UEB Parallel: Distributed Snow Accumulation and Melt Modeling using Parallel Computing

Tseganeh Z. Gichamo [a*], David G. Tarboton [a]

[a] Utah Water Research Laboratory, 8200 Old Main Hill, Utah State University, Logan, UT 84322-8200, USA.
* Corresponding author. Tel.: +1 435-797-3172; fax: +1 435-797-3663
Email addresses: zacctsega@gmail.com (T.Z. Gichamo), dtarb@usu.edu (D.G. Tarboton)

Abstract

The Utah Energy Balance (UEB) model supports gridded simulation of snow processes over a watershed. To enhance computational efficiency, we developed two parallel versions of the model, one using the Message Passing Interface (MPI) and the other using NVIDIA's CUDA code on Graphics Processing Unit (GPU). Evaluation of the speed-up and efficiency of the MPI version shows that the effect of input/output (IO) operations on the parallel model performance increases as the number of processor cores increases. As a result, although the computation kernel scales well with the number of cores, the efficiency of the parallel code as a whole degrades. The performance improves when the number of IO operations is reduced by reading/writing larger data arrays. The CUDA GPU implementation was done without major refactoring of the original UEB code, and tests demonstrated that satisfactory performance could be obtained without a major re-work of the existing UEB code.

Keywords—Utah Energy Balance snowmelt model (UEB); Message Passing Interface (MPI); Compute Unified Device Architecture (CUDA); Graphics Processing Unit (GPU); Parallel IO.

Highlights

- The Utah Energy Balance snowmelt model computational kernel is highly parallelizable
- Input/Output was the major efficiency bottleneck in parallelization
- GPU implementation was achieved without major refactoring of existing UEB code

This is the accepted version of the following article
Gichamo, T. Z. and D. G. Tarboton, (2020), "UEB parallel: Distributed snow accumulation and melt modeling using parallel computing," Environmental Modelling & Software, 125: 104614, https://doi.org/10.1016/j.envsoft.2019.104614.
which has been published in final form at
http://www.sciencedirect.com/science/article/pii/S1364815219304050

Software Availability

Program name: UEB Parallel.

Description: Parallel version of the Utah Energy Balance snowmelt model (UEB).

Platform: Platform independent. Tested on Microsoft Windows & Linux (CentOS 6.x).

Source language: C++ / CUDA.

Cost/License: Free / Open source, GNU General Public License.

Developers: Tarboton research group, Utah State University.

Availability: http://github.com/dtarb/ueb

1. Introduction

Hydrological models are used to predict environmental flow of water under diverse drivers of change that are complex and heterogeneous. One of the prime motivators of current hydrological research is the need to understand and quantify the possible impacts on water resources of changes in climate, land cover, land use, population and urbanization (Fowler et al., 2007). Such studies may require modeling of the hydrologic processes at various scales, ranging from headwater watersheds to river basins scales. As the terrestrial water cycle is affected by its interactions with atmospheric and oceanic processes, hydrological models at river basin or global scale may also need to consider the various pathways of water in the global cycle and magnitudes of feed-backs between different layers/components of the cycle (Levine and Salvucci, 1999; Maxwell et al., 2014; Paniconi and Putti, 2015). A few years ago, Wood et al. (2011) made a call for "Hyperresolution global land surface modeling" to sufficiently resolve local processes in a model of global or continental scale.

This task of modeling the hydrologic cycle at large scale with sufficient resolution of individual processes poses multiple challenges. One of these challenges is the desire to use high-performance computing (HPC) resources to reduce computational time or

increase the level of detail (and hence complexity) at which these problems are investigated. On the other hand, the availability of HPC resources is increasing. This, coupled with the recognition of the scientific needs for undertaking large scale hydrologic simulation, has led to development of simulation models that implement parallel processing technologies. For example, Kollet et al., (2010) present results of a study where an integrated multi-dimensional modeling problem with a number of unknowns in the order of 10^9 was solved within a feasible simulation time. The challenge for hydrological modelers is thus shifting from the lack of computing resources to reconfiguring their modeling software to be able to take advantage of these new resources.

It should be noted here that parallel programming in hydrological and environmental modeling is not a new opportunity or issue (e.g., Paglieri et al., 1997; Rao, 2004). However, in the past two decades a strong argument has been made that the basic approach to software development should incorporate concurrency (multi-processes) programming because of the "power wall," i.e., the upper limit imposed on the clock speed of single core due to overheating of high-frequency cores and other efficiency/optimization considerations (Brodtkorb et al., 2013; Sutter, 2005; Sutter and Larus, 2005). Concurrency programing has also been spurred by the definition of standard programing interfaces that abstract away most of the low level operations and a number of library implementations of these interfaces, thereby freeing a research programmer to focus on domain-specific modeling issues (e.g., MPI http://www.mpi-forum.org/, OpenMP http://openmp.org/wp/).

Given the desire to apply more physically based, distributed, high-resolution

hydrologic models, and given the opportunities offered by the parallel programing standards and libraries, the question has then become what method to choose for a given model and what factors affect efficient scaling of the modeling code. In this study, we evaluated parallel processing implementations of the Utah Energy Balance (UEB) snow accumulation and melt model. We evaluated two implementations: one using the MPICH2 library of the Message Passing Interface (MPI) specification (Gropp et al., 2005), and the other using NVIDIA's Compute Unified Device Architecture (CUDA) code on Graphics Processing Units (GPUs) (Nickolls et al., 2008). The MPI is a distributed memory programming approach that promises good efficiency for the distributed UEB model that requires independent data for different model grid cells. On the other hand, the CUDA code, with its compatibility to C++, enhances the accessibility of general-purpose GPUs that have ability to handle compute-intensive tasks.

The computational performance of the parallel codes was evaluated using simulations of the Logan River Watershed, Utah, for a period of five years. For the implementation with MPI, we evaluated the speed-up and efficiency of the code with increasing number of processor cores and compared the speed-up with the ideal speed-up computed based on Amdahl's law (Amdahl, 1967, 2007). With regard to the application of GPUs, Neal et al. (2010) had found earlier that, even though their GPU code was faster and more efficient than their MPI implementation, the development time it required was prohibitive. In contrast, Tristram et al. (2014) reported that not only were GPUs more cost efficient for their application, but also achieving satisfactory speed-up with GPUs did not require major refactoring of their existing code. For the CUDA implementation of the UEB code in this study, we also evaluated if satisfactory performance could be

achieved by the GPU code without major refactoring of the code.

This paper is organized as follows. In the next section, a brief discussion of factors to consider in parallel programing based on review of literature is given (focusing on hydrologic models). In Section 3, Methods, we describe the UEB model, the algorithms for the parallel implementations, the modeling case study to test the performance of the parallel codes, and the performance metrics. Results from the performance tests and discussion are given in Section 4, followed by conclusions in Section 5.

2. Factors to Consider in Parallel Programing

The choice of a particular parallel programing approach may depend on a number of factors including familiarity with the programing interface, ease of adaptation of existing serial code to a parallel version, and the data and memory configuration of the problem being modeled. Neal et al., (2010) investigated the application for 2D flood inundation modeling of three of the commonly used programming methods: shared memory Open Multi-Processing (OpenMP), distributed memory Message Passing Interface (MPI), and Graphics Processing Unit (GPU). They tested the three approaches with respect to applicability to a given problem solution, parallel code efficiency achieved, and required implementation effort (development time). They concluded that the MPI approach was the most suitable compromise between the efficiency achieved and programming complexity involved. They found that, even though the GPU code was the fastest and most efficient, the development time it required was prohibitive.

Another important factor in parallel programming of simulation models is domain/data decomposition among processes. Data partitioning schemes often try to

address the issue of load balancing between multiple processes. A good example is a 2D flood inundation model where some of the grid cells in the flood plain remain dry for part of the model run time, resulting in some idle process time. Data partitioning schemes should strive to minimize such idle times (Brodtkorb et al., 2012; Sanders et al., 2010). With respect to hydrological models, domain decomposition is related to the flow dependencies (upstream-to-downstream) between computational sub-domains, where the computational sub-domain can be a Representative Elementary Watershed (REW), Hydrologic Response Unit (HRU), a structured or unstructured grid cell, or a river reach. In addition, load imbalance may arise from the spatial variability of processes considered such as areas covered with snow versus those with no snow, upstream hillslope versus riparian area or river channel, presence and type of vegetation, etc. Some of the approaches used in recent research are described below.

