IFM: A Scalable High Resolution Flood Modeling Framework

Swati Singhal[1], Sandhya Aneja[2], Frank Liu[1], Lucas Villa Real[1], and Thomas George[1]

[1] IBM Research
[2] Universiti Brunei Darussalam, Brunei Darussalam

Abstract. Accurate and timely flood forecasts are essential for effective management of flood disasters, which has become increasingly frequent over the last decade. Obtaining such forecasts requires high resolution integrated weather and flood models with computational costs optimized to provide sufficient lead time. Existing overland flood modeling software packages do not readily scale to topography grids of large size and only permit coarse resolution modeling of large regions. In this paper, we present a highly scalable, integrated flood forecasting system called IFM that runs on both shared and distributed memory architectures, effectively allowing the computation of domains with billions of cells. In order to optimize IFM for large areas, we focus on the computationally expensive overland routing engine. We describe a parallelization scheme and novel strategies to partition irregular domains to minimize load imbalance in the presence of memory constraints that results in 40% reduction in time compared to best uniform partitioning. We demonstrate the scalability of the proposed approach for up to 8192 processors on large scale real-world domains. Our model can provide a 48-hour flood forecast on a watershed of 656 million cells in under 5 minutes.

1 Introduction

Operational flood forecasting is becoming increasingly important due to the changing global climate and frequent incidence of flood disasters [1]. The most common causes for flooding are sudden precipitation in urban areas with poor drainage or seasonal storms resulting in persistent rainfall, which results in overflowing water bodies. Hence, in recent years there has been a strong focus on two stage mechanisms to predict flooding events. The first stage employs a weather model to predict precipitation. The second stage uses these predictions as input to an overland flood model, which computes surface runoff and routes the flow taking into account surface characteristics such as variation in land use type and topography. In such a system, the weather forecasting is performed using fine resolution atmospheric models that discretize the partial differential equations representing evolution of atmospheric flows in time [12], while the overland flows are simulated via equations based on conservation of mass and momentum with the vertical effects simplified to yield the 2-D shallow water equation [14].

Advances in scaling high resolution weather models using contemporary HPC systems have made it feasible to obtain highly accurate fine-grained forecasts for large geographical regions [4, 5]. Unfortunately, most of the existing flood modeling packages [13], [2] primarily focus on usability and are designed for hydrologists to work on medium size desktop machines, which does not permit scaling to large size fine resolution domains for which weather forecasts are available. The domains considered in this paper for operational flood forecasting include a grid with 1 km horizontal resolution for the precipitation estimates and 1m LiDAR [9] topography data from the city of Rio de Janeiro – the latter consisting of 2.4 billion cells. With existing flood modeling software it is not feasible to perform modeling on such a large grid due to the large memory requirements and running times. On the other hand, efficient parallelization of the modeling requires a load-balanced partitioning of the domain, which is non-trivial due to domain irregularity and processor memory constraints.

Contributions:

- We describe an integrated flood forecasting system that readily handles grid sizes up to a billion cells and also incorporates high resolution meso-scale weather forecasts and other fine resolution topographical information for a target region with minimal human effort. To the best of our knowledge, this is the first high resolution operational flood forecasting with such capabilities.
- We propose and implement a distributed memory (MPI) parallelization strategy for diffusive water routing algorithms. Our approach is based on statistical modeling of the true workload using observed computational times and a novel iterative partitioning scheme, which improves load balance while taking into account memory constraints on individual processors.
- We evaluate the serial version and the various parallelization strategies on HPC systems for up to 8192 processors on real world domains and demonstrate that a large domain of 656 million cells can be solved under 5 minutes.

The rest of the paper is organized as follows: Section 2 discusses related work on parallelization of flood routing engines. Section 3 provides an overview of our integrated flood modeling framework. Section 4 discusses our distributed memory parallelization strategy whereas Section 5 introduces the various approaches to domain partitioning. Section 6 describes empirical evaluation of our approach on real world domains. Concluding remarks are presented in Section 7.

2 Related Work

Prior approaches on scaling water routing in flood modeling employ multiprocessor distributed architecture and divide the computation either via functional or domain decomposition.

