Math Geosci (2013) 45:131–147
DOI 10.1007/s11004-012-9437-y

Parallel Multiple-Point Statistics Algorithm Based on List and Tree Structures

Julien Straubhaar · Alexandre Walgenwitz · Philippe Renard

Received: 16 January 2012 / Accepted: 6 December 2012 / Published online: 15 January 2013

Abstract Multiple-point statistics are widely used for the simulation of categorical variables because the method allows for integrating a conceptual model via a training image and then simulating complex heterogeneous fields. The multiple-point statistics inferred from the training image can be stored in several ways. The tree structure used in classical implementations has the advantage of being efficient in terms of CPU time, but is very RAM demanding and then implies limitations on the size of the template, which serves to make a proper reproduction of complex structures difficult. Another technique consists in storing the multiple-point statistics in lists. This alternative requires much less memory and allows for a straightforward parallel algorithm. Nevertheless, the list structure does not benefit from the shortcuts given by the branches of the tree for retrieving the multiple-point statistics. Hence, a serial algorithm based on list structure is generally slower than a tree-based algorithm. In this paper, a new approach using both list and tree structures is proposed. The idea is to index the lists by trees of reduced size: the leaves of the tree correspond to distinct sublists that constitute a partition of the entire list. The size of the indexing tree can be controlled, and then the resulting algorithm keeps memory requirements low while efficiency in terms of CPU time is significantly improved. Moreover, this new method benefits from the parallelization of the list approach.

Keywords Geostatistical simulation · Multiple-point statistics · Categorical variable · List and tree structures · Parallel computing

J. Straubhaar (✉) · P. Renard
The Centre for Hydrogeology and Geothermics (CHYN), University of Neuchâtel, rue Emile-Argand 11, 2000 Neuchâtel, Switzerland
e-mail: julien.straubhaar@unine.ch

A. Walgenwitz
Ephesia Consult SA, rue Boissonnas 9, 1227 Geneva, Switzerland

1 Introduction

Spatial features of geological structures play a key role in reservoir modeling because their characteristics (such as size, shape, and connectivity) have a deep impact on flow and transport processes (Journel and Zhang 2006; Renard 2007). Hence, reproducing complex and realistic structures is a crucial issue in geostatistical simulations. Multiple-point statistics allows for stochastically generating complex heterogeneous fields by integrating a conceptual model chosen by the user. Several case studies have shown the applicability of the method (Caers et al. 2003; Liu et al. 2004; Comunian et al. 2011; Hajizadeh et al. 2011). Moreover, de Iaco and Maggio (2011) or Kessler et al. (2012) use several criteria emphasizing that multiple-point statistics gives better results than traditional simulation methods based on bi-point statistics (variograms, transiograms).

Multiple-point statistics was introduced by Guardiano and Strivastava (1993), and the first efficient algorithm, *snesim*, was developed by Strebelle (2002). The method consists in storing the multiple-point statistics inferred from the training image in a catalog. Then each node of the simulation grid is sequentially simulated according to a conditional probability distribution function, which is computed by retrieving from this catalog the entries that are compatible with the neighborhood of the current node. In other algorithms, such as *simpat* (Arpat and Caers 2007) and *filtersim* (Zhang et al. 2006; Wu et al. 2008), the simulation proceeds by sequentially patching patterns also provided by a training image. These latter methods allow continuous variables to be considered, as well as the direct sampling algorithm proposed by Mariethoz et al. (2010), where neither a catalog nor a database is required because the training image is directly sampled for the simulation of each node.

The original multiple-point statistics algorithm *snesim* (Strebelle 2002) uses a large amount of memory to store the statistics in tree structures. This implies some limitations on the size of the template and then complex structures cannot be properly reproduced. Zhang et al. (2012) proposed to use compact search trees for overcoming the memory limitations. Another way to reduce the memory requirements consists in storing multiple-point statistics in a list instead of a tree, as in the parallel algorithm *impala* (Straubhaar et al. 2011). Many extensions, applicable to these classical multiple-point statistics algorithms, have been developed, such as post-processing techniques (Strebelle and Remy 2005; Stien et al. 2007; Suzuki and Strebelle 2007; Mariethoz et al. 2010), simulations involving nonstationary training images (Chugunova and Hu 2008; Straubhaar et al. 2011), soft probabilities (Allard et al. 2012), and connectivity conditioning (Renard et al. 2011).

In this paper, a new approach for classical multiple-point statistics that consists in mixing the tree and the list structures to store multiple-point statistics is proposed. The new approach benefits from the advantages of both storage techniques: The list allows for reducing memory requirements and parallelization, whereas the tree allows for accelerating the computation of the conditional probability distribution function. The method consists in indexing the entries of the list containing all the patterns found in the training image by a tree of reduced size. More precisely, the list is sorted in lexicographical order according to the data event (pattern). The tree has similar branches as in the usual search tree, but the cells contain only the pointers to the beginning and

the end of the corresponding part of the list. The depth of such a tree can be cut everywhere without losing any information and its size is controlled by two parameters. The resulting algorithm benefits directly from the parallelization of the list approach and constitutes a new version of *impala* (Straubhaar et al. 2011). It is presented in detail in this paper and numerical tests show that a significant gain of CPU time is obtained, while the memory requirements are still low. Finally, the performance of the parallel version is evaluated by performing numerical experiments.

2 Principles of Multiple-Point Statistics Algorithms

In this section, we recall the basic concepts of multiple-point statistics and introduce some notations. The aim is to populate a simulation grid (*SG*) with a categorical variable s (facies) in a stochastic framework and in such a way that the structures within a given training image (*TI*) are reproduced. For this, multiple-point statistics of order $N+1$ are used as follows. Let τ be a search template defined as a set of N lag vectors $h_1, \ldots, h_N$, and for a node u, let $d(u) = \{s(u+h_1), \ldots, s(u+h_N)\}$ be the data event d at u. Assume that the categorical variable takes the values $s = 0, \ldots, M-1$, and that the value $s = -1$ is assigned to the unsimulated nodes in *SG*. Then the simulation consists in: defining a random path visiting all the nodes in *SG*; and for each successive node u of this path, randomly drawing a facies $s(u)$ according to the conditional probability distribution function (CPDF)

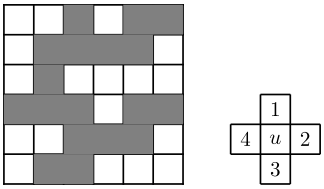
$$\mathbb{P}\big(s(u) = k \mid d(u)\big) = \frac{\#\{v \in TI : s(v) = k \text{ and } s(v+h) = s(u+h)\ \forall h \in \tau \text{ s.t. } s(u+h) \neq -1\}}{\#\{v \in TI : s(v+h) = s(u+h)\ \forall h \in \tau \text{ s.t. } s(u+h) \neq -1\}}. \tag{1}$$

Note that if the denominator is equal to zero, the farthest informed node in the search template centered at u is dropped, that is the last component in $d(u)$ not equal to -1 is set to -1, until the denominator does not vanish.

