Performance Improvement of a Two-dimensional Flood Simulation Application in Hybrid Computing Environments

Ryan Marshall*, Sheikh K. Ghafoor†, Alfred J. Kalyanapu‡, Mike Rogers§ and Tigstu T. Dullo¶
Department of Computer Science,*†§
Department of Civil and Environmental Engineering‡¶
Tennessee Technological University, Cookeville, TN 38505
Email: *rmarshall42@students.tntech.edu, †sghafoor@tntech.edu, ‡akalyanapu@tntech.edu,
§mrogers@tntech.edu, ¶ttdullo42@students.tntech.edu

***Abstract*—This paper presents a practical implementation of a 2D flood simulation model using hybrid distributed-parallel technologies including MPI, OpenMP, CUDA, and evaluations of its performance under various configurations that utilize these technologies. The main objective of this research work was to improve the computational performance of the flood simulation in a hybrid architecture. Modern desktops and small cluster systems owned by domain researchers are able to perform these simulations efficiently due to multicore and GPU computing devices, but lack the expertise needed to fully utilize software libraries designed to take advantage of the latest hardware. By leveraging knowledge of our experimentation environment, we were able to incorporate MPI and multiple GPU devices to improve performance over a single-process OpenMP version up to 18x, depending on the size of the input data. We discuss some observations that have significant effects on overall performance, including process-to-device mapping, communication strategies and data partitioning. The limitations of this work are discussed, and we propose some ideas to relieve or overcome such limitations in future work.**

I. Introduction

Computer models have been applied to simulate floods for over half a century [1]. Conventionally, those models use a one-dimensional (1D) dynamic wave approach, such as the Hydrologic Engineering Center River Analysis System (HEC-RAS) or MIKE 11, because these methods are easier to implement that two-dimensional (2D) models. Unfortunately, 1D models do not represent flood wave propagation correctly, especially in urban flood-prone areas, because 1D models cannot simulate the lateral diffusion and inaccuracies due to cross-section discretization [2], [3]. However, 2D models [4]–[6], eliminate the primary limitations of 1D models by enabling higher-order topographic representation and preferential flood pathways in the simulations [2], [7]–[9]. Therefore, the National Research Council in 2009 recommended that the Federal Emergency Management Agency (FEMA) promote "greater use of 2D hydraulic models" to model the floodplain topography, including preferential flood pathways and existing and planned structures [10].

Given the increased computational complexity inherent in 2D modeling [4], [6] over 1D modeling, building a 2D model that implements the practitioners specific requirements, such as accuracy constraints, data input requirements, and data output requirements, is not trivial. Such requirements can affect computational time, which can violate the overall time constraints defined by the practitioners. For flood risk and warning applications, the average modeling time should be less than an hour [10]. Furthermore, the necessary computational time of a flood model increases with the scale and resolution of the problem domain and complexity of the numerical model. Because the datasets for these type of problems can get prohibitively large for a single process, it is often beneficial to distribute these simulations over multiple processes and machines. Because this problem is well-suited for throughput processing, much of the computational work can be offloaded to a general purpose graphics processing unit (GPGPU) device. Typically, programming expertise and knowledge of parallel platforms and distributed environments are needed to build an efficient implementation that meets the time constraints, but some practitioners may not have access to that expertise or knowledge.

The current generation of desktop computers equipped with multicore and GPU devices provide enormous computing power that is inexpensive. For example, one can buy a desktop computer from a retailer that is capable of many teraflops. Unfortunately, a naïve implementation does not typically utilize all of a system's available resources. Our work attempts to efficiently map computations to a heterogeneous system's available resources to achieve performance significantly better than single processor systems. Programming expertise and knowledge of the parallel environment are necessary to take advantage of the complex operating systems, compilers, tools, libraries, and computational accelerators that make up these platforms. Practitioners can greatly benefit by using techniques and tools that can simplify using these platforms, because it will enable researchers to focus on the science rather than the tools themselves or the architecture of the execution environment.

The main contribution of this paper includes a performance improvement of a OpenMP-based flood simulation code that

2379-1896/17 $31.00 © 2017 IEEE
DOI 10.1109/CANDAR.2017.106

we reimplemented using hybrid distributed parallel technologies, including MPI+OpenMP and MPI+CUDA, and modified to support architecture-agnostic HPC environments. Our implementation can run under a combination of distributed and shared memory environments using multicore CPU and GPU technologies. We perform experimentation and give a performance evaluation of the practical implementation for a small hybrid cluster, with the expectations that our findings could apply to similar problems on systems of a larger and more diverse composition like pervasive grids [11]. Through our modifications, we identified several characteristics of the software organization that can be developed into a generalized solution for simulations over n-dimensional gridded data sets.

The rest of this paper is organized as follows. Section II describes related background research. Section III describes the implementation of the 2D model. Sections IV and V describe the experimental setup and present the experimental results. Finally, Section VI concludes the paper and describes future work.

II. Background

A variety of related research exists on the computational performance improvements of the 2D model via parallelization, the 2D numerical model and the 2D data model. In general, related works are narrow in scope and geared toward homogeneous environments.

A. Parallelizing the 2D Model

Recent research efforts have improved the computational performance of 2D models by implementing thread-based parallelization or incorporating GPUs. In [10], a desktop with an Intel Core 2 Duo processor required over 9 hours to compute 15 minutes of simulation time over a 624 x 1136 grid. Although the simulation application was redesigned to use CUDA on a Tesla C1060 GPU, and reported a speedup by a factor of 88, this application was hand optimized and not very portable because it employed a somewhat unique combination of hardware devices and software requirements. In [4], the authors report a speedup factor of 116 when running a simulation of their JFLOW 2D diffusion wave flood model versus a serialized version of the same program. RiverFLO-2D, an OpenMP implementation of the 2D numerical model, was shown to have a speedup of, approximately, a factor of 4.5 [12]. The authors of [13] developed a simulation application using C++, OpenGL and CUDA, that achieved 98% GPU utilization, and completed a 66 minute dam break simulation of the Malpasset Dam in 27 seconds. The application can produce a visualization of the simulation in near real time.

