High-performance computing in water resources hydrodynamics

M. Morales-Hernández, M. B. Sharif, S. Gangrade, T. T. Dullo, S.-C. Kao, A. Kalyanapu, S. K. Ghafoor, K. J. Evans, E. Madadi-Kandjani and B. R. Hodges

ABSTRACT

This work presents a vision of future water resources hydrodynamics codes that can fully utilize the strengths of modern high-performance computing (HPC). The advances to computing power, formerly driven by the improvement of central processing unit processors, now focus on parallel computing and, in particular, the use of graphics processing units (GPUs). However, this shift to a parallel framework requires refactoring the code to make efficient use of the data as well as changing even the nature of the algorithm that solves the system of equations. These concepts along with other features such as the precision for the computations, dry regions management, and input/output data are analyzed in this paper. A 2D multi-GPU flood code applied to a large-scale test case is used to corroborate our statements and ascertain the new challenges for the next-generation parallel water resources codes.

Key words | GPU, HPC, parallel codes, water resources hydrodynamics

M. Morales-Hernández (corresponding author)
K. J. Evans
Computational Sciences and Engineering Division,
Oak Ridge National Laboratory,
Oak Ridge, TN 37831, USA
E-mail: *moraleshernm@ornl.gov*

M. Morales-Hernández
S. Gangrade
S.-C. Kao
K. J. Evans
Climate Change Science Institute,
Oak Ridge National Laboratory,
Oak Ridge, TN 37831, USA

M. B. Sharif
S. K. Ghafoor
Department of Computer Science,
Tennessee Technological University,
Cookeville, TN 38505, USA

S. Gangrade
S.-C. Kao
Environmental Sciences Division,
Oak Ridge National Laboratory,
Oak Ridge, TN 37831, USA

T. T. Dullo
A. Kalyanapu
Department of Civil and Environmental Engineering,
Tennessee Technological University,
Cookeville, TN 38505, USA

E. Madadi-Kandjani
B. R. Hodges
National Center for Infrastructure Modeling and Management,
University of Texas at Austin,
Austin, TX 78758, USA

INTRODUCTION

Water resources modeling has reached an interesting point where the complexities of our codes and the capabilities of computers are pushing us in two different directions. On the one hand, we recognize that an inordinate amount of time is spent both debugging codes and building/maintaining the cadre of experts to adapt codes for new science. On the other hand, we want to take advantage of the latest, fastest, biggest computers that expand our modeling capabilities. The first pressure moves us toward object-oriented, reusable, and modular techniques, but these same techniques create communication bottlenecks limiting the effectiveness of high-performance computing (HPC).

HPC generally refers to the use of supercomputers and parallel processing to solve advanced problems; however,

doi: 10.2166/hydro.2020.163

the tools and techniques developed in today's HPC environments will also end up in tomorrow's engineering/science workstations and will be integral to on-demand cloud computing services. The conventional single central processing unit (CPU) environment is no longer sufficient for computationally intensive tasks, having made way to CPU clusters, computers using single graphics processing units (GPUs), and combining for the latest morphology: clusters of GPUs.

HPC systems, in general, and GPU systems, in particular, have the potential to significantly reduce overall computational times by calculating multiple operations in a single clock tick. A serial computer can only compute a single operation in a clock tick, so the overall computational time is determined by the chip speed – which (as discussed below) is no longer improving. Multiple CPU systems using both OpenMP and message-passing interface (MPI) strategies (shared and distributed memory parallelization) have been widely used over the past two decades. In a single clock tick, these approaches provide as many operations as there are computational cores. With multi-CPU systems, the controlling computational burden often shifts from the number of operations to the communication between processors. However, GPU computing has emerged in the last few years as one of the most promising and affordable pathways of acceleration due to its massively parallel architecture. A single GPU effectively contains more computational cores than all but the largest multi-CPU HPC machines but has the advantage of simpler communication between the cores. Furthermore, this technology can also be combined with OpenMP and MPI (the so-called multi-GPU) to achieve even faster computations, allowing models to cover larger temporal and spatial scales at finer grid resolution for water resources hydrodynamics problems. However, massive parallelization using HPC brings to the scene two actors that become crucial in the development of efficient models: the structure of data and communication and the choice of algorithm type and implementation within a HPC platform.

Parallel computing has yet to become ubiquitous in mainstream (non-research) computing as the advances are accompanied by two types of increased costs: (1) costs for organizations to create, debug, and maintain new parallel codes and (2) costs for users to adopt and train with new codes. Research codes are developed to advance the state-of-the-art and, unless driven by demand outside an organization, relatively little attention is paid to readability, usability, and maintainability. Thus, the return on investment for parallelization in a non-research organization will be small unless (1) new codes offer attractive speed-up such that the value gained by increased productivity is greater than the cost of moving to a new framework, (2) improved usability and the maintenance of parallel codes have consistent funding streams either inside or outside the user organization, and (3) HPC moves beyond research organizations and into everyday computing to widen the acceptance of parallelization. Presently, non-research users typically use multi-core machines to run multiple model cases, so they already have 10× to 30× overall production speed-up using conventional 10–30 core desktop machines. Therefore, for parallel codes to be attractive to non-research users, they need 100× or 1,000× speed-up over the serial code, which requires moving to HPC and away from desktop machines. However, we are at a tipping point in which the increasing popularity and availability of low-priced cloud services (such as those offered by Amazon or Google with HPC and GPU workstations) are building the non-research community interest, acceptance, and experience with HPC. Consequently, developing new parallel water resources codes and making them broadly available to the community are a timely issue.

In this paper, we explore some of the reasons why the tension between reusability and computational efficiency will push us inexorably toward HPC for water resources hydrodynamics. We will discuss why serial computing is a dead end (see the 'Moore's law is dead – for serial computing' section), the challenges of data communication across processors (see the 'The logistics of data – a thought problem' section), and how algorithm choice and the implementation can affect parallelization (see the 'How algorithm choice, data structure, and the implementation affects performance' section), thus making it difficult to port legacy serial codes to HPC. We close with an illustration and discussion of a benchmark test case (see the 'Benchmark problem: 2D flood code using CUDA and MPI' section) on a GPU cluster to bolster our statements. For this case, the performance of different implementations inside a physically based 2D shallow water model is analyzed.

MOORE'S LAW IS DEAD – FOR SERIAL COMPUTING

Although Moore's law was originally a statement about the number of transistors on a standard silicon chip, as a practical matter it translated into a doubling of the computer clock rate (frequency) every 18 months – i.e., more transistors allowed shorter times between successive operations on a chip. On a serial computer, every operation takes one or more clock ticks, so a 2.0 GHz computer can handle roughly twice the operations of a 1.0 GHz machine. Of course, this is a gross simplification – the chip architecture, bus speed, cache size, etc., will all affect the overall computational speed. Nevertheless, from the perspective of our classic hydrodynamics models from the 1970s through early 2000s, the computer clock rate was the principle factor controlling how fast a code executed – and thus the size of the grid cells, the allowable time step, and the type of the numerical algorithm provided the most efficient solution. For example, if we consider a 2D hydrodynamics model that is constrained by a Courant condition, we find that doubling of the clock rate could either be used to double a model's area of coverage (at a fixed grid size) or, for a fixed domain, decrease the individual cell area by 37%. Note that the practical grid resolution does not linearly decrease with the increasing clock rate as the model time step must also be reduced. In any case, the leap forward in what we could model every 18–24 months made the 1990s and early 2000s a golden era for advances in hydrodynamic modeling.

But everything changed around 2004. From that point forward, improvements in the processor clock rate slowed as shown in Figure 1. Over the past 15 years, the improvements in the effective computational speed have been mainly associated with chip architecture, bus throughput, or multi-core/multi-thread operations. That is, with regard to serial computation – Moore's law is dead (Flamm 2017). However, if we consider the increased speed available through parallel processing, then computer capabilities continue to rapidly expand and Moore's law is very much alive. In early 2019, the latest commercially available CPU has 32 cores (64 threads) on a single chip and is priced at about US$ 1,800. This chip is the equivalent of the high-performance Beowulf computing cluster of only 15 years ago – but for less than 10% of the price.

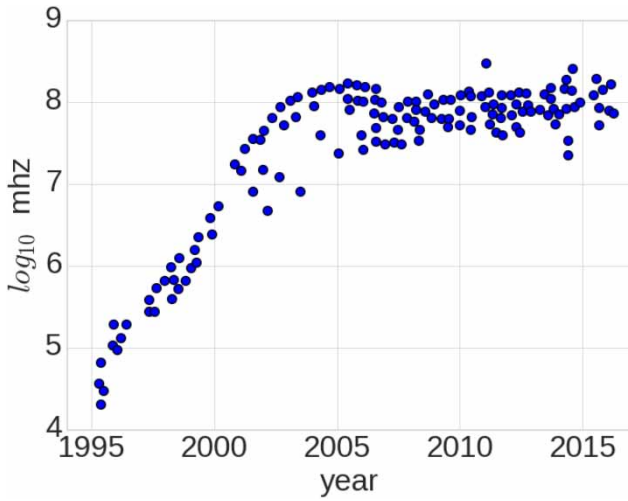

Figure 1 | Logarithm of the processor clock rate for Intel desktop processors running SPEC CPU benchmarks, by the first availability date of tested hardware (revised using data from Flamm (2017)).

