaire (MEC)–Surface and Hydrology (MESH) 1.3, which is based on the Canadian Land Surface Scheme (CLASS), was examined via code profiling to determine the slowest portions of code. Focus was given to determining whether the code could be adapted for parallelism targeting shared-memory processors and whether various code optimizations could be made to the code structure. Initially, the four slowest functions, which comprised about 50% of the run time, were examined. Two, responsible for internal array management, were removed via code restructuring. Further code optimizations were performed via introduction of loop-level parallelism. Results show a 37% speed improvement due to code restructuring and a 47% speed improvement when run with two threads. These results demonstrate sufficient improvement to suggest that parallelism and further code optimization would be a fruitful approach to apply to other portions of the model. However, performance testing showed that the code is still dominated by the serial sections. Through regression testing it was observed that difference compilers produced different results with the same input data. These instabilities were quantified via a relative error which indicated that the differences were larger than what would have been expected due to stable accumulation of round off error. Further investigation into these instabilities is required to properly quantify their effects on the model output.

## 1 Introduction

To use a numerical hydrological model such as MESH for both scientific and operational purposes, it is a great advantage if the model run time is as short as possible while still providing scientifically meaningful results. In many cases, hydrological models require extensive calibration, which may entail thousands of model runs, before the model can be run in an experimental fashion. In addition, the model may be used to simulate macro- or continental-scale river basins, or it may be run for many decades for climate-change studies. In fact, if the model run time is too long, it may not be possible to use the model for the required purpose. Currently, run time is often reduced by utilizing smaller research domains, larger time steps, smaller temporal periods, a limited number of calibration runs, multiple model instances concurrently with different calibration parameters, limited land cover types, and/or faster computer hardware. However, some of these techniques (decreased model domain resolution, for example) can result in a loss of model predictive power and therefore should be avoided whenever possible. The utilization of faster computer hardware has been the standard response to date; however, although transistor counts are increasing in accordance with Moore's Law, overall clock speeds of modern computers have not increased at the rate previously seen due to heat dissipation as well as other physical constraints (*Herlihy and Shavit*, 2008). Therefore, relying on increased clock speed to decrease model run time is no longer a viable option. Recently microprocessor manufacturers such as Advanced Micro Devices, Inc. (AMD) and Intel have begun to ship Central Processing Units (CPUs) with multiple execution "cores", as well as developing multi-CPU, shared-memory platforms that communicate directly through shared hardware caches (*Herlihy and Shavit*, 2008). These multi-CPU, multi-core platforms are able to exploit parallelism by utilizing multiple processors (and/or cores) to decrease execution time. Through such exploitation, problems can be broken down into smaller problems that are solved concurrently. Some problems do not lend themselves to parallelization, while others do. *Pietroniro et al.* (2007) examined this briefly and stated that because "... [MESH model] domain is broken into subareas and the land-surface model is run independently on each area", MESH lends itself well to parallelization.

In this work, MESH 1.3 was examined via code profiling to determine the slow portions of code (termed "hot spots"). Focus was given to determining whether the code, as currently written, could be adapted for parallelization targeting shared-memory processors and whether various code optimizations could be made to the code structure. These optimizations are important for future work that incorporates computationally expensive physics into the model. Given that the MESH usage model requires the user to calibrate the model via multiple model runs, time lost to this stage can hinder the results if the model information cannot be applied in a timely fashion. By decreasing the run time, model users can quickly iterate over calibration parameters allowing more time to be spent on the science rather than waiting for the code to finish.

## 2 Profiling

Code profiling was utilized to determine the time spent in each segment of the MESH code during a typical model run. MESH version 1.3 standalone (to be released Summer/Fall 2009) was compiled with gfortran[1] version 4.5.0 20090421 (experimental) [trunk revision 146519] under Ubuntu Linux 8.10 and profiled with Intel V-Tune[2] version 9.1 Update2-226 for Linux. Third-level optimization (O3) was used. The example basin BWATER was utilized with a total profiling run time of approximately 50 minutes on a CoreDuo laptop running at 1.2GHz with 2GB of RAM and a 5400rpm hard drive. This was selected as a worst-case modeling platform; see Section 6 for justification.

The profiling results are shown in Table 1.

[1]http://gcc.gnu.org/fortran/

[2]http://software.intel.com/en-us/intel-vtune/

Table 1: Profiling results: overall system

```
                42% MESH
             22% libm (math routines)
            6%  libgfortran (runtimes)
          -------------------------------
                  70% total
     +  30% other linux process (kernel activity, etc)
                    ------
                100% of run time
```

Breaking the MESH results into a per-function list provided the results shown in Table 2.

Table 2: Profiling results: MESH on a per-function basis

```
     MAIN     18%
     FLXSURFZ 15%
     CLASSS   12%
     CLASSG    8%
 ---------------------
            = 53% of MESH
    GRDRAN    6.8%
    TPREP     3.9%
    APREP     3.8%
    TMCALC    2.5%
    WPREP     2.4%
    WAT_DRAIN 2.3%
    CLASST    2.2%
   TSOLVC    1.95%
   CLASSW    1.83%
 ---------------------
            = 27% of MESH
total of above functions = 80% of MESH
```

A full explanation of these functions can be found in the CLASS 3.4 documentation; however a brief overview of the top four hot spots follows for the reader's convenience.

**MAIN** is the model driver. It is responsible for loading configuration files, reading forcing data, writing output data, and running the main computation loop that iterates over the temporal model domain. It is the entry point of MESH.

**FLZSURFZ** estimates a stability parameter, the Richardson Bulk number, and uses it to estimate a corresponding Monin–Obukhov length that corresponds to the stability parameter. This is solved for via the Newton–Raphson method.

**CLASSS** "scatters" the 2D arrays into 1D vectors. The rationale behind this is that it is faster to traverse sequential memory than it is to access "2D" memory.

