PERGAMON

Computers & Geosciences 26 (2000) 121–135

COMPUTERS & GEOSCIENCES

A recursive algorithm for connectivity analysis in a grid; application to 2D hydrodynamic modeling in heterogeneous soils[☆]

B. Cappelaere*, J. Touma, C. Peugeot

IRD–UMR HSM 5569, Montpellier, France

Received 15 October 1998; received in revised form 26 April 1999; accepted 26 April 1999

Abstract

A prerequisite for numerical simulation of water flow in heterogeneous soils is to build a discrete model of the soil matrix that is a fair representation of the heterogeneities under study, while being compatible with the numerical equations used to compute unsaturated water flow. When introducing significant amounts of very coarse solid elements (gravels) in a discrete, multi-dimensional soil matrix model, in the form of internal boundaries that occupy a certain fraction of the grid nodes, the need arises to eliminate from the model the possible occurrence of isolated areas of soil, surrounded by continuous gravel barriers that keep them separate from other regular grid nodes. This situation is obviously an artefact resulting from the discrete representation of the physical system, since all nongravel areas ought to be considered as being submitted to at least some hydrodynamic linkage with each other and with the outer boundary conditions (rain and other water input at the top, gravity drainage or water table at the bottom). Computational nodes that do not connect in some way, through the computational grid, to these outer boundary conditions, are the source of computational problems due to system indetermination when solving the hydrodynamic equations. A method has thus been devised to automatically detect and eliminate such situations in order to produce plausible soil models for hydrodynamic simulation, in the presence of highly contrasted grain sizes. The method is based on a proposed recursive algorithm for cluster analysis, an attractive and very simple alternative to existing methods generally used to handle cluster problems. This method and its application to the heterogeneous soil modeling problem is presented as a pseudo-code that can be implemented with any current programming language. Performance figures, and simulation results of the hydrodynamic behavior of such soil models, are shown. A parallel implementation of the algorithm is also proposed. © 2000 Elsevier Science Ltd. All rights reserved.

Keywords: Porous media; Unsaturated flow; Cluster analysis; Recursion; Finite differences

1. Introduction

Many computing problems in geosciences and other fields involve discretization of space as a multi-dimensional lattice that consists of a finite number of nodes and links. Connectivity of the nodes may be total, if there always exists a path made of a series of nodes

[☆] Pseudocode available at http://www.iamg.org/CGEditor/index.htm

* Corresponding author. Tel.: +33-4-67-41-61-00; fax: +33-4-67-54-71-06.

E-mail address: bernard.cappelaere@mpl.ird.fr (B. Cappelaere).

and links between any given couple of nodes, or else partial, leading to a clustered organization. While cluster analysis is generally performed using conventional iterative programming techniques, the use of recursion offers a very attractive means for solving this class of problems, both in terms of robustness and of ease to implement. A recursive object may be defined as one that contains a smaller, entirely similar 'sub-object'. In computer programming, recursion is the property of a program unit to call itself, either directly or through other units in the calling tree, allowing a task to be coded as the nesting of self-similar subtasks. This paper presents the development of such a recursive algorithm to perform cluster identification and labeling, for any grid topology. This algorithm is applied to the generation of 2D models of heterogeneous soils, needed for the numerical simulation of infiltration in such soils. The scope of this article is not to develop every aspect of this very complex modeling problem, but to focus on the connectivity analysis question, and to illustrate how it comes into play in the general soil simulation framework.

Accustomed to imperative, explicit programming style, many computational application developers are unfamiliar with declarative, implicit, artificial-intelligence-type algorithms, which can provide very simple, elegant, and powerful solutions to a lot of otherwise complicated problems. Fortran, a major language for computational applications, did not include recursion until the recent Fortran90. Recursion is now part of almost all modern languages and compilers, including Fortran77 compilers as a widespread language extension. With traditional languages, programmers have been used to thinking of repetitive operation sequences as explicit iterative programming structures (the famous do-loop), instead of considering the problem as a composition of self-similar subproblems. Only with very few languages such as LISP (see for instance Siklossy, 1976) has recursion been enforced as a major construct for problem formulation and programming. Such languages are largely foreign to the field of scientific computing, being rather restricted to artificial intelligence applications. Recursive solutions are very rarely, if ever, proposed in reference books in fields like geosciences. Evolution of computers in general and of compiler efficiency in particular, has largely eliminated the performance limitations that could have shadowed recursive programming in the past.

2. Presentation of the soil water modeling problem

Infiltration in unsaturated soils is a complex two-phase hydrodynamic process that takes place in solid matrices which themselves generally have particularly complex 3D, often deformable, geometries. The most widely accepted mathematical representation of macro-scale water movement in unsaturated rigid porous media is the Richards equation (1931):

$$C\frac{\partial h}{\partial t} = \operatorname{div}(K\ \overrightarrow{\operatorname{grad}}(h - z)) \qquad (1)$$

where t is time [T]; h is the soil water pressure [L] expressed in height of water column relative to the atmospheric pressure; z is the depth [L] below the soil surface, positive downwards; K is the hydraulic conductivity [L T^{-1}], a function of the volumetric moisture content θ [L^3 L^{-3}]; $C = d\theta/dh$ is the capillary capacity [L^{-1}], expressing the ability of a soil to absorb water by capillarity, obtained from the capillary retention curve $h(\theta)$. Analytical solutions seldom exist for real-world applications: Eq. (1) is therefore solved numerically through many possible methods and schemes that discretize both time and space. Soil characteristics are introduced as the $h(\theta)$ and $K(\theta)$ curves at each computational point, heterogeneity of the media being reflected in the mapping of various such curves over the domain under study. However, a common source of heterogeneity can hardly be taken into account through these characteristics curves, that is the presence in the media of very coarse-grained, quasi nonporous solid elements whose size is of the same order of magnitude as the spatial resolution of the numerical model. This is the case for instance when stones or coarse gravel are present in significant amounts in an otherwise finer grained soil. In such cases, the discretized form of Eq. (1) may no longer be considered to be valid at computational points that fall on or immediately next to such nonporous chunks; a better schematization is then to subtract these points (hereafter named gravel nodes) from the domain of Eq. (1), and view these as forming internal domain boundaries with a zero flux condition normal to the boundary. At least 2D representations are therefore needed to handle such soil-water systems, which at any rate do not easily lend themselves to mathematical modeling.