The first one is the use of multiple layers in which simulation units (grids) are divided based on their degree of dependency on upstream units. Accordingly, units that do not depend on other units are placed at highest priority layer, and units that depend only on a single unit are placed in the second highest priority layer, and so on (Liu et al., 2014). Another approach involves dynamic (run-time) allocation of a data partition to an available idle process once the partition no longer depends on upstream processes (Li et al., 2011). Dividing a 2D model domain into strips where the only inter-process communications are at the boundaries between two adjacent sub-domains is another approach (Tarboton et al., 2009c; Tesfa et al., 2011). A different approach, particularly useful for a model with a tightly coupled set of processes, is collecting all the governing equations into a global system of equations which are solved by a matrix solver (Qu and

Duffy, 2007). Such a matrix solver may divide the global matrix into sub-matrices that are mapped onto multiple processes. The examples above do not form an exhaustive list; however, they indicate that, generally, the choice of a given domain decomposition would be dictated by the specific modeling problem (Small et al., 2013). Presently, a researcher or a modeler has to consider their own problem domain and decide whether to use a method similar to those listed above or develop their own. The ability to automatically choose from certain domain decomposition methods, given a problem domain type, is a potential area of future study

The extent to which a parallel program's performance increases with increased availability of computing resources (e.g., number of processor cores) depends primarily on the fraction of code that must be executed in serial. This is the essence of Amdahl's law (Amdahl, 1967, 2007) based on which the maximum possible speed-up for a given problem is computed as a function of the number of cores and the serial fraction of the code. The effect of the serial fraction of the code becomes more important with increasing parallelization. Yavits et al., (2014) review a number of research works that dealt with the effect of data transfer between the serial and parallel portions of the code and the inter-process communications on the maximum speed-up determined from Amdahl's law. They provide models that revise Amdahl's law, incorporating terms that represent arithmetic intensity—the ratio of total compute operations to data transfer computations, data transfer synchronization between the serial and parallel portions of the program, and inter-process communication and synchronization. Accordingly, the amount of inter-process communication and/or serial to parallel data synchronization in a program might inhibit its suitability for extensive parallelization. For problems with

relatively high inter-process communication and data synchronization, they suggest using fewer large cores rather than many small cores.

Finally, the cost efficiency of the parallel method will have to be considered. As indicated earlier, Neal et al., (2010) concluded that the developer time for programming using graphics cards was prohibitive in their modeling case. However, total cost should also include the price of the computing hardware units and operating costs. Tristram et al., (2014) reported results from a parallel hydrologic uncertainty model with multiple ensembles using GPUs. They compared the CPU and GPU performances with respect to speed-up, the cost of processors, and the cost of power usage, and found GPUs to be more cost efficient for their application. Regarding programmer's time, they showed that to achieve satisfactory speed-up with GPUs, major refactoring of their existing code was not necessary. In addition, the performance was further enhanced with an optimization involving memory access configuration. The general purpose programming toolkits such as CUDA (https://developer.nvidia.com/cuda-toolkit) and OpenCL (https://www.khronos.org/opencl/) coupled with the cheap graphics cards on commodity computers make GPU programming more accessible to scientific research programmers (Garland et al., 2008). However, realizing the full benefits still requires learning efficient program organization and optimizations such as latency hiding by overlapping computation with Input Output (IO) operations, wise management of register and caches, and memory layout configuration (De La Asunción et al., 2012; Tristram et al., 2014) which requires more effort and time (Brodtkorb et al., 2013).

3. Methods

3.1 *Utah Energy Balance (UEB) Snowmelt Model*

The Utah Energy Balance (UEB) snowmelt model tracks the accumulation and ablation of a single snow layer at the ground surface by energy and mass balance computations (Tarboton and Luce, 1996). In forested watersheds, the model accounts for canopy snow interception, partitioning of incoming solar and atmospheric radiation through the canopy layer, and turbulent fluxes of latent and sensible heat within and below the canopy layer (Mahat and Tarboton, 2012, 2013; Mahat et al., 2013). The snow surface temperature is computed using the modified Force-Restore method that characterizes the conduction of heat into the snowpack as a function of the temperature gradient between the snow surface and the average temperature of the snowpack, and by taking into account the temperature profile of the snowpack in the past 24 hours (Luce and Tarboton, 2010; You et al., 2014). In addition, glacier melt processes were modeled with UEB (Sen Gupta and Tarboton, 2013). The model equations and further descriptions can be found in previous publications (Mahat and Tarboton, 2012; Mahat et al., 2013; Tarboton and Luce, 1996).

UEB is a point model in that the equations describing the state-flux relationships are applicable for a model element with uniform (or representative) values of terrain characteristics (slope, aspect, etc.), canopy variables, and meteorological forcing. For spatially distributed modeling, earlier research explored the use of depletion curves to deal with sub-watershed variability (Luce et al., 1999). Work by Sen Gupta and Tarboton (2013) configured the model to be run as fully distributed using Cartesian grids. In the gridded model, the model computations are carried out separately on individual grid cells with the only interaction between grid cells occurring when aggregating outputs from

watersheds (or sub-watersheds). This configuration makes UEB amenable to parallel computation with domain decomposition that is constrained only by the aggregation operations.

The UEB model is driven by air temperature, precipitation, radiation, humidity, and wind speed. The Network Common Data Form (NetCDF), a data format that enables storage and manipulation of multi-dimensional arrays (http://www.unidata.ucar.edu/) is used as the input/output format for UEB. NetCDF includes a set of libraries and tools that enable array-oriented data access with advantages that include concurrent access (multiple readers), platform independence, efficient sub-setting, and data appending (i.e., additional data are added to a file without rewriting it or copying the entire contents). A NetCDF file is self-contained as the metadata to describe the contents of the data and other ancillary information are stored within the file (Rew et al., 2014). A benefit of using NetCDF is that many array-oriented datasets come in some form of gridded binary format compatible with NetCDF. The choice to use NetCDF as input/output format for UEB was driven by the vision to couple the model with data sources and other models that follow the same standard.

3.2 *UEB Parallel*

The flow chart for the parallel version of the UEB model with MPI implementation developed in this work is shown in Figure 1. Many of the tasks, including the weather forcing IO operations, are contained in a code block named “Run UEB in the grid cell for all time steps.” This step loops through all grid cells and runs the model for all time steps of the simulation period for a given grid cell. The operations at each grid cell are carried out independently from the other grid cells. This block of code takes more than 99% the total simulation time. This code block was, therefore, parallelized with MPI

in which the active grid cells, excluding the no-compute cells, were divided into the total number of processes. When the total number of grid cells is not evenly divisible by the number of processes, some processes may be allocated an extra grid, leaving the others with an idle time of one grid cell computation. For large sized problems, this idle time is expected to be small.

At the end of the simulation, the processes collectively write results to output NetCDF files, one output file for each output variable. This NetCDF write requires synchronization among the processes as they access the NetCDF file simultaneously. One factor we evaluated in this study was the extent to which the IO operations can be parallelized, and the degree to which the synchronized access to data in NetCDF files by multiple processes can affect the overall performance of the parallel codes. For this implementation of the UEB model with MPI, we compared an IO reading/writing in which multiple arrays of data were handled at once to the 'looping through the grids' approach mentioned above that accesses a single array at a time (multiple arrays versus one-array-at-a-time).