Building on these previous works, we made modifications to a existing implementation of a flood simulation application with the following goals in mind:

- achieve a significant performance gain over the standalone OpenMP and CUDA versions,
- embrace heterogeneous and hybrid computing environments,
- provide a structural codebase that can be reused and built upon in future works.

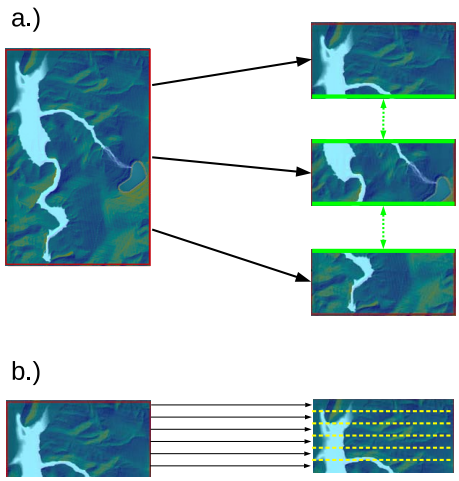

Fig. 1: Partitioning of a domain by a.) MPI processes, where edge data communication occurs between process neighbors, and each process by b.) threads in shared memory space.

The computation and memory access patterns for this application lend themselves to certain problem classes found in other domains, such as thermodynamics, biological systems and cellular automata. In general, these problems have a base layer containing static data, with one or more overlays containing data that changes over time (specific data for our implementation is described in Section II-C). The data is computed for the current time step based on adjacent cell values in one or more layers from the previous time step.

B. The 2D Numerical Model

The numerical algorithm used in this study is based on an existing version of the model, first described in [14]. The model uses a first-order accurate upwind finite difference scheme and solves the non-linear hyperbolic shallow water equations. These equations, also called the St. Venant Equations, are derived by integrating the horizontal momentum and continuity equations over depth. The equations are often referred to as depth-averaged or depth-integrated shallow water equations. The non-linear shallow water equations are presented as:

$$\frac{\delta h}{\delta t} + \frac{\delta (hu)}{\delta x} + \frac{\delta (hv)}{\delta y} = 0, \tag{1}$$

$$\frac{\delta u}{\delta t} + u\frac{\delta u}{\delta x} + v\frac{\delta u}{\delta y} + g\frac{\delta H}{\delta x} + gS_{fx} = 0, \tag{2}$$

$$\frac{\delta v}{\delta t} + u\frac{\delta v}{\delta x} + v\frac{\delta v}{\delta y} + g\frac{\delta H}{\delta y} + gS_{fy} = 0, \tag{3}$$

where the variable t is time in seconds, h is the water depth, H is the water surface elevation, u is the velocity in the x-direction, v is the velocity in the y-direction, t is the time, g is

the acceleration due to gravity, S_fx is the friction slope in the x-direction, and S_fy is the friction slope in the y-direction. This model uses the upwind finite difference numerical scheme with the governing equations 1-3, which yields non-oscillatory solutions through numerical diffusion [14], [15]. The model also uses a staggered grid computational stencil to define the domain with the water depth (h) at the center of the cell and u and v velocities on the cell edges. The model propagates the simulation by constraining the time step using the Courant condition. Because these equations can be applied to every cell in the grid, the computation of a cell value depends on neighboring values and the state of the entire grid at time t depends on the state of the same grid at time $t-1$, this model can be implemented as a cellular automata problem; we discuss the implementation details in Section III.

C. Data Model

The input data requirements for the numerical algorithm include topography, surface roughness, flow hydrograph and its geographic source location (e.g., gauging station). The data model represents terrain with the Digital Elevation Model (DEM). DEM files are available in many different formats from different sources. The formats specified by USGS have been used in the present work [16]. The numerical algorithm computes the solution on a uniform grid, to take advantage of the use of downloadable DEM data from sources such as The National Map [17] or Google Maps API [18].

The commonly used Manning's n coefficient [19] models the surface roughness data. A single Manning's n value in 2D modeling applications [20] typically can represent the entire domain, but our model incorporates a spatial variation of Manning's n values using an appropriate raster dataset (e.g., National Land Cover Dataset from United States Geological Survey).

The flow hydrograph is an input dataset that can be developed from a hydrological model, dam break model, or empirical observations. We used hydrographs in our simulations that describe discharge from one or more source locations in cubic feet per second over simulation time in hours.

III. Implementation

To maintain consistency with the numerical model, we planned our implementation to create a set of 2D grids, H, U, V, to store (for each cell in the grid) depth, velocity in the x-direction, velocity in the y-direction, respectively. Additionally, we need a second copy of this set to store the values from the previous time step, so the full set includes H_a, U_a, V_a, H_b, U_b, V_b, and D for the terrain elevation data. To reduce the number of expensive copy operations, we maintain pointers to each grid, and at the end of each iteration, we swap the pointers between the a and b sets so the state data at the end of the current time step effectively becomes the data for the previous time step. The following limitations apply to our implementation:

- For most cases, we defer to the MPI runtime environment to determine process distribution over compute nodes. Because we use a heterogeneous system, there are some situations where we will need to manually map a process to a specific node in order to use a compute device not found on other nodes.
- We assume all partitions at time step t can potentially contain at least one cell whose value for any layer is not equal to the corresponding cell's value at time step $t-1$.
- We use block row partitioning for MPI and block cyclic threading for OpenMP with the understanding that more efficient schemes possibly exist, but a full comparison and analysis is beyond the scope of this work.