**CLASSG** “gathers” the 1D vectors into 2D arrays.

## 3 Hot-spot functions and proposed changes

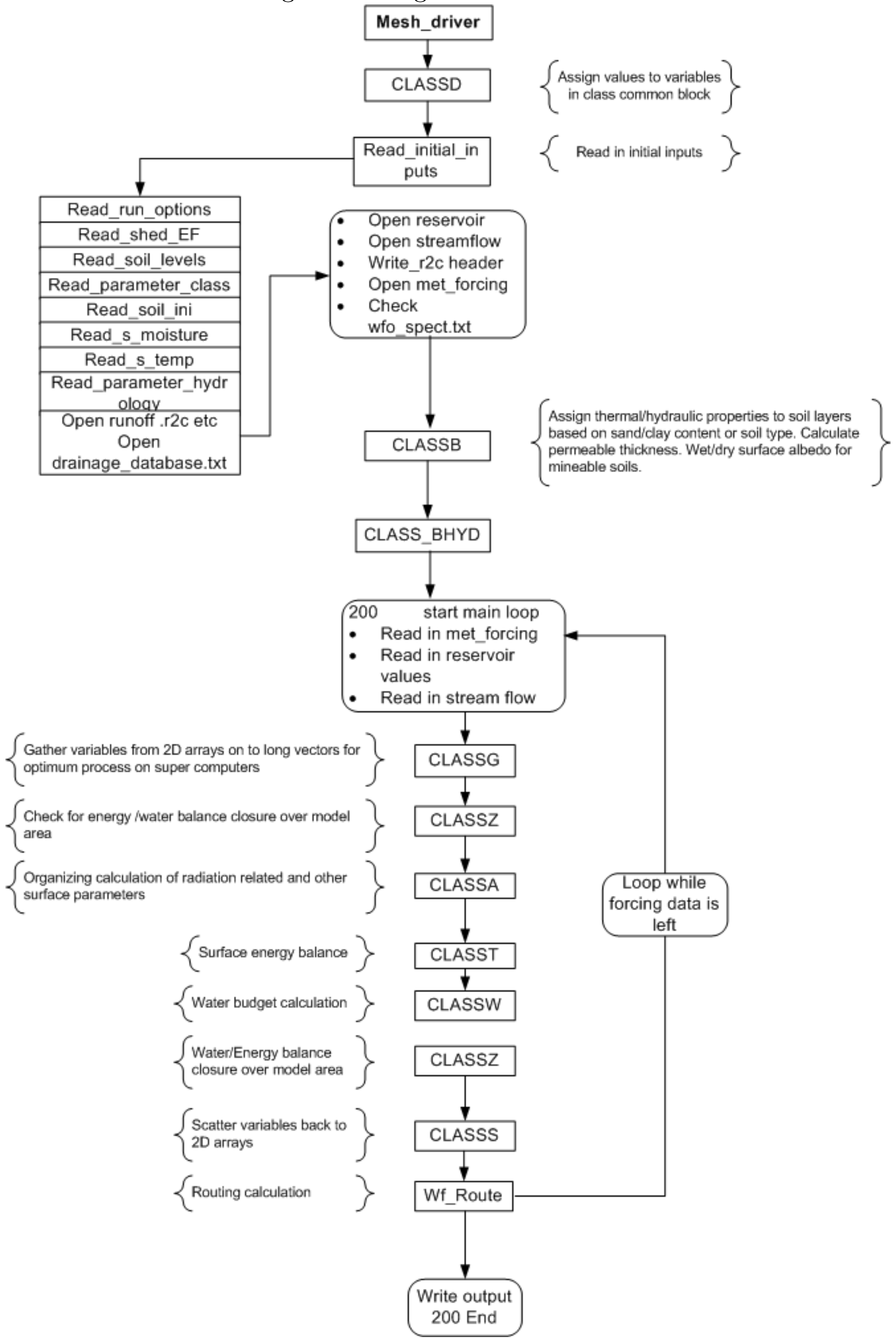
Over 50% of the MESH run time can be accounted for in four functions, MAIN, FLXSURFZ, CLASSS, and CLASSG. Below, each function is examined in detail, outlining where they can be improved. These improvements are examined within the context of the x86 and x64 shared-memory architectures.

### 3.1 MAIN

Figure 1 outlines the general program flow of routine MAIN. MAIN begins by reading in the input configuration files and initializing variables (CLASSB and CLASS_BHYD) that define the model domain. Each iteration of the main loop (200 continue) begins by reading in the current time step’s meteorological forcing data, reservoir values, and stream flow. CLASSG gathers all of the forcing data and computed (from the previous time step, if applicable) values into 1D vectors from the 2D arrays. CLASSZ/A,T,W,Z are run using the 1D vectors. CLASSS scatters the 1D vector into the 2D arrays for use with wf_route and the current time step output routines at the end of the main loop.

It is important to note that the “vectorization” performed by CLASSS and CLASSG is not the data parallel vectorization associated with Single Instruction Multiple Data (SIMD) instruction sets such as SSE. Rather, it is literally the transformation of 2D arrays into 1D arrays, commonly referred to as “vectors”.

Figure 1: Program flow of MAIN


Mesh_driver
CLASSD
Assign values to variables in class common block
Read_initial_inputs
Read in initial inputs
Read_run_options
Read_shed_EF
Read_soil_levels
Read_parameter_class
Read_soil_ini
Read_s_moisture
Read_s_temp
Read_parameter_hydrology
Open runoff .r2c etc
Open drainage_database.txt
Open reservoir
Open streamflow
Write_r2c header
Open met_forcing
Check wfo_spect.txt
CLASSB
Assign thermal/hydraulic properties to soil layers based on sand/clay content or soil type. Calculate permeable thickness. Wet/dry surface albedo for mineable soils.
CLASS_BHYD
200 start main loop
Read in met_forcing
Read in reservoir values
Read in stream flow
CLASSG
Gather variables from 2D arrays on to long vectors for optimum process on super computers
CLASSZ
Check for energy /water balance closure over model area
CLASSA
Organizing calculation of radiation related and other surface parameters
CLASST
Surface energy balance
CLASSW
Water budget calculation
CLASSZ
Water/Energy balance closure over model area
CLASSS
Scatter variables back to 2D arrays
Wf_Route
Routing calculation
Loop while forcing data is left
Write output
200 End

The primary task of the routine MAIN is to control data input/output (I/O), and its performance is bound to the underlying hardware, where and increase in storage medium performance translates into faster overall performance. However, there are still various optimizations that can be performed, specifically in the main loop. Disk input and output transfer rates and access latency are bound to the physical limitations of the hard drive because it is essentially the only mechanical component of a modern computer. Sequential reads and writes to a hard drive are significantly faster than any random access reads or writes. Therefore, care should be taken to limit the random I/O requests. Although this is done in the MAIN setup functions where entire files are read in sequence, each iteration of the main loop has random disk I/O requests that are performed to read in the current time step's forcing data. A proposed solution to this problem is as follows.

Instead of reading in the data at each time step, the meteorological forcing data are read into a buffer before running the main loop. This results in a reduction of disk I/O during the main loop. However, because the forcing data can be gigabytes in size, an algorithm that can allocate a maximum buffer size and subsequently refill it as needed needs to be developed so as to limit the total number of random read/write requests. For example, if the computer running the model has 2GB of RAM and 6GB of forcing data is needed, the buffer is taken to be 1GB and is filled a total of 6 times. The disk output performed at the end of the main loop can be buffered in a similar fashion, allowing for large sequential writes, increasing data throughput.

This solution is presented below in Algorithm 1.

**Algorithm 1** Buffered input for MAIN

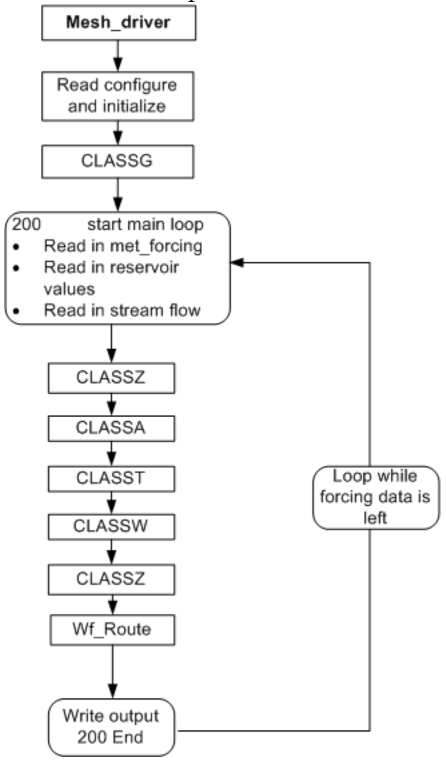
```
Read in configuration data
Initialize variables
Determine maximum input buffer size
Allocate and fill buffer with forcing data
Determine maximum output buffer size
Allocate output buffer
200 start of main loop
    if buffer is not empty then
        proceed to read from buffer
    else
        if not at end of simulation period
            fill buffer from forcing files
        end if
    end if
end 200
```

The problem with this approach is determining a correct buffer size for the I/O. Care must be taken that buffers are not larger than total physical memory or else paging begins, thus negating any performance gains. A memory manager algorithm would have to be able to handle other programs utilizing physical memory and dynamically adjust the buffers to limited paging. A better approach might be to utilize smaller fixed size buffers so as to minimize the risk of paging or memory shortages. User-configurable buffers of about 100MB are likely ideal. It would limit the disk I/O per iteration, and if the user knows there is sufficient memory, they can allocate a large buffer at their discretion. Further work is required to determine an optimal buffer size.