So, why does the end of Moore's law for serial computers matter to water resources? In short, unless a model is written to take advantage of parallel processing, the speed-up advantage of today's computers for the model itself is small. Here, we have the crux of the problem: our legacy hydrodynamic models have been written with algorithms designed for a world of serial processing, and these algorithms typically do not 'scale' well on the parallel architecture. That is, a code that perfectly 'scales' will get 1,000× speed-up with 1,000 cores, but this requires solution algorithms designed from the ground up for parallel processing (see the 'How algorithm choice, data structure, and the implementation affects performance' section, below).

Simply put, our legacy water resources codes are not getting faster as computers get more parallel. Whether or not this matter depends on your viewpoint on artificial intelligence (AI) and machine learning (ML) versus mechanistic models. The advantage of AI–ML methods is that they are easily 'chunked' into sections for parallel processing across any size of the machine and do not have the communications burden (see 'The logistics of data – a thought problem' section) of our legacy mechanistic codes. We expect the speed of AI–ML models to continue to increase linearly with advances in multi-core machines. Thus, if the speed of mechanistic models remains stagnant, then the AI–ML models will win out by solving problems that

mechanistic models cannot (although AI–ML will require extensive data and training). Our view is that both AI–ML and mechanistic models are needed to advance science and both model types should be designed for multi-core computation. Indeed, we believe that one of the greatest long-term applications will be using mechanistic models to train AI–ML where only sparsely observed data are available. We see a future where AI–ML models are an extension of mechanistic water resources models and not a replacement.

THE LOGISTICS OF DATA – A THOUGHT PROBLEM

Efficient parallel code execution balances computation against communication. The latter can be thought of as the 'logistics of data' – getting the right data at the right time to the right processor core. The challenge can be readily illustrated with a thought problem: imagine a system that is defined by 10^6 computational elements where we need 10 operations per element for each time step of an initial value problem. Nominally, we have 10^7 operations per time step, which is the expected time scale for the computation of a single time step on a serial computer. Let us further imagine that we have a machine with 10^4 cores – i.e., each core handles 10^2 elements and is thus responsible for 10^3 operations per core per time step. In a perfectly efficient parallel operation, the computational time per model time step would scale on $10^7/10^4 = 10^3$ computational operations, i.e., 10^4 times faster than the serial computation of 10^7 operations. This result is our ideal – a code where data logistics do not matter and we get linear speed-up by increasing the number of processors.

Now let's consider how the logistics of data affect the computation. We start at some time step n with every element having its own memory and storing all the data for the problem – this is known as a single program, multiple data (SPMD) parallel system. Each processor independently computes the $n+1$ time-step data for its 10^2 elements, which provides a condition where each processor knows its own new data as well as the old data across the entire system, but it knows nothing of the new computations on the other processors. The processors now must communicate. If every element requires the $n+1$ data from every other element to compute the following $n+2$ time step – which is a big 'if' as discussed below – then each of the 10^4 processors must push its 10^2 pieces of time $n+1$ data out to 9,999 other processors. The resulting data communication problem scales on $10^2 \times 10^4 \times 9,999 \approx 10^{10}$. Thus, a complete time step requires the time for both 10^3 computational clock ticks (10^7 operations in parallel across the 10^4 processors) and 10^{10} communication operations. Note the contrast with a serial computer that requires 10^7 computational operations but has zero communication operations. Thus, parallel computing is essentially a tradeoff of computational clock ticks for communication operations, which is only efficient if the time costs of data logistics are less than the time costs of computation.

It should now be clear that the 'bus' that exchanges data across processors plays a critical role in parallel solution efficiency. A related capability is the 'cache' that stores data locally on a multi-core processor for fast access. The ideal parallel efficiency for our thought problem above will only be achieved if the bus/cache capacities are sufficient to handle 10^{10} simultaneous data transfers. This problem is similar to handling traffic in the city – given enough lanes and a high enough traffic speed we can (in theory) handle every car on the street at the same time – but realistically we may not have the lanes available when we need them. The key point is that the parallel performance of a code depends on how much data need to be passed between processors relative to the data transfer bandwidth provided by the machine.

Our thought problem above is arguably a 'straw man' as it assumes that all data throughout the domain are needed to compute the solution at any given point. Although this is true for some problems, for most water resources models the local domain of data dependency is physically limited by the speed of information propagation through the system. For example, information on backwater effects in rivers propagate upstream at the gravity wave speed, which limits the domain neighborhood for the data that affects any particular point over a given time interval. Unfortunately, as discussed in 'How algorithm choice, data structure, and the implementation affects performance' section below, the choice of numerical discretization methods can artificially inflate the domain that mathematically (if not physically) affects the solution at a point. Thus, our

straw man thought problem can become a real problem if our numerical algorithms require extensive data transfer.

In water resources, we generally do not have control over the computer architecture that provides the bus capacity, but we do have some say in the data transfer required by our algorithms. For our codes to operate effectively over a wide range of parallel and cloud computers, the data-dependency domain for any computational element should be limited (as much as possible) so as to minimize data transfers.

HOW ALGORITHM CHOICE, DATA STRUCTURE, AND THE IMPLEMENTATION AFFECTS PERFORMANCE

Overview

In this section, we discuss some of the details that might not be readily apparent when moving from serial to HPC parallel programming. This includes the choice of the time-marching scheme (see the 'Implicit versus explicit' section), the data structures (see the 'Data structure' section), the precision used in computations (see the 'Single versus double precision' section), handling of dry cells (see the 'Computing over wet space or all space?' section), and issues with input/output (I/O) (see the 'Challenges and bottlenecks with input/output (I/O) data' section). Addressing these issues should mean that a legacy water resources code needs to be entirely re-written, or even completely re-thought, for an efficient parallel implementation.

Implicit versus explicit

One of the first concepts taught in the solution of time-dependent numerical modeling is 'explicit' versus 'implicit' algorithms. Although both implicit and explicit algorithms have seen wide use through the literature – arguably with similar emphasis in the 1960s and 1970s – the implicit algorithms became a dominant force in finite-difference models of hydrodynamics in the 1980s and 1990s with the development of efficient linear solvers (Smith 1985), the SIMPLE pressure method (Patankar 1980), and semi-implicit solution methods for free-surface flows (Casulli 1999; Casulli & Zanolli 2002). However, many finite-element hydrodynamics solvers continued to use explicit solution methods, arguably due to the computational expense associated with satisfying the global finite-element problem posed in an implicit form.

Implicit methods are often favored because of their ability to time-march an unsteady problem at a larger time step than allowed with an explicit model. This advantage of the implicit approach is also the source of its major drawback. The solution of the matrix inversion problem $[x] = [A]^{-1}[b]$ implies that the solution at any point in space can (in theory) influence any other point in space – which is what allows an arbitrarily large time step and simultaneously implies an undesirable global data dependency (i.e., our extreme thought problem in 'The logistics of data – a thought problem' section). To illustrate, we can imagine a simple discrete problem for $i \in \{1 \ldots m\}$ elements where the time-march of the ith element depends only on its neighbors, designated as $i+1$ and $i-1$. Each element has an algebraic equation of the form

$$\alpha_{i-1}x_{i-1} + \beta_i x_i + \gamma_{i+1}x_{i+1} = b_i$$

where α, β, and γ are coefficients and the subscripts represent the element location in a vector. The simultaneous solution of all the equations and their dependencies is through a tridiagonal matrix $[A]$ of size $m \times m$ where the only non-zero elements are α, β, and γ values along three main diagonals. Even this simple matrix will have an inverse $[A]^{-1}$ where every term is non-zero. Thus, the solution for the x_i element is, in general

$$x_i = A_{i,1}^{-1}b_1 + A_{i,2}^{-1}b_2 + \ldots + A_{i,m}^{-1}b_m$$

Because of this implicit formulation, a stable solution can (theoretically) be obtained at any size of the time step as information can travel anywhere throughout the domain (i.e., through the $A_{i,j}$ coefficients) in a single time step. Note that a stable solution at a large time step can be highly inaccurate, so the advantage of unlimited stability is curtailed by the need to solve at a time step that is consistent with the underlying temporal variability in the physics. For example, for estuarine tidal dynamics, model stability at a daily time step is irrelevant as a much smaller time step is required to resolve sub-daily tidal oscillations.

