Utrecht University

Master thesis

Parallelization strategies for random path generation

Author:
Atze Schipper
5991781

Supervisor:
Dr. Mathieu Gravey

Second reader:
Drs. Kor de Jong

14 ECTS
Master Applied Data Science

July 1, 2022

Abstract

Geostatistical sequential simulations are becoming larger and larger. In recent time, many parts of the simulation process have been parallelized to accommodate this growth. One part of the process that has not yet been parallelized, but at some point inevitably will have to be, is the calculation of the random path. In this paper, multiple methods of parallelization of the random path generation are presented. The most efficient one will completely parallelize this process and will require no communication between computers about the random path. All methods have been implemented to make analysis of quality of randomness possible.

Contents

1. Introduction

Geostatistics is a subdomain of statistics focusing on statistics in spatial context. It provides a way to quantify uncertainty over spatially distributed variables. It was originally developed to estimate ore reserves in the ground (Krige, 1951; Sichel, 1952), but nowadays is applied in a wide range of applications; ranging from hydrogeology (Kitanidis, 1997) to agriculture (Buttafuoco and Lucà, 2016) for example. An early geostatistical method is called Kriging (Krige, 1951) and was developed to interpolate unknown spatial values. One of the drawbacks of such an estimation method however, is that each estimated value is the most likely value according to the known values (Gómez-Hernández and Srivastava, 2021). This leads to an underestimation of the variance of the underlying random function. To overcome this, simulation methods such as sequential indicator simulation (SIS) (Journel et al., 1998) and sequential Gaussian simulation (SGS) (Deutsch, Journel, et al., 1992) have been developed. These type of methods provide a distribution of values instead of only a single optimal estimation, by implying spatial relations between pairs of points in the form of covariance functions. These sequential simulation methods are used to generate realizations of a random field by discretizing the field into a simulation grid. Then for each point in the grid, an outcome of the random variable is computed based on previously computed points and potential locally available information (Mariethoz, 2010). Theoretically, all previously computed points should be used in the simulation of a new point. However, this would mean too heavy of a computational burden as simulation grids become larger. For that reason usually only a limited neighborhood of the point is used when simulating them. The order in which the points of the simulation grid are computed is pre-arranged in the form of a path. This path is assumed to be optimal when it is completely random. In that way, it does not introduce constant bias between different realizations of the simulation (Nussbaumer et al., 2018). A successor to these sequential simulation methods, is is multiple-point statistics (MPS) (Guardiano and Srivastava, 1993), another sequential simulation method. This method relies on the usage of a training image, from which information is used to simulate the rest of the simulation grid. In early versions of MPS, the entire training image had to be scanned for each simulated point (Guardiano and Srivastava, 1993. This method was too computationally expensive to be used (Mariethoz et al., 2010) and improvements have been made since, for example by (Strebelle, 2002), who proposed storing the information of the training image in a search tree. This however caused problems with excessive memory usage (Mariethoz et al., 2010). More improvements in efficiency have been made among others by (Straubhaar et al., 2011), who proposed using lists instead of trees, and (Mariethoz et al., 2010), who came up with the direct sampling method. These improvements of MPS allowed for parallelization of the simulation

process on shared-memory machines, making larger simulations possible. However, even with these improvements, further parallelization remained necessary to keep up with growing simulation grids and increasingly complex spatial models. Mariethoz came up with a taxonomy of three levels of parallelization: realization-level, path-level and node-level. The aforementioned parallelization was on node-level, where multiple processors can work together to simulate a single node (point in the simulation grid). Path-level parallelization is the division of the simulation grid into multiple sections, each to be simulated by a different processor. How this can be achieved is described in (Mariethoz, 2010). The last level of parallelization on realization level is the easiest one to achieve, and boils down to having each complete simulation done by a different processor (for example Mariethoz et al., 2009). Parallelization can be done either on shared-memory systems, or on clusters of computers where each processor has its own dedicated memory. The problem of shared-memory systems is that they are expensive and limited in amount of memory or number of processors, so often clusters of computers are used. The drawback of these clusters without shared memory is that communication can only be done by sending and receiving messages. This is very slow compared to the speed of the processors and often communication forms the bottleneck in computation time (Mariethoz, 2010).