## 3.2 CLASSS and CLASSG

Combined, the two routines CLASSS and CLASSG account for 20% of the total run time. Although their intent is to reduce access to memory, they end up greatly increasing the model run time. In examining Figure 1, it can be seen that only the subroutines wf_route and the output routines require 2D arrays as input. A proposed change is illustrated in Figure 2, where the gather routine has been placed before the start of the main loop, the scatter routine is completely removed, and input time step values, wf_route, and time step output have all been rewritten to utilize 1D vectors.

Figure 2: Modified MAIN loop to remove CLASSS and CLASSG



### 3.2.1 Parallelization

CLASSG has iteration-independent loops that can be parallelized via an API such as OpenMP. It should be noted that with multiple threads requesting memory, there must be sufficient memory bandwidth for this approach to result in a net performance gain.

Further work is require in order to determine when parallelization of these loops is inefficient.

## 3.3 FLXSURFZ

This function is called many times and highlights one of the larger problems currently present in MESH 1.3. The subroutine logic is outlined in Algorithm 2

**Algorithm 2** FLXSURFZ subroutine control logic

```
let e be the number of elements in 1D domain comprised of all
    the areas over which to solve. Equal to IL2 in the code.
for each i in e
    Calculate Bulk Richardson number (Rib)
    User Rib to estimate 1/L (Monin-Obukhov length)
end for
for 1 to 3
    for each i in e
        Calculate new approximation for 1/L via Newton-Raphson in 3 iterations
    end for
end for
for each i in e
    Calculate 1/L and stability functions from log-linear profile
end for
```

The inefficiency presented in Algorithm 2 is that $3 * e$ iterations are done whereas it can be solved in $e$ iterations. Algorithm 2 can be rewritten as shown in Algorithm 3.

**Algorithm 3** Modified FLXSURFZ

```
let e be the number of elements in 1D domain comprised of all the
    areas over which to solve. Equal to IL2 in the code.
for each i in e
    if first iteration
        Calculate Bulk Richardson number (Rib)
        User Rib to estimate 1/L (Monin-Obukhov length)
    end if
    for 1 to 3

        Calculate new approximation for 1/L via Newton-Raphson in 3 iterations
    end for
    Calculate 1/L and stability functions from log-linear profile
end for
```

There appears to be a bug in the implementation of the **for** 1 to 3 loop in FLXSURFZ as it currently exists in MESH 1.3. The paper cited in the code[(*Sharan and TVBPS*)] indicates that 2 iterations should be necessary for convergence to the root. However, the **do** loop performs 3 iterations, and there is a check immediately before the Newton–Raphson step, which does not update the 1/L approximation on the third iteration. It needs to be determined if this is intentional. Otherwise, the last iteration can be removed, further limiting redundant work.

As stated above, Algorithm 2 is an example of the redundancy present in many of the subroutines. The changes made here can also be applied to the other routines, such as CLASSZ, CLASSA, CLASSB, CLASSW, and wf_route.

### 3.3.1 Parallelization

Due to the iteration-independent nature of FLXSURFZ, the main loop can be parallelized via an API such as OpenMP. This parallelization strategy is applicable throughout the MESH and CLASS code because many loops are iteration independent.

## 4 Code clean up

Throughout the code, there are areas of dead code that are consuming CPU cycles without any purpose. Examples of this are:

- The loop that begins at line 3044 in MESH_DRIVER. It does nothing and should be removed.
- The loop with dead code in CLASSG.

Such pieces of code should be commented out or removed entirely because they add to the computational load.

## 5 OpenMP

It is proposed that the loop parallelization is done with OpenMP[3] which, as per the OpenMP website, is

> ... [an] Application Program Interface (API) [that] supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms. Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

It is proposed that this API is utilized due to its simplicity for integration into the current code base. As well, because the OpenMP compiler directives are specialized comments (in Fortran), if the compiler does not support the OpenMP directives they are ignored.

## 6 Hardware and software survey

Because MESH may be run on a variety of hardware configurations, a hardware survey of six of the main MESH user groups was conducted. The survey questions asked about CPU type and speed, RAM, hard drive size and speed, operating system, and the compiler brand. The results indicate that the average CPU speed is about 2GHz, dominated by Intel Core2Duo CPUs, with only two users reporting Intel Quad Core CPUs. A single user reported a 4x AMD Opteron cluster. Windows XP and Vista were the only operating systems used other than the HP Linux OS for the Opteron cluster. Average RAM was in the 2–3GB range. A single user reported 512MB–1GB of RAM. Because most users are utilizing desktops, 7200rpm hard drives were the majority; however a single 5400rpm and a single 4200rpm drive were also reported — these users were conducting model runs on laptops. The compilers used varied considerably and included:

- Compaq Visual Fortran 6.6.c
- Intel Visual Fortran 10
- Intel Visual Fortran 11
- Portland Group F90
- Unknown
- None

The user who reported "none" was using a precompiled binary. We note that Intel purchased the Compaq Visual Fortran compiler, and as such, the Compaq Visual Fortran compiler can be regarded as an old version of Intel's Visual Fortran Compiler. Only one of these compilers, the Intel Fortran Compiler, has support for OpenMP 3.0. We are currently unsure whether the other compilers support OpenMP because the users did not know when questioned and time constraints limited further investigation.

[3] http://openmp.org/wp/

## 7 Interpretation of the survey

The survey demonstrates that the vast majority of users are running MESH on limited hardware. Although it was not asked, we believe it to be a fair assumption that the RAM speed is also limited (given the limited hardware reported). This can be a problem for highly sustained memory read/writes as well as random access times. As well, more end-user education regarding compilers is needed. Some of the compilers reported are no longer in production or no longer supported, rendering them essentially obsolete. It is probably not well known in this user community that large performance benefits can be obtained by using newer, optimizing compilers. Because most of the researchers are using Intel CPUs, using the Intel Fortran Compiler (which produces optimized code for Intel CPUs) is expected to result in large performance increases. Testing showed that it is possible to halve the run time when moving from an open-source compiler such as g95 to the Intel Fortran Compiler when running on Intel brand CPUs. Becuase most of these users are using shared-memory architectures, they are able to benefit from OpenMP parallelized loops. However, OpenMP requires compiler-level support that requires newer, supported compilers. Many of the old compilers have limited or no support for OpenMP and no support for the new OpenMP 3.0 specification. It is probably safe to assume that the optimizations provided by the "new", "modern" compilers are considerably more efficient than the "older" compilers.

## 8 Code Modifications

Modifications were carried out to the MESH 1.3 class base to bring the internal CLASS subroutines up-to-date as well as to increase performance as per the suggestions in Section 3.

### 8.1 Updates to CLASS code

There have been updates to the core CLASS code since MESH 1.3 was finalized. This CLASS update was termed *CLASS 3.4 revised* abbreviated to *CLASS 3.4r.* A subsequent update from the *Pollux* machine was made in May 2009. The revised code (3.4r) was integrated into MESH 1.3 at the version number 1.3.1. The code from Pollux was integrated at the version number 1.3.2. The modifications described below increased the version to MESH 1.3.3.

