Hydroinformatics advances for operational river forecasting: using graphs for drainage network descriptions

Zhengtao Cui, Victor Koren, Neftali Cajina, Andreas Voellmy and Fekadu Moreda

ABSTRACT

Distributed hydrologic models provide accurate river streamflow forecasts and a multitude of spatially varied products on basin scales. The distributed elements of the basins are pieced together using drainage networks. An efficient representation of drainage networks in computer code is necessary. Graph theory has long been applied in many engineering areas to solve network problems. In this paper we demonstrate that adjacent list graph is the most efficient way of presenting the drainage network in terms of development and execution. The authors have implemented drainage networks using the adjacency-list structure in both the research and operational versions of the US National Weather Service (NWS) distributed model. A parallel routing algorithm based on Dijsktra's shortest path algorithm was also developed using the MPI library, which was tested on a cluster using the Oklahoma Illinois River basin dataset. Theoretical analysis and test results show that inter-processor communication and unbalanced workload among the processors limit the scalability of the parallel algorithm. The parallel algorithm is more applicable to computers with high inter-processor bandwidth, and to basins where the number of grid cells is large and the maximum distance of the grid cells to the outlet is short.

Key words | distributed hydrologic modelling, graph theory, parallel computing

Zhengtao Cui (corresponding author)
Victor Koren
Neftali Cajina
Fekadu Moreda
Office of Hydrologic Development,
NOAA National Weather Service,
1325 East-West Highway,
Silver Spring, MD 21043,
USA
E-mail: *Zhengtao.Cui@noaa.gov*

Andreas Voellmy
Department of Computer Science,
Yale University,
New Haven, CT 06520,
USA

INTRODUCTION

Distributed hydrologic models attempt to account for the spatial variability of land surface features and meteorological forcings such as precipitation and temperature. They accomplish this by disaggregating the geographic area of interest into a number of small control units. A simplistic view of this would be if one placed a grid over a map of the area of interest. Physical features within each individual grid cell are assumed constant, but vary from cell to cell. The distributed model then solves the relevant rainfall to runoff conversion equations in each grid cell.

Such models continue to be developed by many organizations around the world. They take advantage of spatial information and data acquired by today's advanced technologies such as Digital Elevation Models (DEM) and precipitation measured by radar and satellites. Recently the National Weather Service (NWS) implemented a distributed hydrologic model for operational river forecasting. Interested readers are referred to Koren *et al.* (2004) for more information regarding this model. In addition to anticipated benefits for river and flash flood forecasting, distributed models are viewed by the NWS as one of the major pathways through which new water resource forecast products can be provided to the US (Carter 2002).

doi: 10.2166/hydro.2010.023

A distributed hydrologic model computes runoff and route flow in each of the control volumes, which are usually represented by a rectangular grid or Triangulated Irregular Network (TIN). Most distributed hydrologic models assume that surface water from a cell flows towards one and only one of its immediate neighbour cells. The flow direction is usually defined by steepest slope from a cell to its neighbours. All the connected flow paths on the grid cells consist of a drainage network, which is referred to as the hydrologic connectivity.

Using this hydrologic connectivity, the distributed hydrologic model can route flows from each of the grid cells from upstream to downstream. In the real world, there are situations where multiple flow paths exist from a grid cell or a pit (flow from one cell does not flow to any of its neighbours) exists. Nevertheless, the unique flow direction assumption is valid for most of the applications. We therefore only discuss the drainage network with a unique flow path on each of its grid cells and there is no pit or loop in the drainage network. Usually the drainage network at coarser resolution is derived from a finer resolution flow direction grid. Earlier investigators determined the drainage network directly from DEMs using the D8 model where the path flows towards one of the eight directions: north, northeast, east, southeast, south, southwest, west and northwest (O'Callaghan & Mark 1984; Jenson & Domingue 1988). Later, more sophisticated algorithms were developed to overcome errors in the flow directions when the DEM resolution is 30 arc-seconds or larger (Reed 2003).

Distributed hydrologic models continue to be an active area of research and development and are applied to larger and larger geographic domains, increasing the need for efficient ways to manage the drainage networks in computer programs. The purpose of this paper is to discuss how to use graph theory to implement drainage networks and how the graph approach offers efficiencies for model developers and subsequent operational implementation. Here, we extend the work presented by Cui *et al.* (2006).

The remaining part of the paper is organized as follows. After the literature review, we present an overview of graph theory and how to model drainage networks as undirected graphs. We then discuss two graph algorithms, Preorder Tree Walk and Postorder Tree Walk and their variations, for the purpose of visiting each grid cell in a particular order. We then discuss how to use Dijkstra's shortest path algorithm to develop parallel distributed models that can utilize multiple processors. Finally, a case study that uses the parallel algorithm is presented.

LITERATURE REVIEW

Many algorithms have been developed to derive hydrologic drainage networks from DEMs or TINs (Jenson & Domingue 1988; Fairfield & Leymarie 1991; Smith & Brilly 1992; Reed 2003). Implicit in these efforts, but often without the details published in the literature, are schemes to encode drainage networks for efficient computer implementation. Early pioneering research focused on describing the channel network in digital formats.

The binary string method was proposed by Scheidegger (1967) and Shreve (1967). The basic idea of this method is to travel around the network by starting at the outlet, turn left at each junction and turn right back at each source, assign a value of '1' for an exterior link and '0' for an interior link. The resulting binary string has n 1s and $(n - 1)$ 0s. Later, Smart (1970) showed that "the binary string representation can also be used in processing many other types of information on channel networks and that the procedures involved appear particularly suitable for computer retrieval". Coffman & Turner (1971) extended the binary string method by using a numeric code instead of a binary code to overcome some restrictions of binary strings, such as loss of coordinate data.

Verdin (1997) introduced a numbering scheme to code the global drainage basins and stream networks developed by Pfafstetter (1989). The drainage area is first subdivided into four largest tributaries, which are assigned the numbers 2, 4, 6, and 8. Next, the interbasins are numbered 1, 3, 5, 7, and 9. If a closed basin is encountered, it is assigned the number 0. A basin or interbasin may be recursively subdivided by repeating the application of the same rules to the area within it.

Although these early approaches are efficient, they are limited because they cannot describe detailed information about each linkage in the drainage network. As computer hardware became more capable, graph theory emerged

as a powerful tool to solve engineering problems (Kreyszig 1998). Gleyzer *et al.* (2004) developed a recursive Strahler stream ordering algorithm by using graph theory to perform stream network analysis. Mark (1988) used a two-dimensional matrix that implements a graph to describe the flow direction grid. A recursive algorithm was then used to find the drainage accumulation for a given grid cell elegantly and efficiently.

Several researchers have applied graph theory to model the connectivity of elements in landscapes (Cantwell & Forman 1993; Reynolds & Wu 1999; Urban & Teitt 2001; Schroder 2006). Jiang & Claramunt (2004) modelled an urban street network using graph theory and Gupta & Prasad (2000) analyzed pipeline networks using linear graph theory.

A graph-based representation of a flow network comprised of arcs and nodes was used as a data storage object defined in the Management Simulation Engine Network (Park *et al.* 2005). In this work, graph-based data objects serve as state and process information repositories for management processes (algorithms). It provides a mathematical representation of a constrained, interconnected flow network which facilitates efficient graph theory solutions of network connectivity and flow algorithms.