In any case, the long time-step advantage for serial machines becomes a disadvantage for parallel machines because the source term vector, $[b]$, is generally dependent on the solution $[x]$ from a prior time step. Similarly, nonlinearities in the governing equation imply the coefficients of the A matrix can be functions of both space and time, which must be handled either by time lagging (using time n for an approximation of time $n+1$ values) or with a nonlinear matrix solver. These dependencies imply that all data from the prior time step across the entire domain must be provided to each computational core – i.e., a single location must have access to the entire domain of data at every time step. It follows that this global transport of information for a large time step with an implicit solver requires global data dependency, with all the disadvantages for parallelization discussed in the 'The logistics of data – a thought problem' section. Having said that, implicit schemes can be found in the literature using domain decomposition algorithms across multi-core machines (Yang *et al.* 2010), and they can be advantageous when the processes occur at different time scales (Evans *et al.* 2019). Furthermore, the disadvantages of implicit methods can be reduced with the use of standard linear algebra tools such as preconditioners.

In contrast to implicit algorithms, explicit algorithms are based on the idea that the local time n data can be used to predict the time $n+1$ data without recourse to global information. Note that herein, we constrain our discussion on finite-difference, finite-volume, and discontinuous Galerkin algorithms – neglecting finite-element algorithms as they generally require a global solution even when using an explicit time advance. Explicit time-marching methods are inherently constrained by a Courant condition: the local propagation of information must be at a CFL number below some cutoff, which is typically near unity. Formally, the CFL number is defined as $c\Delta t\Delta x^{-1}$ where c is the speed of information propagation (e.g., velocity, gravity wave speed, and acoustic wave speed), Δt is the time step, and Δx is the spatial discretization. Thus, for a fixed value of c (which depends on physics), the time step is linearly related to the spatial discretization. It follows that explicit methods inherently limit the domain of dependence of a model time step to its immediate neighbors, which limits the data transfers if the data structure and connectivity is well designed for the number of computing cores, as discussed in the 'Data structure' section.

Overall, for some interval of the modeled time, an explicit algorithm will use more computational operations (due to smaller time steps) than an implicit algorithm. However, the data transfer requirements are generally far less for the explicit approach. Indeed, for any implicit algorithm in a problem with true global data dependency (e.g., solving incompressible flow in a pressurized pipe), the communication problem will scale as $N_{e/c}N_c^2$, where N_c is the number of processor cores and $N_{e/c}$ is the number of elements solved per core. That is, each of the N_c processors must push all of its $N_{e/c}$ data items to the $N_c - 1$ other cores. More simply, let $N_e = N_{e/c}N_c$ be the number of computational elements, from which it follows that the implicit algorithm communication scales as N_eN_c. In contrast, an explicit approach will have communication that scales on $N_{b/c}N_c$ where $N_{b/c}$ is the number of bounding elements per core – which is where communication between cores takes place. It follows that as long as $N_{b/c} \ll N_e$, the explicit algorithm will have a substantial advantage in data transfer. In particular, where a problem can be posed as a network (e.g., water distribution or stormwater systems), the data structure can be arranged to minimize $N_{b/c}$.

In practice, the situation is not quite as bleak for implicit solutions as the above implies – the physical dependencies between locations depend on the physical transport of information (by velocity, gravity wave, or acoustic wave), so the $[A]^{-1}$ matrix will have many near-zero terms that can be effectively neglected in iterative solution methods (Houzeaux *et al.* 2018; Bruno *et al.* 2019). There has been extensive work done on parallel linear and nonlinear solvers to automatically break a matrix into pieces for effective distribution across a large number of processors. Arguably, the main difference between using implicit solvers and explicit solvers is who is responsible for making the code more parallel. For implicit codes, we are unlikely to be able to match the speed and parallelization of prepackaged matrix solvers designed by numerical linear algebra experts. Thus, if we formulate our models based on implicit algorithms, we will also rely on others to build, test, and maintain the linear/nonlinear solver. Furthermore, our codes must be designed with data structures that are efficiently used by the solver, which narrows our flexibility in the code design. In contrast,

when we formulate the numerical model as an explicit time-march, we have complete control over the parallelization and data structure. We can build our models to take advantage of the peculiarities of our discipline, which are quite different from those in the computational fluid dynamics, ocean, and atmospheric modeling communities that dominate discussions of high-end parallelization.

We believe that the next-generation water resources hydrodynamics models should predominantly use explicit algorithms with data structures (see 'Data structure' section, below) that minimize inter-processor communications. Where implicit methods are desired or deemed necessary, the matrix solvers should not be written by water resources scientists and engineers but instead should use open-source high-performance numerical analysis code modules that have been designed and tested by parallel processing experts (Babuska & Guo 1992; Blackford *et al.* 1997; Anderson *et al.* 1999; Heroux *et al.* 2005; Kolev & Dobrev 2010). Note that implicit codes need to optimize their data structures with respect to the matrix solver – which will typically disadvantage object-oriented codes.

Data structure

The performance of an explicit time-marching algorithm is influenced by the relationship between the compactness of data storage for each computational core and the number of neighbor communications required by each computational element (i.e., $N_{b/c}$ in the 'Implicit versus explicit' section). Let us return to our (see the 'The logistics of data – a thought problem' section) thought problem of 10^6 elements on a 10^4 core machine (10^2 elements per core). We imagine an explicit algorithm where the time advance of the i element depends only on two physical neighbors – e.g., a single line of piping. If the data space is arranged, so the physical neighbors are also the data storage neighbors $i-1$ and $i+1$, then each core will be required to pull only two pieces of data from two adjacent cores – i.e., the simultaneous communication problem scales on 2×10^4 as only two bounding elements between each core need data from another core. In contrast, we can imagine a random distribution within a data vector, e.g., a random element located at $i=1,743$ that has neighbor data stored at $i=74$ and $i=9,235$, which are not computed on the same core. For this random data distribution, the communications pull across cores scales on $2\times10^2\times10^4$, i.e., each element pulls two pieces of data from different cores. To put this in more general terms, the random data distribution for an explicit method has $N_{b/c}\sim N_{f/e}N_{e/c}$ where $N_{b/c}$ is the number of bounding elements per core, $N_{f/e}$ is the number of faces per element, and $N_{e/c}$ is the number of elements per core. As discussed in the 'Implicit versus explicit' section, this implies an explicit communication time scale such that $N_{b/c}N_c=N_{f/e}N_{e/c}N_c$. In contrast, an optimum data distribution has communication scaling on $N_{f/e}N_c$. Thus, where $N_{e/c}\gg1$, the relationship between the data structure and core communication will be important to the overall efficiency of the algorithm. Conversely, as $N_{e/c}\sim1$, i.e., $N_c\rightarrow N_e$, the communication burden will come to dominate even a well-designed data structure. GPU machines introduce an interesting technological twist as they provide a structured arrangement of cores that is similar to a rectilinear grid in a Cartesian space. Thus, structured meshes are more convenient for this paradigm – nevertheless, some reordering algorithms have been successfully applied to improve the speed-up when dealing with unstructured meshes (Lacasta *et al.* 2014). One of the ironic aspects of the introduction of GPU machines is that many 2D and 3D structured-grid codes in the past 20 years have been reformulated to represent space as 1D vectors to make better use of serial processing (Herzfeld 2006).

The important point on data structures is that the data arrangement that is optimum for a given system of N_e elements on a set of N_c cores may not be optimal if the number of cores is doubled or if the system itself changes connectivity. As yet, we do not have a general approach to optimizing data structures for parallel water resources problems as a function of the system type, connectivity, the number of elements, and the number or arrangement of processing cores. Non-optimal data structures are unlikely to scale well as the number of parallel cores is increased, so addressing this issue in a general way is critical to building robust models that are not made obsolete by rapid advances in computer science.

Single versus double precision

Most compilers have an option to use either single- or double-precision computation – a choice that is completely

open to the programmer. The difference between approaches is the number of bits that are used to represent the real number: single-precision floating point arithmetic uses 32-bit floating point numbers, whereas double precision uses 64 bit.

The use of single-precision real numbers is generally deprecated in modern serial CPU systems because the reduced precision does not provide any significant advantage, i.e., there is only a slight computational speed-up despite the dramatic loss in precision. However, GPU processors arose from chips optimized for single-precision operations because double precision is unnecessary for graphics displays viewed by the human eye. The high speed of single-precision GPUs provides opportunities for significantly improving the HPC computational speed when reduced precision can be tolerated (Itu *et al.* 2011; Váňa *et al.* 2017).

The main advantage of double precision is that machine epsilon (relative error due to rounding) is 10^{-15}, which implies the accumulation of the numerical truncation error cannot build rapidly in a time-accurate simulation. In contrast, machine epsilon for single precision is typically 10^{-6}, which can compromise the overall accuracy of a numerical scheme and/or the convergence of an iterative method. In general, double precision is necessary when (i) tiny relative differences are significant, (ii) large variations in the variables are expected, or (iii) a long time period is simulated.

Indeed, single-precision computations would be irrelevant except for two facts: (1) single precision uses half the memory, which has implications for both storage and communications and (2) GPUs are significantly faster with single-precision computations.