Methods based on functional decomposition involve parallelization of nested loops to process the grid cells more efficiently. Neal et al. [7] explored the intrinsic parallelism in the functions that looped around the floodplain cells of a domain via OpenMP and demonstrated a speedup up to 5.8$\times$ relative to the serial algorithm for 8 cores with domain sizes varying from 3,000 to 3 million cells. The key limiting factors for the parallel speedup were the serial time and processor load imbalances. For the scale of problems we are interested in, such a simple shared memory implementation would not suffice due to scalability limits.

The second class of methods employ domain decomposition, where the grid to be simulated is split into smaller domains that are processed in parallel. The main challenge here is to is figure out a partitioning that achieves load balance. This task is particularly difficult due to three main reasons: (a) irregularity of domain, (b) dependence of computation costs on not only static properties of the domain, but also on dynamic attributes (e.g., wet cells in the neighborhood make the routing computation much more expensive than that of dry neighborhoods), and (c) memory constraints of individual processors. For our work, we adopt the domain decomposition approach due to its better scalability.

There is a large body of literature [8,10,16] on using domain decomposition to improve scalability of hydrological models via message-passing interfaces. These modeling approaches involve partitioning into regular rectangular shaped sub-domains primarily due to the huge software changes required to handle irregular shaped sub-domains. In particular, Yu [16] presents an approach to parallelize a two-dimensional model by spatially dividing the target region into sub-regions of equal size and dimension according to the number of available processors. Empirical performance evaluation of that approach on a large domain (232,000 cells) indicates a maximum speedup of 1.75×, 1.98× and 2.71× for MPI simulations using 2, 4, and 8 nodes, respectively, with associated efficiencies of 0.87, 0.50 and 0.33. A recent work [11] presents a hybrid MPI-OpenMP version that incorporates a master-slave model of MPI workload balancing for independent watersheds and OpenMP based shared memory parallelization within each basin. With this hybrid approach, the speedup was reported to be 13× on a 16 core machine. While this approach works on moderate sized systems with large shared memory, it does not scale to large watersheds due to memory limitations at a single processor and load imbalance due to a wide range of basin sizes.

3 Integrated Flood Modeling System

The Integrated Flood Model (IFM) is a hydrological model developed at IBM Research aimed at providing high resolution flood forecasts. IFM consists of two main components, a soil and an overland routing model, shown in Figure 1. Precipitation forecasts are provided by a state of the art weather model. The soil model estimates the *surface-runoff* based on the incoming precipitation, soil, and land use properties. These runoff estimates are input to the overland flood routing engine, which calculates the water in-flows and out-flows on a two-dimensional grid based on topological characteristics. The remnant water-flow from a simulation step is then fed back to the soil model to more accurately determine the water height in the next simulation step. In the serialized implementation of IFM, the overland routing dominates the execution time. Hence, we mainly focus on parallelizing the overland routing component described next.

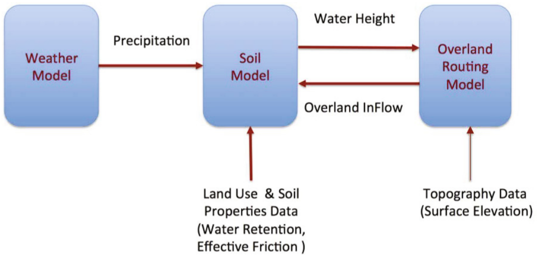

Weather Model
Precipitation
Soil Model
Water Height
Overland InFlow
Overland Routing Model
Land Use & Soil Properties Data (Water Retention, Effective Friction)
Topography Data (Surface Elevation)