Apostolopoulos & Georgakakos (1997) described a graph approach for the streamflow prediction problem by using distributed hydrologic models within a shared memory parallel computing environment. In this work, a basin is disaggregated into river segments, each of which consists of a main channel and a number of reaches and sub-reaches. A hydrologic model is decomposed into subcomponents such as input, soil water accounting and channel routing. The reaches are mapped to the edges of the graph and model subcomponents are mapped to the vertices of the graph. The graph is then used to determine the dependencies of the graph vertices (model components) and the graph edges (river reaches). This approach also adopted the shared memory parallelism in which the user has no control on how the workload is allocated among the processors.

Based on directed trees (a type of graph), Bailly *et al.* (2006) introduced a theoretical framework required to specify geostatistical hypotheses and models. In this framework, for every edge u a river segment linking two vertices is represented and is naturally directed by water flow from upstream to downstream.

Because graph theory is widely used in various scientific areas and must be represented in computer programs, there is a need for graph libraries to facilitate the use of graph theory. Lee *et al.* (1999) first developed the Generic Graph Component Library (GGCL) which became the Boost Graph Library (BGL) in 2000 within the Boost Software License. Just like the C++ Standard Template Library (STL), the BGL was written to be generic: (1) the graph algorithms of the BGL are written to an interface that abstracts away the details of the particular graph data-structure; (2) the graph algorithms of the BGL are extensible through the visitor concept; and (3) it is analogous to the parameterization of the element-type in STL containers. In this work, we used BGL data structures and graph algorithms to develop and manipulate hydrologic connectivity by generic programming. Generic programming is a programming technique that uses templates to abstract away detailed implementation of efficient algorithm, data structures and other software concepts in a systematic way. Generic programming can produce concise, efficient, reusable and extensible code.

Our approach presented in this paper has three distinguishing characteristics: (1) it is suitable for distributed hydrologic models that disaggregate simulation domain into grid cells, TIN or hydrologic response units; (2) the proposed parallelization scheme is based on a message passing concept in which the user has full control of the number of processors and how workloads are distributed among processors and, therefore, the resulting parallel algorithm is more portable; (3) the hydrologic model is implemented by the generic programming technique which produces extensible and reusable computer codes.

GRAPH THEORY REVIEW

Graph definition

Before starting our discussion, it is helpful to introduce some basic terms and concepts. By definition, a graph G is a pair (V, E) where V, the *vertex* set, is a finite set of vertices

and E, the *edge* set, is a set of vertex pairs (u, v) with u and v in V. A graph must have at least one vertex but may have no edges (in which case it is a null graph).

A graph is *undirected* if the edge set E consists of unordered pairs of vertices, rather than ordered pairs (Figure 1). An edge connects the two vertices in both directions. That is, an edge is a set u, v where $u, v \in V$ and $u \neq v$. In an undirected graph, self-loops are forbidden and so every edge consists of exactly two distinct vertices. In contrast, a graph is *directed* if its edges are ordered pairs which connect each source vertex to a target vertex.

The *walk* of a graph is an alternating series of adjacent edges and vertices; a *path* is a sequence of vertices $<v_0, v_1, \ldots, v_l>$ in a graph $G = (V, E)$ such that each vertex is connected to the next vertex in the sequence. The edges (v_i, v_{i+1}) for $i = 0, 1, \ldots, k-1$ are in the edge set E. An undirected graph is *connected* if every pair of vertices is connected by a path. A graph with no cycles is *acyclic*, i.e. no cycle is found along any path. An acyclic undirected graph is a *forest* and a connected acyclic undirected graph is a *tree*.

A *rooted tree* is a tree in which one of the vertices is distinguished from the others. The distinguished vertex is called the *root* of the tree. A vertex of a rooted tree is often referred to as a *node*. Each node is either a leaf or an internal node. An internal node has one or more child nodes and is called the *parent* of its child nodes; each node has one parent except the root. A node without children is called a *leaf*. For a rooted tree, each node has one and only one path to the root, and each node can only be accessed from the *root*. That is the root is predefined.

A graph is *dense* if its number of edges is close to the square of the number of vertices. On the other hand, a graph is *sparse* if its number of edges is less than or close to its number of vertices. A rooted tree is a sparse graph because its number of edges is one less than its number of vertices, i.e. $E = V - 1$.

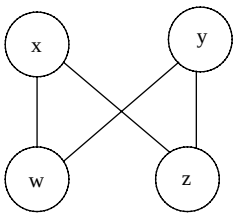

Figure 1 | An undirected graph.

Graph implementation

Three data structures can be used to implement a graph: matrix, adjacency-list and edge-list. Each implementation has time and space performance trade-offs which should be considered when choosing a data structure.

Adjacency-matrix

A graph can be implemented by a $V \times V$ adjacency-matrix where V is the number of vertices. Each element of the adjacency-matrix a_{ij} is a binary bit indicating whether or not there is an edge from vertex i to j.

Figure 2 depicts the adjacency-matrix for the graph in Figure 1. The amount of space required to store an adjacency-matrix is $O(V^2)$. Any edge can be accessed, added or removed in $O(1)$ time. Adding or removing a vertex requires reallocating and copying the whole graph, an $O(V^2)$ operation. The adjacency-matrix approach is suitable for dense graphs.

Here, $O(V^2)$ is called the big-O notation. The big-O notation describes the upper bound magnitude of a function in terms of another, usually simpler, function. The big-O notation is commonly used to measure the complexity of an algorithm in terms of either space or time. It also describes how the runtime (space) grows with the number of inputs. The formal definition is as follows.

For a given function $g(n)$, $O(g(n))$ is the set of functions

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

For example, $f(n) = 3n^2 + 2n + 1$, then $g(n) = n^2$. Because for all $n \geq 1$, we have $c > 6$ such that $0 \leq f(n) \leq cg(n)$.

	x	y	z	w
x	0	0	1	1
y	0	0	1	1
z	1	1	0	0
w	1	1	0	0

Figure 2 | The matrix representation of the graph depicted in Figure 1.

Since the big-O notation describes the upper bound of a function, it is used to bound the worse-case runtime (space) of an algorithm. When we say the time (space) complexity of an algorithm is $O(V^2)$, it is equivalent to saying that the worse-case runtime (space) is bounded by the square of the number of the vertices, or the worse-case runtime (space) grows with the square of the number of vertices. $O(1)$ means that the worse-case runtime (space) is constant, i.e. it doesn't grow as the number of inputs increase.

Adjacency-list

An adjacency-list implementation of a graph has a list of vertices with each vertex containing a sequence of out-edges of this vertex. Since only those edges that exist in the graph are stored, it saves some space for sparse graphs. Figure 3 shows the adjacency list implementation of the graph in Figure 1.

The list of vertices in an adjacency-list can be stored in a vector or a linked-list. The vector has the advantage of fast access time, of the order $O(1)$. However, the disadvantage is computationally expensive insertion and deletion because the entire vector needs to be copied over. On the other hand, the linked-list has the advantages of fast insertion and deletion, but the disadvantage of expensive access time of the order $O(V)$. In addition, the linked-list needs extra space to store pointers for each vertex.

Edge-list

An edge-list implements a graph by simply using a sequence of edges, where each edge is represented as a pair of vertex identifiers (IDs). The memory required is of the order $O(E)$. Edge insertion is typically $O(1)$, although accessing a particular edge is of the order $O(E)$, which is not efficient. Since no vertex is stored, vertices can only be accessed through the edges they are associated with and can only be inserted or deleted by inserting and deleting the corresponding edges.