### 8.2 Instabilities

In order to verify that the changes integrated into the MESH code base during the implementation of the OpenMP commands did not contain bugs, it is important to obtain reference output prior to the code changes. In this way, the output from the new code can be compared to that of the old code. During the creation of this reference output, it was observed that different compilers produced different results with errors larger than single-precision round off. It was hypothesized that the algorithms for solving the equations used in MESH were inherently unstable, and that the small perturbations introduced during the reordering of instructions by the compiler were responsible for producing the different output. It was assumed that the compiler that had the greatest chance of producing instructions that would limit the perturbations would be the commercial Intel Fortran compiler. In order to fully quantify this behaviour the following approach was taken:

1. CLASS 3.4 was compiled with the Intel compiler with no optimizations (O0) and a strict floating-point flag. This was done to reduce as much instruction reordering as possible. This was taken as the known or reference solution.

2. CLASS 3.4 was compiled with g95 and gfortran in a very similar configuration (O0, etc.). This was taken to be an experimental solution.

3. The absolute difference $|experimental - known|$, was computed between the g95 and Intel results and the gfortran and Intel results.

4. The relative difference $\frac{experimental-known}{known}$ was computed between the g95 and Intel results and the gfortran and Intel results.

5. Graphs were constructed that showed that there was accumulation of error.

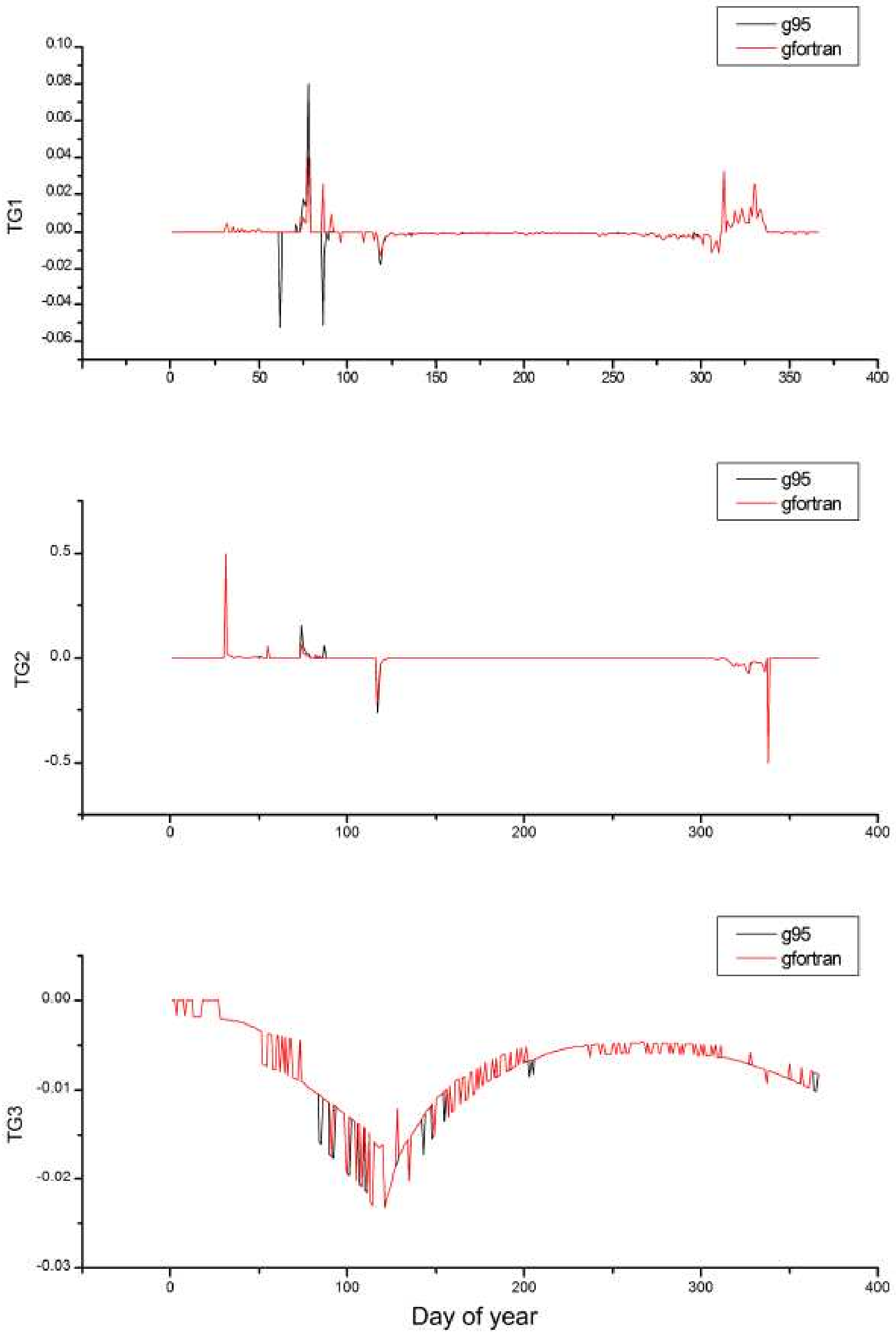
6. Steps 1–5 were repeated for CLASS 3.4r showing the same results.

7. CLASS 3.4r was incorporated into MESH 1.3, under the revision MESH 1.3.1

8. Steps 1–5 were run for MESH 1.3.1, showing the same instabilities.

The results show that the instabilities are present in the CLASS output (however MESH specific subroutines did not seem to be affected). An example of the instabilities is presented below in Figure 3. The variables presented are TG1, TG2, and TG3, the three soil layer temperatures, averaged to daily values. Figure 3 is the relative difference between the output from the gfortran and g95 compilers compared to the Intel Fortran compiler.