A. Computational Model

Equations 1-3 provide the basis for the computational model. We used the following equations to calculate S_fx and S_fy, based on Manning's equation:

$$S_{fx} = n^2(u_{ij})\sqrt{\frac{u^2{}_{ij} + \bar{v}^2_{ij}}{h_{ij} + h_{(i+1)j}}} \tag{4}$$

$$S_{fy} = n^2(v_{ij})\sqrt{\frac{v^2{}_{ij} + \bar{u}^2_{ij}}{h_{ij} + h_{i(j+1)}}} \tag{5}$$

$$\bar{u}_{ij} = \frac{u_{ij} + u_{i(j-1)} + u_{(i-1)(j-1)} + u_{(i-1)j}}{4} \tag{6}$$

$$\bar{v}_{ij} = \frac{v_{ij} + v_{i(j+1)} + v_{(i+1)(j+1)} + v_{(i+1)j}}{4} \tag{7}$$

In Equations 4 and 5, n is the Manning coefficient. For purposes of this study, we used a constant n of 0.035.

Temporal dependencies for each iteration include the interpolated flow that is derived from the input hydrograph using the following interpolation function:

$$h(t_r, f_r, s_r) = f_r + (s_r - t_{(r-1)})\frac{f_r - f_{(r-1)}}{t_r - t_{(r-1)}} \tag{8}$$

We applied this function for the time step r, starting from a initial simulation time where $r = 1$ and s_r is the accumulated simulation time at time step r. Spatial dependencies for a grid cell include specific cell neighbors of the corresponding cell from the previous time step. By examining Equations 4-7, we can identify the specific dependencies. When the domain is partitioned, we introduce a layer of complexity; if a neighboring cell is on the edge of a distributed memory partition, the cell data must be communicated to the neighboring partition, as shown in Figure 1.

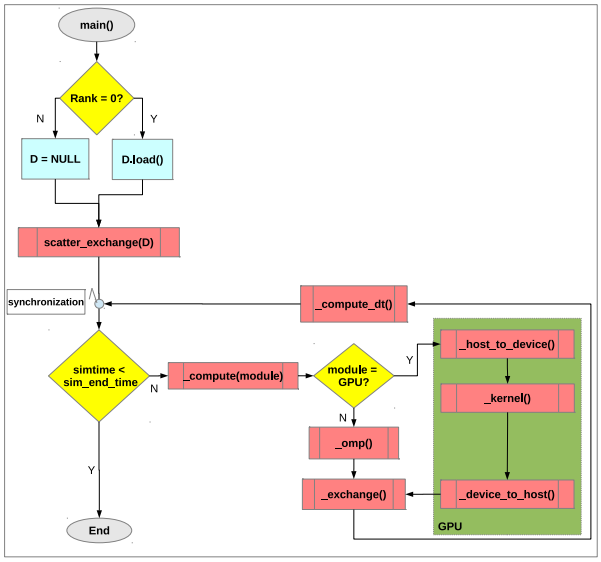

Fig. 2: Execution control flow for our MPI+CUDA implementation. The function $scatter\_exchange()$ loads the input DEM with Process 0, performs a MPI scatter, and exchanges edge data. The $\_omp()$ function or $\_kernel()$ performs the core computations, depending on the module selected by the user.

B. Implementing MPI+OpenMP

Figure 2 illustrates the workflow in general. The simulation begins by reading the configuration and input files from the root process, then calculating the partition sizes based on the requested number of processes. Next, the root process determines which partitions will contain source cells, and performs the translation of source locations at global scope into local coordinates. Then, the root process distributes the D grid plus the localized configuration data over each partition. Each process will then initialize the H, U, V, grids based on the localized dimension data, and set the initial state. The b set contains the initial state at $t = 0$, and the a set will compute the state for time $t = 1$.

The simulation loop begins with an initial value $simtime$ for the simulation time, and control will remain in the loop until the accumulated value of $simtime$ equals or slightly exceeds the simulation end time given in the configuration file. The initial time step is set by the user and stored in the value dt, which can be configured by the user to be fixed or variable. After each iteration, dt is added to $simtime$, and in the case of variable dt, a new value is calculated by $\_compute\_dt()$ for the next time step.

Variable dt can save time by increasing the dt value during periods where the water flows at a lower rate (where the flood extent undergoes small changes) and decreasing dt when water flow is high. We compute dt based on the following equations:

$$\Delta_x = \frac{dx}{(U_{max} + \sqrt{g(hx_{max} + \epsilon)})}, \tag{9}$$

$$\Delta_y = \frac{dy}{(V_{max} + \sqrt{g(hy_{max} + \epsilon)})}, \tag{10}$$

$$dt = \lceil Min(\Delta_x, \Delta_y) \rceil \tag{11}$$

Equation 9 finds U_{max}, the maximum value in the U grid, and hx_{max}, the value in the H grid located at the same index where U_{max} was found, and uses these values to compute Δ_x . Similarly, Equation 10 computes Δ_y by finding V_{max} and hy_{max}, and Equation 11 will set dt to the smaller of the two.

The compute function is called, then edge data is exchanged among the partitions as shown in Figure 1a, and the partition data is transferred from the host to the device. In our implementation, the $\_omp()$ function is a wrapper for a OpenMP $parallel for$ construct, which splits work among the requested number of threads as illustrated in Figure 1b. Synchronization of processes and threads occur after dt for the next time step is calculated.

