# Improved MESH efficiency via parallelization and code optimization


University of Saskatchewan
Environment Canada / Environnement Canada

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

[1] Center for Hydrology, Dept. Geography, University of Saskatchewan, [2] Simulation Research Laboratory, Dept. Computer Science, University of Saskatchewan [3] NHRC, Saskatoon, Saskatchewan

email: cbm038@mail.usask.ca

Centre for Hydrology
MITACS


## Introduction

Currently, advances in hydrological sciences 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. As well as the finite limits on computer processing power currently available, other issues, such as round off errors for example, 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. In this work, Environment Canada's model Modlisation Environmentale Communautaire (MEC)Surface and Hydrology (MESH) 1.3 [which is based on the Canadian Land Surface Scheme (CLASS) and WATFLOOD] 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 (SMP) 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 MESH commonly requires calibrate 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.

## 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 version 4.5.0 Ubuntu Linux 8.10 and profiled with Intel V-Tune version 9.1. Third-level optimization was used (O3). 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 modelling platform.

| Function | Percentage of runtime |
|---|---|
| Main | 18% |
| FLXSURFZ | 15% |
| CLASSS | 12% |
| CLASSG | 8% |
| Total % of MESH | 53% |

Table 1: Profiling results as a percentage of runtime

**MAIN** is the model driver and entry point of MESH. 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.
**FLZSURFZ** estimates a stability parameter, the Richardson Bulk number, and uses this to estimate a corresponding Monin–Obukhov length that corresponds to the stability parameter. This is solved using 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.

## Parallelization

MESH uses numerous do loops to iterate over the model domain. Many of these loops are *iteration independent*, meaning that any given iteration is not dependent upon any other iteration. These types of iterations lend themselves very well to parallelism. An Application Programming Interface (API) OpenMP was used to introduce SMP parallelism into MESH. OpenMP was chosen because it facilitates fast and flexible SMP parallelization into an existing code base and it is jointly defined by major hardware and software vendors such as Intel, Microsoft, and AMD.

## Code Optimization

Given CLASSS and CLASSG were the two easiest to target bottlenecks, these were addressed first. CLASSS was moved to above the main loop in order to process the input files, and CLASSG was removed entirely. The few subroutines that required the 2D arrays produced by CLASSG were rewritten to use the 1D vectors produced by CLASSS. Basic restructuring is outlined below. Due to the use of these 1D vectors, the do-loops could then be parallelized via the OpenMP API. In order to capture the speed-up that was a result of removing CLASSS and CLASSG as well as running on a multi-core SMP computer, the three popular Fortran compilers were compared: Intel VF, gfortran, and g95. The g95 compiler does not support OpenMP, so g95 was only compared to the other two compilers in single-threaded mode.

| 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 |

Intel ifort speedup

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

$$speedup = \frac{Time_{old} - Time_{new}}{Time_{old}} * 100$$

$$= \frac{8.3-5.2}{8.3} * 100\% \approx 37\%$$

Intel ifort speedup (2threads)

