Large Scale Geostatistics with Locally Varying Anisotropy

by
Oscar Francisco Peredo Andrade

Advisor
José R. Herrero

DISSERTATION
Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Architecture

Universitat Politècnica de Catalunya
2022
Barcelona, Spain

Large Scale Geostatistics with Locally Varying Anisotropy

Oscar Francisco Peredo Andrade

Abstract

Classical geostatistical methods are based on the hypothesis of stationarity, which allows to apply repetitive sampling in different locations of the spatial domain, in order to obtain enough information to infer cumulative distributions. In case of non stationarity, anisotropy is observed in the underlying physical phenomena. This feature manifest itself as preferential directions of continuity in the phenomena, i.e. properties are more continuous in one orientation than in another. In the case of local anisotropy, each location of the domain in study presents different preferential directions of continuity. The locally varying anisotropy (LVA) approach in geostatistics allows to incorporate a field of local anisotropy parameters defined for each domain point. With this additional input, more realistic spatial simulations can be generated, including geological features to the computational model such as folds, veins, faults, among others. Since the seminal article published by Boisvert and Deutsch (2011), to the best of the author's knowledge, no further analysis or public code improvements were developed. This is in part because acceleration and parallelization techniques must be applied to the inner kernels of the baseline LVA codes. Large execution time is needed to generate small-scale domain simulations, making large-scale domain simulations a prohibitive task.

The contributions of this thesis are accelerating and parallelizing classical and LVA-based geostatistical simulation methods, particularly sequential simulation, which is one of the most common and computationally intensive methods in the field. This fact was recently remarked by some of the main authors in the field, Gómez-Hernández and Srivastava (2021), which shows the relevance of this work today. Two main parallel algorithms and an optimized version of a kd-tree search implementation are presented, all of them applied to both classical and LVA-based sequential simulation implementations. The first parallel algorithm is related to the parallel simulation of different domain points, after rearranging the order of simulation but preserving the exact results of a single-thread execution. The second parallel algorithm is related to the parallel search of neighbour points in the domain, which will be used to build data dependencies for the parallel simulation of points. The optimized kd-tree search was used in each test case in order to reduce the computational complexity of neighbour search tasks. Its modified implementation reduces the number of branching instructions and introduces specialized code sections to accelerate the execution. The main focus is on multi-core architectures using OpenMP and optimization techniques applied to Fortran and C++ codes.

Additionally, acceleration and parallelization techniques were also applied to auxiliary applications, such as shortest path and variogram calculation on hybrid CPU/GPU architectures using Fortran, C++ and CUDA codes. In the last application, an analytical and heuristic model was developed to estimate the optimal workload distribution between CPU and GPU in the hybrid context.

The overall results of this work are a set of applications that will allow researchers and practitioners to accelerate dramatically the execution of their experiments and simulations, being `sgsim`, `sisim`, `sgs-lva` and `sisim-lva` the accelerated codes presented. Final speedup results of 11× and 50× are obtained for non-LVA codes using 16 threads, and 56× and 1822× are obtained for LVA codes using 20 threads. These tools can be combined with other geostatistical tools, in order to improve the existing landscape of open source codes that can be used in practical scenarios.

Agradecimientos

Antes de agradecer a todos y todas quienes me han ayudado en el desarrollo de esta tesis, quisiera explicar el contexto histórico durante el cual nos encontramos como sociedad, al momento de escribir estas líneas.

A fines del año 2019, se comenzó a propagar un virus llamado COVID-19, coloquialmente llamado "Coronavirus", que llegó a propagarse por la mayoría de los países del mundo en pocos meses. A comienzos de Marzo del año 2020, el virus impactó mi país, Chile, y a partir de esa fecha todo cambió, tanto para mal como para bien. Por un lado, meses de confinamiento y aislamiento, distancia social, precauciones y cuidados sanitarios, eran parte del día a día y poco a poco nos acostumbramos a esa nueva normalidad. Las empresas y comercios se adaptaron, y las familias también. Por otro lado, por primera vez, tuve la oportunidad de compartir con mi familia más tiempo del que hubiera imaginado, siendo parte importante en la cotidianidad de mis hijos y esposa. Siempre recordaré este período por sus luces y sombras, y como una época donde nos replanteamos nuestras actividades y cuestionamos la manera que teníamos de hacer las cosas antes y después de la pandemia.

Es en este contexto, en medio de un fenómeno histórico, que esta tesis se desarrolló. Quiero agradecer primero a Jose Ramón, mi guía en esta aventura, quien semana tras semana me daba ánimo a continuar y seguir perseverando en los objetivos que íbamos definiendo. Contra viento y marea, siempre lograba dar las palabras y consejos que me motivaban a continuar. Siempre recordaré estos años y por cierto que seguiremos colaborando en proyectos futuros, tanto laborales, académicos, como personales y familiares. Todo este esfuerzo no es solo mío o de José Ramón, en gran medida es un esfuerzo de mi esposa Natalia, quien me acompañó en este proyecto y me entrega su amor y apoyo día a día. Sin ese pilar, nada de esto hubiera sido posible. Esa es la verdad. Nuestra familia comenzó a crecer a medida que prograsaba en el trabajo de tesis, y mis 2 hijos, Pascuala y Vicente, nacieron y crecieron durante estos años. En algunos años más, cuando tengan la edad suficiente, quizás se aventuren a leer estas páginas, y eso despierte su motivación por la generación de conocimiento y la curiosidad por la matemática, informática y los algoritmos. Si están leyendo estas líneas ahora mismo, y si recuerdan aquellas noches o tardes en que su padre tenía que trabajar en el "doctorado", acá está el fruto de todas esas horas... Comencemos...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The concept of spatial variability can be understood in statistical terms as the distribution of a particular phenomena with numerical or labeled values and geographical coordinates in the space. Given a set of locations with values and geographical coordinates, a classical problem regards the quantification of uncertainty of the unknown values in the unsampled locations. The uncertainty can be modelled as the response of several random variables geo-located in the unsampled locations, through their cumulative distribution functions and statistical properties using all the available information in the sampled locations.

Geostatistics offers a set of deterministic and statistical tools aimed at understanding and modeling spatial variability. Exploratory data analysis, estimation and simulation are among the most important topics in this field. Each of these topics involves several techniques and methods, many of them with inner kernels that make heavy use of computational resources. These kernels are based in classical algebraic vector operations, such as direct solvers for linear systems and nearest neighbours search routines. In large scale scenarios, where the number of unsampled locations is extremely large or several nearest neighbours must be searched, the usage of acceleration techniques and technologies is mandatory in order to obtain results in reasonable execution times.

The classical approach can be used in many applied cases where the values under study show isotropic or regular trends of preferential directions. However in complex scenarios the results obtained can be unrealistic since the underlying phenomena shows highly anisotropic trends which can not be reproduced by the classical approach. These kind of complex scenarios arise in geological modeling of faults and veins in mineral reserves, sedimentary deposits in oil and gas reservoirs, environmental modeling of pollution spread, rain fall patterns or animal migration, and mobility patterns in highly populated urban areas.

We seek to make a contribution by optimizing and accelerating some of the classical algorithms in the field of geostatistics, together with new algorithms developed by the research community, particularly in the field of geostatistics with locally varying anisotropy (LVA). Our approach will be focused on increasing the performance of the parallelizations and distributions of workloads obtained, as well as improve the algorithmic complexity of some of the proposed methods, implementations and case studies.

1.1 Context/Motivation

Classical geostatistical methods are based in two-point statistics, particularly covariances and variograms Chilès and Delfiner [1999], Deutsch and Journel [1998]. These measures can be inferred assuming stationarity, which means that the cumulative distribution functions of any set of random variables in any locations are invariant under translation. This assumption allows to apply repetitive sampling in different locations of the sampled data, in order to obtain enough information to infer cumulative distributions.

In case of non stationarity, anisotropy is observed in the underlying phenomena. This feature manifest itself as preferential directions of continuity in the phenomena, i.e. properties are more continuous in one orientation than in another. If constant anisotropy is present, a single trend or drift can be observed in the sampled data set or secondary sources of information (Isaaks and Srivastava [1990]). In the case of local anisotropy, each location of the domain in study presents different preferential directions of continuity (Boisvert [2010], Boisvert and Deutsch [2011]). Figure 1.1 shows the contrast between an isotropic (stationary) and a constant anisotropic (non stationary) images. Both images were generated using a classical method (sequential gaussian simulation). In the left image the structural pattern is independent of the direction in which is measured (omni directional) and the right image shows a clear preferential direction in the west-east axis.

Figure 1.2 shows an example of a LVA field, where each node of the discretized domain has an orientation. In this case the underlying phenomena consists in several stratigraphic layers in an oil reservoir (Mariethoz and Caers [2014], part III, West Cost of Africa reservoir). Since the layer is a continuous body with different trends in different zones, the local anisotropy allows to capture all the geometrical and geological complexities of the domain. Extremely large 3D discrete domains must be analyzed and preprocessed in order to infer all field parameters in each domain node, making the oil reservoir scenario one of the most challenging in terms of computational resources invested. In order to visualize the limitations of classical methods, two simulated images can be observed in Figure 1.3, using LVA field and sampled data shown in Figure 1.2. The image in the left-side was generated using a classical method (sequential indicator

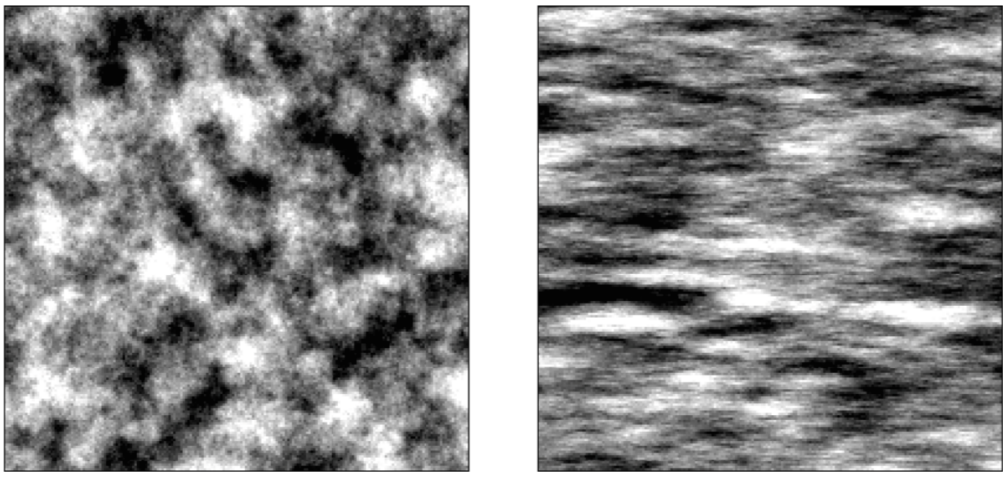

FIGURE 1.1: Comparison between an isotropic (left) and constant anisotropic (right) images. Extracted from Boisvert [2010].

simulation) and the image of the right-side using an LVA-based method. We can observe structural differences between both images, with clear structures following the LVA field in the right-side.

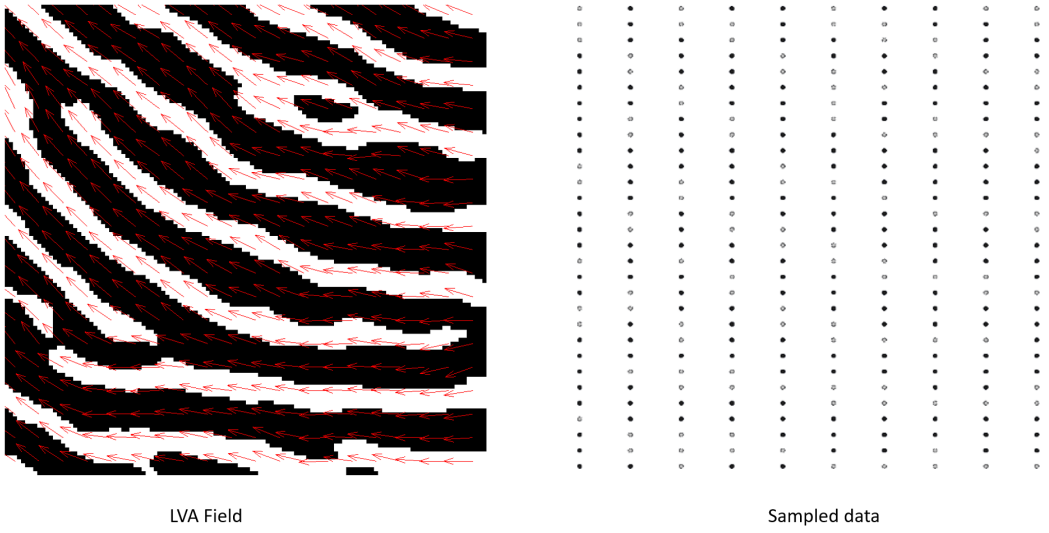

FIGURE 1.2: Example of locally varying anisotropy field (left) and sampled data (right) showing a cross section through an oil reservoir with several stratigraphic layers.

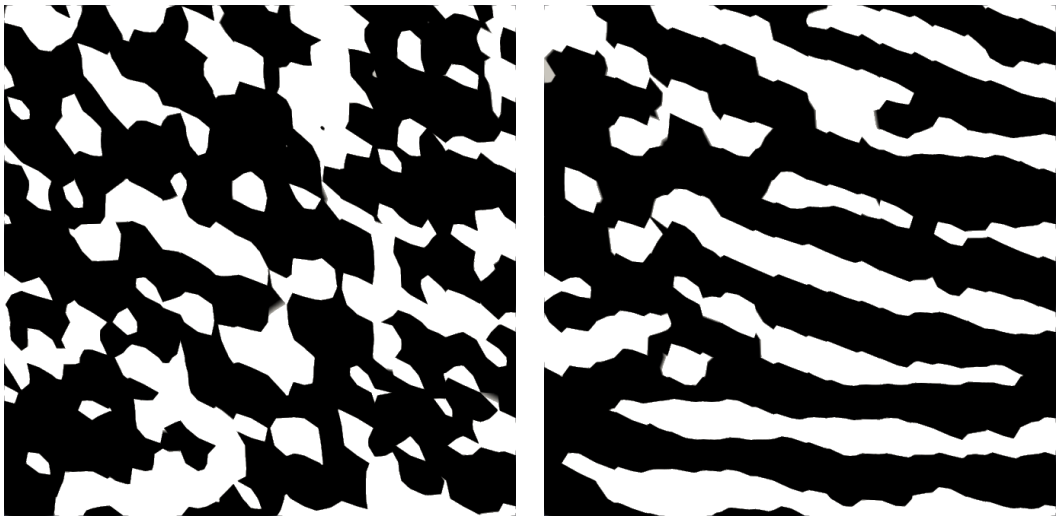

FIGURE 1.3: Comparison between an isotropic (left) and locally varying anositropic (left) images, using LVA field and sampled data from Figure 1.2.

The LVA geostatistical approach, described in detail in Boisvert [2010], sets the initial path for future developments in terms of numerical implementations that can potentially scale to large scenarios. However, to the best of the author's knowledge, no further analysis or public code improvements were developed since the initial article. This is in part because non-standard algorithms, acceleration and distribution techniques must be applied to the inner kernels of the proposed LVA codes for geostatistical analysis. These inner kernels can be clustered in two groups: the classical geostatistical inner kernels, such as variogram computing, kriging estimation and sequential simulation Chilès and Delfiner [1999], Deutsch and Journel [1998]; and dimensionality reduction techniques in high-dimensional spaces, such as PCA, MDS Mardia et al. [1979], ISOMAP Tenenbaum et al. [2000], locally linear embedding Roweis and Saul [2000], local tangent space alignment Zhang and Zha [2002], and others. Both groups are connected by the property of positive definite covariance functions Curriero [2006]. In order to use the classical methods in contexts where LVA is present, non-euclidean distances must be used instead of straight lines connecting different points in the domain. In Figure 1.4 we can observe a geological fold where two points A and B are connected through a non-linear path, where its length defines a non-euclidean distance. The non-linear path that connects both points is the shortest path that follows the underlying LVA field. By computing the non-euclidean distances between each pair of points in the domain of study, a distance matrix is obtained. As mentioned in Curriero [2006], classical geostatistical methods can be used only with euclidean distances to measure proximity, since positive-definiteness of a covariance matrix is guaranteed. For this reason, the non-euclidean distance matrix in $\mathbb{R}^d$ with $d \in \{2, 3\}$, must be transformed into a similar distance matrix generated in a higher-dimensional space $\mathbb{R}^d$ with $d >> 3$ using euclidean distances. Each step

in this process is highly intensive in compute and memory, since huge matrices must be stored and processed in order to obtain the high-dimensional embedding of points that produces a similar distance matrix. If large domains are being studied in $\mathbb{R}^d$ with $d \in \{2, 3\}$, a prohibitive amount of computational resources will be necessary to obtain a kriging estimation or a sequential simulation using an LVA field as proxy of the underlying anisotropy.

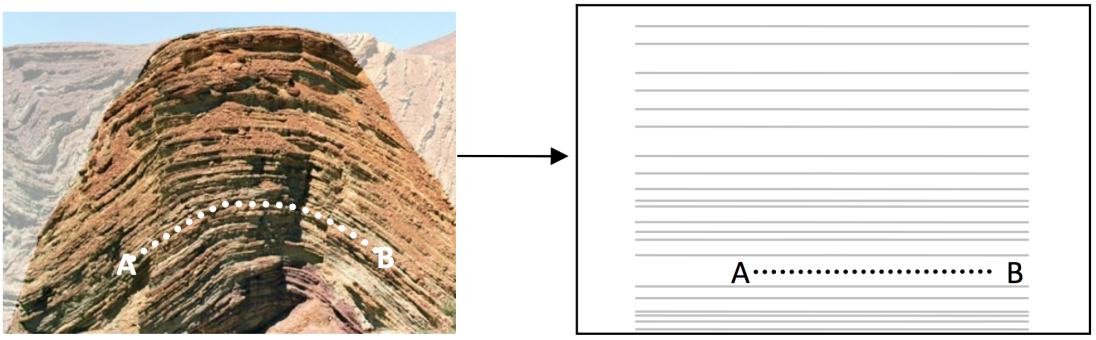

FIGURE 1.4: Left: Geological fold showing LVA with two points connected using a non-euclidean distance. Right: Unfolding the domain highlights the natural path between A and B Boisvert [2010]. Scale not available.

Regarding previous works related to accelerate large scale geostatistical applications, novel attempts in isotropic modeling have been reported in Vargas et al. [2007], Nunes and Almeida [2010], Peredo et al. [2015a] and Rasera et al. [2015], in order to accelerate classical methods using different algorithmic approaches combined with multi-core and distributed architectures, particularly MPI and OpenMP. In terms of other geostatistical methods that can handle LVA features of the phenomena, Multiple Point Statistics (MPS) Mariethoz and Caers [2014] has been one of the most active sub-fields in the last decade in terms of development of accelerated and distributed codes. Novel attempts have been reported in Mariethoz [2010], Straubhaar et al. [2011], Peredo and Ortiz [2011], Tahmasebi et al. [2012] and Peredo et al. [2014], among many other authors. Although MPS theoretically can generate models incorporating the LVA features of the phenomena, the amount of training data that this family of methods needs to accomplish that goal can be prohibitive in many applications, particularly mining resources and environmental modeling. The LVA approach can fulfill the needs of these kind of applications, but further development is needed in order to achieve larger scenarios. It is expected that many research opportunities arise in the topics of acceleration of classical algorithms and explorations in the new approaches related to LVA, particularly the manifold learning algorithms.

1.2 Contributions of this Thesis

The contributions of this thesis are related to accelerate and parallelize classical and LVA-based geostatistical simulation methods, particularly sequential simulation, which is one of the most common and computationally intensive methods in the field. This fact was recently remarked by some of the main authors in the field, which shows the relevance of this work today Gómez-Hernández and Srivastava [2021]. Two main parallel algorithms are presented, both applied to classical and LVA-based sequential simulation implementations. The first one is related to the parallel simulation of different domain points, after rearranging the order of simulation but preserving the exact results of a single-thread execution. The second algorithm is related to the parallel search of neighbour points in the domain, which will be used to build data dependencies for the parallel simulation of points. The main focus is on multi-core architectures using OpenMP and optimization techniques applied to Fortran and C++ codes.

Additionally, acceleration and parallelization techniques were also applied to auxiliary applications, such as variogram and shortest path calculation on hybrid CPU/GPU architectures using Fortran, C++ and CUDA codes.

The overall results of this work are a set of applications that will allow researchers and practitioners to accelerate dramatically the execution of their experiments and simulations. These tools can be combined with other geostatistical tools, in order to improve the existing landscape of open source codes that can be used in practical scenarions.

1.3 Related articles

Two research articles were published during this thesis, both related with parallel sequential simulations using classical and LVA-based codes:

- Oscar Peredo, Daniel Baeza, Julián M. Ortiz, and José R. Herrero. *A path-level exact parallelization strategy for sequential simulation.* Computers and Geosciences, 110:10-22, 2018. https://doi.org/10.1016/j.cageo.2017.09.011
- Oscar Peredo and José R. Herrero. *Acceleration Strategies for Large-Scale Sequential Simulations using Parallel Neighbour Search: non-LVA and LVA scenarios.* Computers and Geosciences, 160:105027, 2022. https://doi.org/10.1016/j.cageo.2021.105027

A third research article is currently under development. It extends the work presented in Chapter 7 of the thesis document. We plan to submit it to a proper journal shortly.

- Oscar Peredo and José R. Herrero. *Hybrid CUDA/OpenMP Acceleration of Semivariogram Computation.* To be submitted.

It is worth mentioning that the author contributed in different research articles in the past, before starting this thesis, in order to increase the overall knowledge in geostatistics in general:

- Oscar Peredo and Julián M. Ortiz. *Parallel implementation of simulated annealing to reproduce multiple-point statistics.* Computers and Geosciences, 37(8):1110-1121, 2011. https://doi.org/10.1016/j.cageo.2010.10.015
- Oscar Peredo, Julián M. Ortiz, José R. Herrero and Cristobal Samaniego. *Tuning and hybrid parallelization of a genetic-based multi-point statistics simulation code.* Parallel Computing 40(5-6):144-158, 2014. https://doi.org/10.1016/j.parco.2014.04.005
- Oscar Peredo, Julián M. Ortiz and José R. Herrero. *Acceleration of the Geostatistical Software Library (GSLIB) by code optimization and hybrid parallel programming.* Computers and Geosciences, 85(A):210-233, 2015. https://doi.org/10.1016/j.cageo.2015.09.016
- Oscar Peredo, Julián M. Ortiz and Oy Leuangthong. *Inverse modeling of moving average isotropic kernels for non-parametric three-dimensional gaussian simulation.* Mathematical Geosciences, 48(5):559-579, 2016. https://doi.org/10.1007/s11004-015-9606-x

Additionally, previous research articles related with LVA applications were published on conference proceedings of geosciences and geostatistics:

- Oscar Peredo, Felipe Navarro, Mauricio Garrido and Julián M. Ortiz. *Incorporating distributed dijkstra's algorithm into variogram calculation with locally varying anisotropy.* In 37th International Symposium APCOM 2015, pages 11621170. S. Bandopadhyay, 2015.
- Oscar Peredo, Mauricio Garrido, and Julián M. Ortiz. *Practical aspects of resources modeling in presence of locally varying anisotropy.* In 17th Annual Conference of the International Association for Mathematical Geosciences IAMG 2015. Schaeben, H., Tolosana-Delgado, R., van den Boogaart, K. G., van den Boogaart, R., 2015.

- Oscar Peredo, José A. García, Ricardo Stuven and Julián M Ortiz. *Urban dynamic estimation using mobile phone logs and locally varying anisotropy.*In Geostatistics Valencia 2016, pages 949964. Springer, 2017.

1.4 Organization

This document is organized as follows. Chapter 2 presents the theoretical background of classical and non-euclidean geostatistics, including implementation details of existing applications. It also presents the state of the art of accelerated and parallel implementations of sequential simulation algorithms. Chapter 3 presents the methodology used in this research, such as the main application library and source codes used for development, common parameters for execution, development techniques, case studies presented in different chapters and metrics used to compare results.

Chapter 4 presents the first contribution of this thesis, which is a parallel algorithm for sequential simulation algorithms. Results using classical sequential gaussian and sequential indicator simulation applications are presented. Chapter 5 presents the second contribution of this thesis, which is a parallel algorithm for neighbour search in sequential simulation methods. This algorithm is coupled with the parallel algorithm of Chapter 4, and it is implemented in classical and LVA-based applications.

In Chapter 6, two faster optimized libraries are used to accelerate other LVA specific routines. An additional parallel application is presented in Chapter 7, which is a hybrid CUDA/OpenMP implementation of the semivariogram computation from classical geostatistics. Finally, conclusions and future work are included in Chapter 8.

Chapter 2

Theoretical Background and State-of-the-Art

2.1 Overview

In this chapter we will review the theoretical background of classical and LVA-based geostatistics, in the context of quantification of uncertainty of unknown values in unsampled locations. As mentioned in the introductory chapter, uncertainty can be modelled as the response of several random variables geo-located in the unsampled locations, through their cumulative distribution functions and statistical properties using all the available information in the sampled locations.

Figure 2.1 depicts the general workflows on both classical (euclidean) and LVA-based (non-euclidean) geostatistics. Both workflows start with conditioning sampled data as input (if exists), but in LVA we have an additional input denoted LVA field. On this workflow, three additional steps are mandatory: non-euclidean distance matrix assembling, multidimensional scaling and triangulation. After these specific steps, both workflows can perform similar compute kernels, namely sequential simulation, kriging and variogram calculation.

In the next sections we will introduce mathematical details of these kernels, with special focus on sequential simulation algorithms for both workflows. These algorithms are the most challenging in terms of computational effort, allowing us to quantify uncertainty in a more realistic way.

Variogram calculation will be revisited in a subsequent chapter, since it can be used as a validation tool for simulated domains. Regarding the kriging compute kernel, there are several fast and parallel implementations that have been published in the last years Cheng [2013], Peredo et al. [2015a], so no further research was developed on this kernel.

However, its description is included in the next section since it is used as inner kernel by all sequential simulation algorithms.

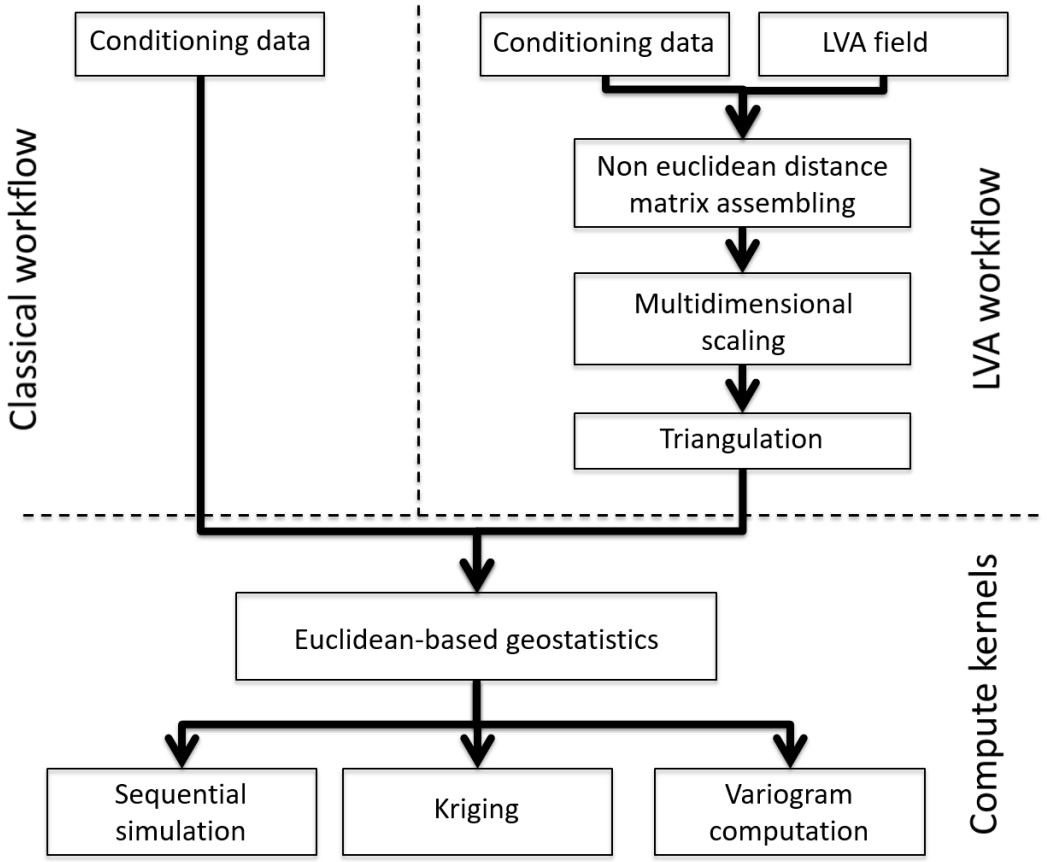

FIGURE 2.1: General workflows in both classical and LVA-based geostatistical methods.

2.2 Classical Geostatistics

Geostatistical simulation provides an approach to quantify uncertainty over spatially distributed variables. As mentioned in the introductory chapter, the classical geostatistical approach is based on euclidean distances to measure proximity between points in space. This aspect will be discussed in detail on Section 2.3. Additionally, well known methods on this track are depending on the properties of the random function considered, which can be continuous or categorical. Both flavors of methods are discussed in detail in the next section. Definitions are extracted from Deutsch and Journel [1998] for any further reference.

2.2.1 Background

2.2.1.1 Random variable

Any unsampled value on location $\mathbf{u} \in \mathbb{R}^3$ can be modelled as a random variable denoted $Z(\mathbf{u})$. These random variables allow to model petrophysical, geophysical, geographical and geological properties, among others. The cumulative distribution function (cdf) of a continuous random variable $Z(\mathbf{u})$ is denoted

$$F(\mathbf{u}; z) = \mathbb{P}\{Z(\mathbf{u}) \leq z\} \tag{2.1}$$

with $\mathbb{P}(A)$ the probability of event A.

Additionaly, when the cdf is made specific to a particular information set, for instance, the n closests neighbours with data values $Z(\mathbf{u}_\alpha) = z(\mathbf{u}_\alpha)$, $\alpha = 1, \ldots, n$, a conditional probability notation "$|(n)$" is used

$$\begin{aligned} F(\mathbf{u}; z|(n)) &= \mathbb{P}\{Z(\mathbf{u}) \leq z|(n)\} & (2.2)\\ &= \mathbb{P}\{Z(\mathbf{u}) \leq z|Z(\mathbf{u}_\alpha) = z(\mathbf{u}_\alpha), \alpha = 1, \ldots, n\} & (2.3) \end{aligned}$$

and the distribution is denoted as conditional cumulative distribution function (ccdf).

2.2.1.2 Random function

A random function $Z(\mathbf{u})$ is defined as a set of random variables defined over some field of interest, such as $\{Z(\mathbf{u}) : \mathbf{u} \in \Omega\}$ with $\Omega \subset \mathbb{R}^3$ a domain in study. As in random variables, a random function is characterized by its cumulative distribution function (cdf) denoted

$$F(\mathbf{u}_1, \ldots, \mathbf{u}_K; z_1, \ldots, z_K) = \mathbb{P}\{Z(\mathbf{u}_1) \leq z_1, \ldots, Z(\mathbf{u}_K) \leq z_K\} \tag{2.4}$$

which allows to model the uncertainty about the value z across unsampled locations of the domain Ω.

2.2.1.3 Stationarity hypothesis

A random function $\{Z(\mathbf{u}) : \mathbf{u} \in \Omega\}$ is said to be stationary within the domain Ω if its multivariate cdf of Equation (2.4) is invariant under traslations of the K coordinate vector $\mathbf{u}_k$, that is:

$$F(\mathbf{u}_1, \ldots, \mathbf{u}_K; z_1, \ldots, z_K) = F(\mathbf{u}_1 + \mathbf{h}, \ldots, \mathbf{u}_K + \mathbf{h}; z_1, \ldots, z_K) \tag{2.5}$$

This equation implies invariance also for lower order cdf, including univariate and bivariate cdfs, and invariance of all their moments, specifically all covariances as described in the next paragraph. In summary, the hypothesis of stationarity allows to infer the unique stationary cdf $F(z) = F(\mathbf{u}; z)$ directly from the cumulative sample histogram extracted from the sampled data at various locations of the domain Ω.

2.2.1.4 Covariance and semivariogram

The basic bivariate momentum is the covariance, extensively explained in Deutsch and Journel [1998] (II.1.1 and II.1.2), defined as

$$C(\mathbf{u}, \mathbf{u}') = \mathbb{E}(Z(\mathbf{u})Z(\mathbf{u}')) - \mathbb{E}(Z(\mathbf{u}))\mathbb{E}(Z(\mathbf{u}')) \tag{2.6}$$

By assuming the stationary hypothesis, the covariance between two points separated by $\mathbf{h}$ can be defined as

$$\begin{aligned} C(\mathbf{h}) &= C(\mathbf{u}, \mathbf{u}+\mathbf{h}) && (2.7)\\ &= \mathbb{E}(Z(\mathbf{u})Z(\mathbf{u}+\mathbf{h})) - \mathbb{E}(Z(\mathbf{u}))^2 && (2.8) \end{aligned}$$

Related with the stationary covariance, another bivariate statistic is the semivariogram, defined as

$$\gamma(\mathbf{h}) = C(\mathbf{0}) - C(\mathbf{h}) \tag{2.9}$$

where $C(\mathbf{0})$ is the stationary variance and $C(\mathbf{h})$ is the stationary covariance defined before.

Some well-known semivariogram models are:

- Spherical: defined by *range* a and *sill* $c > 0$

$$\gamma(\mathbf{h}) = \begin{cases} c\left[1.5\frac{|\mathbf{h}|}{a} - 0.5\left(\frac{|\mathbf{h}|}{a}\right)^3\right] & |\mathbf{h}| \leq a \\ c & |\mathbf{h}| > a \end{cases} \tag{2.10}$$

- Exponential: defined by *effective range* a (distance at which $\gamma(a) = 0.95c$) and *sill* $c > 0$

$$\gamma(\mathbf{h}) = c\left[1 - exp\left(-\frac{3|\mathbf{h}|}{a}\right)\right] \tag{2.11}$$

- Gaussian: defined by *effective range* a and *sill* $c > 0$

$$\gamma(\mathbf{h}) = c\left[1 - exp\left(-\frac{(3|\mathbf{h}|)^2}{a^2}\right)\right] \tag{2.12}$$

- Power model: defined by a power $0 < \omega < 2$ and a positive slope c

$$\gamma(\mathbf{h}) = c|\mathbf{h}|^{\omega} \tag{2.13}$$

When multiple effects or "structures" are combined into one semivariogram model, a nested model can be represented as a sum of multiple semivariogram models:

$$\gamma(\mathbf{h}) = \sum_{i=1}^{K} \gamma_i(\mathbf{h}) \tag{2.14}$$

2.2.1.5 Kriging

The kriging algorithm [Chilès and Delfiner, 1999, Deutsch and Journel, 1998, Journel and Alabert, 1989] was introduced to provide minimum error-variance estimates of unsampled locations using available data. The traditional application of kriging is to provide a regular grid of estimates that acts as a low-pass filter that tends to smooth out details and extreme values of the original dataset. It is extensively explained in Deutsch and Journel [1998] (IV.1). Simple Kriging (SK), in its stationary version, aims to obtain a linear regression estimator $Z^*_{SK}(\mathbf{u})$ (Figure 2.2) defined as

$$Z^*_{SK}(\mathbf{u}) = \sum_{\alpha=1}^{n} \lambda_\alpha(\mathbf{u}) Z(\mathbf{u}_\alpha) + \left[1 - \sum_{\alpha=1}^{n} \lambda_\alpha(\mathbf{u})\right] m \tag{2.15}$$

where $Z(\mathbf{u})$ is the random variable which depends on location $\mathbf{u} \in \mathbb{R}^3$, $m = \mathbb{E}\{Z(\mathbf{u})\}, \forall u$ is the location-independent (constant) expected value of $Z(\mathbf{u})$ and $\lambda_\alpha(\mathbf{u})$ are weights given by the solution of the system of normal equations:

$$\sum_{\beta=1}^{n} \lambda_\beta(\mathbf{u}) C(\mathbf{u}_\beta, \mathbf{u}_\alpha) = C(\mathbf{u}, \mathbf{u}_\alpha), \ \alpha = 1, \ldots, n \tag{2.16}$$

with $\{C(\mathbf{u}_\alpha, \mathbf{u}_\beta)\}_{\alpha,\beta=0,1,\ldots,n}$ the covariance matrix adding the sampled data $\mathbf{u}_0 = \mathbf{u}$.

With the stationary assumption, the covariance can be expressed as a function $C(\mathbf{h}) = C(\mathbf{u}, \mathbf{u} + \mathbf{h})$. Ordinary Kriging (OK) filters the mean m from the SK estimator of eq. (2.15), imposing that the sum of weights is equal to one. The resulting OK estimator is as

$$Z^*_{OK}(\mathbf{u}) = \sum_{\alpha=1}^{n} \lambda^{(OK)}_\alpha(\mathbf{u}) Z(\mathbf{u}_\alpha) \tag{2.17}$$

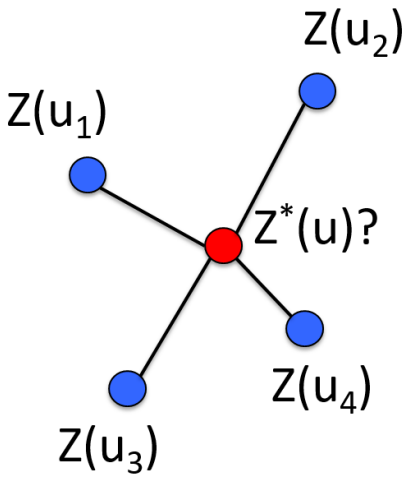

FIGURE 2.2: General schema for kriging estimation.

with the weights $\lambda_\alpha^{(OK)}$ solution of the extended system:

$$\sum_{\beta=1}^{n} \lambda_\beta^{(OK)}(\mathbf{u}) C(\mathbf{u}_\beta - \mathbf{u}_\alpha) + \mu(\mathbf{u}) = C(\mathbf{u} - \mathbf{u}_\alpha),\ \alpha = 1, \ldots, n \tag{2.18}$$

$$\sum_{\beta=1}^{n} \lambda_\beta^{(OK)} = 1 \tag{2.19}$$

with $\mu(\mathbf{u})$ a Lagrange parameter associated with the second constraint (2.19). It can be shown that OK amounts to re-estimating, at each new location $\mathbf{u}$, the mean m as used in the SK expression. Thus the OK estimator $Z_{OK}^*(\mathbf{u})$ is, in fact, a simple kriging estimator where the constant mean value m is replaced by the location-dependent estimate $m^*(\mathbf{u})$ (non-stationary algorithm with varying mean and constant covariance).

Initially, uncertainty quantification was addressed using kriging on any unsampled location of the domain in study. OK or SK deliver the best linear unbiased estimate on $\mathbf{u}$ (see Chapter 4 of Deutsch and Journel [1998] for a detailed description), $Z^*(\mathbf{u})$, which means that:

$$\mathbb{E}\{Z^*(\mathbf{u})\} = \mathbb{E}\{Z(\mathbf{u})\} \tag{2.20}$$

$$Var\{Z^*(\mathbf{u})\} = Var\{Z(\mathbf{u})\} - \sigma_*^2(\mathbf{u}) \tag{2.21}$$

with $\sigma_*^2(\mathbf{u})$ known as kriging variance:

$$\sigma_*^2(\mathbf{u}) = C(\mathbf{0}) - \sum_{\alpha=1}^{n} \lambda_\alpha(\mathbf{u}) C(\mathbf{u}, \mathbf{u}_\alpha) \geq 0 \tag{2.22}$$

The estimate value and variance can be used to define Gaussian-type confidence intervals, such as

$$\mathbb{P}\{Z(\mathbf{u}) \in \left[z^*(\mathbf{u}) - \sigma_*^2(\mathbf{u}), z^*(\mathbf{u}) + \sigma_*^2(\mathbf{u})\right]\} \approx 0.95 \tag{2.23}$$

Equation (2.20) shows that no bias is been added by the estimator $Z^*(\mathbf{u})$ and equation (2.21) shows that the positive factor $\sigma_*^2(\mathbf{u})$ reduces the value of the estimate variance. This variance reduction indicates that the kriging method *smooths* the random function, delivering unaccurate under/over estimations locally. The main reason of this behaviour is the underlying nature of the kriging variance, which is independent of the sampled data values (eq. (2.22)) on any unsampled location, and therefore tends to preserve the smoothness across the domain.

2.2.1.6 Sequential Gaussian Simulation

In order to improve the uncertainty quantification process, the conditional cumulative distribution function from eq. (2.3) should be infered, with additional steps applied to the kriging method. Specifically, two approaches for ccdf inference can be taken: Multivariate Gaussian or Indicator Kriging approach. Both approaches have specific and common features, depending on the nature of the random variables and their cdfs.

The Multivariate Gaussian approach assumes that every random variable $Z(\mathbf{u})$ is continuous and the sample data histogram is normal, which implies that the random function $\{Z(\mathbf{u}) : \mathbf{u} \in \Omega\}$ follows a gaussian distribution across all locations of the domain. If this is not the case, a normal score transformation can be applied on the original z data (see Section V.2.1 from Deutsch and Journel [1998]). With this transformation, the new random function $\{Y(\mathbf{u}) : \mathbf{u} \in \Omega\}$ can be used instead of Z.

Sequential Gaussian Simulation method [Alabert, 1987, Isaaks, 1990] takes the Multivariate Gaussian approach and combines it with a stochastic simulation process. In this stochastic method, equally probable joint realizations of the random variables defined by the random function $\{Z(\mathbf{u}) : \mathbf{u} \in \Omega\}$ are "drawed" or generated. At this point, typically a gridded domain is considered, and L realizations are represented as

$$\{z^{(l)}(\mathbf{u}) : \mathbf{u} \in \Omega\},\ l = 1, \ldots, L \tag{2.24}$$

The key aspect of the method is to allow previously simulated locations to be included as conditoning sampled data in future inferences of the ccdf for unsampled locations. The previous aspect assumes that there is a "path" in which the locations are being simulated and posteriorly can be considered as conditioning sampled data for subsequent locations. This path is commonly known as the random path, which should be obtained by reordering at random the one-dimensional natural indexation of the gridded locations

(each location should be visited once, see Section 2.3.2.2 for details of this indexation). On each visited location, a specified number of nearby neighboring conditioning sampled data points is collected (originally sampled data and previously simulated data). After that, a Simple Kriging estimation is computed, delivering mean and variance of $Y^*(\mathbf{u})$ (eqns. (2.15) and (2.22)). With this estimated parameters, the ccdf (eqn. 2.3) can be infered in order to draw a simulated value $y^{(l)}(\mathbf{u})$, with $l = 1, \ldots, L$. The final step is to back-transform the simulated values $y^{(l)}(\mathbf{u})$ using the inverse normal score transformation, obtaining the values $z^{(l)}(\mathbf{u})$.

Differences between a kriging estimation and sequential gaussian simulation can be observed in Figure 2.3. Each image was generated using the same conditional sampled data, which corresponds to 2376 real 3D samples of copper grades from the Chilean copper deposit Los Bronces Serrano et al. [1998]. Both images are plotted using the same scale, and both have a diagonal line from bottom-left to top-right corner, where their values were plotted through it. On the bottom plot we can observe the differences of each value, were kriging estimate (red) shows the "smoothness" described previously, and the sequential gaussian simulation (green) shows a higher spatial variability of its value. The simulated stochastic variability allows to quantify uncertainty on each unsampled location, through the simulation of several images.

2.2.1.7 Sequential Indicator Simulation

An alternative approach to multivariate Gaussian is the Indicator Kriging approach. As stated in Journel and Huijbregts [1978], the estimators that have been considered until now are linear combinations of the available data, but we can go beyond this definition if additional information is available. In order to expand the definition, mathematical functional analysis concepts are needed. Particularly, the concepts of conditional expectation and indicator functions are relevant in order to describe Indicator Kriging.

Conditional expectation of a random variable $Z(\mathbf{u})$ on sampled data (n) (same as in Equation 2.3) is denoted

$$\mathbb{E}\{Z(\mathbf{u})|(n)\} \in \mathcal{H} \tag{2.25}$$

where $\mathcal{H}$ is a Hilbert space over all random variables defined over the same domain with the following scalar product $\langle X, Y \rangle = \mathbb{E}\{XY\}$ and norm $\|X\| = \sqrt{\mathbb{E}\{X^2\}}$.

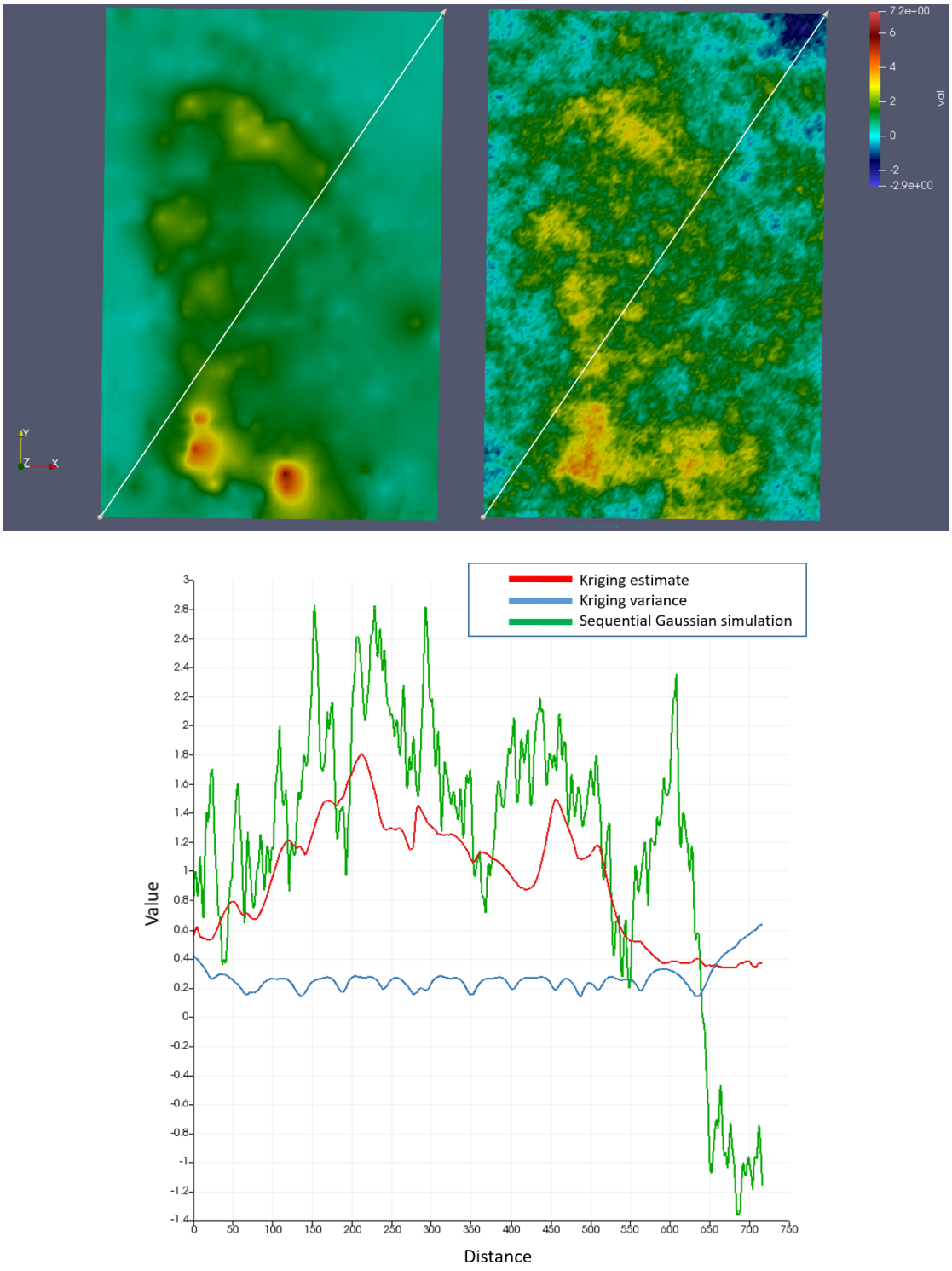

FIGURE 2.3: Comparison between kriging estimation (top-left) and sequential gaussian simulation (top-right). Values generated by each method through a diagonal line (bottom).

Indicator functions are random variables $I(u)$ that can be defined from K mutually exclusive categories s_k, $k = 1, \ldots, K$. Each location $\mathbf{u}$ should belong to one and only one of those K categories. In this case, the indicator of class s_k on location $\mathbf{u}$ is denoted

as

$$i(\mathbf{u}; s_k) = \begin{cases} 1 & \mathbf{u} \in s_k \\ 0 & \mathbf{u} \notin s_k \end{cases} \tag{2.26}$$

In case of a continuous variable $z(\mathbf{u})$, a discretization in K mutually exclusive classes $s_k = (z_{k-1}, z_k]$, $k = 1, \ldots, K$ can be interpreted in the same way as $i(\mathbf{u}; s_k)$.

With both concepts at hand we can introduce two important results. We will assume $K = 2$ for the sake of simplicity in the equations. If the variable to be simulated is already categorical (binary with $K = 2$), namely $i(\mathbf{u})$, then

$$\mathbb{P}\{I(\mathbf{u}) = 1|(n)\} = \mathbb{E}\{I(\mathbf{u})|(n)\} \tag{2.27}$$

On the other hand, if the variable to be simulated is continuous, namely $z(\mathbf{u})$, its ccdf can be written in terms of the corresponding indicator function as

$$\mathbb{P}\{Z(\mathbf{u}) \leq z|(n)\} = \mathbb{E}\{I(\mathbf{u}; z)|(n)\} \tag{2.28}$$

with $I(u; z) = 1$ if $Z(\mathbf{u}) \leq z$ and 0 otherwise. Equations (2.27) and (2.28) show a fundamental relation that allow to infer the ccdf of Equation (2.3) by estimating the conditional expectation. Simple Kriging estimator (eq. (2.15)) of $I(\mathbf{u})$ can be written as

$$I^*_{SK}(\mathbf{u}; z) = \sum_{\alpha=1}^{n} \lambda_\alpha(\mathbf{u}; \mathbf{z}) I(\mathbf{u}_\alpha; z) + \left[1 - \sum_{\alpha=1}^{n} \lambda_\alpha(\mathbf{u}; z)\right] \mathbb{E}\{I(\mathbf{u}; z)\} \tag{2.29}$$

By identifying that

$$\begin{aligned} \mathbb{E}\{I(\mathbf{u}; z)\} &= 1 \cdot \mathbb{P}\{Z(\mathbf{u}) \leq z\} + 0 \cdot \mathbb{P}\{Z(\mathbf{u}) > z\} && (2.30) \\ &= \mathbb{P}\{Z(\mathbf{u}) \leq z\} && (2.31) \end{aligned}$$

equation (2.29) can be written as

$$I^*_{SK}(\mathbf{u}; z) = \sum_{\alpha=1}^{n} \lambda_\alpha(\mathbf{u}; \mathbf{z}) I(\mathbf{u}_\alpha; z) + \left[1 - \sum_{\alpha=1}^{n} \lambda_\alpha(\mathbf{u}; z)\right] \mathbb{P}\{Z(\mathbf{u}) \leq z\} \tag{2.32}$$

Weights $\lambda_\alpha(\mathbf{u}; z)$ are computed by solving the system of normal equations of Equation (2.16) using $C(\mathbf{u}_\beta, \mathbf{u}_\alpha; z)$ instead of $C(\mathbf{u}_\beta, \mathbf{u}_\alpha)$, which corresponds to the indicator covariance at cut-off z. Using K categories or classes, requires to calculate K indicator covariances $C(\mathbf{u}_\beta, \mathbf{u}_\alpha; s_k)$ and cdf values $\mathbb{P}\{Z(\mathbf{u}) \leq s_k\}$. Ordinary Kriging scenario follows the same reasoning, but using equations (2.17), (2.18) and (2.19) instead.

Similarly to sequential gaussian simulation, sequential indicator simulation method [Alabert, 1987] takes Indicator Kriging approach and combines it with a stochastic simulation process. It also applies the key aspect described in Section 2.2.1.6, in order to allow previously simulated locations to be included as conditioning data in future inferences of the ccdf for unsampled locations. This is valid since Indicator Kriging equations are still true when conditional data is used, as consequence of Equation (2.28).

An image generated with a sequential gaussian simulation method can be observed in Figure 2.4. It was generated using 2376 real 3D samples of rock lithology from the Chilean copper deposit Los Bronces Serrano et al. [1998]. The rock types in these samples correspond to Granodiorite (15%, category 1), Diorite (69%, category 2) and Breccia (16%, category 3). It contains a diagonal line from bottom-left to top-right corner, where their category value were plotted through it. On the bottom plot we can observe the three categories values. In this case, the non-linear nature of the simulated variable allows to quantify uncertainty for additional features on each unsampled location, through the simulation of several images.

In terms of usability, the case of categorical variables is particularly suited for high variability deposits where transitions between facies show low correlation. Alternative methods based on truncation of Gaussian random fields, namely Truncated Gaussian and PluriGaussian simulation [Matheron et al., 1987], offer more flexibility to reproduce these transitions Deutsch [2006], but are not as flexible when dealing with secondary variables and trends [Yarus et al., 2012]. The method has been applied to the different areas in Geosciences, such as geological modelling of ore deposits [de Souza and Costa, 2013, Dimitrakopoulos, 1998, Dimitrakopoulos and Dagbert, 1993, Journel and Isaaks, 1984] and oil reservoirs [de Almeida, 2010, dell'Arciprete et al., 2012, Dubrule and Damsleth, 2001, Pan, 1997], as well as in other fields such as rock fractures modelling [Dowd et al., 2007], imaging [van der Meer, 1994], and soil science [Bierkens and Burrough, 1993, Goovaerts, 2001], to name a few.