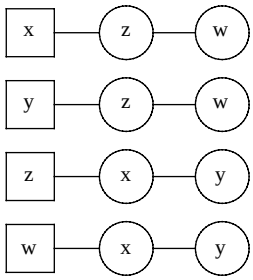

Figure 3 | The adjacency-list representation of graph.

REPRESENTING DRAINAGE NETWORKS WITH GRAPHS

As mentioned before, the flow direction grid is derived from digital topographic information such as a DEM or TIN using specialized algorithms. Figure 4 shows the resulting flow direction grid using the algorithm proposed by Reed (2003) on the Blue River basin in Oklahoma, USA. Each cell in Figure 4 is approximately 4 × 4 km. The black line is the boundary of the drainage basin defined by natural topography, while the dark grey lines represent the same border when the basin is represented by a grid. The arrows show the flow directions from each cell. Only the case where each grid cell has only one outflow is considered here. Braided channels need to be addressed in future work.

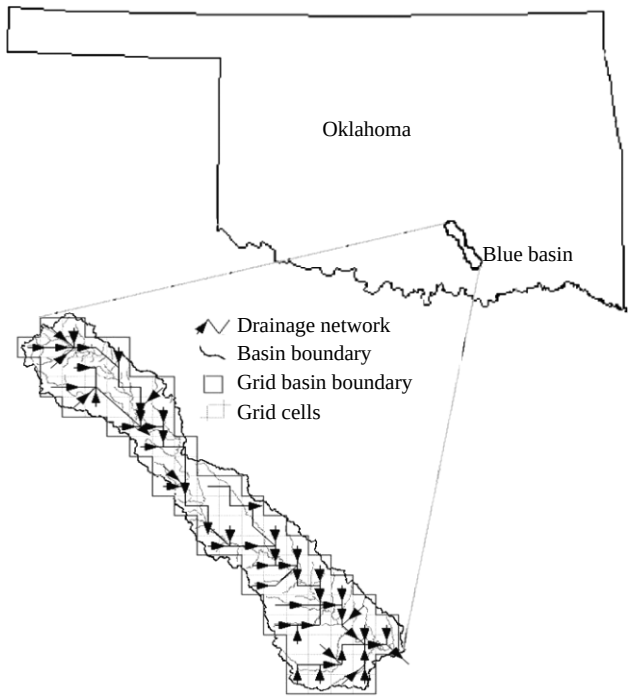

Figure 4 | Connectivity of the Blue River Basin, Oklahoma, USA.

The hydrologic connectivity grid, or drainage network, can be modelled by an undirected graph that has one vertex for each grid cell and one edge (u, v) for each pair of grid cells for which u flows toward v. For each drainage basin, such a graph is connected since each grid cell either flows into another cell or has flow coming from another cell. Furthermore, since the outlet grid cell has no parent cells, distinguishing the vertex corresponding to the outlet makes this graph a rooted tree. We choose the undirected graph instead of the directed graph because we use the rooted tree to represent a basin.

Our approach of applying graph theory to distributed hydrologic modelling differs from that adopted by Apostolopoulos & Georgakakos (1997). In their approach, the basin is divided into river segments and each river segment is sub-divided into a main channel, reaches and sub-reaches. To apply graph theory, the main channel, reaches and sub-reaches are mapped to the edges of the graph, while the starting point and the intermediate points of the reaches and the hydrologic models are mapped to the vertices of the graph. Thus a graph contains both data (reaches) and algorithms (models). In contrast, in our approach, only grid cells are mapped to the vertices of the graph; the graph edges connect grid cells to form a drainage network. The resulting graph acts as an information repository for various hydrologic models. How to manipulate the data depends on the particular application (model) which employs graph theory algorithms to access, retrieve and analyze the data. We believe our approach separates data and algorithms in a way that encourages modular design and can be adapted by a variety of hydrologic models.

We present a simple example (Figure 5) of a flow direction grid in Figure 4 to illustrate the concepts introduced above. The grid can be divided into three basins: A, B and C. Cells in each basin flow to its individual outlet cell. Basin A is in dark grey and has two cells with cell number 3 as the outlet. Basin B is in grey and has nine cells with cell number 11 as the outlet. Basin C is in white and has only one cell, cell number 12. It is also the outlet. The right side of the figure shows the rooted trees (forest) that represent the basin drainage network. Each tree represents an independent basin. The outlet cells of each basin 3, 11 and 12 are the roots of basins A, B and C, respectively.

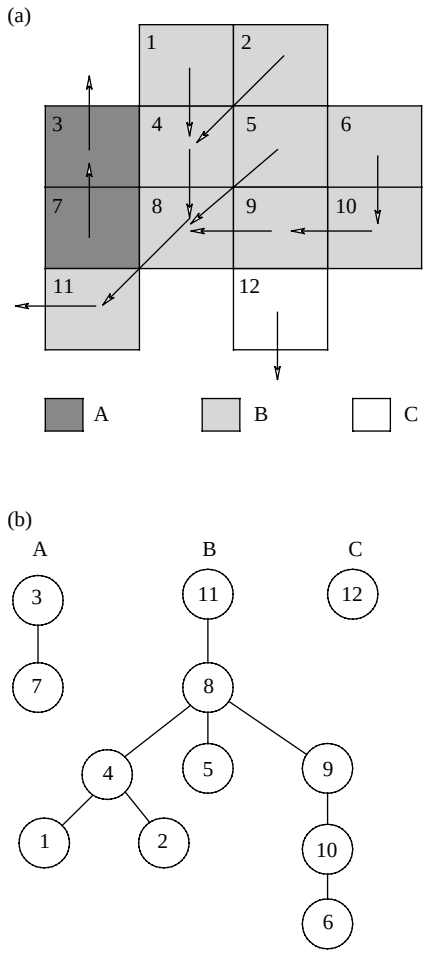

Figure 5 | A small simplified drainage network: the connectivity (left) and the graph representation (right).

Although there is only one cell in basin C by definition, it is a graph having only one vertex without edges i.e. a null graph. The drainage network is a sparse graph because the rooted tree is sparse.

Using graph walks to control processing order

Distributed hydrologic model algorithms need to visit the grid cells in a particular order. For example, to route flows on each grid cell to the outlet cell, we need to visit (calculate) upstream cells before downstream cells. The flow direction grid specifies this kind of order: the hydrologic connectivity order which describes the progression of water flowing along topographic gradients through the watershed to the outlet. In graph theory, the graph walk is the process of visiting each vertex through

the edges. In particular, the Postorder Tree Walk algorithm can be used for this purpose. The Postorder Tree Walk can be implemented by a recursive function in that the root (downstream cell) of a subtree is visited after its children (upstream cells). Recursive functions are functions that call themselves. They are simpler than iterative functions and thus are easier to understand. The following pseudocode describes the algorithm of the Postorder Tree Walk.

```
Postorder-Tree-Walk(Node n)
//visit children first
for each child of n
Postorder-Tree-Walk(child)
end for
//then visit the root
visit n
return
```

Meanwhile, it is also necessary to visit the grid cells in the reverse order: visit the downstream cells before the upstream cells. An example is finding the drainage area for a given grid cell, but excluding grid cells in a sub-basin. This is equivalent to finding all upstream cells (children) for a given grid cell (root). The Preorder Tree Walk algorithm visits the root of the tree before visiting each of its children. In other words, the downstream cells are visited before their upstream cells. This algorithm can therefore find all upstream sub-basins for a given cell in the downstream to upstream order (the reverse connectivity order). The following is the pseudocode of the Preorder Tree Walk algorithm.

```
Preorder-Tree-Walk(Node n)

//visit the root first
visit n
//then visit each child
for each child of n
Preorder-Tree-Walk(child)
end for
return
```

The Preorder Tree Walk and Postorder Tree Walk will visit the grid cells all the way to the leaves (cells which have no incoming neighbours). However, sometimes it is necessary to divide a connected basin into several sub-basins. In this case, the outlet cell of a sub-basin could be an incoming cell of another sub-basin. Therefore one sub-basin is a dependent of another sub-basin. Figure 6 shows an example of a connected basin. There are two sub-basins, B and D. Basin B depends on Basin D because one of its boundary cells (cell number 8) has an incoming cell (cell number 4) which is the outlet of Basin D. The flow routed from sub-basin D flows to sub-basin B. The question is how to visit sub-basin B without visiting cells in Basin D. The previous two algorithms will not work because cells from both basins will be visited if we start from cell 11.