Currently, path-level parallelization methods (e.g. Mariethoz, 2010; Peredo et al., 2018), only parallelize the simulation of the nodes. They still rely on a single computer generating the random path. This becomes a problem as simulations keep growing in size and complexity, the generation of the random path becomes then a non-negligible part of the simulation process. In this paper, multiple methods of (partial and complete) parallelization of path generation are being proposed. Such parallelizations can easily be approximated, but those approximations will always be biased. All methods described here that completely parallelize the process are completely unbiased, except for one method that partially parallelizes. Parallelization of the path generation process will improve the total process of simulation in two ways. Primarily, parallelization of the path generation will reduce (or with some methods proposed here even completely eliminate) the need for communication between computational units about the random path. Secondly, parallelization will reduce the computational time that is needed for the path generation. It also provides a solution for large-scale simulations for which the memory of a single computer is insufficiently large to contain the entire random path. With parallelization of the random path, the size of the random path as limiting factor for the scale of the simulation will be extended and even larger simulations will be possible. The above has led to the following research question:

"How to optimally generate a large random path for geostatistical simulations on a distributed memory system efficiently."

To answer this question, the rest of this paper is structured as follows: section 2 will introduce a background on random paths in general and metrics that can be used to measure the quality of random paths. Section 3 describes five methods of random path generation, ranging from completely un-parallelized to completely parallelized. The different methods are then compared using the metric in section 4, and the strengths and weaknesses of each method will be summed up in section 5. The paper will finally end with conclusions in section 6.

2. Background

A path is a vector of the same size as the simulation grid, indicating the order in which the algorithm visits all the locations (points) in the simulation grid. There is thus a one-to-one correspondence from all *elements* of the path to all *locations* (nodes) in the simulation grid. A random path is a permutation of all the locations in the simulation grid. There cannot be any ambiguity as to which point is to be simulated first for points in each others neighborhood.

To create a random path for a simulation grid that is to be simulated in parallel by multiple computers, first the simulation grid needs to be split into different sections. Each section belonging to a different computer. This may be done in different ways, in this paper it was chosen to divide the simulation grid in vertical slices. See figure 2.1 for an example. The columns are numbered c0 to c14 to ease referencing. In this figure, a simulation grid is drawn. The total length of the path is from here on referenced as n. The number of computers used in the simulation is defined as k. The random path is split in multiple *subpaths*, each of length s. Each subpath will be simulated by a single computer. The margins are the areas of the paths where the elements have elements of the path of the adjacent computer in their neighborhood, and generally are less than 10% the size of s. The margins can be at maximum be $\frac{1}{3}$ of the size of the subpath. The length of the margin is defined as m. The subpath of the middle computer ranges from column 5-9. The margins in columns 4 and 10 are the neighboring margins of the middle computer, and the margins in columns 5 and 9 are its own margins.

As there cannot be any ambiguity as to which point is to be simulated first, this is especially relevant for points with a neighborhood that lies in different sections. This specific situation only arises for points that are located in margins. If we take a look at the middle computer of figure 2.1, we can see that inevitably duplicates occur between the nodes in the outer margins and the nodes of the path between the inner margins. As these elements are not in each others neighborhood, this is of no consequence, and it is arbitrary which point gets simulated first.

The Durstenfeld algorithm (Durstenfeld, 1964) is the basis of all path generation algorithms of this paper. It is used to shuffle a sequence, and is an improvement over an earlier shuffle algorithm by Fisher and Yates (Fisher and Yates, 1953), reducing its time complexity from $\mathcal{O}(n\ log\ n)$ to $\mathcal{O}(n)$. The space needed by the improved algorithm is halved. The pseudocode of the Durstenfeld algorithm can be found in algorithm 1.

7	13	16	4	11	3	10	15	4	12	15	9	21	14	20
8	19	2	12	0	5	2	17	6	7	18	16	22	6	19
24	10	18	20	3	24	14	23	18	20	17	2	13	23	1
17	1	21	9	23	21	1	0	19	13	8	12	0	10	24
22	14	6	5	15	22	16	8	9	11	4	7	3	5	11
c0	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14

Figure 2.1: A random path, with $k = 3$, $n = 75$, $s = 25$, $m = 5$ and the margins drawn in blue.

Algorithm 1 Durstenfeld shuffle

To shuffle an array a of n elements (indices 0..n-1):

1: **for** i from n - 1 downto 1 **do**
2: j ← random integer such that $0 \leqslant j \leqslant i$
3: exchange a[j] and a[i]

2.1 Measure of randomness

Because a single random path is a single realization of a stochastical process, multiple realizations are needed to gauge the quality of randomness for a given method producing random paths. With a perfect method generating the random path, all paths are equiprobable and therefore no pattern can be distinguished between the different realizations. This allows for valid assesment of the output. There are different ways of measuring the quality of randomness of a set of random paths, some of which are more suitable than others. One measure could for example be to use the average, using the following equation with m as the number of locations and n as the number of iterations:

$$distance = \sum_{l=1}^{l=m} \left(\frac{\left|\sum_{i=1}^{i=n}\right|}{n}\right) \tag{2.1}$$

The problem with this measure is that it fails to detect patterns such as even numbers only being in even locations and vice versa. Therefore it is not suitable to be used here. The measure used here compares the distance between each path-value in each location to the theoretical fraction of a value in a location. More on this can be found in section 3.6.