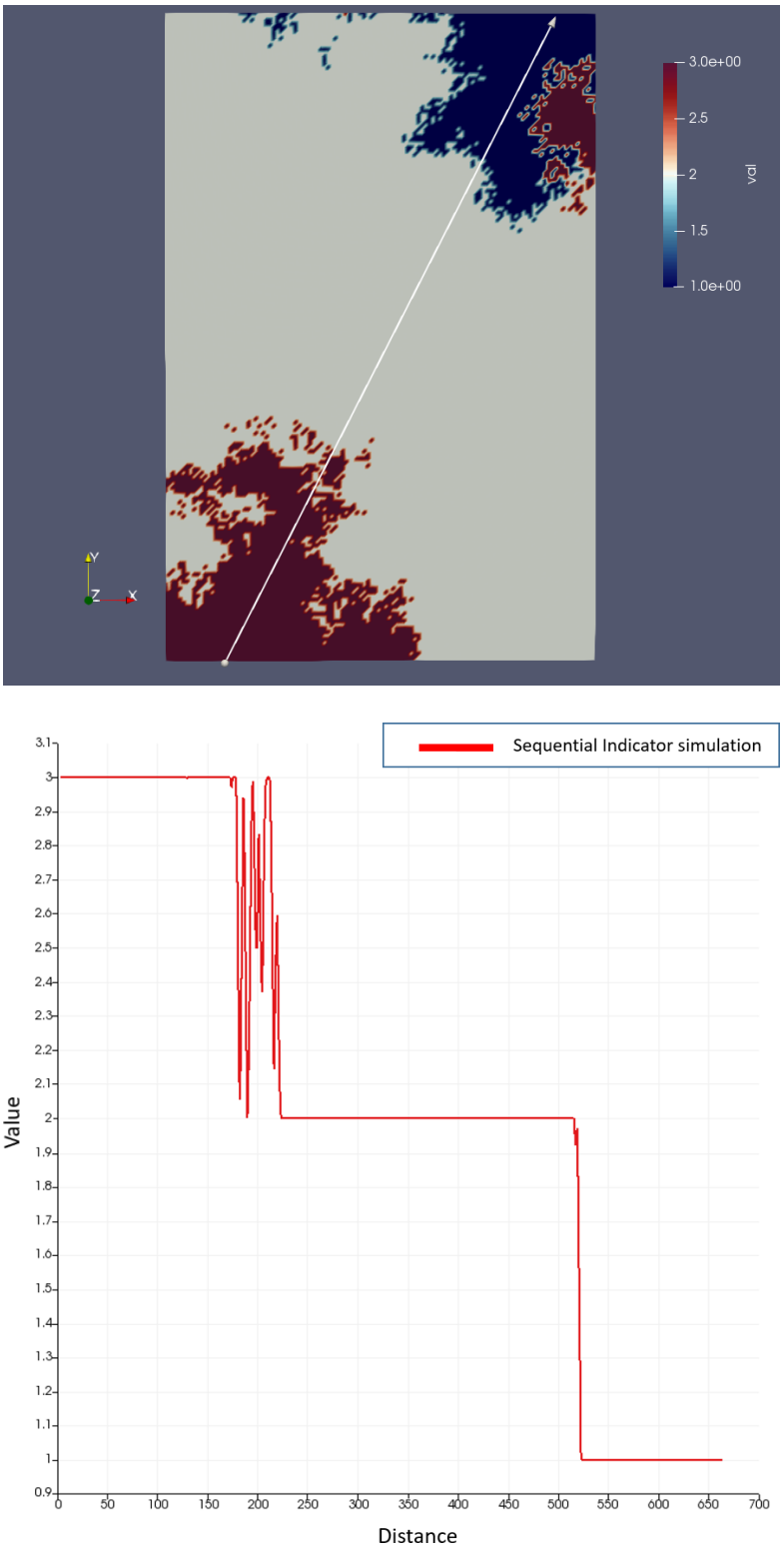

FIGURE 2.4: Comparison between kriging estimation (top-left) and sequential gaussian simulation (top-right). Values generated by each method through a diagonal line (bottom).

2.2.2 Sequential implementations

In this thesis, the main sequential (single-threaded) implementation studied and analyzed is the Geostatistical Software Library (GSLIB), originally presented by Deutsch and Journel Deutsch and Journel [1998]. Some parts of this section are extracted from Peredo et al. [2015a], which was published by the same author of this thesis. It has been used in the geostatistical community for more than thirty years, and is one of the most used still by academics and practioners in the field. It contains plotting utilities, data transformation utilities, measures for spatial continuity (variograms), kriging estimation and stochastic simulation applications. Among these components, estimation and simulation are two of the most used, and can be executed with large data sets and estimation/simulation grids. Large scenarios require several minutes/hours of elapsed time to finish, due to the heavy computations involved and its sequential implementation. Since their original development, these routines have helped many researchers and practitioners in their studies, mainly due to the accuracy and performance delivered by this package. Many efforts have been proposed to accelerate or enhance the scope of the original package, WinGslib Statios LLC [2001], SGeMS Remy et al. [2009], HPGL Savichev et al. and Pyrcz et al. being the most relevant efforts. SGeMS and HPGL moves away from Fortran and implements Python and C/C++ code in conventional and new algorithms. GeostatsPy mixes Python re-implementations of GSLIB routines with wrappers to Fortran pre-compiled code. Although there is a significant gain with these changes, for many practitioners and researchers, the simplicity of Fortran code and the availability of an extensive pool of modified GSLIB-based programs makes it hard to abandon this library.

According to GSLIB documentation Deutsch and Journel [1998], the software package is composed by a set of utility routines, compiled and wrapped as a static library named `gslib.a`, and a set of applications that call some of the wrapped routines. We will refer to these two sets as *utilities* and *applications*. Typically, a main program and two subroutines compose an *application*. The first subroutine is in charge of reading the parameters from the input files, and the second subroutine executes the main computation and writes out the results using predefined output formats. Additionally, two structures of static and dynamic variables are used by the main program and each subroutine: an include file and a `geostat` module. The include file contains static variable declarations, like constant parameters, fixed length arrays and common blocks of variables. The `geostat` module contains dynamic array declarations, which will be allocated in some of the subroutines with the allocate instruction. A utility is self-contained allowing sharing variables with other utilities and applications through common block variable declarations.

In the next paragraphs, we will review the three main applications related with this thesis, `gamv`, `sgsim` and `sisim`.

2.2.2.1 `gamv`

The `gamv` application calculates several spatial variability/continuity measures of a variable, in an experimental way, using the available dataset as source. Available measures to be calculated are: semivariogram, cross-semivariogram, covariance, correlogram, general relative semivariogram, pairwise relative semivariogram, semivariogram of logarithms, semimadogram and indicator semivariogram. The description of each measure can be found in Deutsch and Journel [1998] (III.1). Among the most used, we can mention the experimental semivariogram (theoretically defined in eq. (2.9), which is defined as

$$\gamma(\mathbf{h}) = \frac{1}{2N(\mathbf{h})} \sum_{i=1}^{N(\mathbf{h})} (Z(\mathbf{u}_i) - Z(\mathbf{u}_i + \mathbf{h}))^2 \tag{2.33}$$

where $\mathbf{h}$ is the separation vector, $N(\mathbf{h})$ is the number of pairs separated by $\mathbf{h}$ (with certain tolerance), $Z(\mathbf{u}_i)$ is the value at the start of the vector (tail) and $Z(\mathbf{u}_i + \mathbf{h})$ is the corresponding end (head). In Algorithm 19 we can see the main steps of the algorithm implemented in `gamv` application. We can observe that the steps in this algorithm are essentially the same regardless the measure to be calculated. For example, to calculate the semivariogram (eq. 7.1) using just one variable, one direction $\mathbf{h}$ and 10 lags with separation $h = 1.0$, first we iterate through all pairs of points in the domain Ω (loops of lines 2 and 3 of Algorithm 19), then we have ten iterations in the next loops (line 4 with $ndir = 1$, $nvarg = 1$ and $nlag = 10$). In each iteration, we must check if some geometrical tolerances are satisfied by the current pair of points (first condition of line 6) and then we must check if the separation vector between the points, $(\mathbf{p}_i - \mathbf{p}_j)$, is similar to the separation vector $\mathbf{h}$ multiplied by the current lag $ilag$ and the separation lag ($h = 1.0$). If both conditions are fulfilled, the pseudo-routine `save_statistics` saves the values of the variables in study into array $\boldsymbol{\beta}$. In this case, only one variable is being queried ($h_{iv} == t_{iv}$). For each type of variogram, the variables $\mathbf{V}_{i,h_{iv}}$ and $\mathbf{V}_{j,h_{iv}}$ may be transformed using different algebraic expressions. In the case of the semivariogram, we must save $(\mathbf{V}_{i,h_{iv}} - \mathbf{V}_{j,h_{iv}})^2$. Finally, using the statistics stored in $\boldsymbol{\beta}$, the pseudo-routine `build_variogram` saves the final variogram values in vector $\boldsymbol{\gamma}$, which is stored in file `output.txt`.

Input:
$(\mathbf{V}, \Omega)$: sample data base values $\mathbf{V} \in \mathbb{R}^{|\Omega| \times m}$ defined in a 3D domain Ω;
$nvar$: number of variables in study ($nvar \leq m$);
$nlag$: number of lags;
h: lag separation distance;
$ndir$: number of directions;
$\mathbf{h}_1, \ldots, \mathbf{h}_{ndir}$: directions;
$\tau_1, \ldots, \tau_{ndir}$: geometrical tolerance parameters (azimuth and dip);
$nvarg$: number of variograms;
$(type_1, t_1, h_1), \ldots, (type_{nvarg}, t_{nvarg}, h_{nvarg})$: variogram types, tail and head variables;

1 $\boldsymbol{\beta} \leftarrow$ `zeros`$(nvar \times nlag \times ndir \times nvarg)$
2 **for** $i \in \{1, \ldots, |\Omega|\}$ **do**
3 **for** $j \in \{i, \ldots, |\Omega|\}$ **do**
4 **for** $(id, iv, il) \in \{1, \ldots, ndir\} \times \{1, \ldots, nvarg\} \times \{1, \ldots, nlag\}$ **do**
5 $(\mathbf{p}_i, \mathbf{p}_j) \leftarrow ((x_i, y_i, z_i), (x_j, y_j, z_j)) \in \Omega \times \Omega$
6 **if** $(\mathbf{p}_i, \mathbf{p}_j)$ *satisfy tolerances* $\tau_{id} \wedge ||(\mathbf{p}_i - \mathbf{p}_j) - \mathbf{h}_{id} \times il \times h|| \approx 0$ **then**
7 $\boldsymbol{\beta} \leftarrow$`save_statistics`$(\mathbf{V}_{i,h_{iv}}, \mathbf{V}_{i,t_{iv}}, \mathbf{V}_{j,h_{iv}}, \mathbf{V}_{j,t_{iv}}, type_{iv})$
8 **end**
9 **end**
10 **end**
11 **end**
12 $\boldsymbol{\gamma} \leftarrow$ `build_variogram`$(\boldsymbol{\beta})$
13 `write`(`output.txt`,$\boldsymbol{\gamma}$)

Output: Output file with $\boldsymbol{\gamma}$ values

Algorithm 1: Pseudo-code of `gamv`, measurement of spatial variability/continuity (single-thread algorithm)

2.2.2.2 `sgsim`

The application `sgsim` implements the sequential gaussian simulation algorithm, as described in Deutsch and Journel [1998] (V.2.3). It is considered as the most straightforward algorithm for generating realizations of a multivariate Gaussian field. Its main steps can be viewed in Algorithm 2. The first step is to transform the original dataset into a standard normally distributed dataset (line 1). Then a random path must be generated, visiting all nodes of the simulation grid, for each simulation to be calculated (line 3). At each location $ixyz$, a ccdf must be estimated by simple or ordinary kriging (eqs. (2.15) and (2.16); (2.17), (2.18) and (2.19) using a maximum number of nearby neighbours), and then a random variable must be drawn from the generated ccdf (lines 7 and 8). The next step is to translate the simulated value in the normal distribution to the original distribution of the sample data (line 9). Finally, the back-transformed scalar result is stored in a file `output.txt` using a system call (`write`) for each nodal value (line 10).

Input:
$(\mathbf{V}, \Omega)$: sample data base values defined in a 3D domain;
γ: structural variographic models;
κ: kriging parameters (radius, max number of neigbours and others);
τ: seed for pseudo-random number generator;
N: number of generated simulations;
`output.txt`: output file;

1 $\mathbf{Y} \leftarrow$ `normal_score`$(\mathbf{V})$
2 **for** $isim \in \{1, \ldots, N\}$ **do**
3 $\quad \mathcal{P} \leftarrow$ `create_random_path`(Ω, τ)
4 $\quad \mathbf{Y}^{tmp} \leftarrow$ `zeros`$(\mathbf{Y})$
5 $\quad \mathbf{Y}^{tmp} \leftarrow$ `assign`$(\mathbf{Y})$ *//Sample data assignment*
6 $\quad$ **for** $ixyz \in \{1, \ldots, |\Omega|\}$ **do**
7 $\quad\quad$ `index` $\leftarrow \mathcal{P}_{ixyz}$
8 $\quad\quad$ **LocalNeighbours** $\leftarrow$ `search_neighbours`$(\texttt{index}, \kappa)$
9 $\quad\quad$ $p \leftarrow$ `kriging`$(\texttt{index}, \textbf{LocalNeighbours}, \gamma, \kappa)$
10 $\quad\quad$ $\mathbf{Y}^{tmp}_{\texttt{index}} \leftarrow$ `simulate`$(\texttt{index}, p, \tau)$
11 $\quad\quad$ $\mathbf{V}^{tmp} \leftarrow$ `back_transform`$(\mathbf{Y}^{tmp}_{\texttt{index}})$
12 $\quad\quad$ `write`$(\texttt{output.txt}, \mathbf{V}^{tmp}_{\texttt{index}})$
13 $\quad$ **end**
14 **end**

Output: N stochastic simulations stored in file `output.txt`

Algorithm 2: Pseudo-code of `sgsim`, sequential gaussian simulation program (single-thread algorithm)

2.2.2.3 `sisim`

The application `sisim` implements the sequential indicator simulation algorithm, as described in Deutsch and Journel [1998] (V.3.1). Its main steps can be viewed in Algorithm 3. The main steps are analogous to the sequential gaussian simulation algorithm, implemented by the `sgsim` application. The differences with Algorithm 2 are the addition of a category-based loop (line 6) and the usage of the pseudo-routine `indicator_kriging` instead of the tradidional `kriging` (line 7). This pseudo-routine calculates the Indicator Kriging estimator described in eqn. (2.32) for simple or ordinary kriging versions.

Input:
$(\mathbf{V}, \Omega)$: sample data base values defined in a 3D domain;
C: number of categories to be reproduced;
$\gamma_1, \ldots, \gamma_C$: structural variographic models;
κ: kriging parameters (radius, max number of neigbours and others);
τ: seed for pseudo-random number generator;
N: number of generated simulations;
`output.txt`: output file;

```
1  for isim ∈ {1, ..., N} do
2      P ← create_random_path(Ω, τ)
3      V^tmp ← zeros(V)
4      for ixyz ∈ {1, ..., |Ω|} do
5          index ← P_ixyz
6          LocalNeighbours ← search_neighbours(index, κ)
7          for icut ∈ {1, ..., C} do
8              p_icut ← indicator_kriging(index, LocalNeighbours, γ_icut, κ)
9          end
10         V^tmp_index ← simulate(index, p_1, ..., p_C, τ)
11     end
12     write(output.txt, V^tmp)
13 end
```

Output: N stochastic simulations stored in file `output.txt`

Algorithm 3: Pseudo-code of `sisim`, sequential indicator simulation program (single-thread algorithm)

2.2.3 Parallel implementations

Regarding semivariogram parallel computation, loop parallelization techniques can be applied in the main loops that traverses all pairs of points (lines 2 and 3 on Algorithm 19). A fast parallel implementation of `gamv` application was presented in Peredo et al. [2015a], using OpenMP and MPI for load distribution in large scale scenarios. Additional accelerations using hybrid GPU/CPU are presented in this thesis on Chapter 7.

Regarding sequential simulation methods, novel attempts have been reported in Dimitrakopoulos and Luo [2004], Vargas et al. [2007], Nunes and Almeida [2010], Peredo et al. [2015a], Rasera et al. [2015] and Nussbaumer et al. [2018]. All of them using different algorithmic approaches combined with multi-core and distributed architectures, particularly MPI and OpenMP. A common parallelization framework for sequential simulation was proposed in Mariethoz [2010]. In this article, three main approaches are characterized: realization, path and node level.

Realization level parallelization refers to distribute the generation of different realizations or images over different threads of execution, either on a multi-core or distributed system (line 2 on Algorithms 2 and line 1 on 3). This is a coarse-grained approach, which is straighforward to implement since no costly modifications should be made in the inner kernels of the methods, only that outer loop that traverses the realization identifier should be adapted. An optimized parallel implementation using this approach was also presented in Peredo et al. [2015a].

Path level parallelization refers to distribute the simulation of points across the random path selected. Different groups of points can be simulated in parallel by different threads of execution, by applying domain decomposition or other grouping techniques. Initial proposals were presented by Dimitrakopoulos and Luo [2004] and Vargas et al. [2007], and later by Rasera et al. [2015], all of them with focus on sequential gaussian simulation. Nussbaumer et al. [2018] presented a recent work which combines realization and path level parallelization, also with focus on sequential gaussian simulation. This approach can be viewed as a fine-grained approach, which is considerably more difficult to implement. The difficulty comes from the intrinsic nature of the method, that imposes a sequential order of point simulation. If that order is changed in some way, approximate results will be obtained with the risk of not being honoring the spatial variability/continuity measures of the conditioning sampled data (unwanted artifact generation in the image). Additionally, both methods, gaussian and indicator simulation, intrinsically need an efficient method to find nearest neighbours for each location to be simulated, which imposes a challenge in the scalability of any parallel method proposed. On Chapters 4 and 5 of this thesis we propose two main techniques that allow to accelerate and parallize this approach on both methods.

Node level parallelization in the context of sequential simulation have not been explored extensively, up to the author's knowledge. The main reason is probably the lower scalability that this approach can deliver, since the only possibilities are to parallelize the inner routines for kriging (line 7 of Algorithm 2 and lines 6-8 of Algorithm 3). The reason is that the size of the kriging systems that should be solved are limited by the number of neighbours used to infer the ccdf, denoted (n) as the conditioning information. In practice, the value of n will never be considerably larger.

2.3 LVA-based Geostatistics

Classical geostatistics can be used in many applied cases where the values under study show isotropic or regular trends of preferential directions. However, in complex scenarios the results obtained can be unrealistic since the underlying phenomena shows highly anisotropic trends which can not be reproduced by the classical approach. These kind of complex scenarios arise in geological modeling of faults and veins in mineral reserves, sedimentary deposits in oil and gas reservoirs, environmental modeling of pollution spread, rain fall patterns or animal migration, and mobility patterns in highly populated urban areas.

In the last section, we learn that the classical geostatistical approach is based on the covariance and semivariogram to measure the spatial variability/continuity of the data. It is also based on kriging estimates, which are used by stochastic simulation methods

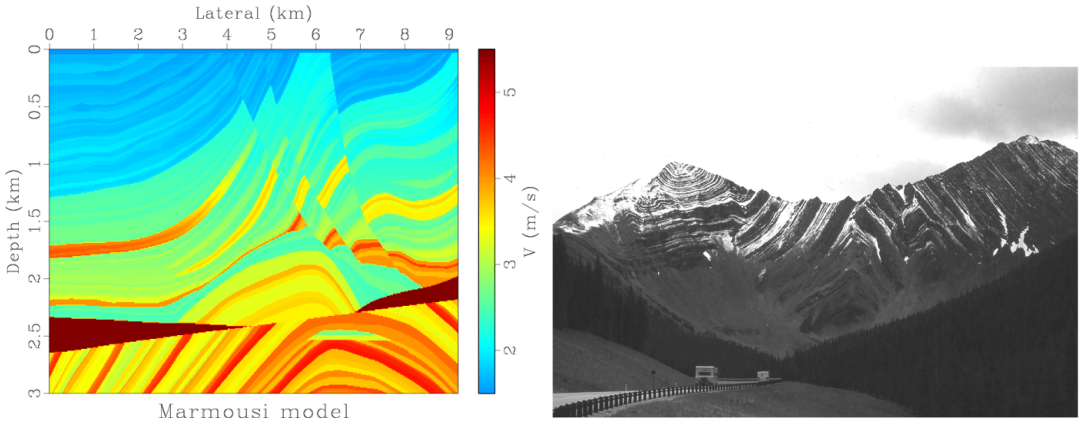

FIGURE 2.5: Two scenarios of locally varying anisotropy, with layer, faults and folds. Left: The Marmousie velocity model [Versteeg, 1994] for the subsurface of the Kwanza Basin, Angola; Right: Folds in Cretaceous strata in the footwall of the Lewis Thrust [Pollard and Fletcher, 2005], Canadian Rockies, Canada.

to generate realizations or images of unsampled locations. At the heart of all these computations resides the distance calculation that is the classical euclidean distance. It is used to measure the separation vector $\mathbf{h}$ in semivariogram functions and also to identify nearby neighbours around each location.

In this section, non-euclidean distance is introduced into the classical applications, with the aim of represent the underlying anisotropic properties of the phenomena. These properties are represented by the LVA field, which defines the non-euclidean distances between each pair of points. Some definitions are extracted from Boisvert [2010] for any further reference.

2.3.1 Background

A first property should be stated, regarding valid covariance functions $C(\mathbf{h})$ to be used. As stated by Matérn [1986], a covariance function is admissible (in the sense that can be used for geostatistical purposes) if and only if it is positive definite. Concretely, for locations $\mathbf{u}_1, \ldots, \mathbf{u}_n \in \Omega$ and $X = \sum_{i=1}^{n} w_i Z(\mathbf{u}_i)$, the variance of X should be positive:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} w_i C(\mathbf{u}_i, \mathbf{u}_j) w_j > 0 \tag{2.34}$$

for all possible choices of n and weights $w_1, \ldots, w_n$.

If the distance used by the covariance function is non-euclidean, eqn. (2.34) might not be valid. A counter example was presented by Curriero [2006], in which the domain is a simple four point regular grid in $\mathbb{R}^2$ with unit spacing, and the points are represented

by (x_i, y_i), $i = 1 \dots 4$. The non-euclidean distance is the city block metric,

$$\rho_{ij} = |x_i - x_j| + |y_i - y_j| \tag{2.35}$$

This yields the following matrix of inter-point city block distances

$$\begin{pmatrix} 0 & 1 & 1 & 2 \\ 1 & 0 & 2 & 1 \\ 1 & 2 & 0 & 1 \\ 2 & 1 & 1 & 0 \end{pmatrix}$$

which when used with the Gaussian covariance function $20 exp(\rho_{ij}^2/4)$, and nugget, sill, and range parameters arbitrarily set at $(0, 20, 4)$ respectively, results in the following proposed covariance matrix,

$$\begin{pmatrix} 20.00 & 15.58 & 15.58 & 7.36 \\ 15.58 & 20.00 & 7.36 & 15.58 \\ 15.58 & 7.36 & 20.00 & 15.58 \\ 7.36 & 15.58 & 15.58 & 20.00 \end{pmatrix}$$

The eigenvalues of this matrix are $(58.52, 12.64, 12.64, -3.80)$, implying the Gaussian covariance is no longer positive definite when used with the city block metric.

A workaround that can be developed to allow non-euclidean distances is based on transform the non-euclidean distance matrix in $\mathbb{R}^d$ with $d \in \{2, 3\}$, into a similar distance matrix generated in a higher-dimensional space $\mathbb{R}^d$ with $d \geq 3$ using euclidean distances. The basic concepts related with this transformation are defined below.

2.3.1.1 Anisotropic distance

The anisotropic distance between two points can be calculated using the following equation

$$d = \sqrt{\left(\frac{d_X}{a_X}\right)^2 + \left(\frac{d_Y}{a_Y}\right)^2 + \left(\frac{d_Z}{a_Z}\right)^2} \tag{2.36}$$

where d_i correspond to the euclidean distance across axis i and a_i is denoted as the range of anisotropy across axis i. A larger range in a particular direction effectively shortens the distance between points in that direction. Additionally, axis rotations can be applied, namely changes on the azimuth (or strike), dip or plunge angles. The azimuth consists in a rotation α about the Z axis (count-clockwise), the dip β is a rotation about the X axis (clockwise) and the plunge φ is a rotation about the Y axis (count-clockwise).

In Figure 2.6 we can observe two scenarios, without azimuth rotation ($\alpha = 0°$) and with azimuth rotation of $\alpha = 45°$. Three possible anisotropy distances can be computed:

- No anisotropy: $a_X = a_Y = a_Z = 1$ and azimuth $\alpha = 0°$. We can assume an euclidean distance between points A and B equal to 1:

$$d_{AB} = \sqrt{\left(\frac{cos(60°)}{1}\right)^2 + \left(\frac{sin(60°)}{1}\right)^2 + \left(\frac{0}{1}\right)^2} \tag{2.37}$$

$$= 1 \tag{2.38}$$

- Anisotropy in XY directions without rotation: $a_X = 4$, $a_Y = a_Z = 1$ and $\alpha = 0°$. Applying eqn. (2.36):

$$d_{AB} = \sqrt{\left(\frac{cos(60°)}{4}\right)^2 + \left(\frac{sin(60°)}{1}\right)^2 + \left(\frac{0}{1}\right)^2} \tag{2.39}$$

$$\approx 0.545 \tag{2.40}$$

- Anisotropy in XY directions with 45° azimuth rotation: $a_X = 4$, $a_Y = a_Z = 1$ and $\alpha = 45°$. Applying eqn. (2.36):

$$d_{AB} = \sqrt{\left(\frac{cos(15°)}{4}\right)^2 + \left(\frac{sin(15°)}{1}\right)^2 + \left(\frac{0}{1}\right)^2} \tag{2.41}$$

$$\approx 0.354 \tag{2.42}$$

The last distance is smaller than the previous cases, which indicates that points A and B are closer using this anisotropic distance.

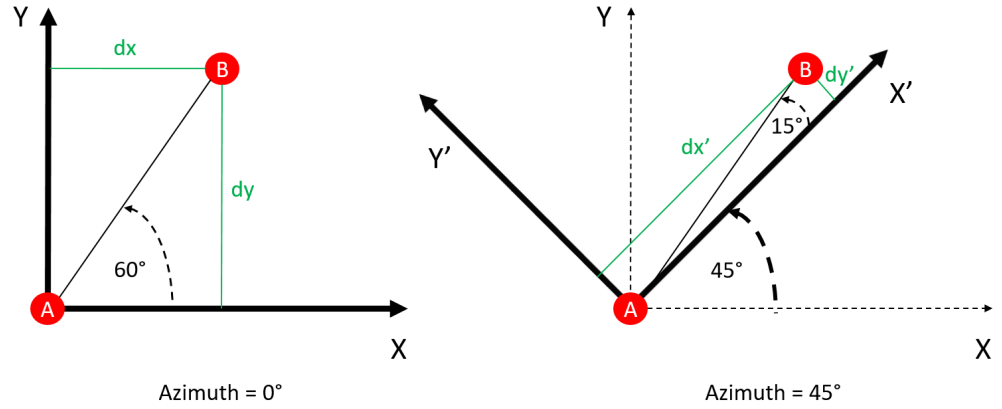

FIGURE 2.6: Two scenarios of anisotropic distance calculation, using azimuth equal to 0° (left) and azimuth equal to 45° (right).

2.3.1.2 LVA field

Expanding the previous definition, we can have local definitions of anisotropic distance on each location of the domain. This expansion will require the definition of a field of anisotropic parameters, also denoted as the LVA field. According to Boisvert [2010], the LVA field should contain 5 variables for each point in the domain: azimuth (or strike), dip, plunge, ratio 1 and ratio 2. The rotation angles are the same as in the previous definition of anisotropic distance. Ratio 1 or r_1 is the proportion between the rotated axis X and Y, and ratio 2 or r_2 is the proportion between the rotated axis Z and Y. For historical reasons, the axis X and Y are denoted the minor and major directions respectively, and Z is denoted the vertical direction. With these parameters, the anisotropic distance can be computed as

$$d(\mathbf{h}) = \sqrt{\mathbf{h}^T \mathbf{R}^T \mathbf{R} \mathbf{h}} \tag{2.43}$$

where $\mathbf{R}$ is the rotation matrix obtained with the LVA parameters of one of the extremes of the vector

$$\mathbf{R}(\alpha, \beta, \varphi, r_1, r_2) = \begin{bmatrix} \cos(\alpha)\cos(\varphi) - \sin(\alpha)\sin(\beta)\sin(\varphi) & -\sin(\alpha)\cos(\varphi) - \cos(\alpha)\sin(\beta)\sin(\varphi) & \cos(\beta)\sin(\varphi) \\ \frac{1}{r_1}\sin(\alpha)\cos(\beta) & \frac{1}{r_1}\cos(\alpha)\cos(\beta) & \frac{1}{r_1}\sin(\beta) \\ \frac{1}{r_2}(-\cos(\alpha)\sin(\varphi) - \sin(\alpha)\sin(\beta)\cos(\varphi)) & \frac{1}{r_2}(\sin(\alpha)\sin(\varphi) - \cos(\alpha)\sin(\beta)\cos(\varphi)) & \frac{1}{r_2}\cos(\beta)\cos(\varphi) \end{bmatrix} \tag{2.44}$$

Figure 2.7 shows a common visualization of the LVA field, which uses the major direction as "arrow" on each location, in order to represent the prescense of local anisotropy. With all anisotropic distances computed between each contiguous locations, different paths between non contiguous locations can be evaluated by adding each contiguous contribution to the anisotropic distance along the path. In order to select the optimal or "shortest" path between two locations in the domain, a connectivity graph should be created, using the contiguous anisotropic distances as edge weights on the graph.

2.3.1.3 Connectivity graph

The distance transformation proposed in Boisvert [2010] is based on the ISOMAP manifold learning method from Tenenbaum et al. [2000] and its sparse version L-ISOMAP from de Silva and Tenenbaum [2004]. Both methods were designed originally for dimensionality reduction. Typically they "learn" how to transform data in a high dimensional space $\mathbb{R}^d$ with $d \geq 3$ into a lower dimensional space typically $\mathbb{R}^d$ with $d = 2$. Figure 2.8 shows a classical example, known as "swiss roll", in which original three-dimensional

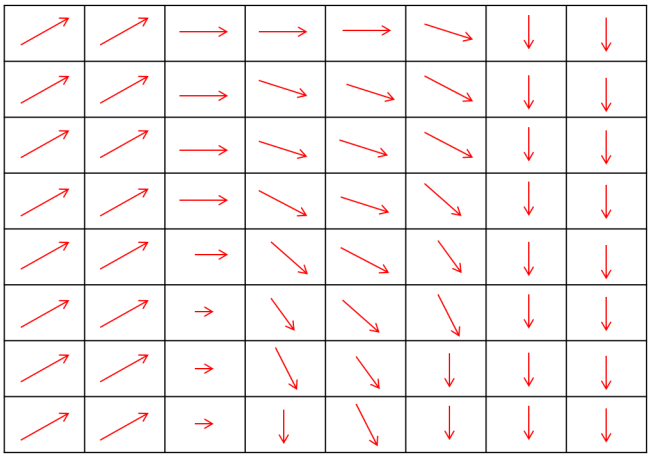

FIGURE 2.7: Synthetic visualization of a LVA field, represented by its major direction on each location.

data that lies into a two-dimensional manifold or surface (A) is transformed in to two-dimensional data (C) preserving the geodesic distances between points. The intermediate step needed for this transformation is the contruction of the connectivity graph identifying the shortest path on it (B) that allows to capture distance relations between well separated points.

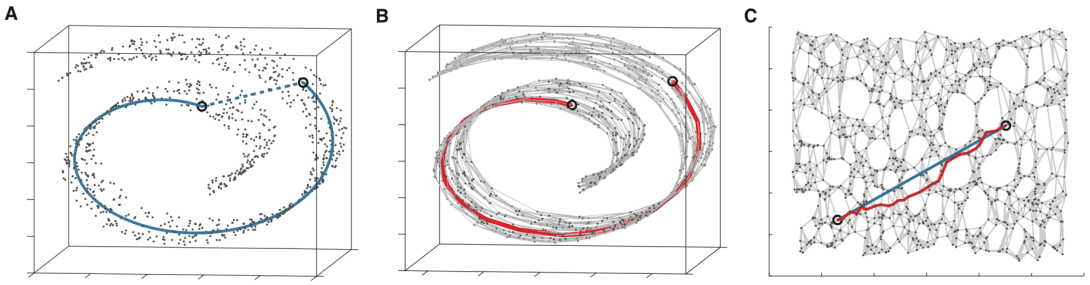

FIGURE 2.8: Dimensionality reduction using ISOMAP, extracted from Tenenbaum et al. [2000]. (A) Original data belongs to a non-linear manifold (surface) on $\mathbb{R}^3$ and is connected through geodesic distances. (B) Graph representation of the manifold. (B) Two dimensional embedding recovered using the method.

The three basic steps in these methods are the connectivity graph building, non-euclidean distance matrix assembly through shortest path calculations, and multidimensional scaling through singular value decomposition (SVD). The main differences between ISOMAP and L-ISOMAP, are related with the use of landmark points. Since both methods should assemble a distance matrix, if a large number of points are being processed, the matrix can be difficult or even computationally unfeasible to calculate. For instance, if we have $N = 10^6$ original points, the upper triangular part of distance matrix will have $N \times (N-1) \times 0.5$ entries, approximately 5×10^{11} entries (3.6TBytes assuming double-precision types). For this reason a small subset of points, denoted landmarks, are used as proxies to infer the distance between any pair of points. In the same example, if

$n = 1000$ landmarks are used, the distance matrix will have $n \times (N-1) \times 0.5$ entries (3.7GBytes assuming double-precision types). An additional step is needed if landmark points are used, in order to infer the distance between non-landmark points, a process of triangulation is performed through extra algebraic operations over the resulting matrix.

As mentioned before, the initial structure that must be computed for L-ISOMAP method is the connectivity graph over the domain Ω. Relevant parameters of this step are the LVA field, denoted $\mathcal{F}$, and the graph connectivity policy π. As commented in previous Section 2.3.1.2, LVA field is processed to calculate local anisotropic distances between contiguous points. If non-contiguous or mixed points should be considered for the local anisotropic distances, those connectivity policies are defined in the policy π.

The pamaterer π is used to define the neighbourhood $\mathcal{N}$ for each domain point which will be considered in the connectivity graph, i.e. for each neighbour an edge will be added to the graph. In practical terms, the policy π consists of a value Δ which sets the number of separation edges ("hops") in the regular grid. For instance, $\Delta = 1$ will set a neighbourhood $\mathcal{N}$ of at most 6 points located at 1 hop of separation. The LVA field $\mathbf{F}$ defines a rotation matrix $\mathbf{R} := \mathbf{R}(\alpha, \beta, \varphi, r_1, r_2)$ (eqn. (2.44)) which was previously computed for each domain point in order to calculate the local anisotropic distance in each cell of the gridded domain. With the neighbourhood and rotation matrix computed, for each neighbour, an edge is added to the graph. Each edge has weight equal to $d = \sqrt{\mathbf{h}^T\mathbf{R}^T\mathbf{R}\mathbf{h}}$ with $\mathbf{h}$ lag vector between the edge endpoints. The resulting graph, denoted $\mathbf{G}$, will contain all relevant connections between points, according to the connectivity policy and local anisotropic distances computed previously. Several paths can be traversed in order to connect two non-contiguous points (Figure 2.9)

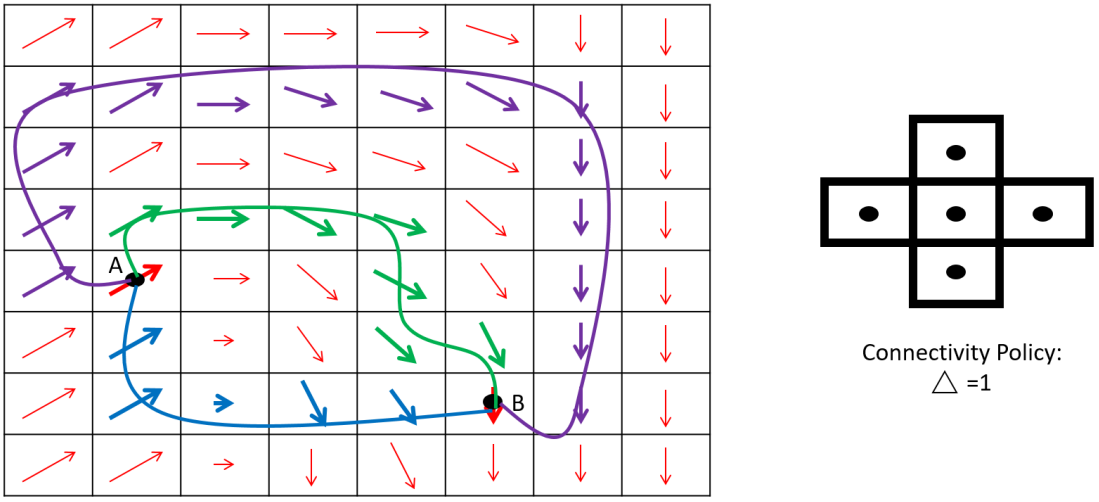

FIGURE 2.9: Synthetic visualization of different paths (green, blue and purple) connecting points A and B through the connectivity graph built using the LVA field and a connectivity policy.

2.3.1.4 Non-euclidean distance matrix

Once the connectivity graph is already assembled, non-euclidean distances should be calculated. Since L-ISOMAP method is being used, we can calculate distances between landmarks and non-landmark points. Each distance is the result of a shortest path computation on top of the connectivity graph, using one landmark point as origin and every non-landmark point as destination. To perform this tasks, well-known Dijkstra algorithm [Dijkstra, 1959] can be used, in its single-source shortest-path (SSSP) version. This algorithm will return all shortest paths between the origin (a landmark point) and every possible destination in the domain. In this way, the number of execution of SSSP Dijkstra algorithm is equal to the number of landmark points used in the method. With every distance already computed, the distance matrix $\mathbf{D} \in \mathbb{R}^{n \times N}$ can be assembled (only its upper triangular part, since it is symmetric), where n is the number of landmarks and N the total number of domain points.

2.3.1.5 Multidimensional Scaling

Based on the distance matrix $\mathbf{D}$, the third step of the L-ISOMAP method is the computation of the transformed points, also known as embedding $\mathcal{Z}$. The method used by Tenenbaum et al. [2000] and de Silva and Tenenbaum [2004] is Multidimensional Scaling, described in detail in Mardia et al. [1979], which uses a singular value decomposition to extract p eigenvalues and eigenvector from a special form of the distance matrix, defined as inner-product matrix, and assembles $\mathcal{Z}$ based on those k objects. The definition for squared distance matrices is as follows: for any distance matrix $\mathbf{D} \in \mathbb{R}^{n \times n}$, let

$$\mathbf{A} = (a_{rs}), \qquad a_{rs} = -\frac{1}{2} d_{rs}^2 \tag{2.45}$$

The inner-product matrix $\mathbf{B}$ is defined as

$$\mathbf{B} = \mathbf{HAH} \tag{2.46}$$

where $\mathbf{H} = \mathbf{I} - \frac{1}{n}\mathbf{1}\mathbf{1}'$ is called the $n \times n$ centering matrix, and $\mathbf{1}$ is the vector of ones of size n. The main result of the MDS method can be stated as a theorem:

Theorem 1 (MDS). Let $\mathbf{D} \in \mathbb{R}^{n \times n}$ be a distance matrix and define $\mathbf{B}$ by (2.46). Then $\mathbf{D}$ is Euclidean if and only if $\mathbf{B}$ is positive semidefinite (p.s.d.). In particular, the following results hold:

(a) If $\mathbf{D}$ is the matrix of Euclidean distances for a configuration of points $\mathbf{U} = (\mathbf{u}_1, \ldots, \mathbf{u}_n)^T$, then

$$b_{rs} = (\mathbf{u}_r - \overline{\mathbf{u}})^T (\mathbf{u}_s - \overline{\mathbf{u}}) \ , r, s = 1, \ldots, n \tag{2.47}$$

In matrix form (2.47) becomes $\mathbf{B} = (\mathbf{HU})(\mathbf{HU})^T$, so $\mathbf{B} \geq 0$. Note that B can be interpreted as the "centered inner product matrix" for the configuration $\mathbf{U}$.

(b) Conversely, if $\mathbf{B}$ is p.s.d. with rank p then a configuration can be constructed as follows. Let $\lambda_1 > \cdots > \lambda_p$ denote the positive eigenvalues of $\mathbf{B}$ with corresponding eigenvectors $\mathbf{X} = (\mathbf{x}_{(1)}, \ldots, \mathbf{x}_{(p)})$ normalized by

$$\mathbf{x}_{(i)}^T \mathbf{x}_{(i)} = \lambda_i, \ i = 1, \ldots, p. \tag{2.48}$$

Then the points $\mathcal{P}$, in $\mathbb{R}^p$ with coordinates $\mathbf{x}_r = (x_{r1}, \ldots, x_{rp})^T$ (so $\mathbf{x}_r$ is the r-th row of $\mathbf{X}$) have inter-point distances given by $\mathbf{D}$. Further, this configuration has centre of gravity $\overline{\mathbf{x}} = 0$, and $\mathbf{B}$ represents the inner product matrix for this configuration.■

Point (b) of the theorem gives the explicit form of the data points in the new space, in this case using the top p eigenvalues and corresponding eigenvectors. The transformation can be represented as

$$[\mathbf{u}_1, \ldots, \mathbf{u}_n] \in \mathbb{R}^3 \longrightarrow \mathbf{D} \in \mathbb{R}^{n \times n}$$

$$\mathcal{Z} = \begin{bmatrix} \frac{1}{\sqrt{\lambda_1}} \mathbf{x}_{(1)}^T \\ \vdots \\ \frac{1}{\sqrt{\lambda_p}} \mathbf{x}_{(p)}^T \end{bmatrix} \in \mathbb{R}^p, \ p \leq n \longrightarrow \mathbf{D}' \in \mathbb{R}^{n \times n}$$

where $\mathbf{D} \approx \mathbf{D}'$, which means that the distances between points are similar either using the original data $\mathbf{u}_i$ with non-euclidean distances in $\mathbb{R}^3$, or using the transformed data $\mathbf{x}_i$ with euclidean distances in $\mathbb{R}^p$.

This similarity allows to switch from the original data space to the new transformed space in order to recover the euclidean distance for covariance function calculations or nearby neighbour search. All classical geostatistical kernels and applications can be used without violating the positive definiteness of the covariance functions.

An additional step discussed on de Silva and Tenenbaum [2004], is called distance triangulation. In practice, n landmark points are used to calculate a squared distance matrix $\mathbf{D} \in \mathbb{R}^{n \times n}$ (distances between landmarks). The output of MDS applied to this smaller matrix is a p-dimensional ($p \leq n$) embedding $\mathcal{Z}$ calculated only for the n landmark points. The new coordinates $\mathbf{x}$ for a non-landmark point $\mathbf{u}$ are obtained by an affine linear transformation of the vector $\mathbf{a_u}$ of its squared distances to the landmark points

$\mathbf{l}_1, \ldots, \mathbf{l}_n$:

$$\mathbf{a}_\mathbf{u} = \begin{pmatrix} (\mathbf{u}-\mathbf{l}_1)^T(\mathbf{u}-\mathbf{l}_1) \\ \cdots \\ (\mathbf{u}-\mathbf{l}_1)^T(\mathbf{u}-\mathbf{l}_n) \end{pmatrix} \tag{2.49}$$

The mean vector $\overline{\mathbf{a}}$ is defined as

$$\overline{\mathbf{a}} = \frac{1}{n}\left(\mathbf{a}_{\mathbf{l}_1} + \cdots + \mathbf{a}_{\mathbf{l}_n}\right) \tag{2.50}$$

and the matrix $\mathbf{L}_p^*$ (pseudo inverse transpose of the p-dimensional embedding vectors):

$$\mathbf{L}_p^* = \begin{bmatrix} \sqrt{\lambda_1}\mathbf{x}_{(1)}^T \\ \vdots \\ \sqrt{\lambda_p}\mathbf{x}_{(p)}^T \end{bmatrix} \tag{2.51}$$

Finally, the affine transformation for $\mathbf{u} \in \mathbb{R}^3$ is as follows:

$$\mathbf{x} = -\frac{1}{2}\mathbf{L}_p^*(\mathbf{a}_\mathbf{u} - \overline{\mathbf{a}}) \tag{2.52}$$

which is a matrix-vector algebraic operation that can be applied to every non-landmark point efficiently.

2.3.2 Sequential implementations

Regarding sequential (single-thread) implementations of LVA-based algorithms, the baseline work was proposed by Boisvert and Deutsch [2011], in which three well known applications were adapted to use LVA: variogram computation, kriging estimation and sequential gaussian simulation. The original source code of the LVA-based applications is based on Fortran and C++, and can be downloaded from the main authors website[1]. Gutierrez and Ortiz [2019] contributed posteriorly with the implementation of sequential indicator simulation, however, its usability was limited to small scenarios due to several performance issues. All of these codes are based on the well known GSLIB code base from Deutsch and Journel [1998]. As mentioned in Section 2.3.1.3, the existing LVA implementations are based on the L-ISOMAP manifold learning method from Tenenbaum et al. [2000].

In the rest of this section, specific details are presented about the implementations of Boisvert and Deutsch [2011] and Gutierrez and Ortiz [2019], connecting theoretical aspects from previous sections.

[1] http://www.ualberta.ca/~jbb/LVA_code.html

2.3.2.1 LVA-based Sequential Simulation

Algorithms 4 and 5 show the unified steps needed to generate sequential gaussian and indicator simulations, using the LVA-approach. Both applications, denoted `sgs-lva` and `sisim-lva`, are equivalent to the classical algorithms `sgsim` and `sisim` (Algorithms 2 and 3), the only major differences between them are the first 4 lines, where three new pseudo-routines are called: `build_connectivity_graph`, `build_distance_matrix` and `build_embedding`. Each of them will be described in the next sections. Additional minor changes between the classical and LVA-based implementations are related with extra parameters used in `search_neighbours` and `kriging` routines. Since the new coordinates calculated in the embedding $\mathcal{Z}$ should be used to compute euclidean distances in $\mathbb{R}^p$, it should be passed as parameter to the corresponding routines. Two parameters can control the maximum number of dimensions used on neighbour search distances and covariance distances, $k^{search} \leq p$ and $k^{cova} \leq p$ respectively. For instance, if the landmark points Ω_L is a sub-grid of dimensions $10 \times 10 \times 10$, we will have $p \leq 1000$. Additionally, the user can select additional uppper bounds to this value, such that $k^{cova} = 3$ and $k^{cova} = 500$, which means that the routine `search_neighbours` will use the first 3 dimensions of the embedding vectors in $\mathcal{Z}$, and the routine `kriging` or `indicator_kriging` will use the first 500 dimensions of $\mathcal{Z}$.

Input:
$(\mathbf{V}, \Omega)$: sample database values $\mathbf{V}$ defined in a 3D domain Ω;
Ω_L: landmark 3D domain (subset of Ω) [Only LVA];
$\mathbf{F}$: LVA field defined in Ω [Only LVA];
π: connectivity graph policy [Only LVA];
k^{search}: maximum search distance dimensions [Only LVA];
k^{cova}: maximum covariance distance dimension [Only LVA];
n^{max}: maximum neighbours for interpolation;
κ: local interpolation parameters;
τ: seed for pseudo-random number generator;
S: number of generated simulations;
`output.txt`: output file name

// Only LVA: First calculate the embedding using L-ISOMAP
$\mathbf{G} \leftarrow$ `build_connectivity_graph`$(\Omega, \mathbf{F}, \pi)$
$\mathbf{D} \leftarrow$ `build_distance_matrix`$(\mathbf{G}, \Omega, \Omega_L)$
$\mathcal{Z} \leftarrow$ `build_embedding`$(\mathbf{D})$
//non-LVA and LVA: Then proceed with the simulation routines using the embedding to calculate distances
$\mathbf{Y} \leftarrow$ `normal_score`$(\mathbf{V})$
for $isim \in \{1, \ldots, S\}$ **do**
  $\mathcal{P} \leftarrow$ `create_random_path`(Ω, τ) *//Array with index random re-ordering*
  $\mathbf{Y}^{tmp} \leftarrow$ `zeros`$(\mathbf{Y})$
  $\mathbf{Y}^{tmp} \leftarrow$ `assign`$(\mathbf{Y})$ *//Sample data assignment*
  for $ixyz \in \{1, \ldots, |\Omega|\}$ **do**
    `index` $\leftarrow \mathcal{P}_{\texttt{ixyz}}$
    LocalNeighbours $\leftarrow$ `search_neighbours`$(\texttt{index}, \kappa, \mathcal{Z}, k^{search})$
    $p \leftarrow$ `kriging`$(\texttt{index}, \textbf{LocalNeighbours}, \gamma, \kappa, \mathcal{Z}, k^{cova}, n^{max})$
    $\mathbf{Y}^{tmp}(\texttt{index}, \texttt{isim}) \leftarrow$ `simulate`$(\texttt{index}, p, \tau)$
  end
  $\mathbf{V}^{tmp} \leftarrow$ `back_transform`$(\mathbf{Y}^{tmp})$
  `write`$(\texttt{output.txt}, \mathbf{V}^{tmp})$
end

Output: S stochastic simulations stored in file `output.txt`

Algorithm 4: Sequential Gaussian Simulation for LVA scenarios

Input:
$(\mathbf{V}, \Omega)$: sample database values $\mathbf{V}$ defined in a 3D domain Ω;
Ω_L: landmark 3D domain (subset of Ω) [Only LVA];
$\mathbf{F}$: LVA field defined in Ω [Only LVA];
π: connectivity graph policy [Only LVA];
k^{search}: maximum search distance dimensions [Only LVA];
k^{cova}: maximum covariance distance dimension [Only LVA];
n^{max}: maximum neighbours for interpolation;
C: number of categories to be reproduced;
$\gamma_1, \ldots, \gamma_C$: structural variographic models;
κ: local interpolation parameters;
τ: seed for pseudo-random number generator;
S: number of generated simulations;
`output.txt`: output file name

// Only LVA: First calculate the embedding using L-ISOMAP
$\mathbf{G} \leftarrow$ `build_connectivity_graph`$(\Omega, \mathbf{F}, \pi)$
$\mathbf{D} \leftarrow$ `build_distance_matrix`$(\mathbf{G}, \Omega, \Omega_L)$
$\mathcal{Z} \leftarrow$ `build_embedding`$(\mathbf{D})$
//non-LVA and LVA: Then proceed with the simulation routines using the embedding to calculate distances
for $isim \in \{1, \ldots, S\}$ **do**
  $\mathcal{P} \leftarrow$ `create_random_path`(Ω, τ) *//Array with index random re-ordering*
  $\mathbf{V}^{tmp} \leftarrow$ `zeros`$(\mathbf{V})$
  $\mathbf{V}^{tmp} \leftarrow$ `assign`$(\mathbf{V})$ *//Sample data assignment*
  for $ixyz \in \{1, \ldots, |\Omega|\}$ **do**
    `index` $\leftarrow \mathcal{P}_{\texttt{ixyz}}$
    LocalNeighbours $\leftarrow$ `search_neighbours`$(\texttt{index}, \kappa, \mathcal{Z}, k^{search})$
    for $icut \in \{1, \ldots, C\}$ **do**
      $p_{icut} \leftarrow$ `indicator_kriging`$(\texttt{index}, \textbf{LocalNeighbours}, \gamma_{\texttt{icut}}, \kappa, \mathcal{Z}, k^{cova}, n^{max})$
    end
    $\mathbf{V}^{tmp}(\texttt{index}, \texttt{isim}) \leftarrow$ `simulate`$(\texttt{index}, p_1, \ldots, p_C, \tau)$
  end
  `write`$(\texttt{output.txt}, \mathbf{V}^{tmp})$
end

Output: S stochastic simulations stored in file `output.txt`

Algorithm 5: Sequential Indicator Simulation for LVA scenarios

2.3.2.2 Connectivity graph

The implementation proposed by Boisvert and Deutsch [2011] to build the connectivity graph can be viewed in Algorithm 6. It gets three input parameters: a 3D domain Ω (gridded), the LVA field represented by the 5 anisotropy parameters for each domain location (see Section 2.3.1.2) and the graph connectivity policy (see Section 2.3.1.3).

The algorithm loops through all domain locations (line 2), and for each location computes the contiguous neighbours according to the connectivity policy π (line 3). This operation doesn't require complex computations since the domain is gridded, using the one-dimensional natural indexation and coordinate indexes back and forth as required:

$$loc = (iz-1)\cdot nx\cdot ny + (iy-1)\cdot nx + ix \tag{2.53}$$

$$iz = 1 + int\left(\frac{loc-1}{nx\cdot ny}\right) \tag{2.54}$$

$$iy = 1 + int\left(\frac{loc-(iz-1)\cdot nx\cdot ny}{nx}\right) \tag{2.55}$$

$$ix = loc-(iz-1)\cdot nx\cdot ny-(iy-1)*nx \tag{2.56}$$

with nx, ny, nz the grid size on each respective dimension. For instance, using a connectivity policy of 1, the corresponding contiguous neighbours of (ix, iy, iz) are $(ix \pm 1, iy, iz)$, $(ix, iy \pm 1, iz)$ and $(ix, iy, iz \pm 1)$. After obtaining the neighbours, the rotation matrix $\mathbf{R}$ should be computed for the location being processed (line 4). After that, a loop through all contiguous neighbours can be traversed in order to calculate the separation vector $\mathbf{h}$ between the neighbour and the center location (line 6), and posteriorly apply the anisotropic distance calculation using $\mathbf{R}$ and $\mathbf{h}$ (line 7). The edge between the neighbour and center location, denoted e (line 8), and the anisotropic distance d are stored in the graph array $\mathbf{G}$ (line 9). The final step consists in removing redundant edges from the graph (line 11).

2.3.2.3 Non-euclidean distance matrix

The implementation proposed by Boisvert and Deutsch [2011] to build the non-euclidean distance matrix can be viewed in Algorithm 7. The baseline implementation reads two files from disk: `nodes2cal.out` (landmark points list) and `grid.out` (connectivity graph) (lines 3 and 4). With both files loaded into memory, for each landmark point a shortest path calculation must be performed through the connectivity graph $\mathbf{G}$ (line 6). This step is computed using Dijkstra's shortest path algorithm [Dijkstra, 1959], implemented in the C++ Boost Library [Boost.org, 2012]. All distances from a landmark to all graph nodes are appended to the file `dist_cpp.out` (line 7). As mentioned before, the baseline implementation performs a system call to launch the execution of a compiled

Input:
Ω: 3D domain Ω;
$\mathbf{F}$: LVA field in each domain point of Ω;
π: graph connectivity policy;
1 $\mathbf{G} \leftarrow \emptyset$ *//Empty graph*
2 **for** $ixyz \in \{1, \ldots, |\Omega|\}$ **do**
3 $\quad \mathcal{N} \leftarrow$ Compute all neighbours of point `ixyz` according to policy π
4 $\quad \mathbf{R} \leftarrow$ Compute rotation matrix of point `ixyz` according to LVA field $\mathbf{F}$
5 $\quad$ **for** $neig \in \mathcal{N}$ **do**
6 $\quad\quad \mathbf{h} \leftarrow$ Lag vector between points `ixyz` and `neig`
7 $\quad\quad d \leftarrow$ Compute anisotropic distance between point `ixyz` and `neig` according to $\sqrt{\mathbf{h}^T\mathbf{R}^T\mathbf{R}\mathbf{h}}$
8 $\quad\quad e \leftarrow \{$`ixyz`, `neig`$\}$ *//Definition of graph edge with weight* d
9 $\quad\quad$ Add (e, d) to graph $\mathbf{G}$
10 $\quad$ **end**
11 $\quad$ Remove redundant edges from $\mathbf{G}$
12 **end**
13 Write $\mathbf{G}$ to file `grid.out`
Output: $\mathbf{G}$ in file `grid.out`

Algorithm 6: Routine `build_connectivity_graph`

C++ code with the Boost Library call to Dijkstra routine, and the data transfer between Fortran and C++ is performed through expensive disk I/O communication.

Input:
`grid.out`: file with graph $\mathbf{G}$ based on domain points Ω;
`nodes2cal.out`: file with landmark points indices of Ω_L
1 $(N, n) \leftarrow$ Read size of domain points N and landmark points n from `nodes2cal.out`
2 $\mathbf{D} \leftarrow$ `zeros`(N, n)
3 $\mathcal{N}^{Landmark} \leftarrow$ Read landmark point indexes from `nodes2cal.out`
4 $\mathbf{G} \leftarrow$ Read connectivity graph from `grid.out`
5 **for** $i \in \mathcal{N}^{Landmark}$ **do**
6 $\quad \mathbf{D}_{:,i} \leftarrow$ `run_dijsktra`$(i, \mathbf{G})$
7 $\quad$ Write distances $\mathbf{D}_{:,i}$ into file `dist_cpp.out`
8 **end**
Output: File `dist_cpp.out`

Algorithm 7: Routine `build_distance_matrix`

2.3.2.4 Multidimensional Scaling

The implementation proposed by Boisvert and Deutsch [2011] to build the multidimensional scaling procedure can be viewed in Algorithm 8. The input of the algorithm is file `dist_cpp.out`, computed in Algorithm 7, that contains the shortest path distances between landmark and domain points.