C. Implementing MPI+CUDA

The MPI+CUDA version operates in a similar manner, excluding the OpenMP $parallel for$ and calling a kernel instead. Multiple processes can be allocated to the same device, if the data is sufficiently small. This strategy can be especially useful when the host-device link can support multiple communication channels.

The compute function invokes the CUDA API to execute the computational kernel that processes the data that was loaded on the device during the initial host-to-device transfer. After each iteration, the CUDA kernel will check to determine if a print operation is needed. If so, it will initiate a device-to-host transfer of the entire partition, and the synchronization point will be reached.

IV. Experimental Setup

The experimental setup was designed to measure key performance metrics for specific test cases that we believe will illustrate general performance improvements. In the following subsections, we present these metrics and test cases, and describe the hardware and runtime configurations used in our experiments.

A. Key Performance Metrics

1) Megacells per second: (Mc/s or M) is the metric we use to compare the computational efficiency across all implementations. Mc/s is defined as the number of cells (in millions) divided by the computational time required to process all cells. Higher Mc/s values correspond to higher performances, since more cells are computed in the same time reference window. This measure allows capturing the average speed during the

computation and generating early performance projections early in the simulation run. In the flood research community, Mc/s is commonly used as a measure of a simulation's efficiency. We compute Mc/s according to Equation 12, where I is the number of iterations, rc is the total number of cells in the grid, and t is the runtime in seconds.

$$M = \frac{Irc}{t(10^{-6})} \tag{12}$$

2) I/O seconds: This metric is the time that is spent specifically on data reading/writing to disk and MPI communications. While disk I/O is a constant overhead, MPI messages may vary in size and frequency, depending on the runtime configurations.

B. Simulation Test Cases

We will use three different simulation data sets of varying size and complexity.

1) Taum Sauk: The first test case is based on a dam break that occurred to the Taum Sauk Dam in Reynolds County, Missouri in December 2005. It is a twin reservoir system, designed to produce electricity during peak periods, by discharging water from the upper reservoir on to the lower reservoir. Water is pumped back to the upper reservoir during off-peak periods. The upper reservoir built on the Proffit Mountain is approximately 760 feet above the floodplain of the East Fork Black River, with a storage capacity of 1.5 billion gallons. The upper reservoir failed on December 14th, 2005, at a 680-foot wide section on the northwest side of the Proffit Mountain, releasing the entire storage into East Fork Black River floodplain, through Johnsons Shut-Ins State Park, and into a lower storage reservoir within 25 minutes. In this study, this dam break scenario is simulated and a comparative analysis of the computational times reported by all the various implementations is performed.

The input data can be represented by a grid consisting of 1136 rows and 624 columns. The total simulation time is one hour. Using dynamically generated time steps that are 0.013 seconds on average, the simulation takes 290,453 iterations to complete

2) Cairo: The second test case simulates a flood event that occurred in 2011, during the weeks of massive flooding on the Mississippi River watershed. The levee system along the Birds Point-New Madrid Floodway was in danger of failure, and threatened to flood the town of Cairo, IL. Beginning on May 2, 2011, a 2-mile section of the levee was detonated by the U.S. Army Corps of Engineers, causing over 100,000 acres of farmland within the watershed to become inundated [21].

The base topographical layer for our Cairo simulations is represented by an array containing a total of 1795 x 1555 cells, which we populate with elevation data to create the terrain. The simulation takes 80,384 iterations to compute 14,400 seconds of simulation time.

3) Large Random: The final test case is a large-scale simulation over randomly generated terrain. Random generation of the topography was performed using a variation of the Diamond-Square method, a inflow location representing the origin of the flood was chosen at random. The purpose of this test case is to observe the performance of the flood simulation when using input data requiring memory allocation near the capacity of certain GPU devices. By using a grid of 16387 x 16387 cells, the memory footprint for all 7 layers is near 7.5GB when using a 4-byte numerical type.

TABLE I: MPI+OpenMP Configurations

Run	OMP Threads	MPI Procs	Node(s)
R1	40	1	N3
R2	40	2	N3, N4
R3	40	4	N1-N4
R4	20	8	N1-N4

C. Hardware Specifications

For our simulation runs, we used a heterogeneous HPC cluster consisting of 4 nodes, each having Intel Xeon E5-2680 processors in 2 sockets, 10 cores each (with hyper-threading), for a maximum of 160 dedicated CPU processes across all 4 nodes. The heterogeneity is introduced with the GPGPU arrangement of each node, where node N1 is equipped with a single Nvidia Tesla K40M (k40) and node N2 with two K20M (k20) GPUs. The GPUs on nodes N3 and N4 will not be used for these simulations.

D. Runtime Configurations

Table I lists a set of simulation runs we will perform using the Taum Sauk and Cairo data sets. Run R1 is the baseline configuration, using one MPI process on a single node and 40 OpenMP threads. Runs R2 and R3 distribute the workload across two and four nodes, respectively, each using 40 OpenMP threads. Run R4 also uses 4 nodes, but uses two processes on each node. To maintain a maximum of 40 threads per node, we allow 20 OpenMP threads for each process. Using more than 40 OpenMP threads per node on our test system results in performance degradation because it exceeds the number of available CPU cores, so some threads will always be forced to wait until an active thread finishes its work and a core becomes available.