Heterogeneities, even when present in relatively small proportions, often have drastic impacts on the overall behavior of the system, depending on their spatial distribution. This brings the need for a careful treatment of heterogeneities in the soil-water model. A current research project is investigating numerically the effects on the infiltration and water redistribution processes, of various randomly-distributed proportions of gravel nodes over the computational grid, which simulate equivalent volumetric proportions of very coarse-grained material in the solid soil matrix. This project is still at its early stages of development, and therefore only qualitative indications of the possible behavior range are currently being sought using a 2D model, before more detailed, 3D analyses are performed. A finite-difference discretization of Eq. (1) is used, which writes:

$$C_{\mathrm{o}}^{k}\frac{h_{\mathrm{o}}^{k+1}-h_{\mathrm{o}}^{k}}{\Delta t}=\frac{q_{r/2}^{k+1/2}-q_{l/2}^{k+1/2}}{\Delta x}+\frac{q_{d/2}^{k+1/2}-q_{u/2}^{k+1/2}}{\Delta z} \qquad (2)$$

with:

$$q_{r/2}^{k+1/2}=K_{r/2}^{k}\frac{h_{\mathrm{r}}^{k+1}-h_{\mathrm{o}}^{k+1}}{\Delta x} \qquad q_{l/2}^{k+1/2}=K_{l/2}^{k}\frac{h_{\mathrm{o}}^{k+1}-h_{\mathrm{l}}^{k+1}}{\Delta x}$$

$$q_{d/2}^{k+1/2}=K_{d/2}^{k}\left(\frac{h_{\mathrm{d}}^{k+1}-h_{\mathrm{o}}^{k+1}}{\Delta z}-1\right)$$

$$q_{u/2}^{k+1/2}=K_{u/2}^{k}\left(\frac{h_{\mathrm{o}}^{k+1}-h_{\mathrm{u}}^{k+1}}{\Delta z}-1\right)$$

when applied to a 2D rectangular grid (Δt, Δx, and Δz are the time, horizontal and vertical steps, respectively; k is the time index; the space indices o, u, d, l, r designate the central node and its grid neighbors in the up, down, left and right directions, respectively; the /2 notation after one of these four directions refers to the point half-way between the central node and its neighbor in that direction; q is the water flux density [L T^{-1}]). Initial moisture distribution is uniform over the whole domain. A water pond of negligible depth is then applied at the soil surface during an infiltration phase, followed by a redistribution phase with a zero-flux upper boundary condition. At the lower boundary the pressure head is kept at its initial value, while a no-horizontal-flux condition is set at all lateral boundary nodes.

To build the discrete soil model, each node from the regularly-spaced grid is given the same chance of being a gravel node (or, conversely a regular, also called 'computational' node), by randomly drawing with a uniform probability equal to the globally-imposed gravel proportion *%gravel* (or, conversely, to its complement to 1). The problem may then arise, especially when working with 2D representations of the system, of a subset of regular grid nodes being isolated from all others by a barrier of surrounding gravel nodes. The corresponding subdomain no longer participates in the soil water dynamics, thereby artificially extending the volumetric proportion of the media that is represented as being nonporous. Simulated water movement would be identical, should these inactive computational nodes be replaced by gravel nodes. Hence, there may be a very significant difference between the modeled and the targeted systems, both in terms of gravel proportion and of the spatial distribution of these gravels (the model includes more clustered gravel nodes than would entail a pure uniformly-random distribution). Another troublesome consequence of having isolated computational nodes is the numerical problems that such peculiar nodes entail in the system solution process.

Likely in 3D models, this problem is highly important in the 2D case, due to the large probability of isolated node occurrence and to the significant share of each such node in the resulting simulated volumetric distribution between gravel and nongravel media. To carry out a proper analysis of the incidence of a given randomly-distributed proportion of gravel on the water infiltration and redistribution processes, it is necessary to identify and remove such undesirable situations from the soil model building procedure. Described hereafter is this process of synthesizing soil models that are free from any isolated node. A major component of it is the finding of such troublesome nodes in the computational grid, through a specially-developed connectivity analysis algorithm.

3. Analysis of the connectivity problem in 2D heterogeneous soil models

In this problem, connection of nodes is achieved by the network of links that associate couples of neighboring computational nodes in the discretized Eq. (2) of the hydrodynamic equation. More precisely, two grid nodes are defined as linked neighbors if and only if both are computational nodes (i.e. nongravel) and are immediately next to each other along the same grid line or column. Diagonals are excluded here, but the algorithm itself is totally independent of the number of potential neighbors, whether it be four as in our example, or eight or any other number depending on the problem being dealt with. As a matter of fact, it is independent of the grid dimension, structure and geometry.

The objective here is, given a random distribution of gravel nodes on the 2D grid, to identify all isolated zones of the grid. For our specific purpose, an isolated zone, or *island*, is defined as a subset of nongravel grid nodes that do not connect to either an upper boundary node or a lower boundary node, i.e. that do not span vertically across the entire lattice. This definition ensures that all regular nodes of the grid will be in computational linkage with both these boundary conditions. The algorithm would be very similar and easily deduced from the one presented hereafter, should the definition of islands be modified, with respect to the various domain boundaries. A nongravel node will be called *isolated* if belonging to an island, or otherwise *connected*. To produce this information, a symbolic array variable, named *state*, will be used, initially set to 'gravel' or 'unknown'; 'unknown' nodes are thus to be transformed by the algorithm into either ‘connected’ or ‘isolated’.

This problem can be analyzed as belonging to the general class of cluster identification problems. In our particular application, only certain clusters, the *islands*, are actually searched for. A possible problem decomposition approach consists of a first step of cluster labeling followed by a second step of selecting islands among labeled clusters. The latter step is straightforward, since it amounts to checking within each cluster whether or not at least one node of each boundary (upper and lower) is present. Therefore, this presentation will largely focus on the cluster labeling subproblem, a general problem potentially encountered in many broad fields such as percolation theory, simulation of spatial dynamic systems (e.g.: models for forest fires or vegetation dynamics), image processing, etc. Labeling is the preliminary step common to many various tasks including production of cluster statistics (count, size distribution, ...).

4. Cluster identification and labeling

4.1. Method

A widely used method for cluster multiple labeling is the Hoshen and Kopelman (1976) algorithm. This is an efficient technique for cluster labeling on regular grids, variants of which can be found in Stauffer and Aharony (1994, in Fortran), or Goult and Tobochnik (1996, in Basic). This method uses a clever two-pass iterative procedure over grid points, which requires a particular data structure consisting of a label-correction tree. It must be programmed with care to ensure foolproofness, and is not easily implemented for irregular grid configurations. A much simpler method can be devised when viewing the problem as recursive by essence[1]. Recursion makes cluster labeling a very straightforward issue, whatever the grid configuration. This is the method proposed and used here. It was not found in any of the examined books offering solutions to this kind of problem. Although cluster labeling is not a strictly necessary step for our particular problem (a recursive procedure, quite similar and just as simple as the one proposed for labeling, can be written specifically to directly answer the question of tagging grid points as ‘isolated’ or ‘connected’), we present this more general approach which can be used for numerous other applications.

Figure 1 presents as pseudo-code the recursion-based algorithm for the LABEL_CLUSTERS module. This module is totally independent of our particular problem, and can be reused as such for any cluster-labeling problem, whatever the grid configuration or cluster definition. Together with the *state* variable defining the nature of each grid node, a *selected_state* is input to the module, to define a particular class of nodes for which clusters are searched for. Setting the *selected_state* variable to ‘all’ indicates that all nodes are to be clusterized, whatever their *state* value (‘all’ should not be a possible *state* value). In this case, an individual cluster is still made only of nodes of the same nature, i.e. with the same *state* value. For our particular soil modeling problem, *label_clusters()* is always called with *selected_state* set to ‘unknown’. The LABEL_CLUSTERS module does not need to know what the possible values of the *state* variable are, which makes it application-independent. The module returns the number of clusters detected, *nclusters*, and the integer array *cluster_labels* over grid nodes. All nodes that belong to the same cluster have the same *cluster_labels* value, between 1 and *nclusters*, while all other nodes, which do not belong to any cluster, get the value 0.