3 Using Lists for Multiple-Point Statistics

3.1 Storing the Statistics in Lists

Storing the multiple-point statistics inferred from the *TI* allows for avoiding having to scan the entire *TI* to compute the CPDF (1) at each node of the *SG*. The storage technique proposed by Straubhaar et al. (2011) consists in recording every distinct data event (pattern) found in the *TI* in a list. Using the notations above, an element of the list is a pair of vector $(d = (s_1, \ldots, s_N), c = (c_0, \ldots, c_{M-1}))$, where c_i is the number of occurrences of the data event $\{s(v+h_1) = s_1, \ldots, s(v+h_N) = s_N\}$ found in the *TI* having the facies $s(v) = i$ at the node v. This list is built by scanning the *TI* once at the beginning of the simulation. This concept is illustrated in Fig. 1.



List elements
$L_0 = ((0,0,1,1),(0,1))$
$L_1 = ((0,1,0,1),(0,2))$
$L_2 = ((0,1,1,0),(0,2))$
$L_3 = ((0,1,1,1),(1,0))$
$L_4 = ((1,0,0,0),(1,0))$
$L_5 = ((1,0,0,1),(0,2))$
$L_6 = ((1,0,1,0),(1,1))$
$L_7 = ((1,0,1,1),(1,0))$
$L_8 = ((1,1,0,1),(0,2))$
$L_9 = ((1,1,1,0),(1,1))$

Fig. 1 Training image (facies code 0 for *white nodes* and facies code 1 for *gray nodes*), search template, and corresponding *list* obtained by scanning the *TI* in such a way that the search template always remains entirely inside the *TI*

Note that in the example of Fig. 1, only the nodes v in the *TI* for which the search template centered at v is entirely inside the *TI* are scanned. It is also possible, however, to scan all the nodes of the *TI*: in that case, the central node of the template visits all the nodes of the *TI*, including those along the boundaries. In that situation, a new value for lacking facies is used to code the data event nodes falling outside the *TI*. This approach is useful because the region in the boundary of the *TI* corresponding to such data events can be important when using a large search template on a coarse multigrid level. Adding a code for lacking facies is also very useful when training images are incomplete and reconstruction must be performed (Mariethoz et al. 2012). Moreover, storing the statistics in a list allows for nonstationary *TI* (Straubhaar et al. 2011) to be efficiently dealt with. In such a case, an auxiliary variable is used for describing the nonstationarity and this additional information is stored in the list.

3.2 Retrieving Conditional Statistics from the List

To compute the CPDF (1) the list is systematically scanned to retrieve all the elements that are compatible with the conditioning data event. This is described in detail in Straubhaar et al. (2011) and summarized here. An element (d, c) of the list is considered as compatible with the conditioning data event $d(u)$ if every informed component of $d(u)$ is equal to the corresponding component in d, that is $d_i = d_i(u) = s(u + h_i)$, for all i such that $s(u + h_i) \neq -1$. The numerator of the CPDF (1) is given by the sum of the k-th counter c_k of every element compatible with $d(u)$, and the denominator by the sum of all the counters $c_0, \ldots, c_{M-1}$ of these elements.

4 Indexing the List by a Tree

To avoid having to scan all the elements of the list to retrieve those that are compatible with a given data event, we propose to index the list by a tree. The branches of that tree are defined as introduced by Strebelle (2002) for the search tree in classical multiple-point statistics algorithm. But, in the proposed approach, the cells contain only pointers to the elements of the list. In the following, we first provide a general description of the proposed structure for the indexing tree, and then show how to build and use it, and giving a detailed example.

4.1 Structure of the Indexing Tree

The list is sorted lexicographically according to the data events as in a previous approach. In a situation with M facies, the indexing tree is an M-ary tree. Each cell of the tree is divided into M subcells which can have or not a child cell. The levels in the tree correspond to successive nodes in the template and are numbered from 1 (root) to a certain depth. The subcells in a cell are numbered from 0 to $M-1$. A subcell position in the tree is defined by a path $\{i(1), i(2), \ldots, i(k)\}$, where $i(j)$ is the identification number of a subcell in a cell of level j. At the location given by the above path, the subcell contains the index of the first element in the list for which the data event d begins by $i(1), i(2), \ldots, i(k)$, and one plus the index of the last of those elements. Each subcell in the tree corresponds to a contiguous part of the list, called a sublist. The following properties characterize the tree:

(P1) The subcells in the root cell provide a partition into M sublists of the whole list.
(P2) The subcells in a child cell constitute a partition into M sublists of the sublist attached to the parent subcell.

Because all the multiple-point statistics are stored in the list, the indexing tree can be cut anywhere. This tree allows for locating the elements of the list that can be compatible with a given data event and then reducing the number of elements that must be scanned to compute the CPDF (1).

It is important to emphasize that the statistics required to compute the CPDF (1) are retrieved from the list, and hence, reducing the size of the indexing tree does not change the computation of the CPDF (1). On the contrary, reducing the size of the classical search tree as in *snesim* requires diminishing the size of the template, which means that some information is lost and that the results could be deteriorated. The indexing tree is used to target the parts of the list to be scanned, hence one has to build a tree allowing for an efficient search in the list, while keeping the memory burden under control. As indicated earlier, note that in the case where all the nodes of the *TI* are scanned for building the list, a new facies for lacking value is used, and appears in some data events stored in the list. In this situation, each cell of the tree is divided into $(M+1)$ subcells and it becomes an $(M+1)$-ary tree.