Fig. 1. IBM Integrated Flood & Weather Modeling System

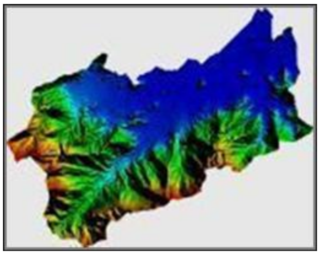


Fig. 2. Rio domain delimited by a bounding box

Overland Flood Routing. Overland water movement in IFM is implemented as diffusive routing, which allows the distribution of lateral inflow in both space and time [6] with significant reduction in computation cost. To be specific, the inflow in the X direction for the i^{th} cell denoted by $OLR_X[i]$ is given by the Manning formula [15] as $OLR_X[i] = \frac{1}{N[j]}\sqrt{|S[i,j]|} * \Delta d * H[j]^{\eta}$, where the j^{th} cell adjoins the i^{th} cell along the X-direction, $H[j]$ denotes the water height or surface-runoff of the j^{th} cell estimated by the soil model, $N[j]$ is the Manning's friction coefficient determined by land use data, $\eta = 5/3$ is based on the laminar and mixed laminar-turbulent conditions of the flow, Δd is the distance between the two cells and the terrain slope $S[i,j]$ indicates a net dip towards the j^{th} cell (i.e., $S[i,j] > 0$). This slope itself is calculated as $S[i,j] = \frac{1}{\Delta d}(H[i] + h[i] - H[j] - h[j])$, where $h[i]$ denotes the natural elevation of the i^{th} cell. The above routing is implemented on a 2D grid along both X and Y directions. First, the flow rate is calculated in the X direction (row-wise), letting the fluid flow from cell i to its neighbors or the other way round. Then, the flow rate is calculated in the Y direction (column-wise) to determine the in-flow in Y direction $OLR_Y[i]$ and the resulting in-flows are used to re-estimate the water height $H[i]$.

4 Distributed Memory Parallelization

In this section, we describe a distributed memory MPI implementation that partitions the domain into disjoint sub areas to be assigned to individual processors. To achieve effective parallelization, we need to address two main challenges:

Partitioning Irregular Shaped Domains. Watersheds tend to be highly irregular in shape as shown in Figure 2. Direct domain decomposition of such grids into possibly irregular sub-regions is non-trivial and the software changes required to support it are enormous. On the other hand, mapping to a regular grid results in grid cells that are not required for simulation and leads to wasted computation and extra book-keeping. Further, balancing the grid point count alone might result in heavy load imbalance among the processors. The processor memory limits also narrow the space of feasible partitionings.

Handling Common Boundaries. Even though the disjoint areas are processed independently, the parallel processing of the domains requires communication of the shared boundary cells (*halo region*) after every time step to synchronize among the different processors.

In our current work, we adopt a simple partitioning approach that divides the bounding rectangular grid of the irregular real world domain into tiles (possibly of different sizes), one for each processor. In Section 5, we discuss strategies to identify such a rectangular partitioning that optimizes load balance and minimizes communication costs in the presence of processor memory constraints.

The regularity of the tile structure allows the common boundaries to be readily handled. To avoid loading the entire dataset on a single processor, processor 0 streams each row of the mask file to identify the valid cells. Once the partitions are decided, processor 0 sends their coordinates and the neighbors at each boundary with their respective extent of overlap. The rest of the processes wait to receive their partition information. Afterwards, each process loads the subdomain based on the received data using a collective parallel I/O operation [3].

To process the partitions in parallel, we define a halo region of fixed width around each partition to store all the required information from neighbors during a simulation. Since the water flow in the domain is simulated from bottom to top and left to right, the top and right halo regions are used for runoff computations while left and bottom halo regions are simply used to pass information. The simulation starts by exchanging static information, e.g., the elevation and Manning's coefficients. In every iteration, the water heights are first updated based on the amount of runoff received from the left and bottom neighbors. After this step, computation is performed using the updated height values. However, processing of top and right boundaries is deferred and performed only after receiving the updated height values from the top and right neighbors. Note that since the tiles are not all of the same size, there can be multiple neighbors in each direction.

5 Domain Partitioning Approaches

Desiderata. Our primary objective is to identify a partitioning of the bounding grid of an irregular region into rectangular tiles that minimizes the net simulation time. This requires balancing workload across nodes while keeping communication costs low. Due to the nature of the computations in overland routing, the workload assigned to a node largely depends on the number of *valid grid points* assigned to it, i.e., all the points in the original irregular domain. Hence, it is highly likely that achieving load balance will require non-uniform tile sizes. Communication costs borne by a node, on the other hand, depend on the number of tiles adjoining the assigned tile and are minimized when the tile boundaries are aligned as in the case of a uniform partitioning. In addition to the load balance and communication reduction, it is also critical that the rectangular tile assigned to each node does not exceed its memory limit.