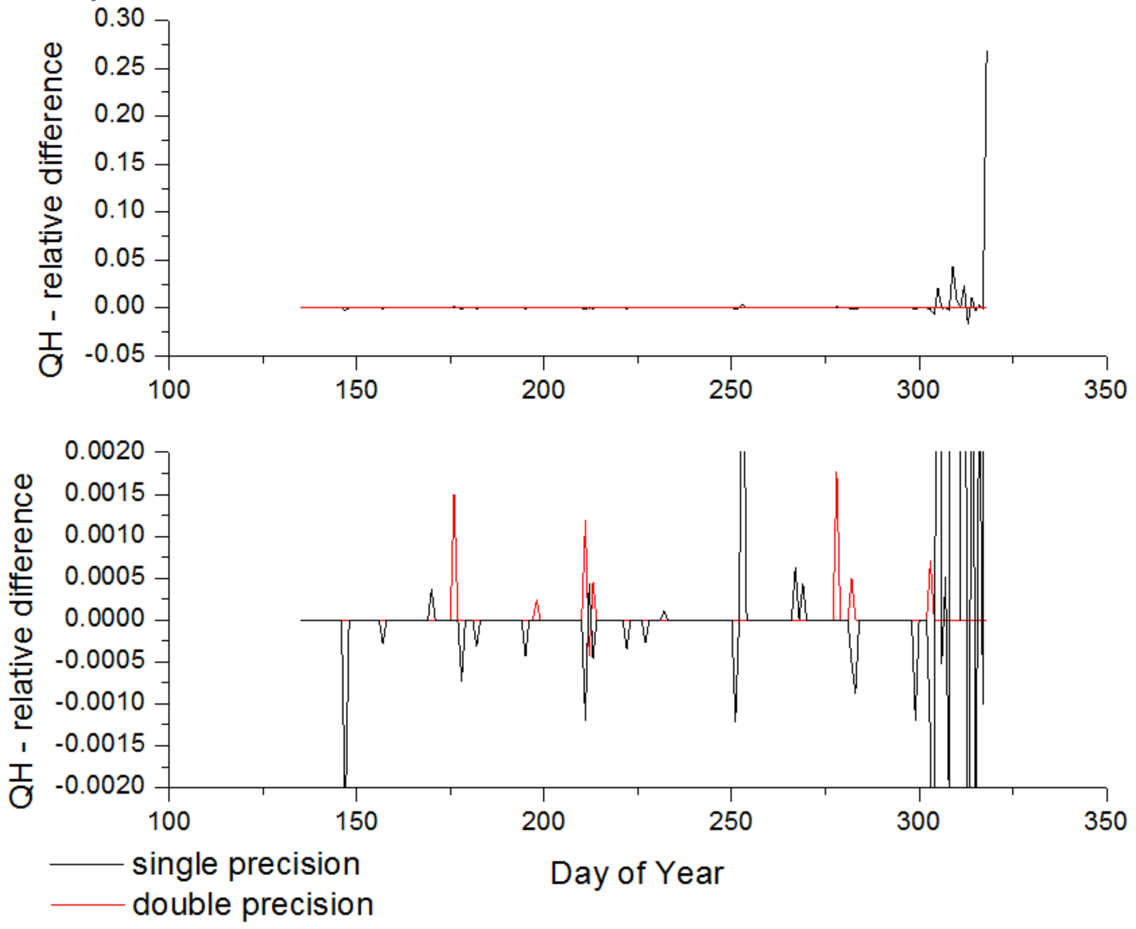
Figure 3: Instability from CLASSOF2 variable TG1–3 — relative differences. X-axis is offset from day 135 (the simulation start day).



## 8.3 Floating-Point Precision

All of MESH (and the internal CLASS code) is written using single-precision data types. This can lead to accumulation of round off errors that can be mitigated by moving to double precision. This is highlighted in Figure 4, which compares the relative difference between compilers using double precision (64bit) and single (real) (32bit) precision. As is evident, the move towards double precision helped to mitigate the round off errors. In the context of most MESH users and researchers, there is no reason not to be using double precision. For most compilers, there are flags which can override the *real* data type, so that double precision can be used without any modifications to the code.

Figure 4: Double precision versus single precision — relative difference. X-axis is offset from day 135 (the simulation start day). The bottom plot shows the top plot in greater detail to more accurately see the small differences.



## 8.4 Removal of CLASSS and CLASSG

As indicated in section 3.2, CLASSS and CLASSG are responsible for reorganizing the internal arrays from a 2D version to a 1D version. This results in a substantial runtime performance impact. Upon examination of the code, it was determined that few code sections were utilizing the 2D arrays and that the code could be partially rewritten to remove *CLASSS* and *CLASSG*. However, due to the mixed code base, there is a requirement that as little code as possible be changed to allow for easy integration of new CLASS features with the existing MESH code. Therefore the following criteria were established:

- Divide the code into five separate sections

  1. Initialization, pre *200 continue*
     - This section contains most of the model I/O required for setting up a MESH run.
     - It reads the input data into the 2D arrays for easy debugging.
  2. Post *200 continue*, pre *CLASSG*
     - This section is from the start of the main loop to to start of CLASSG.
     - This is where the forcing data are read in.
     - The data are read into 2D arrays
  3. Post *CLASSG*, pre *CLASSS*
     - This is where all the CLASS* subroutines are called.
     - These operate on the “gathered” 1D vectors.
  4. Post *CLASSS*, pre *200 end*
     - This is where wf_route is run and output is written to disk
     - Operates on 2D arrays
  5. Post *200 end* to *end*
     - This is where the finalized output is generated.
     - Operates on the 2D arrays.

- For each sections above, the following modifications were performed

  1. Move CLASSG and GATPREP above the 200 continue.
  2. Because some input is done in this section, leave it as is (few variables need to be "gathered"). Therefore a new "gather" routine was created called "gather" that only gathers the few arrays that are modified in this section into the 1D vectors. In addition some arrays need to be zero-ed out each iteration.
  3. No modification
  4. Remove CLASSS. Rewrite all the nested loops of the form:

     ```
       DO  I=1, NA
               DO M=1, NMTEST
     ```

     Into:

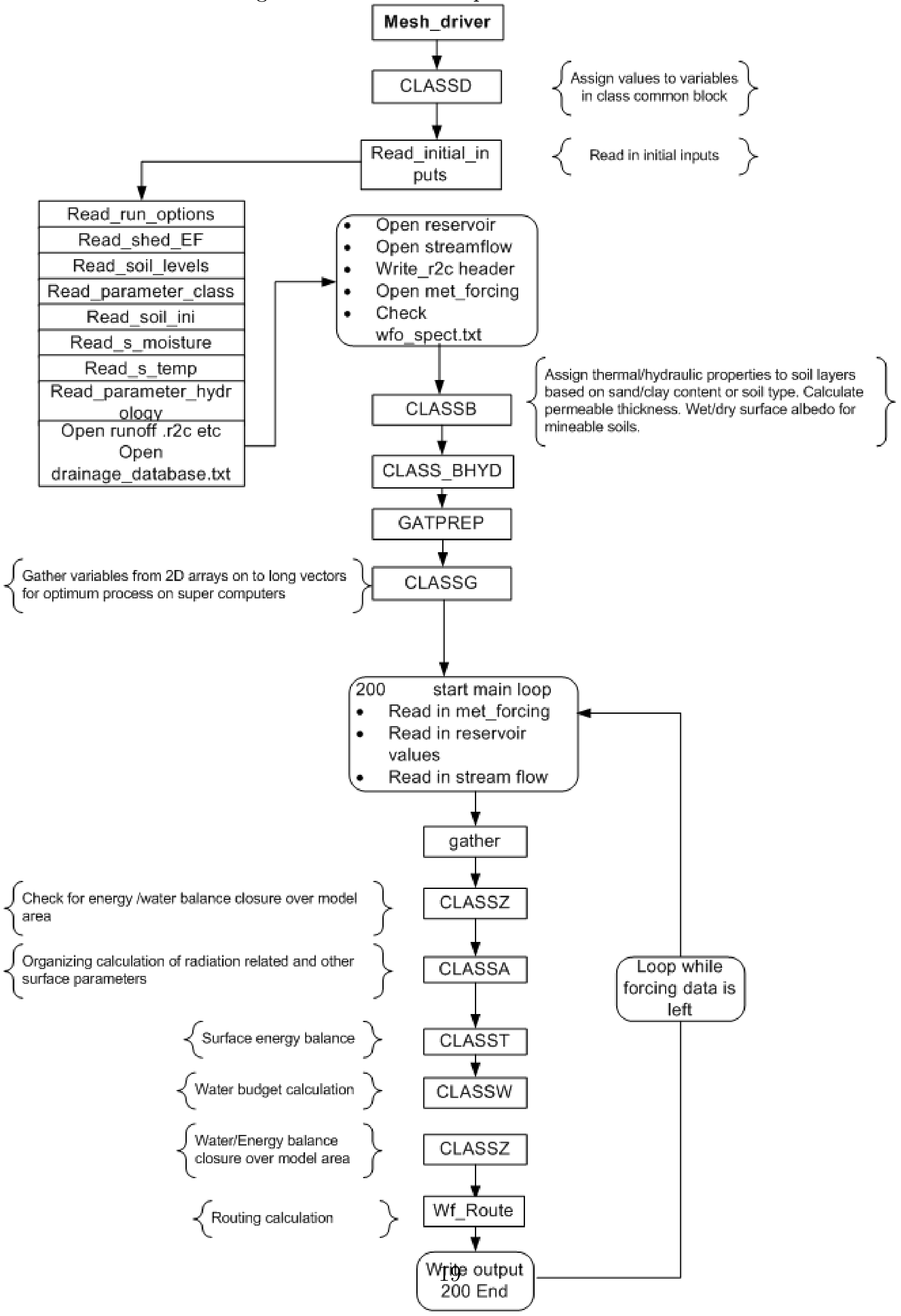
     ```
     DO  I=1, NML
     ```

     And replace all $*ROW$ variables with the $*GAT$ equivalents.