4.2 Building an Indexing Tree of Reduced Size

As a consequence of the properties (P1) and (P2), the set of all the sublists attached to a subcell, which has no child cell (such a subcell is called a leaf of the tree) forms a partition of the whole list. The idea for obtaining an indexing tree allowing an efficient computation of the CPDF (1) is to impose that the size of the sublists in the leaves of the tree are smaller than a given size. Moreover, for avoiding too many recursions for this computation when the data event $d(u)$ is partly informed, a maximal depth can also be imposed. Then, defining the depth of the tree as the number of its levels minus one, we consider the parameters

$$d_{\max}: \quad \text{maximal depth of the tree,} \tag{2}$$

$$s_{\max}: \quad \text{maximal size for the sublists in the leaves of the tree.} \tag{3}$$

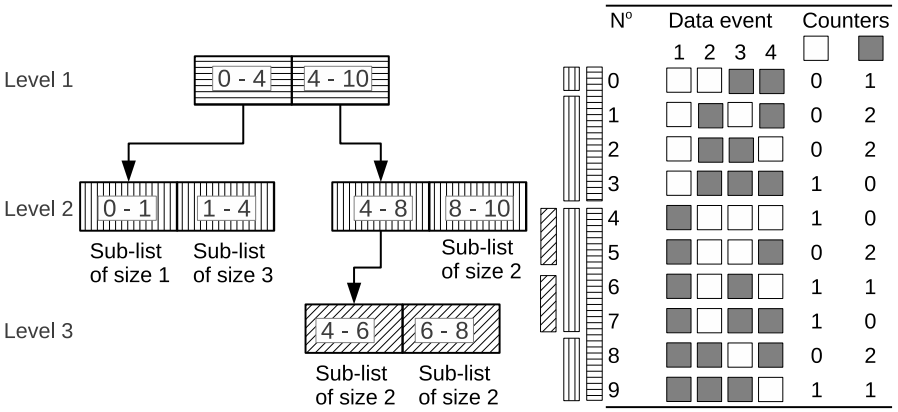


Fig. 2 Illustration of the indexing tree for the example displayed in Fig. 1. See the text for a detailed explanation

The construction of the indexing tree then starts form the root cell (level 1), and is pursued by creating subcells and applying the following rule

$$\begin{aligned} &\textit{A child cell of a subcell is created if and only if} \\ &\quad - \textit{ the subcell has a level } l < 1 + d_{\max}, \textit{ and} \\ &\quad - \textit{ the size of its corresponding sublist is greater than } s_{\max}. \end{aligned} \tag{4}$$

The parameters (2) and (3) allow for controlling the size of the tree and ensuring that the memory usage remains efficient. If a search template of size N is used, setting $d_{\max}$ to $N-1$ or more implies no constraint on the depth of the tree. Note that the maximal indexing tree obtained with the parameter $s_{\max} = 1$ (and $d_{\max} = N-1$) is not necessarily of depth $N-1$. The sensitivity to both parameters in terms of memory requirements and CPU performances is studied in Sect. 5.

4.3 Illustrative Example

The list of Fig. 1 is used to illustrate how to build an indexing tree step by step. The parameter $s_{\max}$ is set to 3 and no constraint is considered on the depth of the tree ($d_{\max}$ is set to infinity) for this example. The resulting indexing tree is shown on the left side of Fig. 2, beside the list on the right side of Fig. 2. The hatched rectangles drawn in Fig. 2 beside the numbers of the elements of the list highlight the sublists indexed by the tree. The horizontal hatchings correspond to the subcells of the root cell (level 1) of the tree, the vertical hatchings to the subcells in level 2 of the tree, and the slanting hatchings to the subcells in level 3.

The first index in a subcell is the number indicating the position of the first element of the corresponding sublist, and the second index is one plus the number of the last element of this sublist; then the size of the sublist is simply obtained by subtracting the first index from the second one. The size of the sublists are explicitly written in Fig. 2 below the subcells that are leaves of the tree.

This indexing tree is built step by step as follows. First, the root cell (level 1) is built: it is made up of two subcells corresponding to two sublists of 4 and 6 elements. They both have more than 3 ($= s_{\max}$) elements. Then a child cell (in level 2) is appended to each subcell of the root cell. Among the four subcells in level 2, only the third one corresponds to a sublist containing more than 3 elements. Then, for this subcell only, a child cell is created (in level 3). The sublists attached to this cell contain 2 elements, hence, the indexing tree is complete.

4.4 Using an Indexing Tree for Retrieving Conditional Statistics

Assume a tree of depth D indexing the list of multiple-point statistics, and assume that the data event $d(u)$ at the node u to be simulated has n informed nodes (components not equal to -1) at positions $i_1 < \cdots < i_n$. In this situation, the tree is recursively explored in the following way for getting the parts of the list that must be scanned to retrieve the elements compatible with $d(u)$:

(1) Compute the maximal explored level $L = \max\{i_j :\ i_j \leqslant 1 + D\}$, that is L is set to the last position in $i_1 < \cdots < i_n$ that does not exceed the number of levels in the tree. In other words, L is the last level in the tree corresponding to a position of an informed node in $d(u)$.
(2) Start to explore the root cell (level $l = 1$), and do the next step recursively.
(3) For the current cell, according to its level l in the tree, do:
  (3a) if the lth component in $d(u)$ corresponds to an informed node ($l = i_j$ for a certain j), then: if $l = L$ or the subcell of index $d_l(u)$ in the current cell has no child cell, then get the list indices stored in this subcell, else explore the child cell of this subcell and go to (3) (recursion step),
  (3b) otherwise, for each subcell of the current cell: if it has no child cell, then get the list indices stored in it, else explore its child cell and go to (3) (recursion step).

The list indices obtained by this exploration give all the parts of the list that have to be scanned for retrieving all the elements of the list that are compatible with the data event $d(u)$. Then the CPDF (1) is computed as explained in Sect. 3.2 and the node u is simulated. Note that the condition given by the maximal level L avoids some useless recursions.

Example Assume that a node u is simulated according to the multiple-point statistics given by the list in Fig. 1. Also, assume that the data event centered at u is $d(u) = (-1, 0, 1, -1)$, that is the nodes $i_1 = 2$ and $i_2 = 3$ are informed with facies 0 (white) and 1 (gray), respectively. By using the indexing tree of depth $D = 2$ in Fig. 2 and following the method above, we have $L = 3$, and the sublists that have to be scanned are those corresponding to the first subcell in the level 2 of the tree, and to the second subcell in the level 3 of the tree; that is the elements $\{L_0\}$ and $\{L_6, L_7\}$ of the whole list are checked. All these elements are compatible with $d(u)$ and the sum of the corresponding counters gives twice the facies 0 and twice the facies 1 at the central node. Hence, the CPDF (1) is a probability of 0.5 for each facies. As another example, assume that $d(u) = (1, -1, -1, 1)$, that is 2 nodes are informed at positions

$i_1 = 1$ and $i_2 = 4$, and contain the facies 1 (gray). Then $L = 1$, and only the sublist $\{L_4, \ldots, L_9\}$ given by the right subcell of the root cell is scanned. In this sublist, the elements L_5, L_7, and L_8 are compatible with $d(u)$ and the sum of the counters gives once the facies 0 (white) and 4 times the facies 1 (gray) at the central node, and then the CPDF is a probability of 0.2 and 0.8 for the facies 0 and 1, respectively.