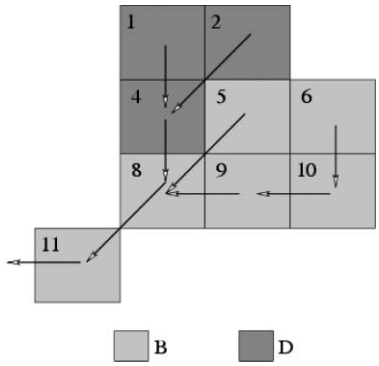

Figure 6 | Example of nested basins.

In order to make a partial visit (i.e. visiting only grid cells in Basin B but not D by starting at cell 11), we modify the two tree walk algorithms by adding a test before visiting each child to see if it is an outlet of another sub-basin. If it is an outlet, the algorithm will skip this child; otherwise, the algorithm will continue as usual. We call these algorithms Partial Preorder or Postorder Tree Walk. The following is the Partial Preorder Tree Walk algorithm's pseudocode. The Postorder Tree Walk can be modified similarly to make a Partial Postorder Tree Walk.

```
Partial Preorder-Tree-Walk(Node n)

//visit the root first
visit n
//then visit each child
for each child of n
//skip the sub-basin
if (child is not an outlet)
Partial-Preorder-Tree-Walk(child)
end for
return
```

An example of the usage of the partial visiting algorithm is to determine the drainage area of Basin B excluding Basin D for the given grid cell 11 in Figure 6.

Originally, the distributed hydrologic model developed at the NWS used one-dimensional arrays which store the downstream grid cell number for each grid cell. This approach was based on that used in the Nile Forecast System (Koren & Barrett 1994). While a one-time sorting of these pixels can yield a computationally efficient model for routing calculations that must be repeated for many time steps, a graph-based approach provides a more flexible solution which can be more efficient for some tasks. By relying on industry standard libraries, the graph-based approach also results in simpler codes that are easier to maintain.

An example where efficiency can be improved is the problem of finding all upstream cells for a particular grid cell, which is required by many operations such as finding drainage area and routing. With the 1D array implementation, we need a loop to search the entire array for this problem. In the following example, each element of the array *downstream*[*i*] is the downstream cell of i. A negative value indicates the cell is an outlet.

```
upstream-cells(Node n)

for each node in downstream
if (downstream[node] = = n)
node is one of upstream cells of n
break
end for
```

The time complexity of this approach is of order $O(n)$, while using an adjacency list (graph) instead of the array *downstream*[*i*], the time complexity is constant: $O(1)$. Although the array implementation of hydrologic connectivity has the advantage of fast access to the downstream cell for each cell with a time complexity of $O(1)$, it is still not convenient to route water from upstream cells to the outlet cell. As a result, our original distributed model requires that the connectivity must be pre-sorted by a pre-processor before running the model. In other words, for the array elements of *downstream*[*i*] and *downstream*[*j*], if cell *i* is upstream of cell *j* then *i* must be less than *j*. Nevertheless, a possible algorithm to sort the cells into connectivity order using the *downstream*[*i*] array with random cell order could be as follows.

```
connectivity-order(downstream[n])

int ordered[n];
bool already-sorted[n];
for (int i = 0; i < n; i++)
{
already-sorted[i] = false;
}
int count = 0;
do{
bool finished = true;
for(int i = 0; i < n; i++)
{
bool hasParent = false;
for(int j = 0; j < n; j++)
{
if(downstream[j] = = i && !already-sorted[j])
{
hasParent = true;
break;
}
}
if(!hasParent && ! already-sorted[i])
{
ordered[count++] = i;
already-sorted[i] = true;
}
}
for(int i = 0; i < n; i++)
{
if(!already-sorted [i])
{
finished = false;
break;
}
}
} while(!finished);
```

This algorithm involves three nested loops. Not only does it involve more programming efforts (more lines of code are written) compared to the Postorder-Tree-Walk function, but it is also inefficient: the time complexity is of the order $O(n^3)$, i.e. polynomial time instead of linear. Using the Postorder-Tree-Walk, the time complexity is only $O(n)$ because each node is visited once. To set up the graph object from a list of n nodes and m edges, the time complexity is $O(n + m)$. Therefore, the graph theory-based algorithms are two orders of magnitude more efficient than our original routing algorithm.

Choice of data structure

The choice of data structure depends on the density of the graph and what kinds of operations can be performed efficiently on the graph. For example, if edges are frequently being accessed, added or removed, such as in the case of editing a flow direction network, the matrix structure should be used to take advantage of the fast access time. Otherwise, if we need to access the vertices quickly and seldom delete or insert an edge, then the adjacency-list would be the choice. For a distributed hydrologic model, the drainage network is usually static, and the grid cells should be accessed quickly to calculate runoff and flow for each of them. For this reason, we chose the adjacency-list implementation of graph as the data structure. The vertices and edges are implemented using vectors to achieve fast access. Moreover, an adjacency-list saves more space than an adjacency-matrix because the drainage network is a sparse graph. An edge-list implementation is not suitable for distributed hydrologic connectivity because vertices can only be accessed via the corresponding edges.

PARALLEL COMPUTATION

Today, parallel computing has become a common technique to improve performance. CPU clock speeds are being pushed to the limit by manufacturers. Consequently, there is little room to improve CPU clock speeds and therefore reduce computing costs. As a result, more and more CPU manufacturers offer multiple core processors instead of processors with faster clock speeds. This trend has a profound impact on programming approaches. Traditionally, programs will run faster on newer hardware because of faster CPU clock speeds. However, with multi-core CPUs that have unimproved clock speed, programs that are not tuned to utilize the multi-core CPUs by parallel programming will not realize any performance gains. Today's programmers are therefore forced to develop parallel programs if they want their program to run faster on newer hardware (Sutter 2005).

Distributed hydrologic modelling normally involves a large amount of spatially distributed data which results in a heavy computation workload. Parallel computation becomes a vital tool to improve distributed hydrologic model performance. Apostolopoulos & Georgakakos (1997) proposed a parallel algorithm for distributed hydrologic models that assumes a shared memory environment–an ENCORE parallel computer with 14 processors and the Encore Parallel FORTRAN compiler (EPF). Under EPF the programmer has no control of the workload allocated to each processor, but only the number of processors to be used. Moreover, the resulting code of EPF is not portable to distributed parallel architectures and is not scalable with the number of processors. Cui *et al.* (2005) parallelized a distributed hydrologic model in a message passing environment. However, only the water balance part of the model is parallelized. The dependence of the connected drainage network is therefore not considered in the research carried out by Cui *et al.* (2005).