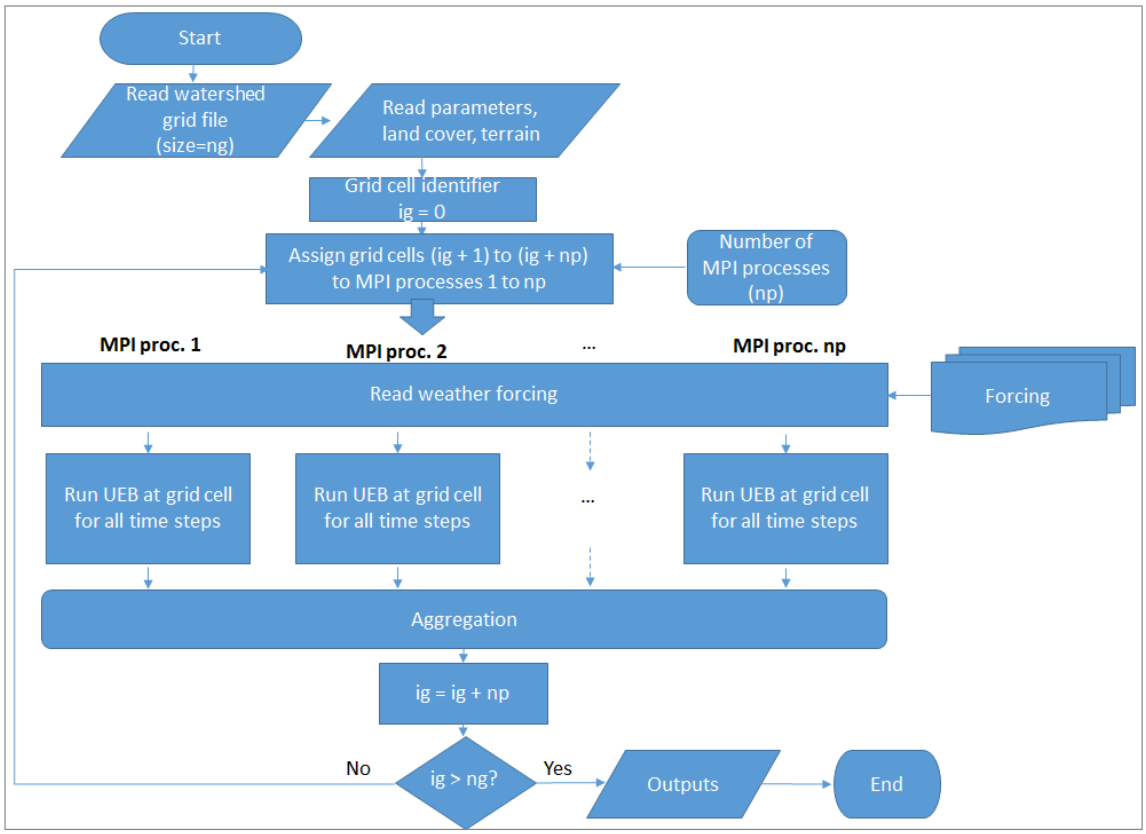

Figure 1: Flow chart for the parallel version of the UEB model (UEB Parallel) with MPI.

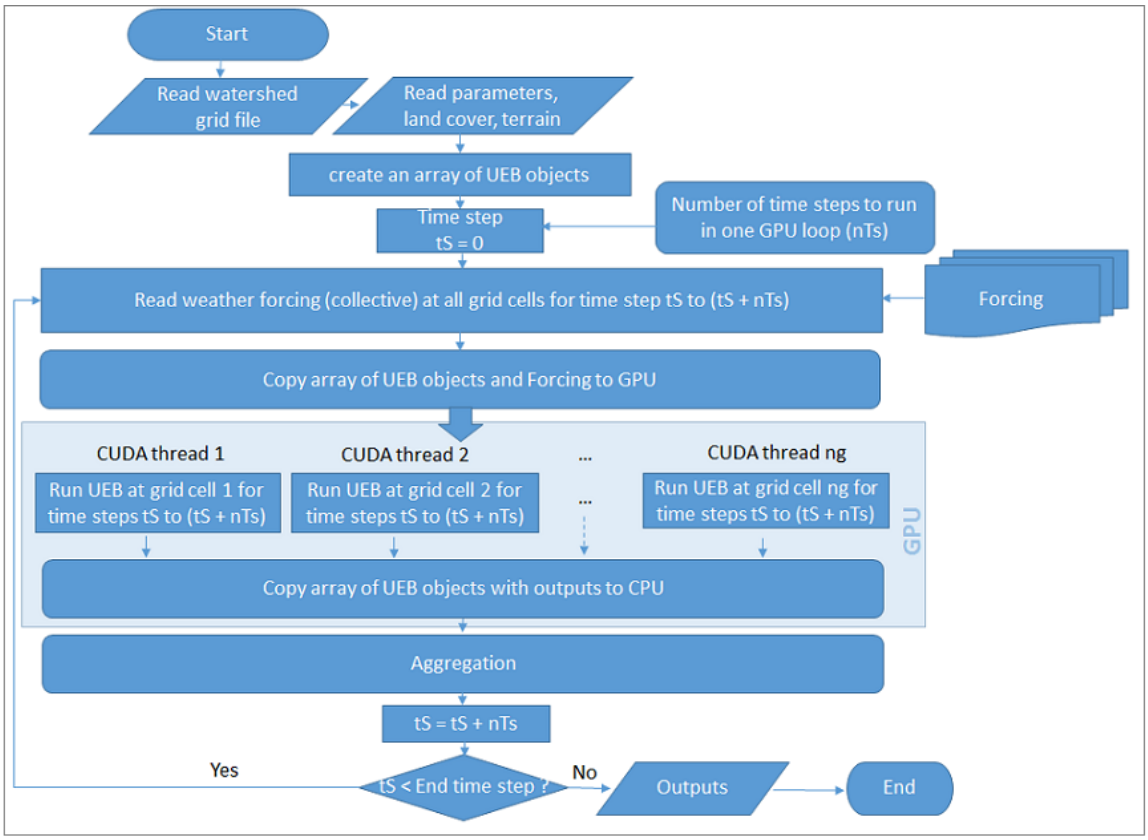

Figure 2: Flow chart for the parallel version of UEB model (UEB Parallel) with GPU.

The flow chart for the model implementation with GPU is shown in Figure 2. In this case, a UEB class was defined first as a class that encapsulates watershed, terrain, and canopy variables as data members and the UEB simulation functions as methods (member functions). This results in an array of active grid cells consisting of an array of UEB class instances (objects). This configuration was chosen because it was easier to copy arrays of objects/structures between the host/CPU and device/GPU nodes than copying individual variable arrays. The CPU (host) allocates GPU (device) memory and copies the data to the device. All the snow accumulation and ablation computations are carried out by the GPU functions, i.e., kernels, and outputs are copied back to the CPU node that writes them into NetCDF output files. In this case, in contrast to the MPI

implementation, only a few CPU nodes carry out the IO operations. We implemented the CUDA code in such a way that the changes to the UEB MPI code were kept minimal, as can be seen from comparison of Figures 1 and 2. The total number of code lines in the MPI version was 4930. Of these, 572 lines (11%) were modified in the GPU version which generally involved edits to ensure compatibility with CUDA devices, while 303 lines (6%) were removed and 570 new lines (11%) were added. The objective here was to evaluate if the observation by Tristram et al., (2014) that implementing GPU code with satisfactory performance may not necessarily require major re-work on an existing programming code also applies to UEB.

An important difference between the GPU implementation and the one with MPI is that in the GPU case the time loop is outside of the grid cell loop, i.e., simulations at all grid cells are carried out for a few time steps (typically a few days) before advancing to the next step. This way, the highly parallel nature of individual grid cell computations is taken advantage of without having to copy all the weather forcing data to the device at once. Copying weather forcing data for all time steps at once would require large memory at the device that could be difficult to allocate.

The flow chart in Figure 2 was drawn assuming that the total number of GPU threads that can be scheduled equals or exceeds the total number of model grid cells, which was the case here. This assumption is unlikely to be an issue for most modeling cases like the one tested in this paper because the upper limit on the number of threads per device is quite large. Where the number of grid cells exceeds the number of CUDA threads available, a strategy similar to the assignment of grid cells to threads used in the MPI approach of Figure 1, would be needed. In the MPI implementation, the active grid

cells were evenly distributed among MPI processes, when that was possible. In both versions of the UEB model, there is spatial variability in grid cell properties, such as vegetation covered versus no vegetation and snow covered versus no snow, that affects the number of iterations to converge to a particular solution in a given cell, and that introduces some variability in the total compute time among processes/threads. This is considered to be part of simulation overhead and will diminish the efficiency in both parallel implementations, compared to the ideal case of uniform grid cells. Two other sources of overhead are reading configuration files at the beginning of the program and an outputs aggregation step where watershed average or total quantities are computed. The aggregation operations involve inter-process communication where all processes transfer the values of the aggregated variables to the root process. Note that input forcing data reading and output writing are handled by the NetCDF IO and, in this study, are not considered part of the inter-process communication.