Table II lists a set of configurations where Runs R5-R24 will use the Taum Sauk and Cairo data sets and Runs R25-R27 will use the Large Random data set. Runs R5-R11 divide the work between a number of processes on the same node; space for each process is allocated on the device, so multiple kernels allocated to separate work groups can run concurrently. Communication over PCIe supports full duplex transfers, but true concurrency is not guaranteed. A sufficiently large number of processes can create a communication bottleneck that overshadows performance gains made by effective utilization of the computational cores. Runs R12-R18 are similar to R5-R11, but they are performed on node N2, which is equipped with two k20 GPUs. Runs R19-R24 utilize two nodes and all GPU devices available across the nodes. Since there are three devices being used, we test the number of processes in

TABLE II: MPI+OpenMP+GPU Configurations

Run	MPI Procs	Devices	Node(s)
R5	2	k40	N1
R6	4	k40	N1
R7	6	k40	N1
R8	8	k40	N1
R9	12	k40	N1
R10	16	k40	N1
R11	18	k40	N1
R12	2	k20 x 2	N2
R13	4	k20 x 2	N2
R14	6	k20 x 2	N2
R15	8	k20 x 2	N2
R16	12	k20 x 2	N2
R17	16	k20 x 2	N2
R18	18	k20 x 2	N2
R19	3	k20, k20, k40	N1, N2
R20	6	k20, k20, k40	N1, N2
R21	12	k20, k20, k40	N1, N2
R22	18	k20, k20, k40	N1, N2
R23	24	k20, k20, k40	N1, N2
R24	30	k20, k20, k40	N1, N2
R25	2	k20, k40	N1, N2
R26	8	k20 x 2	N2
R27	8	k20, k40	N1, N2

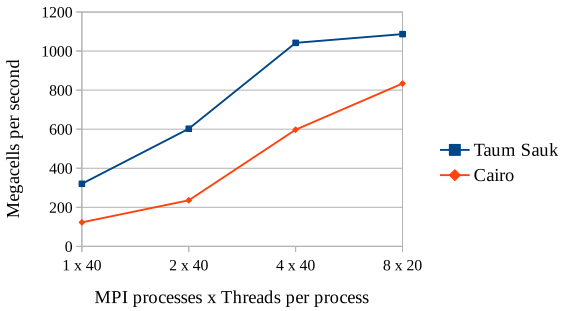

Fig. 3: Comparison of MPI+OpenMP results for two simulations using an increasing number of processes and process mappings (MPI Processes x OMP threads).

increasing multiples of three. Runs R25-R27 utilize one or two nodes, each with a different selection of process distribution and devices used. The objective is to test the process and device distribution when the workload is near the limitations of the devices.

V. Results and Analysis

A. MPI+OpenMP simulations of Taum Sauk and Cairo

Figure 3 shows the results in Mc/s of runs R1-R4 for both the Taum Sauk and Cairo simulations. Performance from 1 to 4 processes improves in both cases in a near linear manner,

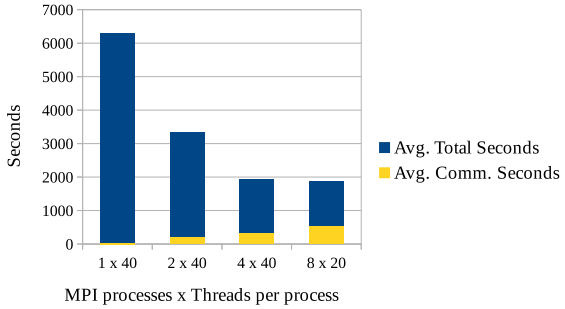

Fig. 4: Comparison of MPI+OpenMP communication time versus total time for the Taum Sauk simulation using an increasing number of processes and process mappings (MPI Processes x OMP threads).

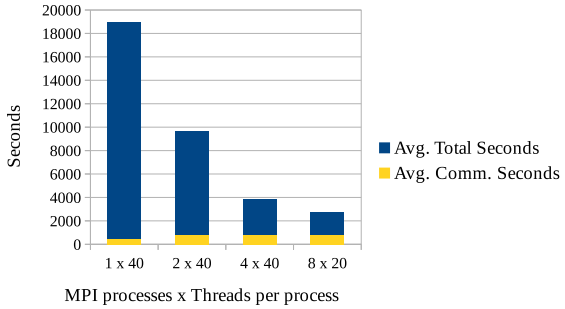

Fig. 5: Comparison of MPI+OpenMP communication time versus total time for the Cairo simulation using an increasing number of processes and process mappings (MPI Processes x OMP threads).

but the performance for the Cairo simulation is about half the performance of Taum Sauk. The row size and overall grid size for Taum Sauk is smaller, so the negative impact of communication overhead is expected to be seen at a lower number of processes than the Cairo simulation, which we see in the results for 8 processes. The performance improvement for Taum Sauk is much smaller when using 8 processes, while the improvement for Cairo remains on a linear path. As the number of processes increase beyond 8, the performance of the Cairo simulation should continue to improve while the performance for the Taum Sauk simulation should approach a constant value or possibly degrade, since there is no added performance benefit when distributing increasingly smaller workloads.