First, the file `dist_cpp.out` is loaded into matrix $\mathbf{D}$ (line 2) and transformed into $\mathbf{B}$ (line 3). After this step, matrix $\mathbf{B}^T\mathbf{B}$ is factorized using a Singular Value Decomposition method, and the $p \leq n$ positive largest eigenvalues are selected, being n the number of landmark points. Finally, the embedding $\mathcal{Z}$ is defined as the rows of the matrix $\mathbf{Y}$ with

columns $\sqrt{\lambda_i}^{-1}\mathbf{B}\mathbf{v}_i$ for $i \in \{1, \ldots, p\}$ (lines 9 and 10), being $\mathbf{v}_i$ the corresponding eigenvector of λ_i. Some of the algebraic operations of type matrix-matrix and matrix-vector multiplication, together with the SVD method, are calculated using BLAS [Blackford et al., 2002] and LAPACK [Anderson et al., 1999] routines, whose original source code is included in a subfolder of the baseline implementation.

Input:
`dist_cpp.out`: file with distance matrix values $\mathbf{D}$;
1 $(N, n) \leftarrow$ Read size N of domain points Ω and size n of landmark points Ω_L from `dist_cpp.out`
2 $\mathbf{D} \leftarrow$ Read distance matrix from `dist_cpp.out` with size $N \times n$
3 $\mathbf{A} \leftarrow a_{ij} = -\frac{1}{2}d_{ij}^2$, $\forall i \in \{1, \ldots, N\}$, $\forall j \in \{1, \ldots, n\}$
4 $\mathbf{H_n} \leftarrow \mathbf{I}_n - \frac{1}{n}\mathbf{J}_n$, with $\mathbf{I}_n$ identity matrix of size n and $\mathbf{J}_n$ matrix of all 1's of size n
5 $\mathbf{H_N} \leftarrow \mathbf{I}_N - \frac{1}{N}\mathbf{J}_N$
6 $\mathbf{B} \leftarrow \mathbf{H}_N\mathbf{A}\mathbf{H}_n$
7 $(\mathbf{\Lambda}, \mathbf{V}) \leftarrow$ `eigen`$(\mathbf{B}^T\mathbf{B})$ *// calculate eigevalues and eigenvectors*
8 $(\mathbf{\Lambda}, \mathbf{V}) \leftarrow$ Select $p \leq n$ positive largest eigenvalues $\lambda_1 \geq \cdots \geq \lambda_p > 0$ of $\mathbf{\Lambda}$ with corresponding eigenvectors $\mathbf{V}$ which satisfy $\left(\frac{\mathbf{B}\mathbf{v}_i}{\sqrt{\lambda_i}}\right)^T \left(\frac{\mathbf{B}\mathbf{v}_i}{\sqrt{\lambda_i}}\right) = 1$ for all $i \in \{1, \ldots, p\}$
9 $\mathbf{Y} \leftarrow \left[\ \frac{1}{\sqrt{\lambda_1}}\mathbf{B}\mathbf{v}_1 \ \middle|\ \cdots \ \middle|\ \frac{1}{\sqrt{\lambda_p}}\mathbf{B}\mathbf{v}_p \ \right] \in \mathbb{R}^{N \times p}$
10 $\mathcal{Z} \leftarrow \{\mathbf{z}_1, \ldots, \mathbf{z}_N\} \subset \mathbb{R}^p$ where $\mathbf{z}_i$ is the i-th row of $\mathbf{Y}$ for all $i \in \{1, \ldots, N\}$. The following property holds: $\|\mathbf{z}_i - \mathbf{z}_j\|_2 = d_{ij}$, $\forall i \in \{1, \ldots, N\}$, $\forall j \in \{1, \ldots, n\}$ (after row and column reordering if necessary)
Output: Embedding $\mathcal{Z} \subset \mathbb{R}^p$, with $p \leq n$

Algorithm 8: Routine `build_embedding`

2.3.3 Parallel implementations

To the knowledge of the author, no public available parallel implementation of the LVA-based geostatistical applications have been published or released. Initial attempts were reported by Peredo et al. [2015b], were the Dijkstra sequential implementation was replaced by its distributed/parallel version, without relevant performance improvements. In the next chapters, parallel algorithms are presented, in order to accelerate key routines and computations.

Chapter 3

Methodology

The main contribution of this thesis is to optimize and accelerate some of the classical and LVA-based algorithms in geostatistics, particularly sequential simulation methods. Baseline codes that will be accelerated are `sgsim` and `sisim` from classical methods (Algorithms 2 and 3), and `sgs-lva` and `sisim-lva` from LVA-based methods (Algorithms 4 and 5). These codes are directly included or modified from the Geostatistical Software Library (GSLIB) Deutsch and Journel [1998], which will be described in this chapter. The methodology used to identify optimization points and parallel strategies in the baseline codes is to calculate a profile of the execution for each code. Based on this profile, improvements can be prioritized. The case studies are related with synthetic and real three dimensional scenarios, where continuous and categorical variables should be simulated by the corresponding methods, generating realizations or "images". Description of parameters used by the simulation methods are included in this chapter. Later, specific parameter values used on each scenario are described in each corresponding chapter. Finally, performance metrics used in this thesis are also presented in this chapter.

3.1 GSLIB

GSLIB[1] is composed by a set of utility routines written in Fortran 77/90, compiled and wrapped as a static library named `gslib.a`, and a set of applications written in Fortran 77/90 that call some of the wrapped routines. We will refer to these two sets as *applications* and *utilities*. In Figure 3.1 we can observe a diagram of GSLIB different applications and utilities, grouped according to their categories. Four main groups of applications can be observed: *variogram*, *kriging*, *simulation* and *others*. The first three groups are related with each specific algorithm or method: variogram calculation

[1]Direct link for source code download: `http://www.statios.com/software/gslib90_ls.tar.gz`

(Section 2.2.1.4), kriging estimation (Section 2.2.1.5) and stochastic simulation (Sections 2.2.1.6 and 2.2.1.7). The fourth group contains applications for plotting, data manipulation and post-processing treatment of other applications.

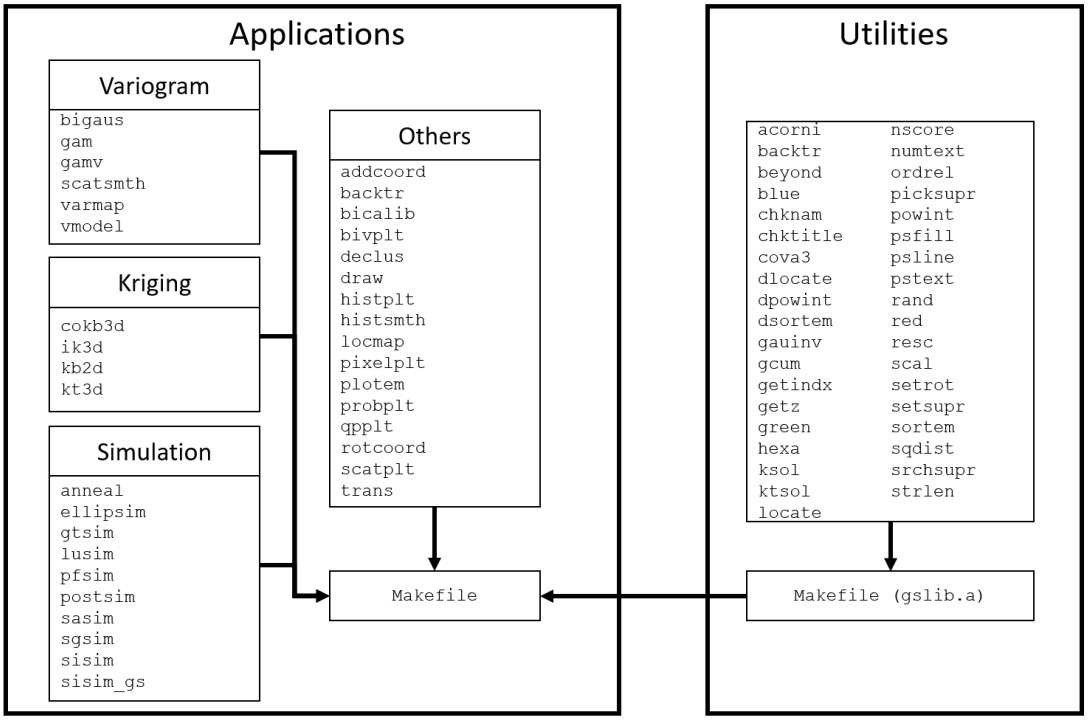

FIGURE 3.1: GSLIB applications and utilies.

Typically, an *application* is composed by a main program and two subroutines (Figure 3.2). The first subroutine is in charge of reading the parameters from the input files, and the second subroutine executes the main computation and writes out the results using predefined output formats. Additionally, two structures of static and dynamic variables are used by the main program and each subroutine: an include file and a `geostat` module. The include file contains static variable declarations, such as constant parameters, fixed length arrays and common blocks of variables. The `geostat` module contains dynamic array declarations, which will be allocated in some of the subroutines with the allocate instruction. A utility is self-contained allowing sharing variables with other utilities and applications through common block variable declarations.

The above-mentioned structure is common to many applications and has advantages and disadvantages. The user/programmer can easily identify each part of the code and where the main computations are occurring. However, the use of implicit typing and module/include variable declarations in the applications and utilities makes it difficult to set up a data-flow analysis Aho et al. [2006] of all the variables in any state of the execution. With this kind of analysis, the user/programmer can estimate the state of the variables in different parts of the code, being able to know if a variable is alive or

dead at some point of execution. From a final user perspective, this information can be seen as irrelevant. However, from a programmer's perspective, who intends to re-design some parts of the code or accelerate the overall execution, the data-flow analysis is an important step.

GSLIB application

```
 1   module geostat
 2      integer,allocatable :: ...
 3      ...
 4   end module
 5
 6   program main
 7      use geostat
 8      include 'application.inc'
 9      call readparm(...)
10      call application(...)
11   end
```

FIGURE 3.2: Original main program for GSLIB applications.

```
                  Parameters for SISIM_LVA
                  ************************

START OF PARAMETERS:
0                               -1=continuous(cdf), 0=categorical(pdf)
3                               -number thresholds/categories
1 2 3                           -codes
0.147 0.688 0.165               -proportions 0.149  0.688 0.163
muestras5.dat                   -file with data
1   2   3   7                   -columns for X,Y,Z, and variable
nodata.ik                       -file with soft indicator input
1   2   0   3 4 5 6 7           -columns for X,Y,Z, and indicators
0                               -Markov-Bayes simulation (0=no,1=yes)
0.61  0.54  0.56  0.53  0.29   -calibration B(z) values
-1.0e21    1.0e21               -trimming limits
0.0    8.0                      -minimum and maximum data value
1      0.0                      -lower tail option and parameter
1      0.0                      -middle     option and parameter
1      0.0                      -upper tail option and parameter
nodata.dat                      -file with tabulated values
3   0                           -columns for variable, weight
0                               -debugging level: 0,1,2,3
sisim.dbg                       -file for debugging output
sisim_lva.out                   -file for simulation output
1                               -number of realizations
400  0.5    1.0                 -nx,xmn,xsiz
600  0.5    1.0                 -ny,ymn,ysiz
12  6.0    12.0                 -nz,zmn,zsiz
27364556                        -random number seed
48                              -maximum original data  for each kriging
48                              -maximum previous nodes for each kriging
48                              -maximum soft indicator nodes for kriging
1                               -assign data to nodes? (0=no,1=yes)
1     3                         -multiple grid search? (0=no,1=yes),num
0                               -maximum per octant    (0=not used)
400.0 400.0  400.0              -maximum search radii
 0.0   0.0   0.0                -angles for search ellipsoid
300  150  60                    -size of covariance lookup table
0    2.5                        -0=full IK, 1=median approx. (cutoff)
0                               -0=SK, 1=OK
1    0.05                       -One   nst, nugget effect
1    0.95 0.0  0.0   0.0        -it,cc,ang1,ang2,ang3
150.0  150.0  150.0             -a_hmax, a_hmin, a_vert
1    0.05                       -Two   nst, nugget effect
1    0.95 0.0  0.0   0.0        -it,cc,ang1,ang2,ang3
150.0  150.0  150.0             -a_hmax, a_hmin, a_vert
1    0.05                       -Three   nst, nugget effect
1    0.95 0.0  0.0   0.0        -it,cc,ang1,ang2,ang3
150.0  150.0  150.0             -a_hmax, a_hmin, a_vert
lva.dat                         -file containing the LVA grid
4 5 6 7 8                       -LVA grid columns
400  0.5    1.0                 -nx,xmn,xsiz
600  0.5    1.0                 -ny,ymn,ysiz
12  6.0    12.0                 -nz,zmn,zsiz
1                               -noffsets for graph
2                               - MDS? 2=L-ISOMAP 3=read dist
10  15  6                       -number of landmark points in x,y,z
300                             -max n of dim (set -1 to use max)
400.0                           -maximum search radii
30                              -max n of dimensions to use in search
```

FIGURE 3.3: Sample parameter file for `sisim_lva` application.

3.2 Application parameters

Each GSLIB application should read a parameter file with specific parameters. In this section we will describe relevant parameters for `sgsim`, `sisim`, `sgs-lva` and `sisim-lva`. A sample parameter file can be viewed in Figure 3.3. For a complete description of each parameter, see Deutsch and Journel [1998] section V.7.2. In some cases, expert knowledge is needed to select proper values for them.

3.2.1 Common parameters

All applications share common parameters. They are described next:

- File with conditioning sampled data:
  Name of the file that contains conditioning data, characterized by each coodinate (x, y, z) and each variable associated to it (1 or more variables can be included).

- Simulation domain size (gridded):
  Three numbers by dimension should be declared: $(ni, isiz, imn)$ with ni the number of points in dimension i, $isiz$ the separation between points in the dimension, and imn the minimum value in the dimension. Values of i are x, y and z.

- Number of simulations:
  Number of simulated images to generate by the method.

- Pseudo random number seed:
  Number used to initialize pseudo-random number generators for each method.

- Kriging type:
  Type of kriging method to use: SK (eqn. (2.15)) or OK (eqn. (2.17)) mainly. Other methods are out of the scope of this thesis.

- Number of neighbours for kriging:
  Number of neighbours to search and be used by the kriging method selected.

- Maximum search radius and ellipsoid parameters:
  Maximum radius on each axis to be used for neighbour search. Axis angle and range for each axis can be defined in case of ellipsoidal search region to be used.

- Number of nested variographic structures and their parameters:
  Number of covariance/semivariogram functions to be computed on each covariance matrix entry for kriging estimation (eqns. (2.16) or (2.18)), according to eqn. (2.14).

3.2.2 `sgsim`

Specific parameters for `sgsim` are described here:

- Z-score transformation flag:
  Binary value that indicates if the normal score transformation should be applied to the conditioning sampled data. If this value is false, it is assumed that the data follows a standard normal distribution $N(0, 1)$.

- File with transformation table:
  Name of the file that contains the normal score transformation parameters as result of a Z-score transformation applied on the conditioning data. It can be generated

by the same method or in a previous transformation by other GSLIB application (`nscore`).

- Multi-grid search flag:
  Binary value that indicates if the multi-grid neighbour search method should be applied. This method will be explained in detail in Chapter 5.

- Covariance lookup table size:
  Three numbers that indicate the dimensions of a lookup table used to speedup the covariance function calculation.

3.2.3 `sisim`

Specific parameters for `sisim` are described here:

- Continuous/categorical variable flag:
  Binary value that indicates if the variable to be simulated is continuous or categorical.

- Number of thresholds/categories:
  Number of mutually exclusive classes or categories to be simulated.

- Thresholds/labels for each class/category:
  For each class or category, a number should be indicated. In case of a continuous variable, thresholds between classes should be declared. In case of a categorical variable, labels of each category should be declared.

- Simulation proportions:
  Global cumulative distribution function (continuous case) or probability distribution function (categorical case). One value for each class/category should be declared.

- Markov-Bayes simulation flag and its parameters:
  Binary value that indicates if Markov-Bayes [Deutsch and Journel, 1998] should be used for simulation. This method is left out of the scope of this thesis.

- Multi-grid search flag:
  Same as `sgsim`.

- Covariance lookup table size:
  Same as `sgsim`.

3.2.4 `sgs-lva` and `sisim-lva`

Specific parameters for `sgs-lva` and `sisim-lva` are described here:

- File with LVA field data:
  Name of the file that contains LVA field data, characterized by each coodinate (x, y, z) and each variable associated to it (Section 2.3.1.2).

- LVA field domain size:
  Same as the simulation domain size (gridded). It should match with it since each domain point should have anisotropic parameters.

- Number of offsets for connectivity graph:
  Connectivity policy used to build the connectivity graph (π from Section 2.3.1.3).

- Multidimensional scaling flag:
  Binary flag that indicates if the multidimensional scaling method should read the distance matrix (Section 2.3.1.5) from an existing file or not.

- Number of landmark points by axis:
  One number by dimension should be declared. Each number defined the subgrid to be used as landmark points. For instance, if a simulation domain grid of $nx \times ny \times nz$ is defined, a landmark grid of $nx_L \times ny_L \times nz_L$ will use domain points defined as

$$\begin{gathered}\left(i \cdot \left\lfloor \frac{nx}{nx_L} \right\rfloor, j \cdot \left\lfloor \frac{ny}{ny_L} \right\rfloor, k \cdot \left\lfloor \frac{nz}{nz_L} \right\rfloor\right), \\ i \in \{1, \ldots, nx_L\}, j \in \{1, \ldots, ny_L\}, k \in \{1, \ldots, nz_L\}\end{gathered} \tag{3.1}$$

- Maximum number of dimensions for neighbour search:
  Parameter n^{search} as defined in Section 2.3.2.1.

- Maximum number of dimensions for covariance calculation:
  Parameter n^{cova} as defined in Section 2.3.2.1.

3.3 Development techniques

In this section we will present some of the techniques used in the development of the accelerated codes. A first key information is the operating system used for development and testing all scenarios, which was Ubuntu Linux 18.04.6 LTS, using GCC and Intel compilers `gfortran` 4.8.5, `g++` 6.2.0 and `ifort` 13.1.1, depending on the code language: Fortran or C++. In order to identify optimization points in the code and apply parallelization code modifications, four steps are needed, which are described below.

3.3.1 Refactoring

First we have to re-design the application/utility code to identify the state of each variable, array or common block during the execution. This step is necessary to enable the user/programmer to identify the scope of each variable (data-flow analysis), in order to insert OpenMP directives into the code in a smooth and easy way.

Code refactoring was previously applied to `sgsim` and `sisim` based on a previous work published by Peredo et al. [2015a]. In case of `sgs-lva` and `sisim-lva`, an additional step should be applied, since disk I/O communication and C++ code execution is performed in the LVA base routines. Refactoring tasks are applied to the corresponding code in order to optimize the execution. The proposed refactoring changes are in favor of a unified in-memory execution (sequential and parallel) which improves performance, code development, debugging and allows future modifications more easily. C++ code was integrated as Fortran calls using C wrappers[2] and the Fortran module `iso_c_binding`[3] [4].

3.3.2 Profiling

Initially, a high-level analysis consists in adding elapsed time measurement instructions in specific parts of the code, such as initial and final lines of subroutines/functions/blocks of code. With Fortran, the preferred intrinsic routine is `system_clock`[5] [6], and with C++ the preferred function is `time`[7] from the C Time library (`time.h`), both of them returning the number of seconds from 00:00 Coordinated Universal Time (CUT) on January 1st

[2] Calling C++ from Fortran example: https://gist.github.com/Luthaf/4df78ca52b3caf7fbe0e

[3] Intel Fortran: https://www.intel.com/content/www/us/en/develop/documentation/fortran-compiler-oneapi-dev-guide-and-reference/top/compiler-reference/mixed-language-programming/standard-tools-for-interoperability/iso-c-binding.html

[4] GNU Fortran: https://gcc.gnu.org/onlinedocs/gfortran/ISO_005fC_005fBINDING.html#ISO_005fC_005fBINDING

[5] Intel Fortran: https://www.intel.com/content/www/us/en/develop/documentation/fortran-compiler-oneapi-dev-guide-and-reference/top/language-reference/a-to-z-reference/s-1/system-clock.html

[6] GNU Fortran: https://gcc.gnu.org/onlinedocs/gfortran/SYSTEM_005fCLOCK.html

[7] https://www.cplusplus.com/reference/ctime/time/

1970. With this code instrumentation already in place, we can obtain a high-level view of the most time consuming parts of the execution.

Secondly, a low-level analysis is conducted in order to study the run-time behavior of the application using a system profiler tool. In our case we choose the Linux-based tools `gprof` [Graham et al., 1982] and `strace/ltrace` [Johnson and Troan, 2004]. These tools can deliver several statistics, among the most important are: elapsed time per routine, elapsed time per line of code, number of system/library calls and number of calls per routine. Figure 3.5 depicts the output of a `gprof` profiling of an application. With this information, we can identify which lines of the application or used utilities are generating overhead. We can modify some parts of the code using the profiled information. For each modification, we must re-measure the elapsed time and statistics, in order to accept or reject the modification.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 50.42    174.72   174.72  4834695     0.00     0.00  __kdtree2_module_MOD_search_opt01
 25.16    261.89    87.17        4    21.79    79.99  __random2_MOD_init_genrand
  9.03    293.18    31.29        8     3.91     5.05  __kdtree2_module_MOD_build_tree_for_range
  5.83    313.38    20.20                             MAIN__
  2.63    322.50     9.12  5243995     0.00     0.00  __kdtree2_module_MOD_kdtree2_sort_results
  2.63    331.62     9.12 43617160     0.00     0.00  __kdtree2_module_MOD_spread_in_coordinate
  1.38    336.39     4.77 91631407     0.00     0.00  __kdtree2_priority_queue_module_MOD_pq_replace_max
  1.33    341.01     4.62        1     4.62     4.62  scramble_
  0.64    343.21     2.21        7     0.32     6.09  __kdtree2_module_MOD_kdtree2_create_v2
  0.43    344.69     1.48  4872863     0.00     0.00  __kdtree2_module_MOD_kdtree2_n_nearest_around_point
  0.33    345.84     1.16 56212985     0.00     0.00  __random2_MOD_grnd
  0.17    346.43     0.59                             __kdtree2_module_MOD_search
  0.03    346.52     0.09                             __kdtree2_priority_queue_module_MOD_pq_delete
  0.01    346.54     0.02                             frame_dummy
  0.00    346.55     0.01                             __kdtree2_priority_queue_module_MOD_pq_create
  0.00    346.55     0.01                             __kdtree2_module_MOD_kdtree2_n_nearest_around_point_omp
  0.00    346.55     0.00    14256     0.00     0.00  getindx_
  0.00    346.55     0.00     1000     0.00     0.00  acorni_
  0.00    346.55     0.00        6     0.00     0.00  chknam_
  0.00    346.55     0.00        4     0.00     0.00  setrot_

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.
^L
Copyright (C) 2012-2018 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
^L
```

FIGURE 3.4: Example of `gprof` output.

Finally, a deeper level of analysis consists in the execution of performance analysis tools, such as Extrae[8] and Paraver[9]. As stated in the official website, Extrae is a package devoted to generate trace-files for a post-mortem analysis based on different events sampled from the application, operating system and hardware. Paraver allows to have a qualitative global perception of the application behavior by visual inspection and then to be able to focus on the detailed quantitative analysis of the problems.

Specifically, Extrae is a tool that uses different interposition mechanisms to inject probes into the target application so as to gather information regarding the application performance. The method used in this thesis is the linker preload, in which the current operating systems allows injecting a shared library into an application before the application gets actually loaded. If the library that is being preloaded provides the same symbols as those contained in shared libraries of the application, such symbols can be wrapped in order to inject code in these calls. In Linux systems this technique is commonly known by using the `LD_PRELOAD` environment variable. Extrae contains substitution symbols for many parallel runtimes, such as OpenMP (either Intel, GNU or IBM runtimes), pthread, CUDA accelerated applications, and MPI applications. Regarding sampling mechanisms, Extrae can use signal timers and hardware performance counters. These last counters are based on PAPI [Mucci et al., 1999] and PMAPI interfaces to collect information regarding the microprocessor performance. With the advent of the components in the PAPI software, Extrae is not only able to collect information regarding how is behaving the microprocessor only, but also allows studying multiple components of the system (disk, network, operating system, among others) and also extend the study over the microprocessor (power consumption and thermal information). Extrae mainly collects these counter metrics at the parallel programming calls and at samples. It also allows capturing such information at the entry and exit points of the user routines instrumented.

Paraver offers a minimal set of views on a trace. Performance information in Paraver is presented with two main displays that provide qualitatively different types of information. The timeline display represents the behavior of the application along time and processes, in a way that easily conveys to the user a general understanding of the application behavior and simple identification of phases and patterns. The statistics display provides numerical analysis of the data that can be applied to any user selected region, helping to draw conclusions on where and how to focus the optimization effort.

[8] https://tools.bsc.es/extrae
[9] https://tools.bsc.es/paraver

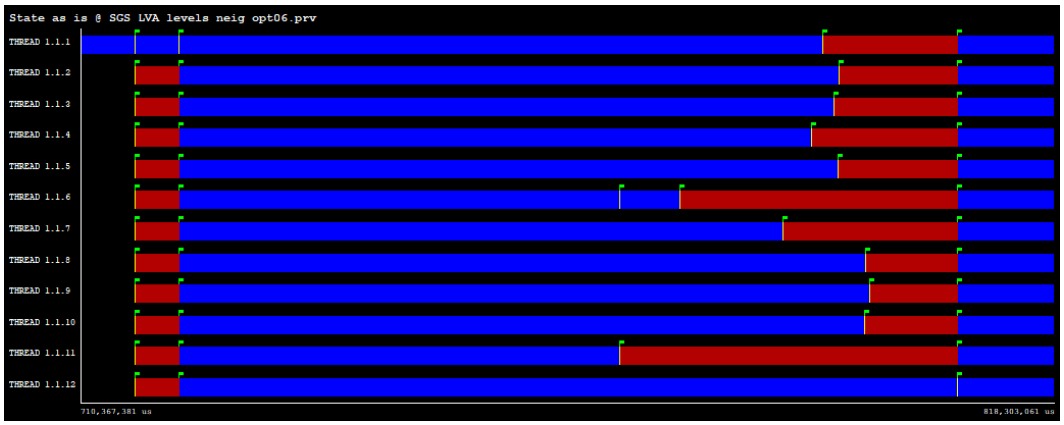

FIGURE 3.5: Example of Paraver output.

3.3.3 OpenMP parallelization

Once an optimized sequential version of the application is released, we can add OpenMP directives in the most time consuming parts of the code. Each directive defines a parallel region, which will be executed by several threads, with a maximum defined by the user. For each directive the user/programmer must study the data-flow of the variables inside the parallel region, and specify if the variables will be shared or private. Specific strategies that were applied to each application are described in detail in Chapters 4 and 5.

3.3.4 CUDA parallelization

In some special cases, a graphical processing unit (GPU) can be used to accelerate even more some routines or functions. This functionality was implemented for the simulation code `sgs-lva` and also for the variogram calculation code `gamv`. Experimental results are described in Chapters 6 and 7, using a hybrid CPU-GPU execution with OpenMP and CUDA parallelization.

3.4 Case studies

Two groups of case studies are included in this work: isotropic and anisotropic simulations. On both of them, continuous and categorical scenarios are presented, using sequential gaussian and sequential indicator simulation to generate 3D images.

For these scenarios, four different conditioning sampled datasets were used:

1. Real continuous 3D mining diamond drillhole samples with information of 2376 points with copper grades described by Serrano et al. [1998] (Figure 3.6).

2. Synthetic categorical 3D dataset of 3000 points with 10 categories generated by truncation of a convoluted Gaussian kernel with a white noise random field according to the procedure described by Peredo et al. [2016] (Figure 3.7).

3. Synthetic continuous 3D dataset of 17280 points with a cylindrical spatial distribution (Figure 3.8). Additionally, a 3D swiss-roll-like LVA field can be attached.

4. Real categorical 3D mining lithology dataset with 4 categories, using the same 2376 points as in the previous dataset (Figure 3.9). Additionally, a 3D folded LVA field can be attached.

Datasets 1 and 2 are reviewed in detail in Chapter 4, and datasets 3 and 4 are reviewed in detail in Chapter 5.

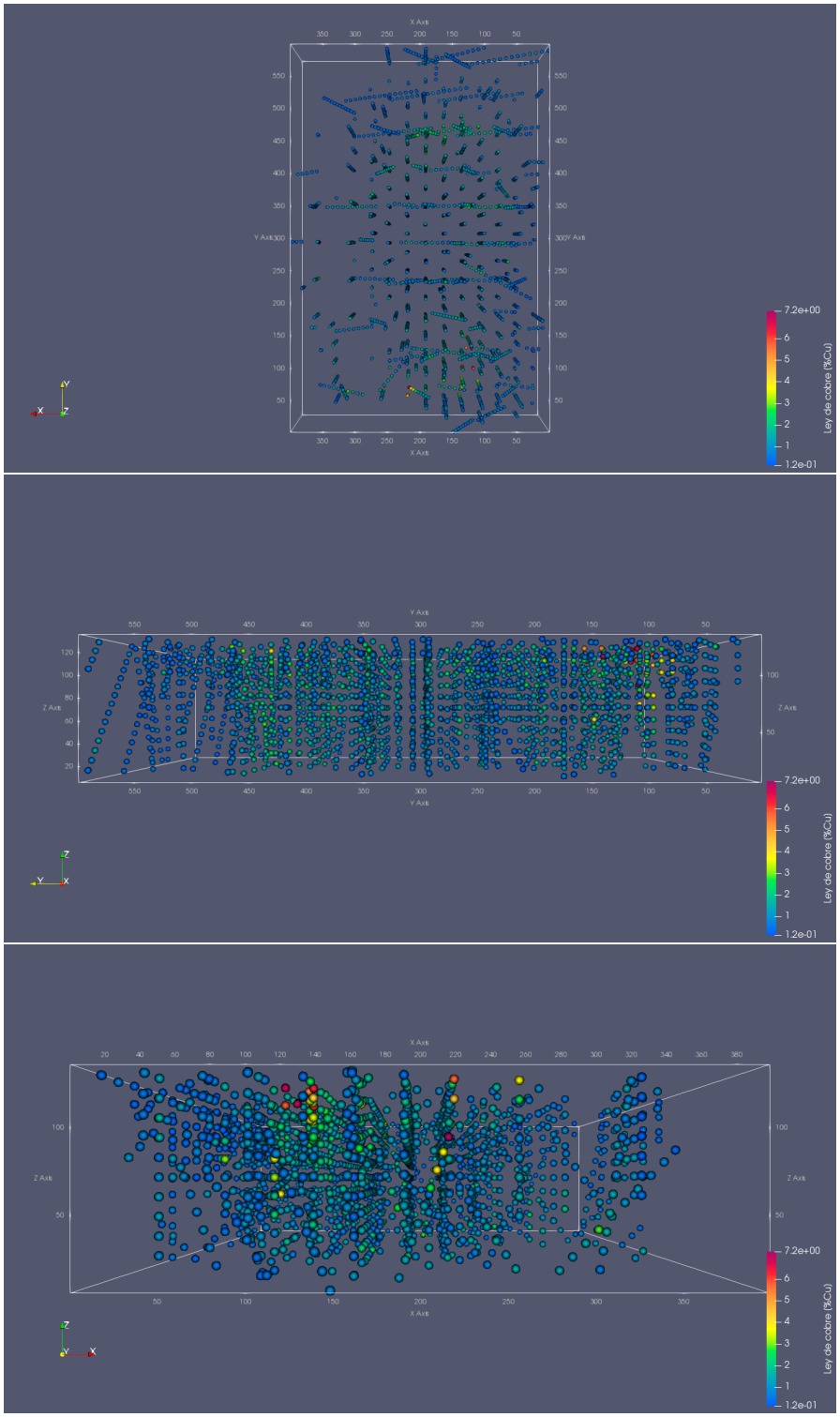

FIGURE 3.6: Conditioning sampled data from real continuous 3D mining diamond drillholes with copper grades.

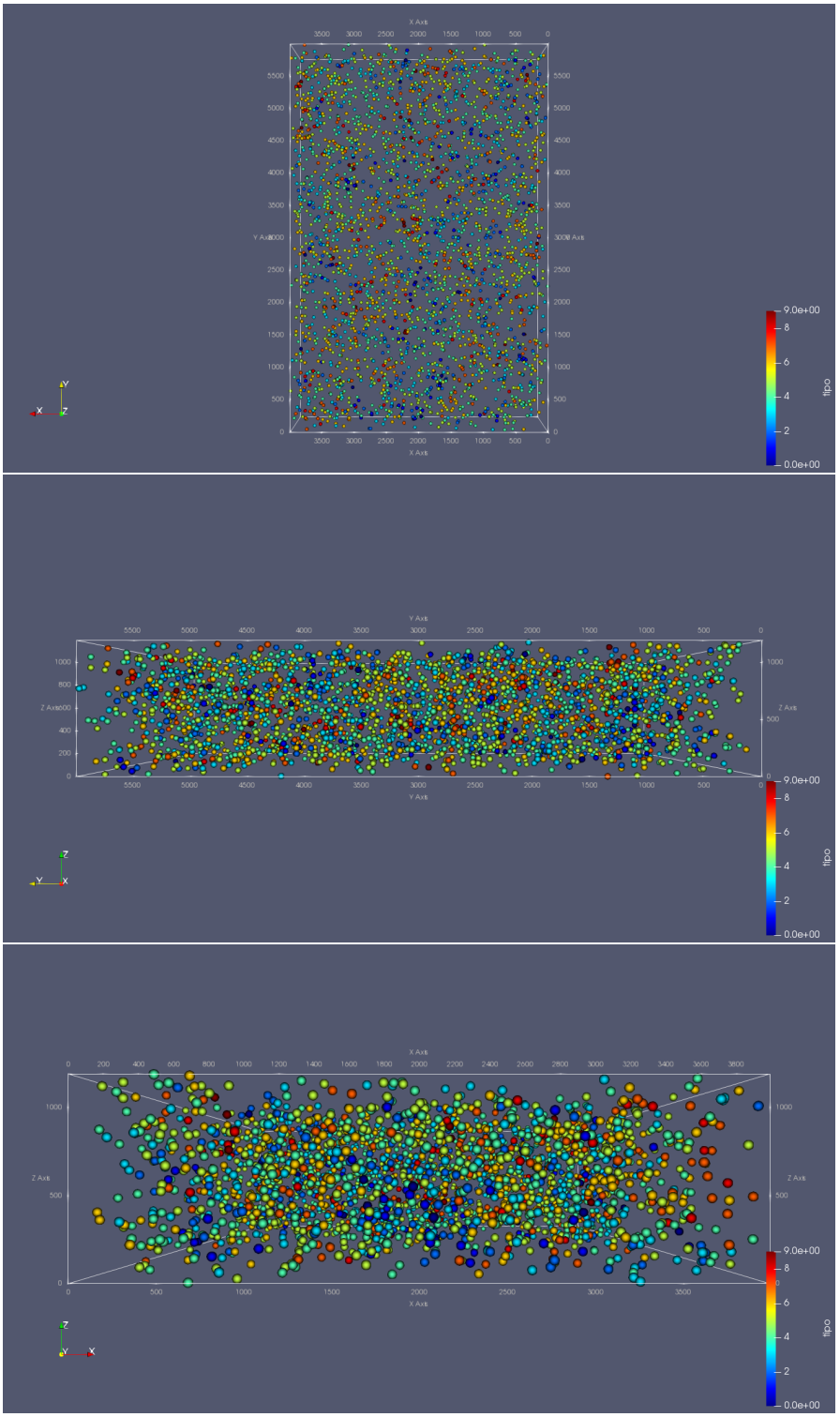

FIGURE 3.7: Conditioning sampled data from synthetic categorical 3D dataset with 10 categories.

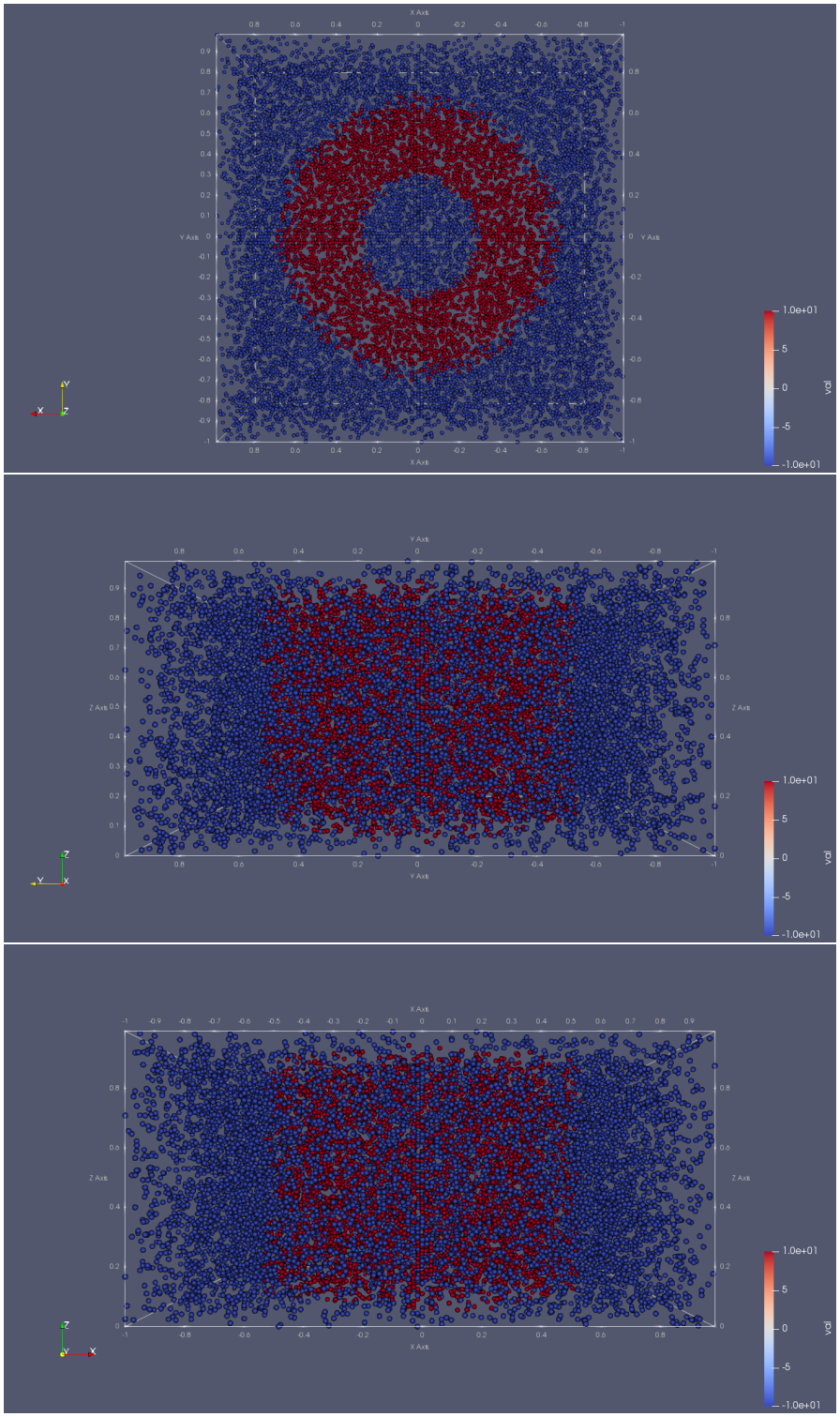

FIGURE 3.8: Conditioning sampled data from synthetic continuous 3D dataset with cylindrical spatial distribution.

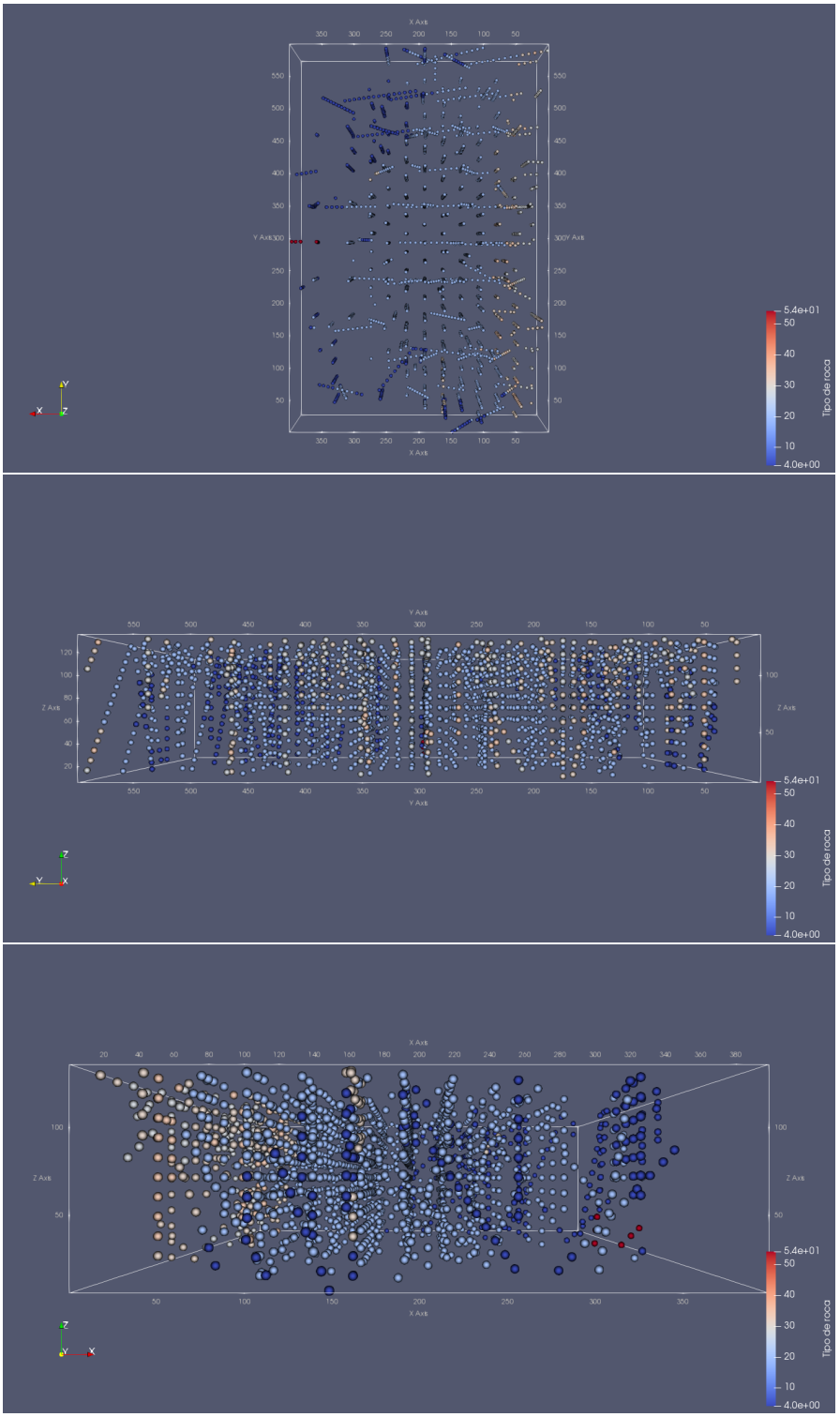

FIGURE 3.9: Conditioning sampled data from real categorical 3D mining diamond drillholes with lithology types.

3.5 Metrics

For each code optimization or parallelization applied, a set of metrics should be calculated based on the execution of the modified code. Elapsed time (real-time clock), speedup compared against baseline executions and accuracy of the results are measured each time. The results are displayed in tables or figures, depending on the amount of information that is being reported.

In some tests, serial and parallelizable parts of the code are measured (t_{ser} and t_{par}) on the corresponding executions. With the percentage of serial time, an estimate of the maximum theoretical speedup can be obtained using the following formula:

$$\begin{aligned} speedup(P) &= \frac{t_{total}}{t_{ser} + \frac{t_{par}}{P}} \\ &= \frac{1}{f + \frac{1-f}{P}} \text{ , with } f = \frac{t_{serial}}{t_{total}} \text{ (serial fraction)} \\ max_{speedup} &= \lim_{P \to \infty} \frac{1}{f + \frac{1-f}{P}} \\ &= \frac{1}{f} \end{aligned}$$

where P is the number of running processes or threads. The efficiency of a parallelization using P running processes or threads is defined as

$$efficiency(P) = \frac{speedup(P)}{P} \tag{3.2}$$

If the efficiency is small, the obtained speedup is not optimal, since the usage of the P processes or threads is not achieving the peak performance ($efficiency(P) \approx 1$).

Regarding accuracy of the results, two groups of metrics are used. The first group measures differences in continuous values, such as those generated by `sgsim`. The most important metrics in this group are the mean absolute error (MAE), mean absolute percentage error (MAPE) and mean square error (MSE):

$$MAE(\mathbf{Z}, \mathbf{Z}^*) = \frac{1}{n} \sum_{i=1}^{n} |z_i - z_i^*| \tag{3.3}$$

$$MAPE(\mathbf{Z}, \mathbf{Z}^*) = 100 \times \frac{1}{n} \sum_{i=1}^{n} \left| \frac{z_i - z_i^*}{z_i^*} \right| \tag{3.4}$$

$$MSE(\mathbf{Z}, \mathbf{Z}^*) = \frac{1}{n} \sum_{i=1}^{n} (z_i - z_i^*)^2 \tag{3.5}$$

The second group measure differences in categorical values, such as those generated by `sisim`. In this group, binary classification metrics are used, such as the true positive

rate (TPR) and false negative rate (FNR) for each category k:

$$TPR(\mathbf{Z}, \mathbf{Z}^*, k) = \frac{|\{i : z_i = z_i^* \wedge z_i^* = k\}|}{|\{i : z_i^* = k\}|} \tag{3.6}$$

$$FNR(\mathbf{Z}, \mathbf{Z}^*, k) = 1 - TPR(\mathbf{Z}, \mathbf{Z}^*, k) \tag{3.7}$$

And their similar representations in multiclass scenarios, multiclass true positive rate (MTPR) and multiclass false negative rate (MFNR):

$$MTPR(\mathbf{Z}, \mathbf{Z}^*) = \frac{\sum_{k=1}^{K} |\{i : z_i = z_i^* \wedge z_i^* = k\}|}{\sum_{k=1}^{K} |\{i : z_i^* = k\}|} \tag{3.8}$$

$$MFNR(\mathbf{Z}, \mathbf{Z}^*) = 1 - MTPR(\mathbf{Z}, \mathbf{Z}^*) \tag{3.9}$$

These last metrics measure the percentage of equal values (MTPR) and the percentage if non equal values (MFNR) simulated across all points.

Chapter 4

Parallel Sequential Simulation

The first contribution of this thesis is described in the current chapter, which is related with the parallel simulation of multiple locations without breaking the sequential order imposed by the instrinsic nature of the sequential simulation algorithms. GSLIB applications `sgsim` and `sisim` are modified accordingly, and two case studies are presented, using real and synthetic 3D datasets. Results of the parallelization proposed are presented in the last section.

4.1 Context

In this section, a complementary view of the theoretical background of sequential simulation is presented. The main concept of the sequential simulation family of algorithms, such as 2, 3, 4 and 5, is based on the Bayes postulate applied to a joint probability distribution of several dependent variables [Devroye, 1986, Johnson, 1987]. The successive application of Bayes postulate to the joint probability leads to a sequential backward inference of marginals and posterior distributions, monotonically increasing the size of the prior data set as different grid nodes are randomly visited and simulated. In order to implement this approach, two key elements should be analyzed: the random path used for simulation of grid nodes, and the neighbour search window used to infer the conditional probability on each location.

4.1.1 Random path

Following ideas from [Journel and Alabert, 1989], let A_1 and A_2 be two random events with joint probability $\mathbb{P}(A_1, A_2)$. Both events can be related with continuous or categorical random variables. For example, A_1 can be the event $Z(\mathbf{u}) \leq z$ related to the random variable $Z(\mathbf{u}) \in \mathbf{R}$ depending on the spatial location $\mathbf{u} \in \mathbb{R}^3$ (e.g. copper grade concentration at location $\mathbf{u}$) and a threshold value $z \in \mathbb{R}$. Similarly, A_1 can be the event

$I(\mathbf{u}) = 1$ related to the indicator random variable $I(\mathbf{u}) \in \{0, 1\}$ (e.g. is the rock type breccia at location $\mathbf{u}$ or not?). Applying Bayes postulate, the conditional probability of event A_2 knowing that event A_1 has ocurred is given by:

$$\mathbb{P}(A_2|A_1) = \frac{\mathbb{P}(A_2, A_1)}{\mathbb{P}(A_1)} \tag{4.1}$$

being $\mathbb{P}(A_1)$ the marginal probability of A_1 and $\mathbb{P}(A_2, A_1)$ being the joint probability of events A_2 and A_1. More generally, any number of dependent events A_j, $j = 1, \ldots, N$, can be sequentially simulated using the following expression:

$$\begin{aligned} \mathbb{P}(A_N, \ldots, A_1) \;=\; & \mathbb{P}(A_N|A_{N-1}, \ldots, A_1) \\ & \times \mathbb{P}(A_{N-1}|A_{N-2}, \ldots, A_1) \\ & \ldots \\ & \times \mathbb{P}(A_3|A_2, A_1) \\ & \times \mathbb{P}(A_2|A_1) \\ & \times \mathbb{P}(A_1) \end{aligned} \tag{4.2}$$

As commented before, each sequential simulation algorithm implements a method to obtain a realization of the joint probability $\mathbb{P}(A_N, \ldots, A_1)$ by applying Equation (4.2). A first implementation element, the random path definition, as described in Section 2.2.1.6 and 2.2.1.7, is the explicit order in which the events $A_j = \{Z(\mathbf{u}_j) \leq z\}$ or $A_j = \{I(\mathbf{u}_j) = 1\}$, with $j = 1, \ldots, N$, are selected in the conditional probabilities of Equation (4.2). For instance, if a gridded domain of 2×2 location points is defined, $4! = 24$ possible visiting orders can be assigned to the random path selected. All 24 possible random paths can be viewed in Figure 4.1. By selecting one of these paths and following the visiting order, Equation (4.2) can be realized.

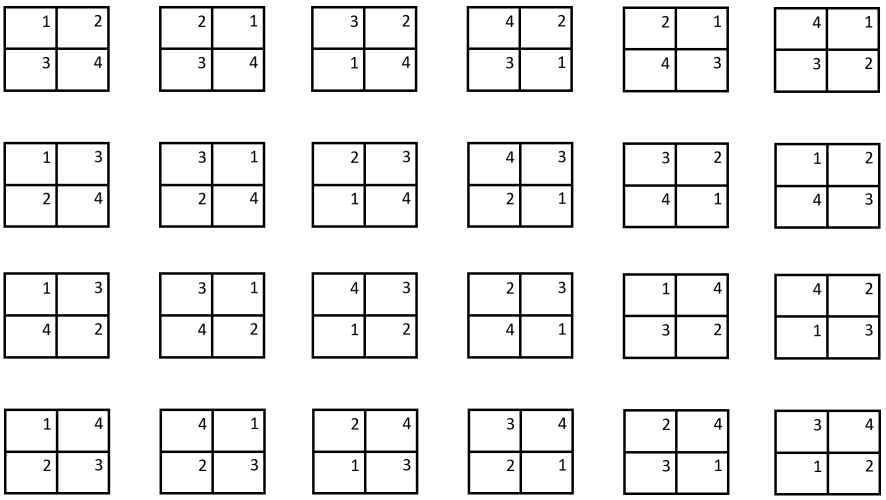

FIGURE 4.1: All possible random paths for a 4×4 gridded domain ($4! = 24$).

4.1.2 Neighbour search window

The second element is the inference of the conditional probability $\mathbb{P}(A_k|A_{k-1}, \ldots, A_1)$, for each $k = 2, \ldots, N$. In order to infer the conditional cumulative density function (ccdf), neighbouring data should be searched, which can be the initial conditioning data or previously simulated data. The neighbour search process is defined by a search window and a maximum number of neighbours to be considered. For this reason, at the begining of the simulation process, only initial conditioning data is used to infer the ccdf in the location being visited. As the simulation progresses, each new simulated location is now considered as a potential conditioning data for the next locations to be simulated.

In this context, four possible data dependency scenarios can happen between two different nodes in the grid, as depicted in Figure 4.2 using a neighbour search window of 3×3. Scenario A happens when there are no common nodes between them, which means that their simulation can be computed simultaneously if all previous nodes have been simulated, according to the random path order. Scenario B happens when both nodes share at least one neighbour, and that neighbour should be simulated before them. This scenario is similar to scenario A, with the additional constraint that both nodes should be simulated after the common neighbour node. Scenario C is similar to scenario B, but in this case the common neighbour should be simulated after them. In this case, the same constraint as scenario B will apply only if it is an initial conditioning data. Finally, in scenario D each node is a neighbour of the other node, so the successor node, which will be visited later in the random path, should wait until the previous node is simulated.

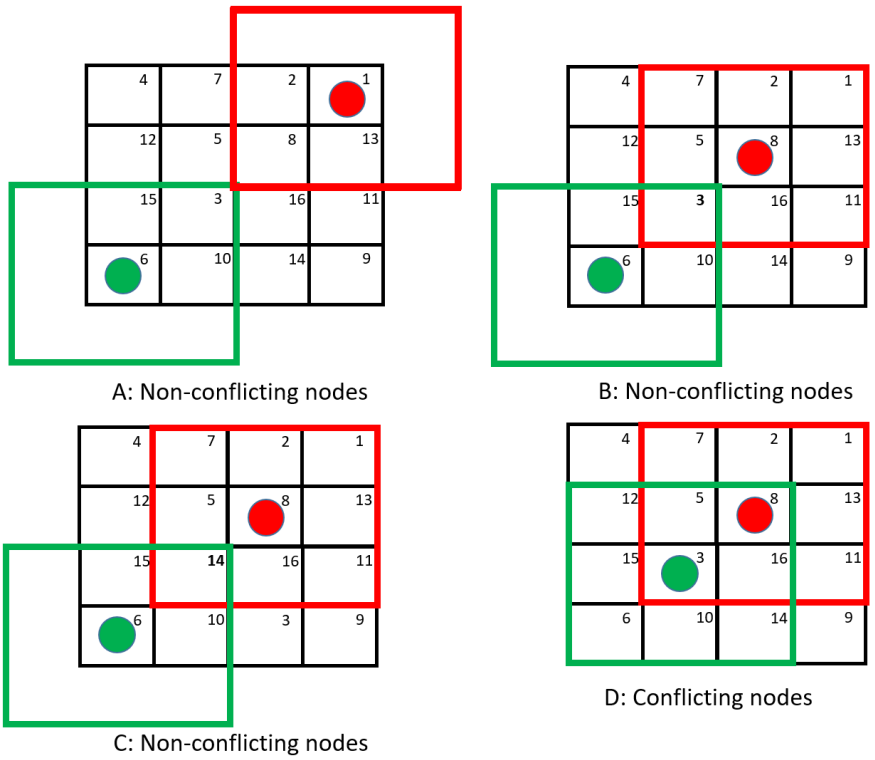

FIGURE 4.2: Four possible scenarios of data dependency in a 4×4 gridded domain with a specific random path for node visiting. A: Non-conflicting nodes. B: Non-conflicting nodes with a common predecessor. C: Non-conflicting nodes with a common successor. D: Conflicting nodes.

With these scenarios in place, the proposed algorithm for parallelization of the sequential simulation algorithm is described in the next section. The key idea is to run an initial pass through the random path, identifying data dependencies between nodes, and assign a "level" to each of them. Afterwards, each node in the same level can be simulated in parallel, because no data dependencies will arise between them.

4.2 Algorithm

The parallel version of the sequential simulation method is presented in Algorithm 9. It is based on two stages. The first one related to node tagging in order to group all nodes with non conflicting neighbourhoods. The second stage is the actual simulation, similarly to the single-thread algorithm, with a different node loop formulation but the same neighbourhood data and simulation method. The pseudo-routines, with their steps detailed, are `search_neighbours_push` (Algorithm 10), `build_level` (Algorithm 11), `order_nodes_by_level` (Algorithm 12) and `search_neighbours_pop` (Algorithm 13).

Regarding the first stage, in steps 7 and 8 of Algorithm 9 two arrays are defined, **Level** and **Neighbours**, which will store the level tags and neighbours information

Input:
$(\mathbf{V}, \Omega)$: sample database values defined in a 3D domain;
κ: kriging parameters (radius, max number of neigbours and others);
τ: seed for pseudo-random number generator;
N: number of generated simulations;
`output.txt`: output file name;
T: number of threads