5 Sensitivity Analysis to the Parameters for Building the Indexing Tree

How do we fix the parameters $d_{\max}$ (2) and $s_{\max}$ (3) involved in the construction of the tree? In this section, numerical tests are performed to test the sensitivity of the (serial) algorithm to each of these parameters. Based on this sensitivity analysis, parameters will be proposed to make the algorithm as efficient as possible in terms of computation time and memory load. Three examples are considered using input parameters corresponding to realistic conditions of simulation:

(I) two-dimensional *TI* of size 250×250 with 2 facies, *SG* of size 300×300, 3 multigrids, and disc-shape search templates containing 100, 60 and 20 nodes from the coarse multigrid to the fine one;
(II) two-dimensional nonstationary *TI* of size 820×208 with 8 facies and 1 auxiliary variable, *SG* of size 820×208, 3 multigrids, and disc-shape search templates containing 128, 68, and 36 nodes from the coarse multigrid to the fine one;
(III) three-dimensional *TI* of size $100 \times 100 \times 60$ with 4 facies, *SG* of size $50 \times 50 \times 40$, 3 multigrids, and spherical search templates containing 618, 250, and 56 nodes from the coarse multigrid to the fine one.

The training images used in these examples are shown in Fig. 3. Note that for each test, the multigrid approach is used for insuring a proper reproduction of the structures, while keeping the search template size reasonable. Moreover, the size of the specified search templates decreases from coarse to fine multigrid level. Indeed, the nodes simulated at the fine multigrid level represent the major part of the simulation grid (about 75 % and 87.5 % in the two- and three-dimensional cases, respectively), then specifying search template of decreasing size saves a considerable amount of time, whereas the quality of the results are affected only minimally.

The procedure for every test is the following. The *TI* is entirely scanned, that is, an additional facies for lacking value is used and the indexing trees are $(M + 1)$-ary trees where M is the number of facies. First, the list-based algorithm is used to generate a set of realizations (10 for examples (I) and (II), and 5 for example (III)). In this situation, no tree is built and the lists are entirely scanned to compute the CPDF (1) required for the simulation of each node. Then the same set of 10 realizations (provided by the same random seed) is generated using the new approach with different values of the parameters $d_{\max}$ and $s_{\max}$ controlling the indexing trees. For each choice of these parameters, the real time spent and the memory requirements are compared to the case where the list-based algorithm (reference) is employed.

The tested values for the two parameters $d_{\max}$ (2) and $s_{\max}$ (3) are set as a fraction of the size (number of nodes) of the search template N and the size (number of

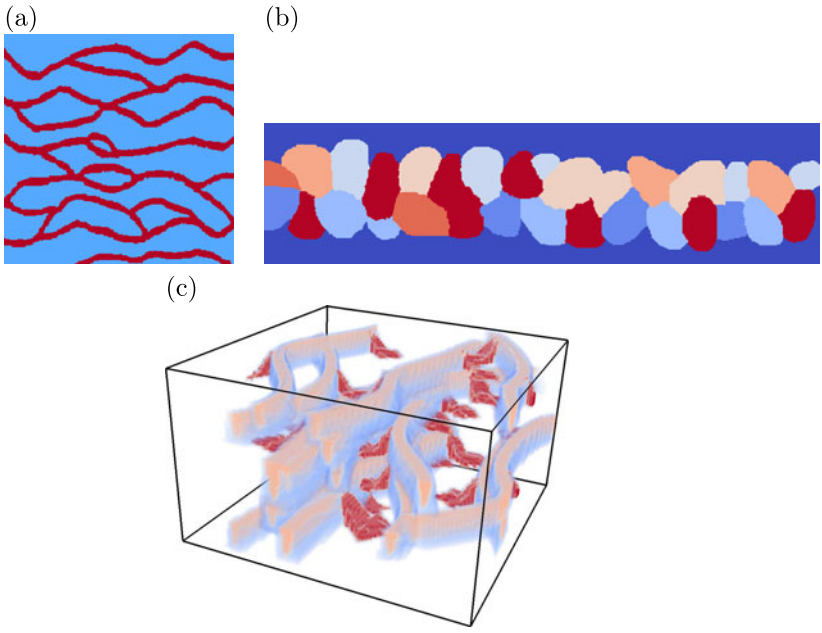


Fig. 3 Training images for examples (I–III): (**a**) *TI* for example (I), 250 × 250, 2 facies (from Strebelle 2002); (**b**) *TI* for example (II), 820 × 208, 8 facies (courtesy of Backer Hughes), the relative vertical location is used as an auxiliary variable; (**c**) *TI* for example (III), 100 × 100 × 60, 4 facies (courtesy of total)

elements) of the list $N_{\mathcal{L}}$

$$d_{\max} = p_d \cdot N, \tag{5}$$

$$s_{\max} = p_s \cdot N_{\mathcal{L}}. \tag{6}$$

Indeed, since each level of multigrid has its own search template and its own list, the values of $d_{\max}$ and $s_{\max}$ depend on the level of multigrid. The results are shown in Fig. 4 for the three examples: The left maps represent the additional memory load and the right maps the fraction of real time spent for the new approach compared to the list-based algorithm. The absolute values for the reference are given in Table 1. All tests presented in this paper are performed using processors Intel(R) Xeon(R) CPU E5620, 2.40 GHz on 64-bits systems.