The main reason for the lack of expected increase in the Taum Sauk performance involves MPI communication overhead as the partition sizes become smaller. We observe two very different patterns in Figures 4 and 5. In the Taum Sauk simulations, the communication time is significant under 4 and 8 processes, while it is less significant in the Cairo simulations, and nearly constant from 2 to 8 processes. The communication time for Taum Sauk increases with the number

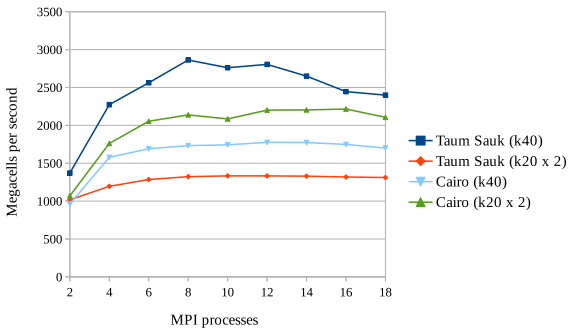

Fig. 6: Comparison of MPI+CUDA runs of Taum Sauk and Cairo simulations using processes allocated to a single GPU versus processes balanced across 2 GPUs.

of processes until it is nearly half the total run time using 8 processes. The number of messages passed in the Taum Sauk simulations is the same as the number of messages passed in Cairo, but the size of the messages in Cairo is over twice the size of the Taum Sauk messages, since the row size in the Taum Sauk grid is 624 and the rows for the Cairo grid have 1555 elements. In Figure 5, the communication time for Cairo shows a less dramatic increase, but we can also expect at some point that the communication overhead will be detrimental to this case, but likely at a higher number of processes than Taum Sauk.

B. MPI+CUDA simulations of Taum Sauk and Cairo

Figure 6 shows the impact of process-to-device mapping of simulations using two different datasets and two generalized mappings, based on runs R5-R15. Performance is improved in all cases while increasing to 8 processes. The most significant improvement is seen when using a single k40. Above 8 processes, we observe performance degradation caused by a bottleneck of host-device transfers over PCIe. Once the number of processes allocated to a GPU device becomes sufficiently large, kernels will be forced to wait for the transfers of other kernels to complete before their own transfer can be serviced.

The results for Runs R5-R18 are shown in Figure 6. The performance of Taum Sauk is dramatically better on a single k40 device than on the two k20 devices. Cairo performs slightly better on the k20 devices than on a single k40. The most efficient number of processes appears to be either 8 or 12 in all cases.

C. MPI+CUDA heterogeneous simulations with three devices

Runs R19-R24 investigate the performance of Taum Sauk and Cairo using three available GPU devices across two nodes under various process-to-device mappings. In Section III, we noted the MPI environment will govern the process to node allocation for most of our simulation runs. For this set of experiments, we will be manually assigning processes to nodes to have more control over the number of processes that can use a given device.

By default, our system will allocate processes in order for each host or node in the list. If the number of processes is greater than the number of hosts, the system will cycle back to the first host on the list and continue this sequence until all processes are allocated. Figure 7 illustrates the communication patterns that result from the combination of our partitioning strategy and the default ”Cyclical” process allocation when using six processes over two hosts n0 and n1, where n0 has a single k40 GPU (n0d0) and n1 has two k20 GPUs (n0d0, n0d1). In our partitioning strategy, communications of edge data will occur between processes of rank r and rank $r+1$, except for the process with the highest rank, and rank r and rank $r-1$ except for the process with the rank of 0. A practical illustration of this partitioning is given in Figure 1, where edge data (highlighted in green) is exchanged after each time step.

We can manually assign processes to hosts in a way that complements our partitioning strategy, as shown in the ”Adjacent” mapping in Figure 7. Under this mapping, we expect to see higher performance in many cases, because more intranode MPI communications will occur than would occur under the default mapping. We assume intranode communications in our test environment are at least as fast as internode communications, because more messages can be passed over the system bus without needing to travel over the full networking stack to another physical node. The difference may be negligible in some cases, but we expect to identify other cases where it would be advantageous to manually define which processes are assigned to each node.

Figure 8 shows the performance of Runs R19-R24 for both Taum Sauk and Cairo using both the cyclical and adjacent mapping strategies. For Cairo, the differences are relatively small and favor adjacent in most cases, while Taum Sauk results show a greater disparity as the number of processes increase to 30. Figure 9 gives some insight, as we see the communication times for Cairo having less variance than Taum Sauk, which shows greater increases in communication overhead from 12 to 30 processes. This is consistent with the results for MPI+OpenMP given in Figures 4 and 5 that also reveal an increasing communication overhead on the smaller grid.

D. MPI+CUDA heterogeneous simulations of a large random grid

Experimentation with the large grid shows different process-to-device mappings have a significant impact on performance, as shown in Figure 10. A heterogeneous HPC environment is expected to have different GPU devices available, some may be more powerful, or otherwise better suited to the problem instance than others. Because of its intentionally large size, attempts to run these simulations on a single k20 GPU will cause the CUDA runtime to generate out-of-memory errors.

The k40 has a higher capacity, so it can successfully run this dataset on its own. In Runs R25-R27, we first distribute the data into 2 processes and run on 2 different GPUs.

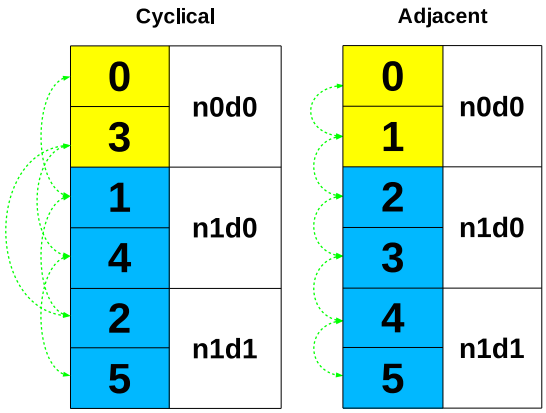

Fig. 7: A cyclical distribution (left) and adjacent distribution (right) of six processes demonstrates the difference in communication patterns based on how processes are mapped. The cyclical distribution does not utilize any intranode links while the adjacent distribution utilizes four.