Partition Representation. Let $m_x \times m_y$ be the size of the bounding rectangular grid and N the number of processor nodes. Common partitioning strategies include: (a) 1-D decomposition where the N processors are arranged in a chain and the resulting tiles are row or column-wise slices of the original domain, (b)

2-D decomposition where the processors themselves are arranged in a $N_x \times N_y$ grid and the original domain is divided into $N_x \times N_y$ tiles slicing along one dimension and then another for better alignment of tiles. In non-uniform slicing, it is preferable to divide along the longer dimension of the domain grid. Without loss of generality, we assume that the tile construction involves a vertical slicing followed by a horizontal slicing of each of the vertical slices. Given a processor grid $N_x \times N_y$ (1-D case corresponds to $N_x = N$ and $N_y = 1$), the partitioning can then be represented in terms of the vertical boundaries $\{x(i),\ [i]_1^{N_x}\}$ and horizontal boundaries $\{y(i,j)\ [j]_1^{N_y},\ [i]_1^{N_x}\}$. In the 2-D decomposition case, it is preferable to choose the processor grid dimensions N_x and N_y to the closest two factors of N, i.e., nearly equal to each other, for better distribution of communication costs. To ensure a balanced aspect ratio for each individual tile, the factors N_x and N_y can be assigned to the X-Y dimensions so that the larger domain dimension corresponds to the larger of N_x and N_y. In each of the above cases, the decomposition could be based either on ensuring that the tiles are nearly equal in size (i.e., number of total grid points) which reduces communication costs or nearly equal in workload (i.e., roughly proportional to the number of valid grid points). These correspond to different variants: (a) 1D-uniform, (b) 1D-nonuniform, (c) 2D-uniform, and (d) 2D-nonuniform. The first three variants are relatively simple, but less likely to achieve load balance and more prone to violate processor memory constraints. For the flood modeling, since computation tends to cost more than communication, the last variant is likely to perform better in terms of net simulation time on large grids. Determining its optimal partition, however, especially in the face of memory constraints, is non-trivial.

5.1 Two Dimensional Non-uniform Partitioning

We now focus on 2-D partitioning where the workload needs to be balanced while satisfying memory constraints. Although it is natural to assume that the workload depends only on valid cells, empirical observations point to a significant variation in computation time among tiles with nearly identical valid cells, as shown in Table 1. Hence, we assume the workload to be a linear function of the number of valid and invalid cells. Let $C^v(x, y)$, $C^a(x, y)$ and $C^w(x, y)$ denote the number of valid grid points, the total number of cells, and the workload in the sub-grid with corners $[(0, 0), (x, 0), (0, y), (x, y)]$ inclusive of the boundaries. Then, the assumption on the workload translates to $C^w(x, y) = C^v(x, y) + \alpha(C^a(x, y) - C^v(x, y))$, where $0 \leq \alpha < 1$ is the weighting factor for invalid cells. The optimal value for α can be chosen based on empirically observed computation times ($\alpha = 0$ equals to focusing on valid cells alone). Experiments in Section 6 are based on choosing $\alpha = 1/73$, which was computed from empirical data.

The total workload of the grid is given by $C^w_{tot} = C^w(m_x, m_y)$ and the workload up to and including the x^{th} column is $C^w(x, m_y)$. Let $W(i, j)$ denote the workload in the j^{th} horizontal tile of i^{th} vertical slice. Using the notation for tile boundaries, this can be computed as $W(i, j) = C^w(x(i), y(i, j)) - C^w(x(i), y(i, j-1)) - C^w(x(i-1), y(i, j)) + C^w(x(i-1), y(i, j-1))$. The number of

Table 1. Computation times of partitions with comparable number of valid cells

	Invalid cells	Valid cells	Time (sec.)
Partition 1	2,457,186	79,998	622.92
Partition 2	840,252	80,708	566.58
Partition 3	418,072	80,808	541.46

valid cells $V(i,j)$ can be similarly obtained while the total number of cells $T(i,j)$ can be computed more efficiently as $T(i,j) = (x(i)-x(i-1))*(y(i,j)-y(i,j-1))$.

In the absence of memory constraints, the ideal 2D partitioning corresponds to the case where $W(i,j) = C^w_{tot}/(N_x \times N_y), \forall i,j$. Such a perfect partitioning is, often, not feasible since the tiles need to be rectangular. However, one can obtain a nearly equal distribution via a two step approach, where in the first step, the vertical slices are each chosen to approximately contain workload equal to C^w_{tot}/N_x and each of these slices is further horizontally partitioned into tiles that roughly contain $1/N_y$ of the workload in entire slice. The i^{th} vertical boundary $x(i)$ is picked so that it is the smallest column index such that the workload over all rows and up to the $x(i)^{th}$ column exceeds $\frac{iC^w_{tot}}{N_x}$, i.e.,

$$C^w(x(i)-1, m_y) < \frac{iC^w_{tot}}{N_x} \leq C^w(x(i), m_y), \ [i]_1^{N_x}. \qquad (1)$$

Similarly, for each vertical slice i, we pick the j^{th} horizontal boundary $y(i,j)$ with the smallest row index such that the workload on all the slice columns and up to the $y(i,j)^{th}$ row exceeds $\frac{j}{N_y}$ fraction of the total workload in that slice.