Here we assume a message passing environment. Message passing parallel software is more scalable and portable than shared memory software (Dowd & Severance 1998; Berthou & Fayolle 2001). In a message passing environment, a number of processors could either be connected by a high-bandwidth network or coupled closely to each other, such as the multi-core architecture. Each processor can either have its own memory or share a common memory with others. The programmer has full control over the workload of each processor. The importance of workload is discussed below.

As stated above, the flow channel network describes the order in which each grid cell's runoff will be routed to the outlet. That is to say there are dependencies among the grid cells in a drainage network. To utilize multiple processors we shall determine which cells can be calculated simultaneously given the grid cell dependencies. As a simple example, we discuss only one basin in which all pixels drain to only one outlet. Such a basin can be represented by a single rooted tree like the basin B in Figure 5. Note that situations with multiple basins, each draining to different outlets, can easily be modelled using multiple processors if the single tree problem is solved.

Referring to Figure 5, the routing process starts from the most upstream grid cells which have no incoming grid cells (leaves) and then proceeds to their downstream cells and then all the way to the outlet cell. Because each grid cell has only one path to the outlet (root), the starting upstream grid

cells are also the farthest cells from the outlet along the path. Therefore the problem of which cells can be calculated simultaneously is equivalent to determining the distance of each grid cell to the outlet along its path. Starting from the farthest cells, cells at the same distance can be calculated simultaneously after all cells at greater distances along the path have been calculated.

The problem of finding the distances of all grid cells to the outlet cell is the same as the single-destination shortest path problem in a rooted tree. The problem is to find all the shortest paths from every vertex in the graph to one vertex, the equivalent of the single-source problem. In the single-source problem, the shortest path is measured by a weight; each edge has a weight $w(v_{i-1}, v_i)$ and the weight of a path $w(p)$ is the sum of all edge weights along the path:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \tag{1}$$

The shortest-path weight from vertex u to v is then

$$\Delta u,v = \begin{cases} \min(w(p)) : u \rightarrow v & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases} \tag{2}$$

The path with minimum weight is the shortest path. When considering only computational order, all edge weights may be set equal to 1 in a drainage network even if the true element sizes vary with location. Because there is one and only one path from each node to the root node, the distance of each node to the root is also the shortest/longest distance.

In graph theory, Dijkstra's algorithm is used to find the shortest path for each node to the root (Cormen *et al.* 1989). Dijkstra's algorithm finds all the shortest paths from the source vertex to every other vertex by iteratively 'growing' the set of vertices S to which it knows the shortest path. Initially, the set S is empty. Another set D contains the length of the shortest path for each vertex, which is initialized to infinity for all the vertices except the source vertex which is zero. At each step of the algorithm, the next vertex added to S is determined by the set Q that contains the vertices in $V - S$ prioritized by their distance in D, which is the length of the shortest path seen so far for each vertex. The vertex u with minimum distance in the set Q is then added to S and removed from Q, and each of its out-edges is relaxed: if the distance to u plus the weight of the out-edge (u, v) is less than the distance for v in D then the estimated distance for vertex v is reduced and D is updated. The algorithm then loops back, processing the next vertex that has the minimum distance in Q. The algorithm finishes when the set Q is empty.

We present Dijkstra's algorithm as follows. Here G is the graph, w is edge weight, s is the start vertex and $d[u]$ is the return value of the algorithm which contains the distances of all vertices to the start vertex. The subroutine *Extract_Min*(Q) searches for the vertex u in the vertex set Q that has the least $d[u]$ value. That vertex is removed from the set Q and returned. The set *previous*[v] records the previous vertex along the shortest path for vertex v.

```
Dijkstra(G, w, s)

for each vertex v in V[G] //initialization
d[v]: = infinity
previous[v]: = nil
end for
d[s]: = 0
S: = empty
Q: = V[G]
while Q is not an empty set //the algorithm
u: = Extract_Min(Q)
S: = S union {u}
for each edge (u,v) outgoing from u
if d[u] + w(u,v) < d[v]
d[v]: = d[u] + w(u,v)
previous[v]: = u
end if
end for
end while
```

Table 1 shows the results of applying Dijkstra's algorithm to Basin B in Figure 5. The distance from the outlet is 4, which is the maximum distance. It needs 5 steps to finish the calculation. First the farthest grid cell 6 is calculated; next 1, 2 and 10 are calculated simultaneously; then 4, 5 and 9; then 8; finally the root 11 is calculated. Table 2 lists the cell numbers on each processor in each

Table 1 | Applying Dijkstra's algorithm to basin B in Figure 5

Node u	1	2	4	5	6	8	9	10	11
$d[u]$	3	3	2	2	4	1	2	3	0

Table 2 | Cell number on each processor in each step

Step	1	2	3	4	5	Total workload
Processor 1	6	1	4	8	11	5
Processor 2	Idle	2	5	Idle	Idle	2
Processor 3	Idle	10	9	Idle	Idle	2

time step when assuming three processors are used in the simulation.

Parallel computing performance can be measured by speedup and efficiency. Speedup (S_n) is the ratio of the run-time of the sequential algorithm using one processor to the run-time of the parallel algorithm using multiple processors:

$$S_n = \frac{T_1}{T_n} \tag{3}$$

where T_1 is the run-time of the sequential algorithm and T_n is the run-time of the parallel algorithm executed on n processors.

The efficiency E_n is the speedup divided by the number of processors used in a run of the parallel algorithm:

$$E_n = \frac{S_n}{n} = \frac{T_1}{nT_n} \tag{4}$$

where n is the number of processors.

Speedup and efficiency are affected by many factors such as inter-processor communication bandwidth, I/O bottle neck, etc. However, one of the most important factors is the load balance among processors. In the above example, assuming that three processors are used (Table 2) and there are no communication and I/O costs, it takes five units of time to finish the simulation because there are five steps when the three processors can run simultaneously in step 2 and 3; meanwhile the run-time of using one processor is nine units of time because there are nine grid cells. Therefore the speedup is $9/5 = 1.8$ and the efficiency is $9/(3 \times 5) = 0.6$. The process is only 60% efficient because the workload of each processor is not balanced very well. The last column in Table 2 shows the workload of each processor. Processor 1 has the greatest workload of five grid cells, while processors 2 and 3 have only two grid cells each. As shown in steps 1, 4, and 5, two processors have to wait for the other processor to finish; their time is wasted.

Because of the dependency of grid cells, the workload of the processors cannot be balanced. Particularly during the last step, the outlet can be assigned to only one processor while other processors are idling. Due to this limitation, the speedup and efficiency are not ideal even though we assumed ideal zero inter-processor communication and I/O costs. Moreover, the resulting parallel algorithm is not scalable. In other words, increasing the number of processors may reduce the speedup and efficiency. It is possible to optimize the number of processors to be used to achieve maximum speedup and efficiency.

To overcome the unbalanced workload limitation, an algorithm decomposition strategy could be an alternative to the data decomposition strategy described here. The decomposition strategy introduced here is called data decomposition, in which each processor runs the same algorithm but uses different datasets. The algorithm decomposition strategy assigns the same dataset to each processor, but each processor runs a different algorithm or a different part of the same algorithm. Therefore, algorithm decomposition may produce a more balanced workload on each processor if the algorithm could be distributed evenly. However, it requires more programming efforts and may not be as scalable as data decomposition. Usually, algorithm decomposition results in complicated parallel algorithms (Dowd & Severance 1998).