These tests show that in comparison with the list-based algorithm, the real time spent can be divided by more than 3 for a cost of an additional memory load less than 10 %. To place this result in perspective, the algorithm based on classical search trees was compared to the list-based algorithm in Straubhaar et al. (2011). The time required by the classical search tree was approximately half of the time required by the list (serial version), whereas the memory burden was quickly multiplied by 25. Note that for the three examples, the high values are not represented in the figures. The maximal value for additional memory load is respectively around 260 %, 315 %, 10 % (in the top left corner in Figs. 4(a), 4(c), and 4(e)) for examples (I), (II), (III),

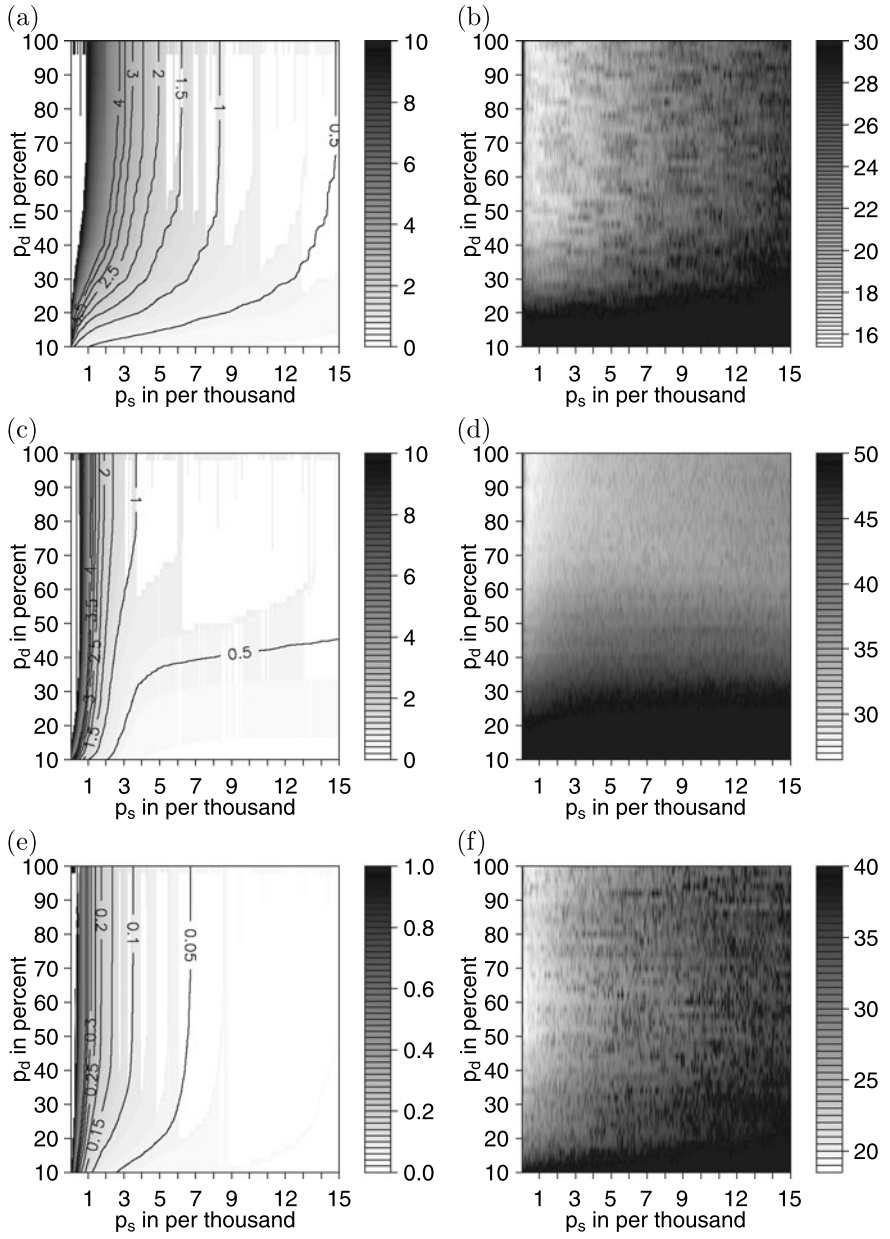


Fig. 4 Sensitivity to parameters for the indexing trees for example (I–III): (*left*) additional memory load used for storing the indexing trees in percent of the amount of memory required by the lists, as a function of p_s and p_d, (**a**) example (I), (**c**) example (II), (**e**) example (III); (*right*) real time in percent of the reference time (list-based algorithm) as a function of p_s and p_d, (**b**) example (I), (**d**) example (II), (**f**) example (III). A ceiling value of (**a**) 10 %, (**b**) 30 %, (**c**) 10 %, (**d**) 50 %, (**e**) 1 %, (**f**) 40 % is applied for the results shown in this figure

Table 1 Amount of memory required for storing the lists in MB and real time spent in seconds for the list-based algorithm (reference) of the three examples

Example	Mem (MB)	Time (sec)
(I)	7.65	123
(II)	30.75	934
(III)	461.80	2,320

whereas the values for real time fraction approximately reach 45 %, 60 %, and 60 % (in the bottom in Figs. 4(b), 4(d), and 4(f)). Hence, the parameters corresponding to these area must be avoided.

To accelerate the algorithm, the parameter p_d should be chosen rather large and p_s small. We propose to fix the parameter p_d to 90 %. The sensitivity analysis to the parameter p_s is given in more detail in Fig. 5: the plots in the first column represent the additional memory load as a function of p_s, the plots in the second column the real time spent as a function of p_s, and the plots in the third column the real time spent as a function of the additional memory load. Obviously, the memory load is decreasing when p_s is increasing (Figs. 5(a), 5(d), and 5(g)). The behavior of the real time spent is more complicated: if we reduce the value of p_s (reading each plot in the second column of Fig. 5 from right to left), the real time spent decreases because the scanned parts of the list are better targeted, and then suddenly increases, which is the result of much more recursions done in the indexing tree during the simulation of each node, because the number of branches in the trees explodes. The plots in the third column of Fig. 5 summarize the two first ones and give the trend of the real time spent as function of the additional memory load.

Finally, for the three examples that were studied, very good performances were obtained by limiting the depth of the indexing tree to 90 % of the search template size and by limiting the sizes of the sublists corresponding to a leaf of the tree to around 1 % of the total length of the list. We expect these results to be rather robust, but we also expect that the optimal parameters will slightly vary depending on the complexity of the *TI*, the number of facies, and other inputs for the multiple-point statistics algorithm.