3.3 *Test Case Study: The Logan River Watershed*

We used simulations of snow accumulation and melt for five years—October 1, 2007- September 30, 2012—in the Logan River watershed, Utah, to evaluate the performance of the parallel codes. The Logan River watershed is a 554 km^2 watershed located in the Bear River Range of Utah in the western U.S. The watershed elevation ranges from 1497 to 3025 m with mean elevation of 2271 m. Most of the upland watershed is covered by shrubs, grass, and forest and is primarily used for grazing while the lower reaches of the river support irrigation. The average precipitation varies between 450 – 1500 mm per year with most of it in the form of snow. The river peaks late in the spring from snowmelt. Figure 3 shows the location map of the Logan River watershed and its digital elevation map.

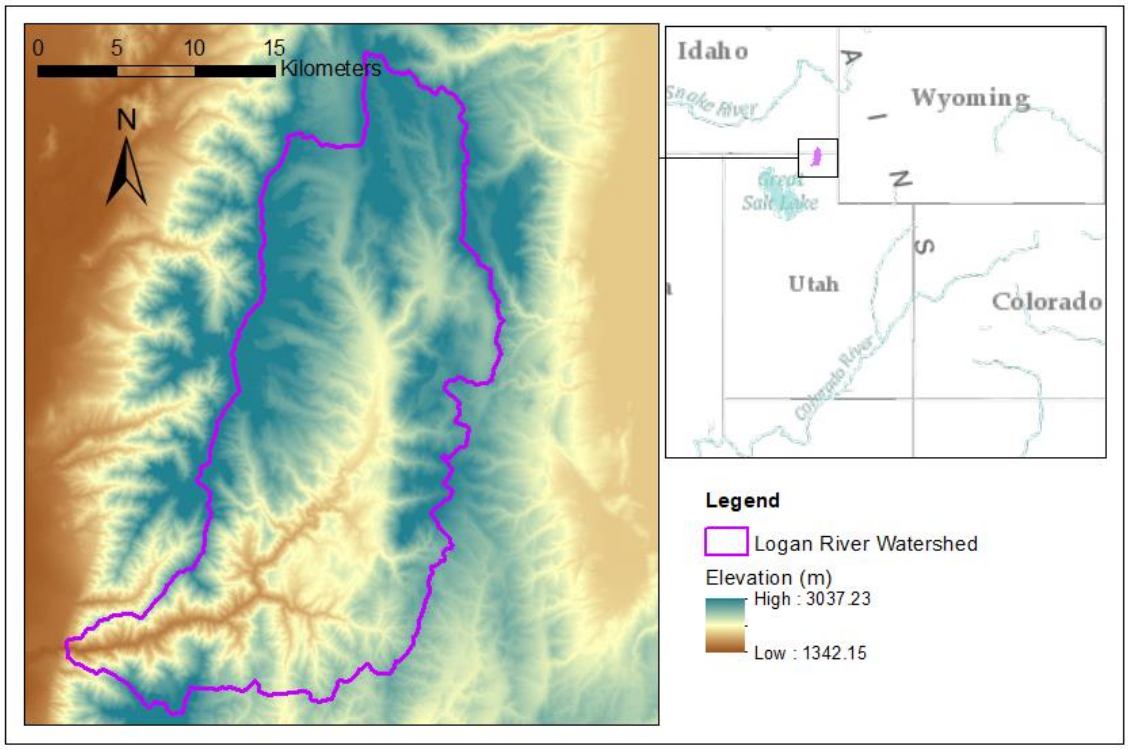

Figure 3: Study area - Location map of the Logan River watershed and its elevation (DEM).

The input data were setup as follows. The watershed domain was delineated from the 30m USGS National Elevation Dataset Digital Elevation Model (NED DEM) using the terrain analysis software TauDEM (Tarboton et al., 2009a; Tarboton, 2015; Tesfa et al., 2011). Terrain variables slope and aspect were calculated from the DEM in ESRI's ArcGIS software (www.esri.com). The canopy leaf area index (LAI) were obtained from the Moderate Resolution Imaging Spectroradiometer (MODIS) product MOD15A2. The other vegetation variables including canopy height and fraction of the grid cell covered by vegetation were determined using the National Land Cover Database 2011 (NLCD 2011) (Homer et al., 2015). Weather forcing data were obtained from SNOTEL stations in and nearby the watershed. These data were gridded with the bilinear interpolation method to grid size of 120m, the model resolution, and downscaled (adjusted for

elevation) using a methodology descripted by Sen Gupta and Tarboton (2016). The focus of this study was evaluation of the performance of the parallel codes; thus, no calibration or validation of model parameters was performed apart from verification to make sure the parallel models' outputs are consistent with those of the original serial version. The UEB model input data for the Logan River Watershed used for testing the parallel models in this paper are provided as a HydroShare resource (Gichamo and Tarboton, 2019).

3.4 *Description of Computing Resources*

Both versions of the UEB Parallel model were written as a platform independent code with the C++ programming language. The GPU version uses CUDA for the device codes. Performance tests were made in a Linux clusters with up to 128 cores for the MPI version, while the GPU code was tested on a Linux machine with a CPU node linked to GPU node. Both these implementations were compared to a simulation with a single MPI process on a desktop PC. The specifications of the computing resources are as follows:

- Linux cluster for MPI: Intel$^{(R)}$ Xeon$^{(R)}$ CPU E5-4620 0 @ 2.20GHz (maximum Turbo Boost frequency 2.60 GHz) with 8 cores per CPU and 4 CPUs per node for a total of 32 cores per node (64 threads per node with hyper-threading enabled) and up to 1 TB memory per node.
- Linux nodes with GPU: Intel$^{(R)}$ E5-2670 (Sandy Bridge) CPU + NVIDIA K20x GPU with 2688 processor cores and processor core clock of 732 MHz.
- Desktop PC with Intel$^{(R)}$ Core™ i7-3770: 4 cores with 3.40 GHz (maximum Turbo Boost frequency 3.90 GHz) and hyper-threading; 32 GB RAM.

3.5 *Performance Metrics*

Total simulation time, speed-up, and efficiency were used to test the performance of the parallel codes. Speed-up is computed as the ratio of the total simulation time by a

single core to that by multiple cores (P number of cores). Speed-up varies with the total number of cores. Efficiency is the speed-up divided by the number of cores (Eager et al., 1989). For a given unit of work, the efficiency may change with the number of cores due to an increased overhead, inter-process communication, and/or increased synchronization. According to Amdahl's law (Amdahl, 1967, 2007), the maximum possible speed-up (ideal speed-up) is constrained by the fraction of the code that has to be executed serially, and hence is executed by all cores. For the UEB model, this was assumed to be the code outside of the "Loop through active grid cells" portion of the code described above. In reality, however, some of the codes inside the loop could also contribute to it. These are generally considered 'overhead,' and this is partly why the "ideal speed-up" is called "ideal"—because, in practice, the overheads further reduce the speed-up.

Note that in an MPI implementation, multiple processes can be executed on a single core. In this study, the tests were carried out to evaluate the performance (speed-up and efficiency) as a function of the number of processor cores; hence, the number of MPI processes equals the number of cores assigned.

Equations 1 - 3 below give Speed-up, Efficiency, and representation of Amdahl's law, given a number of cores P. Equations 1 and 3 represent the ratio of two similar units, often computed as the simulation time per one core divided by simulation time per multiple cores. Both equations (1 & 3) apply to processor cores that are of uniform type, of similar core size and speed. The units of the numerator and the denominator in equation 3 can thus be considered as that of time per unit core.

$$S_p = \frac{T_1}{T_p} \qquad (1)$$

$$E_p = \frac{S_p}{P} \qquad (2)$$

$$S_{pmax} = \frac{1}{f_s + \frac{1 - f_s}{P}} \qquad (3)$$

Where: Sp = Speed-up by P number of cores

T_1 = Execution time for a single core

T_p = Execution time for P number of cores

E_p = Efficiency for P number of cores

P = number of cores

S_{pmax} = maximum speed-up based on Amdahl's law

fs = Fraction of code that can only be executed serially.