The above two-step slicing approach results in a fairly equitable partitioning of workload, but the resulting tiles might not fit into the memory available at a single node of a distributed memory cluster, in which case the partitioning cannot be used for simulation. When there are memory constraints that place an upper bound T_{max} on the tile size (i.e., total number of cells, not just valid ones) that can be accommodated at each node, some of the processors will need to be assigned tiles with total size close to T_{max}, but much smaller number of valid cells. Typically, these would correspond to ocean and land portions outside of the targeted watersheds. To make up for the lighter workload of the maximal tile nodes, it is necessary to increase the workload for all the other nodes in a balanced fashion. Figuring out the optimal partitioning for this scenario is hard since the tiles need to be contiguous and one cannot estimate the number of maximal size tiles and the desired workload distribution in a single step.

Typically, the first stage of vertical slicing results in vertical slices with width much smaller than T_{max}. Hence, for ease of presentation, we consider partitioning taking into account the tile size limit T_{max} only in the second stage of horizontal slicing.[1] To eliminate inefficiencies, we only consider the grid allocation and workload contributions from the set of rows $R(i)$ that have at least one valid cell in the chosen vertical slice. Algorithm 1 provides details of the horizontal

[1] When T_{max} is small, the first stage of horizontal slicing can also be adapted in a similar fashion as in Algorithm 1 to account for memory constraints.

partitioning developed for the i^{th} vertical slice. The key idea in this approach is to perform multiple iterations scanning the vertical slice and in each iteration, construct tiles from one end to another while dynamically re-estimating an equitable distribution of remaining workload among the remaining processors. In the very first iteration, the dynamic estimation of workload share depends only on allocations till that point, whereas in the later iterations, we also incorporate information from the previous iteration on the number of maximal size tiles and the workload covered in those tiles.

Let $W(i) = W$ denote the total workload of the i^{th} vertical slice (based on the rows in $R(i)$) and L_{max} the maximum allowed tile length based on T_{max} and the width of the i^{th} slice. At any stage in the k^{th} scan, let $N_A^{(k)}$ denote the number of nodes that have been assigned tiles and $W_A^{(k)}$ denote the already allocated workload. In the first iteration, the best one can do is to assume that the rest of the workload is going to equally shared among yet to be allocated nodes and the desired workload for the next tile is given by

$$W_{eq}^A = \frac{W - W_A^{(k)}}{N_y - N_A^{(k)}}.$$

If the next tile length required to cover workload of W_{eq} is greater than L_{max}, then the tile length is chosen to be exactly L_{max}. In such a case, the workload to be shouldered by the remaining processors goes up even further resulting in a high load imbalance. Further, the last tile itself may reach the maximal size with additional unassigned workload due to an early under allocation.

To address this issue, we perform additional rounds of assignment where a more balanced workload share is computed incorporating information from the previous iteration. Let $N_M^{(k)}$ and $W_M^{(k)}$ denote the number of nodes that attain maximum tile size and the net workload assigned to them in the k^{th} iteration (or up to that point in case of current iteration), Let $N_E^{(k)}$ and $W_E^{(k)}$ denote the minimum number of tiles required to cover the unassigned region at the end and the associated workload for the k^{th} iteration. In the earlier stages in the $(k+1)^{th}$ iteration, it would be reasonable to assume that among the remaining $(N_y - N_A^{(k)})$ nodes, approximately $(N_M^{(k)} - N_M^{(k+1)} + N_E^{(k)})$ would have to be assigned maximal size tiles roughly accounting for $W_M^{(k)} - W_M^{(k+1)} + W_E^{(k)}$ of the workload. Keeping this in view, a better estimate of the workload to be shared is given by

$$W_{eq}^B = \frac{W - W_A^{(k+1)} - W_M^{(k)} + W_M^{(k+1)} - W_E^{(k)}}{N_y - N_A^{(k+1)} - N_M^{(k)} + N_M^{(k+1)} - N_E^{(k)}}.$$

This estimate tends to overload the processors aggressively from the very beginning and might result in relatively no less or no work for the last few processors. Often, W_{eq}^B is higher than W_{eq}^A, but to address scenarios with complicated arrangement of sparse workload regions, we consider the maximum of the two choices. When T_{max} is large enough to allow a feasible partitioning, it can be shown that the above algorithm converges to a solution (not necessarily optimal) in a finite number of rounds since $W_E^{(k)}$ decreases after each iteration.