6 Parallel Algorithm

6.1 Strategy

This new approach straightforwardly benefits from the parallelization developed earlier for the lists (Straubhaar et al. 2011). The strategy of parallelization allows for an implementation both on machines with distributed memory using MPI (Message Passing Interface) technology as well as on machines with shared memory using OpenMP. The strategy is to subdivide the list into partial lists and to distribute these partial lists over the available processors; then the presented method consisting in building an indexing tree for a list can be separately applied on each processor. More precisely, assume that p processors, numbered from 0 to $p-1$, are used. Then a list $\mathcal{L} = \{L(0), \ldots, L(N_{\mathcal{L}} - 1)\}$ sorted in lexicographical order is distributed

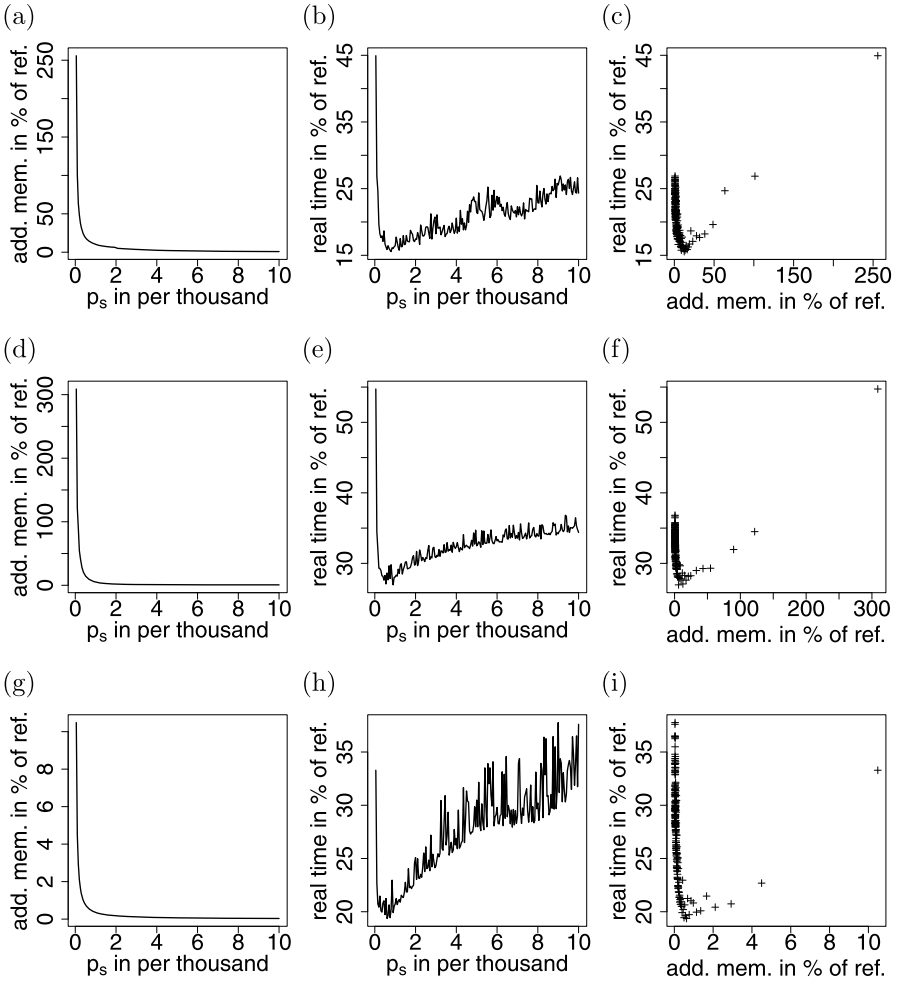


Fig. 5 Sensitivity to parameter p_s, with $p_d = 90$ % (fixed), for the indexing trees: (*left*) additional memory load used for storing the indexing trees in percent of the amount of memory required by the lists, as a function of p_s, (**a**) example (I), (**d**) example (II), (**g**) example (III); (*middle*) real time in percent of the reference time (list-based algorithm) as a function of p_s, (**b**) example (I), (**e**) example (II), (**h**) example (III); (*right*) real time as a function of additional memory load, both in percent of the reference, (**c**) example (I), (**f**) example (II), (**i**) example (III)

over all the processors in the following way. The partial list $\mathcal{L}_j = \{L(k) : k \equiv j \mod p \text{ and } 0 \leqslant k < N_{\mathcal{L}}\}$ is stored into the memory space dedicated to the processor of index j. Hence, each processor has its own local list, and then builds its own indexing tree, as explained in Sect. 4.2. Then the simulation of each successive node in the *SG* is parallelized as follows. Each processor retrieves from its local list the occurrence counters provided by the elements compatible with the data event centered at the current node in the *SG*. This is done by using the new approach explained in Sect. 4.4. Then the counters retrieved by every processor are gathered by a commu-

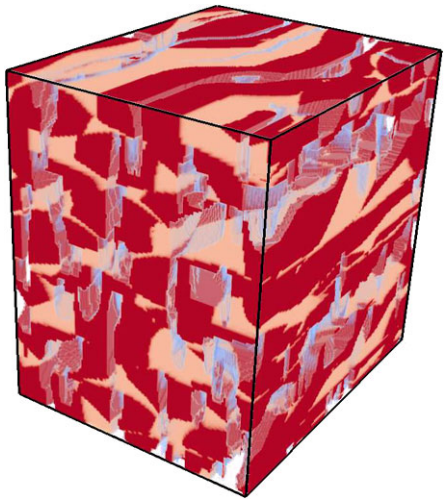


Fig. 6 Training image for example (IV), $200 \times 160 \times 200$, 4 facies (courtesy of Mines Paris Tech)

nication between them, and used together to compute the CPDF (1), and finally the current node is simulated.

6.2 Numerical Tests

According to the tests of Sect. 5, we choose to set the parameters $p_d = 90\ \%$ and $p_s = 1.2\ ‰$ for the indexing trees. The performances of the parallel algorithm is experimentally evaluated on four cases: the examples (I) to (III) whose input parameters are given in Sect. 5, and the following more substantial example:

(IV) three-dimensional *TI* of size $200 \times 160 \times 200$ with 4 facies (Fig. 6), *SG* of size $120 \times 96 \times 120$, 3 multigrids, and spherical search templates containing 618, 250 and 56 nodes from the coarse multigrid to the fine one.

The number of realizations is set to 10 for two-dimensional examples (I and II), and set to 5 for three-dimensional examples (III and IV). For each example, the new parallel algorithm (based on MPI) is launched using different numbers of processor(s) between $p = 1$ and $p = 48$. Then the CPU time (maximal CPU time over all the processors used) of each run is retrieved. Let T_p be the CPU time when p processors are used. The first way to evaluate the performance is to see how T_p decreases when p increases. The plot of T_p as a function of p is shown in Fig. 7 for each example. In this figure, the CPU time required by the list-based algorithm (not using trees) is also drawn for comparison. In particular, the time required by the two approaches becomes close from a certain number of processors, showing that the indexing trees can be useless when the corresponding lists are sufficiently small, provided by the subdivision of the entire list into partial lists.