```
1  n ← obtain_max_neighbour_number(κ)
2  for isim ∈ {1, ..., N} do
3      P ← create_random_path(Ω, τ)
4      V^tmp ← zeros(|Ω| × 1)
5      V^tmp ← assign(V) //Sample data assignment
6      //Stage 1
7      Level ← zeros(|Ω| × 1)
8      Neighbours ← zeros(|Ω| × n × 2) //store the neighbours local and global index
9      for ixyz ∈ {1, ..., |Ω|} do
10         index ← P_ixyz
11         Neighbours(index) ← search_neighbours_push(index, κ)
12         Level(index) ← build_level(index, κ, Neighbours)
13     end
14     IndexSort, LevelCount, LevelStart ← order_nodes_by_level(Level)
15     //Stage 2
16     for lev ∈ {1, ..., max(Level)} do
17         lbegin ← LevelStart(lev)
18         lend ← LevelStart(lev) + LevelCount(lev) − 1
19         for ixyz ∈ {lbegin, ..., lend} in parallel with T threads do
20             index ← IndexSort(P_ixyz)
21             LocalNeighbours ← search_neighbours_pop(index, κ, Neighbours)
22             V^tmp(index) ← simulate(index, LocalNeighbours)
23         end
24     end
25     write(output.txt, V^tmp)
26 end
```

Output: N stochastic simulations stored in file `output.txt`

Algorithm 9: Pseudo-code sequential simulation program (multi-thread algorithm)

(local and global indices). A first pass through all nodes is performed between steps 9 and 13. The simulation random path is walked sequentially, scanning for neighbours around the current node and storing basic information about them in the pseudo-routine `search_neighbours_push` (Algorithm 10). This routine is the same as in the original neighbour search from GSLIB, with the only difference that, instead of actually calculating the coordinates and other information about the neighbours, it only stores the neighbours indices by *pushing* (copying) them into the array **Neighbours**. After the neighbours have been calculated, a level tag is assigned to the current node according to the pseudo-routine `build_level` which scans for the maximum of all neighbours' level tags and adds 1 to that value (Algorithm 11). Initially all nodes with conditioning data are assigned with level tag 0 and non informed nodes are assigned with level tag -1. With this initial assignment, nodes with some conditioning data inside their search window are assigned with level tag 1, nodes with a level 1 neighbour are assigned with level

tag 2, and so on. The last part of the first stage is at step 14 of Algorithm 9, where the pseudo-routine `order_nodes_by_level` performs a rearrangement procedure, storing the indices of the new order in the array **IndexSort**, and the number of nodes and initial index per level in the arrays **LevelCount** and **LevelStart** (Algorithm 12).

In Figure 4.3 an example of the level assignment is presented using a search lookup window of size 3×3. Initially the conditioning data nodes are placed in the locations 6, 13, 15, 18, 24 and 25 of the random path (value in the top-right of each grid cell). The level tag for those conditioning nodes is zero. Starting the assignment, the node in location 1 is visited resulting in a level tag assignment of 1, since in its search lookup window of 3×3 there are only nodes with level tags of 0 (neighbours in locations 13, 15 and 24). Similarly, nodes in locations 2, 3 and 4 are assigned with level tag 1. Node in location 5 is assigned with level tag 2, since in its search lookup window a neighbour with level tag 1 is located (neighbour in location 4). Node in location 7 is assigned with level tag 1 (neighbour in locations 6 and 25 with level 0), and node in location 8 is assigned with level tag 3 (neighbour in location 5 with level 2), and so on.

The second stage of Algorithm 9 involves the simulation in parallel of all nodes in the same level, since no data dependencies arise between those nodes. For each level, as shown in step 16, the initial and final indices are calculated, `lbegin` and `lend` respectively in steps 17 and 18. The index of the node to be simulated is obtained in step 20 using the re-ordered array **IndexSort**. In step 21 the pseudo-routine `search_neighbours_pop` (Algorithm 13) is called, which essentially is a query to extract local neighbour indices from the array **Neighbours**, previously stored by using `search_neighbours_push` in step 11 (Algorithm 10). With the local neighbour indices, the coordinates are computed for each neighbour, and the simulation can be performed in step 22 with the pseudo-routine `simulate`, which can be a mix of kriging, back-transformations and pseudo-random sampling, according to the simulation method used. Specifically, the steps for sequential gaussian simulation method are defined in lines 9, 10 and 11 from Algorithm 2, and the steps for sequential indicator simulation are defined in lines 7 to 10 from Algorithm 3.

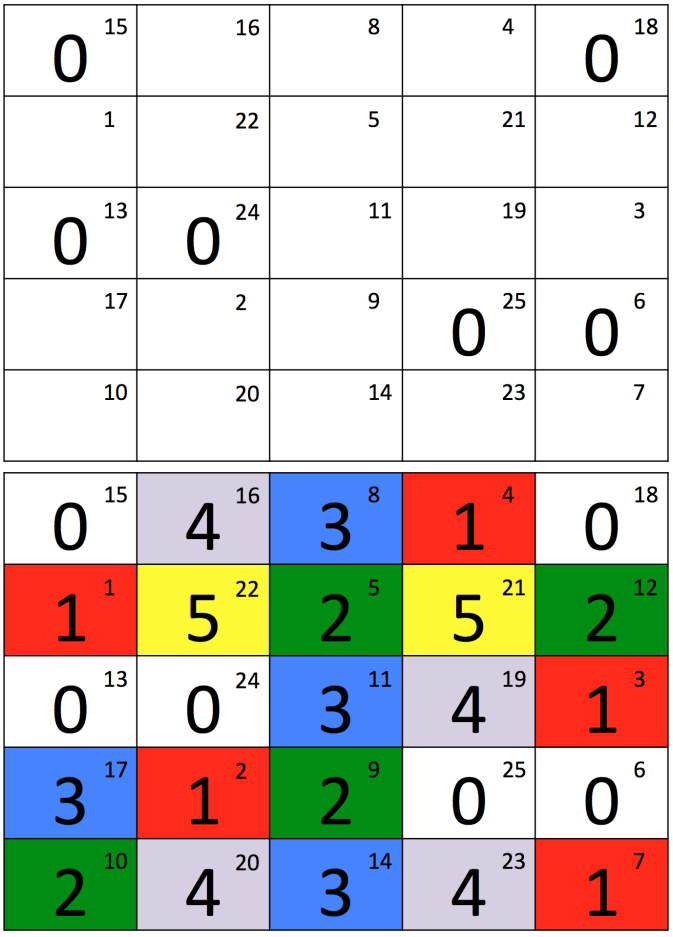

FIGURE 4.3: Top: Random path index (top-right corner or each cell) and initial assignment of level tags (only zeros for nodes with conditioning data). Bottom: Final assignment of level tags, with different color for different levels. The search lookup window in this example is a 3×3 square centered in the node of interest. By walking through the random path and scanning the max level tag in each window, adding 1 to it, the final assignment of levels can be obtained.

In this work, OpenMP directives are included into the modified code. A synchronization method must be used in order to keep the order of the levels being processed, since threads can spend different time in the simulation of their assigned nodes, causing race conditions when accessing neighbour values that are currently being simulated or not simulated yet. A first alternative is to use the implicit OpenMP barrier declared at the end of a parallel loop region. Since this barrier adds a major overhead to the parallelization, a second alternative was chosen based on lock variables that control when all neighbour nodes of a node being simulated are available (have a defined value). A pseudo-code of this strategy is depicted in Algorithm 14, using an extra shared array **Lock** with size $|\Omega|$ and values 1 or 0 indicating if the corresponding grid node has been simulated or not. As the neighbour node indices are collected, each thread waits until all neighbours have lock value **Lock**(`i`)=1, in order to get out of the waiting loop and continue with the simulation steps.

Input:
`index`: grid node index;
κ: local interpolation parameters;
Neighbours(`index`): array with neighbour indices for grid node `index`;

1 $m \leftarrow$ `obtain_search_window_size`(κ)
2 $n \leftarrow$ `obtain_max_neighbour_number`(κ)
3 **Offset** $\leftarrow$ `obtain_spiral_search_offsets`(κ) // *distances between a centering location and each other location inside the search window, ordered by a spiral path*
4 `numberOfLocalNeighbours` $\leftarrow 0$
5 **for** *ind* $\in \{1, \ldots, m\}$ **do**
6 **If** `numberOfLocalNeighbours` $\geq n$ **then Return**
7 $(i, j, k) \leftarrow (\texttt{index}_x, \texttt{index}_y, \texttt{index}_z) +$ **Offset**(ind) // *spiral node visiting centered in node* `index`
8 **if** (i, j, k) *is inside the domain* **then**
9 `indexglobal` $\leftarrow i + (j-1) * nx + (k-1) * nx * ny$
10 **if** *node* ***indexglobal*** *has been previously simulated* **then**
11 **Neighbours**(`index`) $\leftarrow \{$`ind`, `indexglobal`$\}$ //*local and global indexes*
12 `numberOfLocalNeighbours` $\leftarrow$ `numberOfLocalNeighbours` $+ 1$
13 **end**
14 **end**
15 **end**

Algorithm 10: Pseudo-routine `search_neighbours_push`. In this case, a spiral search is being described, according to Deutsch and Journel [1998], Chapter II.4, depicted in Figure 5.1.

Input:
`index`: grid node index;
κ: local interpolation parameters;
Neighbours: array with neighbour indices of all domain nodes;
Level: array with level tags of all domain nodes;

```
// Obtain the number of valid neighbours for node index
numberOfLocalNeighbours ← obtain_total_neighbour_number(Neighbours(index))
maxLevel ← −1
Level(index) = 0
for ind ∈ {1, . . . , numberOfLocalNeighbours} do
    if Level(Neighbours(index, ind, 1)) > maxLevel then
        maxLevel ← Level(Neighbours(index, ind, 1)) //obtain level of neighbour using global index
    end
end
Level(index) = maxLevel + 1
```

Output: **Level**(`index`): level assigned to grid node `index`

Algorithm 11: Pseudo-routine `build_level`

Input:
Level: array with level tags of all domain nodes;
IndexSort, **LevelCount**, **LevelStart**: arrays with the reordering of node visits using the level tag as grouping identifier;

```
numberOfLevels ← max(Level)
count ← 0
lastCount ← 0
LevelCount ← zeros((numberOfLevels + 1) × 1)
LevelStart ← zeros((numberOfLevels + 1) × 1)
for lev ∈ {0, . . . , numberOfLevels} do
    LevelStart(lev + 1) ← count + 1
    LevelCount(lev + 1) ← 0
    for ixyz ∈ {1, . . . , |Ω|} do
        if Level(ixyz) == lev then
            count ← count + 1
            IndexSort(count) ← ixyz
            LevelCount(lev + 1) ← LevelCount(lev + 1) + 1
        end
    end
end
```

Algorithm 12: Pseudo-routine `order_nodes_by_level`

Input:
`index`: grid node index;
κ: local interpolation parameters;
Neighbours: array with neighbour indices

```
// Obtain the number of valid neighbours for node index
numberOfLocalNeighbours ← obtain_total_neighbour_number(Neighbours(index))
for ind ∈ {1, . . . , numberOfLocalNeighbours} do
    indexglobal ← Neighbours(index, ind, 0) //indexglobal is neighbour global index
    LocalNeighbours(index, ind) ← get_coordinates(indexglobal)
end
```

Output: **LocalNeighbours**: neighbour coordinates of grid node `index`.

Algorithm 13: Pseudo-routine `search_neighbours_pop`

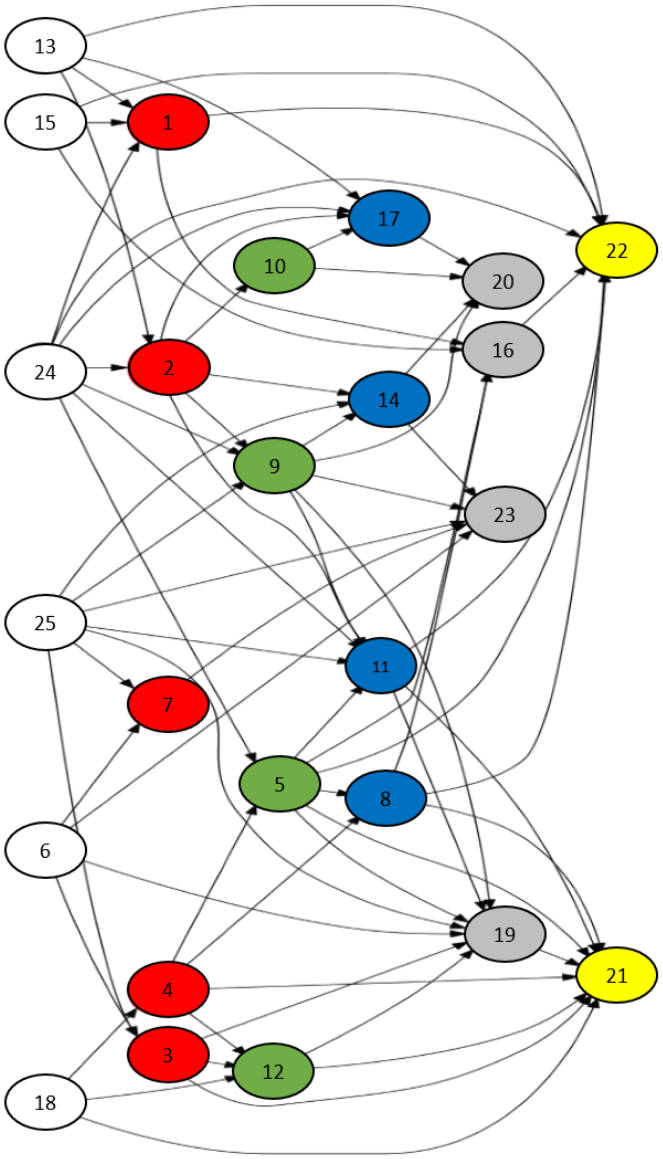

FIGURE 4.4: Data dependency graph associated with the level tags and neighbour relationships (follow-up of Figure 4.3). Left-most nodes correspond to level 0 (conditioning nodes), right-most nodes correspond to level 5.

```
1  for lev ∈ {1, ..., max(Level)} do
2  |  lbegin ← LevelStart(lev)
3  |  lend ← LevelStart(lev) + LevelCount(lev) − 1
4  |  for ixyz ∈ {lbegin, ..., lend} in parallel do
5  |  |  LocalNeighbours ← spiral_search_neighbours_pop(P_ixyz, κ, Neighbours)
6  |  |  // Wait until all local neighbours are ready with a defined value, Lock is shared array
7  |  |  ilock ← 0
8  |  |  while ilock == 0 do
9  |  |  |  ilock ← 1
10 |  |  |  for i ∈ {1, ..., numberOfLocalNeighbours} do
11 |  |  |  |  ilock ← ilock * Lock(Neighbours(P_ixyz, i, 1)) //global index
12 |  |  |  end
13 |  |  end
14 |  |  // Proceed to simulate the current node
15 |  |  V^tmp(P_ixyz) ← simulate(P_ixyz, LocalNeighbours)
16 |  |  Lock(P_ixyz) = 1 //Mark this node as unlocked for all other nodes waiting for it
17 |  end
18 end
```

Algorithm 14: Pseudo-code for thread synchronization using locks.

4.3 Results

In this section, two cases are presented, including their execution times and speedup. The base codes are `sgsim` and `sisim` from GSLIB, developed by Deutsch and Journel [1998], and posteriorly code-optimized by Peredo et al. [2015a].

All runs were executed in a single-node machine with Ubuntu 14.04.5 LTS with 2×8-cores Intel(R) Xeon(R) CPU E5-2673 v3 at frequency 2.40GHz, and a memory hierarchy of 116GB RAM, 30MB L3 cache, 256KB L2 cache and 32KB/32KB L1d/L1i caches. All programs were compiled using GCC `gfortran` version 4.8.4 supporting OpenMP version 3.1, with the options `-O2 -march=native -ffast-math -ftree-vectorize` in all cases and `-fopenmp` in the multi-thread executions. All results are the average value of 5 runs, in order to reduce external factors in the measurement.

4.3.1 `sgsim`

The case study for the parallel `sgsim` code uses a real mining 3D dataset of 2376 diamond drill-hole samples with information of copper grade composites. In Figure 4.5 a realization sample is depicted, with standardized values simulating the copper grades. Table 4.1 contains all relevant parameters, such as grid sizes, search lookup windows and variographic parameters. The local interpolation method is Ordinary Kriging.

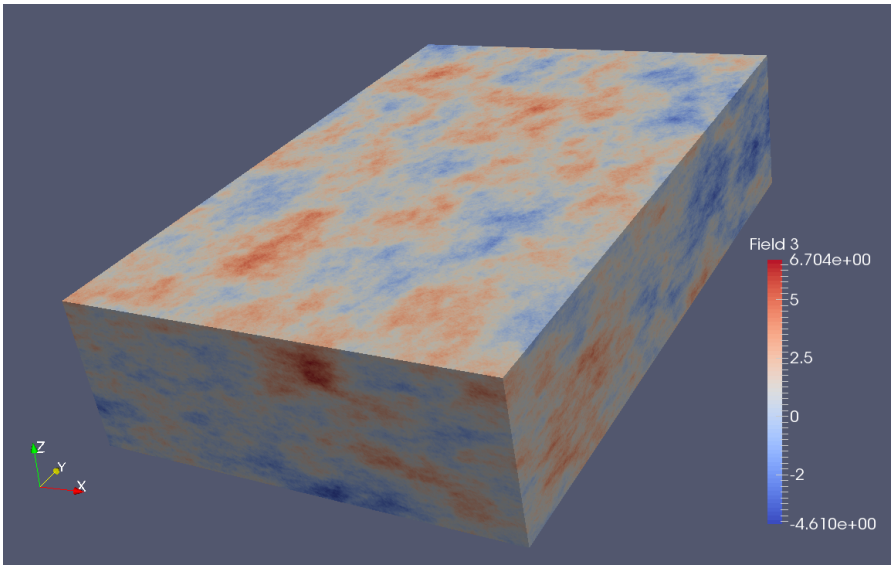

FIGURE 4.5: Realization sample of the SGSIM case study.

Parameter	Values
$nx \times ny \times nz$	$\{800 \times 800 \times 160, 400 \times 800 \times 160\}$
$xsiz, ysiz, zsiz$	$0.5, 0.75, 0.9$
max data for kriging	$\{16, 32, 64, 128\}$
max search radii	$300, 300, 300$
size of covariance lookup table	$201 \times 201 \times 201$
number of structures and type	3,{spherical,exponential,gaussian}

TABLE 4.1: Parameters for SGSIM case study: grid sizes, search lookup window and variography for all categories (parameter description can be reviewed in Section 3.2).

The results are depicted in Tables 4.2 and 4.4 for the larger grid size with $800 \times 800 \times 160$ nodes ($102,400,000$ nodes), and 4.5 and 4.7 for the smaller grid size with $400 \times 800 \times 160$ nodes ($51,200,000$ nodes). We can observe in the speedup results that as the number of maximum kriging neighbours increases, the achieved speedup using 16 threads also increases. A key factor for this behaviour is related with amount of work that should be done in the simulation part (lines 16 to 24 of Algorithm 9), which increases if a larger maximum kriging neighbour value is used as parameter. As counterpart, the neighbour calculation part (lines 7 to 14 of Algorithm 9), not parallelized, decreases its proportional contribution to the overall elapsed time. The percentage of neighbour calculation time using the larger grid is approximately 39.5%, 19.8%, 7.1% and 2.1% with 16, 32, 64 and 128 maximum kriging neighbours (Table 4.3). By using the smaller grid, this percentage is approximately 39.1%, 21.6%, 7.2% and 2.2% respectively (Table 4.6). The neighbour calculation is the most relevant contribution of the part of code that remains sequential, which impacts in the maximum theoretical speedup that can be obtained. This topic will be discussed extensively in Chapter 5.

# Threads	Elapsed time [s] 16 neigs	Elapsed time [s] 32 neigs	Elapsed time [s] 64 neigs	Elapsed time [s] 128 neigs
1 (gslib)	766.574	2743.532	14795.288	106393.643
1 (omp)	811.090	3036.122	16337.288	109554.268
2 (omp)	588.970	1823.014	8719.349	51280.557
4 (omp)	522.970	1217.604	5116.514	27401.376
8 (omp)	381.586	918.545	3235.145	15561.031
16 (omp)	362.666	743.966	2278.863	9175.207

TABLE 4.2: Execution time of SGSIM with 102,400,000 grid nodes and maximum of 16, 32, 64 and 128 neighbours to infer conditional probability.

# Threads	Neighbour calculation time [%] 16 neigs	Neighbour calculation time [%] 32 neigs	Neighbour calculation time [%] 64 neigs	Neighbour calculation time [%] 128 neigs
1 (omp)	39.5	19.8	7.0	2.2
2 (omp)	55.2	32.7	13.5	4.7
4 (omp)	59.5	48.5	23.1	8.7
8 (omp)	81.3	65.5	36.7	15.7
16 (omp)	89.8	78.7	51.8	26.4

TABLE 4.3: Percentage of execution time of non-parallel neighbour calculation of SGSIM with 102,400,000 grid nodes and maximum of 16, 32, 64 and 128 neighbours to infer conditional probability.

# Threads	Speedup 16 neigs	Speedup 32 neigs	Speedup 64 neigs	Speedup 128 neigs
1 (gslib)	1.00	1.00	1.00	1.00
1 (omp)	0.94	0.90	0.90	0.97
2 (omp)	1.30	1.50	1.69	2.07
4 (omp)	1.46	2.25	2.89	3.88
8 (omp)	2.00	2.98	4.57	6.83
16 (omp)	2.11	3.68	6.49	11.59

TABLE 4.4: Speedup of SGSIM with 102,400,000 grid nodes and maximum of 16, 32, 64 and 128 neighbours to infer conditional probability.

# Threads	Elapsed time [s] 16 neigs	Elapsed time [s] 32 neigs	Elapsed time [s] 64 neigs	Elapsed time [s] 128 neigs
1 (gslib)	393.929	1453.179	7921.930	51726.258
1 (omp)	413.479	1497.249	7908.930	48939.258
2 (omp)	299.263	888.965	4148.421	26576.200
4 (omp)	235.154	600.941	2382.522	14121.606
8 (omp)	204.316	448.859	1474.447	7574.541
16 (omp)	190.589	371.572	989.462	4328.232

TABLE 4.5: Execution time of SGSIM with 51,200,000 grid nodes and maximum of 16, 32, 64 and 128 neighbours to infer conditional probability.

# Threads	Neighbour calculation time [%] 16 neigs	Neighbour calculation time [%] 32 neigs	Neighbour calculation time [%] 64 neigs	Neighbour calculation time [%] 128 neigs
1 (omp)	39.1	19.6	6.8	2.1
2 (omp)	55.4	32.4	12.8	4.0
4 (omp)	70.5	48.0	22.3	7.8
8 (omp)	81.9	64.8	36.0	14.4
16 (omp)	87.1	78.1	52.1	25.3

TABLE 4.6: Percentage of execution time of non-parallel neighbour calculation of SGSIM with 51,200,000 grid nodes and maximum of 16, 32, 64 and 128 neighbours to infer conditional probability.

# Threads	Speedup 16 neigs	Speedup 32 neigs	Speedup 64 neigs	Speedup 128 neigs
1 (gslib)	1.00	1.00	1.00	1.00
1 (omp)	0.95	0.97	1.00	1.05
2 (omp)	1.31	1.63	1.90	1.94
4 (omp)	1.67	2.41	3.32	3.66
8 (omp)	1.92	3.23	5.37	6.82
16 (omp)	2.06	3.91	8.00	11.95

TABLE 4.7: Speedup of SGSIM with 51,200,000 grid nodes and maximum of 16, 32, 64 and 128 neighbours to infer conditional probability.

4.3.2 `sisim`

A minor modification must be done in the base code **`sisim`**, in order to run simulations on large domains ($> 2^{24} = 16,777,216$ nodes). The array that stores the random path visiting order, denoted **order**, is defined originally as a `real` structure. Since `real` is a single-precision floating point representation, the maximum integer value that can be represented with this data type is 2^{24}, since the size of the significant precision bits is 24 [IEEE, 2008]. By changing the data type of **order** to `integer`, a maximum of $2^{32} - 1 = 2,147,483,647$ nodes can be achieved.

The case study for the parallel **`sisim`** code uses a synthetic 3D dataset of 3000 random samples with 10 categories generated by truncation of a convoluted Gaussian kernel with a white noise random field according to the procedure described by Peredo et al. [2016] (Figure 4.6 shows a realization using three categories). Table 4.8 contains all relevant parameters, such as grid sizes, search lookup windows and variographic parameters. In all cases the method of local interpolation was simple kriging, with the option *full indicator kriging* active.

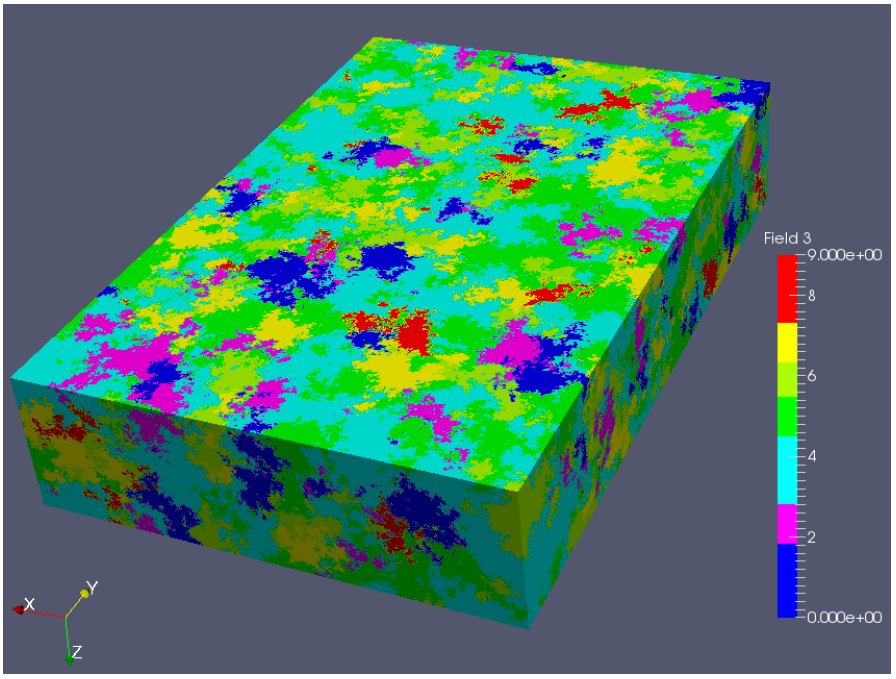

FIGURE 4.6: Realization sample of the SISIM case study.

Parameter	Values
$nx \times ny \times nz$	$\{420 \times 600 \times 400, 210 \times 600 \times 400\}$
$xsiz, ysiz, zsiz$	$9.5, 10.0, 3.0$
max data for kriging	$\{16, 32, 64, 128\}$
max search radii	∞, ∞, ∞
size of covariance lookup table	$51 \times 51 \times 166$
number of categories	10
number of structures and type	10,{spherical}

TABLE 4.8: Parameters for SISIM case study: grid sizes, search lookup window and variography for all categories (parameter description can be reviewed in Section 3.2)

The results are depicted in Tables 4.9 and 4.11 for the larger grid size with $420 \times 600 \times 400$ nodes ($100,800,000$ nodes), and 4.12 and 4.14 for the smaller grid size with $210 \times 600 \times 400$ nodes ($50,400,000$ nodes). Similarly to the SGSIM case, as the maximum number of neighbours for kriging increases, the achieved speedup using 16 threads also increases since the amount of work in the simulation part increases, and as consequence the percentage of neighbour calculation time decreases proportionally. Using the larger grid, the percentage is approximately 8.4%, 3.2%, 1.1% and 0.2% with 16, 32, 64 and 128 maximum kriging neighbours. By using the smaller grid, the percentage is approximately 8.9%, 3.4%, 0.9% and 0.2% respectively. These percentage of neighbour calculation are considerably lower that these percentages of the SGSIM case, which indicates that the speedup results obatined are higher.

# Threads	Elapsed time [s] 16 neigs	Elapsed time [s] 32 neigs	Elapsed time [s] 64 neigs	Elapsed time [s] 128 neigs
1 (gslib)	4755.103	22560.543	140548	1017870
1 (omp)	4522.103	21325.543	137425.264	1001197.200
2 (omp)	2533.220	11251.314	71605.673	511434.417
4 (omp)	1501.155	6221.798	36470.465	252730.204
8 (omp)	931.031	3399.888	18590.755	125209.547
16 (omp)	718.128	2332.582	10310.647	63913.105

TABLE 4.9: Execution time of SISIM with 100,800,000 grid nodes and maximum of 16, 32, 64 and 128 neighbours to infer conditional probability.

# Threads	Neighbour calculation time [%] 16 neigs	Neighbour calculation time [%] 32 neigs	Neighbour calculation time [%] 64 neigs	Neighbour calculation time [%] 128 neigs
1 (omp)	8.4	3.2	1.0	0.2
2 (omp)	15.0	5.8	1.9	0.4
4 (omp)	25.2	10.6	3.4	0.9
8 (omp)	40.6	19.4	6.7	1.8
16 (omp)	52.8	28.3	13.4	3.5

TABLE 4.10: Percentage of execution time of non-parallel neighbour calculation of SISIM with 100,800,000 grid nodes and maximum of 16, 32, 64 and 128 neighbours to infer conditional probability.

# Threads	Speedup 16 neigs	Speedup 32 neigs	Speedup 64 neigs	Speedup 128 neigs
1 (gslib)	1.00	1.00	1.00	1.00
1 (omp)	1.05	1.05	1.02	1.01
2 (omp)	1.87	2.00	1.96	1.99
4 (omp)	3.16	3.62	3.85	4.02
8 (omp)	5.10	6.63	7.56	8.12
16 (omp)	6.62	9.67	13.63	15.92

TABLE 4.11: Speedup of SISIM with 100,800,000 grid nodes and maximum of 16, 32, 64 and 128 neighbours to infer conditional probability.

# Threads	Elapsed time [s] 16 neigs	Elapsed time [s] 32 neigs	Elapsed time [s] 64 neigs	Elapsed time [s] 128 neigs
1 (gslib)	2566.105	12093.603	73424.527	490844
1 (omp)	2456.625	11347.603	70250.527	487311.691
2 (omp)	1328.791	5850.835	34813.426	255689.720
4 (omp)	793.696	3191.038	18327.576	128435.340
8 (omp)	505.963	1824.460	9629.237	64432.234
16 (omp)	362.476	1091.702	5132.127	32454.387

TABLE 4.12: Execution time of SISIM with 50,400,000 grid nodes and maximum of 16, 32, 64 and 128 neighbours to infer conditional probability.

# Threads	Neighbour calculation time [%] 16 neigs	Neighbour calculation time [%] 32 neigs	Neighbour calculation time [%] 64 neigs	Neighbour calculation time [%] 128 neigs
1 (omp)	8.9	3.4	0.9	0.2
2 (omp)	16.5	6.6	1.9	0.5
4 (omp)	27.3	12.2	3.6	0.9
8 (omp)	42.6	22.1	6.9	2.0
16 (omp)	60.0	35.1	13.1	4.0

TABLE 4.13: Percentage of execution time of non-parallel neighbour calculation of SISIM with 50,400,000 grid nodes and maximum of 16, 32, 64 and 128 neighbours to infer conditional probability.

# Threads	Speedup 16 neigs	Speedup 32 neigs	Speedup 64 neigs	Speedup 128 neigs
1 (gslib)	1.00	1.00	1.00	1.00
1 (omp)	1.04	1.06	1.04	1.00
2 (omp)	1.93	2.06	2.10	1.91
4 (omp)	3.23	3.78	4.00	3.82
8 (omp)	5.07	6.62	7.62	7.61
16 (omp)	7.07	11.07	14.30	15.12

TABLE 4.14: Speedup of SISIM with 50,400,000 grid nodes and maximum of 16, 32, 64 and 128 neighbours to infer conditional probability.

4.4 Analysis

In this section, a detailed analysis of the results is included. Regarding the SGSIM case study, the parallelization shows an increase in speedup values in the largest scenarios and underperforms in the smaller scenarios. SISIM shows higher speedup values thanks to the lower fraction of neighbour calculation which results from an increase of work in the simulation step (more kriging system solving in each node). Considering that no additional libraries or external tools were used in the parallelization (with exception of OpenMP), further gains can be achieved by reducing the serial time, which will be discussed in Chapter 5.

4.4.1 Efficiency

Figure 4.7 shows the relationship between efficiency of the parallelization (Equation (3.2)) and the maximum number of neighbours for kriging, according to the previous results for SGSIM and SISIM using 16 threads from Tables 4.4-4.7 and 4.11-4.14. As mentioned before, as the number of maximum kriging neighbours increases, the efficiency increases as well. The lower efficiency obtained in the overall SGSIM results can be explained in part by the relative small amount of computation involved in the execution of these cases, compared against the SISIM case. The number of kriging computations per node is exactly one, in contrast to SISIM where ten interpolations must be solved (ten categories to simulate). As shown in Figure 4.8, a small number of grid nodes are simulated in parallel in the first levels, which adds a large amount of overhead to thread initialization, such as shared/private variables setup. The best result in terms of efficiency for SGSIM is obtained using the largest maximum number of neighbours, 128, which is directly related to higher number of computations in the local interpolations. The efficiency obtained in all SISIM cases is higher than the SGSIM cases and can be explained by the higher amount of computation involved in the parallel step while the serial part is kept identical. As mentioned before, by using ten categories for simulation, ten local interpolation systems must be solved for each grid node. Regarding the number of grid nodes per level, since a larger number of levels contain sufficiently large number of grid nodes (Figure 4.9), high parallel efficiency values are obtained with more than 90% in almost all cases. The best result for SISIM is obtained using the largest maximum number of neighbours, 128, for the same reasons as the best SGSIM case.

In Figure 5.24 we can observe an execution profile obtained with Extrae/Paraver tools, in the SGSIM scenario using 16 threads. Each stage of the execution can be identified approximately in this figure, using the "State as is" visualization, which depicts the state of execution of each thread. As stated in Table 4.6, the largest portion corresponds to neighbour calculation (52.1%), which can be observed explicitly in the trace.

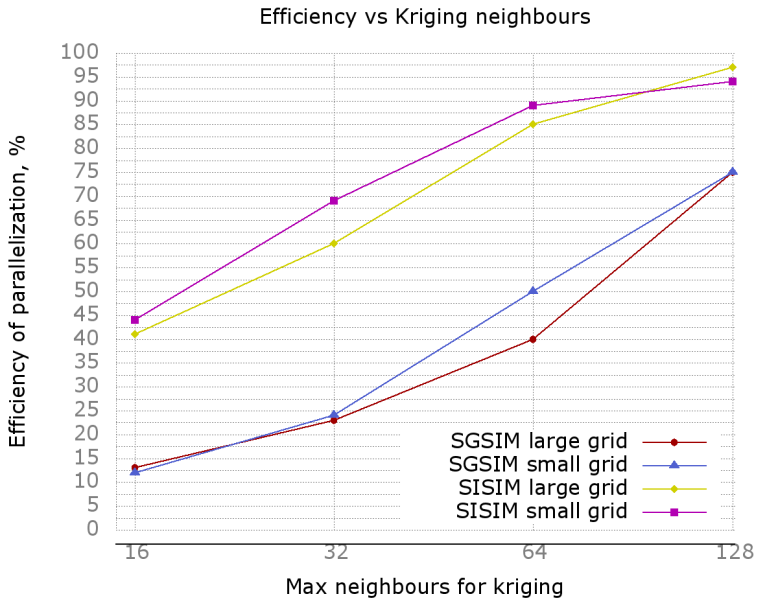

FIGURE 4.7: Relationship between efficiency of the parallelization and kriging neighbours using 16 threads in all cases.

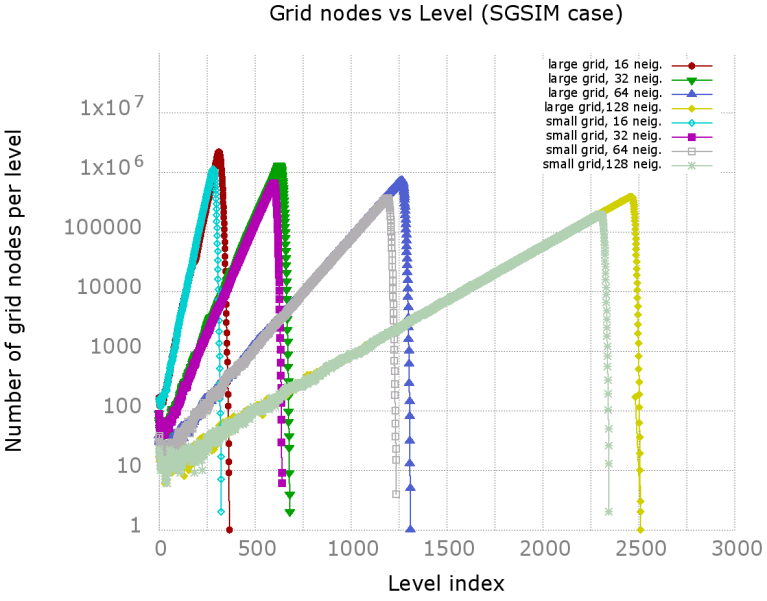

FIGURE 4.8: Number of grid nodes per level in SGSIM case.

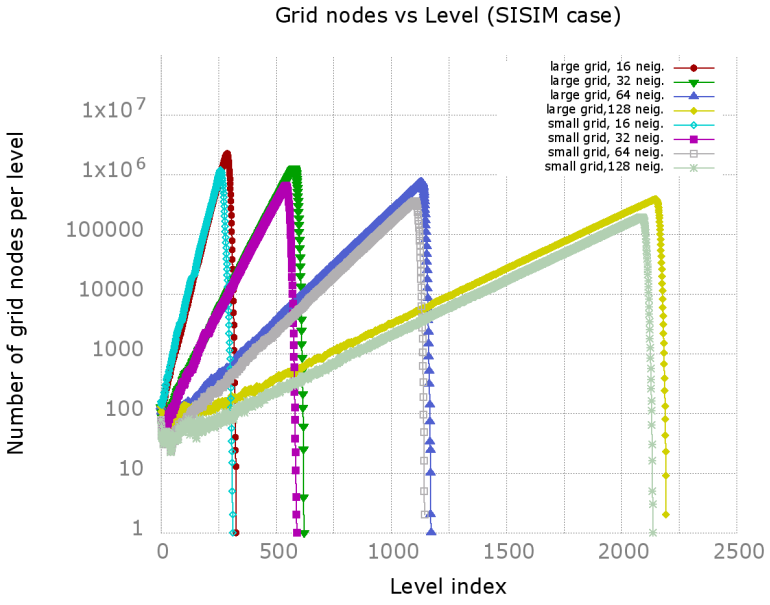

FIGURE 4.9: Number of grid nodes per level in SISIM case.

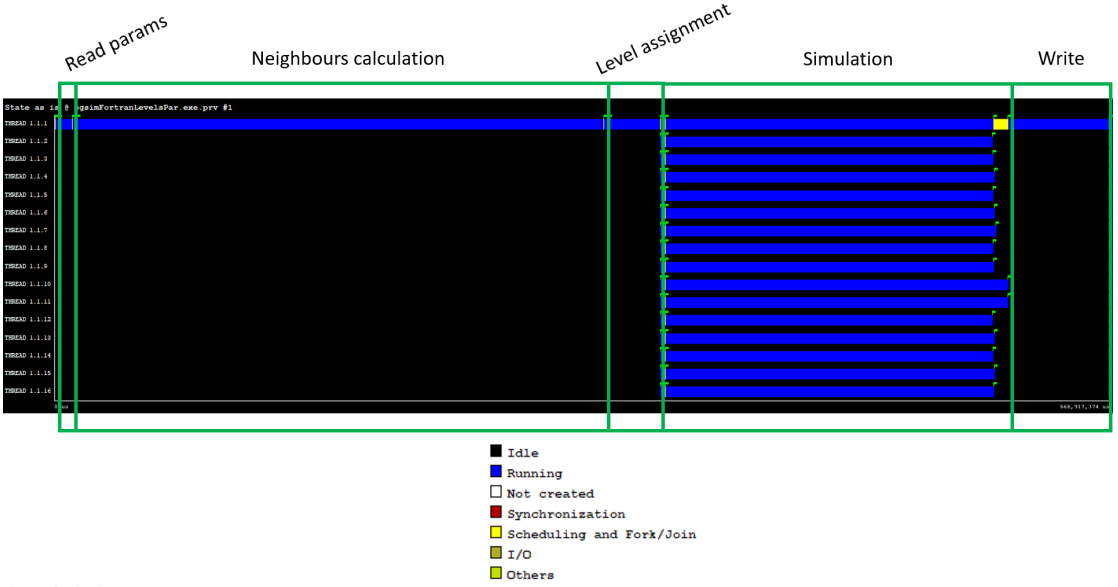

FIGURE 4.10: Profile of SGSIM case using parallel code with 16 threads and 64 maximum kriging neighbours, obtained with Extrae/Paraver tools.

4.4.2 Accuracy

Regarding numerical precision of the results, in SGSIM only small errors with absolute value less than 1.0^{-6} are present, as a result of non-commuting floating-point operations using the different order of simulation. As a reference, the results returned by SGSIM are single-precision floats with 6 to 9 significant decimal digits [IEEE, 2008]. To obtain the error values a simple node by node substraction is calculated between the simulated values using the original SGSIM non-parallel code and the values obtained using the parallel version, and then the histogram of errors is calculated. In the case of SISIM, the results are exactly the same since integer values are rounded for all categories, without small errors in lower decimal digits as SGSIM.

In comparison with other reported parallelization strategies, particularly Rasera et al. [2015], the efficiency obtained is comparable only in the larger cases of SISIM with 64 or 128 maximum neighbours. However, the results reported must not be compared directly, since different base codes and parameters were used. Since the proposed method of this work aims to generate the exact same results as the non-parallel versions of the simulation algorithms, the serial part of node reordering adds a major bottleneck if small domains or small maximum neighbour number are used in the configuration parameters. However, in some applications the exactness property can be particularly useful, like audited practices in mineral and ore reserves estimation [JORC, 2012].

4.4.3 Computational resources

In terms of computational resources, the parallelization strategy uses a large amount of memory to perform the level and neighbourhood storage in the current implementation version. The reason of this requirement is that many additional shared arrays with the same dimension of the simulation array must be allocated, and also additional space is needed by the neighbour information array **Neighbours**, extracted in the push stage of the spiral search (Algorithm 10). In the largest cases, with approximately 100 million nodes and 128 maximum kriging neighbours, around 96GB of memory where needed. This size comes largely from the array **Neighbours** which stores approximately $100,000,000 \times 2 \times 128$ 4-byte integers. With 16, 32 and 64 maximum kriging neighbours, the memory usage is around 12GB, 24GB and 48GB respectively. Since several cloud computing providers offer computational services at affordable prices, these memory usage values are not prohibitive given the current technological trends. For instance, a Linux virtual machine with 16 CPU-cores, 112GB RAM and 800GB of disk can be rented by 1 dollar per hour [Microsoft Azure, 2021].

Chapter 5

Parallel Neighbour Search

The second contribution of this thesis is related with the neighbour search used in the first stage of Algorithm 9. As described in Chapter 4, a key bottleneck that will allow improving the performance metrics is the neighbour search for node tagging according to their data dependency level. A parallel neighbour search method is described with details in this chapter. Four applications are modified, the parallel versions of `sgsim` and `sisim`, and two GSLIB-based applications, `sgs-lva` and `sisim-lva`. These last two applications are adapted versions that allow the usage of LVA-based simulation, as described in Section 2.3, and also will be adapted according to the parallel sequential simulation method. Additional case studies are presented for the LVA-based scenarios, using real and synthetic 3D datasets. Results of the parallelization proposed are presented in the last section.

5.1 Context

The neighbour search is a mandatory process in the sequential simulation method that needs to be computed before any kriging estimation. On each of the sequential simulation algorithms proposed in Chapter 2 (Algorithms 2, 3, 4 and 5), the pseudo-routine `search_neighbours` is present. The implementation available on each of these algorithms is described in detail on Deutsch and Journel [1998] Section II.4, where three approaches are presented: exhaustive search, super-block search and spiral search.

5.1.1 GSLIB search methods

Each of the existing approaches scan nearby data, and define neighbours to be used for kriging. Three conditions need to be fulfilled in order to be included as a neighbour:

- Only data points falling within a search ellipsoid centered at the location being estimated are considered. This ellipsoid is defined in the parameters of each GSLIB application (search radii in two or three directions).
- Only the closest data points, up to a maximum number parameter, are retained. Closeness between points is measured by the euclidean distance (possibly anisotropic).
- An octant search is available as an option to ensure that data points are taken on all sides of the point being estimated. This is particularly important when working in 3D with data often aligned along drillholes; an octant search ensures that data are taken from more than one drillhole. An octant search is specified by choosing the number of data to retain from each octant.

The first of the search approaches is exhaustive search, which is the simplest approach. It consists in a systematic check of all sampled conditioning data and retain the closest points that meet the three conditions noted above. This strategy is inefficient when there are many data points and has been adopted only in the straightforward 2D kriging program of GSLIB.

Super block search is the second search strategy, in which the data points are partitioned into a grid network superimposed on the domain being considered. When estimating any one location, it is then possible to limit the search to those data falling in nearby super blocks. This search has been implemented in most GSLIB kriging and simulation applications to search in non-gridded data points.

The last approach is called spiral search, which is defined only for searching on gridded data points. Algorithm 10 depicts the steps of this approach, in the context of the parallel sequential simulation method described on Chapter 4. The idea of this approach is to visit the closest nearby grid nodes first and spiral away until either enough data have been found or the remaining grid nodes are beyond the search limits. Figure 5.1 depicts an example of this spiral path, traversing from point 1 to 5, assuming a search window of 5×5. In order to include non-gridded data points, an initial re-location process can be applied, in which each non-gridded data point is placed in the closest grid location. This search has been implemented in all GSLIB sequential simulation applications.

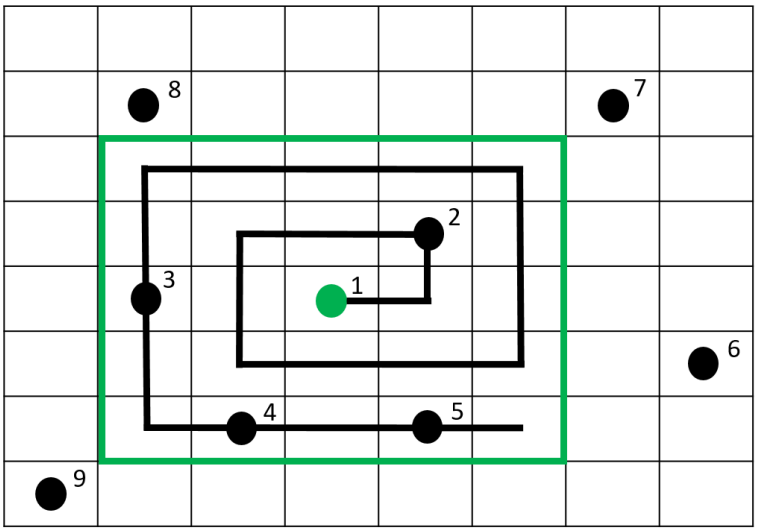

FIGURE 5.1: Spiral search example, centered in point 1.

5.1.2 kd-tree search methods

An additional algorithm included in the baseline implementations, particularly in `sgs-lva`, is based on a kd-tree data structure [Bentley, 1975] to support fast search of neighbours in the domain. The kd-tree (short for k-dimensional tree) data structure consists in a binary tree in which every terminal node is a k-dimensional point. Figure 5.2 shows an example of a kd-tree construction using 9 data points. Since LVA-based codes calculate euclidean distances between k-dimensional data points, with $k \geq 3$, kd-tree search methods are specially suitable in this case.

The current implementation used in the baseline code is the kd-tree neighbour search method from Kennel [2004], denoted as KDTree. It is implemented entirely in Fortran 90, as a module with a set of public and private variables, data structures and routines. Relevant variables in this module are bucket size (public), precision type (public), and the tree search record of the current search (private). Bucket size controls the number of data points that can be associated to a terminal node, allowing to combine a fast binary search with an exhaustive scan inside each bucket. The precision type controls if the floating-point type of variables in the module should be `real(4)` (single precision floats) or `real(8)` (double precision floats). The tree search record keeps the internal tree node where the search is actually running. This private variable will have an important role in the parallel algorithm proposed in the next section.

Compared against existing GSLIB search methods, for large scenarios, kd-tree search works faster due to its lower complexity. In case of exhaustive and spiral search, the algorithmic complexity in the average case is $O(N)$ with N the number of nodes in the domain. In practice, both methods will scan for neighbours until the maximum number

is achieved. Spiral search is faster since it starts scanning in nearby grid nodes. On the other hand, kd-tree search has algorithmic complexity $O(\log N)$ in the average case.

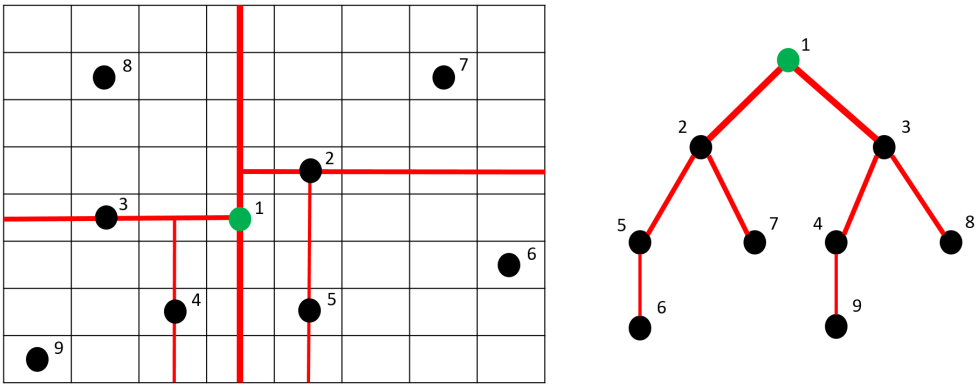

FIGURE 5.2: kd-tree data structure, centered in point 1.

Due to its lower algorithmic complexity, better software usability and its capabilities to manage searches for k-dimensional data points, we will adapt the usage of the KDTree implementation to every baseline code, and posteriorly those codes are adapted to run parallel and faster neighbour searches.

5.2 Algorithm

The proposed algorithm is based on a combination of an optimized version of the sequential KDTree neighbour search described in Section 5.1.2, and a parallel implementation presented in Algorithm 15, which is used in stage 1 of Algorithm 9. These search routines were implemented on the four codes under study, non LVA-based `sgsim` and `sisim`, and LVA-based `sgs-lva` and `sisim-lva`.

In case of `sgs-lva`, the neighbour search can be applied using KDTree search originally, since the baseline code already uses it. However, in `sisim-lva` and non LVA-based `sgsim` and `sisim`, the only options implemented are the original GSLIB methods exhaustive search and spiral search. In these cases the first task was to adapt the original code to include the KDTree method. In some cases, this modification already induced an improvement in the execution time.

In the next section a detailed explanation of the sequential optimizations applied to KDTree code are presented. After this description, the parallel algorithm is described in Section 5.2.2.

5.2.1 KDTree optimizations

In the existing implementation of kd-tree data structure and related methods, denoted KDTree, developed by Kennel [2004], the inner-most part of the computation should calculate distances between the query point and all points inside a terminal node of the tree. The points that are inside a fixed-size ball around the query points are marked as neighbours until the maximum number is reached. Figure 5.3 depicts a sample search of n neighbours using the routine `kdtree_n_nearest_around_point` from KDTree module. We can observe the recursive call to the inner method `search`, which eventually calls the terminal node processing routine `process_terminal_node`.

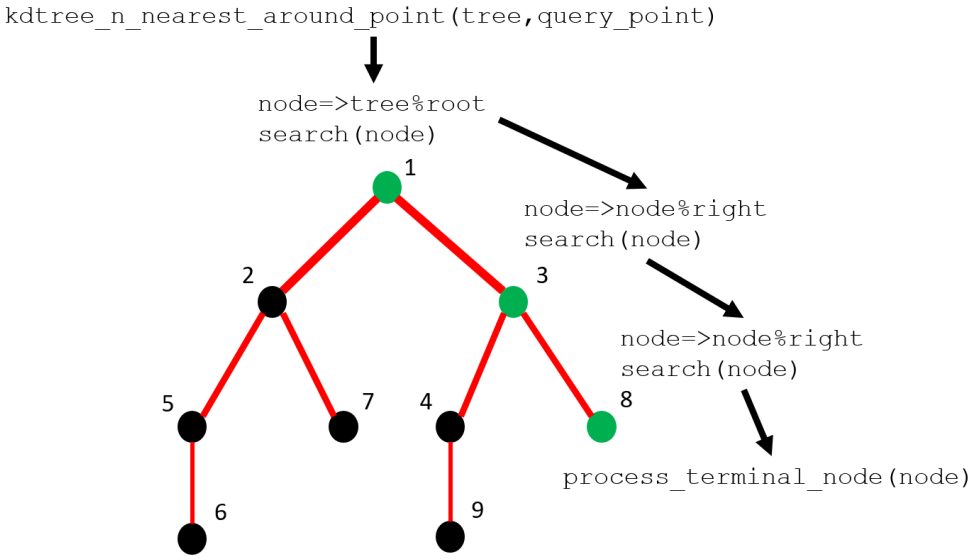

FIGURE 5.3: Sample execution of a search using KDTree code.

Using the command line profiler tool of `gprof` [Graham et al., 2004], we identify lines in KDTree code which are top contributors in the sequential part of execution. In Figure 5.4, the left block contains parts of the original code from routine `process_terminal_node`. The specific lines with a high contribution to execution time are lines 2 and 3. A first optimization is related with a reorder of the code, where the branching line `if(rearrange)` from line 6 is moved outside of the main loop. Additionally, a specialization of the main loop should be included on both sides of the branch (loops `mainlooprea` and `mainloopnorea`). With this optimization, we can reduce the number of branch instructions from δ=`node%u-node%l` to only 1 ($\delta = 12$ is the default bucket size, and can be modified in the module variables). Considering that millions of neighbour searches should be computed, and those searches will execute at least one time this routine, the gain in execution time is considerable (see Section 5.3)

The second optimization is based on the unroll of the loop that computes the squared distance between the query point and each potential neighbour, which again reduces the

number of branching instructions processed by the CPU. In Figure 5.5, the right block contains the optimized code of the routine `process_terminal_node` (including the first optimization). Lines 4 to 8 show the unrolling applied four times. The same unrolling is applied to the similar loop `mainloopnorea` from line 15. With this optimization, we can reduce the number of branching reduction but it is less evident since it will depend on the dimension of the data points. In the LVA-based scenarios this is relevant since dimensions can be in the order of $k \approx 1000$. Non LVA-based scenarios won't benefit strongly since the dimension is $k = 3$ (unrolling should not be used).

Original code