The recursive function *label_neighborhood()* performs nearly all the algorithm’s work. It simply expresses the propagation of the property that a node belongs to a given cluster, to all its neighbors with identical *state*. This is a particularly striking example of the capacity of recursive programming to perform otherwise complex tasks through a very simple and general formulation of the problem to be solved. Another example would be the following, if the more direct approach for our soil modeling problem were used instead of going through the cluster labeling step: the recursive procedure would then consist in the forwarding to all similar neighbors of the question of a node being ‘isolated’ or not, until some node can answer (e.g.: a boundary node, defined as ‘connected’, or any node already tagged as ‘connected’) or until no node is left to be questioned. Since all questioned nodes ought to be granted the same *state* value, this should be ‘connected’ in the first case, and ‘isolated’ in the latter. With the direct scheme (i.e. no cluster labeling), the recursive process needs to be repeated as many times as there are boundaries to the soil model, since the ‘connected’ state requires an *AND* condition on connections to individual boundaries.

If some difficulty were to be mentioned with the implementation of recursive programming in general, this would reside in the need for making sure that the forwarding process never loops, and for proper handling

[1] It is interesting to point out that Goult and Tobochnik’s (1996) implementation of the Hoshen-Kopelman algorithm itself includes a tiny recursive piece that serves in the second pass of the algorithm to handle the tree structure already mentioned; trees are the most obvious candidates for recursion, but grid problems too, as well as many others, may be analyzed wholly in terms of recursive solutions.

```
module LABEL_CLUSTERS :
/* labels all nodes where the value of the state array is selected_state (all nodes if selected_state='all')
   labels are integers 1 to nclusters; nclusters is the number of clusters detected
   label value for each node is returned in integer array cluster_labels; contains 0 at unlabelled nodes
         (array is allocated before label_clusters() is called)
   function label_neighborhood() is the recursive engine of the algorithm */

     label_clusters(in: selected_state, state; out: nclusters, cluster_labels) {
           for all nodes, n :        {cluster_labels(n) = 0}
           nclusters=0
           for all nodes, n :        {
              if (node_needs_label_?(selected_state, state(n), cluster_labels(n))) {
                  nclusters=nclusters+1
                  label_neighborhood(n, state(n), state, nclusters, cluster_labels)
              }
           }
     }
     recursive label_neighborhood( in: node n, selected_state, state, label;
                                   out: cluster_labels) {
           cluster_labels(node n)=label
           for all neighbors of node n, nn :       {
              if (node_needs_label_?(selected_state, state(nn), cluster_labels(nn)))    {
                 label_neighborhood(nn, selected_state, state, label, cluster_labels)
              }
           }
     }
     logical node_needs_label_?(in: selected_state, node_state, node_label)    {
           node_needs_label_? =
              (node_label==0) and ((node_state==selected_state) or (selected_state='all'))
     }
```

Fig. 1. Algorithm for module LABEL_CLUSTERS.

of all particular conditions that bring this forwarding process to a stop (the so-called terminal subproblems; see for instance Meyer and Baudoin, 1984). In our cluster-labeling example, these issues are very simply managed, through the *node_needs_label_?()* logical function.

Translating the LABEL_CLUSTERS pseudo-code into source code for any compiler is totally straightforward, and produces a remarkably simple and condensed piece of program which can be reused or adapted to any user's needs, such as a different grid topology, with no difficulty or programming trap. This is due to the fact that the program itself directly expresses the basic principles of a very slim algorithm, which is not the case of the other methods quoted above. As a matter of fact, when using Goult and Tobochnik's (1996) algorithm, the program failed for large problems (more than ~45,000 clusters, a magnitude that can theoretically be reached for instance with a 300×300 grid, and that we observed for 600×600 grids or larger with random draws), due to overflow on 4-byte integer arithmetics. The bug was not easily detected, located and fixed, due to the significantly more complex algorithm and computer code. Such problems could not occur with the recursive algorithm, since it does not rely on any arithmetics.

4.2. Performance considerations

Performances were compared for the Goult–Tobochnik algorithm (with correction) and for the proposed recursive algorithm, in terms of run time for various grid sizes (up to 1000×1000 nodes) and various proportions of randomly distributed cluster-making nodes. Both algorithms were written in Fortran77 and run on a Sun-UltraSparc workstation under Solaris2.5, with the same compile-link options. Fig. 2 is a plot of the logarithm of the ratio of execution times for the two algorithms (recursive over Goult–Tobochnik) as contours lines against grid size and node proportion. The zero contour line corresponds to equal run times, whereas negative contour values indicate better performance of the recursive algorithm, and vice-versa. Positive and negative values share the plot space, with an overall lead for the recursive algorithm, especially for smaller grid sizes where the recursive algorithm performs considerably better than Goult and Tobochnik's. For larger grids, the two algorithms tend to take a comparable share, with better recursive performance for lower cluster-node proportions and vice versa. These tests were made to check that the proposed recursive algorithm compares favorably even on performance grounds, although this is not a major concern in our application where the critical step in terms of execution time is by far the hydrodynamic simulation of the generated soil models.

It also is the critical step in terms of memory requirements, but having an idea of how much space is needed by such a recursive algorithm may be of interest for its general use. Recursive calling of a function entails stack growth due to piled storage of each call's own context. Hence, stack memory requirement is roughly proportional to the recursive call piling depth, and was estimated to be around 100 bytes per call of the *label_neighborhood()* recursive function, in our Solaris2.5 Fortran implementation. The recursion depth depends on cluster size and shape, and increases with grid size and fraction of cluster-making nodes. Fig. 3 shows how this measure of algorithmic complexity varies with the cluster node fraction, for a 1000 × 1000 grid. It can be seen that the maximum depth of recursion starts to increase markedly when the cluster fraction value is near the so-called site percolation threshold (theoretical value of 0.593 above which spanning clusters are produced in infinite lattices, see for instance Schroeder, 1991), as does the maximum cluster size (MCS). However the growth is more gradual for the recursion depth: unlike the MCS, it does not show an abrupt step increase near the site percolation threshold and remains significantly smaller than the MCS for all fraction values. Fig. 4 is a plot of the stack memory used by our implementation of the