The evaluation is also given by the speed-up and the efficiency, respectively, defined as

$$S_p = \frac{T_1}{T_p} \quad \text{and} \quad E_p = \frac{S_p}{p}. \tag{7}$$

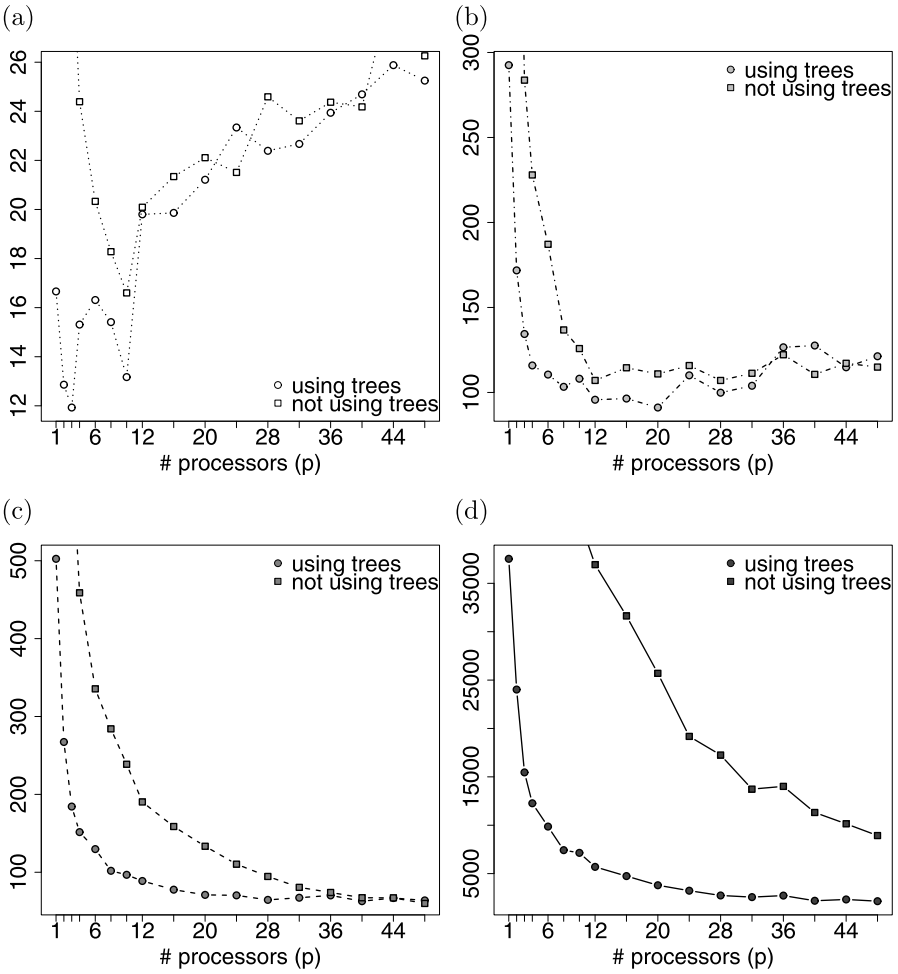


Fig. 7 CPU time in seconds as a function of the number of processors used, for the parallel algorithm based on lists and indexing trees (new approach) and the algorithm based on lists only: (**a**) for example (I), (**b**) for example (II), (**c**) for example (III), (**d**) for example (IV)

These measurements can be compared to the ideal case $S_p = p$ and $E_p = 1$, which corresponds to the situation where the time is divided by p when the number of processors is multiplied by p. Note that the efficiency is a normalization of the speed-up. The speed-up and efficiency curves for examples (I) to (IV) are shown in Fig. 8.

The curves of CPU times drawn on Fig. 7 are strongly decreasing from $p = 1$ processor to a certain number of processors, showing that the parallelization of the proposed algorithm is worthy in each of the four examples. The performances of the parallelization depend on the size of the problem to be solved. Large problems benefit from a parallel run using a large number of processors, whereas time required for completing small problems is penalized by the communications between processors, which are not negligible compared to the actual computations. This classical observa-

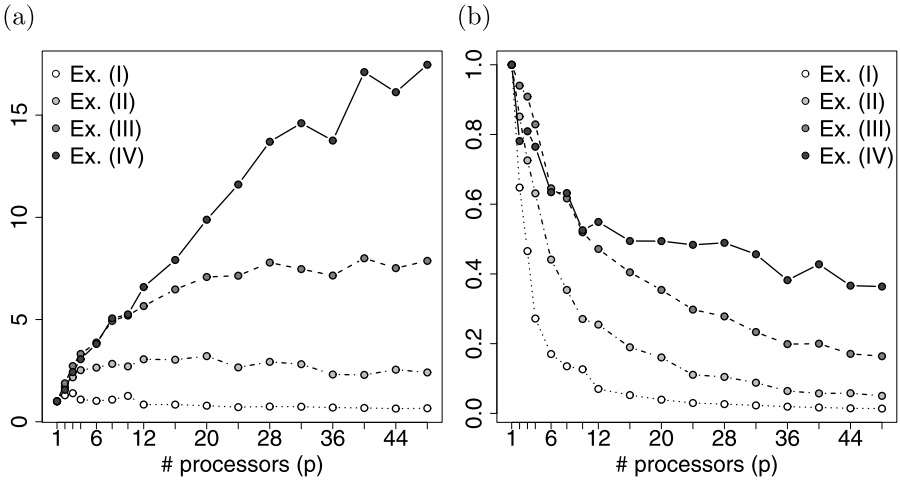


Fig. 8 Speed-up and efficiency for the new parallel algorithm based on lists and indexing trees: (**a**) speed-up, (**b**) efficiency

Table 2 Approximate number of elements in each entire list, for the examples used for the evaluation of parallelization

Multi-grid level	Ex. (I)	Ex. (II)	Ex. (III)	Ex. (IV)
Coarse	54,000	93,000	580,000	6,397,000
Middle	23,000	33,000	353,000	5,817,000
Fine	1,000	12,000	91,000	1,343,000

tion for parallel implementations is emphasized in Figs. 7 and 8 for the new parallel algorithm proposed in this paper, since examples (I) to (IV) involve increasing computational burden due to the increasing size of the lists used, as shown in Table 2. Note that Fig. 7(a) shows that the communications between processors are rapidly prohibitive provided increasing CPU times from a small number of processors; the reason is that the example (I) is a small problem. On the contrary, the best speed-up and efficiency observed are for example (IV), which depicts the largest problem presented.