```
01     mainloop: do i = node%l, node%u
02        if (rearrange) then
03           sd = 0.0
04           do k = 1,dimen
05              sd = sd + (data(k,i) - qv(k))**2
06              if (sd>ballsize) cycle mainloop
07           end do
08           indexofi = ind(i)
09        else
10           indexofi = ind(i)
11           sd = 0.0
12           do k = 1,dimen
13              sd = sd + (data(k,indexofi) - qv(k))**2
14              if (sd>ballsize) cycle mainloop
15           end do
16        endif
17
18        ...
19
20     end do mainloop
```

Optimized code

```
01     if (rearrange) then
02        mainlooprea: do i = nodel, nodeu
03           sd = 0.0
04           do k = 1,dimen
05              sd = sd + (data(k,i) - qv(k))**2
06              if (sd>ballsize) cycle mainlooprea
07           end do
08           indexofi = ind(i)
09
10           ...
11
12        end do mainlooprea
13     else
14
15        mainloopnorea: do i = nodel, nodeu
16           indexofi = ind(i)
17           sd = 0.0
18           do k = 1,dimen
19              sd = sd + (data(k,indexofi) - qv(k))**2
20              if (sd>ballsize) cycle mainloopnorea
21           end do
22
23           ...
24
25        end do mainloopnorea
26     end if
```

FIGURE 5.4: First optimization of routine `process_terminal_node` from KDTree implementation. Branching reduction by removing `if(rearrange)` branch outside of the main loop, and duplicating it with specialized code.

Original code

```
01     mainloop: do i = node%l, node%u
02        if (rearrange) then
03           sd = 0.0
04           do k = 1,dimen
05              sd = sd + (data(k,i) - qv(k))**2
06              if (sd>ballsize) cycle mainloop
07           end do
08           indexofi = ind(i)
09        else
10           indexofi = ind(i)
11           sd = 0.0
12           do k = 1,dimen
13              sd = sd + (data(k,indexofi) - qv(k))**2
14              if (sd>ballsize) cycle mainloop
15           end do
16        endif
17
18        ...
19
20     end do mainloop
```

Optimized code

```
01 if (rearrange) then
02    mainlooprea: do i = nodel, nodeu
03       sd = 0.0
04       do k = 1,4,dimen-3
05          sd = sd + (data(k,i) - qv(k))**2
06          sd = sd + (data(k+1,i) - qv(k+1))**2
07          sd = sd + (data(k+2,i) - qv(k+2))**2
08          sd = sd + (data(k+3,i) - qv(k+3))**2
09          if (sd>ballsize) cycle mainlooprea
10       end do
11       indexofi = ind(i)
12       ...
13    end do
14 else
15    mainlooprea: do i = nodel, nodeu
16       ...
17    end do mainlooprea
18 end if
```

FIGURE 5.5: Second optimization of routine `process_terminal_node` from KDTree implementation. Loop unrolling applied in multidimensional squared euclidean distance calculation.

5.2.2 Parallel neighbour search

With the optimized KDTree module at hand, we will review how to use it by multiple parallel searches, following Algorithm 15. The parallel implementation is based on a modified OpenMP-compliant version of KDTree search and a block cyclic decomposition strategy of the random path. The block cyclic approach is necessary since an underlying unbalance exists in the amount of work for neighbour search (early points require more effort than later points).

Input:
Levels: array defined in Algorithm 9 (line 7);
Neighbours: array defined in Algorithm 9 (line 8);
$|\Omega|$: number of points in 3D domain Ω;
$\mathcal{P}$: random path defined in Algorithm 9 (line 3);
$\mathcal{Z}$: embedding of point coordinates as defined in Algorithm 4 (line 4). For non-LVA scenarios, the embedding is exactly the 3D coordinates in Ω;
k^{search}: defined as input in Algorithm 4. For non-LVA scenarios, this value is 3;
n^{max}: defined as input in Algorithm 4;
T: number of parallel threads of execution;
b: block size

```
1  n ← |Ω|
2  Level ← zeros(n × 1)
3  Neighbours ← zeros(n × n^max × 2) // Next loop runs in a parallel OpenMP region
4  for threadId ∈ {1,...,T} in parallel do
5      B ← ⌈n/b⌉
6      //Each thread will have a copy of its own KDTree structure
7      Tree ← kdtree_create(Z, n^max, k^search)
8      //Block cyclic through blocks
9      nlast ← 1
10     for iblock ∈ {1,...,B} do
11         nmin, nmax ← (b − 1) ∗ B , min{b ∗ B, n}
12         if iblock % T == (threadId − 1) then
13             for ixyz ∈ {nlast,...,nmin − 1} do
14                 point_marked(Tree, P_ixyz)
15             end
16             nlast ← nmax + 1
17             //Each thread computes the valid neighbours of each point
18             for ixyz ∈ {nmin,...,nmax} do
19                 //Search and push operation to save neighbours in array
20                 Neighbours(P_ixyz) ← search_neighbours_push(P_ixyz, κ, Z, Tree, k^search)
21                 point_marked(Tree, P_ixyz)
22             end
23         end
24     end
25     omp_barrier //Only thread 1 will compute the levels (intrinsically sequential)
26     if threadId = 1 then
27         for ixyz ∈ {1,...,n} do
28             Level(P_ixyz) ← build_level(P_ixyz, κ, Neighbours)
29         end
30     end
31 end
```

Output: **Neighbours**, **Levels**

Algorithm 15: Routine `parallel_neighbour_search` (KDTree-based)

In order to use the existing KDTree implementation, a key change involves the private variable `sr` used internally in the module. It should be declared as `threadprivate` with an OpenMP directive, as depicted in Figure 5.6, implying only one line (line 18). With this change, different threads can create private search records on the same tree and search independently for neighbours of different points in parallel (line 7 of Algorithm 15).

Original code

```
01 module kdtree2_module
02   use kdtree2_precision_module
03   use kdtree2_priority_queue_module
04   !!!!!!!! PUBLIC DATA TYPE, CREATION, DELETION
05   public :: kdkind
06   ...
07   public :: kdtree2_create_v2
08   !!!!!!!! PUBLIC SEARCH ROUTINES
09   ...
10   public :: kdtree2_n_nearest_around_point
11   ...
12   !!!!!!!! PUBLIC GLOBAL VARIABLES
13   integer, parameter :: bucket_size = 12
14   ...
15   !!!!!!!! PRIVATE GLOBAL VARIABLES
16   private
17   type(tree_search_record), save, target :: sr
18
19 contains
20    !!!!!!!! ROUTINE DEFINITION
21    ...
22 end module kdtree2_module
```

Parallel code

```
01 module kdtree2_module
02   use kdtree2_precision_module
03   use kdtree2_priority_queue_module
04   !!!!!!!! PUBLIC DATA TYPE, CREATION, DELETION
05   public :: kdkind
06   ...
07   public :: kdtree2_create_v2
08   !!!!!!!! PUBLIC SEARCH ROUTINES
09   ...
10   public :: kdtree2_n_nearest_around_point
11   ...
12   !!!!!!!! PUBLIC GLOBAL VARIABLES
13   integer, parameter :: bucket_size = 12
14   ...
15   !!!!!!!! PRIVATE GLOBAL VARIABLES
16   private
17   type(tree_search_record), save, target :: sr
18   !$omp threadprivate(sr)
19 contains
20    !!!!!!!! ROUTINE DEFINITION
21    ...
22 end module kdtree2_module
```

FIGURE 5.6: OpenMP directive to allow multiple parallel searches using KDTree module (single change in line 18).

In the next paragraph we describe in detail the steps of Algorithm 15. In order to search for all neighbours of each location following the sequential random path in a parallel multi-threaded way, each thread will create a private copy of the KDTree data structure storing all domain gridded locations, depicted in line 7. Initially each tree contains the same information, and only initial conditioning sampled data is *marked* (a global logical array is used to mark each grid node as usable or not). As the algorithm progresses, only previously simulated nodes or initial conditioning data (*marked* nodes) are considered in KDTree searches, since the neighbour search should be compliant with the sequential random path (see Section 4.1.1). This constraint is applied in lines 13 to 15 of the Algorithm 15, by setting as *marked* all previous nodes for each thread of execution. By combining this strategy with a block cyclic distribution of iterations (lines 10 to 12 of Algorithm 15), the final workload is balanced through all threads, as shown in Figure 5.7. In this figure, the random path index is presented from 1 to 80, with blocks of size $B = 10$. Four threads are depicted, each one with a different color and number. We can observe that threads 1, 2 ,3 and 4 start the execution with the same value `nlast` $= 1$, however, before processing each block, thread 2 needs to mark points 1 to 10, thread 3 needs to mark points 1 to 20 and thread 4 needs to mark points 1 to 30, depicted as a gray line. Continuing with the block cycling, the second block is proceesed by each thread but this time the value of `nlast` is different for every thread, and the

marked nodes are depicted as the intermediate gray line. Figure 5.8 depicts each thread processing blocks for neighbour search, mixed with marked nodes to be skipped. In lines 18 to 22 of Algorithm 15, the neighbour search calculation is executed, with a subsequent marking step of the corresponding node. Line 25 depicts an OpenMP barrier which is needed in order to wait for all threads before computing the levels of dependency. The final steps, depicted in lines 26 to 30 of Algorithm 15, use the computed neighbours to infer the *level* of each point, which indicates the degree of dependency of that point on previously simulated or initial conditioning points. Specifically, initial conditioning points are level 0 and a point is level n if the maximum level of all its neighbours is $n-1$.

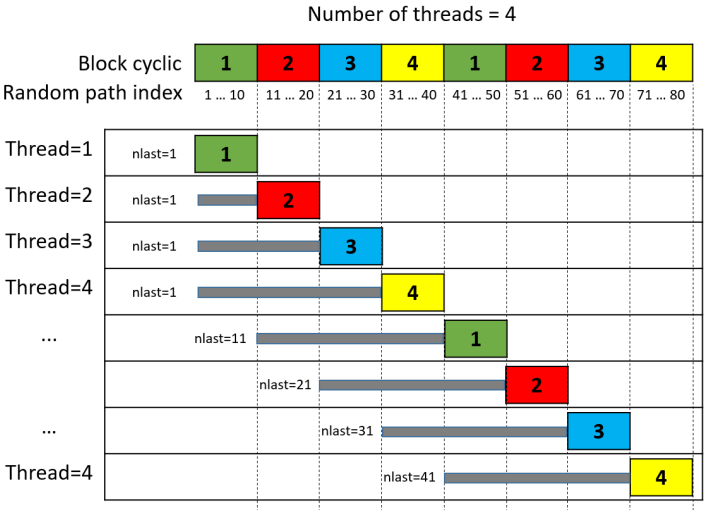

FIGURE 5.7: Load balancing of workload through a block cyclic strategy for parallel neighbour search. In this example, 4 threads are computing neighbours of different blocks of points (block size equal to 10, domain size equal to 80). Before processing a block of points, each thread should declare as *marked* all previous points which are not marked yet by this thread (gray color line). Variable `nlast` is used to indicate the starting index of marked points for the next block.

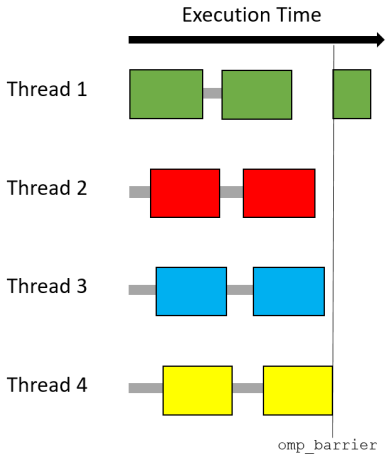

FIGURE 5.8: Each thread processes its own cyclic blocks from Figure 5.7 (colored blocks) and pass through other blocks marking the corresponding points as simulated (grey lines). After all threads end their processing, an OpenMP barrier is set, and after it the main thread finishes the level computation.

In summary, the proposed algorithm re-formulates an intrinsically sequential part of the sequential simulation method in geostatistics. The neighbour search computation in this context has the additional complexity of accounting for previously simulated points in the domain. Application of this algorithm is presented in the next section, with promising results for non LVA and LVA based codes.

5.3 Results

This section is divided in two subsections. The first one shows performance tests of the proposed implementation from Section 5.2 using two scenarios extracted from Section 4.3, with adapted parallel versions of non LVA-based `sgsim` and `sisim` respectively. In the second subsection, performance tests of the proposed implementation are presented for two LVA-based scenarios, using parallel LVA-based codes `sgs-lva` and `sisim-lva`.

5.3.1 Performance tests for parallel non LVA-based codes

In order to measure the performance of the proposed parallel algorithm, simulations are generated using non LVA-based baseline codes `sgsim` and `sisim`, developed in Chapter 4. Both scenarios are denoted *sgsim* and *sisim* respectively. Scenario *sgsim* uses an initial 3D dataset of 2376 diamond drill-hole samples with information of copper grade composites. Scenario *sisim* uses a synthetic 3D dataset of 3000 random samples with 10 categories (see Section 3.4). The parameters in each scenario can be viewed in Table 5.1 and their descriptions in Section 3.2.

Parameter	*sgsim*	*sisim*
Domain $nx \times ny \times nz$	$400 \times 400 \times 120$	$210 \times 600 \times 400$
Max nodes for simulation	$\{16, 32, 48, 64\}$	$\{16, 32, 48, 64\}$
Kriging	OK	OK
Number of structures (type)	3 (spherical, exponential, gaussian)	10 (sphericals)

TABLE 5.1: Default parameters for *sgsim* and *sisim*.

All runs were executed in a single-node machine with Ubuntu 18.04.5 LTS with 2 $\times$ 10-cores Intel(R) Xeon(R) CPU Silver 4210R at frequency 2.40GHz and a main memory of 128 GB RAM. All Fortran programs were compiled using GNU Fortran version 4.8.5 supporting OpenMP version 3.1, with options -cpp -O2 -ffast-math -ftree-vectorize. Execution time results are depicted in Figures 5.9 and 5.11, and speedup results are depicted in Figures 5.10 and 5.12.

In the *sgsim* scenario depicted in Figure 5.10 we can observe that the speedup increased consistently accross all tests, using up to 16 threads (this number of threads is selected in order to compare results from Section 4.3). Speedups using 16 threads between the proposed implementation against the baseline version without parallel neighbour search are 1.33×, 1.79×, 1.85× and 2.14× in cases with 16, 32, 48 and 64 maximum neighbours for simulation respectively. However, it is important to notice that using lower numbers of neighbours, such as 16 and 32, the execution time is considerably lower in the baseline escenario using less than 8 and 4 threads of execution respectively. The reason for this behaviour is the amount of work that needs to be done to initialize the KDTree parallel structures, which in these cases is higher than the baseline search methods. On all other cases the parallel adaptation has lower execution time than the baseline code.

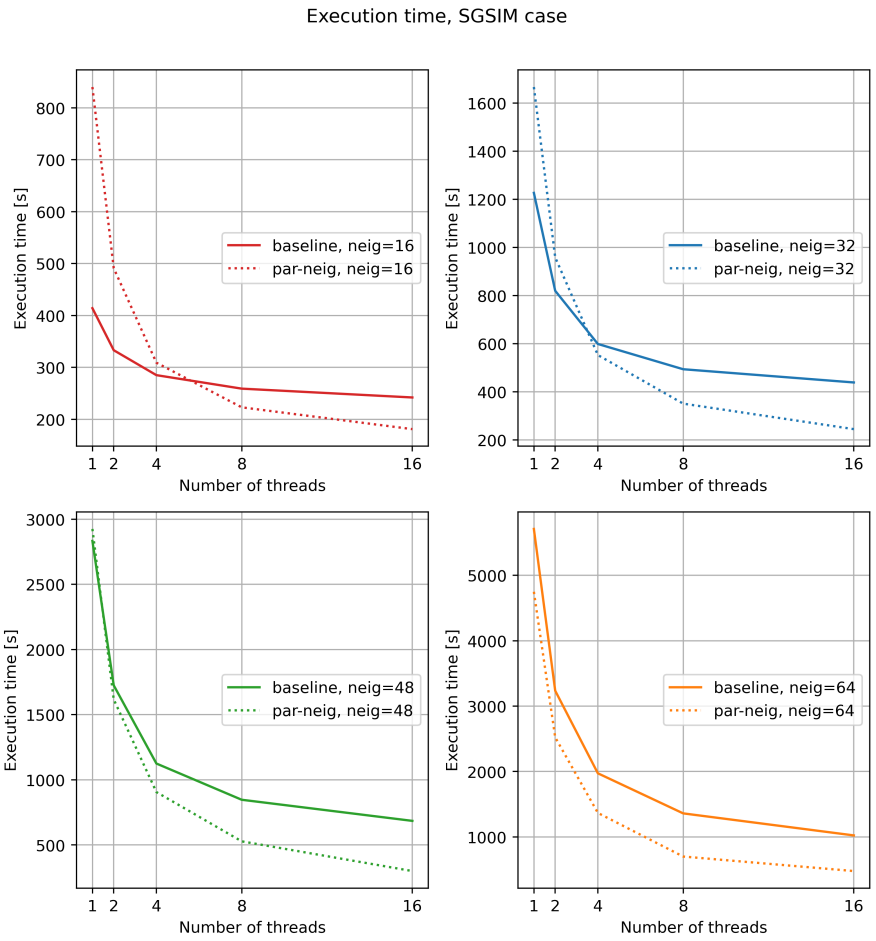

FIGURE 5.9: Execution time [seconds] comparison between baseline `sgsim` parallel code and adapted `sgsim` parallel code using the parallel neighbours search.

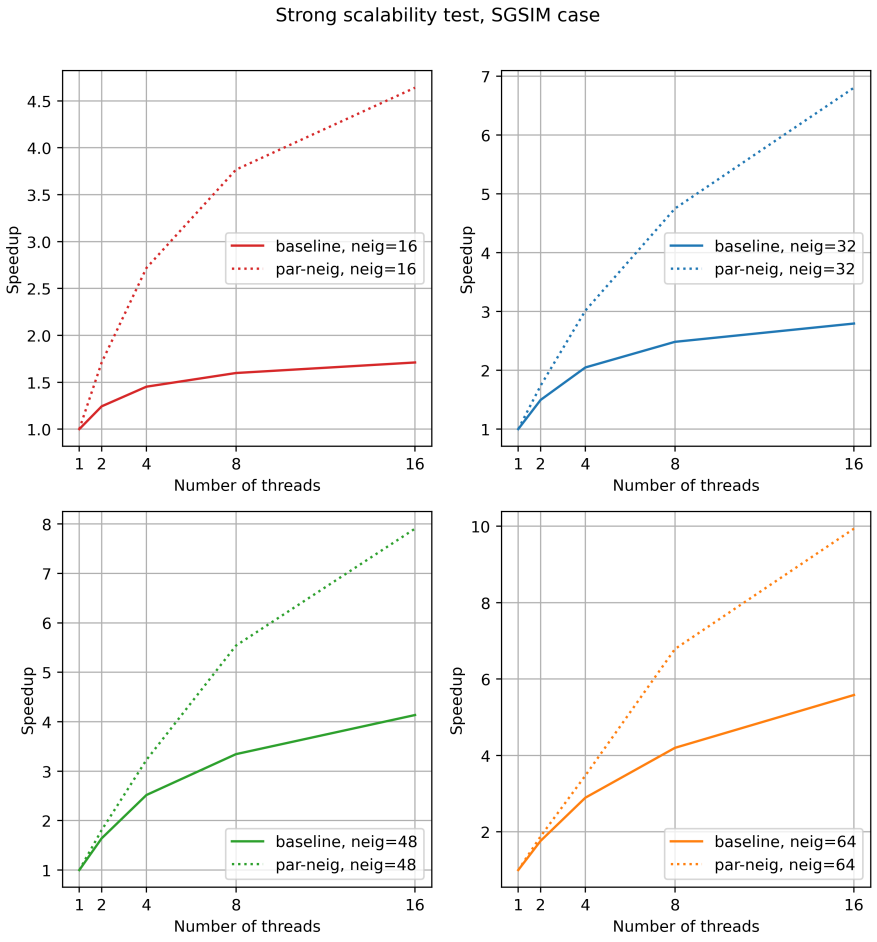

FIGURE 5.10: Speedup comparison between baseline `sgsim` parallel code and adapted `sgsim` parallel code using the parallel neighbours search algorithm. Each speedup curve is calculated as the time of a single-thread execution divided by a multi-thread execution, using each code separately.

In the *sisim* scenario depicted in Figure 5.12 we can observe no significative differences in speedup trends between the baseline and the new parallel adaptation. From Figure 5.12, speedups using 16 threads between the proposed implementation against the baseline version without parallel neighbour search are 1.55×, 2.42×, 3.06× and 4.11× in cases with 16, 32, 48 and 64 maximum neighbours for simulation respectively. In this scenario the execution time is consistently lower in the new parallel implementation. This can be explained since this scenario involves more work (10 kriging linear systems should be solved for each domain point versus only one linear system in *sgsim*), so the initialization of KDTree structures is not significative in the execution time.

Finally, if we compare the aggregated contribution to speedup of the new parallel neighbour search and additional optimizations, plus the previous parallelization work from Chapter 4, against a single thread execution of the previous parallelization work, the obtained speedups using 16 threads for *sgsim* scenario are 2.2×, 5.0×, 7.6× and 11.9×, using 16, 32, 48 and 64 maximum neighbours respectively. Similarly for *sisim* scenario, speedups using 16 threads are 7.8×, 20.3×, 32.7× and 50.4×, using 16, 32, 48 and 64 maximum neighbours respectively. A summary of the results can be viewed in Figure 5.13.

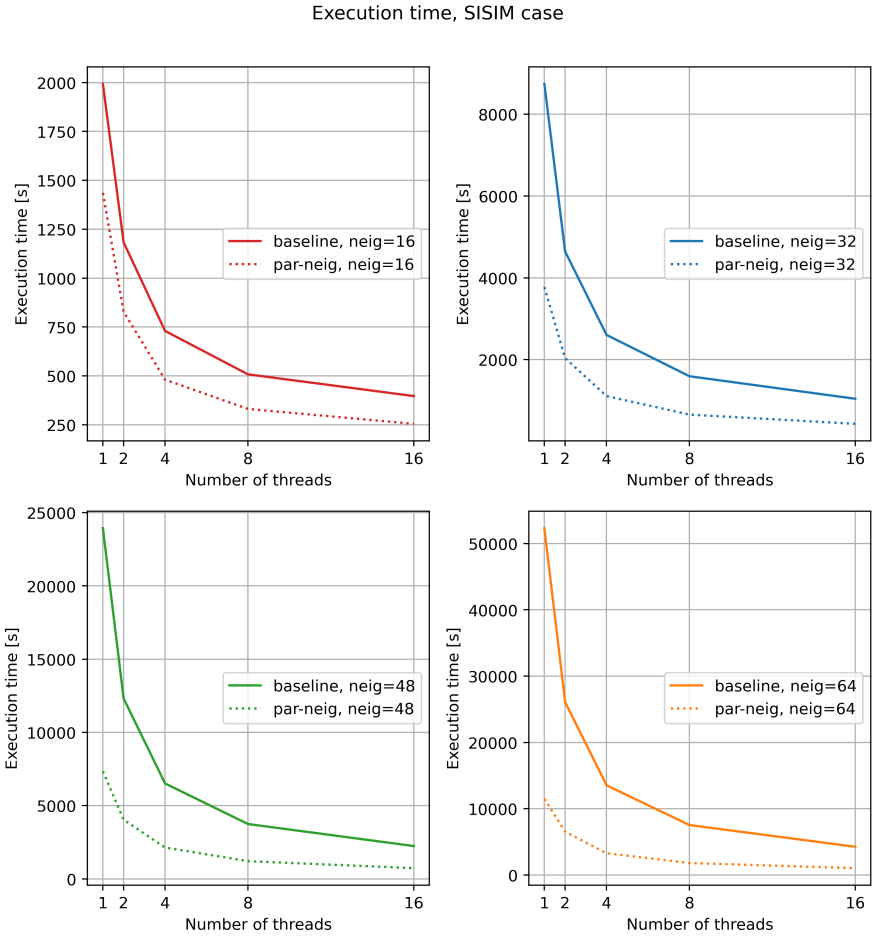

FIGURE 5.11: Execution time [seconds] comparison between baseline `sisim` parallel code and adapted `sisim` parallel code using the parallel neighbours search algorithm.

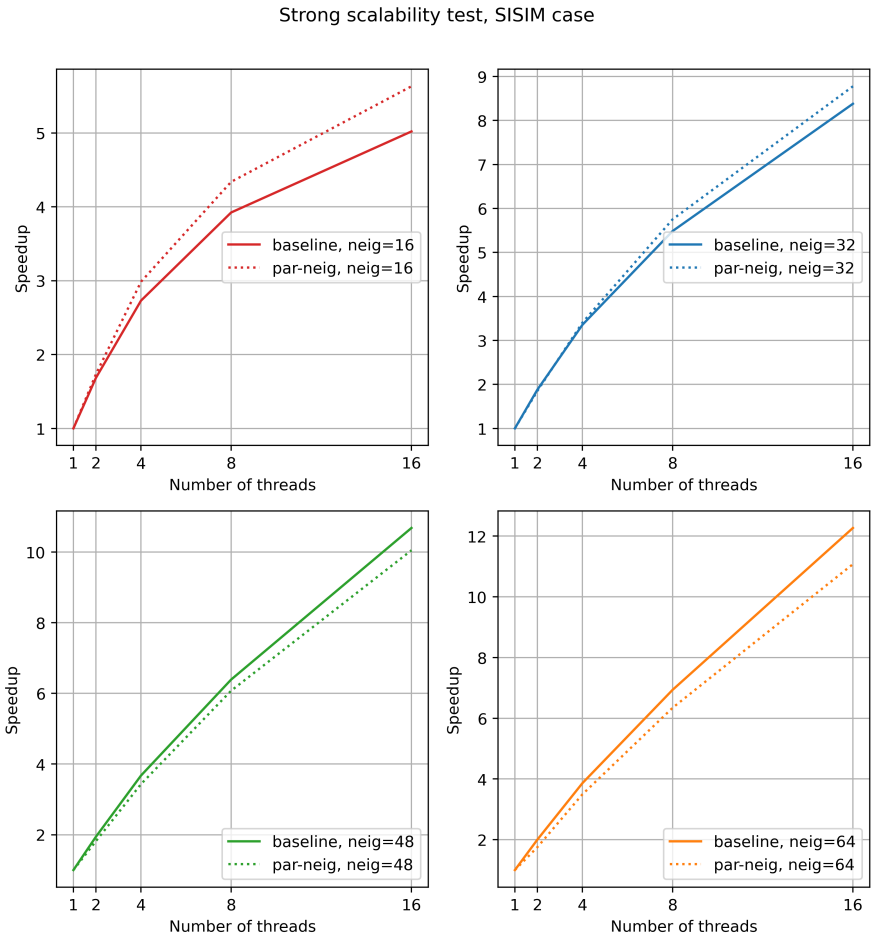

FIGURE 5.12: Speedup comparison between baseline `sisim` parallel code and adapted `sisim` parallel code using the parallel neighbours search algorithm. Each speedup curve is calculated as the time of a single-thread execution divided by a multi-thread execution, using each code separately.

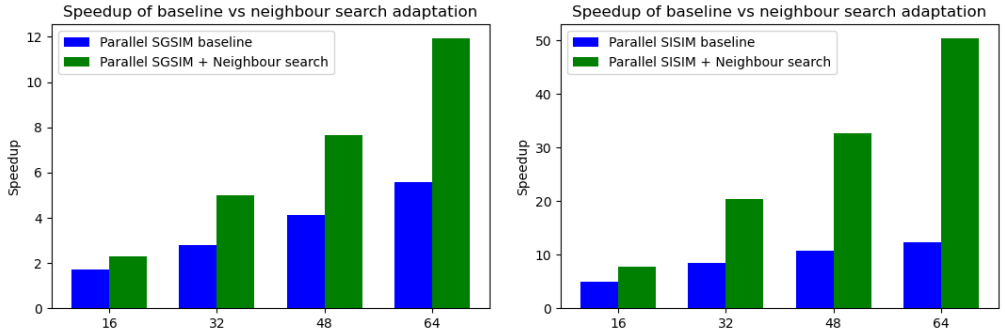

FIGURE 5.13: Speedup of parallel baseline codes versus parallel baseline with the parallel neighbours search algorithm adapted.

5.3.2 Performance tests for parallel LVA-based codes

The first scenario, namely *swiss-roll*, is based in the swiss roll testing scenario, which is extensively used in the Machine Learning community [Tenenbaum et al., 2000]. In our case, a 3D swiss roll is prepared, which is posteriorly transformed into a 3D LVA field. A synthetic dataset of 17280 samples is designed and attached to the domain as sample data. The second scenario, namely *escondida*, is based on real mining 3D data of 2376 diamond drill-hole samples with information of copper grade composites and lithologies per sample. In this case, a synthetic fold-like LVA field is used for simulation. The parameters of each scenario are detailed in Table 5.2. A schema of the LVA fields and the drillhole data are depicted in Figures 5.14 and 5.15. In Figures 5.16 and 5.17 we can observe simulated domains in the *swiss-roll* scenario using LVA-based `sgs-lva`. In the fist figure, 6 slices of simulated domains are presented, each one generated with different values of r_1 ratio. The second figure shows two 3D simulated domains generated with LVA fields with different r_1 ratio values. In Figures 5.18 and 5.19 we can observe simulated domains in the *escondida* scenario using LVA-based `sisim-lva`, with 4 categorical values. As in the previous figures, the first one shows 3 slices of simlated domains each one generated with different values of r_1 ratio. The second one shows a 3D simulated domain generated with an LVA field.

Parameter	*swiss-roll*	*escondida*
Domain $nx \times ny \times nz$	$120 \times 120 \times 120$	$148 \times 220 \times 52$
LVA field $nx \times ny \times nz$	same as domain	same as domain
Graph connectivity (offset)	1	1
Landmarks $nx \times ny \times nz$	$10 \times 10 \times 10$	$8 \times 12 \times 14$
k^{cova}	$\{32, 64, 125, 250, 500, 1000\}$	$\{42, 84, 168, 336, 672, 1344\}$
k^{search}	$\{32, 64, 125, 250, 500, 1000\}$	$\{42, 84, 168, 336, 672, 1344\}$
Max nodes for simulation	48	48
Kriging	SK	SK
Number of structures (type)	1 (exponential)	4 (exponentials)

TABLE 5.2: Default parameters for *swiss-roll* and *escondida*.

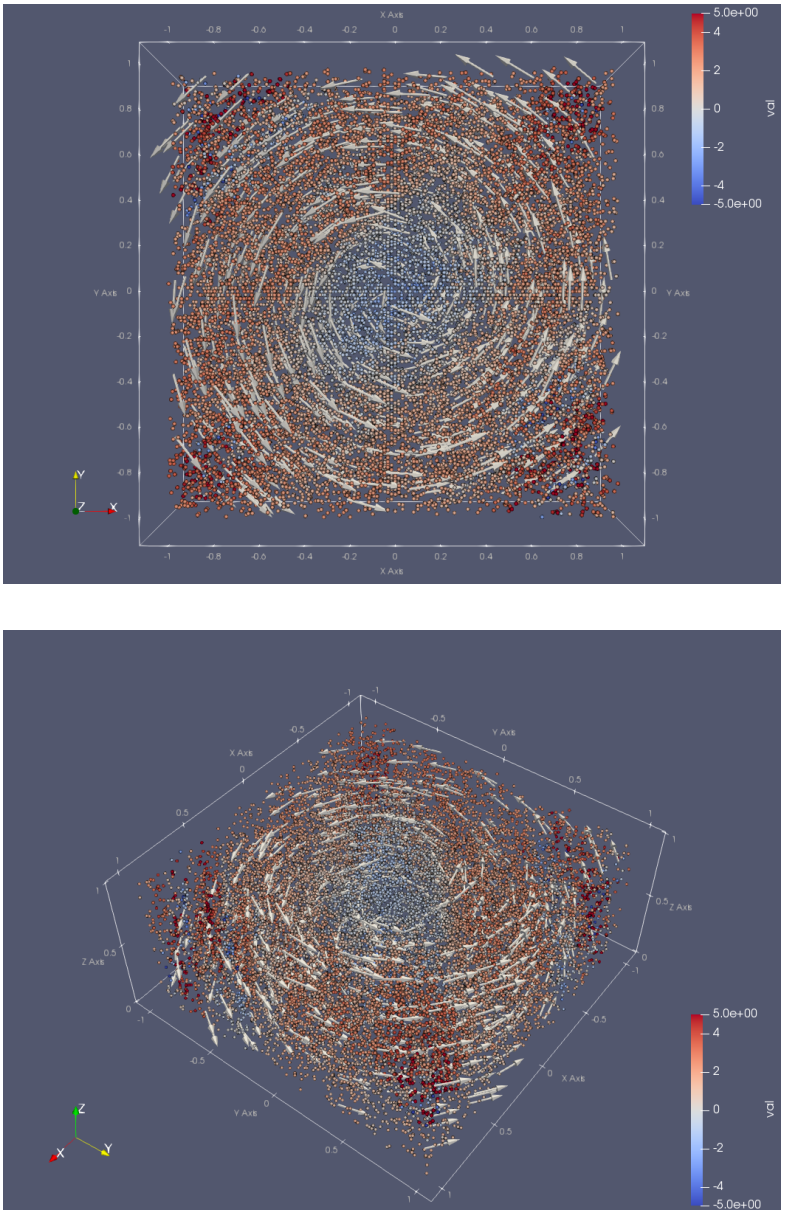

FIGURE 5.14: *swiss-roll*: different views of sample points with LVA field dataset (sample).

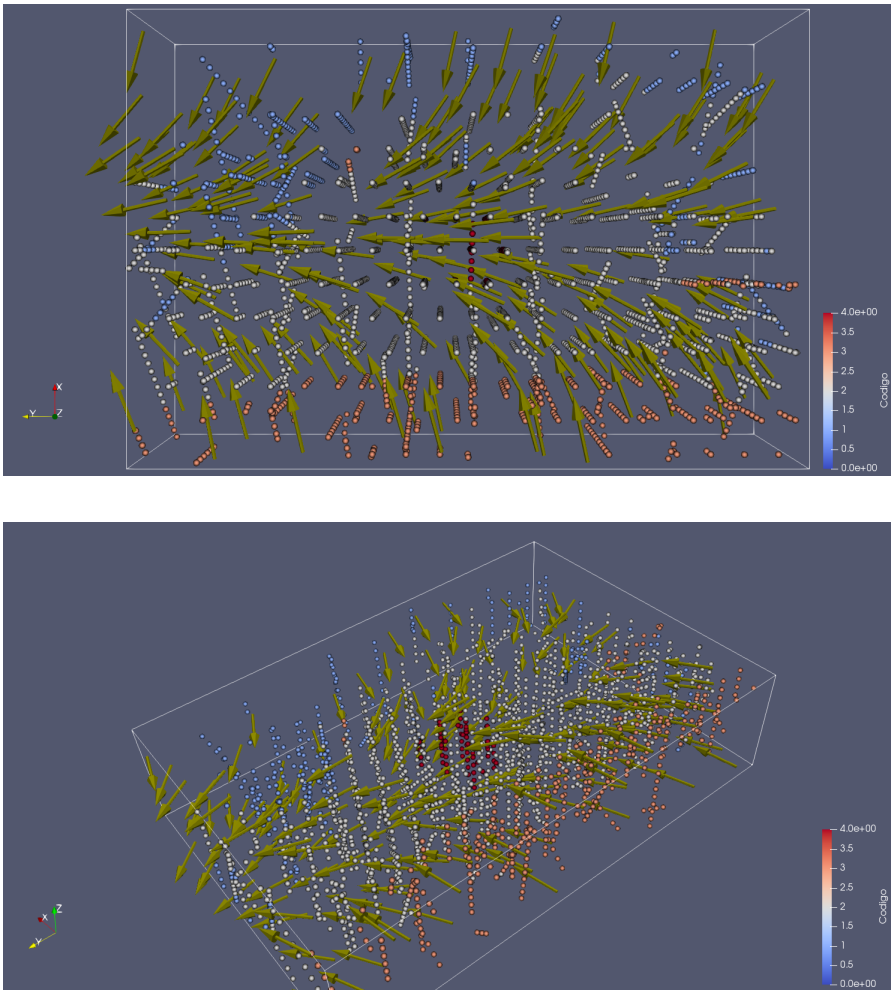

FIGURE 5.15: *escondida*: different views of sample drillhole points with LVA field dataset (sample).

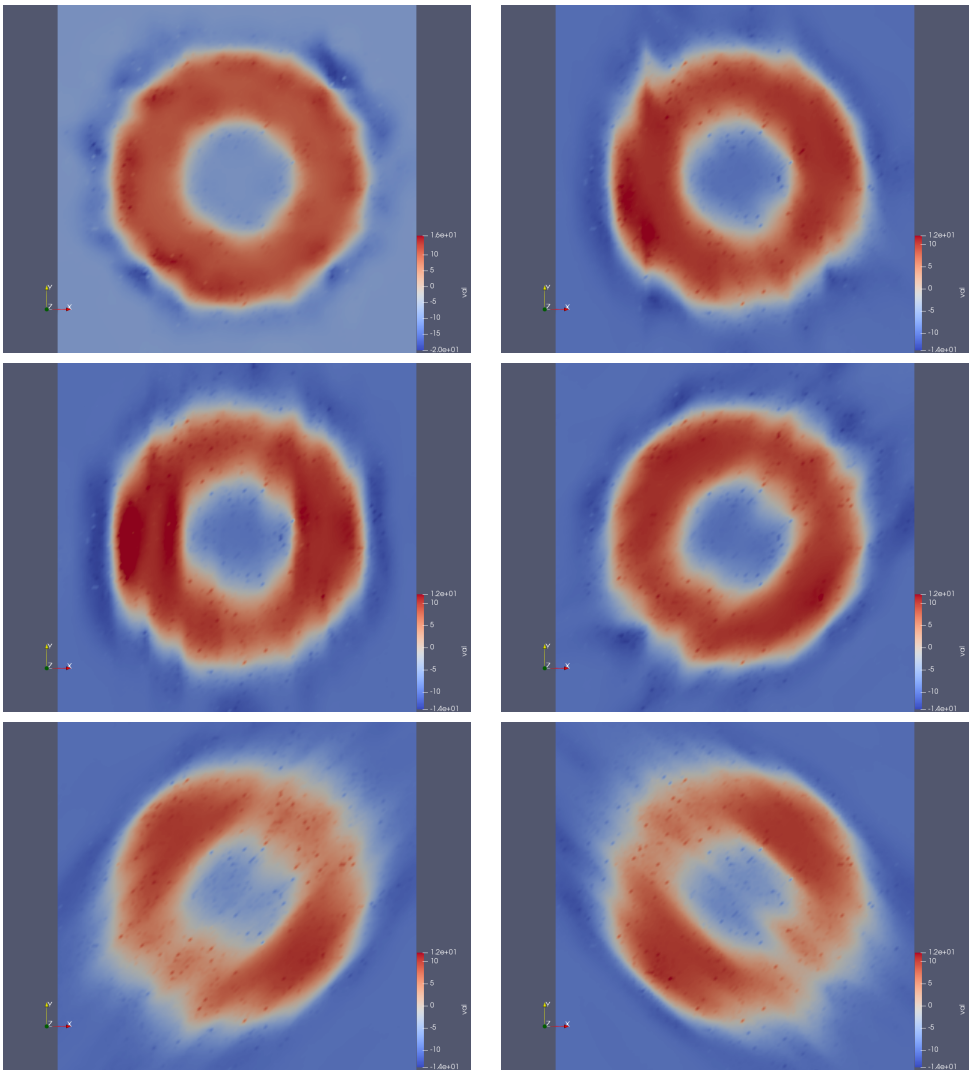

FIGURE 5.16: Slices of simulated domains for *swiss-roll* scenario using parallel LVA-based SGS with different r_1 ratio values from LVA field parameters.

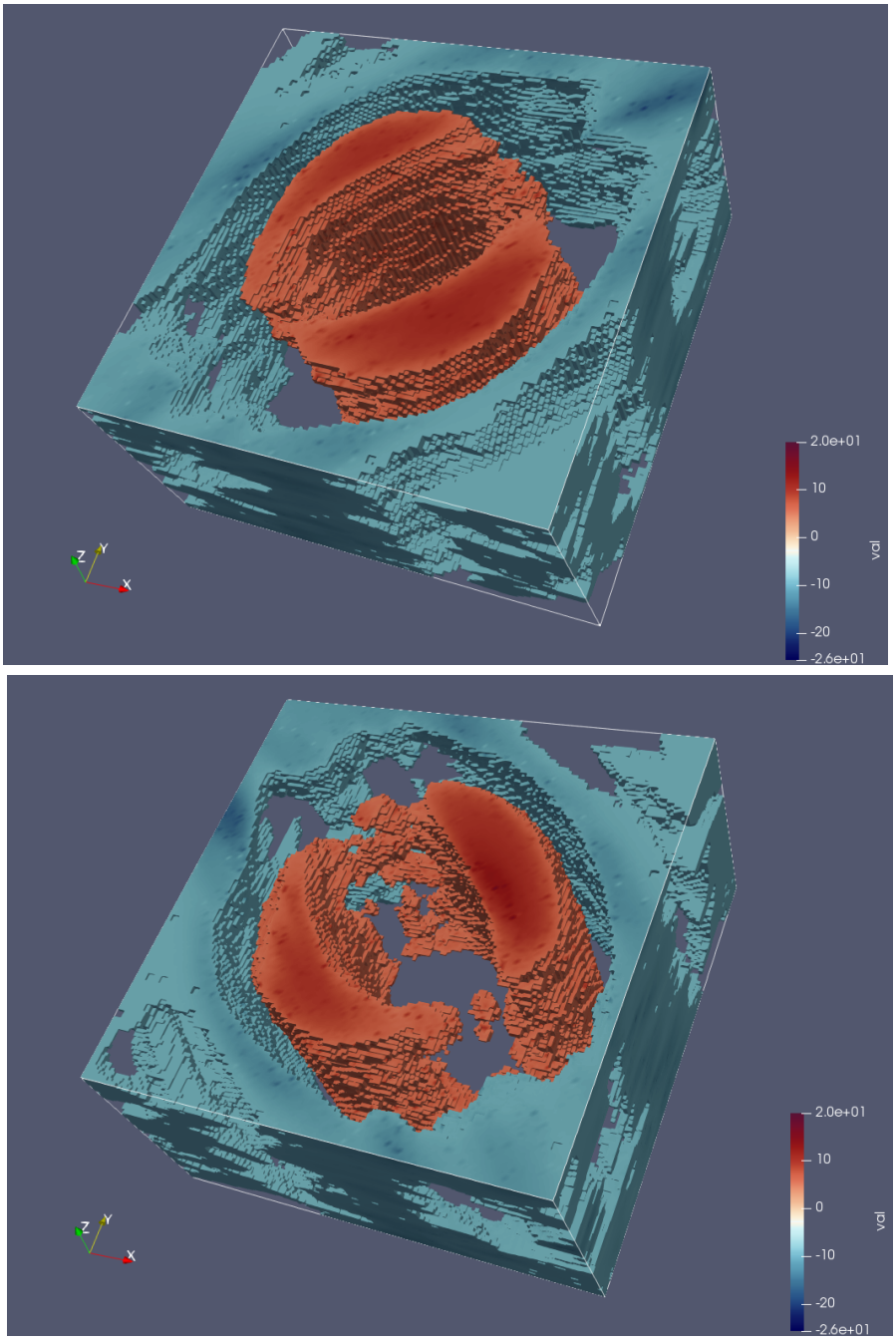

FIGURE 5.17: Top: Simulated domain for *swiss-roll* scenario using parallel LVA-based SGS with $r_1 = 5$ and two threshold views. Bottom: Similar view with LVA parameter $r_1 = 0.2$.

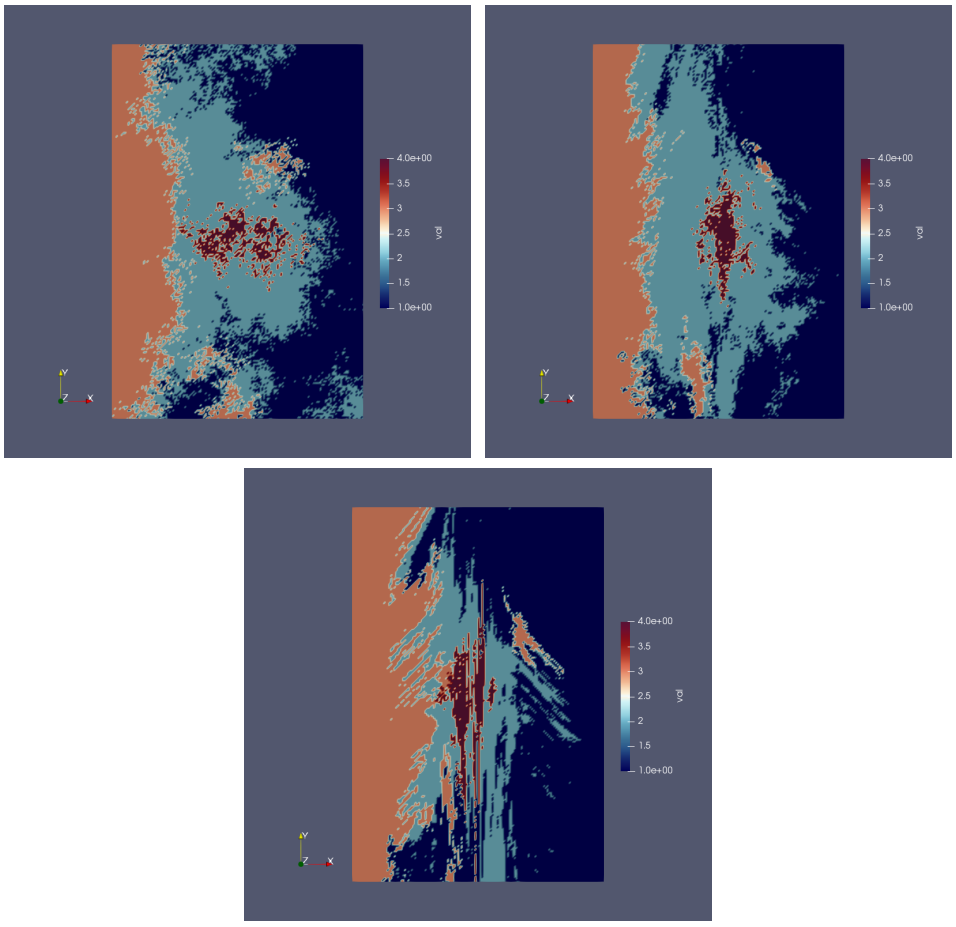

FIGURE 5.18: Slices of simulated domains for *escondida* scenario using parallel LVA-based SISIM with different r_1 ratio values from LVA field parameters.

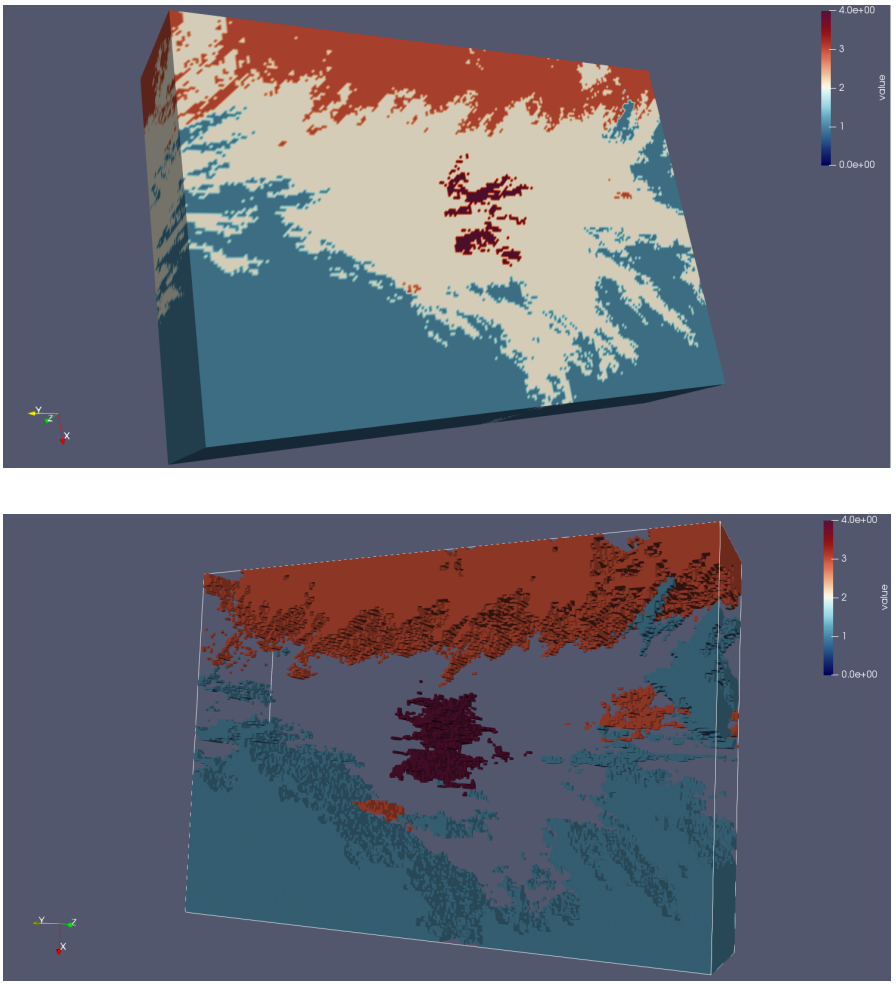

FIGURE 5.19: Top: Simulated domain for *escondida* scenario using parallel LVA-based SISIM. Bottom: Three categories of the simulation on top.

All runs were executed in a single-node machine with Ubuntu 18.04.5 LTS with 2 $\times$ 10-cores Intel(R) Xeon(R) CPU Silver 4210R at frequency 2.40GHz and a main memory of 128 GB RAM. All Fortran programs were compiled using Intel ifort version 13.1.1 supporting OpenMP version 3.1, with options -fpp -mkl -openmp -O3 -mtune=native -march=native. All C++ programs were compiled using GCC g++ version 6.2.0 supporting OpenMP version 4.5, with options -Ofast -fopenmp -funroll-loops -finline-functions -ftree-vectorize.

Figure 5.20 shows execution time and speedup of the LVA-based `sgs-lva` parallel code for the *swiss-roll* scenario, using $k^{cova} = k^{search} = 32$ and $k^{cova} = k^{search} = 1000$ control dimensions. Figure 5.21 shows execution time and speedup of the LVA-based `sisim-lva` parallel code for the *escondida* scenario, using $k^{cova} = k^{search} = 42$ and $k^{cova} = k^{search} = 1344$ control dimensions. On both figures, speedup is computed using the baseline sequential code (middle plot) and fully optimized code (bottom plot). Differences in speedup values on both plots, can be explained by large differences in

elapsed time of the original baseline code and the optimized single threaded execution of the OpenMP code. In case of `sisim-lva`, this effect reduces three orders of magnitude the baseline execution time, mostly due to the refactoring of neighbour search using KDTree instead of exhaustive search.

Finally, to illustrate the contribution of the neighbour search acceleration and parallelization to the overall speedup, in Table 5.3 we can observe execution time and speedup against the baseline code version for LVA-based `sgs-lva` and `sisim-lva` using $k^* = 1000$ and $k^* = 1344$ respectively. The purpose of this information is to show the relevance of the acceleration and parallelization of the neighbour search, which allows to improve the speedup an order of magnitude for each scenario. Special relevance has the inclusion of KDTree search in LVA-based `sisim-lva` which delivered a single contribution of two orders of magnitude in a sequential execution.

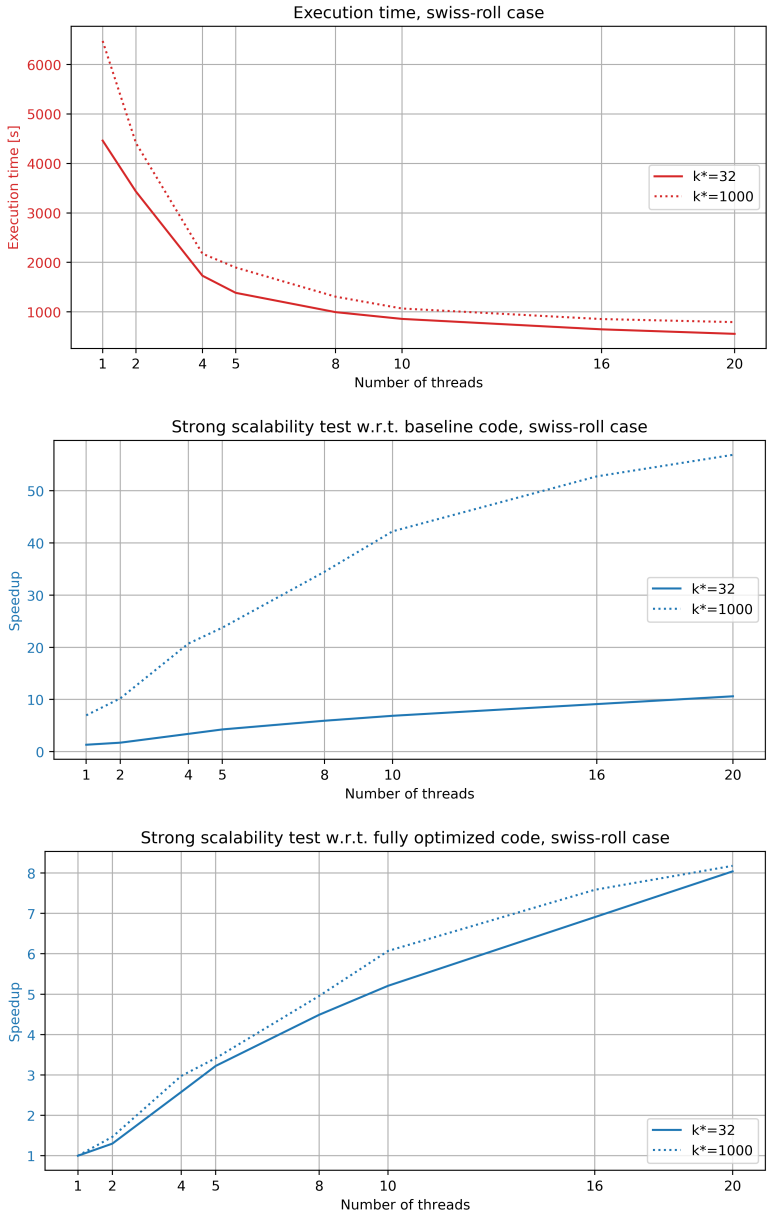

FIGURE 5.20: Execution time [seconds] and speedup results for *swiss-roll* scenario using parallel LVA-based SGS code.

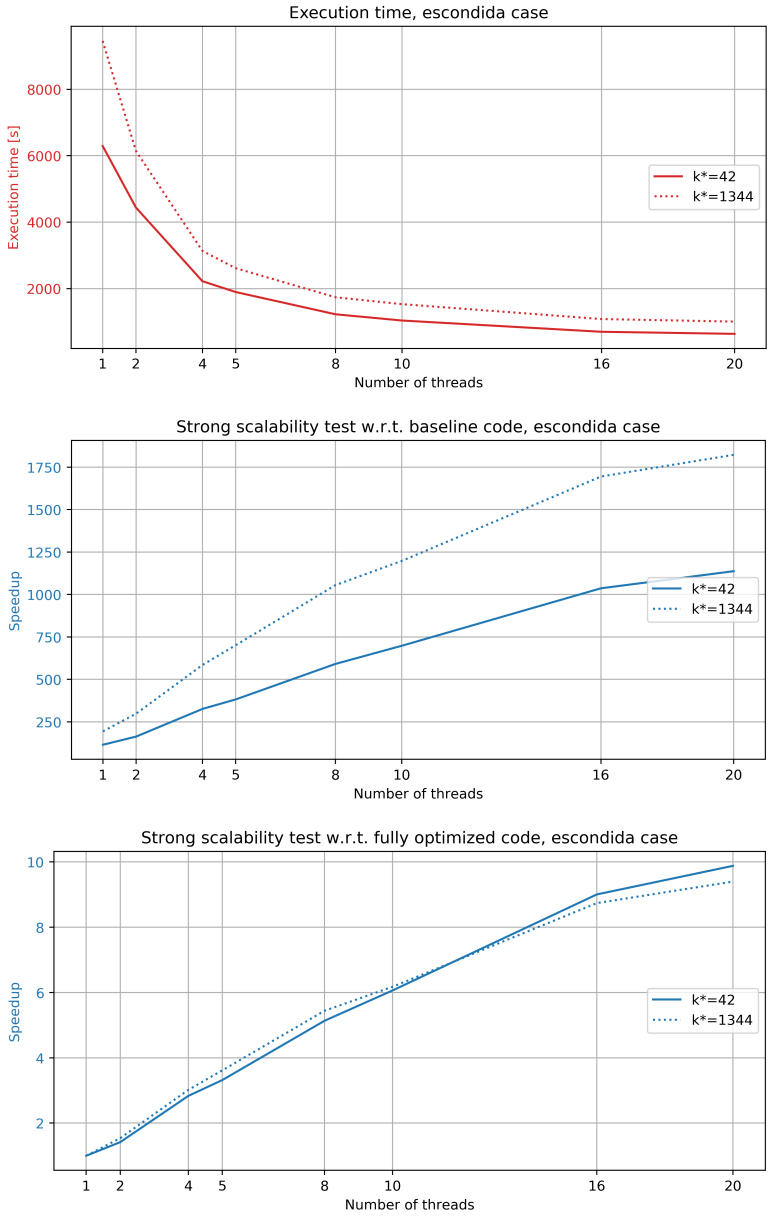

FIGURE 5.21: Execution time [seconds] and speedup results for *escondida* scenario using parallel LVA-based SISIM code.

Version	*swiss-roll* $k^* = 1000$ time [seconds]	*swiss-roll* $k^* = 1000$ speedup	*escondida* $k^* = 1344$ time [seconds]	*escondida* $k^* = 1344$ speedup
Baseline	45060	1×	1833020	1×
Parallel except NS 1 thread	12132	3.71×	16438	111.51×
Parallel except NS 20 threads	8615	5.23×	8596	213.24×
Parallel except NS + KDTree optimized 20 threads	1682	26.77×	1648	1111.91×
Parallel with NS + KDTree optimized 20 threads	792	56.89×	1006	1822.08×

TABLE 5.3: Contribution to speedup of neighbour search (NS) acceleration and paralellization on the *swiss-roll* and *escondida* scenarios. The initial parallel code didn't include parallel neighbour search, only calculation of distance matrix, embedding and simulation were parallelized. Additionally, for LVA-based SISIM, KDTree was adapted and included for execution.

5.4 Analysis

According to the results presented in Section 5.3, two aspects of the proposed implementation are reviewed in detail in this section: accuracy and efficiency.

5.4.1 Accuracy

In terms of accuracy, all parallel codes match exactly the baseline results (assuming the same parameters and the same neighbour search method), regardless of the number of threads used in the execution. This level of accuracy is obtained as result of the explicit replication of the random path simulations, although re-ordering non-conflicting nodes in order to allow parallel simulation of nodes in the same level.

Particularly for LVA-based codes, by decreasing the values of control variables k^{search} and k^{cova} (which define the number of dimensions to use for neighbour search and covariance distance calculation), both parallel and baseline codes can achieve faster results, but with lower accuracy values (compared against larger values of control variables). It will be a final user's decision if he or she can tolerate lower levels of accuracy. In case of LVA-based `sgs-lva` (Figure 5.20), the accuracy loss comparing executions with $k^* = 1000$ versus $k^* = 32$ is 44.19% measured as the mean absolute percentage error from Equation (3.4).

In order to visualize the effect of a decrease in these control variables, early experiments were calculated using initial versions of the parallel codes `sgs-lva` and `sisim-lva`. Tables 5.4, 5.5 and 5.6 show the elapsed time, speedup and accuracy of several simulations

using different values of k^{search} and k^{cova}. The version of parallel code corresponds to an adaptation of the parallel simulation method described in Chapter 4, without the parallel neighbour strategy and also without optimized KDTree code (third row of Table 5.3). We can observe that reductions in k^{cova} have higher impact than reductions in k^{search} in the overall numerical accuracy. As mentioned before, the final user should decide if lower values of the control variables are allowed, but these results indicate that lower values of k^{search} should be selected instead of k^{cova}.

k^{search} \ k^{cova}	32	64	125	250	500	1000
32	837.11	822.05	844.21	829.43	831.87	870.28
64	810.21	1138.59	1150	1151	1155	1137
125	863.7	1064.45	1647.53	1630	1692	1712
250	794.38	1218.87	1749	2868.92	2831	3043
500	884.9	1090.6	1652	2963	4955	5129
1000	857.9	1098.12	1653	2859	4853	8615

TABLE 5.4: Execution time [seconds] of *swiss-roll* scenario with 20 threads and different values of control dimensions k^{search} and k^{cova}.

k^{search} \ k^{cova}	32	64	125	250	500	1000
32	52.62	53.59	52.18	53.11	52.95	50.62
64	54.37	38.69	38.30	38.27	38.14	38.74
125	51.00	41.38	26.73	27.02	26.03	25.73
250	55.45	36.14	25.18	15.35	15.56	14.47
500	49.78	40.39	26.66	14.86	8.89	8.58
1000	51.35	40.11	26.65	15.40	9.07	5.11

TABLE 5.5: Speedup of *swiss-roll* scenario with 20 threads and different values of control dimensions k^{search} and k^{cova}.

k^{search} \ k^{cova}	32	64	125	250	500	1000
32	44.19	37.27	27.45	19.88	13.97	8.48
64	44.19	38.99	28.52	20.25	14.1	8.32
125	44.19	38.99	27.86	18.16	11.58	5.72
250	44.19	38.99	27.86	16.32	9.18	3.14
500	44.19	38.99	27.86	16.32	7.96	1.81
1000	44.19	38.99	27.86	16.32	7.96	0.00

TABLE 5.6: Mean Absolute Percentage Error [%] (MAPE from Eq. (3.4)) of *swiss-roll* scenario with 20 threads and different values of control dimensions k^{search} and k^{cova}.

In case of LVA-based `sisim-lva` (Figure 5.21), the accuracy loss comparing executions with $k^* = 1344$ versus $k^* = 42$ is 11.07% measured as the multiclass false negative rate from Equation (3.9). Nonetheless, the proposed parallel codes deliver the same results as the baseline for any fixed value of the control variables. Note that using the new parallel codes, simulations are executed much faster. Therefore, performing computations for low values of the control variables in search of a reduction of the execution time becomes meaningless when the accuracy loss is high.

Similarly to `sgs-lva` case, early experiments were calculated using initial versions of the parallel code `sisim-lva`. Tables 5.7, 5.8 and 5.9 show the elapsed time, speedup and accuracy of several simulations using different values of k^{search} and k^{cova}. Unlike the previous case, if lower values of the control variables are allowed, these results indicate that lower values of k^{cova} should be selected instead of k^{search}. The reason of this behaviour is that categorical values are simulated in this case, so numerical accuracy is less sensitive to variations in the parameters, which is not the case using continuous values.

k^{search} \ k^{cova}	42	84	168	336	672	1344
42	1073,47	1220,85	1073,8	1086,65	1112,97	1179,43
84	1098,63	1495,45	1432,43	1456,27	1486,73	1495,97
168	1174,55	1514,7	2283,69	2242,73	2238,97	2270,47
336	1209,34	1535,82	2369,87	3420,79	3405,61	3470,72
672	1282,15	1604,53	2349,54	3464,45	5450	3746
1344	1283,06	1668,35	2402,35	3500,69	5588	8596

TABLE 5.7: Execution time [seconds] of *escondida* scenario with 20 threads and different values of control dimensions k^{search} and k^{cova}.

k^{search} \ k^{cova}	42	84	168	336	672	1344
42	1707,57	1501,43	1707,04	1686,85	1646,96	1554,16
84	1668,46	1225,73	1279,66	1258,71	1232,92	1225,31
168	1560,61	1210,15	802,66	817,32	818,69	807,33
336	1515,72	1193,51	773,47	535,85	538,24	528,14
672	1429,65	1142,40	780,16	529,09	336,33	489,33
1344	1428,63	1098,70	763,01	523,62	328,03	213,24

TABLE 5.8: Speedup of *escondida* scenario with 20 threads and different values of control dimensions k^{search} and k^{cova}.

k^{search} \ k^{cova}	42	84	168	336	672	1344
42	11.07	8.87	8.92	9.06	8.87	9.04
84	8.46	8.25	8.53	8.51	8.76	8.56
168	7.51	6.85	6.54	6.59	6.66	6.67
336	7.18	6.26	6.18	6.81	6.23	6.36
672	6.55	7.89	5.97	5.64	5.81	4.45
1344	6.28	5.97	5.95	4.86	3.91	0.00

TABLE 5.9: Multiclass False Negative Rate [%] (MFNR from Eq. (3.9)) of *escondida* scenario with 20 threads and different values of control dimensions k^{search} and k^{cova}.

5.4.2 Efficiency

In terms of the achieved efficiency by the proposed parallelization, in Tables 5.12 and 5.13 we can observe a detailed profile of the refactored codes using 16 OpenMP threads in *sgsim* and *sisim*, and 20 OpenMP threads in *swiss-roll* and *escondida* scenarios,

all executions using a maximum of 48 neighbours per simulation, and $k^* = 1000$ and $k^* = 1344$ respectively on each LVA-based scenario. It is worth mentioning that these tables are very different from the initial baseline profiling in Tables 5.10 and 5.11, where the most relevant part was the neighbour calculation plus simulation as a whole. Now, the relevant parts for non LVA-based scenarios are neighbours calculations for `sgsim` and, again, simulation for `sisim`. For LVA-based scenarios, the relevant parts are distance calculation, neighbours calculation and simulation (embedding building is not relevant after applying optimizations in matrix operations and memory accesses). On non LVA-based scenarios, both results have efficiencies of 62% and 63% respectively. On LVA-based scenarios, both results have efficiencies of 42% and 44% respectively. These results are acceptable for these applications since the baseline code and algorithms were not designed originally to run on parallel architectures. Additionally, Figures 5.22 and 5.23 contain efficiency percentages for non LVA-based scenarios for values of up to 16 threads, and LVA-based scenarios for values up to threads.

Execution step	$\%t_{total}$ (*sgsim*)	$\%t_{total}$ (*sisim*)
Read params	1.311	0.004
Neighbours calculation	68.965	26.092
Simulation	24.535	72.216
Write out	5.187	0.012

TABLE 5.10: Profiling of executions [% of elapsed time] with baseline non-LVA codes. Left: *sgsim* scenario using baseline non LVA-based SGSIM with 50×10^6 domain points, 48 maximum neighbours for kriging and 16 threads; total elapsed time was 11 minutes and 25 seconds. Right: *sisim* scenario using baseline non LVA-based SISIM with 50×10^6 domain points, 48 maximum neighbours for kriging and 16 threads; total elapsed time was 37 minutes and 21 seconds.

Execution step	$\%t_{total}$ $k^{cova} = k^{search} = 1000$ (*swiss-roll*)	$\%t_{total}$ $k^{cova} = k^{search} = 1344$ (*escondida*)
Read params	0.012	0.001
Connectivity graph building	0.233	0.001
Distance matrix building	10.341	2.178
Embedding building	9.831	5.134
Neighbours calc. + Sim.	79.576	96.585
Write out	0.006	0.001

TABLE 5.11: Profiling of executions [% of elapsed time] with baseline LVA codes. Left: *swiss-roll* scenario using baseline LVA-based SGS with 1.7×10^6 domain points, 48 maximum neighbours for kriging and 1000 landmarks; total elapsed time was 12 hours and 31 minutes. Right: *escondida* scenario using baseline LVA-based SISIM with 1.7×10^6 domain points, 48 maximum neighbours for kriging and 1344 landmarks; total elapsed time was 509 hours and 17 minutes (21 days and 5 hours).

Execution step	$\%t_{total}$ $(sgsim)$	$\%t_{total}$ $(sisim)$
Read params	1.807	0.101
Neighbours calculation	46.240	15.111
Simulation	43.325	84.395
Write out	8.695	0.391

TABLE 5.12: Profiling of executions [% of elapsed time] with parallel refactored non-LVA codes using 16 threads. Left: *sgsim* scenario using accelerated non LVA-based SGSIM with 50×10^6 domain points, 48 maximum neighbours for kriging; total elapsed time was 6 minutes and 10 seconds. Right: *sisim* scenario using accelerated non LVA-based SISIM with 50×10^6 domain points, 48 maximum neighbours for kriging; total elapsed time was 21 minutes and 26 seconds.

Execution step	$\%t_{total}$ $k^{cova} = k^{search} = 1000$ $(swiss\text{-}roll)$	$\%t_{total}$ $k^{cova} = k^{search} = 1344$ $(escondida)$
Read params	0.03	0.01
Connectivity graph building	0.09	0.06
Distance matrix building	61.81	51.40
Embedding building	3.99	4.59
Neighbours calculation	15.20	10.22
Simulation	18.84	33.71
Write out	0.05	0.01

TABLE 5.13: Profiling of executions [% of elapsed time] with parallel refactored LVA codes using 20 OpenMP threads. Left: *swiss-roll* scenario using accelerated LVA-based SGS with 1.7×10^6 domain points, 48 maximum neighbours for kriging and 1000 landmarks; total elapsed time was 1 hour 47 minutes. Right: *escondida* scenario using accelerated LVA-based SISIM with 1.7×10^6 domain points, 48 maximum neighbours for kriging and 1344 landmarks; total elapsed time was 2 hours 37 minutes.

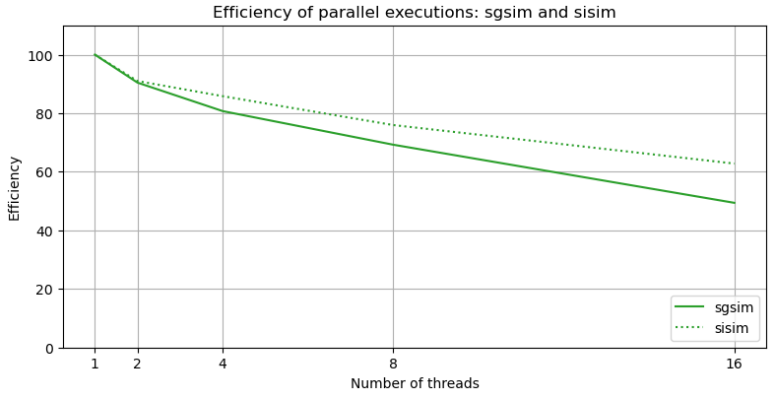

FIGURE 5.22: Efficiency of parallel executions for *sgsim* and *sisim* scenarios compared against theoretical maximum speedup. In this case a maximum of 16 threads are used in order to compare results with a previous work from Section 4.3.

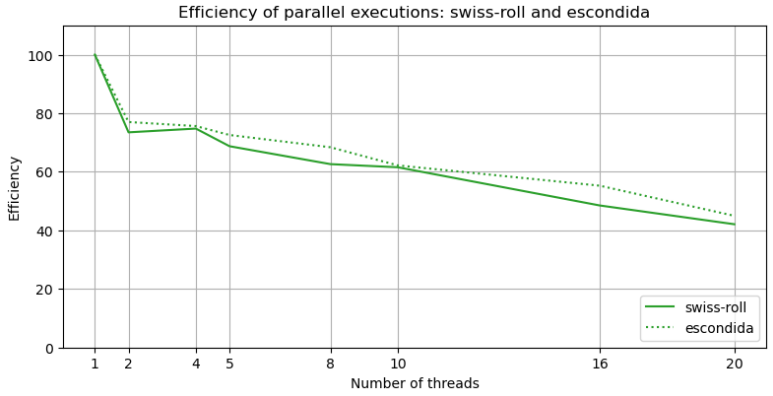

FIGURE 5.23: Efficiency of parallel executions for *swiss-roll* and *escondida* scenarios compared against theoretical maximum speedup.

In Figure 5.24 we can observe an execution profile obtained with Extrae/Paraver tools of *sgsim* non-LVA scenario using 16 threads. Each stage of the execution can be identified approximately in this figure, using the "State as is" visualization, which depicts the state of execution of each thread. As stated in Table 5.10, the largest portions corresponds to neighbours calculation and simulation. Figure 5.25 depicts a comparison between the baseline from Chapter 4, and adapted non-LVA code `sgsim` with parallel neighbour calculation. We can observe that the neighbour calculation part was accelerated by using the parallel strategy which improves the overall execution. A small difference can be observed between simulation stage on both profiles, which is explained by the usage of a covariance lookup table in the baseline code. The usage of this lookup table will be included in future versions of the new code.

In Figure 5.26 we can observe an execution profile obtained with Extrae/Paraver tools of *swiss-roll* LVA scenario using 20 threads. As stated in Table 5.13, the largest portion corresponds to distance matrix building, followed by neighbours calculation and simulation. Additionally, in Figure 5.27 we can observe the cache misses at the three architectural levels L1, L2 and L3. We can observe a higher number of cache misses in L1 for distance matrix building and simulation stages, which indicate more memory operations accesing non-reusable locations. For L2 and L3 levels, distance matrix calculation also contributes with the highest number of cache misses.

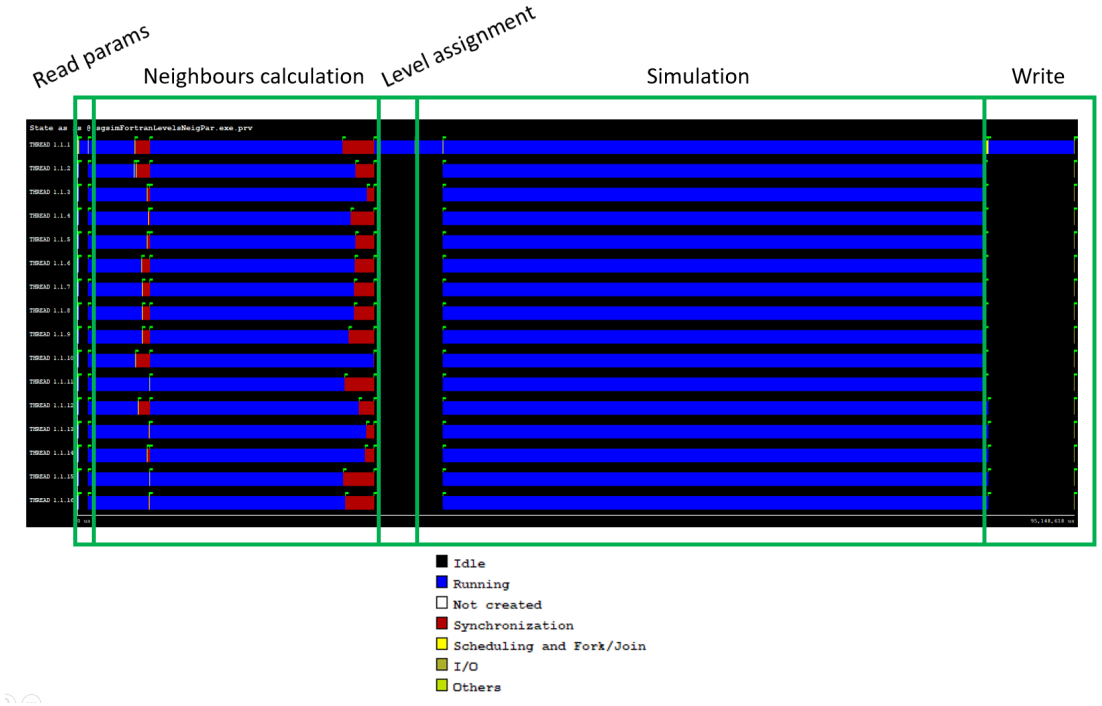

FIGURE 5.24: Profile of *sgsim* non-LVA scenario using `sgsim` parallel code with 16 threads, obtained with Extrae/Paraver tools.

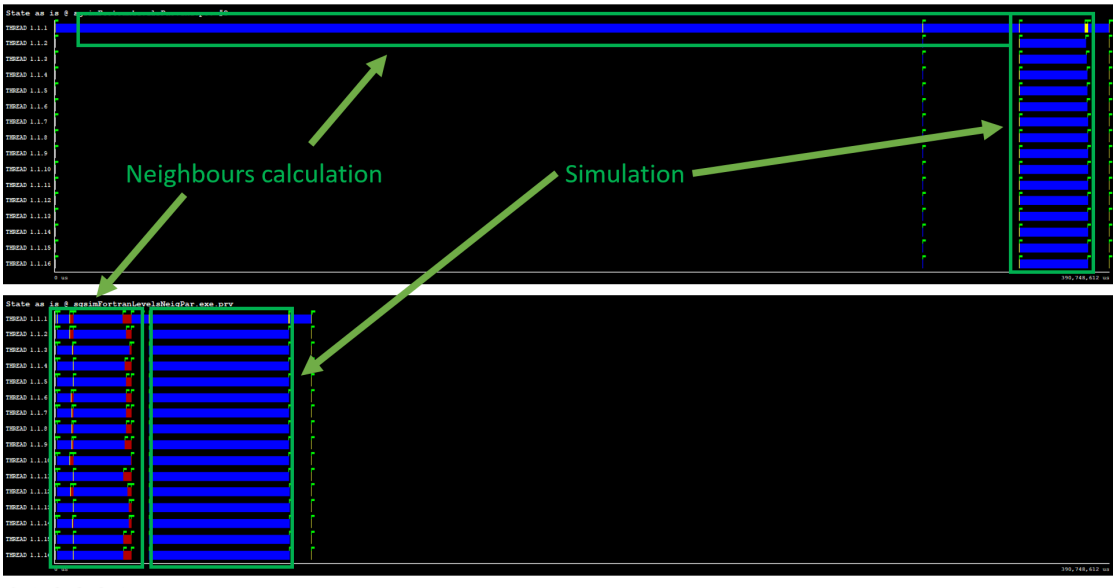

FIGURE 5.25: Comparison of Extrae/Paraver execution profiles between *sgsim* non-LVA scenarios: baseline (top) and with parallel neighbour calculation (bottom), using the same time scale. Baseline execution of simulation stage uses a covariance lookup table, not included yet in the new parallel code.

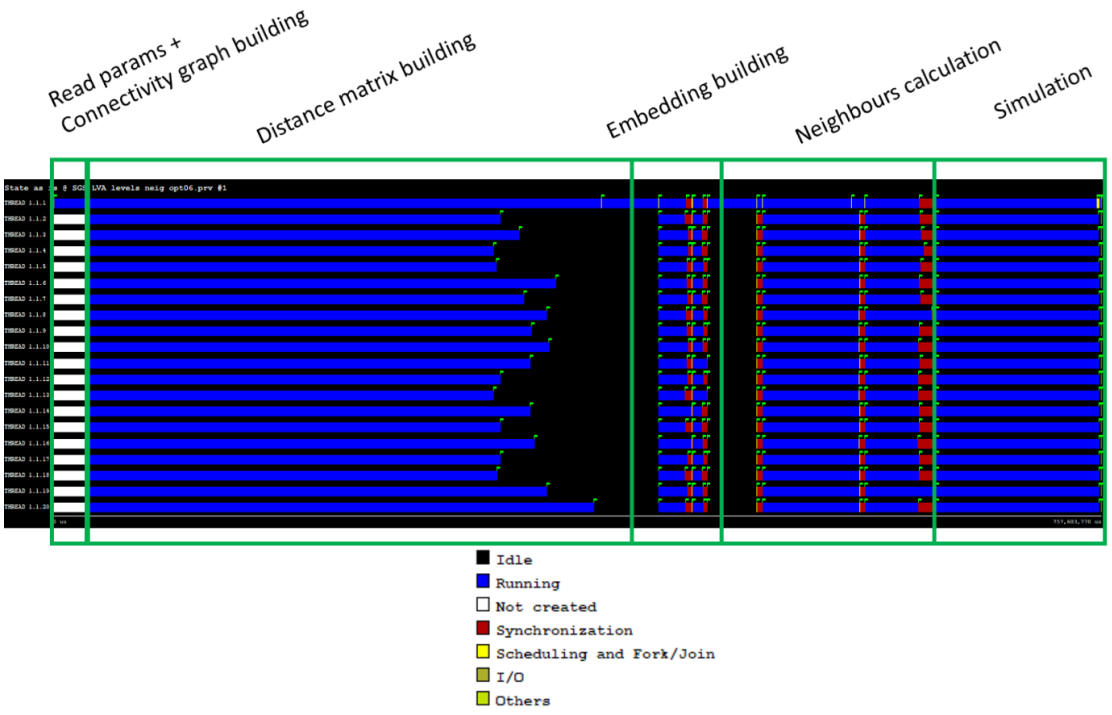

FIGURE 5.26: Profile of *swiss-roll* scenario using `sgs-lva` parallel code with 20 threads, obtained with Extrae/Paraver tools.

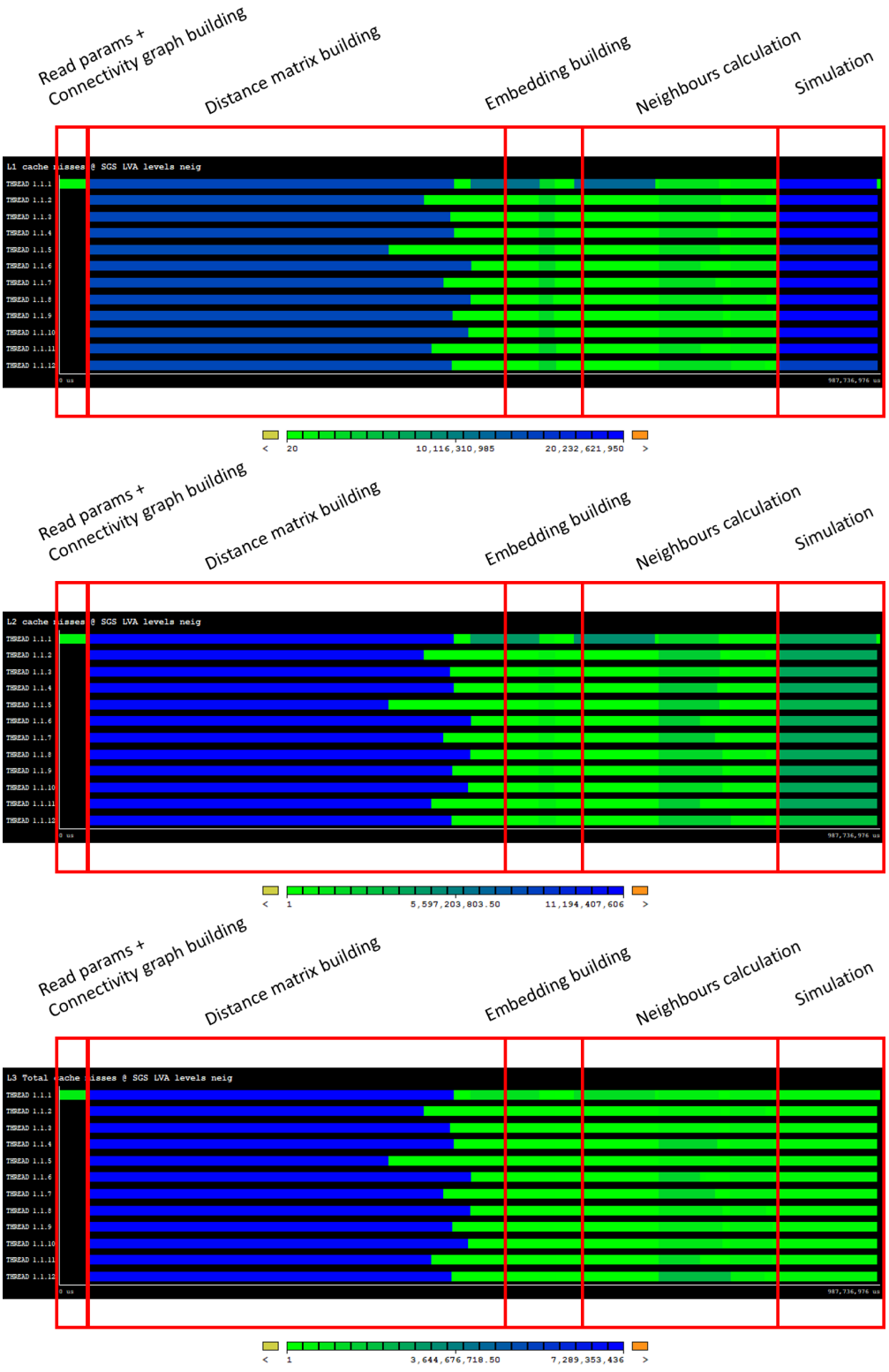

FIGURE 5.27: L1 (top), L2 (middle) and L3 (bottom) cache misses of *swiss-roll* scenario using `sgs-lva` parallel code with 12 threads, obtained with Extrae/Paraver tools.

The achieved speedup and efficiency obtained on both scenarios can be explained mostly by three factors: 1) intrinsic efficiency of external libraries (C++ Boost and Intel MKL), 2) different sizes in the initial conditioning datasets, and 3) amount of work performed in the simulation step.

Factor 1 impacts only on LVA-based scenarios, and depends explicitly on the performance delivered by the external libraries (this topic is covered in Chapter 6). Even though further optimizations can be done in these libraries, it is left out of the scope of this work. In any case, on both scenarios we obtained similar performance since these executions only depend on the number of domain points, number of landmarks and LVA additional parameters, which remain on the same orders of magnitude across scenarios.

Regarding factor 2, also impacting only on LVA-based scenarios, in Figure 5.28 we can observe the number of points assigned to each level, using a maximum of 48 neighbour points per simulation for both scenarios. In the *swiss-roll* scenario, 396 initial conditioning points are used, which results in a larger point-level curve, and translates into more overhead (resulting in less speedup and efficiency) due to multi-thread contingency and context switching from level to level. On the contrary, in the *escondida* scenario, 2313 initial conditioning points are used, which results in a shorter curve, and consequently less overhead (resulting in more speedup and efficiency). We can infer that a large number of initial conditioning data will generate a short point-level curve with better speedup and efficiency at lower execution time. In Figure 5.29 we can observe different point-level curves for the same scenario with equal parameters, except the number of initial conditioning data points. Curves using 48 maximum neighbours and 4% and 0.25% of initial conditioning data (continuous blue and orange curves) exemplify this phenomenon.

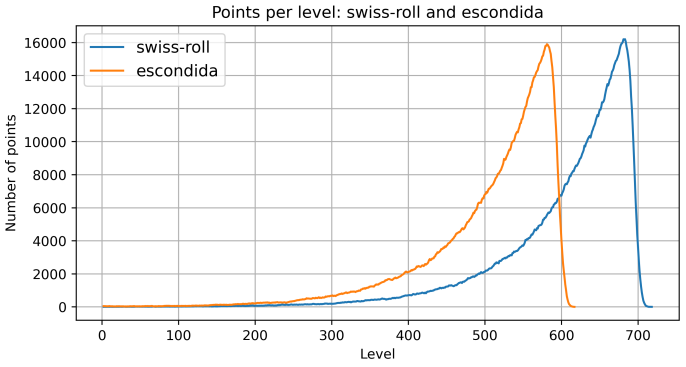

FIGURE 5.28: Points per level on both test scenarios, *swiss-roll* (396 initial conditioning data) and *escondida* (2313 initial conditioning data). All points in the same level are simulated in parallel by P threads.

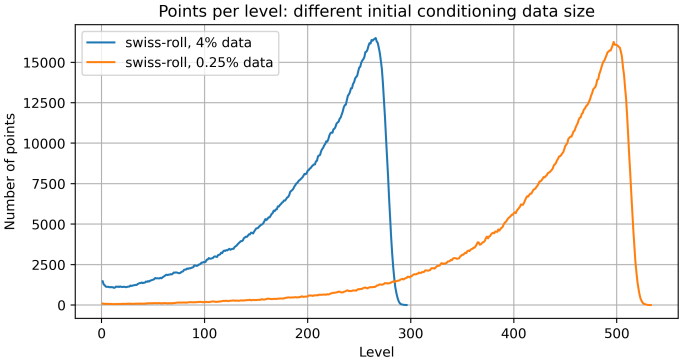

FIGURE 5.29: Points per level on *swiss-roll* scenario using different percentages of initial conditioning data (4% and 0.25%, from a total of 1,728,000 points).

Regarding factor 3, which impacts on both non LVA and LVA-based scenarios, we can identify two cases: keeping constant versus increasing the maximum number of neighbours to use for simulation. Assuming the maximum number of neighbours doesn't change, the amount of work in the simulation step can be increased by assembling and solving more kriging linear systems per simulation point. This situation occurs on non LVA-based and LVA-based SISIM implementations, since for each category, a kriging linear system should be assembled and solved per each simulation point. In *sisim* scenario, 10 categories are simulated, which result in 10× more work per simulation compared against *sgsim* scenario. In *escondida* scenario, 4 categories are simulated, which results in 4× more work per simulation point compared against *swiss-roll* scenario. The increment of work per simulation point doesn't change the form of the point-level curve, but it will impact on performance by delivering better speedup and efficiency values, at higher execution time. Multi-threaded execution becoming more efficient as the problem becomes larger (more computing needed) is a well-known behaviour denoted *weak scalability* (the reader can refer to the definition of Gustafson's law from Hennessy and Patterson [2012] or the original reference from Gustafson [1988]). On the other hand, if the maximum number of neighbours increases, the amount of work in the simulation step will be increased automatically, since the size of the kriging linear systems will increase accordingly. Figure 5.30 shows the speedups obtained using 32 and 64 neighbours as the maximum values for *sgsim* and *sisim* scnearios. Similarly, Figure 5.31 shows the equivalent using 48 and 96 as the maximum for the *swiss-roll* and *escondida* scenarios. On all figures we can observe an increment of the speedup values when higher number of neighbours are used. A slight decline in the speedup trend can be observed only for *escondida* using 96 neighbours, which is caused by overhead in the execution due to multicore contingency and higher memory usage overall.

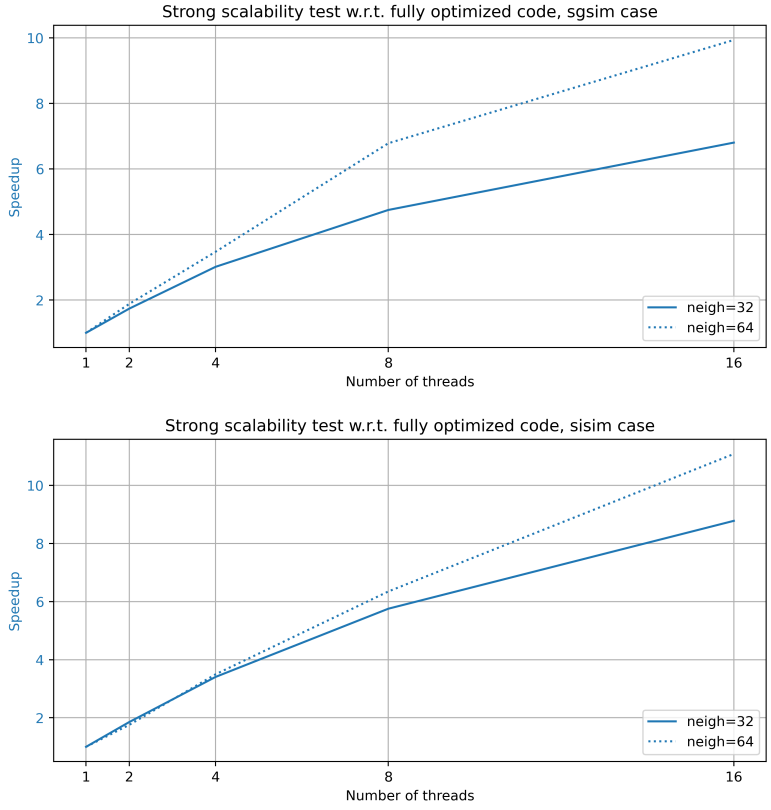

FIGURE 5.30: Speedup results for scenarios *sgsim* and *sisim* using 32 and 64 maximum number of neighbours.

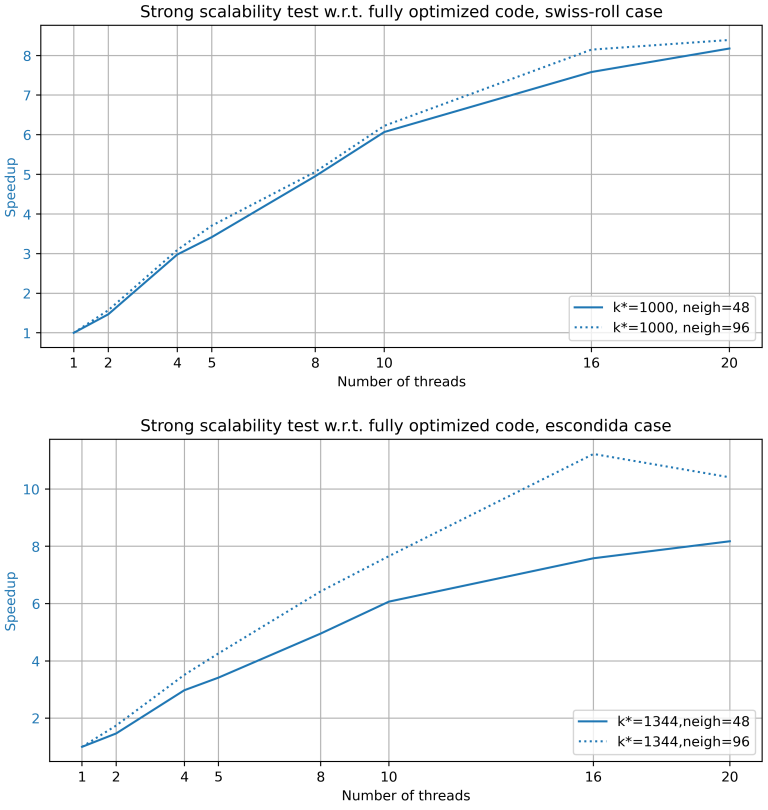

FIGURE 5.31: Speedup results for scenarios *swiss-roll* and *escondida* using 48 and 96 maximum number of neighbours.

5.4.3 Computational resources

In terms of computational resources, it uses all the arrays and data structures similarly to the main algorithm of Chapter 4. Additionally, the parallelization strategy uses a large amount of memory to perform the parallel neighbour search in the current implementation version. The most memory consuming structure is the KDTree tree that should be replicated by each thread in their private memory, which roughly needs the same amount of space to store the embedding coordinates (grid nodes $\times$ landmark nodes $\times$ floating-point type size). For this reason, LVA-based scenarios could only be tested up to 1.7 million approximately, using almost all memory space (120 GB RAM in the test machine). As explained in Chapter 4, several cloud computing providers can offer virtual instances with even larger RAM memory sizes, so this usage pattern is not prohibitive given the current technological trends.

In the next chapter, we will review additional acceleration topics needed only by LVA-based codes. The effects in performance delivered by these additional topics were already included in the results presented in this chapter, but their detailed description was left aside in their own chapter.

Chapter 6

Parallel LVA routines

As described in Algorithms 4 and 5, three extra tasks are needed to execute LVA-based methods: build a connectivity graph, build a distance matrix and build the final embedding to use for further distance calculations. In this chapter we will review acceleration aspects related with the last two tasks, where parallel algebraic operations and parallel shortest path computation are included. Even though these contributions do not represent an advancement in terms of algorithmic contributions, their integration in the parallel LVA-based codes `sgs-lva` and `sisim-lva` represent a significant impact towards their adoption from scholars and practicioners of Geostatistics.

6.1 Algebraic operations

6.1.1 Context

In the baseline implementation of `sgs-lva`[1], several algebraic optimizations can be applied, which can be divided in two groups: memory access optimizations and use of an optimized BLAS implementation. The first group is related with an extensive analysis of memory access in Fortran arrays, which should follow column-major order. Although this optimization is basic, it is worth to mention in its own section, due to the amount of different Fortran codes that will be modified. The second group is related specifically with the embedding building described in Algorithm 8. Usage of Intel MKL matrix-matrix and eigenvalue/eigenvector calculation is presented, in replacement of intrinsic Fortran routines or non-optimized versions of the same algebraic operations.

[1] https://sites.ualberta.ca/$\sim$jbb/LVA_code.html

6.1.2 Memory access optimizations

As described in Algorithm 4, lines 2, 3 and 4 represent the building steps of the embedding $\mathcal{Z}$. In terms of code, the embedding is stored in a two dimensional array denoted `coord_ISOMAP`. This array is used to store intermmediate calculations in the routines `building_distance_matrix` and `building_embedding`, and finally it contains the multidimensional coordinates of each domain point. In lines 13 and 14 of the same algorithm, the array `coord_ISOMAP` is accessed several times to calculate euclidean distances between multidimensional points.

The main problem with this array is that the original allocation instruction is