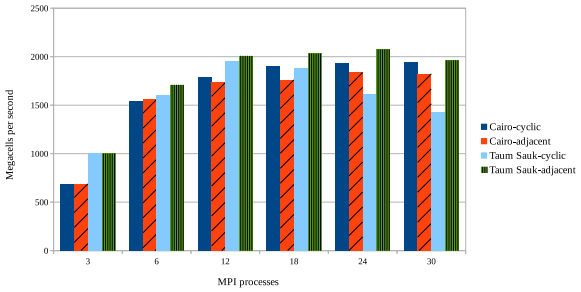

Fig. 8: Comparison of MPI+CUDA runs using a k40 and two k20 GPUs across two physical nodes using cyclic versus adjacent process distribution.

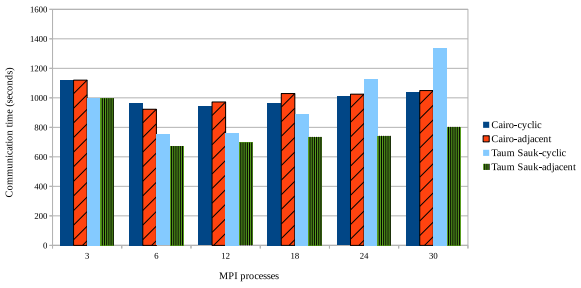

Fig. 9: Comparison of communication times for MPI+CUDA runs using a k40 and two k20 GPUs across two physical nodes using cyclic versus adjacent process distribution.

Then we split into 8 processes and run on the pair of k20 devices, then a single k40 plus a single k20. Performance is improved dramatically from Runs R25 to R27, although the number of processes remain the same and the 8 processes in Run R25 require no ethernet communications between the processes. The k40 is slightly faster than the k20, with a higher cache availability. Additionally, the k40 has higher throughput speeds, around 2Gbps versus 1.6Gbps for the k20.

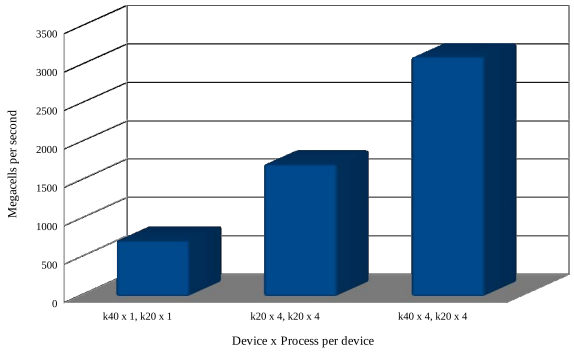

Fig. 10: Comparison of MPI+CUDA simulation runs using the Large Random dataset under different process-to-device mappings.

VI. Conclusion and Future Work

We have shown in all cases that decomposing the problem and distributing the data across multiple processes will result in higher performance than running the same simulation with a single process. Noting that each physical node is equipped with identical CPUs, Figure 11 gives a summary of the performance in Mc/s of all runs, where R1 represents the performance of the original OpenMP code, showing around a 9x improvement at R8 for Taum Sauk and around a 18x improvement for Cairo at R17. For the OpenMP implementation in Runs R1-R4, we demonstrated how improvement follows a linear path as the number of processes increase (although there is a limitation to these performance gains, based on the size of the simulation domain and communication patterns). In Runs R5-R18, we illustrate how knowledge of the underlying hardware can be a significant advantage, as it is possible in some cases to give a single device the same workload as two less powerful devices and achieve better results.

Runs R19-R24 highlighted how knowledge of process mapping on a system and software-defined partitioning can be used to increase performance if they have complementary strategies. With Runs R25-R27 we found that knowledge of the underlying hardware devices is essential for developing software meant for large scale grid simulations, since data transfer operations will fail when the data is too large to fit in device memory. Future exploration includes developing a set of generalized data structures that support a number of different partitioning strategies, runtime configuration parameters that match a selected partitioning strategy with a complementary mapping strategy, and a metacomputing API that is simple and user-friendly, to allow domain scientists access to more efficient hardware without the expertise that is required today.

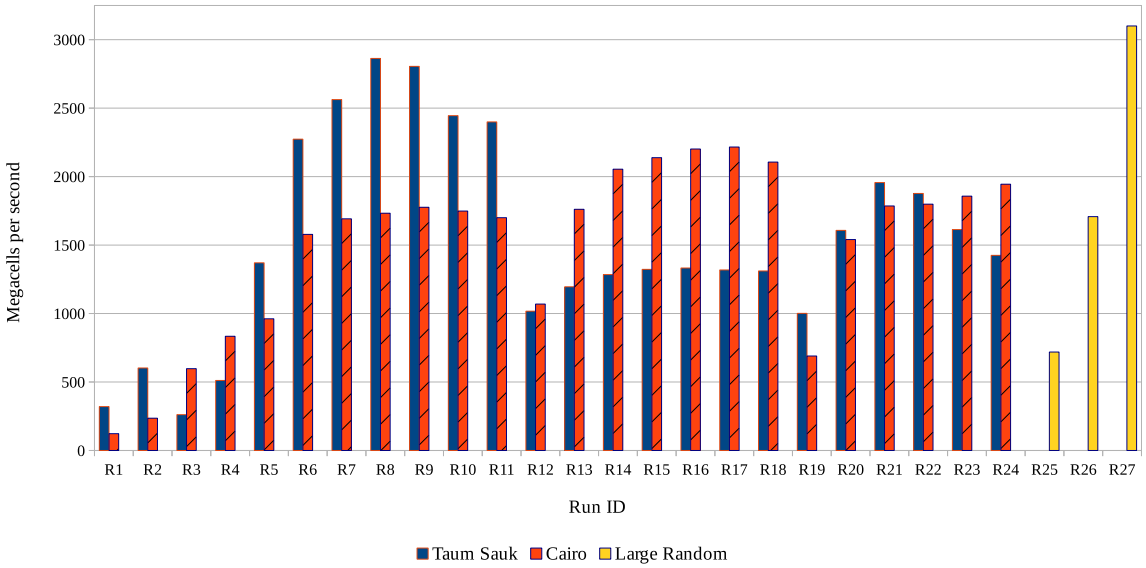

Fig. 11: Performance in Mc/s for each of the runs listed in Tables I and II. Runs R1-R4 used MPI+OpenMP, R5-R11 used MPI+CUDA on a single GPU, R12-R18 used two GPUs and R19-R24 used three GPUs. Runs R1-R24 used the Taum Sauk and Cairo datasets, while Runs R25-R27 used only the Large Random dataset.