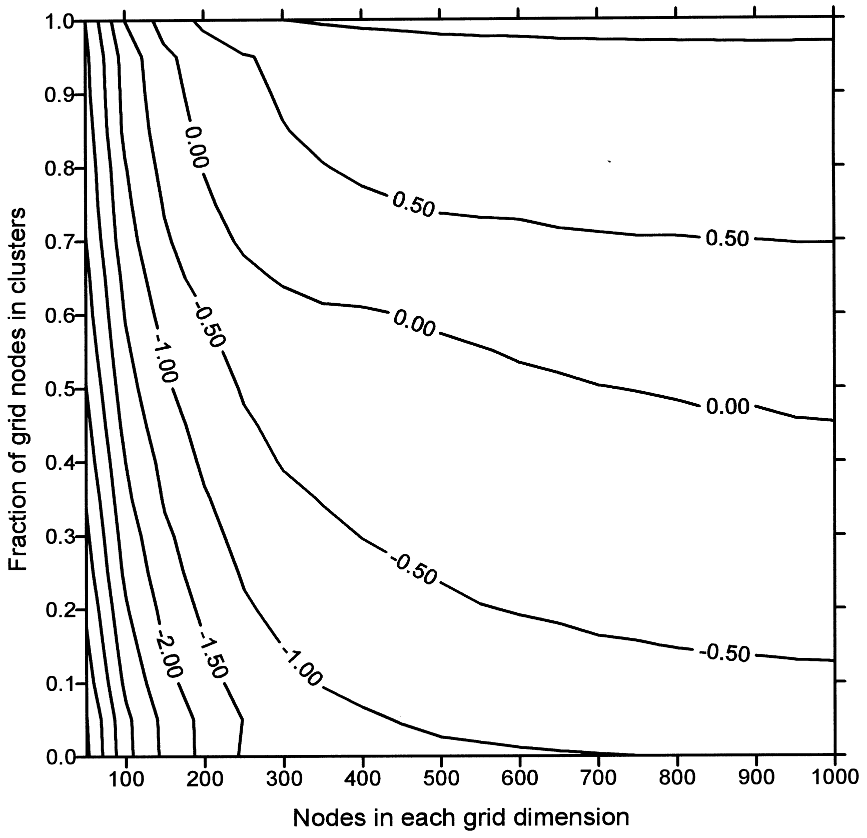


Fig. 2. Compared performances of the recursive and Goult–Tobochnik algorithms: Contour map of ln(Rt/Gt) against grid size and cluster-node fraction. (Rt and Gt: execution times for the recursive algorithm and for Goult and Tobochnik's algorithm, respectively; ln = natural logarithm).

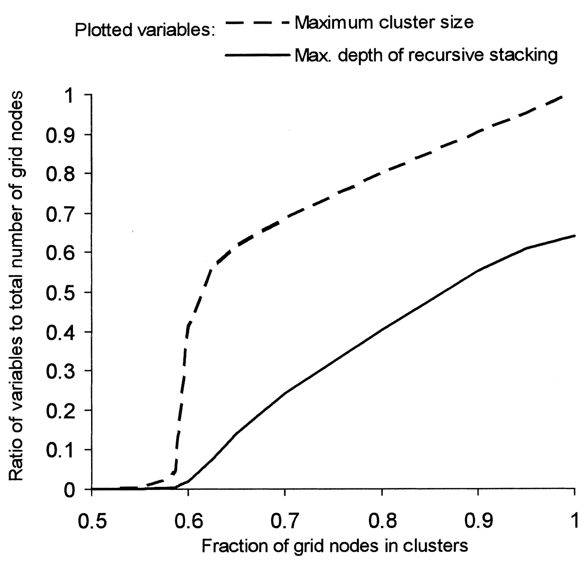


Fig. 3. Algorithmic complexity versus cluster-node fraction for the 1000 × 1000 grid size case.

recursive algorithm, as a function of grid size and cluster-node fraction. Because the number of variables in the recursive function *label_neighborhood()* is small (5 arguments, no local variable), the stack growth is slow. In fact, only the first argument *node n* is really needed as a true recursive argument; all others can be treated as global variables that do not need to be piled up during the recursive calling sequence.

In order to maximize performances (execution times and, most significantly, memory requirements) for huge problems such as 3D applications, parallelization may be achieved by performing concurrent, independent cluster labeling within subdomains. A necessarily sequential task of consolidating subdomain labeling into global domain labeling is then needed, which can use much the same recursive principles as already presented: a global label is recursively propagated through neighboring subdomains when subdomain clusters connect along the subdomain boundary. On distributed memory computers, efficiency of parallel cluster labeling may however be hindered, not by this extra com-

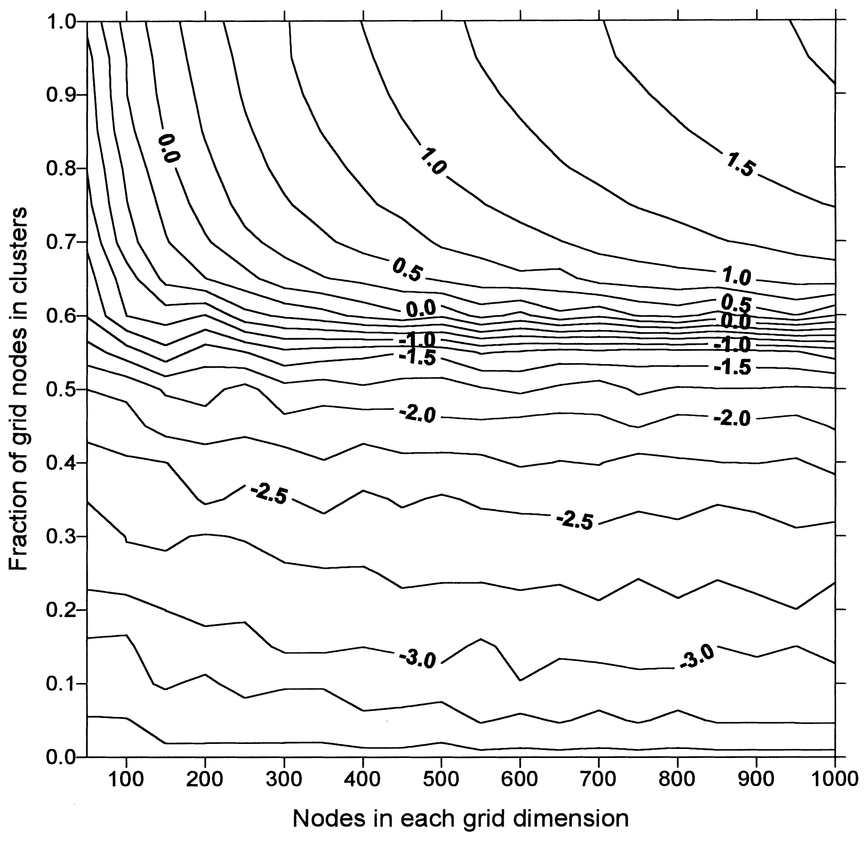


Fig. 4. Occupation of stack memory by the recursive algorithm: Contour map of Log_{10}(Stack) against grid size and cluster-node fraction. (Stack = Mbytes of memory required in stack; Log_{10}: decimal logarithm).