In the case of UEB with MPI, with much of the code in the parallelizable "Loop through active grid cells" block as described earlier, speed-up approaching the number of cores (P) is to be expected given little model setup overhead, inter-process communications that occurs only at the output aggregation step, and few blocking operations that force processes to wait for each other. Some degradation from maximum efficiency is expected due to variability in the processing time for each cell. An increase in the workload by increasing the size of the watershed and/or the duration of simulation would primarily increase the tasks inside the loop and is expected to increase the speed-up per total number of cores (i.e., efficiency).

When computing the ideal speed-up and efficiency using Amdahl's law above, we considered the inter-process communications time to be part of the fraction of the code executed in serial and assumed that IO operations were parallelizable. An alternative analysis is to consider the inter-process communications and IO operations separately. In

such a situation, we have UEB code sections for model setup (run in serial), inter-process communication (unlikely to be parallelizable), IO operations (possibly parallelizable), and model computational kernel (expected to be highly parallelizable). To demonstrate the effect of IO operations on the parallel performance of UEB, we use Equation 4, which is a slightly modified equation from Yavits et al., (2014 p. 7 Eqn 13) for symmetric compute cores of uniform compute ability. The modification in Equation 4 here from Equation 13 of Yavits et al., (2014) is that we assume the inter-process communication to be negligible in UEB, hence the term for inter-process communication was dropped.

$$S_{pmaxr} = \frac{1}{(1 - f_p) + \frac{f_p}{P} + \frac{T_s}{T_1}} \qquad (4)$$

Where: f_p = Fraction of code that can be executed in parallel

P = number of cores

T_1 = Execution time for a single core

T_s = IO synchronization time (Sequential-to-parallel data synchronization time in Yavits et al., (2014))

S_{pmaxr} = maximum speed-up based on Amdahl's law, revised to account for IO.

The term $\frac{T_s}{T_1}$ in equation (4) is referred to as "Synchronization Intensity" in Yavits et al., (2014). Here, it accounts for the time spent reading input forcing data and writing UEB outputs from or to NetCDF files. The operations of file access and partitioning of data to the respective processes (synchronizing) are considered separate from the inter-process communication in this study. The effect of Synchronization Intensity is to decrease the ideal speed-up, and its importance increases for larger number of parallel cores.

4. Results and Discussion

Figure 4 shows plots of the total simulation time, speed-up, efficiency, and ratio of IO time to total simulation time as a function of the number of MPI processes (same as number of cores). These values were computed after running the model twice for each case (number of processes) and taking the average of the two simulation times. It can be seen that the slope of the speed-up curve decreases with increase in the number of processes. Figure 4b also compares the speed-up against the maximum speed-up based on Amdahl's law. Excluding the code inside the "loop through the grids" which also includes IO, the remaining part of the code takes less than 0.01% of the total simulation time. This initially suggested a highly parallelizable code, which led to expectation of speed-up close to the ideal speed-up. The actual speed-up curve, however, is much lower than the ideal speed-up curve, and its slope decreases with increasing number of processes.

Figures 4b and 4c also include speed-up and efficiency plots for the model computational kernel, i.e., excluding IO operations. As can be seen from the figures, the speed-up of the computational kernel is closer to the ideal speed-up. The reason for the poor performance of the total code compared to the computational kernel is that the IO is not as readily parallelizable as the rest of the code. For the serial version of the program, the IO takes about 1.7 % of the total execution time of the code. Because this fraction of code ended up not being parallelized, contrary to the assumption in equation (3), it affects the performance of the whole model with increasing importance as the number of processes is increased, as shown in Figure 4d.

The deviation of the computational kernel speed-up from the ideal line can be

caused by any of, or a combination of, the factors that were considered "overhead" during the design in Section 2. In addition, after the total active grid cells were evenly divided between the processes, a few processes would be allocated one additional grid cell to simulate. This means some processes may have to stay idle for a duration of one grid cell computation. The time it takes for a full computation of one grid cell, on average, is about 2.5 seconds.

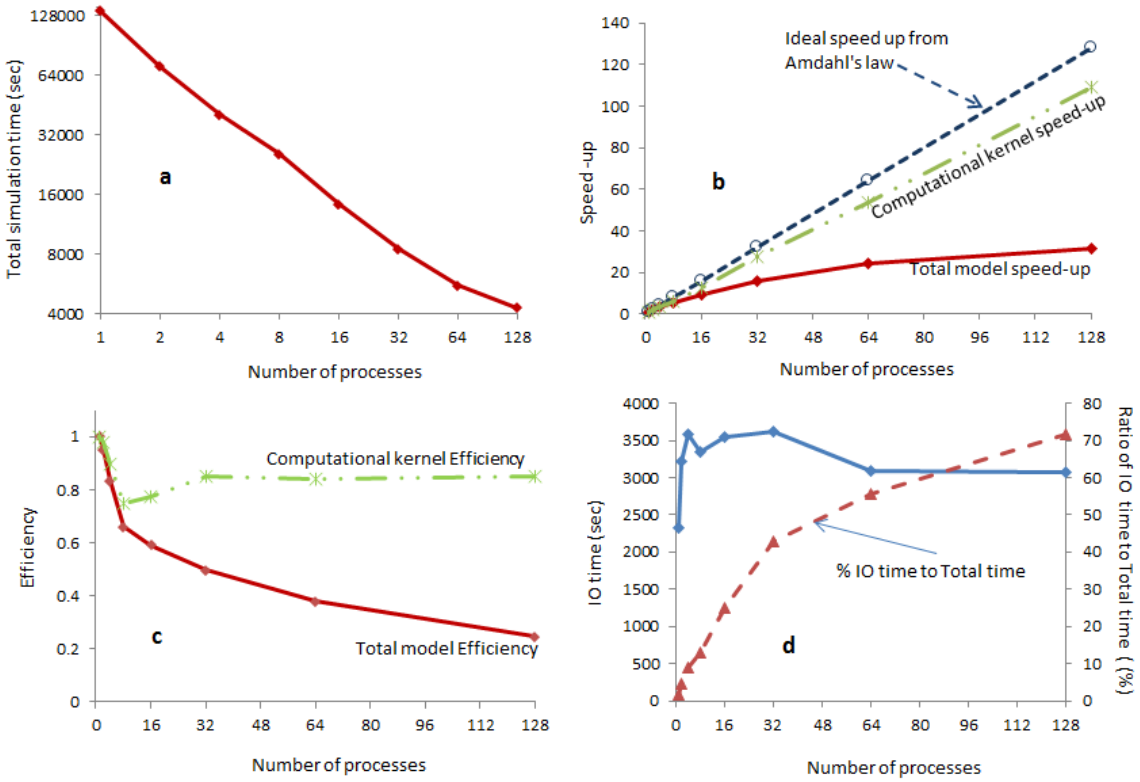

Figure 4: Total simulation time (a), speed-up (b), efficiency (c), and ratio of IO time to total time (d) vs the number of processes (same as number of cores) for the MPI implementation.

Bridging the gap between the good scaling of the computational kernel and the poor scaling of the total model run caused by poor IO scaling is important as the IO starts to become dominant with increasing parallelism so much so that increasing the number of

processor cores beyond 64 may not be justifiably useful. While the parallel NetCDF4, which is based on HDF5's MPI IO feature, would enable synchronized file access by multiple processes, efficient IO scaling requires coupling it with some file access strategies that take advantage of the synchronization (Chilan et al., 2006). Figure 5 shows the results for a modified UEB IO reading/writing in which multiple arrays of data were handled at once instead of the 'looping through the grids' approach used in Figure 4. As can be seen in Figure 5, the performance improves appreciably, particularly for the higher number of cores. For 64 cores, the speed-up increases from 24 to 42 while the efficiency increases from about 0.38 to 0.67. Similarly, for 128 cores, the speed-up and efficiency increase from 31 to 63 and 0.25 to 0.49 respectively. This approach would reduce the total file access operations; however, it may require larger memory per core to be effective.