     Some loops require looping over NA, so utilize *ilmos(I)* for the correct indexing.
  5. Following the same method in point four, rewrite the loops to operate on the 1D vectors ($*GATs$ instead of $*ROWs$).

These changes allow large sections of code to maintain their structure while incorporating the new changes. All of the input remains in 2D form for easier debugging, and because of the iteration sequence, the output maintains its form. The addition of the minor gather routine preserves the logic at the start of the main loop and provides little overhead. The new code structure is shown below in Figure 5.

Figure 5: MESH 1.3.2 updated control flow


Mesh_driver
CLASSD
Assign values to variables in class common block
Read_initial_in puts
Read in initial inputs
Read_run_options
Read_shed_EF
Read_soil_levels
Read_parameter_class
Read_soil_ini
Read_s_moisture
Read_s_temp
Read_parameter_hydr ology
Open runoff .r2c etc Open drainage_database.txt
• Open reservoir
• Open streamflow
• Write_r2c header
• Open met_forcing
• Check wfo_spect.txt
CLASSB
Assign thermal/hydraulic properties to soil layers based on sand/clay content or soil type. Calculate permeable thickness. Wet/dry surface albedo for mineable soils.
CLASS_BHYD
GATPREP
Gather variables from 2D arrays on to long vectors for optimum process on super computers
CLASSG
200 start main loop
• Read in met_forcing
• Read in reservoir values
• Read in stream flow
gather
Check for energy /water balance closure over model area
CLASSZ
Organizing calculation of radiation related and other surface parameters
CLASSA
Loop while forcing data is left
Surface energy balance
CLASST
Water budget calculation
CLASSW
Water/Energy balance closure over model area
CLASSZ
Routing calculation
Wf_Route
Write output 200 End

### 8.5 Other improvements

Other improvements and optimizations were done in addition to the above section:

- Collapsed the three loops in FLXSURFZ into one loop and parallelized it.
- Parallelized all NML loops in mesh_driver.f90.
- Removed dead code in CLASSG.
- Parallelized NML loop in gather and CLASSG.

### 8.6 Known problems

Currently Green Kenue (formerly EnSim Hydrologic) output and save/restore state subroutines do not work as anticipated. These routines are only called if certain flags are set and did not impact the benchmarking results.

## 9 Run time improvements

### 9.1 Speedup between MESH 1.3.2 and 1.3.3

MESH 1.3.2 was benchmarked against MESH 1.3.3. They were run on an Intel Core2Quad Q6600 at 3.2GHz with 8GB of DDR2 888MHz RAM. Each build was run five times for the BWATER MESH example and the fastest of these five was taken. The following compilers were used:

- gfortran version 4.5.0 20090421 (experimental) [trunk revision 146519]
- g95 version 0.92 (GCC 4.0.3 May 7 2009)[4]
- Intel Fortran (ifort) 11.1.038 [5]

The first two compilers were run under Cygwin[6] 1.5.25-15.

Because three different compilers were used, the optimization flags present for one compiler may not be present for another. In such cases, the flag was still used to create a best-case scenario.

The flags used for the ifort, gfortran, and g95 compilers are shown in Table 3, Table 4, and Table 5 respectively.

[4]http://www.g95.org/
[5]http://software.intel.com/en-us/intel-compilers/
[6]http://www.cygwin.com/

Table 3: Optimizations for ifort

| Flag | Description |
|---|---|
| O3 | Level 3 optimization |
| Og | Global optimization |
| QxSSSE3 | Core2Quad extended SSE3 |
| QaxSSSE3 | Core2* SSE3 optimizations |
| arch:SSE3 | Optimize and generate SSE3 code that runs on both Intel and AMD architecture. |
| assume:buffered_io | I/O accumulates in a buffer and is written out. No code modification. |
| heap_arrays0 | Allocate all arrays on heap. See note below. |
| Qip | Inter-procedural optimization |
| Openmp | Enable processing of OpenMP directives |
| fp:strict | Strict floating point |
| Qfp-speculation=off | Turn off floating-point speculation |

Note 1: In order for MESH 1.3.x to run without crashing, the flag heap_arrays0 must be enabled on the ifort compiler; otherwise all of the arrays are declared on the stack instead of the heap, resulting in a stack overflow. It requires further investigation to see if this flag had an effect on the overall run time. There does not appear to be an equivalent flag for the GCC compilers.
Note 2: On this test platform, the prefetch optimization in the ifort compiler degrades performance significantly and should not be used.

Table 4: Optimizations for gfortran

| Flag | Description |
|---|---|
| O3 | Higher level optimizations |
| ftree-vectorize | Allow vectorization (turned on under O3) |
| msse3 | SSE3 vectorization |
| fopenmp | Process OpenMP directives |

Table 5: Optimizations for g95

| Flag | Description |
|---|---|
| O3 | Higher level optimizations |
| ftree-vectorize | Allow vectorization (turned on under O3) |
| msse3 | SSE3 vectorization |

Note: There is no OpenMP support for the g95 compiler.