Algorithm 1. Two-dimensional partitioning with memory constraints

```
**Input:** Vertical slice with column indices $x(i)$ and $x(i-1)$, Workload matrix $C^w$ computed over rows in $R(i)$ with at least one valid cell, Max. tile size $T_{max}$, Processors along Y dimension $N_y$, Max. iterations $Kmax$
**Output:** Partitioning of the slice into horizontal tiles $y(i,j)$, $[j]_1^{N_y}$
**Method:** $k \leftarrow 1$;
Max. tile length $L_{max} \leftarrow ceil(\frac{T_{max}}{x(i)-x(i-1)})$
Last valid row $y_{max} \leftarrow \max(R(i))$
Total workload of $i^{th}$ slice, $W \leftarrow C^w(x(i), m_y) - C^w(x(i-1), m_y)$;
while $(k \leq 2)$ or $((k <= Kmax)$ and $(W_E^{(k-1)} > 0))$ do
    $(N_A^{(k)}, W_A^{(k)}, N_M^{(k)}, W_M^{((k)}, y[i,0]) \leftarrow (0,0,0,0,0)$
    for $j = 1$ to $N_y$ do
        $W_{eq}^A \leftarrow \frac{W-W_A^{(1)}}{N_y-N_A^{(1)}}$
        if $(k > 1)$ then
            $W_{eq}^B = \frac{W-W_A^{(k)}-W_M^{(k-1)}+W_M^{(k)}-W_E^{(k-1)}}{N_y-N_A^{(k)}-N_M^{(k-1)}+N_M^{(k)}-N_E^{(k-1)}}$
            $W_{eq}^A = \max(0, W_{eq}^A, W_{eq}^B)$
        $y_{tmp} \leftarrow \arg\min_y = \{y|C^w(x(i),y) - C^w(x(i-1),y) > W_A^{(k)} + W_{eq}^A\}$
        if $(y_{tmp} - 1 < \min(y(i,j-1) + Lmax, y_{max}))$ then
            $y(i,j) \leftarrow y_{tmp} - 1$
        else
            $y(i,j) \leftarrow \min(y(i,j-1) + Lmax, y_{max})$
            $N_M^{(k)} \leftarrow N_M^{(k)} + 1$
            $W_M^{(k)} \leftarrow W_M^{(k)} + W(i,j)$
        $N_A^{(k)} \leftarrow N_A^{(k)} + 1$
        $W_A^{(k)} \leftarrow W_A^{(k)} + W(i,j)$
    $N_E^{(k)} \leftarrow ceil(\frac{|R(i) \bigcap \{y(i,N_y),\cdots,m_y\}|}{Lmax})$
    $W_E^{(k)} \leftarrow W - W_A^{(k)}$
```

6 Empirical Evaluation

6.1 Experimental Setup

Hardware & Software Configurations. For our experiments, we used an IBM Blue Gene BG/P computer that has four 850 MHz embedded PowerPC 450 cores with a peak floating point throughput of 13.6 GF/node. For compiling the software, we used IBM XLC compilers on BG/P with -O3 optimization. In order to handle various platform independent binary files as input and output, we incorporated Network Common Data Format (NetCDF) support for I/O. A version of NetCDF dubbed PnetCDF [3] that is built on top of MPI-IO provides an easy to use interface to perform parallel I/O on large scale supercomputers and was, therefore, integrated into IFM for all I/O.

Experiments were performed on two real world domains (Brunei and Rio) using multiple partitioning schemes, with details given in Table 2. For the Brunei domain, a topography grid with spatial resolution of 90m and 1688×1318 cells (of which 72% are valid) was used. The Rio domain was processed with a grid of 1-meter resolution derived from LiDAR, with 46% of its 18369×35726 cells being valid.