puting task at the larger scale (subdomain to domain), but by the overhead communication time cost of transferring the arrays of labeled subdomain nodes, which can lead to an excessive communication-to-computation ratio for most current hardware parallel architectures. However, parallel cluster labeling should be of particular interest when the subsequent, label-using treatments are themselves made parallel according to the same domain decomposition scheme, so that labeled node arrays do not need to be transferred across processors, but remain local to each subdomain processor. In this case, only cluster labels along subdomain boundaries (or lists of nodes per cluster per boundary, for each subdomain) need to be transferred to the consolidating processor, the latter sending back to each subdomain processor the global cluster label

```
module MAKE_SOIL_MODEL:
/* input is %gravel, the volumetric fraction of gravels in the soil; output is the state array with values
      either 'gravel' or 'connected' (see Fig. 6 for the internal flow of this variable);
   only the any_islands_left_?() engine is detailed as pseudo-code, using a local cluster_status array
      in addition to the nclusters and cluster_labels variables needed for the label_clusters() calls;
      the value 'TRUE' is returned as long as there remain 'isolated' states.*/

   make_soil_model(in: %gravel; out: state) {
      make_grid(state)
      assign_gravel(%gravel, state)
      do while( any_islands_left_?(state))      remove_islands(state)
   }
   logical any_islands_left_?(in/out: state) {
      label_clusters( 'unknown', state, nclusters, cluster_labels)
      for all boundaries in (upper,lower), b :   {
         for all clusters 1 to nclusters, icluster :        {
               cluster_status(icluster, b) = FALSE            /* initialization */
         }
         for all 'unknown' nodes on boundary b, n :  {
               cluster_status(cluster_labels(n), b) = TRUE
         }
      }
      any_islands_left_?=FALSE        /* initialization */
      for all 'unknown' grid nodes, n : {
         iff( cluster_status(cluster_labels(n),upper) and
            cluster_status(cluster_labels(n),lower) )  {state(n)= 'connected'}
         else      {state(n)='isolated'; any_islands_left_?=TRUE}
      }
   }
   make_grid(out: state) {
         /* construct nodes and for all nodes n initialize state(n)='unknown',
            define links, i.e. neighbors;
            please note comment in text, on one-node extension of allocated domain
               in all directions, */
   }
   assign_gravel(in: %gravel; in/out: state) {
      /* independent, random assignment of %gravel nodes n as state(n)='gravel'*/
   }
   remove_islands(in/out: state) {
   /*   move gravel nodes that are next to 'isolated' nodes, and
      reinitialize all 'isolated' and 'connected' nodes to 'unknown' */
   }
```

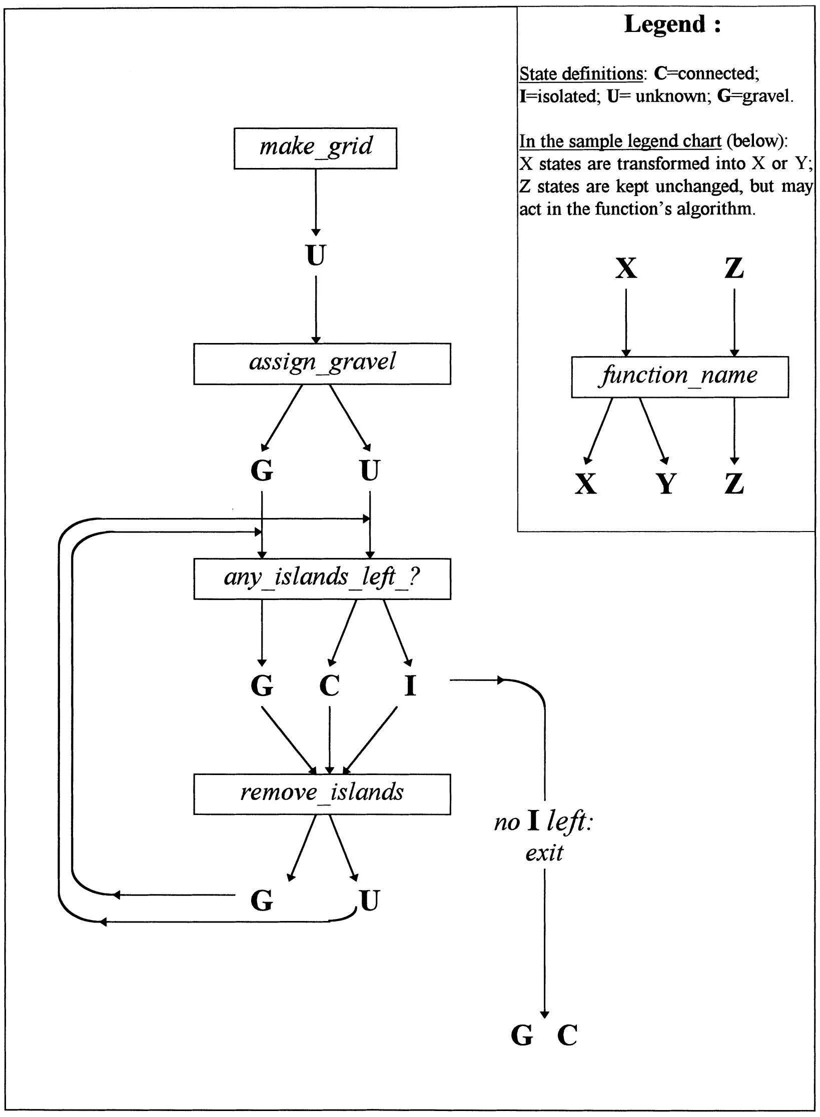
Fig. 5. Algorithm for module MAKE_SOIL_MODEL.

for each local cluster of that subdomain. A parallel implementation of the cluster labeling algorithm is proposed as pseudo-code in the appendix.

5. Building workable 2D heterogeneous soil models

Figure 5 shows the algorithm of the MAKE_SOIL_MODEL module that builds acceptable soil models for numerical simulation of infiltration in heterogeneous soils, using the generic LABEL_CLUSTERS module previously described (Fig. 1). Such soil models are produced in the form of 2D grids composed of nodes flagged either as *gravel* or *connected*, in proportion prescribed by the *%gravel* constant. The *make_grid( )* and *assign_gravel( )* functions create the initial model, while *remove_islands( )* attempts to eliminate undesirable situations, detected by *any_islands_left_?( )*, until a valid model is obtained, i.e. till *any_islands_left_?( )* returns as FALSE. A flow-chart tracing the changing values of the *state* variable through the various function calls of the MAKE_SOIL_MODEL module is presented in Fig. 6.