REFERENCES

[1] N. H. Crawford, R. K. Linsley, S. U. D. of Civil Engineering, and F. Linsley Jr., *Digital simulation in hydrology: Stanford watershed model IV*, ser. Technical report (Stanford University. Dept. of Civil Engineering). Stanford, CA: Dept. of Civil Engineering, Stanford University, 1966. [Online]. Available: http://books.google.com/books?id=EPrHW6rK5_8C

[2] P. D. Bates and A. P. J. D. Roo, "A simple raster-based model for flood inundation simulation," *Journal of Hydrology*, vol. 236, no. 12, pp. 54–77, 2000. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002216940000278X

[3] V. Tayefi, S. N. Lane, R. J. Hardy, and D. Yu, "A comparison of one- and two-dimensional approaches to modelling flood inundation over complex upland floodplains," *Hydrological Processes*, vol. 21, no. 23, pp. 3190–3202, 2007. [Online]. Available: http://dx.doi.org/10.1002/hyp.6523

[4] A. Crossley, R. Lamb, and S. Waller, "Fast solution of the Shallow Water Equations using GPU technology," pp. 1–6, 2010.

[5] K. Marks and P. Bates, "Integration of high-resolution topographic data with floodplain flow models," *Hydrological Processes*, vol. 14, no. 11-12, pp. 2109–2122, 2000. [Online]. Available: http://dx.doi.org/10.1002/1099-1085(20000815/30)14:11/12%3C2109::AID-HYP58%3E3.0.CO;2-1

[6] P. G. Samuels, "Cross-section location in 1-D models," in *Proc., Int. Conf. on River Flood Hydraulics*. Wiley Chichester, UK, 1990, pp. 339–350.

[7] P. D. Bates, M. G. Anderson, L. Baird, D. E. Walling, and D. Simm, "Modelling floodplain flows using a two-dimensional finite element model," *Earth Surface Processes and Landforms*, vol. 17, no. 6, pp. 575–588, 1992.

[8] N. R. Council, *Mapping the Zone: Improving Flood Map Accuracy*. Washington, D.C.: National Academies Press, 2009.

[9] P. García-Navarro, P. Brufau, J. Burguete, and J. Murillo, "The shallow water equations : An example of hyperbolic system," *Monografías de la Real Academia de Ciencias de Zaragoza*, pp. 89–119, 2008.

[10] A. J. Kalyanapu, S. Shankar, E. R. Pardyjak, D. R. Judi, and S. J. Burian, "Assessment of GPU computational enhancement to a 2D flood model," *Environmental Modelling & Software*, vol. 26, no. 8, pp. 1009–1016, aug 2011. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1364815211000582

[11] C. Bertolli, G. Mencagli, and M. Vanneschi, "Adaptivity in risk and emergency management applications on pervasive grids," *I-SPAN 2009 - The 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, pp. 550–555, 2009.

[12] N. Gonzalez-Ramirez, "Simulating Flood Propagation in Urban Areas using a Two-Dimensional Numerical Model," 2010.

[13] S. S. Alfred J Kalyanapu, Sheikh K Ghafoor, Ryan J Marshall, Tigstu T Dullo, David R Judi, "Benchmark Exercise for Comparing the Computational Performance of Two-Dimensional Flood Models in CPU, Multi-CPU, and GPU Frameworks," in *World Environmental and Water Resources Congress 2014*, ch. 133, pp. 1322–1331. [Online]. Available: http://ascelibrary.org/doi/abs/10.1061/9780784413548.133

[14] S. V. Patankar, *Numerical heat transfer and fluid flow*. Taylor & Francis, 1980.

[15] J. H. Ferziger and M. Perić, *Computational methods for fluid dynamics*. Springer Berlin, 1996, vol. 3.

[16] USGS, "National Mapping Program Technical Instructions: Standards for Digital Elevation Models," U.S. Geological Survey National Mapping Division, Reston, Virginia, Tech. Rep., 1998.

[17] "The National Map: Topographic Maps for the 21st Century," Reston, VA, Tech. Rep., 2002. [Online]. Available: http://pubs.er.usgs.gov/publication/fs01802

[18] "The google geocoding API - google maps API web services google developers." [Online]. Available: https://developers.google.com/maps/documentation/geocoding/

[19] R. Manning, J. P. Griffith, T. F. Pigot, and L. F. Vernon-Harcourt, *On the flow of water in open channels and pipes*, 1890.

[20] A. J. Kalyanapu, S. J. Burian, and T. N. McPherson, "Effect of land use-based surface roughness on hydrologic model output," *Journal of Spatial Hydrology*, vol. 9, no. 2, pp. 51–71, 2009. [Online]. Available: http://spatialhydrology.net/index.php/JOSH/article/view/84

[21] T. A. Koenig and R. R. Holmes, "Documenting the stages and streamflows associated with the 2011 activation of the New Madrid Floodway, Missouri," 2013.