Depending on the graphics card, single-precision GPU computation can be up to eight times faster than double precision. Furthermore, memory requirements can be a limiting factor for large-scale problems and using single precision allows storage of twice the computational area. Finally, data transfers and communications between nodes in multi-GPU computing and between device/host are doubled with double-precision numbers, which can cause a bottleneck that slows the overall simulation time.

Thus, although our legacy water resources hydrodynamics models are designed, calibrated, and validated with double-precision numbers, there are good reasons to work toward adapting such models to single-precision numbers for use on fast GPU machines. As yet, we simply do not know what portions of our algorithms could be robust on single-precision machines, or what new techniques could be used to control error accumulation.

Computing over wet space or all space?

Although models for hydrology and atmospheric science typically involve a space-filling grid, hydrodynamic models for flow across the landscape must answer an important question – do we compute over only the cells that are wet at this moment, or do we include all the cells in the landscape that might *potentially* become wet? For serial computers and multi-CPU machines with only a few processors, the answer is clear: compute only on the actually wet cells as any computational cycles spent on dry cells are wasted. Indeed, part of the popularity of unstructured and adaptive meshes in water resources hydrodynamics is the ability to *a priori* limit the computational domain to the wet area without the 'wasted' space implied by a 2D-structured grid with dry cells. Of course, the 'wasted' space in a 2D-structured grid is readily overcome with 1D array mapping (Herzfeld 2006), but that returns us to the problem of optimum data arrangement for communication as discussed in the 'Data structure' section.

Massively parallel multi-CPU and GPU computers change the calculus for handling the wet/dry problem. In a serial computer, the wet/dry problem is addressed with an 'if' statement that simply skips the computational cycle for a dry cell and the compute point moves onto the next cell. This approach is also valid for a small multi-CPU machine where a large number of computational elements are assigned to each compute node. However, for an efficient massively parallel computation, the grid cells become geographically constrained – i.e., there is an advantage (in communication) for all the grid cells on a particular compute node being in the same local geographic area, which means that it is likely that a simple 'if' approach (or 'where' in Fortran) will result in many (or even all) of the cells at some compute nodes being dry. Thus, the traditional approach to handling the wet/dry dilemma for water resources hydrodynamics can result in load imbalance and computational inefficiencies that can affect scalability (Tallent *et al.* 2010).

For strict computational efficiency and load balancing, the obvious answer for a GPU machine is to routinely recalculate and redistribute the wet cells over the compute nodes. However, such recalculation and redistribution have its own costs (Brodtkorb *et al.* 2012). At this time, it is not clear how best to balance the inefficiencies of computing dry cells versus the inefficiencies of redistributing data across the compute nodes. What is clear is that finding effective approaches for handling the wet/dry conundrum will determine how well our water resources hydrodynamics models make use of the next generation of computers.

Challenges and bottlenecks with I/O data

Although HPC allows faster computations over larger machine memory spaces, this growth has not been accompanied by comparable advances in the efficient management of I/O data (Cardone-Noott *et al.* 2018). Indeed, for hydrodynamics simulation in water resources over large areas over long timescales, the scalability of an HPC application and its overall runtime might depend on how well or badly managed is the input and output data. Parallel I/O procedures have been demonstrated to improve I/O scalability (Behzad *et al.* 2019), but these typically require application-specific tuning to achieve an optimum improvement. Additionally, performance portability is not guaranteed when moving to different computers or supercomputers due to the complexity of custom middleware and hardware required for parallel I/O. Furthermore, large hydrodynamics simulations require large input data sets (e.g., topography, urban infrastructure, rainfall, and groundwater) that might require the interface with other simulation models or real-time data sets, which creates additional challenges and potential computing bottlenecks that cannot be directly solved with HPC hardware or software improvements.

Output management for hydrodynamic simulations can be seen as a tradeoff between raw data and statistical processing designed into the code. For example, to reduce the output load, it is straightforward to accumulate hydrodynamic model results (at time steps scaling on 1 min) over 15 min or 1 h time scales to output mean, median, and variance along with the instantaneous values at the desired output time interval. However, such output data are inherently limited in its future use; it is impossible to reprocess for a different set of statistical metrics to answer questions not envisioned when the output processing routines were designed – e.g., the 5 min mean cannot be extracted from data processed to a 1 h output interval unless it was *a priori* designed as an output data type. Thus, raw data output at a relatively short time interval is preferred when a large investment of the computer time is made in generating a simulation data set for future inquiry – which implies an output bottleneck that can limit HPC scalability.

The format of output data can also be a concern. In general, users often prefer ASCII format files for machine independence and human readability. However, computers can transfer binary data more quickly as it does not require an intermediary interpretation step. For large HPC simulations, machine binary outputs will generally be more efficient and translations to standard ASCII or HTML formats should be considered an offline problem. However, ASCII output does lend itself to writing individual files from each GPU (as tested in the 'Results' section, below) rather than a comprehensive binary file that requires coordination across multiple GPUs. Thus, despite the conventional wisdom that prefers binary files, there may be opportunities to optimize output using customized ASCII files or compression (Liu *et al.* 2013).

There are two main input data challenges for HPC in water resources hydrodynamics: (i) pre-processing of data sets from disparate sources and (ii) efficient linking with other simulation models (e.g., groundwater, rainfall, and coastal inundation). As an example of the former, in developing a comprehensive flood model across a catchment with multiple cities, the stormwater infrastructure data and urban building data from different cities will likely be available in different data formats and with different accuracies. The time required to understand, process, and validate the synthesis of such data can be longer than running the hydrodynamics model itself. The second challenge – linking of models – has been a well-recognized problem that simply does not have a good answer for HPC. The Open Modeling Interface (OpenMI) and the Earth Surface Modeling Framework (ESMF) both have aspects for model integration, but their capabilities are limited for HPC, particularly for GPU systems (Buahin & Horsburgh 2018). For the present, the efficient integration of input for multi-model simulations for HPC requires customization of the model linkages at

the model source-code level to prevent communication bottlenecks from dominating the runtime and scalability.

Overall, efficient I/O remains a troubling challenge for HPC. It is not clear how much effort should be directed at optimizing I/O through custom software as such methods might be pushed rapidly into obsolescence by new hardware. For example, NVIDIA recently announced 'GPUDirect' storage, which bypasses the CPU in communicating data from GPU local memory to flash storage (Feldman 2019). Such advances in hardware are likely to scale better than any workaround developed in the custom software.

BENCHMARK PROBLEM: 2D FLOOD CODE USING CUDA AND MPI

Description

Results for a benchmark 2D flood code are described below to help illustrate some of the challenges in HPC described in the section 'How algorithm choice, data structure, and the implementation affects performance'. The model solves the 2D shallow water equations on a structured (Cartesian) mesh by means of a finite-volume, upwind, explicit scheme based on Roe's linearization of the governing equations posed as a Riemann problem. The derivation of the numerical scheme follows (Murillo & García-Navarro 2010; Morales-Hernández *et al.* 2015) for the fluxes and bed slope source terms, including the correct estimation of bed slope source terms at each edge to avoid dramatic reductions in the time step size for numerical stability. Roughness terms are discretized following Xia & Liang (2018) using a local implicit formulation – note that this approach does not alter the overall explicitness of the time-march and does not introduce non-local data dependencies. Thus, the time step is restricted by the Courant condition. The specific aspects of the method are omitted here to focus on aspects of parallelization. Further details about the numerical scheme can be found in Murillo & García-Navarro (2010); Morales-Hernández *et al.* (2015); and Xia & Liang (2018). A flowchart of the implementation of this code is displayed in Figure 2. Note the colors representing the different processing groups (I/O, MPI, GPU, and CPU) are the same as used further below in the analyses of the results.

The flow chart of Figure 2 follows a classical flow diagram for this type of scheme. A critical aspect of the HPC computational flow is the domain decomposition. It is necessary to exchange some data as well as to copy some information to/from the neighboring subdomains (*halo_copy_to_gpu* and *halo_copy_from_gpu*) using auxiliary variables. The communications are done using the MPI, which is a message-passing library interface specification that allows a portable and scalable communication for large-scale parallel applications. The main feature of MPI is that it does not need shared memory, which makes it valuable in the programming of distributed systems (i.e., across multiple GPUs), which is unavoidable for large domains.

As discussed in the 'The logistics of data – a thought problem' section, the main goal of HPC is to subdivide the computational domain into subdomains of equal or variable size trying to guarantee the following requirements:

- Each subdomain contains the same amount of computational effort.
- The communication between the subdomain and its neighbor subdomain is minimal.

In this work, a 1D row-wise domain decomposition is applied, which can be imagined as each node having access to an east–west strip of contiguous data and communicating with other nodes across north–south boundaries. Although for the 2D framework there exists other ways of partitioning (e.g., optimized 2D blocks with east–west and north–south edges can require less communication with asynchronous data transfer), the row-wise 1D partitioning has the advantage of simplicity in its implementation.