The model resulting from modifications of the initial gravel distribution cannot be considered to stem from the same statistical population as this initial distribution. Indeed, any reworking of the initial distri-



Legend :
State definitions: C=connected; I=isolated; U= unknown; G=gravel.
In the sample legend chart (below): X states are transformed into X or Y; Z states are kept unchanged, but may act in the function's algorithm.
X Z
function_name
X Y Z
make_grid
U
assign_gravel
G U
any_islands_left_?
G C I
remove_islands
G U
no I left: exit
G C

Fig. 6. Flow-chart of the *state* variable within the MAKE_SOIL_MODEL module.

bution will alter some of its statistical properties: the absence of initial spatial autocorrelation due to the independent node-by-node gravel assignment in *assign_gravel( )* can hardly be expected to be preserved, as proximity of gravel nodes has a bearing on island existence and therefore on ultimate gravel distribution. In line with the rather qualitative level of investigation pursued at the present stage of the study, the choice was made here to characterize the soil model used for hydrodynamic simulation by a simple construction rule, rather than trying to view it as representative of a population with predefined statistical characteristics. One such characteristic though was felt as needing to be preserved through the island elimination process, namely the total gravel proportion. Hence the *remove_islands( )* algorithm basically consists in moving around gravels that are responsible for the existence of islands, and replace with ordinary soil nodes the sites thus freed, with a displacement strategy that ensures the production of an island-free grid after a finite number of iterations (main do-while loop of the *make_soil_model( )* function) and that minimizes that number. There are many different strategies that could be envisaged, either deterministic or stochastic. Stochastic strategies have the advantage of keeping a random component throughout the soil model production process, but may turn out to be considerably less efficient with respect to the number-of-iterations criterion. A very simple and robust, deterministic strategy was preferred, which ensures convergence of the island-elimination algorithm within a small number of iterations. It consists in swapping nodes within couples made of a *gravel* node and an *isolated* node located to its immediate right. The effect is to collapse every island, by having the left and right strings of gravel that make up the island boundary come side by side; previously isolated nodes thus get released by being cast outside this boundary (see Fig. 7). This tends to produce a somewhat higher degree of aggregation of gravel nodes compared to an independent random distribution, as illustrated by a value of 1.4 for the mean number of gravel neighbors per gravel node in the

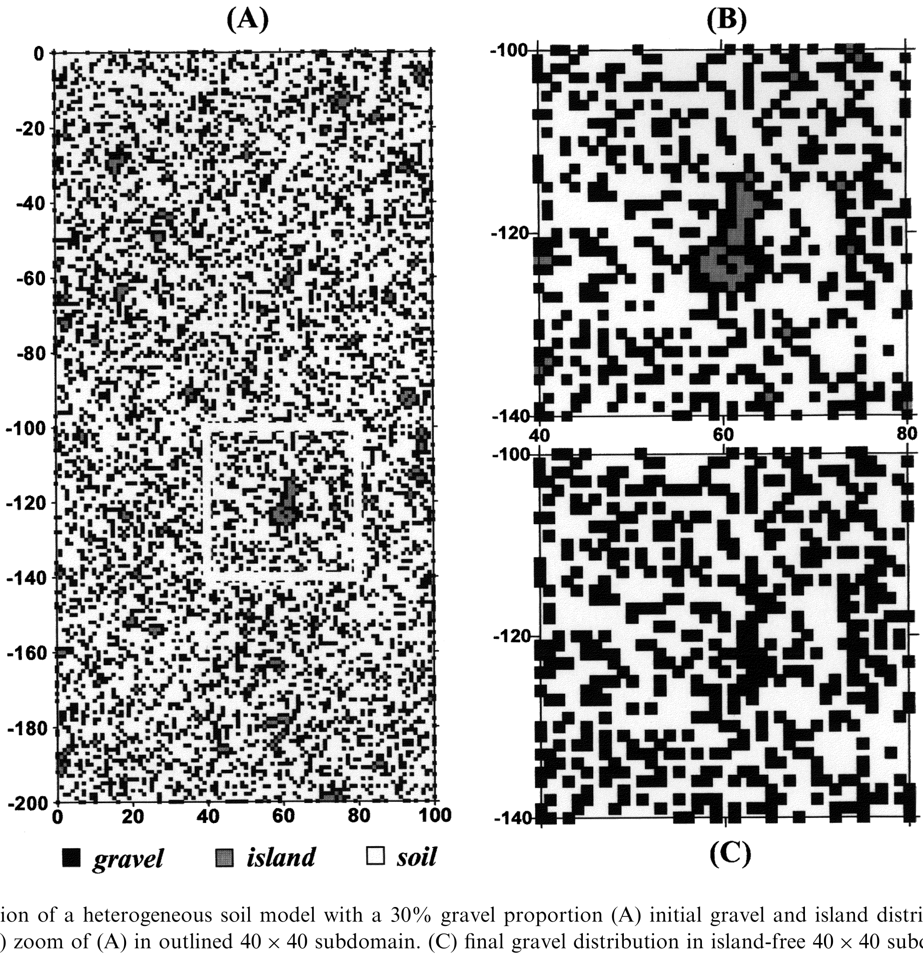


Fig. 7. Generation of a heterogeneous soil model with a 30% gravel proportion (A) initial gravel and island distribution in 100 × 200 domain. (B) zoom of (A) in outlined 40 × 40 subdomain. (C) final gravel distribution in island-free 40 × 40 subdomain.

30% gravel case, compared to 1.2 for the initial, random situation.

The *any_islands_left_?()* function uses three local variables: *nclusters* and the *cluster_labels* integer array, assigned by the call to *label_clusters()*, as well as the *cluster_status* logical array indexed both on cluster labels (1 to *n-clusters*) and on model boundaries ('upper' and 'lower' in our situation). Allocation of these arrays is not shown; it should take into account the multiple calls to the *any_islands_left_?()* function if performance is to be optimized. After array initialization to FALSE, one instance of the *cluster_status* logical is assigned to TRUE for each cluster that includes a boundary node. A final loop over all 'unknown' grid nodes allows flagging of each node as 'connected' if the cluster it belongs to connects to all boundaries (*cluster_status* logical is TRUE for each boundary, for this cluster), or else as 'isolated'. In order to simplify the definition of neighbors by using the same number of them (4 in our case) for all nodes, the allocated domain is extended one line/row outside all boundaries, and all these fictitious nodes are initialized to state = 'gravel'. Not all variables and data structures (nodes, neighbors, ...) are actually represented in Fig. 5: only those that control the program flow, particularly the *state* array, are described explicitly throughout the presentation. A discussion of the fate of this variable through the algorithm, illustrated by Fig. 6, follows.

In MAKE_SOIL_MODEL, nodes are initialized to either 'gravel' or 'unknown', denoted G and U in Fig. 6, by functions *make_grid()* and *assign_gravel()* (the latter performs the initial random drawing of node nature with a uniform gravel probability equal to *%gravel*). Every iteration of the subsequent do_while loop that calls *remove_islands()* restores a new situation with only 'gravel' and 'unknown' states. Hence, the *any_islands_left_?()* routine is always called with only 'gravel' and 'unknown' states, and returns the same 'gravel' nodes untouched, and 'unknown' states transformed to 'connected' or 'isolated' (none of the latter when the returned *any_islands_left_?()* function value is FALSE, i.e. when exiting *make_soil_model()*), denoted C and I in Fig. 6. Finally MAKE_SOIL_MODEL returns all nodes as either 'gravel' or 'connected', by moving 'gravel' nodes (done by *remove_islands()*) until no more 'isolated' nodes are returned by *any_islands_left_?()*.