```
real*8, allocatable, dimension (:,:) :: coord_ISOMAP
...
allocate(coord_ISOMAP(NODES,xyzland))
```

being `NODES` the number of domain points and `xyzland` the number of landmark points. This memory allocation doesn't follow the optimal access pattern when covariance calculation is computed for kriging estimation. Figure 6.1 shows the lines of code used for this task. In lines 6 to 13, euclidean distance between `coord_ISOMAP(ind1,1:dim)` and `coord_ISOMAP(ind2,1:dim)` is computed, and in lines 25 to 30 the squared euclidean distance is computed between data points `coord_ISOMAP(ind1,d_tree+1:dim)` and `coord_ISOMAP(index,d_tree+1:dim)`.

On each of these memory accesses, the pattern is as follow

```
coord_ISOMAP(i,j)
...
coord_ISOMAP(i,j+k)
```

which doesn't follow the column-major order defined in Fortran, contrary to

```
coord_ISOMAP(j  ,i)
...
coord_ISOMAP(j+k,i)
```

and consequently generates many more cache memory misses, increasing the overall execution time.

The solution implemented to avoid this issue was to introduce a transposed array denoted `coord_ISOMAP_trans`, allocated as

```
real*8, allocatable, dimension (:,:) :: coord_ISOMAP_trans
...
allocate(coord_ISOMAP_trans(xyzland,NODES))
```

```
01 do i=1,nclose
02    ind1=results(i).idx
03    do j=i,nclose
04        ind2=results(j).idx
05        dist=
06            sqrt(
07                sum(
08                    (
09                        coord_ISOMAP(ind1,1:dim) -
10                        coord_ISOMAP(ind2,1:dim)
11                    )**2
12                )
13            )
14        call cova3_1D(dist,1,nst,MAXNST,c0,it,cc,aa,cmax,
15                      a(neq*(i-1)+j))
16        a(neq*(j-1)+i) = a(neq*(i-1)+j)
17    end do
18 end do
19 ...
20 do i=1,nclose
21    ind1 = results(i).idx
22    vra(i,:)=sim(ind1,:)
23    if(d_tree/=dim) then
24        dist=results(i).dis +
25        sum(
26            (
27                coord_ISOMAP(ind1,d_tree+1:dim) -
28                coord_ISOMAP(index,d_tree+1:dim)
29            )**2
30        )
31    else
32        dist=results(i).dis
33    end if
34    dist=sqrt(dist)
35    call cova3_1D(dist,1,nst,MAXNST,c0,it,cc,aa,cmax,r(i))
36 end do
```

FIGURE 6.1: Original usage of array `coord_ISOMAP` in baseline code `sgs-lva` for covariance calculation used by kriging estimation.

and each memory access to `coord_ISOMAP` is changed for compliant column-major order memory access to `coord_ISOMAP_trans`, as depicted in Figure 6.2. Early tests showed speedup improvements between 1.5× and 3× only introducing the memory access optimization. Although it consists in a simple and basic optimization, it is an important step in any performance improvement analysis, specially if baseline legacy sub-optimal codes are used as starting points.

```
01 do i=1,nclose
02    ind1=results(i).idx
03    do j=i,nclose
04        ind2=results(j).idx
05        dist=
06            sqrt(
07                sum(
08                    (
09                        coord_ISOMAP_trans(1:dim,ind1) -
10                        coord_ISOMAP_trans(1:dim,ind2)
11                    )**2
12                )
13            )
14        call cova3_1D(dist,1,nst,MAXNST,c0,it,cc,aa,cmax,
15                      a(neq*(i-1)+j))
16        a(neq*(j-1)+i) = a(neq*(i-1)+j)
17    end do
18 end do
19 ...
20 do i=1,nclose
21    ind1 = results(i).idx
22    vra(i,:)=sim(ind1,:)
23    if(d_tree/=dim) then
24        dist=results(i).dis +
25        sum(
26            (
27                coord_ISOMAP_trans(d_tree+1:dim,ind1) -
28                coord_ISOMAP_trans(d_tree+1:dim,index)
29            )**2
30        )
31    else
32        dist=results(i).dis
33    end if
34    dist=sqrt(dist)
35    call cova3_1D(dist,1,nst,MAXNST,c0,it,cc,aa,cmax,r(i))
36 end do
```

FIGURE 6.2: Optimized usage of array `coord_ISOMAP_trans` in accelerated code `sgs-lva` for covariance calculation used by kriging estimation.

6.1.3 Intel MKL implementation

The second group of algebraic optimizations is related with the usage of specialized libraries to perform matrix-matrix and eigenvalue/eigenvector computations. Particularly, we introduce the usage of Intel MKL library in the baseline code `sgs-lva`. As described before, the main routine that computes this kind of operations is related with the embedding building depicted in Algorithm 8. At code level, this algorithm is implemented as depicted in Figure 6.3. In line 4 a matrix-matrix multiplication is applied using the native operation `matmul`, coupled with `transpose` ($\mathbf{B}^T\mathbf{B}$ from line 7 of Algorithm 8). Immediately after that, eigenvalue/eigenvector calculation is computed using the routine `eig`, which internally calls LAPACK [Anderson et al., 1999] method `DSYEV`[2].

[2] http://www.netlib.org/lapack/explore-html/d2/d8a/group__double_s_yeigen_ga442c43fca5493590f8f26cf42fed4044.html

Finally, a sequence of matrix-vector multiplications are applied in lines 9 to 14 between arrays `coord_ISOMAP` and `vectors` ($\mathbf{Bv}_i$ from lines 8 and 9 of Algorithm 8).

```
01 subroutine MDS_ISOMAP(...)
02     ...
03     !!! B^T*B
04     subB2=matmul(transpose(coord_ISOMAP),coord_ISOMAP)
05     ...
06     call eig(subB2,evalues,vectors,ubound(subB2,1))
07     ...
08     !!! B*V
09     do i=1,NODES
10         do j=1,dim
11             vec_temp(j)=sum(coord_ISOMAP(i,:)*vectors(:,j))
12         end do
13         coord_ISOMAP(i,1:dim) = vec_temp
14     end do
15     ...
16 end subroutine MDS_ISOMAP
```

FIGURE 6.3: Original usage of matrix-matrix and eigenvalue/eigenvector calculation in baseline code `sgs-lva` for embedding $\mathcal{Z}$ generation.

In these three cases, a routine from Intel MKL library is used instead, specifically `DGEMM`[3] and `DSYEV`. In case of `DSYEV` the change is transparent since it is already calling the same LAPACK method, so only changes in the `Makefile` are needed (remove LAPACK objects compiled from source and add link to the library instead). In case of `DGEMM`, Figure 6.4 depicts its usage instead of line 3 from Figure 6.3, and Figure 6.5 depicts its usage instead of lines 7 to 12 from Figure 6.3.

```
1 M=xyzland
2 K=NODES
3 N=xyzland
4 ALPHA=1.0
5 BETA=0.0
6
7 CALL MKL_SET_NUM_THREADS(num_threads)
8 CALL DGEMM('T','N',M,N,K,ALPHA,
9     coord_ISOMAP,K,coord_ISOMAP,K,BETA,subB2,M)
```

FIGURE 6.4: Optimized usage of matrix-matrix product $\mathbf{B}^T\mathbf{B}$ calculation in baseline code `sgs-lva` for embedding $\mathcal{Z}$ generation.

[3] http://www.netlib.org/lapack/explore-html/d1/d54/group__double__blas__level3_gaeda3cbd99c8fb834a60a6412878226e1.html

```
01 M=NODES
02 K=xyzland
03 N=dim
04 ALPHA=1.0
05 BETA=0.0
06 allocate(coord_ISOMAP_out(M,N))
07
08 CALL MKL_SET_NUM_THREADS(num_threads)
09 CALL DGEMM('N','N',M,N,K,ALPHA,
10     coord_ISOMAP,M,vectors,K,BETA,coord_ISOMAP_out,M)
11
12 coord_ISOMAP(:,1:dim) = coord_ISOMAP_out
13 deallocate(coord_ISOMAP_out)
```

FIGURE 6.5: Optimized usage of matrix-matrix product $\mathbf{BV}$ calculation in baseline code `sgs-lva` for embedding $\mathcal{Z}$ generation.

Alternative methods can be applied to compute the eigenvectors and eigenvalues of $\mathbf{B}^T\mathbf{B}$. The first alternative uses the symmetric rank-k operation `DSYRK`[4] applied on $\mathbf{B}$, which allows us to obtain the upper or lower part of $\mathbf{B}^T\mathbf{B}$. With this operation we can skip the first matrix-matrix multiplication using `DGEMM` to compute $\mathbf{B}^T\mathbf{B}$, and posteriorly we can still use `DSYEV` with the corresponding parameter `UPLO` equal to 'U' or 'L' accordingly. A second alternative is based on the singular value decomposition operation `DGESVD`[5] applied on $\mathbf{B}$. With this operation we can skip the first matrix-matrix multiplication `DGEMM` and also the posterior call to `DSYEV`. By computing the singular values and vectors of $\mathbf{B} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$, the eigenvalues and vectors of $\mathbf{B}^T\mathbf{B}$ can be computed as follows:

$$\begin{aligned} \mathbf{B}^T\mathbf{B} &= \mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \\ &= \mathbf{V}(\mathbf{\Sigma}^2)\mathbf{V}^T \end{aligned} \tag{6.1}$$

Comparative tests using both alternative methods are presented in the next section.

6.1.4 Results

In order to measure the performance of the parallel methods, we will use the *swiss-roll* scenario described in Section 5.3.2 with parameters described in Table 6.1. With these parameters, the matrix $\mathbf{B}$ of Algorithm 8 will have dimensions $N = 1,728,000$ and $n = 1000$.

All runs were executed in a single-node virtual machine with Ubuntu 18.04.5 LTS with 32-cores Intel(R) Xeon(R) CPU at frequency 2.30GHz and a main memory of 118 GB

[4] http://www.netlib.org/lapack/explore-html/d1/d54/group__double__blas__level3_gae0ba56279ae3fa27c75fefbc4cc73ddf.html

[5] http://www.netlib.org/lapack/explore-html/d1/d7e/group__double_g_esing_ga84fdf22a62b12ff364621e4713ce02f2.html

Parameter	*swiss-roll*
Domain $nx \times ny \times nz$	$120 \times 120 \times 120$
LVA field $nx \times ny \times nz$	same as domain
Graph connectivity (offset)	1
Landmarks $nx \times ny \times nz$	$10 \times 10 \times 10$
k^{cova}	1000
k^{search}	1000
Max nodes for simulation	48
Kriging	SK
Number of structures (type)	1 (exponential)

TABLE 6.1: Default parameters for *swiss-roll* to test performance of Intel MKL routines.

RAM. The cloud provider in this case is Google Cloud Platform [Google, 2021], and the virtual machine is a `n1-standard-32` with CPU platform Intel Haswell. All Fortran programs were compiled using Intel Fortran `ifort` version 2021.2.0 supporting OpenMP version 5.1, with options -fpp -mkl -qopenmp -O3 -mtune=native -march=native.

In terms of performance improvement, by default the Intel MKL library will use the maximum number of parallel threads available in the operating system, which is why we set a guardrail by setting `MKL_SET_NUM_THREADS` equal to the number of threads defined at execution time in the environment variable `OMP_NUM_THREADS`. By using a single-thread, the baseline execution time is 566 seconds for $\mathbf{B}^T\mathbf{B}$ from line 7 of Algorithm 8, and 6362 seconds for $\mathbf{BV}$ from lines 8 and 9 of Algorithm 8. Figure 6.6 depicts the execution time using the optimized code based on Intel MKL library implementation of `DGEMM`. Figures 6.8 and 6.7 depicts speedup values compared against the baseline single-thread execution. We can observe that a considerable gain in performance (7.0× and 69× using 1 OpenMP thread, and 81× and 557× using 16 OpenMP threads) is achieved just by using Intel MKL for these two algebraic operations. If a large number of threads are used, it is more difficult to keep increasing the speedup since less work is assigned to each thread and more overhead is generated to synchronize all threads. This fact can be observed in the profiles of Figures 6.9, where the computation of $\mathbf{B}^T\mathbf{B}$ and $\mathbf{BV}$ contains several synchronization steps at the end of each thread execution (red states in the profiles). For completeness, Figure 6.10 depicts the computation of the eigenvalues and eigenvectors using `DSYEV`, which shows several calls to inner routines such as `DGEMM` and `DTRMM`.

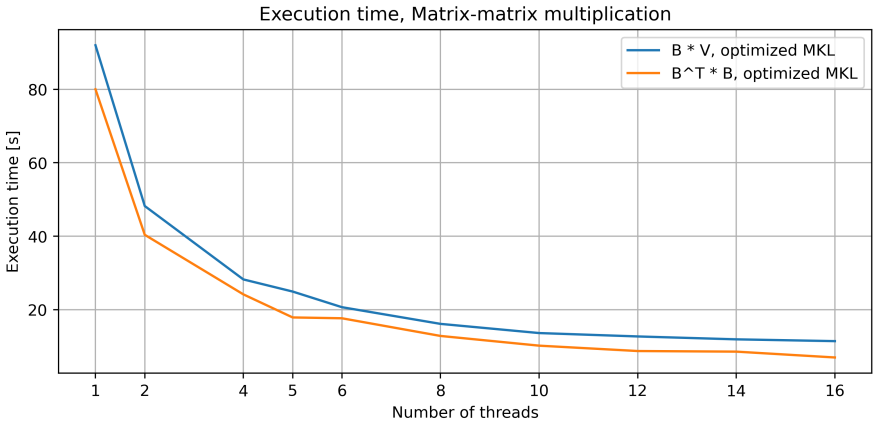

FIGURE 6.6: Execution time of optimized matrix-matrix product calculations $\mathbf{B}^T\mathbf{B}$ and $\mathbf{BV}$.

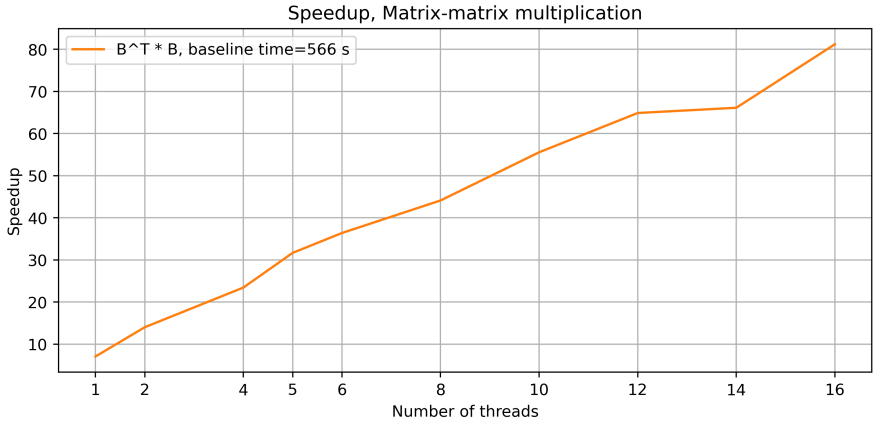

FIGURE 6.7: Speedup of optimized matrix-matrix product calculation in the case $\mathbf{B}^T\mathbf{B}$.

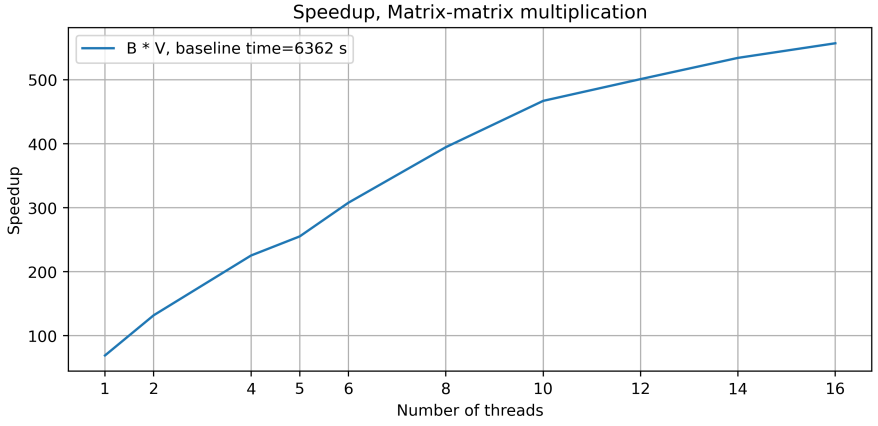

FIGURE 6.8: Speedup of optimized matrix-matrix product calculation in the case $\mathbf{BV}$.

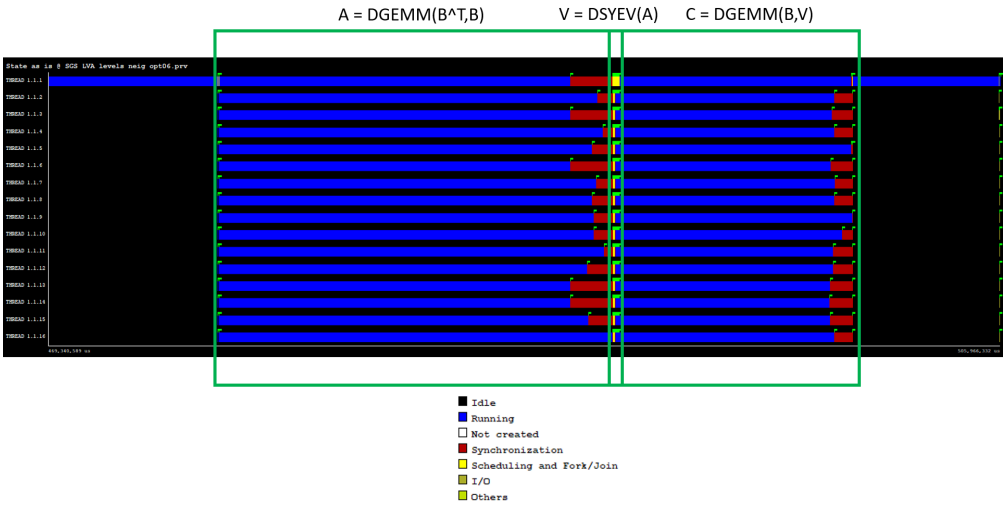

FIGURE 6.9: Profile of embedding building routines obtained with Extrae/Paraver, involving matrix-matrix product calculations $\mathbf{B}^T\mathbf{B}$ and $\mathbf{BV}$.

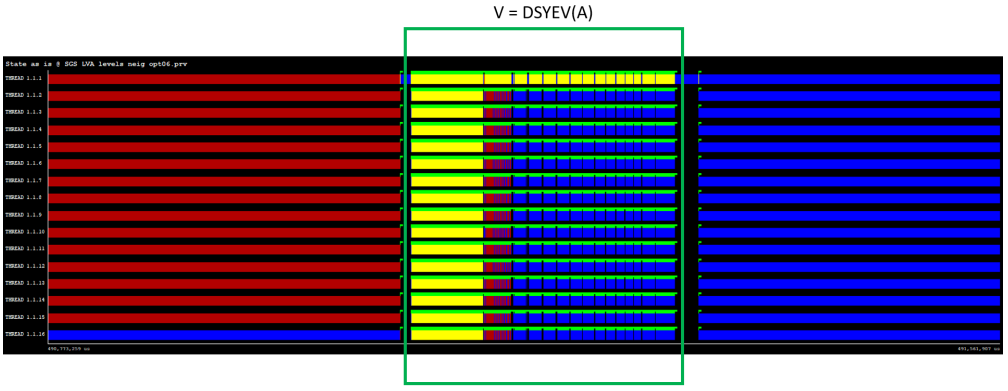

FIGURE 6.10: Zoom in the profile of Figure 6.9, with focus on the eigenvalue and eigenvector calculation with `DSYEV`.

As an end-to-end performance test, the MKL library is integrated in the code `sgs-lva` from Chapter 5, in order to measure the overall performance impact in the application. The scenario simulated in this test is *swiss-roll* using 17280 sample conditioning data points. Two versions of this code were executed using 20 threads, with and without the integrated MKL code. The version without MKL integration for embedding calculation uses the original code from Figure 6.3. The execution time for embedding calculation without MKL integration was 6321 seconds, and the execution time obtained with MKL integration was 68 seconds. Regarding the overall execution time, without MKL integration it was 7507 seconds, and with MKL integration it was 625 seconds.

Regarding the alternative methods using `DSYRK` and `DGESVD`, although both of them can improve some aspects of the computation, no significant improvements are obtained empirically. In the first alternative, the use of `DSYRK` effectively reduces the number of operation by half compared against `DGEMM`, which reduces the execution time according to Table 6.2. In the second alternative, the usage of `DGESVD` improves the numerical stability of the problem, however it increases the overall execution time in this part of the application. Based on these results, the usage of `DSYRK` can improve the execution of this part, making it a feasible alternative in case of even more acceleration is needed in this part or larger scenarios are being studied. It is left as future work the usage of improved versions of singular value decomposition instead of `DGESVD`, such as `HQRRP` [Martinsson et al., 2017] or `randUTV` [Martinsson et al., 2019].

N Threads	Elapsed time [s] N threads `DGEMM`+`DSYEV`	Elapsed time [s] N threads `DSYRK`+`DSYEV`	Elapsed time [s] N threads `DGESVD`
1	143.346	79.142	196.611
2	71.181	45.343	104.482
4	41.463	26.135	71.649
8	34.743	20.099	58.915
16	22.565	18.198	62.089

TABLE 6.2: Execution time (seconds) for computing eigenvalues and eigenvectors of $\mathbf{B}^T\mathbf{B}$ using three methods with different MKL routines.

The integration of the MKL library accelerates dramatically the execution of the embedding calculation step, speeding up the overall execution as a consequence (12× in the end-to-end test). Its SIMD computation model and the multithreading capabilities, makes this library suitable for being integrated in any further test and additional implementations (in the next section, the MKL integration is active on all tests).

6.2 Single Source Shortest Path

6.2.1 Context

A brief description of the sequential implementation used in the baseline code is included in Section 2.3.2.3. More specifically, the current baseline codes `sgs-lva` and `sisim-lva` execute a system call to a compiled C++ application with the sequential implementation of the Boost Library. This application implements the steps described in Algorithm 7, by using routines from the Boost Graph Library:

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
```

In line 6 of this algorithm, the routine `dijkstra_shortest_paths` is called in order to return the shortest distances through an undirected graph $\mathbf{G}$ from node i to every other node. In line 7 we can observe an I/O operation to write/append those distances in a file named `dist_cpp.out`, which can be large (in the order of GB) if large domains are being simulated.