MESH 1.3.2 run time = 8.3min
MESH 1.3.3 run time = 4.4min

$$speedup = \frac{Time_{old} - Time_{new}}{Time_{old}} * 100$$

$$= \frac{8.3-4.4}{8.3} * 100\% \approx 47\%$$

Table 2: Code speedup and inter-compiler comparison

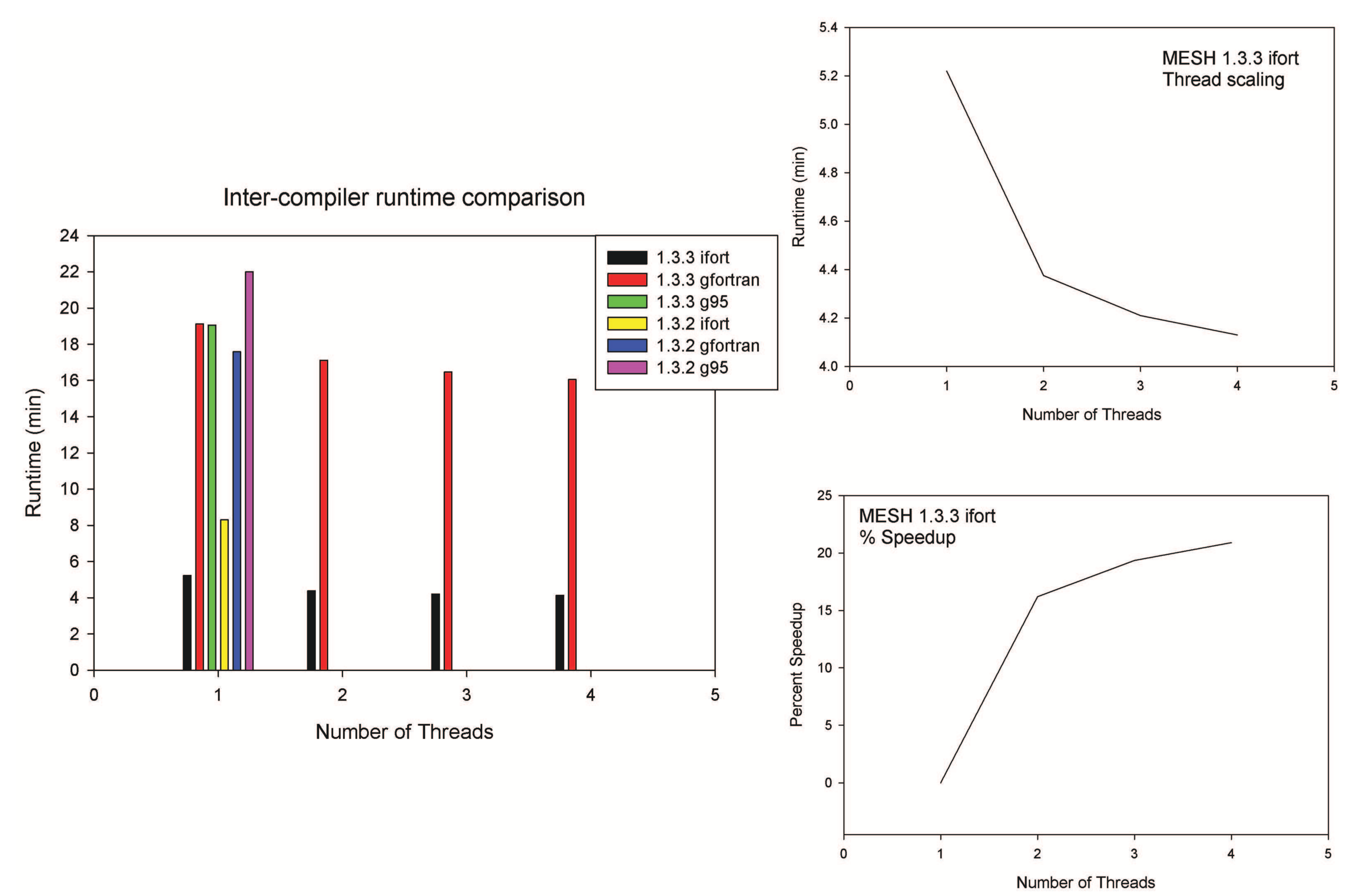


Figure 1: (Left): Compiler performance comparison (Right):Speed up as a function of threads

## Conclusions and future work

Preliminary results to consider the impact of parallelism on runtime speed demonstrated sufficient improvements to suggest that this would be a fruitful approach to apply to other portions of the code in order to achieve even greater increases in speed. The performance numbers show that the code is still dominated by the serial sections. It was also observed that different compilers produced different results, which were greater than what would have been expected due to round off error. CLASS 3.4 revised was used in isolation from MESH and run for a year with the example data set. Intel Visual Fortran, gfortran, and g95 were compared via the relative difference $\frac{experimental - known}{known}$. This is shown in Figure 2 and the variables presented are the three soil layer temperatures, averaged to daily values. Further investigation into these instabilities is required to properly quantify their effects on the output.