6. Application

The above recursive algorithm was applied to a 2D square-meshed grid with 200 nodes vertically and 100 nodes horizontally, with a grid-cell size of 1 cm in both directions. Successive soil models were built with volumetric gravel proportions *%gravel* ranging from 5 to 40% with a step of 5%. With these values, the nongravel fraction of the lattice always stays above the percolation threshold of 0.593. Model building was replicated several times for each *%gravel* value, to account for the random nature of the gravel distribution procedure. Visual verification of the resulting models showed no failure of the algorithm in all investigated cases. The number of iterations of the do_while loop in *make_soil_model()* (i.e. number of calls to *remove_islands()*) typically ranged from 1 for the lower gravel proportion values to a maximum of 10 for the largest proportion. The number of islands detected by the first iteration varied from a couple for the 5% proportion, to as many as 590 for the 40% gravel case. The latter case corresponds to 12.9% of the grid nodes being isolated, with a maximum island size of 115 nodes i.e. 0.57% of the total grid surface. For the 30% gravel case, these figures fall to about 170 for the number of islands, 1.4% of isolated nodes and a maximum island size of only 16 nodes (see Fig. 7A; an excerpt of the corresponding final valid model, obtained after 4 iterations, is shown in Fig. 7C).

The generated soil models were used to simulate soil characterization experiments (i.e. experimental determination of $h(\theta)$ and $K(\theta)$ relationships) that can be performed in the field using a double-ring infiltrometer (Touma and Albergel, 1992). Water is supplied to maintain a small ponding depth above the soil surface, until the infiltration front has reached the 80-cm depth under the surface at least at some point of the central 30-cm wide soil column. After this infiltration phase, water redistribution is simulated without further supply until the 2-m depth is reached. Grenoble sand (Parlange et al., 1985) is used for the nongravel fraction, with a saturated hydraulic conductivity Ks = 157 mm/h and a saturated volumetric water content of 0.312.

For the 30%-gravel example case of Fig. 7, gravel incidence on the hydrodynamics of the soil model is high, as can be inferred from Fig. 8 at the end of a 2.4-h infiltration phase. The random distribution of a significant proportion of gravel in the modeled soil matrix produces contrasted macro-structures in the drainage pattern. The nonuniformity of the percolation across a horizontal plane induces an 80% reduction of overall infiltration capacity compared to pure Grenoble sand, much above the 30% that one may have expected through a simplistic rule of volumetric proportionality. Besides, with the 100-node domain width, the resulting percolation is also highly dependent on the random distribution drawn. Fig. 9 shows the distribution of water at the end of the 6-day redistribution phase, for the simulated soil configuration of Fig. 7. The study of the heterogeneous soil water movement

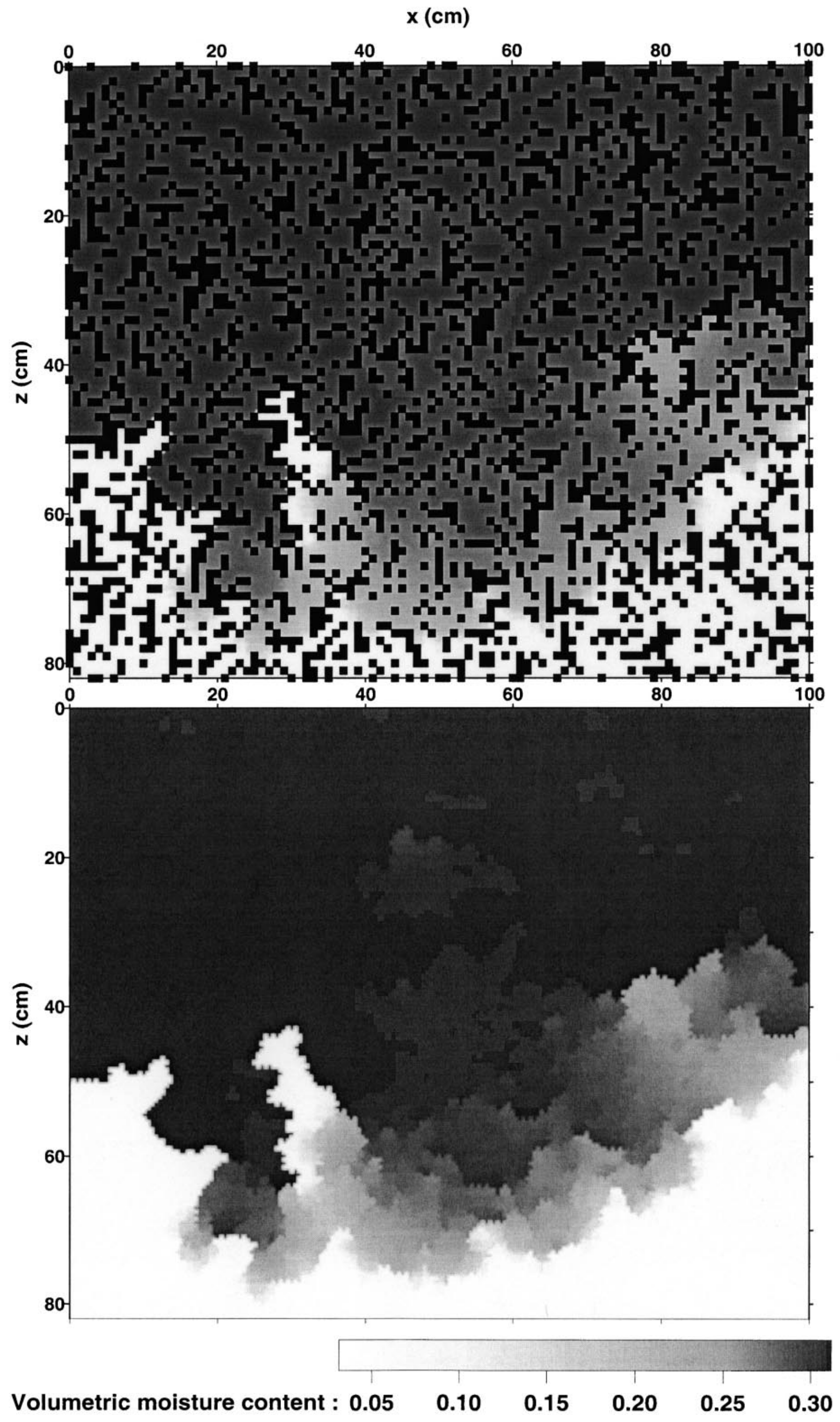


Fig. 8. Water distribution after a 2.4-h infiltration phase (with and without gravel display).