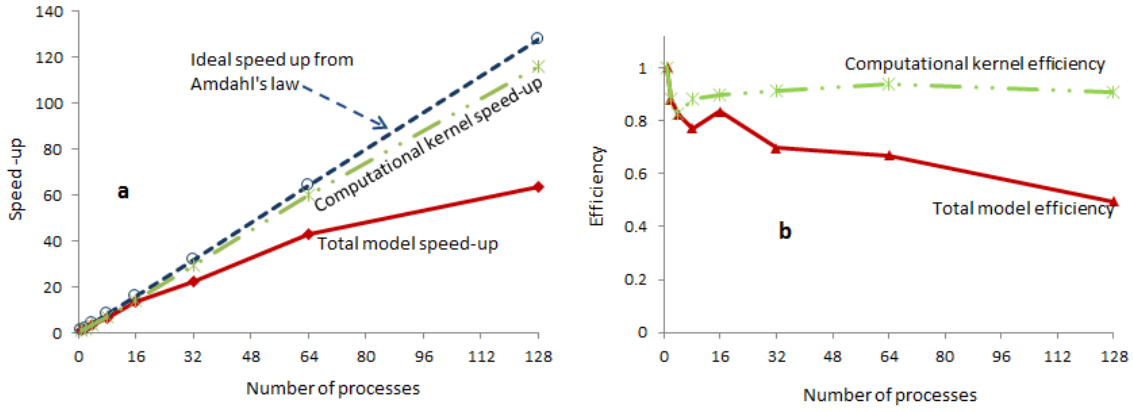

Figure 5: Speed-up (a) and efficiency (b) vs the number of processes (number of cores) for the MPI implementation with reduced IO operations.

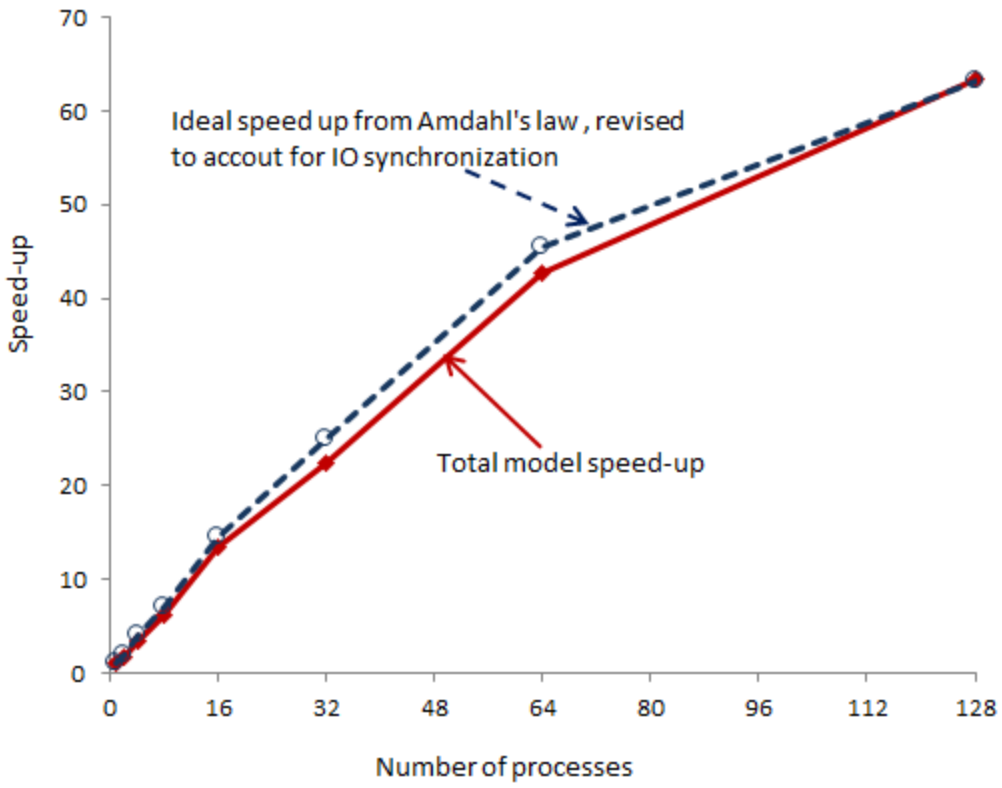

Figure 6: Speed-up and ideal speed-up from Amdahl's law revised to account for IO operation.

Figure 6 is a re-plot of Figure 5a with the ideal speed-up revised according to equation (4) to account for the effect of IO operations. This figure indicates that any further performance improvement would only come from better IO parallelization.

The model run times for the GPU version with CUDA code are shown in Table 1 together with run times on desktop PC with one process. The run time on desktop PC with one process is equivalent to that of a serial code on desktop PC. Table 1 reports the speed-up with the CUDA GPU relative to the serial code on desktop PC. We do not have multiple simulations in GPU, so there are no plots for GPU comparable to those for MPI. In the GPU case, the IO operations were carried out by the CPU (host) while the numerical simulations were performed by the GPU (device). In addition to IO operations,

data synchronization between the host and device is required. Table 1 also includes run times from the MPI implementations in Linux cluster with one and 64 processes and their respective speed-ups over the serial code on desktop PC.

The GPU implementation presents a slightly better speed-up when compared to the MPI code executed on the CPU cluster of 64 processes (Table 1). Our result leads us to a similar observation by Tristram et al., (2014) that, implementing a GPU code with a performance comparable to other parallel methods may not necessarily require a major re-work of an existing programming code. In our case, the computational kernel and much of the data partitioning code remained the same, as it was written in C++, which is compatible to the CUDA specification (NVIDIA, 2015). Given the fact that GPUs provide superior performance per total resource cost (price of hardware and power usage, see e.g., http://www.fmslib.com/mkt/gpus.html), and considering the comparatively short developer time for some existing codes like UEB, GPUs appears to be a worthwhile alternative to the MPI implementation.

Table 1 Run times (seconds) and speed-up for different computing resources

Computing Resource	Model setting	Input reading	Output Writing	Computation	Host-Device data copy	Total *	Speed-up **
1CPU + 1GPU on Linux cluster	1.2	168.6	43.3	2321.5	109.6	2644.2	10.1
64 MPI processes on CPU Linux cluster	210.2	157.8	595.7	2354.9	NA	3318.7	8.0
1 MPI process on CPU Linux cluster	1.0	21.1	134.4	141730.0	NA	141886	0.2
1 process on desktop PC (CPU)	0.1	321.4	451.7	25812.3	NA	26586	1.0

* Including overhead
**Compared to one process on desktop PC CPU

The speed-up shown in the last column of Table 1 is computed in a different way

from the speed-ups and efficiencies reported in Figures 4 to 6. While the numbers in Figures 4 – 6 help evaluate the performance of the parallel code with increasing cores, the speed-up values in Table 1 are measures of the performance enhancement achieved by implementing concurrency programming using clusters of CPU and/or GPU resources, compared to a serial code on a desktop PC (Brodtkorb et al., 2013). Table 1 shows that our parallel implementations achieve speed-ups of 8 – 10 times over a serial code on a desktop PC. This represents the actual enhancement to hydrologic research due to improved access to high performance computation resources. These results report the actual reduction in time spent simulating a UEB model instance using the Linux cluster or the GPU resources rather than the desktop PC. Such reduction in modeling time facilitates the evaluation and adoption of physically based models like UEB in operational settings such as streamflow forecasting where computational time can be critical or for studies of effects of climate change which require large scale simulations.

The first column of Table 1 for the "model setting", which involves reading the model domain, terrain, and parameter files by all processes, serially, has a very large value for MPI with 64 processes. This large overhead was unexpected and contradicted our assumption earlier that the overhead would be negligible with an increase in the number of processes. This overhead was likely caused by competitive file reading by multiple processes. Having a single (root) process read the model setting files and then broadcast the values to all processes reduces the model setting time to 1 second. However, the reduction in model setting time does not have a significant effect on the total simulation time, and it does not change the conclusion about the overall speed-up.