3. Methods

3.1 Non-parallelized (baseline)

This completely non-parallelized way of path generation will function as a baseline method to compare the other methods to. See algorithm 2 for the pseudocode. A single computer calculates a path of length n (with $n = k \cdot s$), and then splits up the path into subpaths for the computers to use. Both the time and space complexity of this method is $\mathcal{O}(k \cdot s)$. It could occur that there are duplicates in neighboring margins from two different computers, but this poses no problem as the order from the original path of length n can be used to resolve ambiguity regarding the order in those margins.

Algorithm 2 Non parallelized path generation

1: Shuffle an array of length n using the Durstenfeld algorithm
2: Split this array into k equal parts
3: Send each computer its subpath & neighboring margins

3.2 Only margins non-parallelized (random order)

This method is a first step in the complete parallelization of the path calculation by only computing the margins beforehand by a single computer. This is an operation of $\mathcal{O}(2 \cdot k \cdot m)$. All computers then get sent their own and neighboring margins, and can in parallel compute the rest of the subpaths. This is done according to algorithm 3, in $\mathcal{O}(s)$.The filling in of the margins happens in a random order, to guarantee no introduction of bias from the order of filling in the margins. How this filling in happens is further described in algorithm 4.

3.3 Only margins non-parallelized (sequential order)

This method is a slight variation to the method described in section 3.2. In this method, a single computer calculates the margins in sequential order instead of in random order, with a complexity of $\mathcal{O}(2 \cdot k \cdot m)$. See algorithm 5. The margins of all computers are being filled in a sequential order, beginning at the margins of the first computer and ending at the margins of the last computer. See figure 3.1 for an example. Depicted are the margins of 10 computers, with $m = 2$. the number of

Algorithm 3 Permutation supplementor

1: marginValues ← [ ]
2: **for** all values in the front margin **do**
3: Add to marginValues
4: **for** all values in the back margin **do**
5: Add to marginValues
6: Populate all empty locations with the remaining values that are not in marginValues
7: Shuffle the area between the margins using algorithm 1

Algorithm 4 Only margins non-parallelized (random order)

1: **for** all margins **do**
2: create a set a of all allowed values for that margin
3: **while** not all margins completely filled **do**
4: Randomly pick an empty location in some margin
5: Randomly draw a value v from the available values of that margin
6: Remove v from a of that margin and from both neighboring margins
7: Send each computer its own and neighboring margins
8: Each computer calculates the rest of its subpath using algorithm 3

the computer to which the margin belongs is written underneath each margin. Note that the outer margins of the first and last computer have been omitted, as those computers have no adjacent computers on those sides. The algorithm its next step in the figure is drawing values for the right margin of computer number three. One can see that there are no duplicate values between adjacent margins, or within a margin itself. When all margins are calculated, the margins get sent to the computers (a memory transfer of $\mathcal{O}(4m)$). The computers can then in parallel calculate the rest of the subpath, just as with the previous method.

3.3.1 Further parallelization

The process of calculating the margins may be parallelized by dividing the margins that need to be calculated between a number of computers, to decrease calculation time. The computers that calculate the margins only need to communicate the values of the margins that are adjacent to margins that are calculated by a different computer. This implies that there is some optimal value for the number of computers that is being used to calculate the margins. If this value is too large, the increased need for communication between computers about adjacent margins will lead to a larger run time compared to a single computer calculating all the margins. Calculating the margins with two computers require splitting the margins once. The number of splits is defined as c. The complexity for the calculation of the margins with this parallelization strategy is $\mathcal{O}(\frac{1}{c}(2 \cdot k \cdot m))$, but with an added cost of $\mathcal{O}(c \cdot m)$ of memory transfer.

0	2	6	2	0	6	...											
3	1	3	1	7	5	...											
0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	9

Figure 3.1: Sequential filling of margins.

Algorithm 5 Permutate margins in sequential order

1: **for** each margin from left to right **do**
2: Draw a new random value for each empty location in the current margin
3: if that value is already present in the previous or current margin, redraw
4:
5: Send each computer its margins & neighboring margins
6: Each computer calculates the rest of its subpath using algorithm 3

3.4 parity consistent margins

This method of path calculation is done completely in parallel, without any communication between the computers about the subpath. Each computer only needs to receive their neighboring margins from the respective computers, meaning a memory transfer of $\mathcal{O}(2m)$. An extra constraint for the margins is added: in the left margin may only be values of even parity, and vice versa for the right margin. If this is the case, the margins are parity consistent. This automatically ensures that there are no duplicates in neighboring margins. The cost of this method however, is that not all values have an equal probability for all locations in the path (all even numbers have a probability of 0 to end up in the right margin, and vice versa). This means that there is some information on points repartition, and the process is not completely random. Whether this poses a problem will be further discussed in section 5.1.