The speed up was then calculated as

$$\frac{Time_{v1.3.2} - Time_{v1.3.3}}{Time_{v1.3.2}} \times 100\%.$$

## 9.2 Benchmark results

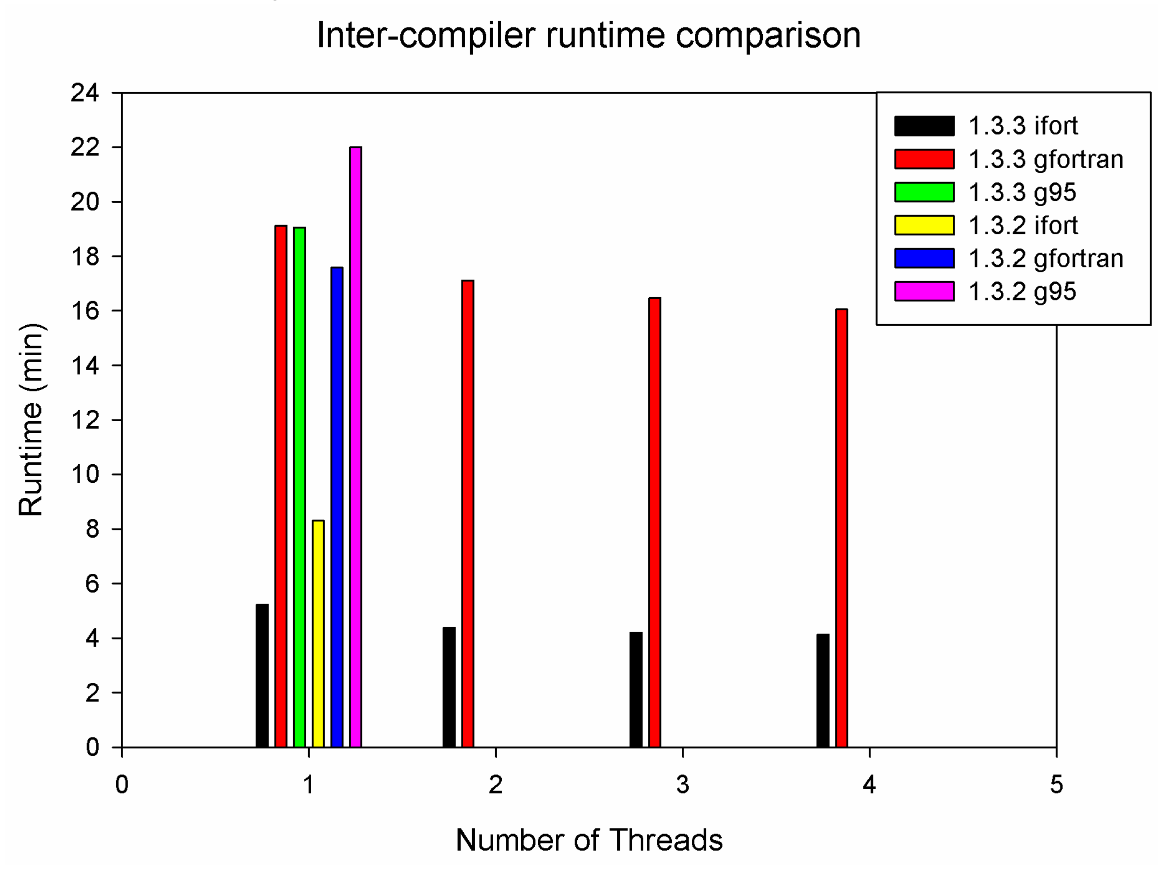
Table 6 shows the results of this benchmarking.

Table 6: Run time improvements

| Version | Compiler | Run Time | Number of Threads |
|---|---|---|---|
| 1.3.3 | ifort | 4m8.5s | Auto |
| 1.3.3 | gfortran | 16m2.5s | Auto |
| 1.3.3 | ifort | 4m7.7s | 4 |
| 1.3.3 | gfortran | 16m2.7s | 4 |
| 1.3.3 | ifort | 4m12.6s | 3 |
| 1.3.3 | gfortran | 16m28.0s | 3 |
| 1.3.3 | ifort | 4m22.5s | 2 |
| 1.3.3 | gfortran | 17m7.2s | 2 |
| 1.3.3 | ifort | 5m12.9s | 1 |
| 1.3.3 | gfortran | 19m9.6s | 1 |
| 1.3.3 | g95 | 19m3.8s | 1 |
| 1.3.2 | ifort | 8m20.2s | 1 |
| 1.3.2 | gfortran | 17m34.6s | 1 |
| 1.3.2 | g95 | 22m0s | 1 |

Figure 6 shows a comparison of all the run times.

Figure 6: Inter-compiler and Thread runtime comparison



## 9.3 Intel ifort percent speedup

Between versions 1.3.3 and 1.3.2 of MESH, a significant speed improvement was realized when using the Intel Fortran compiler.

best MESH 1.3.2 run time = 8.3 minutes
best MESH 1.3.3 run time = 5.2 minutes

and consequently the performance speed up is

$$\frac{Time_{v1.3.2} - Time_{v1.3.3}}{Time_{v1.3.2}} \times 100\% = \frac{8.3 - 5.2}{8.3} \times 100\% \approx 37\%.$$

If however, we count the speed up associated with moving to just two cores, a very common situation as indicated on the hardware survey (Section 6), the speed up is:

MESH 1.3.2 run time with 1 thread = 8.3 minutes;
MESH 1.3.3 run time with 2 threads = 4.4 minutes;

and consequently the performance speed up is

$$\frac{Time_{v1.3.2} - Time_{v1.3.3}}{Time_{v1.3.2}} \times 100\% = \frac{8.3 - 4.4}{8.3} \times 100\% \approx 47\%.$$

Figure 7 shows this in a graphical form. Percent speed up is measured based on 1 thread, that is, the percent speed up gained from adding multiple threads (using MESH 1.3.3).

## 9.4 gfortran

The runtime improvements associated with the gfortran compiler are certainly not as substantial as that of the ifort compiler. The speedup between MESH 1.3.2 and MESH 1.3.3 is:
best MESH 1.3.2 run time = 17.58 minutes
best MESH 1.3.3 run time = 19.16 minutes

$\frac{Time_{v1.3.2}-Time_{v1.3.3}}{Time_{v1.3.2}} \times 100\% = \frac{17.58-19.16}{17.58} \times 100\% \approx -9\%$

This result is undeniably surprising, and currently there is no explanation for why the new build in single-threaded mode causes a performance decrease. The benchmarks have been rerun, recompiled with no OpenMP support, yet the results remain the same. This is currently under investigation. The addition of multiple threads leads to a disappointing performance gain compared to Intel's ifort compiler:

best MESH 1.3.2 run time with 1 thread = 17.58 minutes;
best MESH 1.3.3 run time with 2 threads = 17.10 minutes;

and consequently the performance speed up is

$$\frac{Time_{v1.3.2} - Time_{v1.3.3}}{Time_{v1.3.2}} \times 100\% = \frac{17.58 - 17.1}{17.58} \times 100\% \approx 3\%.$$

Utilizing four threads results in the following:

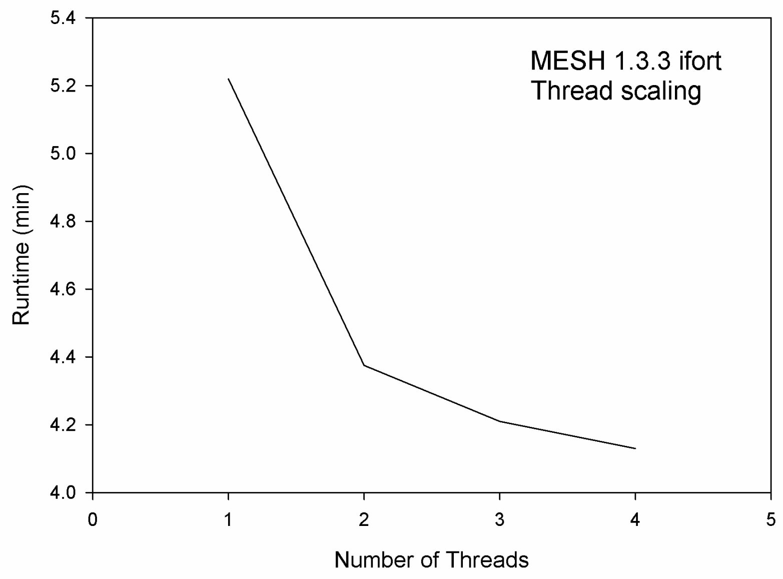
best MESH 1.3.2 run time with 1 thread = 17.58 minutes;
best MESH 1.3.3 run time with 4 threads = 16.05 minutes;

and consequently the performance speedup is

$$\frac{Time_{v1.3.2} - Time_{v1.3.3}}{Time_{v1.3.2}} \times 100\% = \frac{17.58 - 16.05}{17.58} \times 100\% \approx 9\%$$

The performance scaling with respect to number of threads is shown in Figure 8.