As a first task, before acceleration, a refactor is neeeded in orded to unify the execution, avoiding I/O communication through files. This is mentioned in Section 3.3.1, through the integration of C++ code into Fortran using C wrappers and the Fortran module `iso_c_binding`. In the next two sections, we will describe details about the parallelization of Algorithm 7 using OpenMP, CUDA and a hybrid approach.

6.2.2 OpenMP implementation

The OpenMP implementation of Algorithm 7 is based on a straight-forward split of the landmark point loop in line 5 across T threads involved in the parallel execution. Algorithm 16 depicts the modified steps in order to use OpenMP. The split was implemented inside a parallel region where each thread defines lower and upper indices to traverse in the loop (lines 5 to 8). In this parallel region, $\mathbf{G}$ and `coordsIsomap` should be shared among all threads, in order to reduce the memory usage in this application. No block cyclic strategies are needed to balance the workload. The reason for this is that each iteration comprises the same amount of work, which is a single run of `dijkstra_shortest_paths` starting in a specific landmark point (line 10). After shortest path calculation, the result should be saved in the shared array `coordsIsomap` (line 11).

Input:
`edges`: two dimensional array with edges defining graph $\mathbf{G}$;
`weights`: one dimensional array with edge weights (distances) defining graph $\mathbf{G}$;
`nodes2cal`: one dimensional array with landmark points indices of Ω_L;
`coordsIsomap`: multi-dimensional array with embedding coordinates from $\mathcal{Z}$;
(N, n): size of domain points N and landmark points n;
T: number of parallel threads;

```
1  D ← zeros(N, n)
2  N^Landmark ← Load from array nodes2cal
3  G ← Load from arrays edges and weights
4   // array coordIsomap and graph G are shared among threads
5  for threadId ∈ {1, ..., T} in parallel do
6      B ← ⌈n/T⌉
7      nmin, nmax ← (threadId − 1) * B, min{threadId * B, n}
8      for j ∈ {nmin ..., nmax} do
9          i ← N^Landmark(j)
10         D_:,i ← run_dijsktra_boost(i, G)
11         Save D_:,i into array coordsIsomap
12     end
13 end
```

Output: Array `coordIsomap` with shortest distances between landmarks and domain points

Algorithm 16: Routine `build_distance_matrix` using OpenMP and refactored code

It is important to notice that only this implementation was included in the results presented in Section 4.3 and 5.3. Additional parallel versions, such as using CUDA or hybrid OpenMP/CUDA were implemented a posteriori as experimental improvements.

6.2.3 CUDA implementation

An alternative implementation is based on CUDA, specifically using the cuGraph[6] library. It belongs to the RAPIDS[7][8] open source software project, that enables several data science and analytics pipelines entirely on GPUs. The implementation proposed doesn't split the landmark loop of line 5 from Algorithm 7. Instead, it uses a specialized routine from the library, named `sssp`, and additional routines from the libraries:

```
#include <cugraph/algorithms.hpp>
#include <cugraph/graph.hpp>
```

Algorithm 17 describes the modified steps in order to use CUDA. Its steps are similar to the OpenMP version of the previous section, with key differences in the memory allocation. The distance matrix $\mathbf{D}$ and graph $\mathbf{G}$ should be defined in the GPU device and also in the CPU host (lines 1, 2 and 4). As mentioned before, no split of the landmark loop is applied, instead of that, a GPU device routine is executed to search

[6] https://github.com/rapidsai/cugraph
[7] https://rapids.ai/
[8] https://www.nvidia.com/en-us/deep-learning-ai/software/rapids/

for shortest distances (line 6). After that, the resuting matrix should be copied back from the GPU device to the CPU host (line 7), and posteriorly the result should be saved in the array `coordsIsomap` (line 8).

Additional changes are needed in the `Makefile` of the project, in order to compile CUDA codes and link with cuGraph library application. In order to identify and commit these changes, an exhasutive revision of cuGraph testing cases using C++ was done.

Input:
`edges`: two dimensional array with edges defining graph $\mathbf{G}$;
`weights`: one dimensional array with edge weights (distances) defining graph $\mathbf{G}$;
`nodes2cal`: one dimensional array with landmark points indices of Ω_L;
`coordsIsomap`: multi-dimensional array with embedding coordinates from $\mathcal{Z}$;
(N, n): size of domain points N and landmark points n;

1 $\mathbf{D}^{cpu} \leftarrow$ `zeros`(N, n)
2 $\mathbf{D}^{gpu} \leftarrow$ `zeros`(N, n) *// memory allocation on GPU device through thrust library*
3 $\mathcal{N}^{Landmark} \leftarrow$ Load from array `nodes2cal`
4 $\mathbf{G}^{gpu} \leftarrow$ Load from arrays `edges` and `weights` *// memory allocation on GPU device through cugraph::Graph object*
5 **for** $i \in \mathcal{N}^{Landmark}$ **do**
6 $\quad \mathbf{D}^{gpu}_{:,i} \leftarrow$ `run_dijsktra_cugraph`$(i, \mathbf{G}^{gpu})$ *// operation executed on GPU device*
7 $\quad$ `cudaMemcpy`$(\mathbf{D}^{cpu}_{:,i}, \mathbf{D}^{gpu}_{:,i})$ *// cudaMemcpy is needed to extract the result from GPU device to CPU host*
8 $\quad$ Save $\mathbf{D}^{cpu}_{:,i}$ into array `coordsIsomap`
9 **end**

Output: Array `coordIsomap` with shortest distances between landmarks and domain points

Algorithm 17: Routine `build_distance_matrix` using CUDA and refactored code

6.2.4 Hybrid OpenMP/CUDA implementation

An experimental feature was developed, which allows the combined execution between the CPU and GPU, through the OpenMP and CUDA implementations from the previous sections. Algorithm 18 depicts the modified steps in order to use hybrid OpenMP/CUDA implementation. The key parameter is λ which represents the percentage of landmark points that should be processed by the GPU device. With this parameter, two parallel threads are spawned, one to execute the CUDA-based method from Algorithm 17 (line 5), and the other to execute OpenMP-based method from Algorithm 16 (line 9). In case of OpenMP, two additional threads are needed to manage the parallel executions (`OMP_NUM_THREADS` should be increased by 2), and also nested active parallel regions should be activated with at least 2 levels (`OMP_MAX_ACTIVE_LEVELS` environment variable greater or equal to 2).

The hybrid approach is designed to explore potential advantages when the CPU can also process a part of the workload. As we will see in the next section, marginal gains are observed using hybrid execution according to the performance tests presented. However,

Input:
`edges`: two dimensional array with edges defining graph **G**;
`weights`: one dimensional array with edge weights (distances) defining graph **G**;
`nodes2cal`: one dimensional array with landmark points indices of Ω_L;
`coordsIsomap`: multi-dimensional array with embedding coordinates from $\mathcal{Z}$;
(N, n): size of domain points N and landmark points n;
λ: percentage of workload to process by the CPU;

```
// array coordIsomap is shared among threads  for threadId ∈ {1,2} in parallel do
    // Each thread will process in parallel GPU and CPU blocks
    if threadId = 1 then
        // GPU block
        Process CUDA-based build_distance_matrix for the first (1 − λ) ∗ n landmark points
    end
    if threadId = 2 then
        // CPU block
        Process OpenMP-based build_distance_matrix for the last λ ∗ n landmark points
    end
end
```

Output: Array `coordIsomap` with shortest distances between landmarks and domain points

Algorithm 18: Routine `build_distance_matrix` using OpenMP/CUDA and refactored code

it is an interesting case that can be exploited in future applications or scenarios, not exclusively for LVA-based geostatistical simulations.

6.2.5 Results

In this section we present execution time measurements of the parallel single source shortest path implementation, using only OpenMP, only CUDA or a hybrid OpenMP/CUDA implementation. We use as baseline the LVA-based code `sgs-lva` using 1000 landmark points, with the shortest path calculation implemented in C++ using the Boost Graph Library. The parallel implementations are described in Sections 6.2.2, 6.2.3 and 6.2.4.

All runs were executed in a single-node virtual machine with Ubuntu 18.04.5 LTS with 32-cores Intel(R) Xeon(R) CPU at frequency 2.30GHz and a main memory of 118 GB RAM. The cloud provider in this case is Google Cloud Platform [Google, 2021], and the virtual machine is a `n1-standard-32` with CPU platform Intel Haswell. The system contains 2× NVIDIA Tesla P100 GPUs, with CUDA version 11.0 and driver version 450.51.06. All Fortran programs were compiled using Intel Fortran `ifort` version 2021.2.0 supporting OpenMP version 5.1, with options -fpp -mkl -qopenmp -O3 -mtune=native -march=native. All C++ programs were compiled with GNU C++ compiler `g++` version 9.4.0 supporting OpenMP version 4.5, with options -fopenmp -mcmodel=medium -Wall -Wextra -pedantic -Ofast -funroll-loops -finline-functions -ftree-vectorize. All CUDA programs were compiled with NVIDIA `nvcc` release 11.0

version V11.0.194, with several options mostly extracted from RAPIDS cuGraph testing `Makefile` files.

Execution time results are showed in Figure 6.11. It presents results of hybrid executions, ranging from λ equal to 0% (CUDA-only) to 16% on the top image, and from λ equal to 84% to 100% (OpenMP-only) on the bottom image. OpenMP tests were executed using 16 and 32 threads. CUDA tests were executed using a single Tesla P100 GPU device.

We can observe that on each test, the CUDA only implementation is faster than the OpenMP-only. Performance improvements are obtained using the hybrid implementation in ranges near $\lambda = 6\%$ using 16 OpenMP threads. However, the additional speedup with respect to the CUDA-only implementation is only 1.032×, which is not conclusive enough to be considered better than the only CUDA execution. Performance obtained by the best hybrid implementeation is 3.40× using 16 OpenMP threads and 2.43× using 32 OpenMP threads.

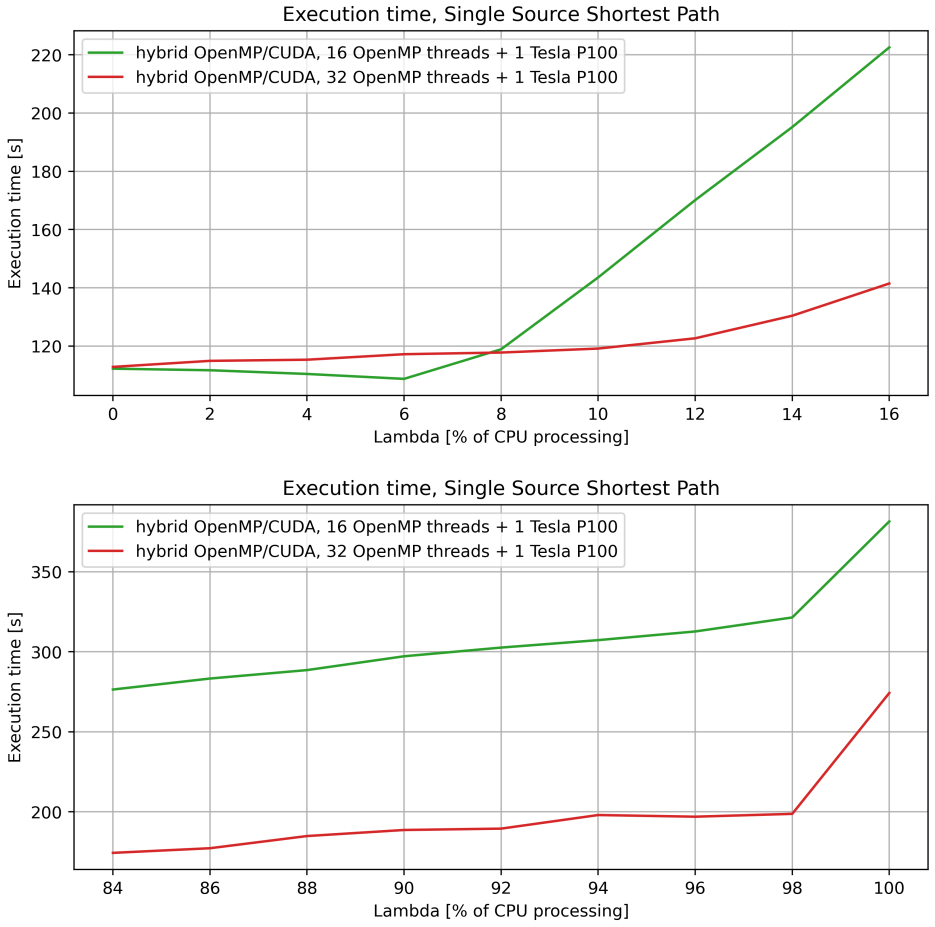

FIGURE 6.11: Execution time using hybrid OpenMP/CUDA execution, from 0% to 16% of CPU usage (top), and from 84% to 100% of CPU usage (bottom).

As mentioned in Table 5.13 from Section 5.3, computing the distance matrix takes the largest portion of the execution time using the fully accelerated versions of LVA-based simulation codes `sgs-lva` and `sisim-lva`. By analyzing the execution profile of Figures 6.12 and 6.13, obtained with `nvprof` and NVIDIA Visual Profiler, the largest part of execution inside the GPU corresponds to the kernel SSSP cuGraph routines `relax_edges` with 41% and `populate_frontier_and_preds` with 33%.

Similarly to the previous section, as an end-to-end performance test, the cuGraph CUDA library is integrated in the code `sgs-lva` in order to measure the overall performance impact in the application. Table 6.3 shows the elapsed time and speedup obtained using different number of threads with the integrated cuGraph code versus the baseline code (which already was optimized and uses the MKL library for algebraic operations, according to Section 6.1).

N Threads	Elapsed time [s] N threads	Speedup Baseline/ N threads	Elapsed time [s] N threads + CUDA	Speedup Baseline/ N threads + CUDA	Speedup N threads/ N threads + CUDA
1	5761	7.821	1763	25.558	3.267
2	3052	14.764	975	46.215	3.130
4	1677	26.869	594	75.858	2.823
5	1404	32.094	521	86.487	2.694
8	984	45.792	414	108.840	2.376
10	854	52.763	385	117.038	2.218
16	685	65.781	349	129.111	1.962
20	629	71.637	343	131.370	1.833

TABLE 6.3: Execution time (seconds) and speedup of `sgs-lva` obtained using cuGraph CUDA-based shortest path calculation. Single-thread baseline execution takes 45060 seconds (12 hours and 31 minutes).

Based on the results presented in this section, the adoption of CUDA in the shortest path calculation for distance matrix building can improve the performance considerably, which opens new ways to accelerate the overall execution.

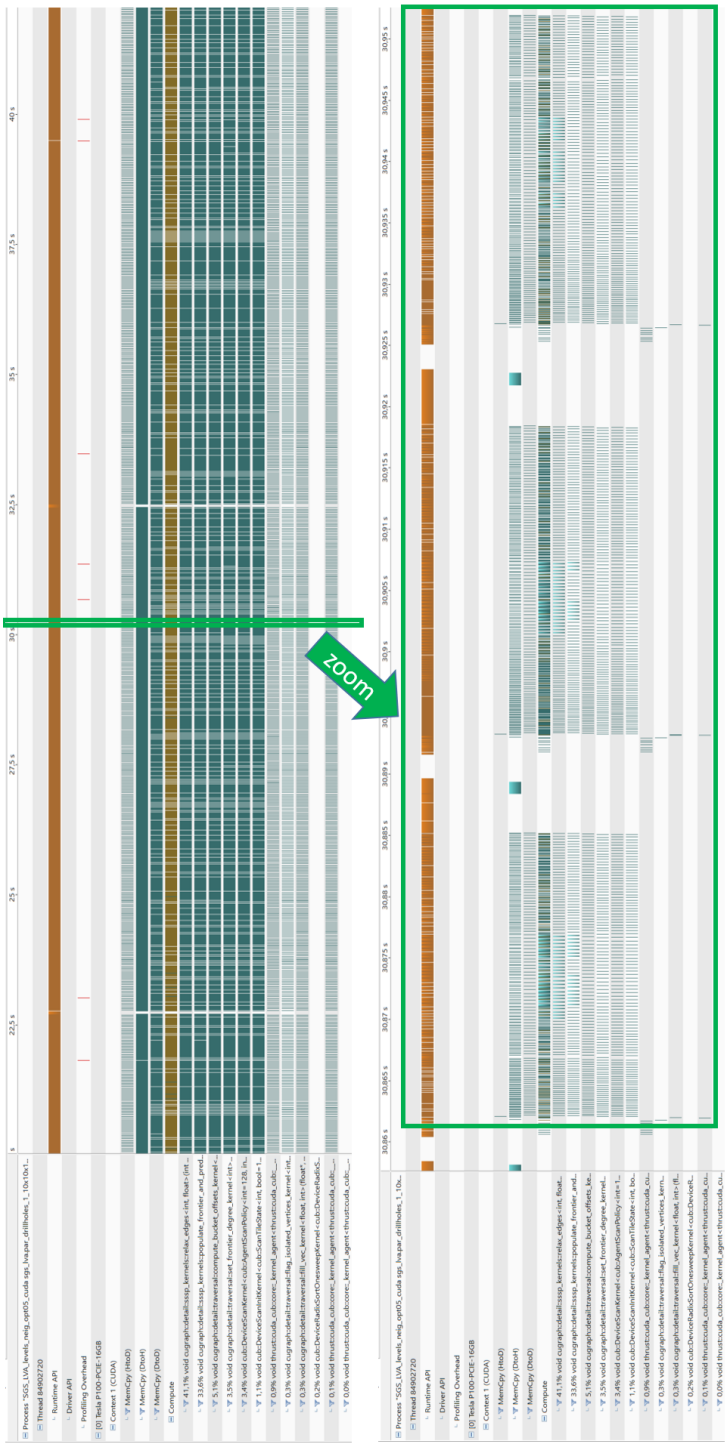

FIGURE 6.12: Profile of execution in the GPU device using NVIDIA Visual Profiler (first zoom).

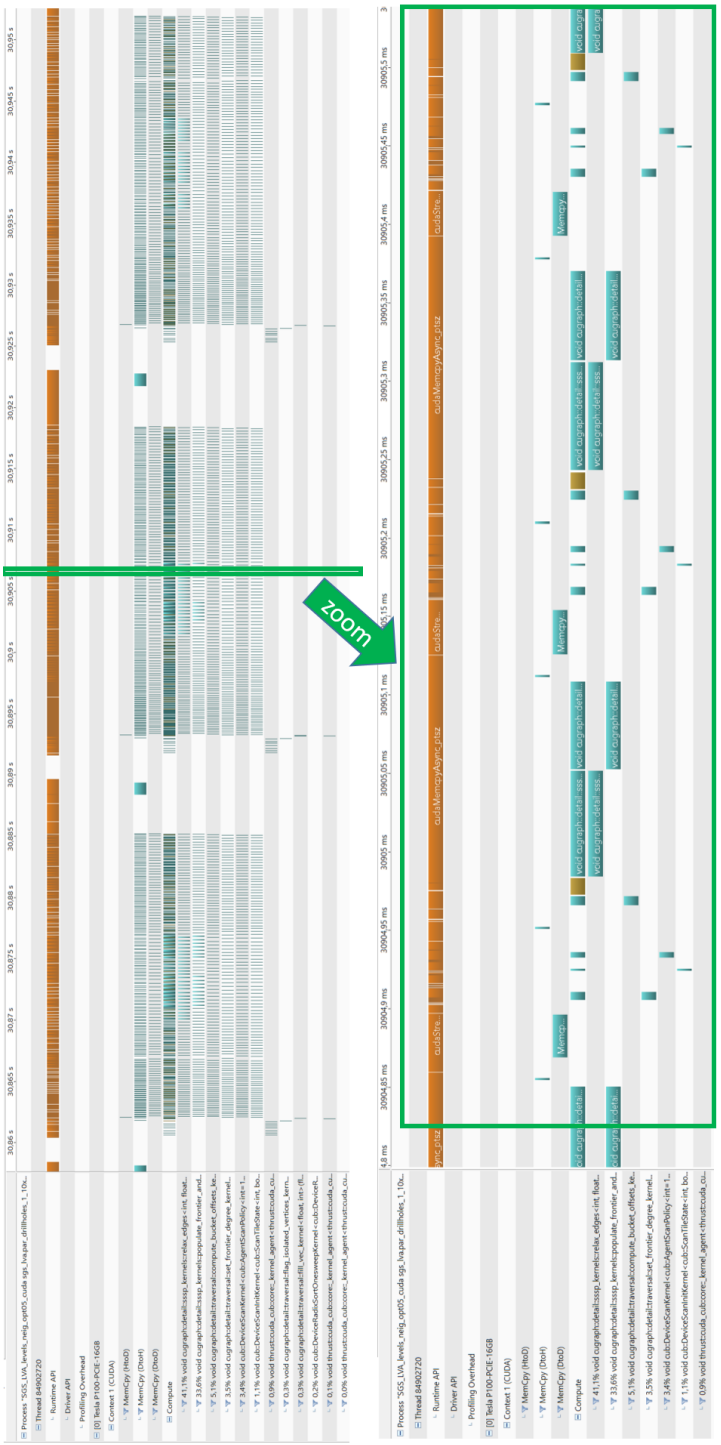

FIGURE 6.13: Profile of execution in the GPU device using NVIDIA Visual Profiler (second zoom).

Chapter 7

Additional parallel applications

A parallel implementation of the semivariogram computation is presented using multi-core processors coupled with a GPU as hardware accelerator. The implementation is based on OpenMP and CUDA and `gamv` application from GSLIB library is used as baseline. Several tests are shown, calculating semivariograms for one and two millions of scattered data points. Scalability and elapsed time results are reported, using double-precision floating point arithmetic on a multi-core CPU coupled with two models of GPU devices: Tesla T4 and Volta V100. Finally, an analytical model is presented to estimate the optimal point used to split the workload between both systems.

7.1 Context

The measurement of spatial variability or continuity of a variable in a geographic region of study is a key step in any geostatistical analysis. Inference of the spatial structural model (semivariogram) has been a constant challenge, since its definition can affect further processes, such as uncertainty quantification of variables in space. In order to minimize the degree of uncertainty involved, the inference process must be as accurate as possible, attaining all spatial variations and patterns given the current sample data.

In the particular case of large scattered data sets, this task can be cumbersome and error-prone, and even prohibitive when large grids of nodes are analyzed. Even though not all industrial or academic case studies will have this particular constraint, there are still some scenarios where the task of spatial structure inference in large grids must be computed as accurate and fast as possible. One of these scenarios arises with the advent of parallel and distributed large-scale applications. In these cases, the geostatistical codes can generate estimations and simulations on large grids in affordable execution time, with almost the same numerical precision as the non-parallel codes. By using parallel two-point statistics, such as sequential simulation methods (see Sections 2.2.1.6

and 2.2.1.7), or parallel multi-point statistics [Mariethoz, 2010, Peredo and Ortiz, 2011, Peredo et al., 2014, Straubhaar et al., 2011, Tahmasebi et al., 2012], many large-scale realizations can be generated. In this scenario, structural validation, cross-validation or comparison between different realizations of large-scale results becomes prohibitive. Another similar scenario is related with the usage of large-scale unstructured grids for estimation and simulation, as mentioned by Manchuk et al. [2005], Zaytsev et al. [2016] and Biver et al. [2017]. If millions of estimated/simulated scattered nodes must be spatially analyzed to test the validity of the predictions, a fast method that allows that task becomes essential. Large unstructured grids are intrinsically more difficult to analyze since they are by definition irregular, which can increase the computational work needed to obtain insights about the spatial structure.

In this chapter, a hybrid parallel implementation of the semivariogram computation is presented, designed to run in CPUs and GPUs simultaneously. A similar approach was presented in Section 6.2.4, in the context of LVA-based codes and shortest path calculation for distance matrix calculation. The current implementation for CPU is based in OpenMP and the implementation for GPU is based on CUDA. With this hybrid parallel approach, large datasets can be processed fast in order to obtain an experimental semivariogram. Additionally, since GPUs are designed to be energy-efficient, lower computing costs can be reached at higher performance rates.

In the next section, a description of the non-parallel version is presented, using as base code the `gamv` routine from GSLIB library[Deutsch and Journel, 1998]. Section 7.2 the parallel versions based on CUDA and OpenMP/CUDA are described, implemented in the same base code. Finally, results are shown in terms of execution time and speedup.

7.1.1 Baseline `gamv` implementation

The `gamv` routine calculates several spatial continuity measures, in an experimental way, using the available dataset as source. Available measures to be calculated are: semivariogram, cross-semivariogram, covariance, correlogram, general relative semivariogram, pairwise relative semivariogram, semivariogram of logarithms, semimadogram and indicator semivariogram. The description of each measure can be found in Deutsch and Journel [1998] (III.1). The semivariogram is one the most used, which is defined assuming stationarity as

$$\gamma(\mathbf{h}) = \frac{1}{2N(\mathbf{h})} \sum_{i=1}^{N(\mathbf{h})} (Z(\mathbf{u}_i) - Z(\mathbf{u}_i + \mathbf{h}))^2 \tag{7.1}$$

where $\mathbf{h}$ is the separation vector, $N(\mathbf{h})$ is the number of pairs separated by $\mathbf{h}$ (with certain tolerance), $Z(\mathbf{u}_i)$ is the value at the start of the vector (tail) defined in the

geographical location $\mathbf{u}_i$ and $Z(\mathbf{u}_i + \mathbf{h})$ is the corresponding value at the end (head) defined in the translated geographical location $\mathbf{u}_i + \mathbf{h}$. In Algorithm 19 we can see a simplified version of the algorithm implemented in `gamv`, using a single variable and direction. We can observe in lines 3 and 4 that a triangular iteration space must be traversed in order to compare all possible pairs of locations $(\mathbf{u}_i, \mathbf{u}_j)$. For each pair, denoted tail and head, the approximate distance between each location is measured and compared with a lag separation $k\mathbf{h}$, as shown in lines 5 and 6. Without loss of generality and for the sake of simplicity, tolerance parameters in the tail and head separation are not included in this algorithm. If the separation between the tail and head is similar to $k\mathbf{h}$, statistics related to spatial measure type τ are computed between $Z(\mathbf{u}_i)$ and $Z(\mathbf{u}_j)$, and stored in the array $\boldsymbol{\beta}$. These operations are encapsulated in the pseudo-routine `save_statistics`, which has different spatial measures implemented, indexed by τ. The steps of the algorithm are essentially the same regardless of the spatial measure to be calculated. For instance, to calculate the semivariogram of equation 7.1, the squared differences $(Z(\mathbf{u}_i) - Z(\mathbf{u}_j))^2$ must be stored in $\boldsymbol{\beta}$. For each separation lag indexed by k, the statistics are accumulated in $\boldsymbol{\beta}(k)$, for instance $\boldsymbol{\beta}(k) = \boldsymbol{\beta}(k) + (Z(\mathbf{u}_i) - Z(\mathbf{u}_j))^2$. Finally, using the statistics stored in $\boldsymbol{\beta}$, the pseudo-routine `build_variogram` computes the final spatial measure values in vector $\boldsymbol{\gamma}$, which is stored in file `output.txt`.

Input:
Separation vector $\mathbf{h}$;
Number of lags $\#lags$;
Spatial measure type τ;
Set of geographical locations $\{\mathbf{u}_1, \ldots, \mathbf{u}_n\}$ with values $Z(\mathbf{u}_i)$;

```
1  Ω ← {u_1, ..., u_n} ;
2  β ← zeros(#lags);
3  for i ∈ {1, ..., |Ω|} do
4      for j ∈ {1, ..., i} do
5          for k ∈ {1, ..., #lags} do
6              if ||(u_j − u_i) − kh|| ≈ 0 then
7                  save_statistics(β, k, Z(u_i), Z(u_j), τ);
8              end
9          end
10     end
11 end
12 γ ← build_variogram(β, τ);
13 write(output.txt,γ);
```

Output: Output file with $\boldsymbol{\gamma}$ values

Algorithm 19: Simplified pseudo-code of `gamv`, measurement of spatial variability/continuity (single-thread algorithm)

7.2 Algorithm

Two implementations are presented, a fully CUDA version and a hybrid OpenMP/CUDA version. A fully OpenMP version was previously published by Peredo et al.

[2015a], so no further analysis was done on this specific version. Nevertheless, the hybrid version is based on this previous implementation, with slight modifications that allow the distribution of workload between the CPU and GPU.

7.2.1 CUDA implementation

The parallel implementation of the semivariogram computation is based on the fact that all statistics between pairs of points computed by the pseudo-routine `save_statistics` of Algorithm 19 (line 7) can be performed independently. By allowing parallel computation of pair statistics, the array $\boldsymbol{\beta}$ can be computed faster. The parallelization pattern in this case is known as domain decomposition or partitioning [Wilkinson and Allen, 2005], and is based on the division of the initial data of the problem and the parallel processing upon the divided data by multiple tasks. The values $\boldsymbol{\beta}(k)$ for each k are computed in parallel by aggregating several results obtained in small sub-domains. For this particular problem, the CUDA framework is specially well-suited, since it is designed to manage massive amounts of threads to perform parallel tasks efficiently in the GPU.

Figure 7.1-top depicts the non-parallel computation between pairs of points (denoted as pairs hence forward) acoording to the triangular iteration space defined by loops in lines 3 and 4 of Algorithm 19. In this case, the indexes are ranging from $i \in \{1, \ldots, N\}$ and $j \in \{1, \ldots, i\}$, with $N = 8$ and a total of 36 pairs. The processing order of all pairs is indexed by a number from 1 to 36 following a row-wise order. The domain, in this case the triangular matrix, say $\mathbf{A}$, is partitioned in three parts, two smaller triangular sub-matrices $\mathbf{A_{11}}$ and $\mathbf{A}_{22}$ and a smaller square matrix $\mathbf{A}_{21}$ with sides $\lfloor N/2 \rfloor$, as shown in Figure 7.1-bottom for $N = 8$. Using this domain decomposition, the CUDA grid of thread blocks is defined to fit inside the sub-matrix $\mathbf{A}_{21}$. In this case, four 4 blocks (blue, pink, green and orange) with 2×2 threads each can be launched. Each thread will perform two or three computations, one in the sub-matrix $\mathbf{A}_{21}$ and the others in some of the triangular sub-matrices. Following the diagram of Figure 7.1-bottom, the first thread of the first block will process the pair with index 11 in $\mathbf{A}_{21}$ (dashed dark blue/pale blue), and the pairs with indexes 1 (dark blue) and 15 (pale blue) in the other sub-matrices. The second thread will process the pair with index 12 (pale blue) and 20 (pale blue). The third thread will process the pair with index 16 (dark blue) and 2 (dark blue), and so on.

In Figure 7.2 a diagram of the computation flow for the first thread block (blue) is depicted. Each thread, represented by a different colored arrow, traverses through the corresponding pairs and computes statistics of two pairs (red and gray arrows) or three pairs (yellow and white arrows). The same flows are depicted in Figure 7.3 for the second (pink) and third thread blocks (green). In these cases, only two computations

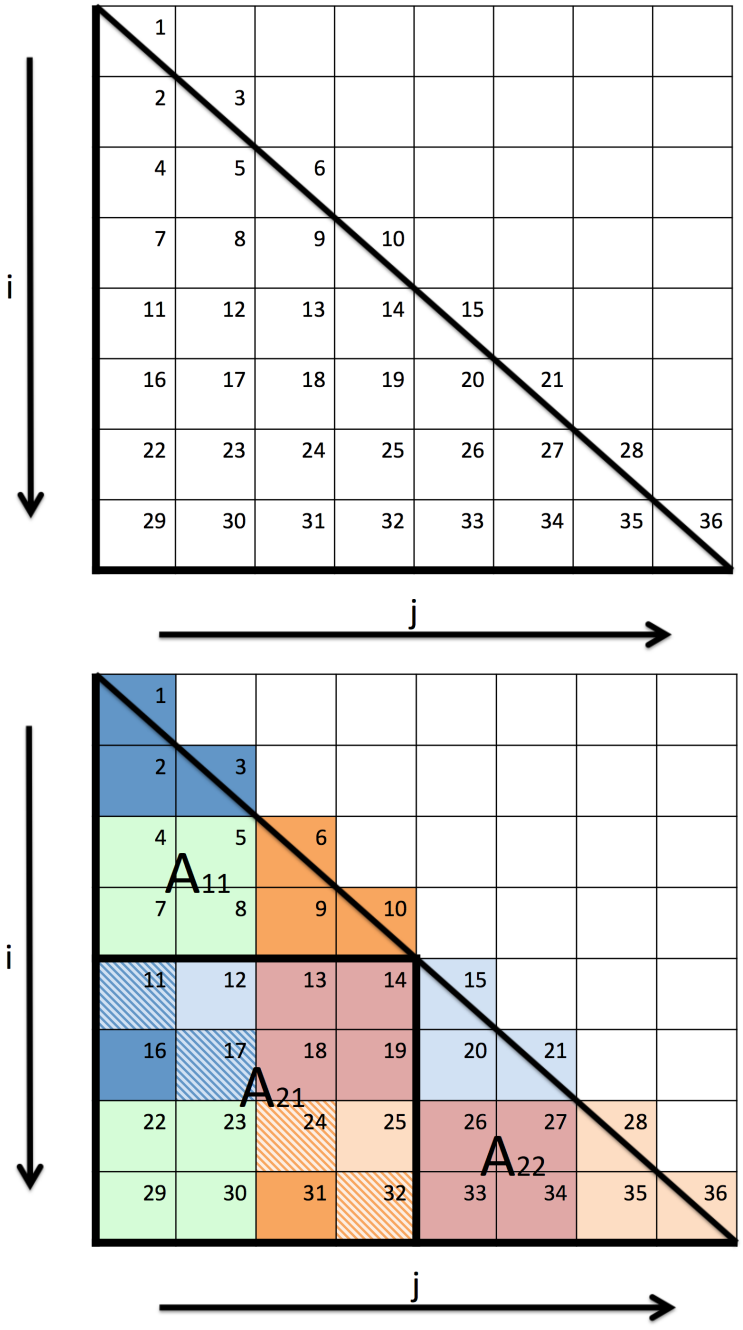

FIGURE 7.1: Top: Non-parallel computation through pairs of values. Bottom: Domain decomposition using four thread blocks with 2×2 threads each ($\mathbf{A}_{11}$, $\mathbf{A}_{21}$ and $\mathbf{A}_{22}$). The colors of the thread blocks are blue, pink, green and orange, with pale and dark colors to differentiate computations performed in the upper sub-matrix or lower sub-matrix.

are performed by each thread, allowing to calculate statistics of two pairs of locations. The total number of threads to be launched must be equal to the number of elements of $\mathbf{A}_{21}$, i.e. equal to the size of the smaller square matrix defined by the domain partitioning. Since each GPU device has specific parameters regarding the maximum number of threads per block and maximum number of blocks to be launched, the maximum

scenario to be handled by the proposed parallelization will depend on those parameters and the available hardware to be used.

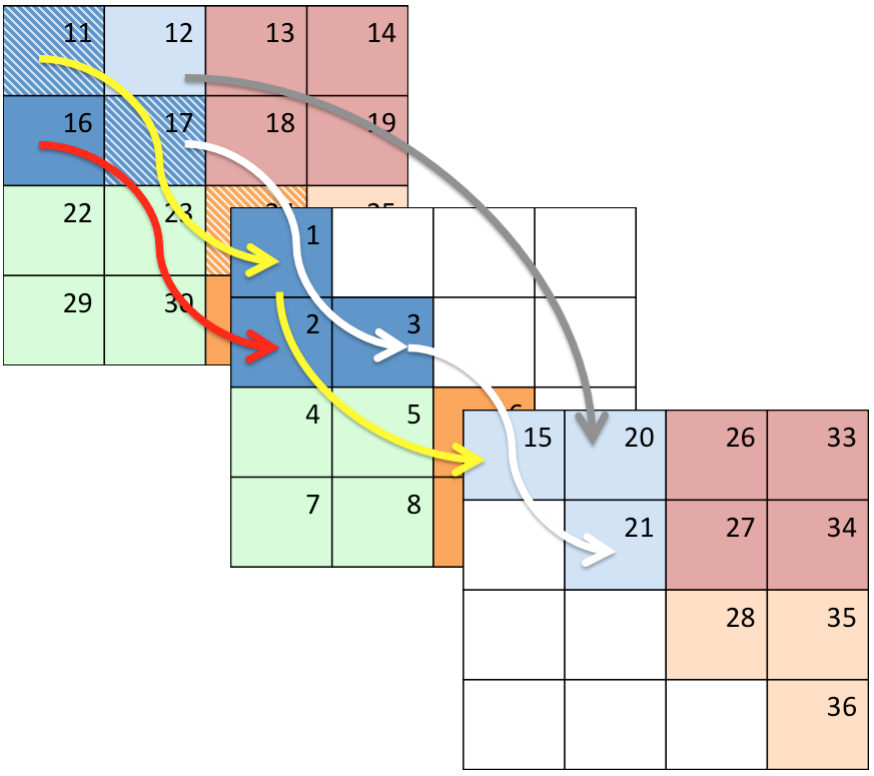

FIGURE 7.2: Parallel computations in the first thread block (blue) using 2×2 threads per block. $\mathbf{A}_{22}$ is depicted transposed to facilite the reading.

The steps of the CUDA based parallelization are described in Algorithm 20. The first step is the definition of the grid of blocks and the number of threads per block that must fit in $\mathbf{A}_{21}$, $\lfloor N/2 \rfloor \times \lfloor N/2 \rfloor$ threads with $N = |\Omega|$ (lines 3 and 4). After that, all threads belonging to all blocks in the grid are executed in parallel, and each thread is responsible of computing statistics on two or three pairs of locations. The local indexes in $\mathbf{A}_{21}$, denoted as (i^*, j^*), are computed according to the thread and block indexes previously defined (pseudo routine `compute_indexes_small_square` in line 6). For completeness, the global indexes in the large square matrix with dimensions $N \times N$ are denoted as (i, j). In the example of Figure 7.1-bottom, the local index $(i^*, j^*) = (0, 0)$ is assigned to the pair 11 with global index $(4, 0)$; the local index $(0, 1)$ is assigned to the pair 12 with global index $(4, 1)$; and so on. In sub-matrix $\mathbf{A}_{21}$, pairs are processed using the global indexes $(i, j) = (i^* + |\Omega|/2, j^*)$, since the row indexes of $\mathbf{A}_{21}$ start in $|\Omega|/2$ (lines 7 to 12). Pairs in $\mathbf{A}_{11}$ are processed using the global indexes $(i, j) = (i^*, j^*)$, since both indexes start at 0 (lines 13 to 16). Next, pairs in $\mathbf{A}_{22}$ are processed using the global indexes $(i, j) = (i^* + |\Omega|/2, j^* + |\Omega|/2)$, since both indexes start at $|\Omega|/2$ (lines 17 to 24). The diagonal values in $\mathbf{A}_{11}$ are processed using the global indexes $(i, j) = (i^*, i^*)$ (lines 18 to 21). This last step is complemented with the processed pair from the lower

triangular sub-matrix, because if $i^* = j^*$ then the global indexes $(i^* + |\Omega|/2, i^* + |\Omega|/2)$ and (i^*, i^*) are processed, as shown for the pairs 11 and 17 from Figure 7.1-bottom. For each block of threads, all statistics computed by those threads are accumulated in shared arrays stored in each block's shared memory, which are then aggregated in a device global memory array copied into the host array $\boldsymbol{\beta}$.

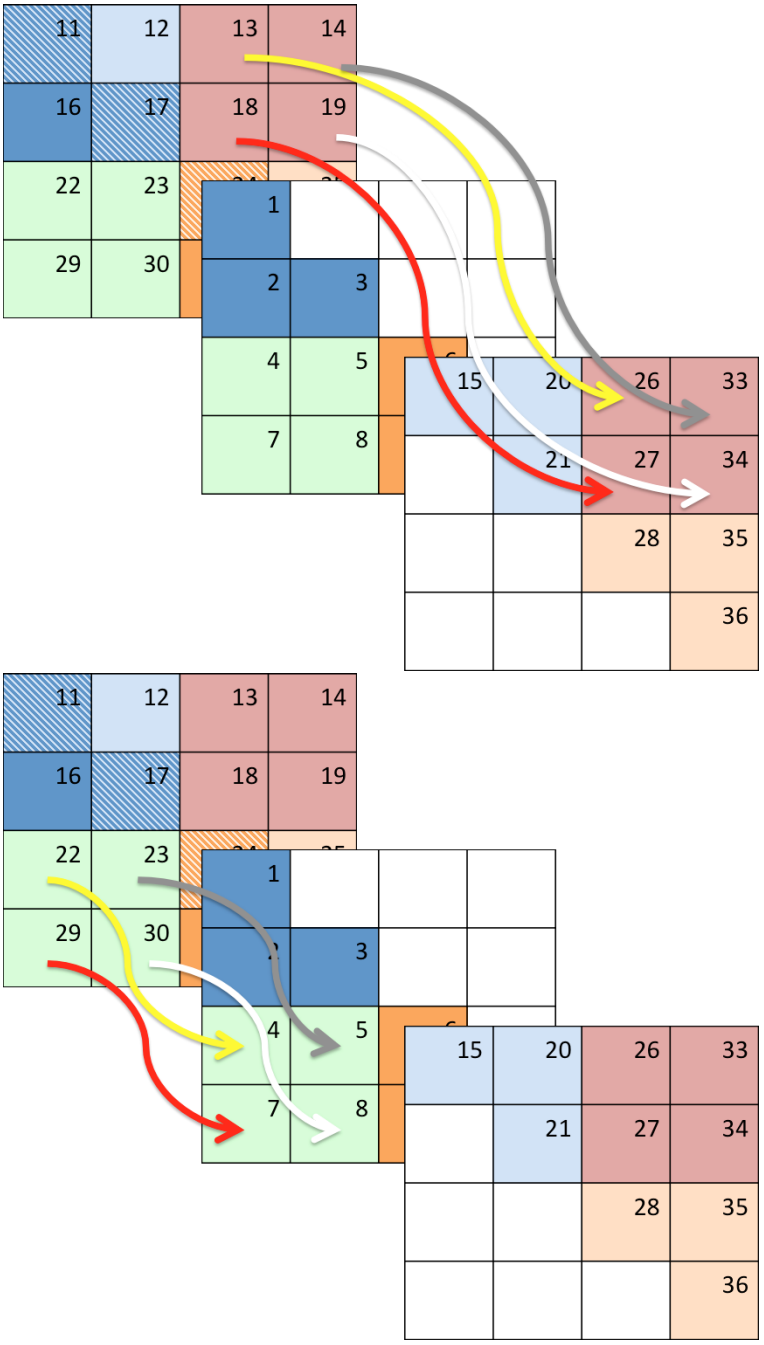

FIGURE 7.3: Parallel computations in the second (pink color, left) and third thread block (green color, right) using 2×2 threads per block. $\mathbf{A}_{22}$ is depicted transposed to facilite the reading.

Input:
Separation vector $\mathbf{h}$;
Number of lags $\#lags$;
Spatial measure type τ;
Set of geographical locations $\{\mathbf{u}_1, \ldots, \mathbf{u}_n\}$ with values $Z(\mathbf{u}_i)$;
(nthreads.x, nthreads.y) dimensions of each thread block;
$\left(\left\lfloor \frac{|\Omega|/2}{\text{nthreads.x}} \right\rfloor, \left\lfloor \frac{|\Omega|/2}{\text{nthreads.y}} \right\rfloor\right)$ dimensions of thread block grid;

1 $\Omega \leftarrow \{\mathbf{u}_1, \ldots, \mathbf{u}_{|\Omega|}\}$;
2 $\boldsymbol{\beta} \leftarrow$ `zeros`$(\#lags)$;
3 $T \leftarrow$ nthreads.x $\times$ nthreads.y;
4 $B \leftarrow \left\lfloor \frac{|\Omega|/2}{\text{nthreads.x}} \right\rfloor \times \left\lfloor \frac{|\Omega|/2}{\text{nthreads.y}} \right\rfloor$;
5 **for** *all threads* $t \in \{1, \ldots, T\}$ *in all blocks* $b \in \{1, \ldots, B\}$ **do**
*/*Initialize local indexes in* $\mathtt{A}_{12}$**/*
6 $(i^*, j^*) \leftarrow$ `compute_indexes_small_square`(t, b) ;
*/*Compute global indexes from local indexes*/*
7 $(i, j) \leftarrow (i^* + |\Omega|/2, j^*)$;
8 **for** $k \in \{1, \ldots, \#lags\}$ **do**
9 **if** $||(\mathbf{u}_j - \mathbf{u}_i) - k\mathbf{h}|| \approx 0$ **then**
10 `save_statistics`$(\boldsymbol{\beta}, k, \mathbf{Z}(\mathbf{u}_i), \mathbf{Z}(\mathbf{u}_j), \tau)$;
11 **end**
12 **end**
13 **if** $i^* > j^*$ **then**
14 $(i, j) \leftarrow (i^*, j^*)$;
15 Repeat lines 8 to 12 with the updated indexes (i, j);
16 **end**
17 **else if** $i^* \leq j^*$ **then**
18 **if** $i^* == j^*$ **then**
19 $(i, j) \leftarrow (i^*, i^*)$;
20 Repeat lines 8 to 12 with the updated indexes (i, j);
21 **end**
22 $(i, j) \leftarrow (j^* + |\Omega|/2, i^* + |\Omega|/2)$;
23 Repeat lines 8 to 12 with the updated indexes (i, j);
24 **end**
25 **end**
26 $\boldsymbol{\gamma} \leftarrow$ `build_variogram`$(\boldsymbol{\beta}, \tau)$;
27 `write`(`output.txt`,$\boldsymbol{\gamma}$);

Output: Output file with $\boldsymbol{\gamma}$ values

Algorithm 20: Simplified pseudo-code of `gamv-cuda`, measurement of spatial variability/continuity (multi-thread CUDA algorithm)

7.2.2 Hybrid OpenMP/CUDA implementation

The hybrid parallelization is based on the CUDA parallel algorithm described in the previous section, coupled with OpenMP multi-thread processing, as depicted in Algorithm 22. The strategy in this case is to split the triangular matrix from Figure 7.1-left into two sub-domains, as depicted in Figure 7.4-left. The parameter $\lambda \in [0, 1]$ controls the amount of work to be performed in the CPU, leaving the remaining $1 - \lambda$ portion to be performed in the GPU. Both workloads are executed concurrently using CUDA Streams [NVIDIA Corporation, 2017], which allows to run two overlapping compute kernels in the CPU and GPU. The CUDA parallelization strategy is the same as described in the previous section, but applied to a smaller triangular sub-matrix (lines 4 and 5 of

Algorithm 22). The only difference is the grid of thread blocks provided to the CUDA kernels. In this case, the grid of thread blocks has size $\left\lfloor \frac{((1-\lambda)|\Omega|)/2|}{\text{nthreads.x}} \right\rfloor \times \left\lfloor \frac{((1-\lambda)|\Omega|)/2}{\text{nthreads.y}} \right\rfloor$, and the thread indexes must be translated by $\lambda|\Omega|$ in the X and Y dimensions. The OpenMP parallelization strategy consists in a parallel loop traversing the index j from 1 to $\lambda|\Omega|$, where each thread processes a chunk of rows, each one ranging from j to $|\Omega|$ (Algorithm 21, called from line 6 of Algorithm 22). Each parallel part stores their results in separate arrays, $\boldsymbol{\beta}_{OMP}$ and $\boldsymbol{\beta}_{CUDA}$ respectively, which are added and processed to build the spatial measure $\boldsymbol{\gamma}$ (line 8 of Algorithm 22). In Figure 7.4-right we can observe different distributions of the execution time for each workload, using different values of λ. The objective of the hybrid parallel strategy is to exploit all resources of the machine, not only the GPU, resulting in a faster execution where the processing power of the CPU can handle some of the workload otherwise delivered to the GPU. For instance, example (B) of Figure 7.4-right shows a scenario where the overall execution time is reduced by delegating 15% of the workload to the CPU. The optimal value for λ must be estimated according to the hardware features of each system, and also after empirical validation testing different values of λ.