Each subdomain sends information to the north and south neighbor subdomains and receives information from them. As the MPI subroutines are called in the CPU, these values are copied for each subdomain to auxiliary variables in the CPU. Once the exchange has been done, these values are copied back to the GPU for the next iteration.

This 2D flood code is applied to a real-world test case to evaluate the performance of the scheme. The test case is the massive flood produced by Hurricane Harvey in the summer

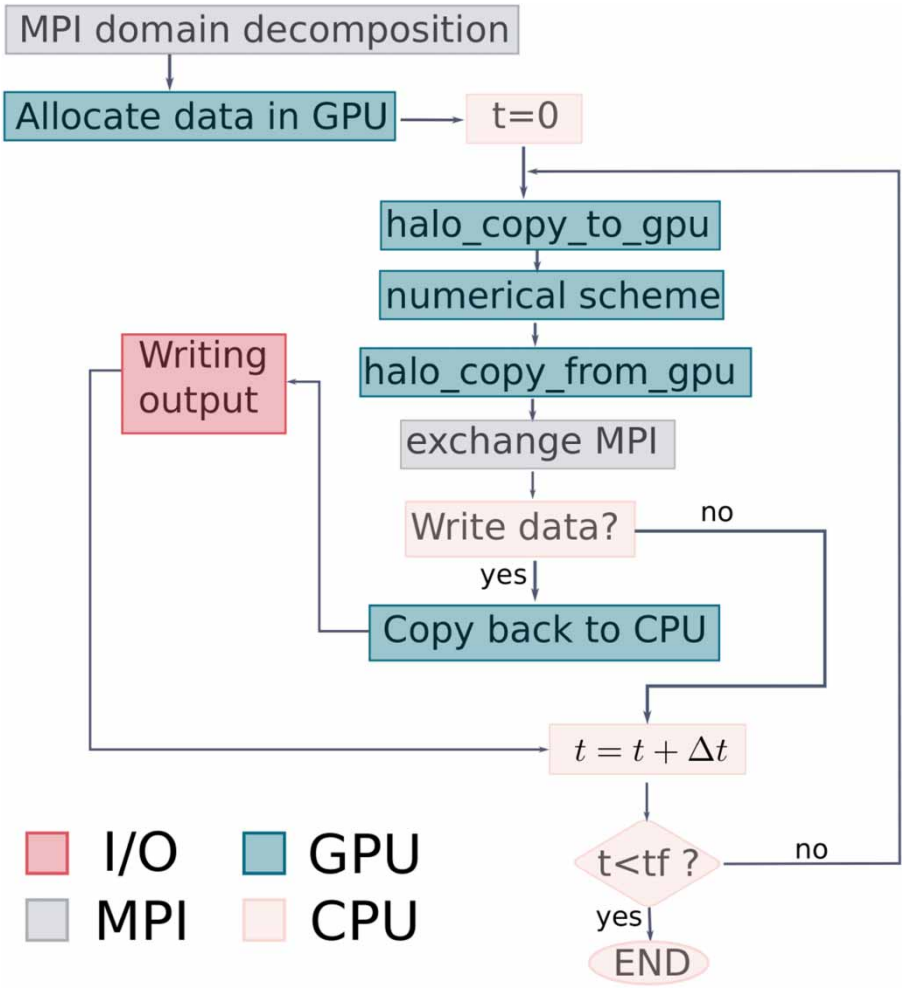

Figure 2 | Flowchart of the GPU implementation.

2017 along the Gulf Coast of the United States, which was the heaviest large-scale rainfall event in the US history. The overall spatial scale of the flooding makes this a challenging test case. The domain is about 7,000 km^2, which we model over a 10-day event (6 days of model spin-up followed by the heaviest 4 days Hurricane Harvey rainfall). Simulations have been conducted with two Cartesian grid meshes: coarse (30 m × 30 m) and fine (10 m × 10 m). The output data interval is set to 1,800 s. Further details can be found in Dullo *et al.* (2018).

The simulations were carried out using Oak Ridge National Laboratory's 200 petaflop supercomputer, Summit, consisting of 4,608 nodes with each node containing 6 Tesla V100 GPUs. Four different implementations are considered, as shown in Table 1.

Implementation double wet binary (DWB) is considered the base implementation, that is, computation in double precision, writing the information in the binary format and computing just on wet cells, excluding the dry cells with an 'if statement'. Cases single wet binary (SWB), double all binary (DAB), and double wet ASCII (DWA) represent variations in the baseline: SWB explores effects of a single-precision implementation, DAB evaluates the inclusion of all cells during the computation, and DWA analyzes the impact of writing the information in a conventional ASCII format in contrast to the binary format.

The last case introduces an important difference in the implementation when using multi-GPU computing: in the ASCII output, each subdomain is in charge of writing its

Table 1 | Tested implementations

Implementation	Precision (see the 'Single versus double precision' section)	Wet/all (see the 'Computing over wet space or all space?' section)	I/O (see the 'Challenges and bottlenecks with input/output (I/O) data' section)
DWB	Double	Wet	Binary
SWB	Single	Wet	Binary
DAB	Double	All	Binary
DWA	Double	Wet	ASCII

own information without joining them in a single file; in contrast, the binary format gathers the information for each MPI task and consolidates it within a single domain before its sequential write.

Note that the data structure challenge described in the 'Data structure' section is analyzed by means of different grid resolutions (consequently different numbers of grid cells) and the computation on multiple GPUs. However, the dichotomy between explicit and implicit schemes is outside the scope of this work. An explicit time-marching algorithm has been chosen, being more suitable for this type of problems as argued in the 'Implicit versus explicit' section.

Results

The performance of the model with the four implementations shown in Table 1 is analyzed in Figure 3. Each implementation is used to run both grid resolutions (30 and 10 m) with a varying number of GPUs: 1, 2, 4, 8, and 16, for a total of 40 test cases. Note that the 30 m grid resolution corresponds to 7.5 M computational cells, whereas the 10 m resolution corresponds to 68 M cells. As the increasing number of cells is directly related to computational costs, it is convenient to use 7.5 and 68 M to distinguish the different grid meshes in the following analyses. To account for system variability (Evans *et al.* 2019), times are measured as the 'average out of 5' runs for the same configuration.

The problems of scaling can be quantified by considering the relative speed-up with respect to 1 GPU which, for a fixed system size, can be computed as follows:

$$\text{Speed-up}(x\ \text{GPUs}) = \frac{\text{Time}(1\ \text{GPU})}{\text{Time}(x\ \text{GPUs})} \quad (1)$$

The speed-up is plotted in Figure 4 for test cases with 7.5 and 68 M grid cells. The theoretical perfect speed-up is also included in these graphs. We can obtain further insight into parallel scaling by evaluating the computational time contribution from different processes, i.e., using the color index from Figure 2. The breakdown of the computational time across all 40 simulations is shown in Figure 5.

As another approach for comparison, Figure 6 displays the average time consumed by each process (CPU, I/O, MPI, and GPU, from upper-left to lower-right, respectively) in the logarithmic scale against the number of GPUs used. The same color represents the same implementation. Empty symbols refer to the 7.5 M test case, while filled symbols are the 68 M test case.

To better understand how 'wet' versus 'all' cell computations (see 'Computing over wet space or all space?' section) affect the load balancing, the percentage of the overall computational time consumed by each process (CPU, I/O, MPI, and GPU) along the different MPI ranks is shown for both 7.5 (Figure 7) and 68 M (Figure 8) for the DWB and DAB test cases. Note that each MPI rank (from zero to the number of GPUs minus one) is in charge of computing its own subdomain; therefore, these plots can be seen as the load balancing between subdomains.

Discussion

The results displayed in Figure 3 show that, unsurprisingly, the single-precision (SWB) implementation is the fastest approach, being around 20–40% faster than the baseline DWB implementation. The DWA implementation is always the slowest: this case writes the information in the ASCII format and takes between 40% and 300% of extra time to run the solution despite the use of individual files for each GPU. The DAB implementation is the slowest of the cases using binary output, indicating that the solution of 'all' cells (versus 'wet' cells) remains important despite the wide area inundated during Hurricane Harvey. Naturally, the smaller 7.5 M cases are always faster than the equivalent 68 M cases, but of greater interest is the contrast in overall trends as more GPUs are applied. It can be seen that the 68 M cases remain scalable up to 16 GPUs – i.e., the computational time decreases by somewhat less than 50% with each doubling of the applied GPUs. In contrast,

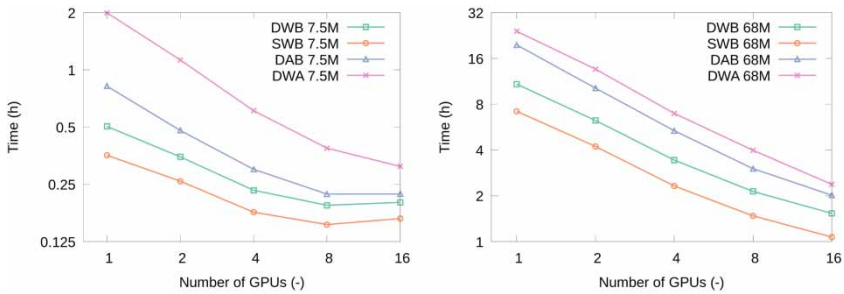

Figure 3 | Wall-clock computational time consumed by each implementation for the test case 7.5 M (left) and the test case 68 M (right). Figure scales are log–log.