Figure 7: MESH 1.3.3 (ifort) Mulithreaded Speedup.



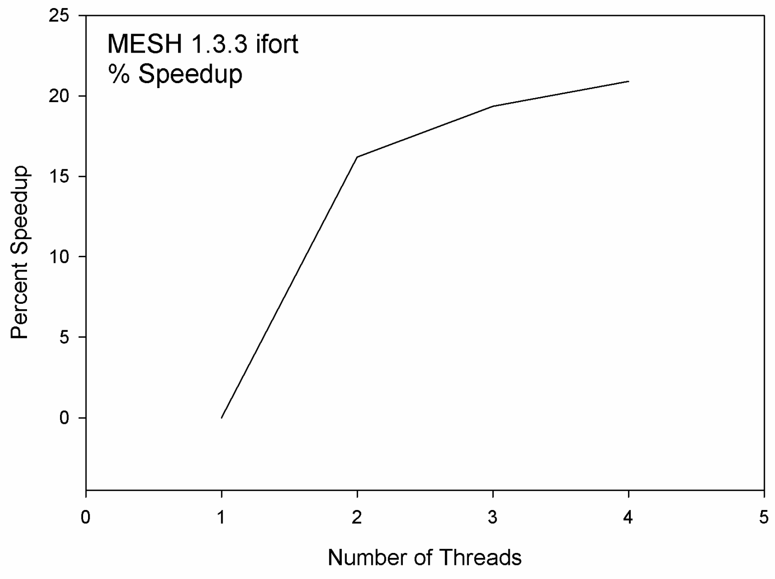
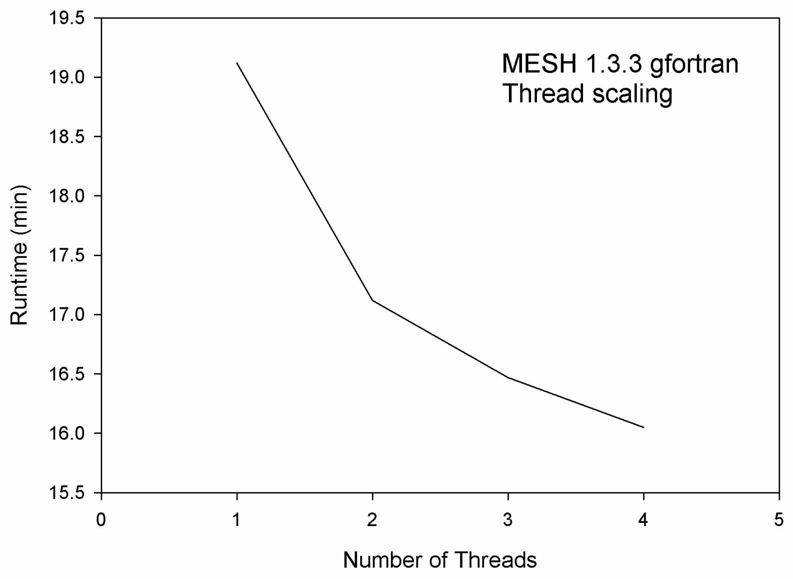
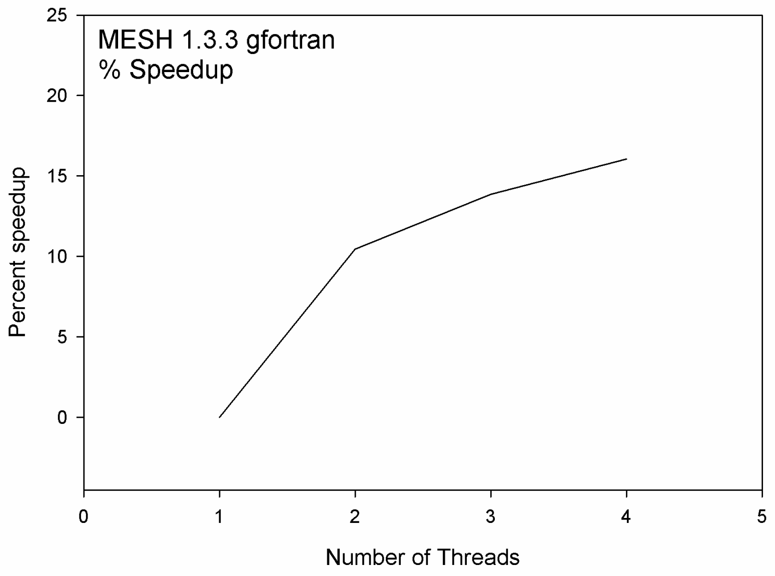


Figure 8: MESH 1.3.3 (gfortran) Mulithreaded Speedup.





### 9.5 g95

The g95 Fortran compiler does not provide support for the OpenMP directives. Therefore it is not possible to measure the speed up obtained from multiple threads. However, it is still possible to examine the performance speedup when comparing a single thread:

$$\text{best MESH 1.3.2 run time} = 22.00 \text{ minutes;}$$
$$\text{best MESH 1.3.3 run time} = 19.06 \text{ minutes;}$$

and consequently the performance speedup is

$$\frac{Time_{v1.3.2} - Time_{1.3.3}}{Time_{v1.3.2}} \times 100\% = \frac{22.0 - 19.06}{22.0} \times 100\% = 13.3\%.$$

## 10 Conclusion

Profiling of MESH has been conducted of MESH 1.3 under Ubuntu Linux 8.10 with Intel VTune. Significant performance bottlenecks were found in the FLXSURFZ, CLASSS, and CLASSG subroutines as well as in MAIN. MAIN's performance bottleneck is mostly attributed to intensive I/O routines at each time step. An overall pattern throughout much of MESH (specifically the CLASS routines) is a duplication of work by having too many iterations due to redundant loops and is exemplified by FLXSURFZ (Algorithm 2). These multiple loops can be merged into one loop, which can be easily parallelized using OpenMP, provided the loop iterations are independent (as is much of the current code). CLASS and CLASSG are only needed for select sections of MESH which can be rewritten to handle 1D vectors. CLASSS can then be completely eliminated and CLASSG moved above the main iteration loop, significantly decreasing the MESH run time. A buffered I/O approach for MAIN was discussed that could be used to reduce the number of random disk input requests. Compiler settings were examined, and the move towards the high-end Intel Fortran compiler can net between a 2x and 3x speed improvement depending on the enabled hardware optimization. Numerical stability was discussed, and it was found that using double precision can mitigate most of the stability problems. However, there exists locations in the code where improved stability can only be achieved via improved numerical methods. For the "average" MESH user, we could think of no reason why double precision should not be used.

## References

Herlihy, M., and N. Shavit (2008), *The art of multiprocessor programming*, Elsevier.

Pietroniro, A., V. Fortin, N. Kouwen, C. Neal, R. Turcotte, B. Davison, D. Verseghy, E. D. Soulis, R. Caldwell, N. Evora, and P. Pellerin (2007), Development of the MESH modelling system for hydrological ensemble forecasting of the laurentian great lakes at the regional scale, *Hydrological Earth Systems Science*, *11*, 1279-1294.

Sharan, M., and R. TVBPS (), Aditi (2003) On the bulk Richardson number and flux-profile relations in an atmospheric surface layer under weak wind stable conditions, *Atmos Environ*, *37*, 3681-3691.