See algorithm 6 for the pseudocode, and figure 3.3 for a visualization of the algorithm. Before algorithm 6 is ran, first the computer creates a subpath using algorithm 1. The figure should be interpreted as follows. The 7 and 17 from the left margin are being swapped with the 18 and 6 of the right margin. Upon finishing the right margin, the 21 remains as last odd number in the left margin. That value is then being swapped with a random number of even parity from the rest of the path.

7	13	16	4	11	10	3	12	4	15	16	9	21	14	20
8	19	2	12	9	2	5	17	6	7	18	15	22	6	19
24	10	18	20	3	24	14	23	18	19	2	17	13	23	1
17	1	21	0	23	0	1	21	20	13	8	12	0	10	24
22	14	6	5	15	22	16	8	9	11	4	7	3	5	11

Figure 3.2: Parity consistent margins

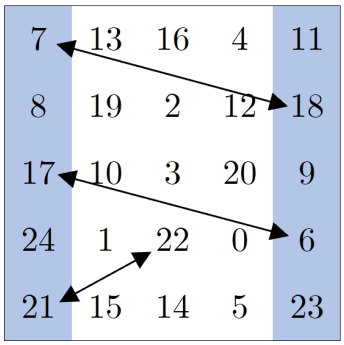

Figure 3.3: A visualization of the parity consistent margins algorithm.

The complete time complexity of this method is $\mathcal{O}(s + 2m)$: $\mathcal{O}(s)$ for the initial shuffling of the subpath, and $\mathcal{O}(2m)$ for making the inner margins parity consistent.

Algorithm 6 Parity consistent margins

1: **for** all elements in the left margin **do**
2: **if** an element is odd **then**
3: Swap with the first even element in the right margin
4: **if** one of the margins is finished before the other margin **then**
5: Swap the remaining values of wrong parity with values of correct parity from the rest of the path

3.5 Shared seed

The last method is the only method done completely in parallel, without need for any communication about margins or any other part of the path. A visualization of the algorithm can be found in figure 3.4. For the pseudocode see algorithm 7. The computers that will together simulate the simulation grid are each numbered consecutively with a natural number. This number doubles as identification number of the computer, as well as the seed that the computer uses in the randomization process. We focus on computer k, with neighboring computers $k-1$ and $k+1$. In the first part of the figure the starting state is depicted, with no path yet calculated. In the second part, computer k uses the seed $k-1$ to calculate the right side (with a size of $2m$) of the path of $k-1$, which is also the left side of the path of k. In the third part, computer k uses its own seed to calculate a permutation of length $2m$ from the original sequence s, which will be the right side of the path. Then finally in the fourth part of the figure, the rest of the subpath is created. This is done by removing the values from the inner margins from a sequence with the length of s, and shuffling that sequence. At this point, computer k has a complete subpath and both adjacent margins and needs no further information to proceed. computer $k+1$ can likewise use the seed k to calculate the margins that k and $k+1$ share; et cetera. This algorithm runs with a time complexity of $\mathcal{O}(s+2m)$. A variant on this method would be one where the right margin is sent instead of computed locally.

Observe that, because the subpath is calculated using two different seeds, duplicates are possible between the parts that were calculated with a different seed.

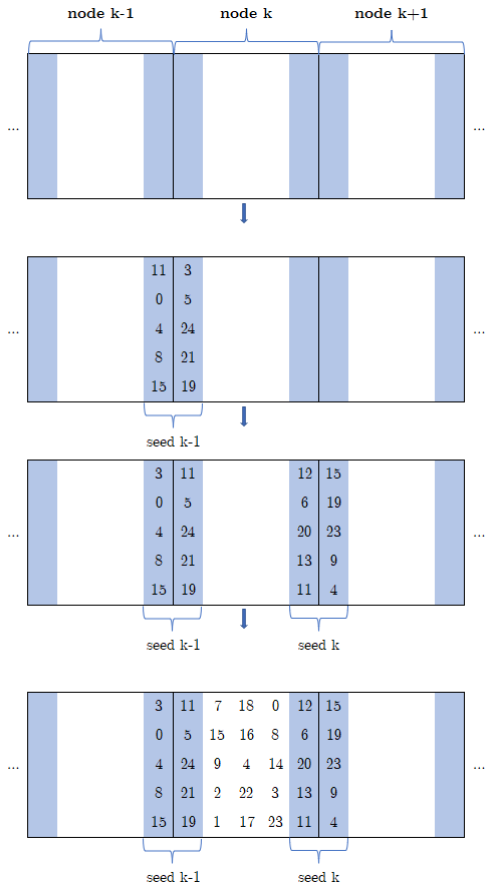

Figure 3.4: Completely parallelized path generation with shared seed.