Table 2. Details of the partitioning schemes

Partitioning Scheme	Description
1D-N	Uniform 1D split along longer dimension
1D-VM	1D split that balances #valid cells under memory constraints
2D-N	Uniform 2D split that balances #total cells
2D-VM	2D split that balances #valid cells under memory constraints
2D-WVM	2D split that balances workload assumed to be linear function of #valid and #invalid cells with $\alpha = 1/73$ (see Section 5.1)

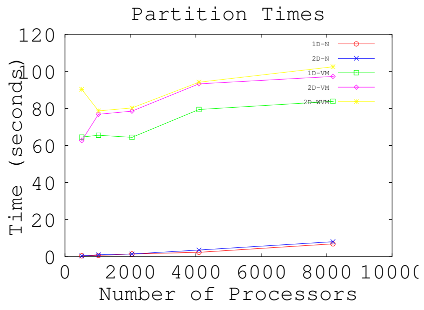


Fig. 3. Partitioning times for varying number of processors

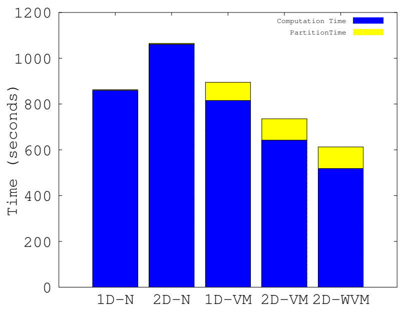


Fig. 4. Partition vs total times on 4096 processors (Rio domain)

6.2 Results

We now present results of empirical evaluation of our partitioning strategies.

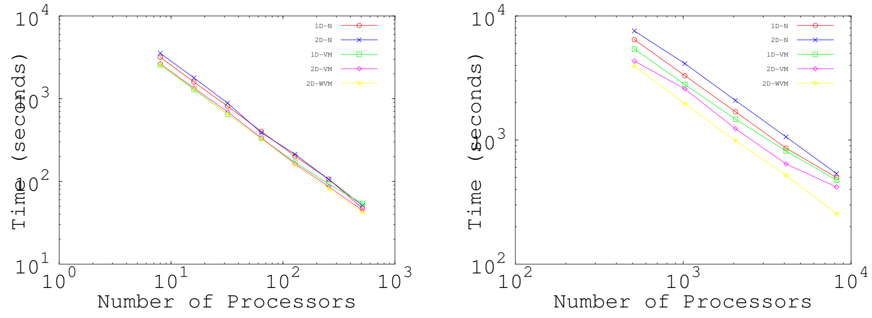
Partitioning Costs. Figure 3 shows the partitioning costs for the various schemes with increasing number of processors. As the number of processors grow, a modest increment in partitioning costs is seen. Figure 4 shows the split up of partitioning and total computation times for a 48 hour forecast for the Rio domain based on 1440 simulation steps. The naïve partitioning schemes take negligible time. However, the computation times for 2D-N partitioning scheme is almost double that of the best case (2D-WVM). These figures indicate that it is worthwhile to spend the extra partitioning time since it is a one time cost that can be amortized while simulating larger number of timesteps. (and which can be cached to save computation time in future runs)

Effect on Load Balancing. Table 3 shows the load balance achieved by the various partitioning schemes. 2D-N has the highest load imbalance (116.34%) and our weighted 2D-WVM partitioning scheme has the lowest load imbalance (6.36%), which is also significantly superior to the unweighted version 2D-VM(32%). Improving the load balance results in a 40% reduction in total time over the best naïve partitioning scheme, which is 1D-N for the Rio domain.

Scaling of MPI Implementation. We now present the results of experimentation with the various partitioning schemes for varying number of processors. Figure 5a shows the scaling behavior for Brunei domain for 8 – 512 processors. The 1D-N scheme outperforms the 2D-N scheme for almost all the processor

Table 3. Cell distribution for partitions with maximum times and the percentage imbalance with respect to average time across all processors (Rio domain)

Scheme	Invalid Cells	Valid Cells	Max Time(s)	Avg. Time (s)	%Imbalance
1D-N	35357	129964	860.20	495.77	73.51
2D-N	0	160433	1061.27	490.57	116.34
1D-V	39937	125384	815.50	486.44	67.65
2D-VM	2440289	81151	642.28	486.43	32
2D-WVM	0	75775	520.86	489.72	6.36