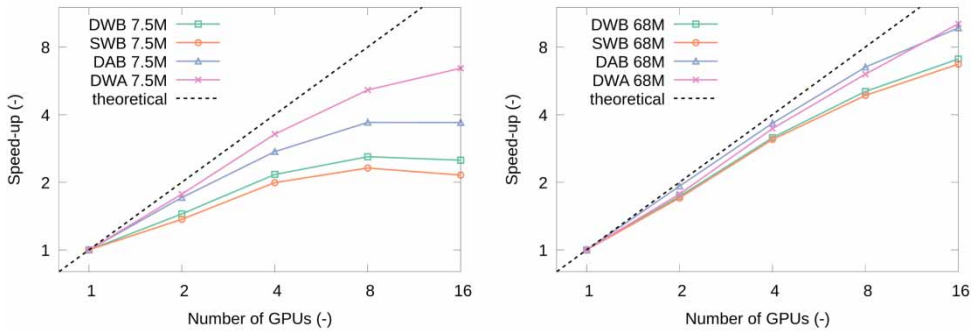

Figure 4 | Speed-up with respect to 1 GPU achieved by each implementation for the test case 7.5 M (left) and the test case 68 M (right). Scales are log–log.

the 7.5 M case shows performance saturation beginning at about 4 GPUs. Indeed, the 7.5 M model actually requires more wall-clock time to compute with 16 than 8 GPUs when using binary data writing.

Additionally, as pointed out in Figure 4, the parallel scalability (closeness to theoretical perfect scaling) for the present 2D flood model depends on the number of computational cells. Note that a close comparison of Figures 3 and 4 highlights an interesting effect – in the latter figure, the DAB and DWA cases have the best scalability but actually have the worst overall performance in the former figure. That is, the DAB case introduces the unnecessary solution of the dry cells, which makes the simulation more scalable (but slower) simply by making the problem larger. Similarly, the DWA case introduces inefficient computations through the ASCII writing that effectively make the number of computations greater, and because the ASCII writing is separate for each GPU, the overall parallel scaling is improved. The lesson here is that the reader must be careful in considering a speed-up analysis that does not have a corresponding wall-clock time analysis. Models that obtain scalability by introducing inefficient computations that extend the wall-clock time are not what the community needs!

Some additional points can be extracted from Figures 5 and 6:

- I/O time is relatively invariant with the number of GPUs for the binary output DWB, SWB, and DAB cases. However, I/O time in binary for a double-precision computation (DWB and DAB) is higher than that for a single-precision computation (SWB) although the difference is not remarkable. The DWA implementation (ASCII output format) is dramatically slowing the overall runtime. There is more time spent on writing than computing. However, the I/O time decreases with the increasing number of GPUs since subdomain writes are handled separately.
- GPU computation is consistent: more GPUs result in lowering the GPU computational time. In fact, GPU time scales according to the number of GPUs, although a small loss of performance, is seen for 16 GPUs in the 7.5 M cells. That is, the GPUs themselves are never

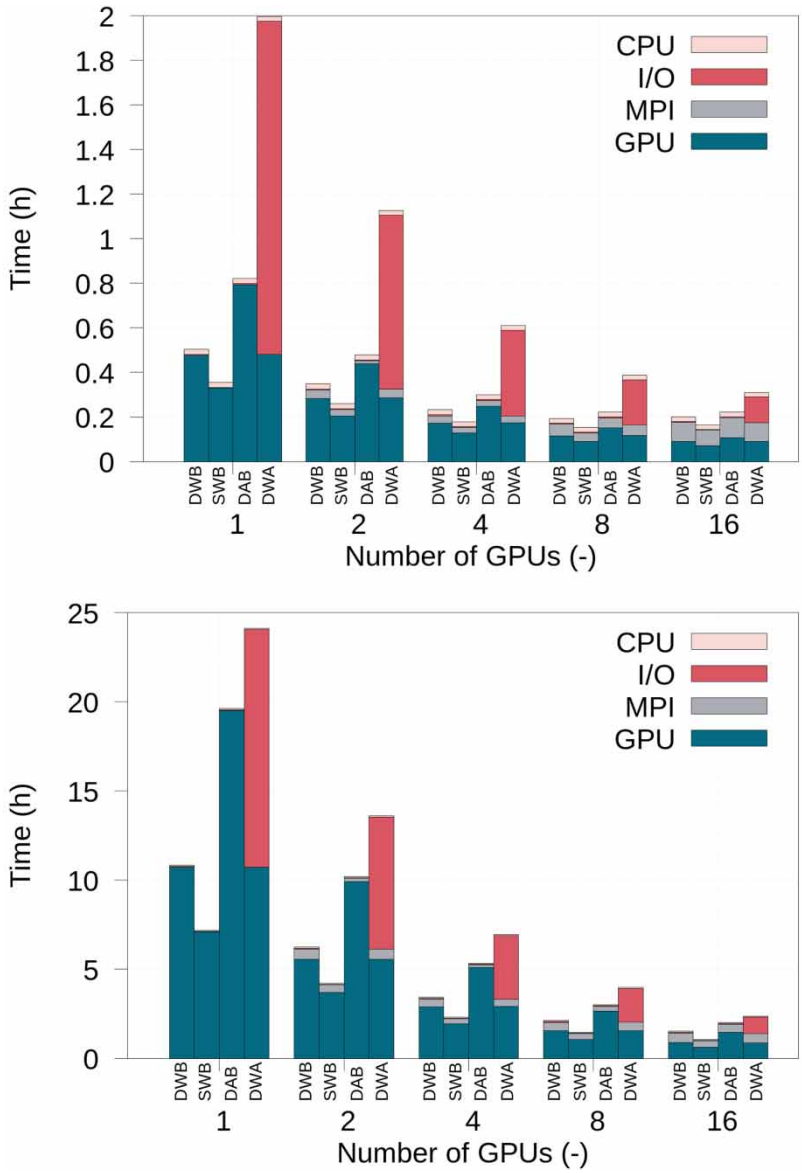

Figure 5 | Computational time of processes of Figure 2 for each implementation and each test case: 7.5 M cells (upper) and 68 M cells (lower).

responsible for the decline in scalability. As expected, there is no noticeable difference between GPU time for DWB and DWA implementations as the only difference between these cases is I/O. The SWB GPU time is faster than DWB since the graphic card is optimized for single precision. Furthermore, DAB is slower than DWB since the computations are done in all the domain instead of only on wet cells.

- For a given resolution, the CPU time is almost constant for all the simulations and implementations and is a trivial portion of the overall computational time. However, it slightly increases from the 7.5 M test case to the 68 M test case. This is consistent since the 68 M test case requires more iterations inside the temporal loop to compute the solution (the time step size is smaller) than the 7.5 M test case.
- The number of communications and the amount of data to be transferred among subdomains (MPI time) increase with the number of GPUs and can dominate the GPU time (e.g., in the case of 16 GPUs with 7.5 M cells).

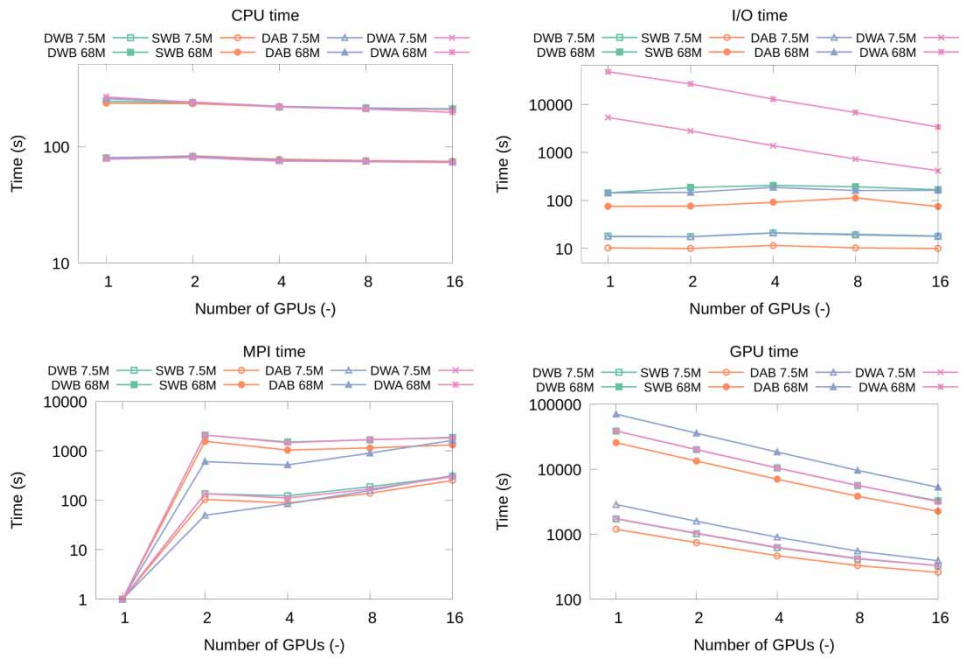

Figure 6 | Time consumed by each process for the 7.5 M test case (empty symbols) and the 68 M test case (filled symbols). CPU (upper left), I/O (upper right), MPI (lower left), and GPU (lower right).