7 Conclusions

Tree-based multiple-point statistics algorithms are limited by the important memory usage available. On the contrary, list-based implementations demand low memory requirements but are generally slower in serial versions, because the branches of the tree speeds up the retrieval of the multiple-point statistics. The new approach presented in this paper consists in storing the multiple-point statistics in lists and then indexing the lists by trees. The indexing trees allows for a substantial gain of time when retrieving the multiple-point statistics required for the simulation of each node

as they provide shortcuts to access the information contained in the lists. Since all the statistics are stored in the lists, the indexing trees are of reduced sizes, and thus use a smaller amount of memory. Their construction is controlled by two parameters that are empirically set based on numerical tests, for keeping a tree of small size and optimizing the computational time. Hence, this new method benefits from both structures of tree and list: improved CPU performances and low memory requirements. Moreover, since the statistics are stored in lists, all the features of the list-based algorithm are still available for the new method (nonstationary simulations using auxiliary variables, simulation by zones, rotations, affinities). In particular, the proposed method benefits from the parallelization of the lists, and is therefore well-suited for multicores desktop machines or High Performance Computing clusters. Thus, the resulting new parallel algorithm constitutes an important improvement of the list-based algorithm *impala* (Straubhaar et al. 2011), in terms of CPU performances.

Acknowledgements We are grateful to Christian Höcker from Backer Hughes, to Pierre Biver from Total and to Jacques Rivoirard from Mines Paris Tech for providing us training images. We also thank the anonymous reviewers for their comments.

References

Allard D, Comunian A, Renard P (2012) Probability aggregation methods in geoscience. Math Geosci 44(5):545–581. doi:10.1007/s11004-012-9396-3

Arpat G, Caers J (2007) Conditional simulation with patterns. Math Geol 39(2):177–203. doi:10.1007/s11004-006-9075-3

Caers J, Strebelle S, Payrazyan K (2003) Stochastic integration of seismic data and geologic scenarios: a West Africa submarine channel saga. Lead Edge 22(3):192–196

Chugunova T, Hu L (2008) Multiple-point statistical simulations constrained by continuous auxiliary data. Math Geosci 40(2):133–146. doi:10.1007/s11004-007-9142-4

Comunian A, Renard P, Straubhaar J, Bayer P (2011) Three-dimensional high resolution fluvio-glacial aquifer analog—part 2: geostatistical modeling. J Hydrol 405(1–2):10–23. doi:10.1016/j.jhydrol.2011.03.037

de Iaco S, Maggio S (2011) Validation techniques for geological patterns simulations based on variogram and multiple-point statistics. Math Geosci 43(4):483–500. doi:10.1007/s11004-011-9326-9

Guardiano F, Strivastava R (1993) Multivariate geostatistics: beyond bivariate moments. In: Soares A (ed) Geostatistics Troia, vol 1. Kluwer Academic, Dordrecht, pp 133–144

Hajizadeh A, Safekordi A, Farhadpour FA (2011) A multiple-point statistics algorithm for 3D pore space reconstruction from 2D images. Adv Water Resour 34(10):1256–1267. doi:10.1016/j.advwatres.2011.06.003

Journel A, Zhang T (2006) Necessity of a multiple-point prior model. Math Geol 38(5):591–610. doi:10.1007/s11004-006-9031-2

Kessler TC, Comunian A, Oriani F, Renard P, Nilsson B, Klint KE, Bjerg PL (2012) Modeling fine-scale geological heterogeneity—examples of sand lenses in tills. Ground Water. doi:10.1111/j.1745-6584.2012.01015.x

Liu Y, Harding A, Abriel W, Strebelle S (2004) Multiple-point simulation integrating wells, three-dimensional seismic data, and geology. Am Assoc Pet Geol Bull 88(7):905–921. doi:10.1306/02170403078

Mariethoz G, McCabe M, Renard P (2012) Spatiotemporal reconstruction of gaps in multivariate fields using the direct sampling approach. Water Resour Res 48 W10507. doi:10.1029/2012WR012115

Mariethoz G, Renard P, Straubhaar J (2010) The direct sampling method to perform multiple-point geostatistical simulations. Water Resour Res 46 W11536. doi:10.1029/2008WR007621

Renard P (2007) Stochastic hydrogeology: what professionals really need? Ground Water 45(5):531–541. doi:10.1111/j.1745-6584.2007.00340.x

Renard P, Straubhaar J, Caers J, Mariethoz G (2011) Conditioning facies simulations with connectivity data. Math Geosci 43(8):879–903. doi:10.1007/s11004-011-9363-4

Stien M, Hauge R, Kolbjørnsen O, Abrahamsen P (2007) Modification of the SNESIM algorithm. In: EAGE: petroleum geostatistics, Cascais, Portugal, 2007

Straubhaar J, Renard P, Mariethoz G, Froidevaux R, Besson O (2011) An improved parallel multiple-point algorithm using a list approach. Math Geosci 43(3):305–328. doi:10.1007/s11004-011-9328-7

Strebelle S (2002) Conditional simulation of complex geological structures using multiple-points statistics. Math Geol 34(1):1–21. doi:10.1023/A:1014009426274

Strebelle S, Remy N (2005) Post-processing of multiple-point geostatistical models to improve reproduction of training patterns. In: Leuangthong O, Deutsch C (eds) Geostatistics banff 2004. Springer, Berlin, pp 979–988

Suzuki S, Strebelle S (2007) Real-time post-processing method to enhance multiple-point statistics simulation. In: EAGE: petroleum geostatistics, 2007, Cascais, Portugal

Wu J, Zhang T, Journel A (2008) Fast filtersim simulation with score-based distance. Math Geosci 40(7):773–788. doi:10.1007/s11004-008-9157-5

Zhang T, Pedersen SI, Knudby C, McCormick D (2012) Memory-efficient categorical multi-point statistics algorithms based on compact search trees. Math Geosci 44:863–879. doi:10.1007/s11004-012-9412-7

Zhang T, Switzer P, Journel AG (2006) Filter-based classification of training image patterns for spatial simulation. Math Geol 38(1):63–80. doi:10.1007/s11004-005-9004-x