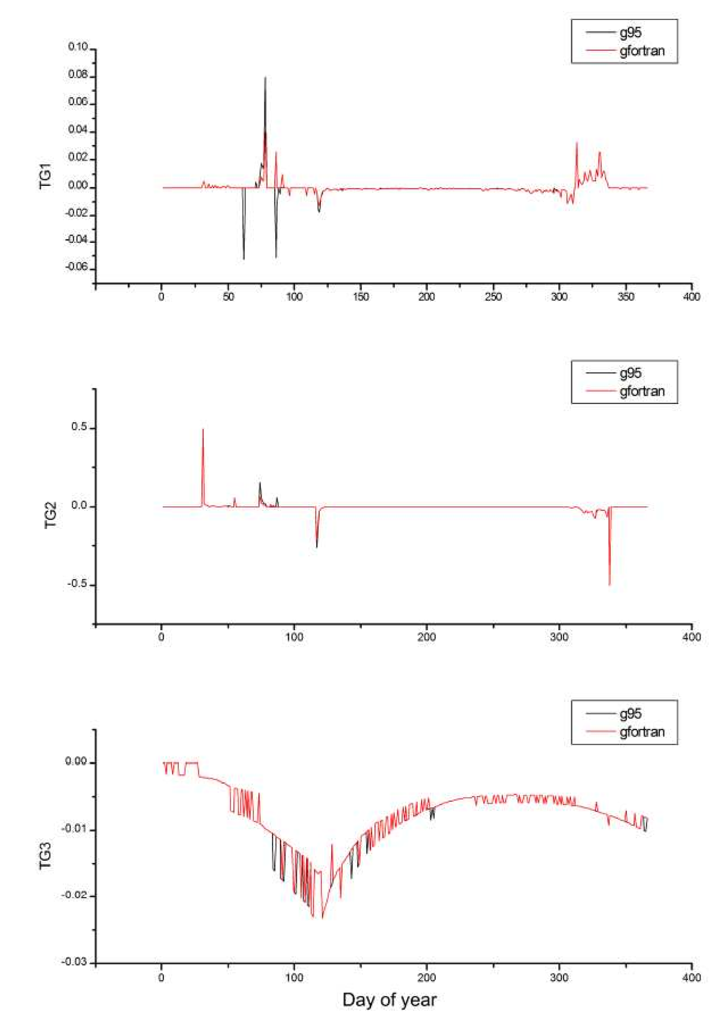


Figure 2: Relative difference between g95 and gfortran compilers compared to IVF

Instabilities can be mitigated (sometimes fully) by moving to a greater floating point precision where the accumulated round off error can have less of an effect on the unstable algorithms. Figure 3 shows a comparison between double and single precision data types showing that the instabilities are partially mitigated.

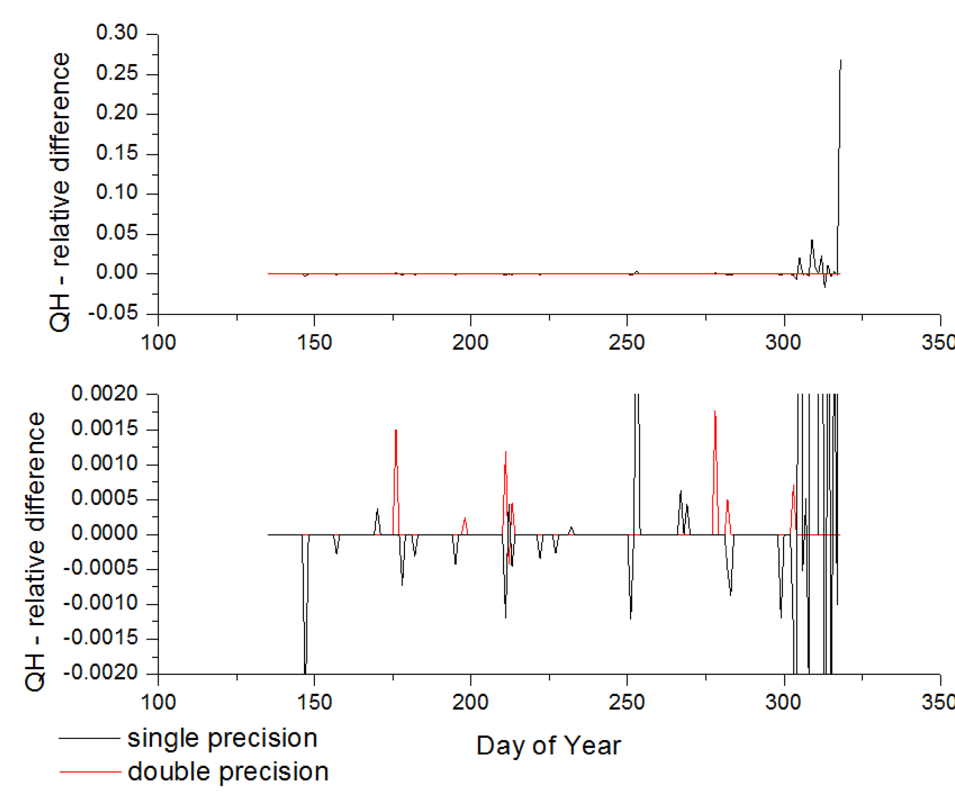


Figure 3: Errors as a result of double precision and single precision.