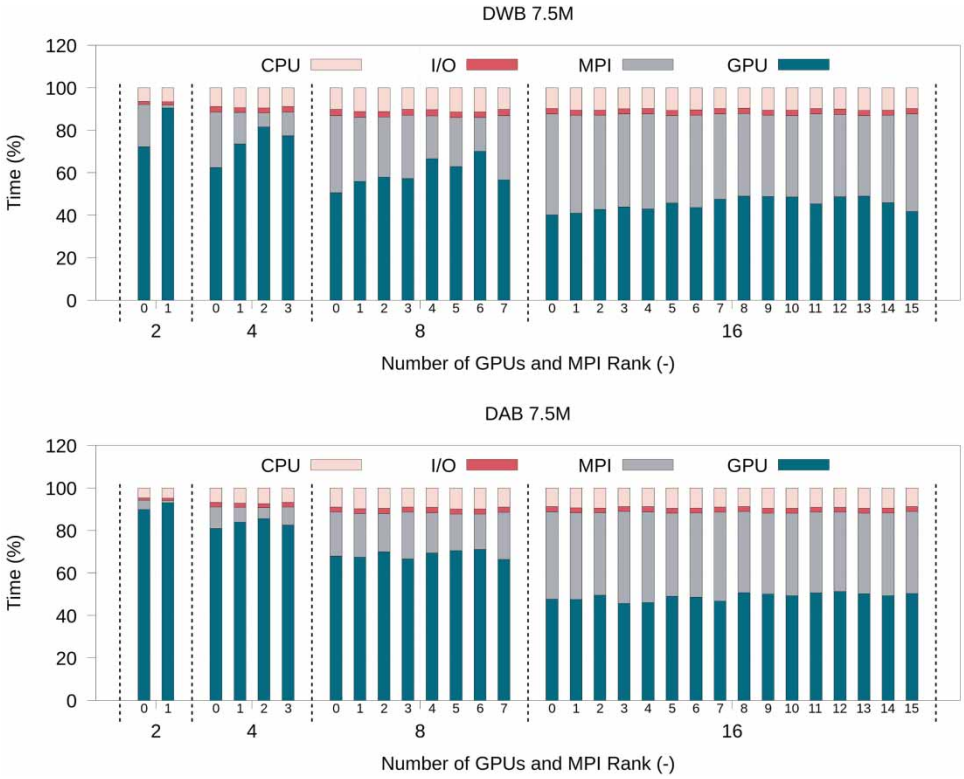

Figure 7 | MPI load for the 7.5 M test case for implementations DWB (upper) and DAB (lower).

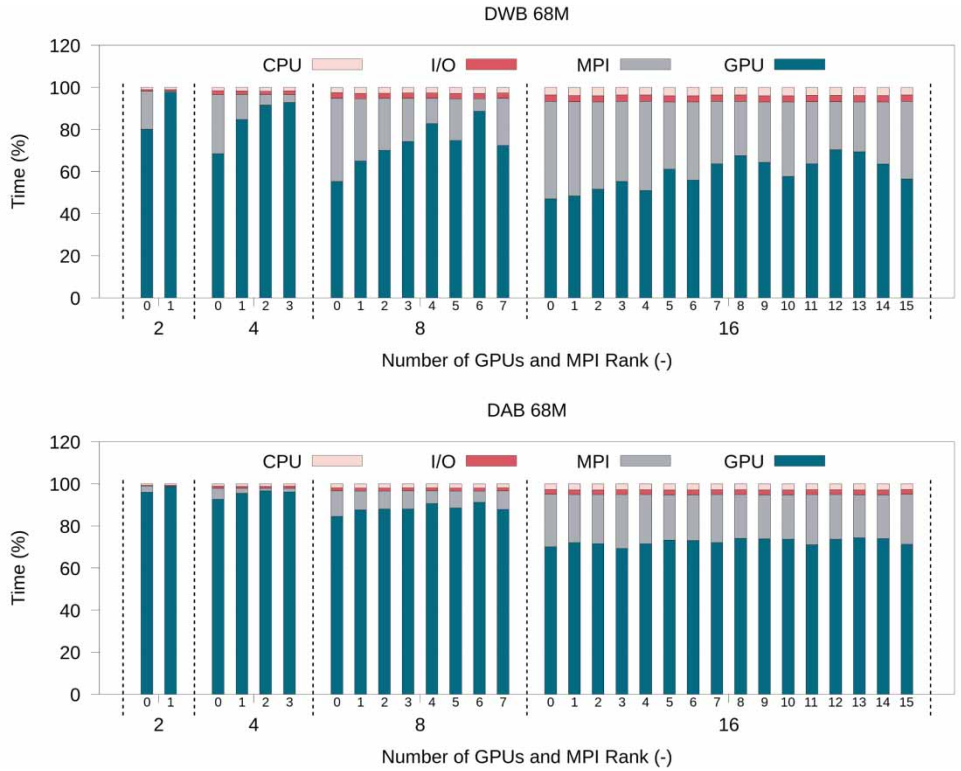

Figure 8 | MPI load for the 68 M test case for implementations DWB (upper) and DAB (lower).

Thus, MPI time is a bottleneck for a large number of GPUs. It is likely that this effect is exacerbated in the present model by the use of row-wise domain decomposition, which does not provide the minimum communication boundaries. The MPI time for SWB is slightly lower than that for the baseline DWB. The reason is purely a question of the amount of information (number of bytes) exchanged between the subdomains for single precision rather than double. MPI time in the DAB implementation is, on average, smaller than in the DWB approach. The reason for this is due to the MPI load imbalance that is behind the DWB implementation. However, as long as the number of GPUs increases, the difference becomes insignificant. The load imbalance is discussed below.

Load balancing among subdomains is plotted in Figures 7 and 8. As observed, the CPU and I/O times are almost constant among MPI ranks. However, an imbalance is detected for the GPU computation in the DWB implementation and is more noticeable in the 68 M test case. Consequently, this develops into an imbalance in the MPI time because the subdomains have to exchange information but cannot do so until the neighbor subdomain has finished its computation. Indeed, the process of sending/receiving the information to/from the neighboring subdomains acts as a synchronization, mimicking the *MPI_Barrier* statement. On the other hand, the DAB implementation makes a better balance between MPI tasks, resulting in an improvement in the scalability. However, this effect arises by computing over all cells instead of only wet cells, which is of dubious value.

CONCLUSIONS

Computing on HPC machines presents a number of challenges to the structure of our legacy water resources hydrodynamics models. Addressing these challenges argues in favor of entirely rewriting our codes and/or inventing new algorithms that take advantage of the peculiarities of HPC machines. As discussed in the 'The logistics of

data – a thought problem' section, the structure of how data are stored and communicated across computational nodes will control how effectively we can use massively parallel machines. Codes whose performance saturates and degrades as the number of processors increases will not survive in a world of ubiquitous HPC. To address these issues, we need to reconsider the fundamentals of our hydrodynamic models, including the underlying algorithms, data structure, real number precision, and I/O methods (see the 'How algorithm choice, data structure, and the implementation affects performance' section).

The results in the 'Results' section illustrate a key conundrum of HPC that has yet to be overcome: models are most scalable on large multi-GPU machines for a large number of computational elements, but such problems will take a correspondingly longer time to solve, e.g., in the example above the 68 M case has 9× the grid cells of the 7.5 M case and takes roughly 9× longer to solve. Furthermore, the comparison of the 7.5 M case and the 68 M case demonstrates the theory outlined in the 'Data structure' section that the relationship between the number of bounding elements per core and the overall number of elements per core will be a control on the scalability of communication, i.e., the 68 M case has roughly 9× the number of the elements per core as the 7.5 M case, but the number of boundary elements only increases by 3×. Hence, the scalability is better with the larger system where the ratio of boundary elements to computational elements is smaller. These results illustrate a danger of analyses focusing on scalability and neglecting the actual computational time associated with different algorithm choices. As shown with the computations of the 'all cells' versus 'wet cells', increasing the number of computations per core while holding the number of boundary communications fixed will always improve scalability – even as the model becomes slower.