This is different compared to all the other methods, as now duplicates within the same subpath are possible as well. This however does not pose a problem, as those duplicates can only occur between the two inner margins. These values are by definition not in each others neighborhood, and therefore their mutual order is of no consequence for the outcome of the simulation.

Algorithm 7 Random path generation with shared seed

1: For the two left margins: get a sequence of length s and shuffle a part of length $2m$ using seed $k-1$
2: For the two right margins: get a sequence of length s and shuffle a part of length $2m$ using seed k
3: Get a sequence of length s and remove the values that are already in both the inner margins
4: Shuffle this sequence and use it for the part of the path between the margins

3.6 Metric

To measure the quality of randomness of some method providing a random path, the algorithm described in algorithm 8 is used. Recall from section 2.1 that multiple realizations of a random path are needed to measure the quality of randomness.

That number of realizations is expressed in the number of iterations that the algorithm runs for. The algorithm starts by filling the *countArray*, a square array that is the Cartesian product of all possible values and locations. It then calculates the Euclidean distance of each value x location pair to the theoretical fraction ($\frac{1}{\text{length of random path}}$), and adds all those up to get the total distance. With a perfect randomization method and the total distance as a function of iterations (i):

$$\lim_{i \to \infty} f(i) = 0 \tag{3.1}$$

The shortcomings of this metric will be discussed in section 5.2.

Algorithm 8 Total distance to mean

```
countArray ← square array of the same length as the random path
for i = 0 to maximum number of iterations do
    randomPath ← a random path provided by some method
    for j = 0 to length of random Path do
        value ← randomPath[j]
        countArray[value, j]++

distSum = 0
theoreticalFraction = 1 / length of random path
for i = 0 to length of random path do
    for j = 0 to length of random path do
        fraction = countArray[i,j] / iterations
        distance = (fraction - theoreticalFraction)^2
        distSum += distance
return √distSum
```

4. Results

To assess the quality of randomness of the paths provided by the different methods, all methods have been implemented to make testing possible. The implementation was done in the programming language C# and a reference to the implementation can be found in the appendix. All the plots show the application of the metric as described in section 3.6, with the total distance to the mean plotted against the number of iterations. Recall from the same section that a perfect randomization method shows a decreasing plot that tends to 0 as the number of iterations increases. Both plots are generated with $s = 100$, $m = 10$, $k = 10$ and with 1000 as maximum number of iterations, unless specified otherwise.

The results of the application of the metric onto the method described in section 3.2 and section 3.3 can be found in figure 4.1, together with the baseline method for reference. For the method filling the margins in random order, different ratios of length of margin versus length of subpath have been plotted. There is no observable difference in trend between the plots for the different ratios. The figure also shows the plot for the method filling in the margins in sequential order instead of random order. Again, no difference in trend is present, indicating that the quality of randomness as it can be measured by the metric used here is equivalent for these methods. It can thus be concluded that filling the margins beforehand in sequential order is to be preferred over filling them in random order, as the technique is simpler and opportunities for further parallelization are possible with sequential-order filling.

Figure 4.2 plots the method with parity consistent margins against baseline. The plot visualized the fact that the method performs worse the as the proportion of m to s increases. As we have seen in section 3.4, all elements have a probability of zero for some locations in the simulation grid. As m increases relative to s, the number of locations where this is true increases as well, influencing the performance of the method negatively.

The method with the shared seed is only different from the baseline method in the fact that the randomized permutation of length s is shifted right for the length of $2m$. This has no influence on performance as measured by this metric.