CASE STUDY

Implementing the parallel algorithm

The parallel algorithm discussed in the previous section was implemented into a routing model. This implementation uses the Message Passing Interface library, openMPI. As discussed, the distances from each grid cell to the outlet cell is first calculated. Starting from the longest distance, the flow on the grid cells at the same distance are computed simultaneously. The grid cells within the same distance are allocated to the available processors in a manner such that the workload of each processor is as even as possible. After all processors have done their jobs at the current distance and before starting the jobs at the next distance, each processor first find out who will need its results for the computation at the next distance and then sends its results.

After sending results, the processor will also find out from whom it will receive data for its own computation at the next distance and then do the receiving. After sending and receiving, the processors will have outflow from upstream cells and resume the computation at the next distance. This process is repeated until the working distance is zero where the root (outlet) cell is processed.

This communication step is necessary because the processors must know the outflow value from upstream cells before proceeding to the next level in distance; however, the upstream cells might have been assigned to another processor. Thus the results have to be obtained from the other processors who computed the upstream cells. To minimize inter-processor communication, the MPI point-to-point method is used in which only processors involved in the computation will do the communication. If a processor is not assigned a job or other processors do not need its computation results, it will not participate in the communication activities.

Hardware

The simulation is conducted on a distributed memory cluster that has 33 compute nodes. Each compute node has two dual-core AMD Opteron processors and 13 GB of memory. The compute nodes are connected by an Infini-Band switch, which is a fabric communication link used to connect the compute nodes in high-performance computing. The number of processors used in this study is varied from 1 to 16 to study the performance of the algorithm.

Simulation dataset

We chose the Oklahoma Illinois River Basin at Tahlequah as the test basin (Figure 7) because it is one of the basins recently modelled in the US NWS Distributed Model Intercomparison Project (Smith *et al.* 2004). This basin has a drainage area of 2,484 km^2. The connectivity file contains 151 grid cells. Each grid cell has a size of 4 × 4 km. The longest distance from the outlet cell calculated by Dijkstra's shortest distance algorithm is 32 grid cells. The distribution of the grid cells at different distances is shown in Table 3.

Figure 7 | Location of the Illinois River Basin.

The precipitation data is also in a gridded format with a resolution of 4 × 4 km obtained from the NWS NEXRAD Stage III hourly precipitation archive. Temporal resolution is one hour of precipitation data. The simulation was conducted from 1 October 1995 to 30 September 2002.

RESULTS

Figure 8 is the resulting discharge from the simulation compared to the observed discharge values at Tahlequah station of the Oklahoma Illinois River basin. The simulated discharges agree well with observed values.

Figure 9 shows both the wall clock time of average runtime and average inter-processor communication time of the parallel routing model for the number of processors ranging from 1 to 16. The total runtime was reduced from 140 seconds on 1 processor to 60 seconds on 16 processors. The reduction of runtime is more significant at a small number of processors (less than 6) than at a large number of processors (greater than 6). This pattern is also shown in Figure 10 where higher speedups were achieved when the number of processors is less than 6. The runtime did not decrease significantly and was relatively constant when the number of processors is greater than 6. There was a slight increase in runtime when the number of processors increased from 3 to 4 and 7 to 8. The inter-processor communication time increased from a near-zero value to 23 seconds on 4 processors and remained relative constant thereafter. Although the inter-processor communication did not increase after 6 processors, the runtime did not decrease when the number of processors increased. This means that the computation time did not decrease as the number of processors increased when the number of processors is greater than 6. The parallel routing algorithm is only scalable up to 6 processors. At 4 and 8 processors, the

Table 3 | Number of grid cell at each distance

Distance	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
No. of cells	1	2	3	2	3	2	1	1	2	2	4	2	2	4	3	5	8
Distance	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	
No. of cells	7	9	7	5	4	5	5	7	10	17	11	8	3	3	2	1	

communication time is about 23 seconds which is one of the largest communication times. This could contribute to the increased runtime at 4 and 8 processors.

Figure 10 shows the speedup of the parallel routing model up to 16 processors. As the number of processors increased, the speedup first increased and the maximum speedup was about 2.2 at 16 processors. However, the increase was not significant when number of processors is greater than 6.

DISCUSSION

The parallel routing has limited scalability as shown in Figures 9 and 10. The runtime did not decrease and the speedup did not increase significantly when the number of processors is greater than 6. Perhaps two reasons contribute to the scalability.

The first reason is the overhead time due to inter-processor communications. For example, at 6 processors, the average communication time is about 20 seconds which consists of 29% of total runtime, 70 seconds. The downstream cells depend on upstream cells; the parallel algorithm has to coordinate the processors to communicate to others at each distance for processors to obtain necessary data before proceeding to the next level of distance. The number of communications is proportional to the maximum distance to the outlet of the connectivity. The maximum distance of the Illinois basin is 32 units, thus the algorithm will coordinate the processors 32 times at each distance to send and receive data. When the number of processors increases, the communication required at each of the 32 distances increases because more processors are involved and therefore the overall communication time increases.

This experiment was conducted on a cluster where memories are distributed among the compute nodes connected by a network. The performance would be better if a shared-memory SMP computer was used. The SMP is a multiprocessor computer architecture where two or more identical processors are connected to a single shared

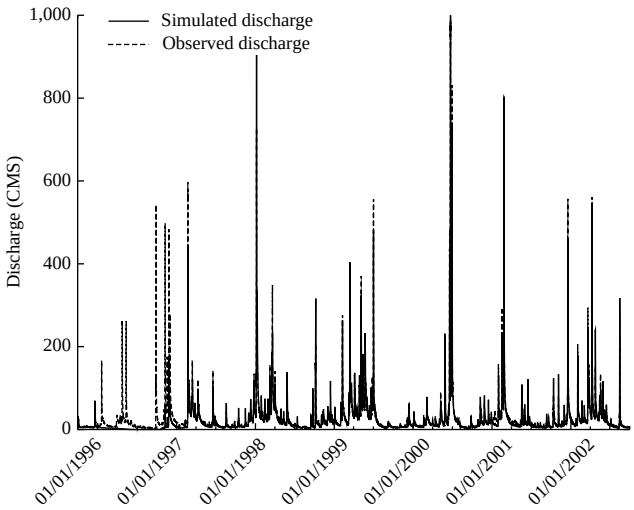

Figure 8 | Simulated versus observed discharge at Tahlequah, the Oklahoma Illinois River basin.

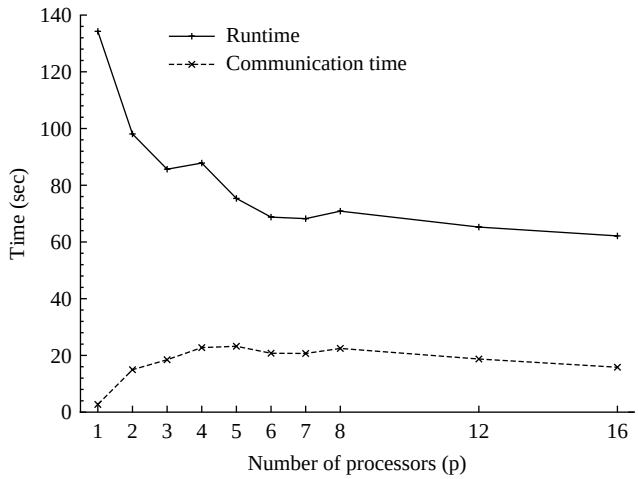

Figure 9 | Simulation runtime and communication time versus number of processors.