An anomaly in the table 1 results is the large computation and hence large total

time for a single MPI process on the Linux cluster. We would have anticipated this to be about the same as the desktop PC, but it is about five times slower. This difference cannot be explained by the difference in processor core speed between the two resources alone. We do not know what the impacts of different amounts of memory are and whether there is an overhead involved in addressing the memory available on the nodes of the Linux cluster or how comparable the memory access times are across these systems. This difference remains a result that we do not fully understand. It is partly because of this anomaly that we chose to report speed-up relative to the desktop PC which, although it has different hardware, provides a more realistic and honest reporting of the speed up we have demonstrated that a researcher will experience. Further investigation is required to find out the causes for this difference such as the possible introduction of poorly optimized code sections during modification of the serial code to the parallel version and the choice of compiling optimization parameters for different platforms. It is possible that the MPI implementation in this paper, just like the GPU one, can be further optimized to gain even better performance.

Finally, sample outputs of snow water equivalent (SWE) simulated by the two parallel versions of UEB (MPI and CUDA GPU) are shown in Figure 7. This figure demonstrates that the outputs from the two implementations are, for practical purposes, equivalent. This is expected since the data types and the computation logic remain unchanged in both implementations. As stated earlier, no parameter calibration and/or validation by comparison to observations was done given that the objective of this study was assessing computational efficiency of the parallel implementations.

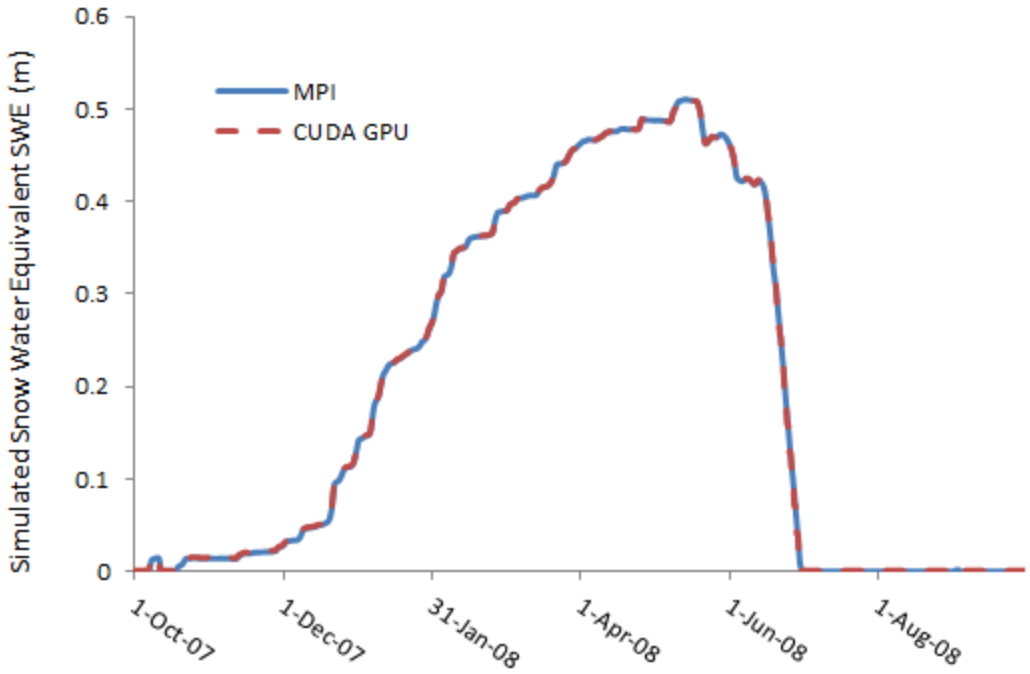

Figure 7: Simulated Snow Water Equivalent (SWE) using the two parallel implementation of UEB.

5. Summary and Conclusions

The implementation and evaluation of parallel processing methods in the Utah Energy Balance (UEB) distributed snow accumulation and melt model is presented in this paper. Two parallel implementations of UEB were evaluated: one using the Message Passing Interface (MPI) and the other using CUDA GPU. Continuous simulation of the Logan River watershed for five years was used to test the performance of the parallel codes, and the speed-up and efficiency as function of number of processor cores and plots of speed-up compared to the ideal speed-up based on Amdahl's law were used as performance metrics of the parallel codes.

For the MPI implementation, results show the importance of input/output (IO) operations in the degradation of efficiency with increase in the number of processes. In

the serial code, IO accounts for about 1.7% of the code run time. However, as this is not reduced by parallelization, its impact becomes more pronounced with increased number of processes. This was verified using the revised form of Amdahl's law (Yavits et al., 2014) that separately accounts for IO operations and inter-process communications. The plot of this revised form of Amdahl's law indicates that further performance increase of the parallel code could only be possible with improved performance of the IO operations. The performance of the MPI implementation improves when utilizing an IO strategy that reduces the number of file accesses by reading and writing multiple arrays of data in one step.

The CUDA GPU implementation achieves slightly better performance with one GPU node when compared with the MPI implementation executed on 64 cores. The GPU implementation was done without major refactoring of the original code, as the computational kernel and much of the data partitioning code remain the same. This shows that, for some models such as UEB, obtaining a CUDA GPU performance comparable to other parallel methods does not necessarily require a major re-work of an existing programming code. Given the fact that GPUs provide superior performance per total resource cost (price of hardware and power usage), this makes it a worthwhile alternative to the MPI implementation.

Overall our parallel implementations help achieve speed-ups of 8 – 10 times over a serial code on a desktop PC. This represents the actual speed up available to hydrologic researchers from use of high performance computing resources instead of a desktop PC.

Most distributed physically based hydrological models are data intensive. This work demonstrates the importance of including IO operations within the parallelizable

code section and using efficient IO handling together with distributed computing resources. Efficient IO scaling requires adopting file read/write strategies that take advantage of parallel file access or separate files for different processes. The approach we used in this paper is a simple one, which involves reading and writing multiple arrays of data in one-step, and it could be limited by the availability of memory per core. There is a need for more advanced IO strategy that also accounts for the available memory.

Finally, the modeling work presented here is only for the case of snow accumulation and melt. Outputs from this model are used as input to runoff and river routing models. We can qualitatively predict that coupling UEB to a runoff model would increase the arithmetic intensity because more computations would be done without significantly increasing the IO volume. Additional inter-process communications would be introduced but would likely be smaller than the added arithmetic operations. Therefore, it would be interesting to extend this study to examine if a better efficiency may be achieved with the coupled model.

Acknowledgments

This work was supported by the National Science Foundation under collaborative grants EPS 1135482 and 1135483. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Compute, storage, support, and other resources from the Division of Research Computing in the Office of Research and Graduate Studies at Utah State University and Advanced Research Computing Center at the University of Wyoming are gratefully acknowledged. We appreciate the anonymous EMS reviewer whose suggestions have improved this

manuscript.

References

Amdahl, G.M., 1967. Validity of the single processor approach to achieving large scale computing capabilities, Proceedings of the April 18-20, 1967, spring joint computer conference. ACM, pp. 483-485.

Amdahl, G.M., 2007. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, Reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, NJ, Apr. 18–20), AFIPS Press, Reston, Va., 1967, pp. 483–485, when Dr. Amdahl was at International Business Machines Corporation, Sunnyvale, California. Solid-State Circuits Society Newsletter, IEEE 12(3) 19-20.

Brodtkorb, A.R., Hagen, T.R., Sætra, M.L., 2013. Graphics processing unit (GPU) programming strategies and trends in GPU computing. Journal of Parallel and Distributed Computing 73(1) 4-13.

Brodtkorb, A.R., Sætra, M.L., Altinakar, M., 2012. Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. Computers & Fluids 55 1-12.

Burnash, R., Singh, V., 1995. The NWS river forecast system-Catchment modeling. Computer models of watershed hydrology. 311-366.

Chilan, C.M., Yang, M., Cheng, A., Arber, L., 2006. Parallel I/O performance study with HDF5, a scientific data package. TeraGrid 2006: Advancing Scientific Discovery.

De La Asunción, M., Mantas, J.M., Castro, M.J., Fernández-Nieto, E.D., 2012. An MPI-CUDA implementation of an improved Roe method for two-layer shallow water systems. Journal of Parallel and Distributed Computing 72(9) 1065-1072.

Eager, D.L., Zahorjan, J., Lazowska, E.D., 1989. Speedup versus efficiency in parallel systems. Computers, IEEE Transactions on 38(3) 408-423.

Fowler, H., Blenkinsop, S., Tebaldi, C., 2007. Linking climate change modelling to impacts studies: recent advances in downscaling techniques for hydrological modelling. International Journal of Climatology 27(12) 1547-1578.