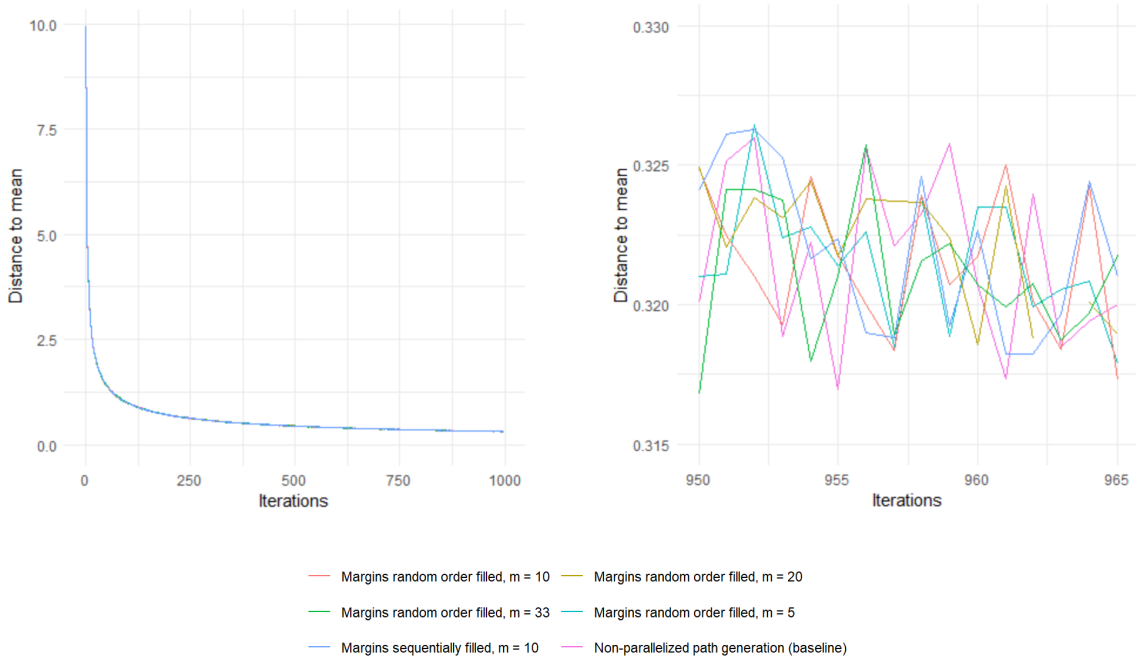

Figure 4.1: Results for sequential and random order filling of margins with different values of m, against baseline (the right figure is a zoomed in version).

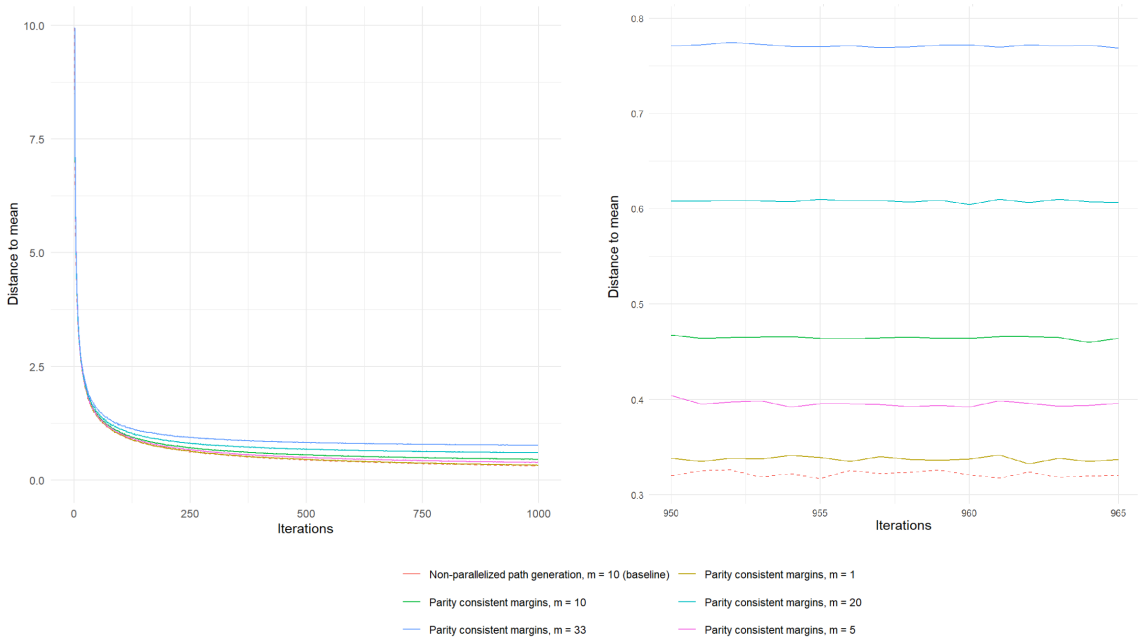

Figure 4.2: Results for methods with parity consistent margins against baseline, with different values for m (the right figure is a zoomed in version).

5. Discussion

5.1 Methods

Table 5.1 provides an overview of the different methods, with respect to time- and memory transfer complexity. The method of filling in margins in random order has been left out of this summary, as it has no advantages over filling margins in sequential order. Recall that s is the length of the subpath (including 2 margins), k is the number of computers used in parallelization and m is the length of the margin. The number of splits in the parallelization of sequentially computed margins is defined as c. Note that the big O notation can be simplified in some parts of the table, but this has been omitted to make comparisons possible between $\mathcal{O}(4m)$ and $\mathcal{O}(2m)$ for example.