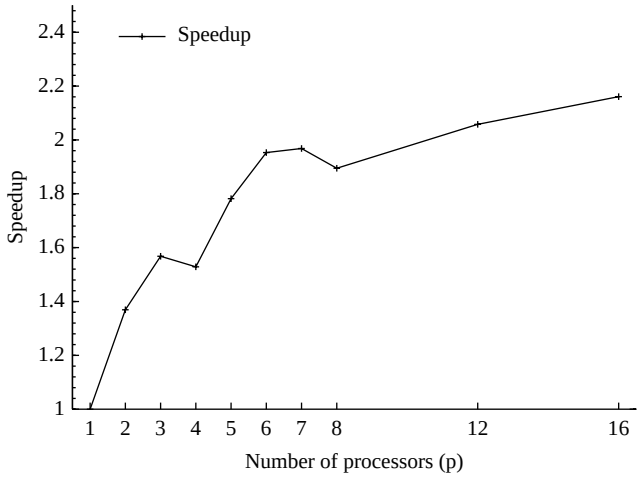

Figure 10 | Speedup on various numbers of processors.

memory. The inter-processor bandwidth is much higher than the distributed memory cluster.

The second reason for a limited scalability is the unbalanced workload as discussed in the previous section. As shown in Figure 9, the runtime and communication time curves are almost parallel from 6 to 16 processors. It indicates that the computation time remained constant even though more processors participated in computation.

The number of grid cells at each distance varies greatly from 1 to 17 as shown in Table 3: 4 distances have 1 grid cell, 8 distances have 2 grid cells and 5 distances have 3 grid cells, etc. The greater the numbers of processors, the more often the processors are idling at the same distances. The idling processors at these distances are just wasting resources and cannot proceed until the busy processors finish their jobs. The results are low speedups and poor scalability. Table 4 lists the distribution of jobs at each distance for each processor.

Here we use the ratio of maximum to minimum workload called job ratio to measure the degree of workload balance. When the workloads are perfectly balanced, the job ratio is 1. The larger the ratio, the more the workloads are unbalanced. In the above example of 4 processors, the job ratio is 52/24 = 2.166. It also can be seen from Table 4 that Processor 4 has more than the twice the workload of Processor 1. Figure 11 shows the job ratio versus the number of processors for the Illinois River Basin from 1 processor to 16 processors.

Table 4 | Workload on each processor at each distance with 4 processors

Distance	Processor 1	Processor 2	Processor 3	Processor 4	Total
0	0	0	0	1	1
1	0	0	1	1	2
2	0	1	1	1	3
3	0	0	1	1	2
4	0	1	1	1	3
5	0	0	1	1	2
6	0	0	0	1	1
7	0	0	0	1	1
8	0	0	1	1	2
9	0	0	1	1	2
10	1	1	1	1	4
11	0	0	1	1	2
12	0	0	1	1	2
13	1	1	1	1	4
14	0	1	1	1	3
15	1	1	1	2	5
16	2	2	2	2	8
17	1	2	2	2	7
18	2	2	2	3	9
19	1	2	2	2	7
20	1	1	1	2	5
21	1	1	1	1	4
22	1	1	1	2	5
23	1	1	1	2	5
24	1	2	2	2	7
25	2	2	3	3	10
26	4	4	4	5	17
27	2	3	3	3	11
28	2	2	2	2	8
29	0	1	1	1	3
30	0	1	1	1	3
31	0	0	1	1	2
32	0	0	0	1	1
Total	24	33	42	52	151

The job ratio at 6, 7 and 8 processors is about 5 and quickly increases to 34 at 12 and 16 processors. The unbalanced jobs at high number of processors limited the parallel scalability.

The job ratio is also controlled by the distribution of grid cells along the depth of the tree. There are 151 grid cells

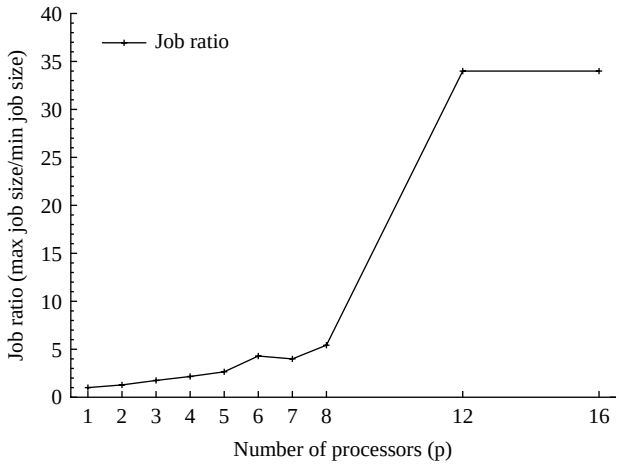

Figure 11 | Maximum to minimum workload ratio on various numbers of processors.

all together which is a relatively small number, but the maximum distance is 32. The largest number of grid cells at one distance (17) occurred at distance 26. The job would have been more balanced if there were more grid cells at each distance and the maximum distance was shorter.

CONCLUSIONS

This paper discusses the use of graph theory for distributed hydrologic models that use a drainage network derived from DEM data. Graph theory has a rich set of well-studied algorithms that can be used by scientists to facilitate their modelling efforts and avoid duplicating past work. The two most commonly used algorithms, the Postorder and Preorder Tree Walk, and their variations are presented. These are used to visit the grid cells in particular orders.

Three data structures (adjacency-matrix, adjacency-list and edge-list) can be used to implement a graph. The vertex and edge set can be implemented as a vector or a linked-list. Each of these structures has different trade-offs on performance and storage.

The graph-based implementation of a drainage network and algorithms discussed in this paper are applicable to distributed hydrologic models that have structures similar to regular grids and TINs. The authors have implemented drainage networks using the adjacency-list structure in both the research and operational versions of the NWS distributed model. The graph-based architecture provides both scientific and system software developers with ready-to-use algorithms in a generic object-oriented development environment. The resulting sequential computer codes are efficient, extensible and reusable.

As parallel computing becomes more commonplace on today's computing hardware, a parallel algorithm for the routing model has been developed. The parallel algorithm is based on Dijkstra's shortest-path algorithm which determines which cells are to be calculated simultaneously in a drainage network at each distance. However, because of the dependencies in the drainage network, upstream cells must be computed before downstream cells. Consequently, workload on the multiple processors is very difficult to balance.

The parallel algorithm has been implemented in a distributed hydrologic routing model and run on a cluster using the Oklahoma Illinois River basin dataset. The results show that inter-processor communication and unbalanced workload are the bottlenecks of the performance due to the nature of the problem being solved. Because of the dependency of the grid cell in the hydrologic connectivity, the computation on grid cells must start from the grid cells at the greatest distance from the outlet, one distance at a time and progressively to the outlet cell. The inter-processor communication is required at each distance and the workload of the processors is highly unbalanced as the number of processors increases. Thus, the scalability is limited.

The inter-processor communication plays a key role for the parallel algorithm presented in this paper. A shared-memory SMP computer would improve the scalability considerably because of high inter-processor bandwidth. On the other hand, the relative small number of grid cells and the long maximum distance to the outlet cell produce high job ratios that will deteriorate scalability. The job ratios could be improved for basins with a large number of grid cells at each distance. We conclude that the parallel algorithm is more applicable to computers with high inter-processor bandwidth and basins where the number of grid cells is large and the maximum distance of the grid cells to the outlet is short.