A major challenge that we have *not* addressed is the conflict between the modular object-oriented code and the optimized HPC code. However, above we have described a range of challenges that require us to carefully design the structure of our models to take advantage of the peculiarities of massively parallel computing – an area that still has substantial unknowns. Today, there are simply no object-oriented approaches that can step in to do the hard work of ensuring communications are minimized and the data structure is scalable. Of course, it may be tomorrow that advances in object-oriented programming will make it the obvious choice for HPC, but for now, we encourage model developers to focus on the scalability of the data structure and communications rather than the reusability of objects.

We have entered an exciting new era for water resources hydrodynamics with the decline of serial computing and the rise of multi-CPU and multi-GPU high-performance computers. This new era demands a new focus for the model user community and research organizations to work together to make parallel water resources codes more broadly available and practical for users. This effort will require designing the next generation of codes with an eye toward long-term maintenance and commercialization rather than just building the next research stage. The HPC community is aware of the technology transfer problem and, for more than two decades, has been trying to address it through the development of open-source high-performance modules and libraries. However, these efforts have yet to make significant headway within the water resources community. Indeed, there is a critical open question for the community in how to pay for the technology transfer costs of moving codes from research to engineering/science usability, as well as the long-term maintenance costs for such codes. Today's HPC machines will define the structure of tomorrow's desktop workstations and the cloud computers that will be routinely used by water resources scientists and engineers for hydrodynamics modeling. One way or another, today's advances in HPC need to be translated to the broader community – our models take a long time to write, debug, and validate, so we need to be working today on the models that will run on tomorrow's computers.

ACKNOWLEDGEMENTS

The views expressed in this document are solely those of the authors and do not necessarily reflect those of the agencies named in this acknowledgement. The agencies do not endorse any products or commercial services mentioned in this publication. The research used resources of the Oak Ridge Leadership Computing Facility. Some of the

co-authors are employees of the Oak Ridge National Laboratory, managed by UT Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy. The publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript or allow others to do so, for U.S. Government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

FUNDING

This article was supported by the U.S. Air Force Numerical Weather Modeling Program to the Oak Ridge National Laboratory (M.M.-H., M.B.S., S.G., T.T.D., S.C.K., A.K., S.K.G., and K.J.E.), as well as by the U.S. Environmental Protection Agency (EPA) under Cooperative Agreement No. 83595001 awarded the University of Texas at Austin (E.M.-K. and B.R.H.). This article has not been formally reviewed by the EPA.

REFERENCES

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A. & Sorensen, D. 1999 *LAPACK Users' Guide*, 3rd edn. Society for Industrial and Applied Mathematics, Philadelphia, PA.

Babuska, I. & Guo, B. Q. 1992 The h, p and h-p version of the finite element method: basis theory and applications. *Advances in Engineering Software* **15**, 3–4.

Behzad, B., Sure Byna, P. & Snir, M. 2019 Optimizing I/O performance of HPC applications with autotuning. *ACM Trans. Parallel Comput.* **5** (4), 27.

Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D. & Whaley, R. C. 1997 *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics. Available from: https://epubs.siam.org/doi/book/10.1137/1.9780898719642.

Brodtkorb, A. R., Saetra, M. L. & Altinakar, M. 2012 Efficient shallow water simulations on GPUs: implementation, visualization, verification, and validation. *Computers & Fluids* **55**, 1–12.

Bruno, O. P., Cubillos, M. & Jimenez, E. 2019 Higher-order implicit-explicit multi-domain compressible Navier-Stokes solvers. *Journal of Computational Physics* **391**, 322–346.

Buahin, C. A. & Horsburgh, J. S. 2018 Advancing the Open Modeling Interface (OpenMI) for integrated water resources modeling. *Environmental Modelling & Software* **108**, 133–153.

Cardone-Noott, L., Rodriguez, B. & Bueno-Orovio, A. 2018 Strategies of data layout and cache writing for input-output optimization in high performance scientific computing: applications to the forward electrocardiographic problem. *PLoS ONE* **13** (8), 1–16.

Casulli, V. 1999 A semi-implicit finite difference method for non-hydrostatic, free-surface flows. *International Journal for Numerical Methods in Fluids* **30**, 425–440.

Casulli, V. & Zanolli, P. 2002 Semi-implicit numerical modeling of nonhydrostatic free-surface flows for environmental problems. *Mathematical and Computer Modelling* **36**, 1131–1149.

Dullo, T. T., Gangrade, S., Kalyanapu, A. J., Kao, S.-C., Ghafoor, S. K. & Evans, K. J. 2018 High-resolution modeling of Hurricane Harvey Flooding for Harris County, TX using a calibrated GPU-accelerated 2D flood model. In *American Geophysical Union 2018 Fall Meeting*, December 10–14, Washington, DC.

Evans, K. J., Archibald, R. K., Gardner, D. J., Norman, M. R., Taylor, M. A., Woodward, C. S. & Worley, P. H. 2019 Performance analysis of fully explicit and fully implicit solvers within a spectral element shallow-water atmosphere model. *The International Journal of High Performance Computing Applications* **33** (2), 268–284.

Feldman, M. 2019 *NVIDIA GPU Accelerators Get a Direct Pipe to Big Data*. *The Next Platform*. Stackhouse Publishing. Available from: https://www.nextplatform.com/2019/08/08/nvidia-gpu-accelerators-get-a-direct-pipe-to-big-data/ (accessed 13 August 2019).

Flamm, K. 2017 Has Moore's law been repealed? an economist's perspective. *Computing in Science and Engineering* **2**, 29–40.

Heroux, M. A., Bartlett, R. A., Howle, V. E., Hoekstra, R. J., Hu, J. J., Kolda, T. G., Lehoucq, R. B., Long, K. R., Pawlowski, R. P., Phipps, E. T., Salinger, A. G., Thornquist, H. K., Tuminaro, R. S., Willenbring, J. M., Williams, A. & Stanley, K. S. 2005 An overview of the Trilinos project. *ACM Transactions on Mathematical Software* **31** (3), 397–423.

Herzfeld, M. 2006 An alternative coordinate system for solving finite-difference ocean models. *Ocean Modelling* **14**, 174–196.

Houzeaux, G., Borrell, R., Cajas, J. C. & Vazquez, M. 2018 Extension of the parallel Sparse Matrix Vector Product (SpMV) for the implicit coupling of PDEs on non-matching meshes. *Computers & Fluids* **173**, 216–225.

Itu, L. M., Suciu, C., Moldoveanu, F. & Postelnicu, A. 2011 Comparison of single and double floating point precision performance for tesla architecture GPUs. *Bulletin of the Transilvania*

University of Brasov Series I: Engineering Sciences **4** (53), 131–138.

Kolev, T. & Dobrev, V. 2010 MFEM: Modular Finite Element Methods Library. Computer Software. https://github.com/mfem/mfem. USDOE. doi:10.11578/dc.20171025.1248.

Lacasta, A., Morales-Hernández, M., Murillo, J. & García-Navarro, P. 2014 An optimized GPU implementation of a 2D free surface simulation model on unstructured meshes. *Advances in Engineering Software* **78**, 1–15.

Liu, Q., Logan, J., Tian, Y., Abbasi, H., Podhorszki, N., Choi, J. Y., Klasky, S., Tchoua, R., Lofstead, J., Oldfield, R., Parashar, M., Samatova, N., Schwan, K., Shoshani, A., Wolf, M., Wu, K. & Yu, W. 2013 Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks. *Concurrency Computation: Practice and Experience* **26**, 1453–1473.

Morales-Hernández, M., Lacasta, A., Murillo, J., Brufau, P. & García-Navarro, P. A. 2015 Riemann coupled edge (RCE) 1D-2D finite volume inundation and solute transport model. *Environmental Earth Sciences* **74** (11), 7319–7335.

Murillo, J. & García-Navarro, P. 2010 Weak solutions for partial differential equations with source terms: application to the shallow water equations. *Journal of Computational Physics* **229** (11), 4327–4368.

Patankar, S. V. 1980 *Numerical Heat Transfer and Fluid Flow*. CRC Press, New York.

Smith, G. D. 1985 *Numerical Solution of Partial Differential Equations: Finite Difference Methods*, 3rd edn. Oxford University Press, Oxford.

Tallent, N. R., Adhianto, L. & Mellor-Crummey, J. M. 2010 Scalable identification of load imbalance in parallel executions using call path profiles. In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, LA, pp. 1–11.

Váňa, F., Düben, P., Lang, S., Palmer, T., Leutbecher, M., Salmond, D. & Carver, G. 2017 Single precision in weather forecasting models: an evaluation with the IFS. *Monthly Weather Review* **145** (2), 495–502.

Xia, X. & Liang, Q. 2018 A new efficient implicit scheme for discretising the stiff friction terms in the shallow water equations. *Advances in Water Resources* **117**, 87–97.

Yang, C., Cao, J. & Cai, X. 2010 A fully implicit domain decomposition algorithm for shallow water equations on the cubed-sphere. *SIAM Journal on Scientific Computing* **32** (1), 418–438.

First received 20 September 2019; accepted in revised form 16 January 2020. Available online 4 March 2020