	Non-parallelized time complexity	Parallelized time complexity	Memory transfer
Baseline	$\mathcal{O}(k \cdot s)$	0	$\mathcal{O}(k \cdot s)$
Sequential margins	$\mathcal{O}(2 \cdot k \cdot m)$	$\mathcal{O}(s)$	$\mathcal{O}(2 \cdot k \cdot m)$
Sequential margins parallelized	$\frac{1}{c}(2 \cdot k \cdot m)$	$\mathcal{O}(s)$	$\mathcal{O}(4m) + \mathcal{O}(c \cdot m)$
Parity consistent margins	0	$\mathcal{O}(s + 2m)$	$\mathcal{O}(2m)$
shared seed	0	$\mathcal{O}(s + 2m)$	0
shared seed with sent margin	0	$\mathcal{O}(s)$	$\mathcal{O}(2m)$

Table 5.1: A comparison of the different methods of random path calculation.

One can see that preparing the margins beforehand by a single computer and allowing the calculations of the subpaths to be parallelized as is done in the sequential margins method already greatly improves performance over the baseline (completely non-parallelized) method. Note that there is also an improvement in space complexity, as it is not necessary anymore to have the entire random path n in the memory of a single computer. Note that for the parallelization of the computation of the margins in this method, an additional memory transfer is required between the computers calculating the margins beforehand. This is needed for the pair of adjacent margins between the sets of margins that are being calculated by different computers. This means that memory transfer costs go up the more parallelization of the margin calculation occurs. This implies that there is some optimal number of c, where the balance between time won by parallelization and lost by memory transfer is optimal. The method of parity consistent margins completely removes the need for non-parallelized computation, but introduces some bias in the random path. The amount of bias depends on the size of m relative to s, as we have seen in section 4. Whether this is acceptable or not (or for what value of m)

should be further researched. The method with the shared seed is the most efficient in parallelizing a random path completely without bias, requiring no unparallelized computation and being the only method without need for memory transfer. The variation on this method where the left margins are received instead of calculated by the computer itself is a trade off between calculation time and memory transfer time. Whether this is approach is superior depends on different variables; such as memory transfer speed, computation speed, relative size of m to s et cetera. Finding the preferred method will require real world testing.

An important factor in efficiency of the parallelization of random path generation is the chosen size of the neighborhood for the nodes. It affects the efficiency of the parallelization methods that are described in this paper in two ways, as the size of the neighborhood directly determines m. The efficiency of parallelization is negatively correlated to the size of m relative to the size of s, as having relatively small margins means less memory transfer is needed for most methods. Another advantage of having a relative small size of m, is that the maximum value for k increases. As m cannot be larger than $\frac{1}{3}$ of s, m is the deciding factor in the maximum number of subpaths and thus computers used. Note that all the methods that are described in this paper assume a 2-dimensional simulation grid. Future research could include methods to extend path-generation parallelization methods to handle 3-dimensional simulation grids.

5.2 Metric

The metric that has been used in this research is no perfect measure of randomness, which is in any case impossible to achieve with a single metric. This metric tests relations between locations and values of the path, by counting the fraction of times that some value in the path is in some location of the simulation grid. What it therefore fails to detect are relations between values in the path. If for example the first and last value of the path would always occur next to each other in a random part of the simulation grid, the metric used here would not be able to detect this. This poses a problem, as the question of the relative order between points is just as relevant as the question of whether some location is biased to being simulated earlier in the simulation. After all, the most relevant problem during simulation of some point for all the points in its neighborhood is whether they should be simulated before or after the point in question. Future research could include different metrics, that could overcome these limitations.

An interesting observation that has been done during this research was when experimenting with the distance metric used by algorithm 8. Throughout the paper Euclidian distance has been used. When substituting this for Manhattan distance, 'bumps' appear between ranges of iterations that are exactly the size of the path that is to be randomized. See figure 5.1 for an example. The plot here is exactly the same plot as the baseline plot in figure 4.1, barring the distance metric. For this effect to happen, the length of the randomly permutated sequence must be sufficiently large. With a length of 10 the effect is not visible, with lengths over 100 the effect starts to become clearly visible.