Algorithm decomposition is another strategy to overcome the inter-processor communication and unbalanced workload limitation. However, algorithm decomposition requires a greater programming effort to decompose the system of equations, and usually results in complicated

parallel algorithms. Other considerations such as computer architectures, inter-processor communications and parallel I/O will also affect the performance of the parallelized system, and should be taken into consideration when developing parallel algorithms for distributed hydrologic models.

ACKNOWLEDGEMENTS

The authors thank the anonymous reviewer for their valuable comments. Reviewer #5 also contributed ideas on improving the performance of the parallel algorithm.

REFERENCES

Apostolopoulos, T. K. & Georgakakos, K. P. 1997 Parallel computation for streamflow prediction with distributed hydrologic models. *J. Hydrol.* **197**, 1–24.

Bailly, J., Monesez, P. & Lagacherie, P. 2006 Modelling spatial variability along drainage networks with geostatistics. *Math. Geol.* **38** (5), 515–539.

Berthou, J. & Fayolle, E. 2001 Comparing OpenMP, HPF, and MPI programming: a study case. *Int. J. High Perform. Comput. Appl.* **15** (3), 297–309.

Cantwell, M. D. & Forman, R. T. T. 1993 Landscape graphs: ecological modeling with graph theory to detect configurations common to diverse landscapes. *Landsc. Ecol.* **8** (4), 239–255.

Carter, G. C. 2002 Infusing new science into the National Weather Service River Forecast System. *Second Federal Interagency Hydrologic Modeling Conference*, Las Vagas, Nevada, p. 10.

Coffman, D. M. & Turner, A. K. 1971 Computer determination of the geometry and topology of stream networks. *Water Resour. Res.* **7** (2), 419–423.

Cormen, T. H., Leiserson, C. E. & Rivest, R. L. 1989 *Introduction to Algorithms*. The MIT electrical engineering and computer science series, MIT Press, (ISBN 0-262-03141-8).

Cui, Z., Vieux, B. E., Neeman, H. & Moreda, F. 2005 Parallelisation of a distributed hydrologic model. *Int. J. Comput. Appl. Technol.* **22** (1), 42–52.

Cui, Z., Koren, V., Moreda, F. & Smith, M. 2006 A common programming framework for distributed hydrologic modeling research: an overview of the architecture. *Third Federal Interagency Hydrologic Modeling Conference*, Session 5E, Las Vegas, Nevada, April 2–6.

Dowd, K. & Severance, C. R. 1998 *High Performance Computing*. O'Reilly & Associates, Inc, Sebastopol, CA.

Fairfield, J. & Leymarie, P. 1991 Drainage networks from grid digital elevation models. *Water Resour. Res.* **27** (5), 109–717.

Gleyzer, A., Denisyuk, M., Rimmer, A. & Salingar, Y. 2004 A fast recursive GIS algorithm for computing Strahler stream order in braided and nonbraided networks. *J. Am. Water Resour. Assoc (JAWRA)* **40** (4), 937–946.

Gupta, R. & Prasad, T. D. 2000 Extended use of linear graph theory for analysis of pipe networks. *J. Hydraulic Eng.* **126** (1), 56–62.

Jenson, S. K. & Domingue, J. O. 1988 Extracting topographic structure from digital elevation data for geographic information system analysis. *Photogramm. Eng. Remote Sens.* **54**, 1593–1600.

Jiang, B. & Claramunt, C. 2004 A structural approach to the model generalization of an urban street network. *GeoInformatics* **8** (2), 157–171.

Koren, V. & Barrett, C. B. 1994 A satellite based river forecast system for the Nile River. *Proceedings of the 21st Annual Conference sponsored by the Water Resources Planning and Management Division, ASCE, held in Denver, Colorado, May 23–26, 1994*.

Koren, V., Reed, S., Smith, M., Zhang, Z. & Seo, D. 2004 Hydrology laboratory research modeling system (HL-RMS) of the US national weather service. *J. Hydrol.* **291**, 297–318.

Kreyszig, E. 1998 Graphs and combinatorial optimization. In *Advanced Engineering Mathematics* (ed. E. Kreyszig), 8th edition. John Wiley & Sons.

Lee, L., Siek, J. G. & Lumsdaine, A. 1999 The generic graph component library. *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages and applications, Denver, Colorado, United States, 399–414*, ISBN:1-58113-238-7.

Mark, D. M. 1988 Network models in geomorphology. In *Modelling Geomorpholgical Systems* (ed. M. G. Anderson), pp. 73–97. John Wiley & Sons Ltd.

O'Callaghan, J. F. & Mark, D. M. 1984 The extraction of drainage networks from digital elevation data. *Comput. Vision Graphics Image Process* **28**, 323–344.

Park, J., Obeysekera, J. & Vanzee, R. 2005 Graph theory data objects applied to stream flow network representation in an integrated hydrological model. American Geophysical Union, Spring Meeting 2005, 2005AGUSM.H21C.01P.

Pfafstetter, O. 1989 Classification of hydrographic basins: coding methodology. Unpublished manuscript, Departamento Nacional de Obras de Saneamento, August 19, 1989, Rio de Janeiro, Brazil [Translated by J.P. Verdin, U.S. Bureau of Reclamation, Denver, Colorado, September 5, 1991].

Reed, S. M. 2003 Deriving flow directions for coarse resolution (1–4 km) gridded hydrologic modelling. *Water Resour. Res.* **39** (9), 1238.

Reynolds, J. F. & Wu, J. 1999 Do landscape structural and functional units exist? *Integrating Hydrology, Ecosystem, Dynamics, and Biogeochemistry in Complex Landscapes* (ed. J. D. Tenhunen & P. Kabat),. John Wiley & Sons Ltd.

Scheidegger, A. E. 1967 On the topology of river nets. *Water Resour. Res.* **3** (1), 103–106.

Shreve, R. L. 1967 Infinite topologically random channel networks. *J. Geol.* **75**, 178–186.

Schroder, B. 2006 Pattern, process, and function in landscape ecology and catchment hydrology–how can quantitative landscape ecology support predictions in ungaged basins (PUB)? *Hydrol. Earth Syst. Sci. Discuss.* **3**, 1185–1214.

Smart, J. S. 1970 Use of topologic information in processing data for channel networks. *Water Resour. Res.* **6** (3), 932–936.

Smith, M. B. & Brilly, M. 1992 Automated grid element ordering for GIS-based overland flow modelling. *Photogramm. Eng. Remote Sens.* **58** (5), 579–585.

Smith, M. B., Seo, D. J., Koren, V. I., Reed, S. M., Zhang, Z., Duan, Q., Moreda, F. & Cong, S. 2004 The distributed model intercomparison project (DMIP): motivation and experiment design. *J. Hydrol.* **298** (1–4), 4–26.

Sutter, H. 2005 A fundamental turn toward concurrency in software. *Dr. Dobb's J.* **30** (3) (Available online: http://www.ddj.com/web-development/184405990).

Urban, D. & Teitt, T. 2001 Landscape connectivity: a graph-theoretic perspective. *Ecology* **82** (5), 1205–1218.

Verdin, K. L. 1997 A system for topologically coding global drainage basins and stream networks. *Proceedings of the Seventeenth Annual ESRI User Conference*, San Diego, California.

First received 21 March 2008; accepted in revised form 2 November 2009. Available online 13 April 2010