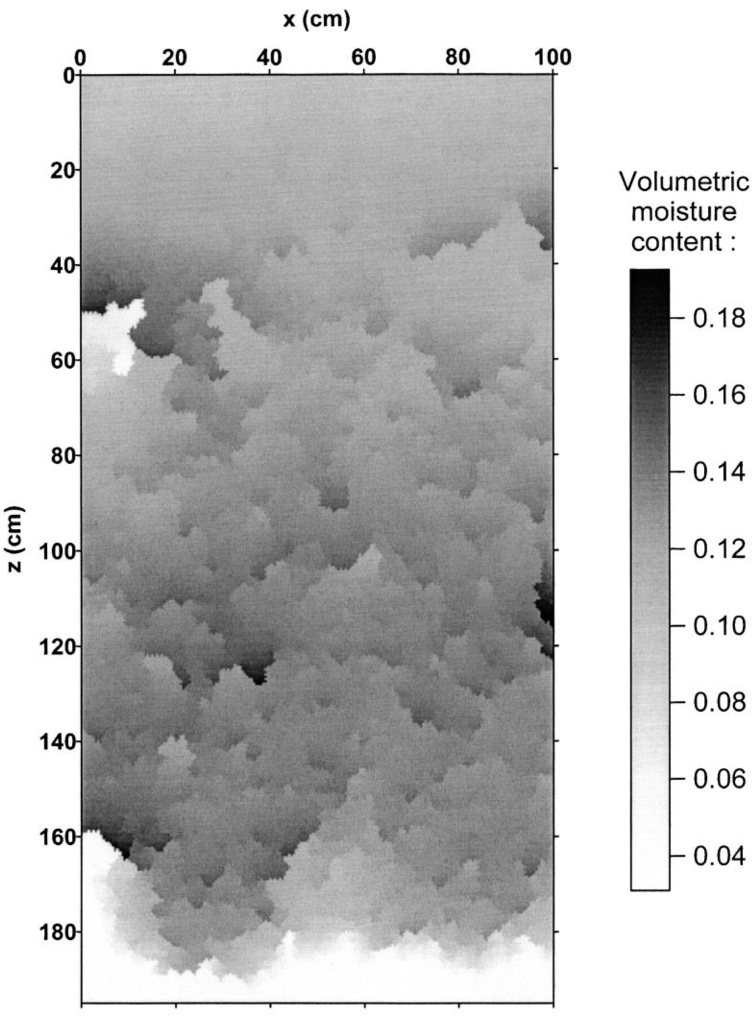


Fig. 9. Water distribution after a 6-day redistribution phase (without gravel display).

problem now requires working with larger lattices to improve the representativity of the performed simulations. In particular, the horizontal grid size should be increased to account for the variability of the generated drainage macro-structures. Also, such modeling ought to be extended from two to three dimensions to better capture the 3D nature of this physical system.

7. Conclusion

In order to build acceptable soil matrix models for numerical simulation of water infiltration and redistribution in heterogeneous soils, a generic, recursive algorithm for cluster analysis was devised and is presented as a pseudo-code. It can be used to perform the very general tasks of cluster identification and labeling, needed in many various fields and problems. Because recursion allows a very condensed formulation of an otherwise complicated problem, the proposed algorithm is easy to implement, and circumvents the risk of programming pitfalls such as arithmetic overflow that may occur for large cases with the more conventional algorithms. The provided pseudo-code may be readily implemented with any programming language, and for any grid configuration since it is entirely independant of grid topology. Performances (execution times and memory requirements) of the proposed algorithm are excellent in the range of problem sizes investigated, but may become a concern for huge problem cases, in particular for 3D applications. However, in our application, hydrodynamic modeling is by far the limiting component with respect to increasing problem size.

A parallel version of the algorithm that may facilitate the very large problem runs is suggested in the appendix of this paper. Parallel cluster labeling is highly desirable if other application components are also parallelized over the same subdomains. Recursion can also be used in the serial task of subdomain-to-domain label consolidation.

Appendix A. Algorithm for module PARALLEL_CLUSTER_LABEL

<u>module PARALLEL_CLUSTER_LABEL</u> *(simplified writing)*

```
/* produces the number of clusters global_nclusters
and the label-mapping array global_labels, over the
whole domain; local_labels and global_from_local
are temporary arrays, the scope of which is
assumed here to be the whole module ('local' and
'global' refer to subdomain and whole-domain
entities, respectively) */
    parallel_cluster_label(in: selected_state, state; out:
    global_nclusters, global_labels) {
        /* STEP 1: parallel local labeling: */
        parallel do for each subdomain, sd: {
            call label_clusters( ) within sd
            obtain: number of local cluster labels in sd
                and label array: local_labels(sd, node in
                sd)
                    /* only values for boundary nodes
                    are needed if STEP 3 is implemented
                    in the parallel case, see below */
        }
        wait for all tasks
        /* STEP 2: sequential consolidation: */
        initialize array:
            global_from_local(1..max number of local
            labels, 1..number of subdomains) = 0
        global_label = 0
        for each subdomain, sd: {
            for each local cluster label in sd,
            local_label: {
                if (global_from_local(local_label,
                sd) = = 0) {
                    global_label + = 1
                    call local_to_global(sd, local_label,
                    global_label)
                }
            }
        }
        global_nclusters = global_label
        /* STEP 3: make final global label array
        (sequential case) or send back global labels to
        subdomain processors for local construction
        of final label arrays (parallel case) */
        /* sequential case: */
        for all nodes in domain, n: {
            determine subdomain sd containing node
            n
            global_labels(n) = global_from_local
            (local_labels(sd,n), sd)
        }
        /* or, parallel case: */
        parallel do for each subdomain, sd: {
            transfer array global_from_local(1..number
            of local labels in sd, sd) to processor sd
            for all nodes in sd, n: {
                global_labels(n) = global_from_local
                (local_labels(sd,n), sd)
            }
        }
    }

recursive local_to_global (in: subdomain sd,
local_label, global_label) {
/* propagates the global_label value across sub-
domain boundaries */
    global_from_local(local_label,sd)
     = global_label
    for all boundaries b of subdomain, sd: {
        determine subdomain next to sd, across
        boundary b: sda
        for all nodes on boundary b in sd, nb: {
            if(local_labels(sd,nb) = = local_label) {
                determine node next to nb, in sub-
                domain sda: nba
                set llabel = local_labels(sda,nba)
                if((llabel ! = 0 and (global_from_lo-
                cal(llabel,sda) = = 0)) {
                    call local_to_global(sda, llabel,
                    global_label)
                }
            }
        }
    }
}
```

References

Goult, H., Tobochnik, J., 1996. An Introduction to Computer

Simulation Methods; Applications to Physical Systems, 2nd edn. Addison-Wesley, Reading, MA.

Hoshen, J., Kopelman, R., 1976. Percolation and cluster distribution. 1. Cluster multiple labeling technique and critical concentration algorithm. Physical Review B14, 3438.

Meyer, B., Baudoin, C., 1984. Méthodes de Programmation, 3rd edn. Eyrolles, Paris.

Parlange, J.Y., Haverkamp, R., Touma, J., 1985. Infiltration under ponded conditions. 1. Optimal analytical solution and comparison with experimental observations. Soil Science 139, 305–311.

Richards, L., 1931. Capillary conduction of liquids in porous medium. Physics 1, 318–333.

Schroeder, M., 1991. Fractals, Chaos, Power Laws. W.H. Freeman Co. New York.

Siklossy, L., 1976. Let's Talk LISP. Prentice-Hall, Englewood Cliffs, NJ.

Stauffer, D., Aharony, A., 1994. Introduction to Percolation Theory, 2nd edn. Taylor & Francis, London.

Touma, J., Albergel, J., 1992. Determining soil hydrological properties from rain simulator or double-ring infiltrometer experiments: a comparison. Journal of Hydrology 135, 73–86.