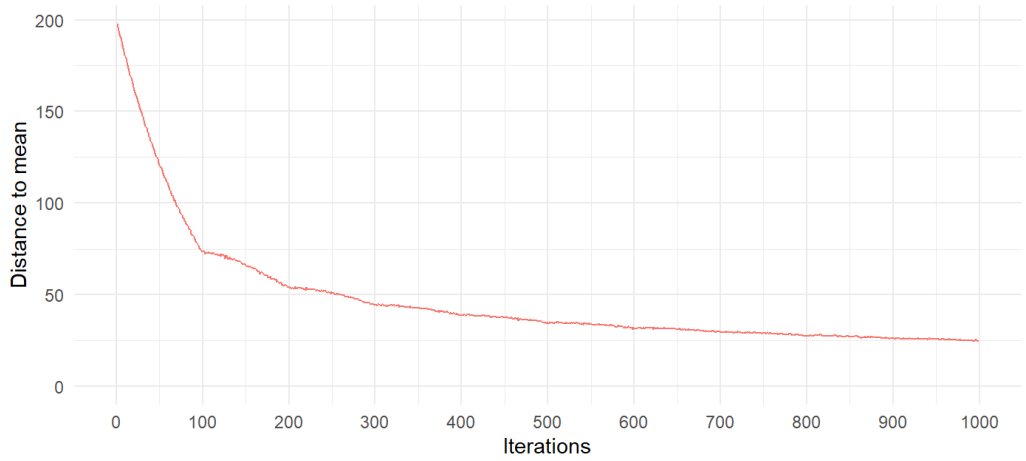

Figure 5.1: Manhattan distance instead of Euclidian distance used as distance metric.

6. Conclusion

In this paper, multiple methods for parallelization of random path generation for sequential geostatistical simulations have been proposed. The method with the shared seed is the largest improvement over the completely un-parallelized method, requiring no un-parallelized computing time. It also completely eliminates the need for any memory transfer between computers. A variation on it exchanges some computing time for an increase in memory transfer. From the above can be concluded that the method with the shared seed (or possibly its variation) would be a valid answer to the research question as posed in section 1. Future research could include generalization of these findings to 3-dimensional simulation grids and utilization of different metrics.

References

Buttafuoco, G., & Lucà, F. (2016). The contribution of geostatistics to precision agriculture. *Ann. Agric. Crop Sci*, *1*(2), 1–2.

Deutsch, C. V., Journel, A. G. et al. (1992). Geostatistical software library and user's guide. *Oxford University Press*, *8*(91), 0–1.

Durstenfeld, R. (1964). Algorithm 235: Random permutation. *Communications of the ACM*, *7*(7), 420.

Fisher, R. A., & Yates, F. (1953). *Statistical tables for biological, agricultural and medical research*. Hafner Publishing Company.

Gómez-Hernández, J. J., & Srivastava, R. M. (2021). One step at a time: The origins of sequential simulation and beyond. *Mathematical Geosciences*, *53*(2), 193–209.

Guardiano, F. B., & Srivastava, R. M. (1993). Multivariate geostatistics: Beyond bivariate moments. *Geostatistics troia'92* (pp. 133–144). Springer.

Journel, A., Gundeso, R., Gringarten, E., & Yao, T. (1998). Stochastic modelling of a fluvial reservoir: A comparative review of algorithms. *Journal of Petroleum Science and Engineering*, *21*(1-2), 95–121.

Kitanidis, P. K. (1997). *Introduction to geostatistics: Applications in hydrogeology*. Cambridge university press.

Krige, D. G. (1951). A statistical approach to some basic mine valuation problems on the witwatersrand. *Journal of the Southern African Institute of Mining and Metallurgy*, *52*(6), 119–139.

Mariethoz, G. (2010). A general parallelization strategy for random path based geostatistical simulation methods. *Computers & Geosciences*, *36*(7), 953–958.

Mariethoz, G., Renard, P., & Straubhaar, J. (2010). The direct sampling method to perform multiple-point geostatistical simulations. *Water Resources Research*, *46*(11).

Mariethoz, G., Renard, P., Cornaton, F., & Jaquet, O. (2009). Truncated plurigaussian simulations to characterize aquifer heterogeneity. *Groundwater*, *47*(1), 13–24.

Nussbaumer, R., Mariethoz, G., Gloaguen, E., & Holliger, K. (2018). Which path to choose in sequential gaussian simulation. *Mathematical Geosciences*, *50*(1), 97–120.

Peredo, O. F., Baeza, D., Ortiz, J. M., & Herrero, J. R. (2018). A path-level exact parallelization strategy for sequential simulation. *Computers & Geosciences*, *110*, 10–22.

Sichel, H. S. (1952). New methods in the statistical evaluation of mine sampling data. *Lond Inst Min Metall Trans*, *61*, 261–288.

Straubhaar, J., Renard, P., Mariethoz, G., Froidevaux, R., & Besson, O. (2011). An improved parallel multiple-point algorithm using a list approach. *Mathematical Geosciences*, *43*(3), 305–328.

Strebelle, S. (2002). Conditional simulation of complex geological structures using multiple-point statistics. *Mathematical geology*, *34*(1), 1–21.

Appendix

Implementations of all algorithms described in pseudocode in this paper are available at the following location: https://github.com/Schipper5/Random-Path-thesis.