(a) Times for the Brunei domain (b) Times for the Rio domain

Fig. 5. Log-log plot of scaling behavior with increasing number of processors

configurations. Since this domain has a high number of valid cells in comparison to the total number (72%), 2D-N partitioning does almost as well as 2D-VM and 2D-WVM schemes for 512 processors. This is because only a small fraction of processors are not completely explored and there is only a slight load imbalance even for the 2D-N scheme. Overall we observe a 15–27% improvement in performance for the 2D-WVM scheme in comparison to a naïve 2D decomposition for this range of processors. Figure 5b shows the scaling behavior for all the partitioning schemes for the Rio domain. Here, the 1D-N naïve partitioning scheme consistently outperforms the 2D-N scheme for up to 8192 processors. This can be due to the fact that very few processors are kept idle at times due to the distribution of valid and invalid cells in the Rio domain. However, we do see signs of flattening of the curves beyond 4096 for all the schemes except 2D-WVM.

7 Conclusion

Operational flood forecasting is an important problem requiring a scalable high resolution integrated modeling solution. Our current work presents such an integrated modeling system IFM comprising soil model, and a water routing engine. In particular, we focus on the routing process, which is the most compute intensive and propose a distributed memory parallelization scheme to scale it up to large grid sizes. We also present novel partitioning techniques to minimize load imbalance subject to memory constraints. Empirical evaluation of our proposed approach on large scale real-world domains demonstrates that it scales well up to 8192 processors, and can enable a number of applications and services to be built around flood forecasts that are delivered in a timely fashion. Though inspired by the constraints of the flood-modeling problem, the proposed 2D-WVM partitioning scheme presents two key ideas that are likely to have broad applicability in other areas (e.g., computational seismology) that involve irregular and/or heterogeneous domains and resources constraints: (a) iterative refinement of partitioning by using information from previous iteration(s) on partitions that achieve the constraints (b) statistical modeling of the true workload of a partition in terms of the constituent grid cell properties.

References

1. 10 costliest floods worldwide ordered by overall losses, `http://www.munichre.com/app_pages/www/res/pdf/NatCatService/significant_natural_catastrophes/2012/NatCatSERVICE_significant_floods_eco_en.pdf`
2. Gill, M.A.: Flood routing by the Muskingum method. Journal of Hydrology 36(34), 353–363 (1978)
3. Li, J., Liao, W.K., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B., Zingale, M.: Parallel netCDF: A High-Performance Scientific I/O Interface. In: SC (2003)
4. Malakar, P., et al.: A divide and conquer strategy for scaling weather simulations with multiple regions of interest. In: SC 2012, pp. 37:1–37:11 (2012)
5. Michalakes, J., et al.: WRF Nature Run. In: SC (2007)
6. Moussa, R., Bocquillon, C.: Algorithms for solving the diffusive wave flood routing equation. Hydrological Processes 10(1), 105–123 (1996)
7. Neal, J., Fewtrell, T., Trigg, M.: Parallelisation of storage cell flood models using OpenMP. Environmental Modelling & Software 24(7), 872–877 (2009)
8. Neal, J.C., Fewtrell, T.J., Bates, P.D., Wright, N.G.: A comparison of three parallelisation methods for 2D flood inundation models. Environ. Model. Softw. 25(4), 398–411 (2010)
9. Priestnall, G., Jaafar, J., Duncan, A.: Extracting urban features from LiDAR digital surface models. Computers, Environment and Urban Systems 24(2) (2000)
10. Sanders, B.F., Schubert, J.E., Detwiler, R.L.: ParBreZo: A parallel, unstructured grid, Godunov-type, shallow-water code for high-resolution flood inundation modeling at the regional scale. Advances in Water Resources 33(12), 1456–1467 (2010)
11. Singhal, S., Villa Real, L., George, T., Aneja, S., Sabharwal, Y.: A hybrid parallelization approach for high resolution operational flood forecasting. In: HiPC 2013 (2013)
12. Skamarock, W.C., et al.: A description of the Advanced Research WRF version 3. Tech. Rep. TN-475, NCAR (2008)
13. Todini, E.: The ARNO rainfall runoff model. J. Hydrology 175(14), 339–382 (1996)
14. Vreugdenhil, C.: Numerical Methods for Shallow-Water Flow. NATO Asi Series. Series C, Mathematical and Physical Science. Springer (1994)
15. Yen, B.: Channel Flow Resistance: Centennial of Manning's Formula. Water Resources Pub. (1992)
16. Yu, D.: Parallelization of a two-dimensional flood inundation model based on domain decomposition. Environmental Modelling & Software 25(8), 935–945 (2010)