Franz, K.J., Hogue, T.S., Sorooshian, S., 2008. Operational snow modeling: Addressing the challenges of an energy balance model for National Weather Service forecasts. Journal of Hydrology 360(1) 48-66.

Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., Volkov, V., 2008. Parallel computing experiences with CUDA. IEEE micro(4) 13-27.

Gichamo, T. Z., D. G. Tarboton (2019). Logan River Watershed data used for testing parallel implementations of Utah Energy Balance Snowmelt Model, HydroShare, http://www.hydroshare.org/resource/95dfb865cadd4e0da6548d5952d679bc

Gropp, W., Lusk, E., Ashton, D., Balaji, P., Buntinas, D., Butler, R., Chan, A., Goodell, D., Krishna, J., Mercier, G., 2005. MPICH2 user's guide. Mathematics and Computer Science Division-Argonne National Laboratory, Version 0.4.

Homer, C.G., Dewitz, J.A., Yang, L., Jin, S., Danielson, P., Xian, G., Coulston, J., Herold, N.D., Wickham, J., Megown, K., 2015. Completion of the 2011 National Land Cover Database for the conterminous United States-Representing a decade

of land cover change information. Photogrammetric Engineering and Remote Sensing 81(5) 345-354.
Kauffeldt, A., Wetterhall, F., Pappenberger, F., Salamon, P., Thielen, J., 2014. Technical review of large-scale hydrological models for implementation in operational flood forecasting schemes on continental level.
Kollet, S.J., Maxwell, R.M., Woodward, C.S., Smith, S., Vanderborght, J., Vereecken, H., Simmer, C., 2010. Proof of concept of regional scale hydrologic simulations at hydrologic resolution utilizing massively parallel computer resources. Water Resources Research 46(4).
Levine, J.B., Salvucci, G.D., 1999. Equilibrium analysis of groundwater–vadose zone interactions and the resulting spatial distribution of hydrologic fluxes across a Canadian Prairie. Water Resources Research 35(5) 1369-1383.
Li, T., Wang, G., Chen, J., Wang, H., 2011. Dynamic parallelization of hydrological model simulations. Environmental Modelling & Software 26(12) 1736-1746.
Liu, J., Zhu, A., Liu, Y., Zhu, T., Qin, C.-Z., 2014. A layered approach to parallel computing for spatially distributed hydrological modeling. Environmental Modelling & Software 51 221-227.
Luce, C.H., Tarboton, D.G., 2010. Evaluation of alternative formulae for calculation of surface temperature in snowmelt models using frequency analysis of temperature observations. Hydrology and Earth System Sciences 14(3) 535-543.
Luce, C.H., Tarboton, D.G., Cooley, K.R., 1999. Sub-grid parameterization of snow distribution for an energy and mass balance snow cover model. Hydrological Processes 13(12) 1921-1933.
Mahat, V., Tarboton, D.G., 2012. Canopy radiation transmission for an energy balance snowmelt model. Water Resources Research 48(1) W01534.
Mahat, V., Tarboton, D.G., 2013. Representation of canopy snow interception, unloading and melt in a parsimonious snowmelt model. Hydrological Processes n/a-n/a.
Mahat, V., Tarboton, D.G., Molotch, N.P., 2013. Testing above- and below-canopy representations of turbulent fluxes in an energy balance snowmelt model. Water Resources Research 49(2) 1107-1122.
Maxwell, R.M., Putti, M., Meyerhoff, S., Delfs, J.O., Ferguson, I.M., Ivanov, V., Kim, J., Kolditz, O., Kollet, S.J., Kumar, M., 2014. Surface‐subsurface model intercomparison: A first set of benchmark results to diagnose integrated hydrology and feedbacks. Water Resources Research 50(2) 1531-1549.
Neal, J.C., Fewtrell, T.J., Bates, P.D., Wright, N.G., 2010. A comparison of three parallelisation methods for 2D flood inundation models. Environmental Modelling & Software 25(4) 398-411.
Nickolls, J., Buck, I., Garland, M., Skadron, K., 2008. Scalable Parallel Programming with CUDA. Queue 6(2) 40-53.
NVIDIA, 2015. Cuda C Programming Guide.
Paglieri, L., Ambrosi, D., Formaggia, L., Quarteroni, A., Scheinine, A.L., 1997. Parallel computation for shallow water flow: A domain decomposition approach. Parallel computing 23(9) 1261-1277.
Paniconi, C., Putti, M., 2015. Physically based modeling in catchment hydrology at 50: Survey and outlook. Water Resources Research 51(9) 7090-7129.
Qu, Y., Duffy, C.J., 2007. A semidiscrete finite volume formulation for multiprocess

watershed simulation. Water Resources Research 43(8).
Rao, P., 2004. A parallel hydrodynamic model for shallow water equations. Applied mathematics and computation 150(1) 291-302.
Rew, R., Davis, G., Emmerson, S., Davies, H., Hartnett, E., Heimbigner, D., Fisher, W., 2014. NetCDF Documentation (http://www.unidata.ucar.edu/software/netcdf/docs/). Unidata, University Corporation for Atmospheric Research (UCAR) Community Programs (UCP).
Sanders, B.F., Schubert, J.E., Detwiler, R.L., 2010. ParBreZo: A parallel, unstructured grid, Godunov-type, shallow-water code for high-resolution flood inundation modeling at the regional scale. Advances in Water Resources 33(12) 1456-1467.
Sen Gupta, A., Tarboton, D.G., 2013. Using the Utah Energy Balance Snowmelt model to quantify snow and glacier melt in the Himalayan region.
Small, S.J., Jay, L.O., Mantilla, R., Curtu, R., Cunha, L.K., Fonley, M., Krajewski, W.F., 2013. An asynchronous solver for systems of ODEs linked by a directed tree structure. Advances in Water Resources 53 23-32.
Sutter, H., 2005. The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's journal 30(3) 202-210.
Sutter, H., Larus, J., 2005. Software and the concurrency revolution. Queue 3(7) 54-62.
Tarboton, D.G., Luce, C.H., 1996. Utah energy balance snow accumulation and melt model (UEB). Citeseer.
Tarboton, D.G., Schreuders, K., Watson, D., Baker, M., 2009. Generalized terrain-based flow analysis of digital elevation models, Proceedings of the 18th World IMACS Congress and MODSIM09 International Congress on Modelling and Simulation, Cairns, Australia, pp. 2000-2006.
Tarboton, D.G., 2015. Terrain Analysis Using Digital Elevation Models (TauDEM). Utah Water Research Laboratory, Utah State University, Logan, Utah.
Tesfa, T.K., Tarboton, D.G., Watson, D.W., Schreuders, K.A., Baker, M.E., Wallace, R.M., 2011. Extraction of hydrological proximity measures from DEMs using parallel processing. Environmental Modelling & Software 26(12) 1696-1709.
Tristram, D., Hughes, D., Bradshaw, K., 2014. Accelerating a hydrological uncertainty ensemble model using graphics processing units (GPUs). Computers & Geosciences 62 178-186.
Winsemius, H., Van Beek, L., Jongman, B., Ward, P., Bouwman, A., 2013. A framework for global river flood risk assessments. Hydrology and Earth System Sciences 17(5) 1871-1892.
Wood, E.F., Roundy, J.K., Troy, T.J., van Beek, L.P.H., Bierkens, M.F.P., Blyth, E., de Roo, A., Döll, P., Ek, M., Famiglietti, J., Gochis, D., van de Giesen, N., Houser, P., Jaffé, P.R., Kollet, S., Lehner, B., Lettenmaier, D.P., Peters-Lidard, C., Sivapalan, M., Sheffield, J., Wade, A., Whitehead, P., 2011. Hyperresolution global land surface modeling: Meeting a grand challenge for monitoring Earth's terrestrial water. Water Resources Research 47(5) W05301.
Yavits, L., Morad, A., Ginosar, R., 2014. The effect of communication and synchronization on Amdahl's law in multicore systems. Parallel computing 40(1) 1-16.
You, J., Tarboton, D., Luce, C., 2014. Modeling the snow surface temperature with a one-layer energy balance snowmelt model. Hydrology and Earth System Sciences

18(12) 5061-5076.