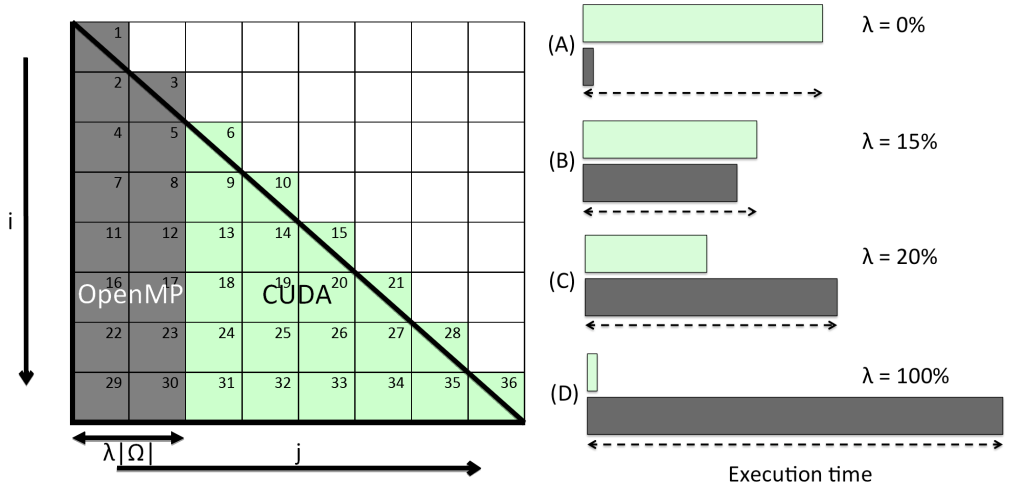

FIGURE 7.4: Left: Domain decomposition in the hybrid parallel algorithm, using OpenMP (gray sub-domain) and CUDA (green sub-domain). Right: Different timing distributions varying λ, using the hybrid parallel strategy with OpenMP (gray) and CUDA (green). (A) Using only the GPU with $\lambda = 0\%$. (B) Execution acceleration using $\lambda = 15\%$. (C) CPU dominance against the GPU using $\lambda = 20\%$. (D) Using only the CPU with $\lambda = 100\%$.

Input:
Separation vector $\mathbf{h}$;
Number of lags $\#lags$;
Spatial measure type τ;
Set of geographical locations $\{\mathbf{u}_1, \ldots, \mathbf{u}_n\}$ with values $Z(\mathbf{u}_i)$;
Portion of work to be performed in CPU λ;

```
1  Ω ← {u_1, ..., u_|Ω|} ;
2  β_OMP ← zeros(#lags);
3  for j = 1 to λ|Ω| in parallel (OpenMP) do
4      for i = j to |Ω| do
5          for k ∈ {1, ..., #lags} do
6              if ||(u_j − u_i) − kh|| ≈ 0 then
7                  save_statistics(β_OMP, k, Z(u_i), Z(u_j), τ);
8              end
9          end
10     end
11 end
```

Output: Vector $\boldsymbol{\beta}_{OMP}$ with computed statistics

Algorithm 21: Simplified pseudo-code of OpenMP part from `gamv-cuda-omp`

Input:
Separation vector $\mathbf{h}$;
Number of lags $\#lags$;
Spatial measure type τ;
Set of geographical locations $\{\mathbf{u}_1, \ldots, \mathbf{u}_n\}$ with values $Z(\mathbf{u}_i)$;
Portion of work to be performed in CPU λ

1 $\Omega \leftarrow \{\mathbf{u}_1, \ldots, \mathbf{u}_{|\Omega|}\}$;
2 $\boldsymbol{\beta}_{OMP} \leftarrow$ `zeros`$(\#lags)$;
3 $\boldsymbol{\beta}_{CUDA} \leftarrow$ `zeros`$(\#lags)$;
//Work offloaded to the GPU using CUDA Streams to overlap CPU work
4 Modify Algorithm 2 by replacing $|\Omega|/2$ for $((1-\lambda)|\Omega|)/2$ (shrinking), adding $\lambda|\Omega|$ to each component of (i, j) (translation) and returning the statistics vector $\boldsymbol{\beta}$ instead of writing the variogram results;
5 $\boldsymbol{\beta}_{CUDA} \leftarrow$ Compute previously modified Algorithm 2 with parameters $\mathbf{h}$, $\#lags$, τ, $\{(\mathbf{u}_i, Z(\mathbf{u}_i) : i = 1, \ldots, n\}$, (nthreads.x, nthreads.y) and $\left(\left\lfloor \frac{((1-\lambda)|\Omega|)/2}{\text{nthreads.x}} \right\rfloor, \left\lfloor \frac{((1-\lambda)|\Omega|)/2}{\text{nthreads.y}} \right\rfloor\right)$;
//Work processed in parallel with OpenMP in the CPU
6 $\boldsymbol{\beta}_{OMP} \leftarrow$ Compute Algorithm 3 with parameters $\mathbf{h}$, $\#lags$, τ, $\{(\mathbf{u}_i, Z(\mathbf{u}_i) : i = 1, \ldots, n\}$ and λ;
7 Synchronize the offloaded CUDA Stream and the CPU processing;
8 $\boldsymbol{\gamma} \leftarrow$ `build_variogram`$(\boldsymbol{\beta}_{OMP} + \boldsymbol{\beta}_{CUDA}, \tau)$;
9 `write`(`output.txt`,$\boldsymbol{\gamma}$);

Output: Output file with $\boldsymbol{\gamma}$ values

Algorithm 22: Simplified pseudo-code of `gamv-cuda-omp`, measurement of spatial variability/continuity (multi-thread CUDA/OpenMP algorithm)

7.3 Results

Two methods are presented to show the results of the proposed hybrid parallelization. The first one is based on a set of experiments using a pre-defined search space for λ in each case study. This can give us an initial estimation of the speedup values that can be obtained with the application. The second method is based on an analytical expression

to infer the optimal value for λ, with an additional heuristic to search the optimal value in practical scenarios.

The case studies presented in this section use scattered data extracted at random from large-scale simulations with approximately 100 million grid nodes ($800 \times 800 \times 160$ and $420 \times 600 \times 400$), generated using parallel versions of the classical simulation codes `sgsim` and `sisim`, developed by Deutsch and Journel [1998] and parallelized as described in Chapter 4. Two random scattered data sets were extracted from each grid (1M and 2M locations, with M=10^6). The parameters used in each data set are depicted in Table 7.1, and some samples of obtained semivariograms are depicted in Figure 7.5. For each data set, three execution modes were compared: Fortran-based single-thread classical implementation of `gamv` as described in Deutsch and Journel [1998]; CUDA-based multi-thread parallelization; and hybrid CUDA/OpenMP parallel implementation. Additionaly, all executions are compared against a benchmark implementation using optimized Fortran code and multi-thread OpenMP parallelization as described in Peredo et al. [2015a].

Data sets	`sgsim`	`sisim`
variogram type	exponential	indicator categorical (10 categories)
# lags	40	20
lag separation	20	5
lag tolerance	5	1
azimuth, dip	(0,0)	(0,0)
tolerance	(90,90)	(90,90)
bandwidth	(20,20)	(20,20)

TABLE 7.1: `gamv` parameters for different scattered data sets. Description of each parameter can be reviewed in Deutsch and Journel [1998], Section III.1.

All runs were executed on two cloud virtual instances. The first one, `g4dn.8xlarge`, using double-precision floating point arithmetic in a single-node machine with Ubuntu 18.04.5 LTS, with a 32-core Intel(R) Xeon(R) CPU E5-2690 v3 at frequency 2.60GHz, and a memory hierarchy of 112GB RAM, 30MB L3 cache, 256KB L2 cache and 32KB/32KB L1d/L1i cache. A single Tesla T4 GPU is attached to the system, with 40 streaming multiprocessors (SM) and CUDA driver version 11.0. The second one, `g5.8xlarge`, using double-precision floating point arithmetic in a single-node machine with Ubuntu 18.04.5 LTS, with a 32-core Intel(R) Xeon(R) CPU E5-2686 v4 at frequency 2.30GHz, and a memory hierarchy of 240GB RAM, 45MB L3 cache, 256KB L2 cache and 32KB/32KB L1d/L1i cache. A single Volta V100 SXM2 GPU is attached to the system, with 80 streaming multiprocessors and CUDA driver version 11.0. Details of these machines can be found in technical documents from the cloud computing provider Amazon Web Services [AWS, 2021].

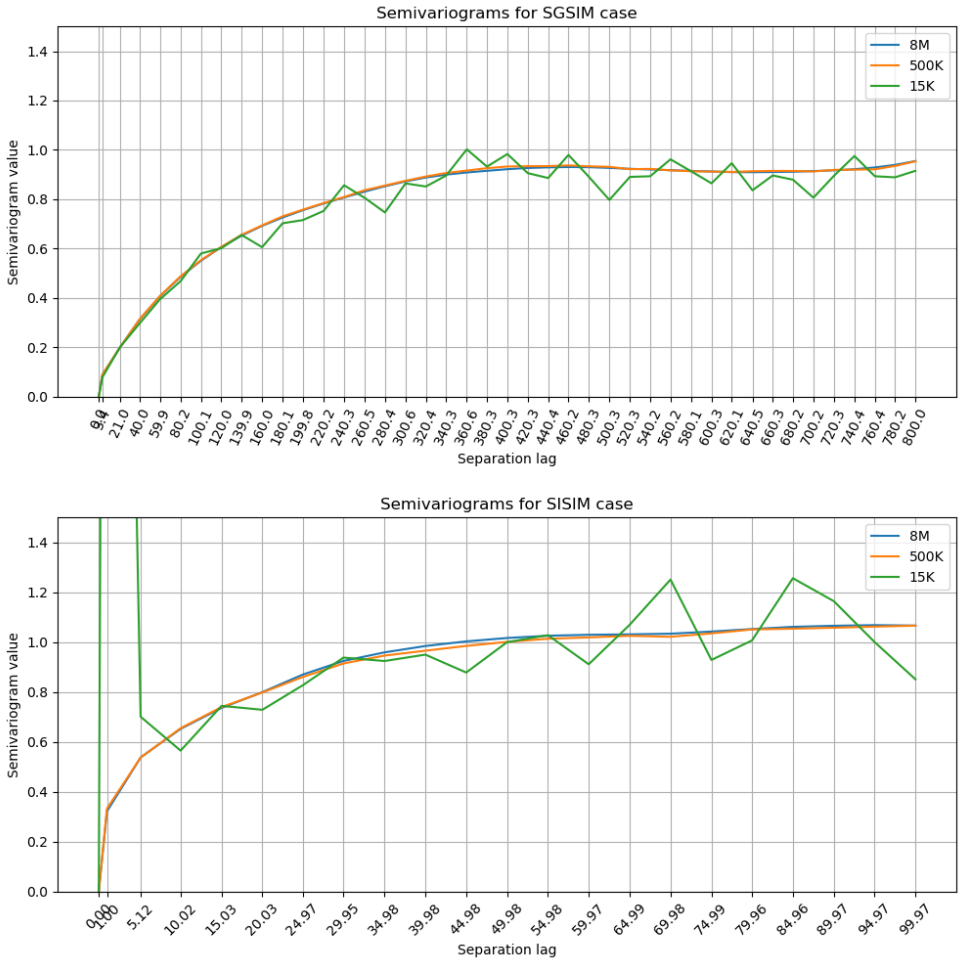

FIGURE 7.5: Sample semivariograms obtained using `sgsim` (top) and `sisim` (bottom) scattered data sets.

All programs were compiled using GCC `gfortran` version 7.5.0 supporting OpenMP version 4.5, with the options `-O3 -cpp` in all cases and `-fopenmp` in the OpenMP multi-thread executions. The CUDA programs were compiled using NVIDIA `nvcc` release 11.0, version V11.0.221, with the options `-m64 -c -O3 -arch=sm_35`.

7.3.1 Experimental results

Initially, experimental values for λ need to be calculated. In order to do that, execution time results are depicted in Figures 7.6, 7.7, 7.8 and 7.9. On each figure, we can observe the execution time using different values of λ, ranging from $\lambda = 0\%$ to 12.5%, with different steps between 0% to 0.1%, 0.1% to 1.6%, 1.6% to 3.2%, 4% to 6%, and 6.25% to 12.25%. For each scattered data set, the same experimental optimal values, denoted λ^*_{exp}, are obtained consistently, independently of the size of the scattered dataset. In `sgsim` scenario and using the Tesla T4 GPU, using 4 OpenMP threads the experimental minimum estimate is $\lambda^*_{exp} = 2.0\%$, using 8 threads the value is $\lambda^*_{exp} = 4.5\%$, and using

16 threads the value is $\lambda^*_{exp} = 8.25\%$. Results for `sisim` scenario using the Tesla T4 GPU, using 4, 8 and 16 threads are 2.4%, 5.0% and 9.25% respectively. Results for `sgsim` using the Volta V100 and 4, 8 and 16 threads are 0.8%, 1.6%, 3.2% respectively, and results for `sisim` using the Volta V100 and 4, 8 and 16 threads are 0.8%, 2.0% and 4.0% respectively.

The results of the executions in single-thread, CUDA and hybrid mode (best scenario) can be viewed in Tables 7.2 and 7.3. OpenMP benchmark results, using optimized code from Peredo et al. [2018], can be viewed in Table 7.4. These codes have been compiled and executed in the instance `g4dn.8xlarge`. Single-thread GSLIB sequential code was compiled and executed in the instance `g4dn.8xlarge` (2.60GHz), and the same execution time was used to benchmark executions in the instance `g5.8xlarge` (2.30GHz). It is expected to adjust the experiments according to the execution time in the instance `g5.8xlarge`, to be included in the final version of the document, but in any case, speedup results should be higher according to the CPU frequency difference of both instances.

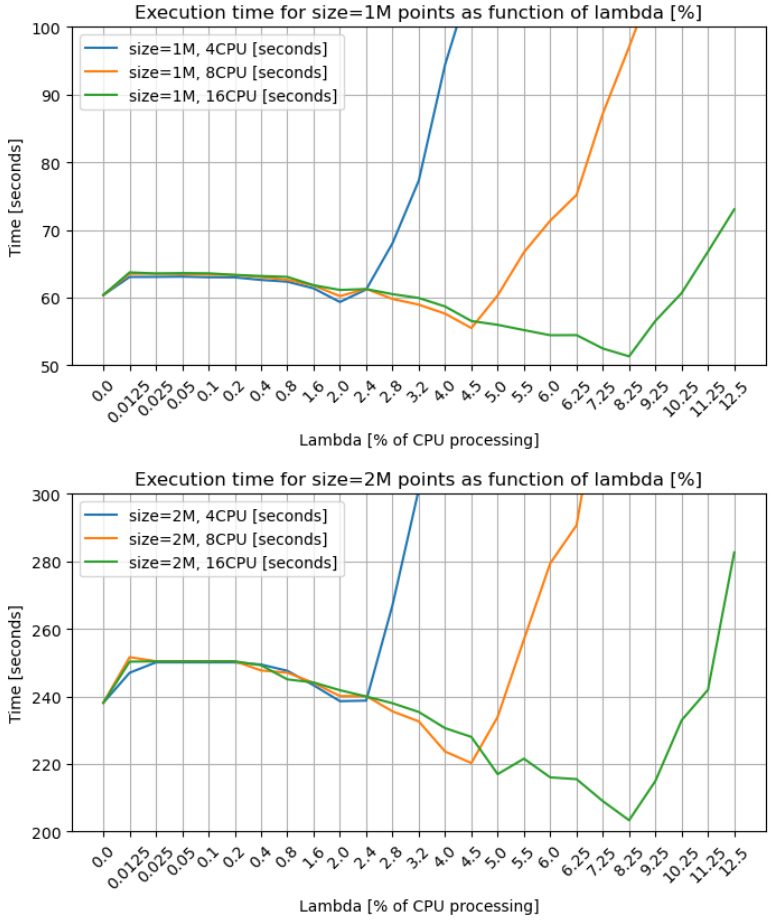

FIGURE 7.6: Execution time using different values for λ, with 4, 8 and 16 threads in `sgsim` scattered data set, using a GPU Tesla T4. Case studies with 1M (top) and 2M (bottom) points.

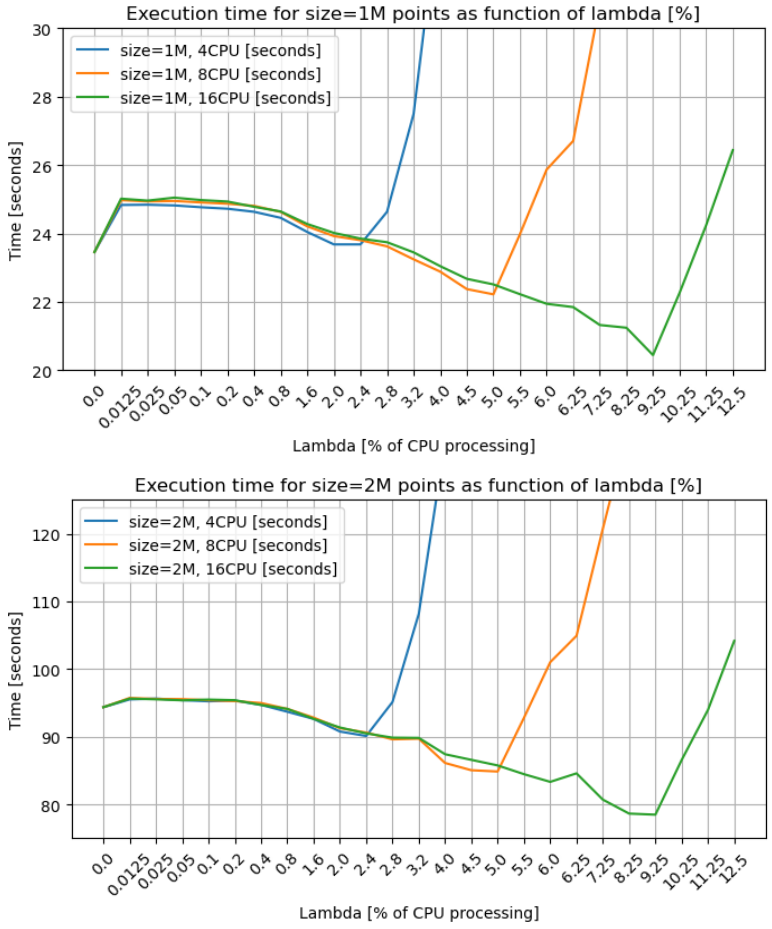

FIGURE 7.7: Execution time using different values for λ, with 4, 8 and 16 threads in `sisim` scattered data set, using a GPU Tesla T4. Case studies with 1M (top) and 2M (bottom) points.

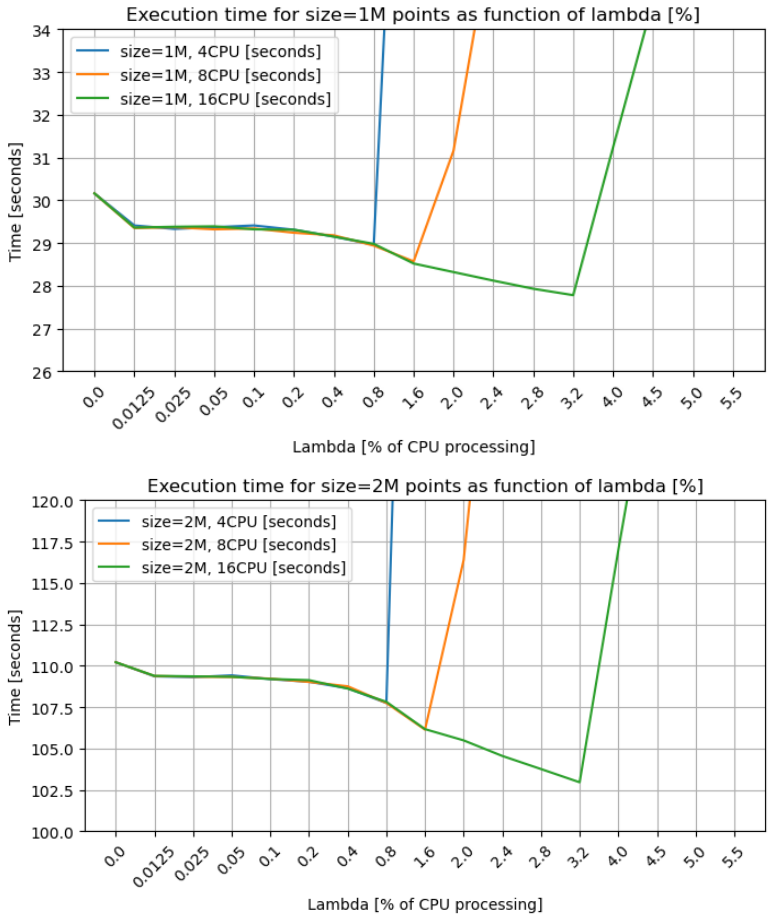

FIGURE 7.8: Execution time using different values for λ, with 4, 8 and 16 threads in `sgsim` scattered data set, using a GPU Volta V100. Case studies with 1M (top) and 2M (bottom) points.

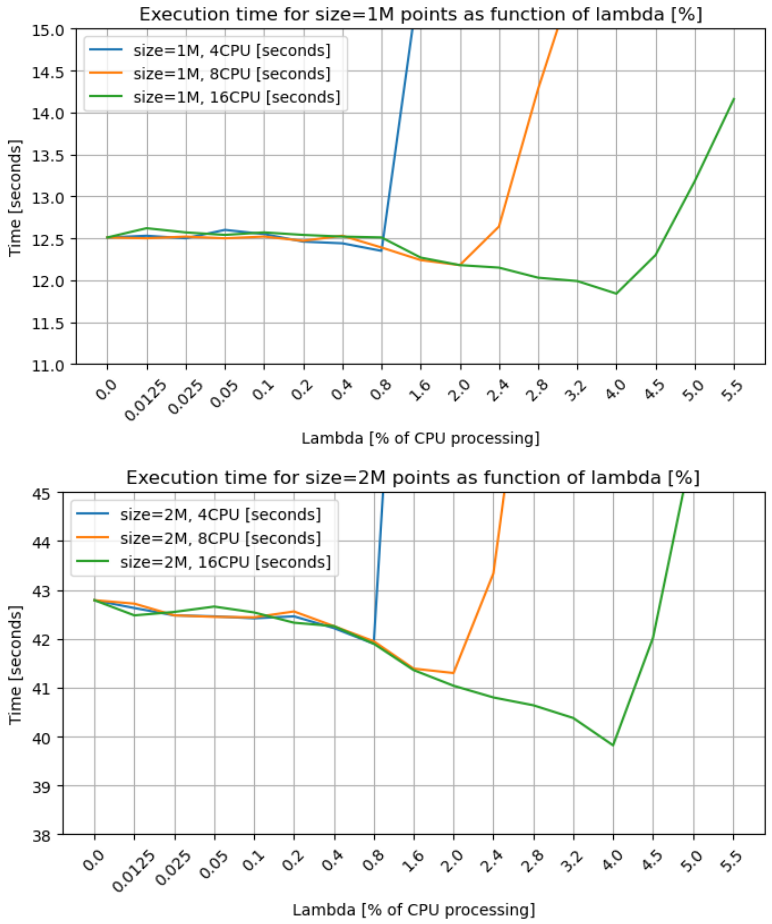

FIGURE 7.9: Execution time using different values for λ, with 4, 8 and 16 threads in `sisim` scattered data set, using a GPU Volta V100. Case studies with 1M (top) and 2M (bottom) points.

Case	# points	N Threads	GSLIB (Single thread)	CUDA (Tesla T4)	Hybrid (Tesla T4 + N threads)	Speedup CUDA/ Hybrid	Speedup GSLIB/ CUDA	Speedup GSLIB/ Hybrid
`sgsim`	1,000,000	4	26050	60.12	59.35	1.01	433.301	438.920
`sgsim`	1,000,000	8	26050	60.12	55.5	1.08	433.301	469.369
`sgsim`	1,000,000	16	26050	60.12	51.32	1.17	433.301	507.599
`sgsim`	2,000,000	4	103783	238.11	238.59	0.99	435.861	434.984
`sgsim`	2,000,000	8	103783	238.11	220.26	1.08	435.861	471.184
`sgsim`	2,000,000	16	103783	238.11	203.32	1.17	435.861	510.441
`sisim`	1,000,000	4	2938	23.456	23.680	0.99	125.255	124.070
`sisim`	1,000,000	8	2938	23.456	22.220	1.05	125.255	132.223
`sisim`	1,000,000	16	2938	23.456	20.443	1.14	125.255	143.716
`sisim`	2,000,000	4	11754	94.363	90.113	1.04	124.561	130.436
`sisim`	2,000,000	8	11754	94.363	84.863	1.11	124.561	138.505
`sisim`	2,000,000	16	11754	94.363	78.486	1.20	124.561	149.759

TABLE 7.2: Execution time (seconds) and speedup obtained using a GPU Tesla T4. Experimental optimal values for `sgsim` case: $\lambda^*_{exp} = 2.0\%$ (4 threads), $\lambda^*_{exp} = 4.5\%$ (8 threads), and $\lambda^*_{exp} = 8.25\%$ (16 threads). Experimental optimal values for `sisim` case: $\lambda^*_{exp} = 2.4\%$ (4 threads), $\lambda^*_{exp} = 5\%$ (8 threads), and $\lambda^*_{exp} = 9.25\%$ (16 threads).

Case	# points	N Threads	GSLIB (Single thread)	CUDA (Volta V100)	Hybrid (Volta V100 + N threads)	Speedup CUDA/ Hybrid	Speedup GSLIB/ CUDA	Speedup GSLIB/ Hybrid
`sgsim`	1,000,000	4	26050	30.16	28.96	1,04	863.726	899.516
`sgsim`	1,000,000	8	26050	30.16	28.57	1,05	863.726	911.795
`sgsim`	1,000,000	16	26050	30.16	27.78	1,08	863.726	937.724
`sgsim`	2,000,000	4	103783	110.21	107.75	1,02	941.684	963.183
`sgsim`	2,000,000	8	103783	110.21	106.14	1,03	941.684	977.793
`sgsim`	2,000,000	16	103783	110.21	102.96	1,07	941.684	1007.993
`sisim`	1,000,000	4	2938	12.51	12.35	1.01	234.852	237.894
`sisim`	1,000,000	8	2938	12.51	12.18	1.02	234.852	241.215
`sisim`	1,000,000	16	2938	12.51	12.18	1.02	234.852	241.215
`sisim`	2,000,000	4	11754	42.79	41.90	1.02	274.690	280.525
`sisim`	2,000,000	8	11754	42.79	41.30	1.03	274.690	284.600
`sisim`	2,000,000	16	11754	42.79	39.82	1.07	274.690	295.178

TABLE 7.3: Execution time (seconds) and speedup obtained using a GPU Volta V100. Experimental optimal values for `sgsim` case: $\lambda^*_{exp} = 0.8\%$ (4 threads), $\lambda^*_{exp} = 1.6\%$ (8 threads), and $\lambda^*_{exp} = 3.2\%$ (16 threads). Experimental optimal values for `sisim` case: $\lambda^*_{exp} = 0.8\%$ (4 threads), $\lambda^*_{exp} = 2\%$ (8 threads), and $\lambda^*_{exp} = 4\%$ (16 threads).

# points	N Threads	`sgsim` Optimized OpenMP (N threads)	`sisim` Optimized OpenMP (N threads)	`sgsim` Speedup GSLIB/ OpenMP	`sisim` Speedup GSLIB/ OpenMP
1,000,000	4	1336	371	19.498	7.919
1,000,000	8	669	186	38.938	15.795
1,000,000	16	336	94	77.529	31.255
2,000,000	4	5347	1493	19.409	7.872
2,000,000	8	2675	747	38.797	15.734
2,000,000	16	1342	376	77.334	31.260

TABLE 7.4: Execution time (seconds) and speedup obtained using optimized Fortran code and multi-thread OpenMP parallelization. These results are computed for benchmark purposes.

7.3.2 Analytical results

In order to understand the intrinsic relation between different values for λ and the overall performance, a key concept is the total number of pairs of points processed by the method. As depicted in Figure 7.4, if N scattered datapoints are being processed,

the total workload can be represented as

$$W_{total} = \frac{N * (N+1)}{2} \tag{7.2}$$

the GPU and CPU workloads as

$$\begin{aligned} W_{gpu} &= (1-\lambda)^2 W_{total} && (7.3) \\ W_{cpu} &= W_{total} - (1-\lambda)^2 W_{total} \\ &= (1 - 1 + 2\lambda - \lambda^2) W_{total} \\ &= (2\lambda - \lambda^2) W_{total} && (7.4) \end{aligned}$$

With this definitions, the execution time of the hybrid mode can be expressed as

$$T_{hybrid}(W_{total}) = \max\{T_{gpu}(W_{gpu}), T_{cpu}(W_{cpu})\} \tag{7.5}$$

with T_{hybrid}, T_{gpu} and T_{cpu} the execution time of each mode applied to each specific workload. If we assume that the execution time is a linear function of the workload, Equation (7.5) can be expressed as

$$\begin{aligned} T_{hybrid}(W_{total}) &= \max\{K_{gpu}(1-\lambda)^2 W_{total}, K_{cpu}(2\lambda - \lambda^2) W_{total}\} \\ &= K_{gpu} W_{total} \max\left\{(1-\lambda)^2, \frac{K_{cpu}}{K_{gpu}}(2\lambda - \lambda^2)\right\} && (7.6) \end{aligned}$$

The values of K_{gpu} and K_{cpu} can be interpreted as the experimental speedup rate obtained on each system. In our case, since the Tesla T4 GPU contains 40 SM[1], we can consider $K_{gpu} \approx \frac{1}{40}$ and $K_{cpu} \approx \frac{1}{T}$ where T is the number of threads used by the CPU (4, 8 or 16). Based on this assumption, we have

$$\frac{K_{cpu}}{K_{gpu}} > 1 \tag{7.7}$$

In order to find the value of λ that minimizes Equation (7.6), Figure 7.10 depicts the constraint that should be fulfilled:

$$(1-\lambda)^2 = \frac{K_{cpu}}{K_{gpu}}(2\lambda - \lambda^2) \tag{7.8}$$

By solving the previous equation, the minimum values for λ are (only solutions in range $[0, 1]$ are allowed):

$$\lambda^*_{analytic} = \frac{K_{gpu} + K_{cpu} \pm \sqrt{(K_{gpu} + K_{cpu}) K_{cpu}}}{K_{gpu} + K_{cpu}} \tag{7.9}$$

[1] https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

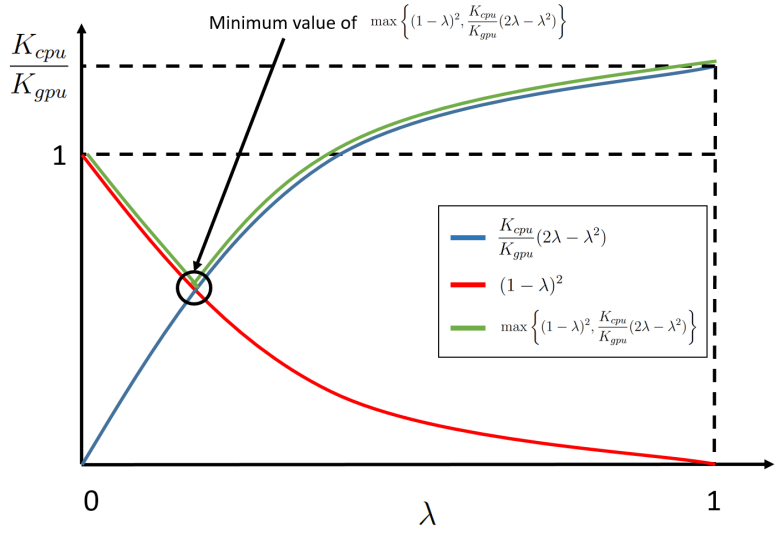

FIGURE 7.10: Execution time of hybrid mode across different values of λ according to Equations (7.6) and (7.7).

By applying the analytical expression from Equation (7.6), we obtain analytical optimal values, denoted $\lambda^*_{analytic}$, described in Tables 7.5 and 7.6. K_{gpu} and K_{cpu} are calculated from speedup measurements reported in Tables 7.5 and 7.6, columns "Speedup GSLIB/CUDA" and "Speedup GSLIB/OpenMP" respectively (K is the inverse value of the speedup reported in each case). For each value $\lambda^*_{analytic}$ inferred by the analytical expression with the prior speedup value, an execution was tested, obtaining its specific elapsed time and speedup against the sequential GSLIB execution.

Case	# points	N Threads	K_{gpu}	K_{cpu}	$\lambda^*_{analytic}$[%]	Elapsed time [s]	Speedup GSLIB/ Hybrid
`sgsim`	1000000	4	0.002278	0.051287	2.149	56.61	460.166
`sgsim`	1000000	8	0.002130	0.025681	3.906	56.26	463.028
`sgsim`	1000000	16	0.001970	0.012898	6.860	54.31	479.653
`sgsim`	2000000	4	0.002298	0.051522	2.159	241.52	429.707
`sgsim`	2000000	8	0.002122	0.025775	3.879	234.12	443.289
`sgsim`	2000000	16	0.001959	0.012930	6.810	220.14	471.440
`sisim`	1000000	4	0.008059	0.126582	3.039	24.02	122.314
`sisim`	1000000	8	0.007562	0.063492	5.471	23.00	127.739
`sisim`	1000000	16	0.006958	0.031994	9.370	21.17	138.781
`sisim`	2000000	4	0.007666	0.127032	2.887	90.21	130.295
`sisim`	2000000	8	0.007219	0.063556	5.237	86.35	136.120
`sisim`	2000000	16	0.006677	0.031989	9.043	79.93	147.053

TABLE 7.5: Execution time (seconds) and speedup obtained from `sgsim` and `sisim` scattered data sets using a GPU Tesla T4. K_{gpu}, K_{cpu} and analytical optimal λ^* values.

Case	# points	N Threads	K_{gpu}	K_{cpu}	$\lambda^*_{analytic}$[%]	Elapsed time [s]	Speedup GSLIB/ Hybrid
`sgsim`	1000000	4	0,001111	0,051285	1,066	28.29	920.820
`sgsim`	1000000	8	0,001096	0,025681	2,069	27.73	939.415
`sgsim`	1000000	16	0,001066	0,012898	3,894	25.89	1006.179
`sgsim`	2000000	4	0,001038	0,051520	0,992	98.58	1052.779
`sgsim`	2000000	8	0,001022	0,025774	1,926	96.12	1079.723
`sgsim`	2000000	16	0,000992	0,012930	3,628	90.14	1151.353
`sisim`	1000000	4	0.004257	0.126276	1.644	14.29	205.598
`sisim`	1000000	8	0.004257	0.063308	3.202	14.04	209.259
`sisim`	1000000	16	0.004257	0.031994	6.056	13.22	222.239
`sisim`	2000000	4	0.003640	0.127020	1.402	45.44	258.670
`sisim`	2000000	8	0.003640	0.063552	2.746	44.85	262.073
`sisim`	2000000	16	0.003640	0.031989	5.246	42.68	275.398

TABLE 7.6: Execution time (seconds) and speedup obtained from `sgsim` and `sisim` scattered data sets using a GPU Volta V100. K_{gpu}, K_{cpu} and analytical optimal λ^* values.

7.3.3 Heuristic results

Analytical results need at least two prior executions to estimate $\lambda^*_{analytic}$, using only the GPU device and using only the CPU cores. A drawback of this requirements is that the only-CPU execution probably will take much more time than what we are trying to optimize. In real applications, several runs of semivariogram calculation can be executed by researchers and practicioners over different simulated domains with the same amount of points (hundresds of simulations that should be statistically analyzed using `gamv`). Based on this empirical usability feature, an heuristic is proposed to search quickly for the heuristic optimal value $\lambda^*_{heuristic}$. It is based in the same analytical equations (7.6) and (7.7), but instead of using two prior executions, it uses a starting value λ^0 obtained using $K_{gpu} = \frac{1}{\#SM}$ and $K_{cpu} = \frac{1}{\#Cores}$. After this, a binary search can be applied in the direction towards $\lambda = 0$ first, or posteriorly towards $\lambda = 1$ if needed. Sequential iterations can be applied to update λ^k for $k > 0$. In Table 7.7 we can observe seven steps of a binary search starting from $\lambda^0 = 10.55\%$. The optimal value $\lambda^*_{heuristic}$ in this case is obtained in step $k = 5$.

Tables 7.8 and 7.9 show columns K_{gpu} and K_{cpu} based on the SMs and Cores of each system, elapsed time and speedup, after applying the heuristic in each case after different number of iterations of the dichotomic search. The stopping criteria is set in $k = 7$ iterations or until the execution time is faster than the experimental execution time.

k	Step applied	λ^k	Elapsed time [s]
0	λ^0	10.55	251,11
1	$\lambda^1 = \frac{\lambda^0}{2}$	5.27	226,76
2	$\lambda^2 = \frac{\lambda^1}{2}$	2.63	239,00
3	$\lambda^3 = \frac{\lambda^1+\lambda^2}{2}$	3.95	231,88
4	$\lambda^4 = \frac{\lambda^1+\lambda^3}{2}$	4.61	228,64
5	$\lambda^5 = \frac{\lambda^1+\lambda^4}{2}$	4.94	223,91
6	$\lambda^6 = \frac{\lambda^1+\lambda^5}{2}$	5.11	224,46

TABLE 7.7: Dichotomic search example for `sgsim` case with 2000000 points, starting from $\lambda^0 = 10.55\%$ using a GPU Tesla T4 and 16 OpenMP threads.

Case	# points	N Threads	K_{gpu}	K_{cpu}	$\lambda^*_{heuristic}$[%]	Iterations	Elapsed time [s]	Speedup GSLIB/ Hybrid
sgsim	1000000	4	0.025	0.2500	2.326	7	57.77	450.926
sgsim	1000000	8	0.025	0.1250	4.356	7	55.1	472.777
sgsim	1000000	16	0.025	0.0625	7.742	7	51.04	510.384
sgsim	2000000	4	0.025	0.2500	2.326	7	230.6	450.056
sgsim	2000000	8	0.025	0.1250	4.084	8	217.98	476.112
sgsim	2000000	16	0.025	0.0625	8.226	11	201.38	515.359
sisim	1000000	4	0.025	0.2500	2.908	7	23.19	126.693
sisim	1000000	8	0.025	0.1250	5.445	7	22.15	132.641
sisim	1000000	16	0.025	0.0625	11.613	7	20.33	144.515
sisim	2000000	4	0.025	0.2500	2.326	7	88.59	132.679
sisim	2000000	8	0.025	0.1250	4.356	7	84.39	139.282
sisim	2000000	16	0.025	0.0625	9.677	7	77.73	151.216

TABLE 7.8: Execution time (seconds) and speedup obtained from `sgsim` and `sisim` scattered data sets using a GPU Tesla T4 (40 SM). K_{gpu} and K_{cpu} are used to obtain λ^0 and heuristicaly obtained optimal $\lambda^*_{heuristic}$ values.

Case	# points	N Threads	K_{gpu}	K_{cpu}	$\lambda^*_{heuristic}$[%]	Iterations	Elapsed time [s]	Speedup GSLIB/ Hybrid
sgsim	1000000	4	0.0125	0.2500	0.903	7	24.54	1061.532
sgsim	1000000	8	0.0125	0.1250	1.745	7	23.89	1090.414
sgsim	1000000	16	0.0125	0.0625	3.267	7	22.32	1167.114
sgsim	2000000	4	0.0125	0.2500	0.903	7	90.08	1152.120
sgsim	2000000	8	0.0125	0.1250	1.745	7	87.48	1186.362
sgsim	2000000	16	0.0125	0.0625	3.267	7	81.63	1271.383
sisim	1000000	4	0.0125	0.2500	1.054	8	10.44	281.417
sisim	1000000	8	0.0125	0.1250	2.036	8	10.19	288.321
sisim	1000000	16	0.0125	0.0625	3.811	8	9.64	304.771
sisim	2000000	4	0.0125	0.2500	0.903	7	31.04	378.672
sisim	2000000	8	0.0125	0.1250	2.036	7	34.92	336.597
sisim	2000000	16	0.0125	0.0625	3.811	7	32.8	358.353

TABLE 7.9: Execution time (seconds) and speedup obtained from `sgsim` and `sisim` scattered data sets using a GPU Volta V100 (80 SM). K_{gpu} and K_{cpu} are used to obtain λ^0 and heuristicaly obtained optimal $\lambda^*_{heuristic}$ values.

Based on these results, we can conclude that the proposed model of Equations (7.6) and (7.7) is a good analytical tool to estimate the optimal λ which splits the workload in the hybrid execution mode of the parallel `gamv` application. It is expected that the proposed hybrid parallelization method can be applied to other applications, specificically in triangular symmetric computing scenarios, such as `gamv`. The developed model for optimal λ and its corresponding practical heuristic to search it, can be used to accelerate even more this type of applications.

7.4 Analysis

The results of Section 7.3 show that the hybrid mode outperforms the CUDA-only mode in the majority of tests, specially after applying the proposed heuristic. A summary of the speedup results using three different approaches can be observed in Figure 7.11. We can observe differences between both GPU devices, being the Volta V100 faster than the Tesla T4 on all tests. Additionally, the heuristic approach delivers faster executions than the other approaches (the stop criteria was set until the execution time is faster than the experimental execution time). If several executions of the application need to computed, following the optimal λ obtained after a few steps of the heuristic is a reasonable strategy to accelerate the overall workload. If only a single execution needs to be computed, using CUDA-only mode or estimate λ^0 are sub-optimal strategies, but they doesn't require computing priors which involves more execution time.

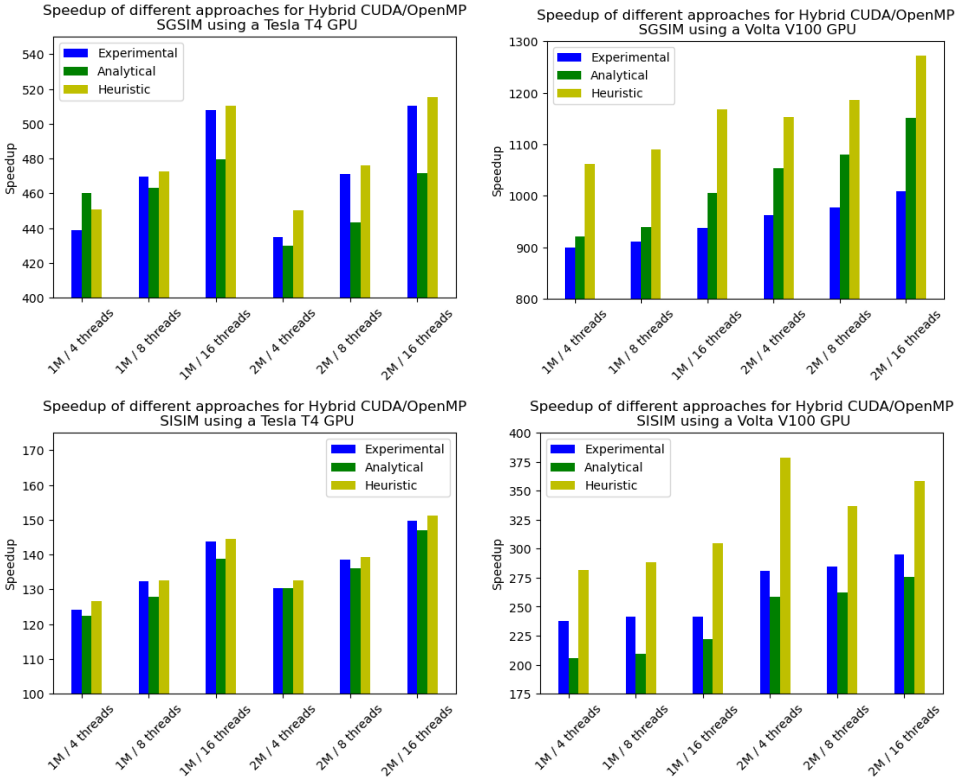

FIGURE 7.11: Speedup obtained with 3 approaches for `sgsim` and `sisim` case on two GPU devices.

In terms of numerical accuracy, all implementations use double-precision floating-point arithmetic, and there are no numerical differences with respect to the original GSLIB results. The reason of this accuracy level is that the exact same operations are applied on each multi or single thread execution. The application doesn't have stochastic or random operations internally, so each execution delivers the same results.

In terms of computational resources, the CUDA and hybrid parallelization uses a GPU device which defines the performance of the results according to its architectural features. Some of the results showed in this work are computed using a GPU model Tesla T4, which can be viewed as prohibitive since its market price is approximately 3000 USD [2]. However, the prices can be dramaticaly reduced by renting compute instances in cloud computing services, such as Microsoft Azure, Google Cloud or Amazon Web Services. For instance, in this work we choose the instance `g4dn.8xlarge` from Amazon Web Services which allocates a virtual machine with the same specifications described in the previous section, with a Tesla T4 device attached, for approximately 2 USD an hour[3].

[2] https://www.amazon.com/s?k=tesla+t4+gpu
[3] https://calculator.aws/#/createCalculator/EC2

Chapter 8

Conclusions

The present work focuses on improving the performance of a classical geostatistical simulation method: sequential simulation. Specifically, performance improvements were applied to two well-known methods: Sequential Gaussian and Sequential Indicator Simulation. Both methods were accelerated on two algorithmic modes: classical and non-euclidean mode. The approach followed to allow non-euclidean mode is through Locally Varying Anisotropy (LVA) represented by an additional LVA field defined as input in the specific applications.

This final chapter first reviews all the important results achieved in this dissertation, and later discusses future work that will follow from our research.

8.1 Summary of results

Parallel sequential simulation

The first task developed in order to accelerate the execution of sequential simulation methods was to decouple the computation of neighbours and the execution of stochastic simulation based on those neighbours. By doing this, the proposed parallel algorithm classifies each grid point according to its level of dependency from other points, and performs parallel simulation of points in the same level. The main applications adapted to this parallel algorithm were `sgsim` and `sisim`, both part of the GSLIB library for classical geostatistics, a well known tool in the applied geostatistics community. Best results were obtained by computing the simulations using wide search windows of 128 neighbours on each scenario (12× and 15× speedup respectively, both using 16 OpenMP threads). This contribution was published in the scientific journal *Computers and Geosciences*, and it served to open the path in our research.

Parallel neighbour search

A key bottleneck discovered in the initial contribution was related with the neighbour search. This was the first step that needed to be computed in the proposed parallel sequential simulation algorithm, and in many scenarios (small or mid size search windows) it was one of the largest contributors in the overall elapsed time. The second task in this work was accelerating the neighbour search of the proposed algorithm. The proposed search method is based on parallel kd-tree searches where previously simulated points are marked, and only marked points can be elegible to be selected as neighbours. Additional code optimizations were applied to the baseline code, and classical and non-euclidean scenarios were tested. The main applications adapted were classical `sgsim` and `sisim`, following the initial parallel method, and non-euclidean LVA-based `sgs-lva` and `sisim-lva`, using the same simulation methods. Regarding `sgsim` and `sisim`, best performance results were obtained using 64 neighbours as window search, with 12× and 50× speedup respectively. Regarding `sgs-lva` and`sisim-lva`, best performance results were obtained using 48 neighbours as window search and 1000/1344 landmark points, with 56× and 1822× speedup respectively. This last result was possible after a careful adaptation of the kd-tree search method to the baseline `sisim-lva` code, which improved the sequential execution by 111×. This contribution has been published in the scientific journal *Computers and Geosciences*, and it consolidated the research path initiated with the first article.

Parallel LVA routines

Two acceleration aspects of LVA-based applications were also analyzed and accelerated: parallel Dijsktra computation and parallel algebraic operations. Parallel Dijkstra computation was used in the LVA-related method that build the non-euclidean distance matrix required by the non-euclidean simulation methods. Parallel algebraic operations were used in the LVA-related method that build the multidimensional embedding that contains the coordinates in the p-dimensional space. These coordinates can be used to compute euclidean distances instead of non-euclidean distances in equivalent three dimensional space. Significant performance improvements were obtained by using optimized libraries in the applications, combined with the previous parallel methods, allowing us to speed-up the execution of LVA-based simulations.

Additional parallel applications

The last result is related with an additional parallel application, namely the semivariogram calculation in large domains. A CUDA-based and hybrid OpenMP/CUDA implementations were developed, allowing us to calculate fast spatial statistics in large domains, which can be used to validate statistical and structural similarities between simulated 3D images. The development of this accelerated application was intended as a helper tool when large simulations are being generated and fast statistical checks are

needed. An analytical model is proposed to estimate the optimal value used to split the workload between GPU and CPU.

8.2 Future work

Four main open research areas have been identified during the development process of this work:

- CUDA adaptation of parallel algorithms:
  Significant performance improvements can be obtained by switching the parallel framework from OpenMP to CUDA, particularly for parallel simulation of points in the same dependency level. CUDA-based algebraic operations can also be included in future versions of the parallel applications presented in this work (leveraging available functionality in libraries such as cuBLAS, cuSOLVER or similar).

- Approximation methods applied to sequential simulation:
  The approach developed in this work follows the standard generation of stochastic simulations by strictly visiting all domain points in a specific order defined at random initially. Every simulated node can be used in future iterations as conditioning data for neighbour non-simulated points. Potential methods that allow us to relax this constraints, in an approximate way, can be explored using the applications presented in this work as baseline.

- Accelerated LVA-based algorithms:
  The baseline LVA-based application selected in this work is based on the L-ISOMAP method for multidimensional scaling. Several other methods can be explored to expand the non-euclidean distances in three dimensions to euclidean distances in p dimensions: locally-linear embedding (LLE), local tangent space alignment (LTSA), t-distributed stochastic neighbor embedding (t-SNE), autoencoder neural networks, etc.

- Improve usability:
  Python and R wrappers can be developed to serve as interfaces to Fortran and C++/CUDA compiled applications. Additionally, cloud environments can be prepared in order to allow potential users to execute large computations remotely and without extra costs.

Bibliography

A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

F. Alabert. Stochastic Imaging of Spatial Distributions using Hard and Soft Information. Master's thesis, Stanford University, 1987.

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. ISBN 0-89871-447-8 (paperback).

AWS. Amazon Web Services Platform Price Calculator (December 2021). https://calculator.aws/, 2021.

J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509517, sep 1975. ISSN 0001-0782. doi: 10.1145/361002.361007.

M. F. P. Bierkens and P. A. Burrough. The indicator approach to categorical soil data. *Journal of Soil Science*, 44(2):361–368, 1993.

P. Biver, V. Zaytsev, D. Allard, and H. Wackernagel. *Geostatistics on Unstructured Grids, Theoretical Background, and Applications*, pages 449–458. Springer International Publishing, Cham, 2017.

L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

J. B. Boisvert. *Geostatistics with Locally Varying Anisotropy*. PhD thesis, University of Alberta, 2010.

J. B. Boisvert and C. V. Deutsch. Programs for kriging and sequential gaussian simulation with locally varying anisotropy using non-euclidean distances. *Comput. Geosci.*, 37(4):495–510, Apr. 2011.

Boost.org. Boost C++ libraries. 2012. URL http://www.boost.org.

T. Cheng. Accelerating universal kriging interpolation algorithm using cuda-enabled gpu. *Comput. Geosci.*, 54:178183, apr 2013. ISSN 0098-3004. doi: 10.1016/j.cageo.2012.11.013. URL https://doi.org/10.1016/j.cageo.2012.11.013.

J.-P. Chilès and P. Delfiner. *Geostatistics : modeling spatial uncertainty*. Wiley series in probability and statistics. Wiley, New York, 1999. ISBN 0-471-08315-1. URL http://opac.inria.fr/record=b1098313. A Wiley-Interscience publication.

F. C. Curriero. On the use of non-euclidean distance measures in geostatistics. *Mathematical Geology*, 38(8):907–926, 2006.

J. A. de Almeida. Stochastic simulation methods for characterization of lithoclasses in carbonate reservoirs. *Earth-Science Reviews*, 101(3-4):250 – 270, 2010.

V. de Silva and J. B. Tenenbaum. Sparse multidimensional scaling using landmark points. Technical report, Stanford, 2004.

L. E. de Souza and J. F. C. Costa. Sample weighted variograms on the sequential indicator simulation of coal deposits. *International Journal of Coal Geology*, 112: 154 – 163, 2013. Special issue on geostatistical and spatiotemporal modeling of coal resources.

D. dell'Arciprete, R. Bersezio, F. Felletti, M. Giudici, A. Comunian, and P. Renard. Comparison of three geostatistical methods for hydrofacies simulation: a test on alluvial sediments. *Hydrogeology Journal*, 20(2):299–311, 2012.

C. V. Deutsch. A sequential indicator simulation program for categorical variables with point and block data: Blocksis. *Computers & Geosciences*, 32(10):1669 – 1681, 2006.

C. V. Deutsch and A. G. Journel. *GSLIB: Geostatistical Software Library and user's guide*. Applied geostatistics series. Oxford Univ. Press, New York, NY, 2. ed. edition, 1998.

L. Devroye. *Non-Uniform Random Variate Generation*. Springer, 1 edition, Apr. 1986.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269271, 1959.

R. Dimitrakopoulos. Conditional simulation algorithms for modelling orebody uncertainty in open pit optimization. *International Journal of Surface Mining, Reclamation and Environment*, 12(4):173–179, 1998.

R. Dimitrakopoulos and M. Dagbert. *Sequential Modelling of Relative Indicator Variables: Dealing with Multiple Lithology Types*, pages 413–424. Springer Netherlands, Dordrecht, 1993.

R. Dimitrakopoulos and X. Luo. Generalized sequential gaussian simulation on group size ν and screen-effect approximations for large field simulations. *Mathematical Geology*, 36:567591, 2004. doi: 10.1023/B:MATG.0000037737.11615.df.

P. A. Dowd, C. Xu, K. V. Mardia, and R. J. Fowell. A comparison of methods for the stochastic simulation of rock fractures. *Mathematical Geology*, 39(7):697–714, 2007.

O. Dubrule and E. Damsleth. Achievements and challenges in petroleum geostatistics. *Petroleum Geoscience*, 7(S):S1–S7, 2001.

J. Gómez-Hernández and R. Srivastava. One step at a time: The origins of sequential simulation and beyond. *Mathematical Geosciencies*, (53):193–209, 2021. doi: https://doi.org/10.1007/s11004-021-09926-0.

Google. Google Cloud Platform Price Calculator (December 2021). https://cloud.google.com/products/calculator, 2021.

P. Goovaerts. Geostatistical modelling of uncertainty in soil science. *Geoderma*, 103 (1–2):3 – 26, 2001. Estimating uncertainty in soil models.

S. L. Graham, P. B. Kessler, and M. K. Mckusick. `gprof`: A Call Graph Execution Profiler. *SIGPLAN Not.*, 17(6):120–126, June 1982.

S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 39(4):4957, Apr. 2004. ISSN 0362-1340. doi: 10.1145/989393.989401. URL https://doi.org/10.1145/989393.989401.

J. L. Gustafson. Reevaluating amdahl's law. *Communications ACM*, 31(5):532533, May 1988.

R. Gutierrez and J. Ortiz. Sequential indicator simulation with locally varying anisotropy simulating mineralized units in a porphyry copper deposit. *Journal of Mining Engineering and Research*, 1(1):1–7, 01 2019. doi: 10.35624/jminer2019.01.01.

J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Amsterdam, 5 edition, 2012. ISBN 978-0-12-383872-8.

IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

E. Isaaks. The application of monte carlo methods to the analysis of spatially correlated data. Master's thesis, 1990.

E. H. Isaaks and R. M. Srivastava. *An Introduction to Applied Geostatistics*. Oxford University Press, USA, Jan. 1990.

M. E. Johnson. *Multivariate Statistical Simulation*. John Wiley, New York, NY, 1987.

M. K. Johnson and E. W. Troan. *Linux Application Development (2nd Edition)*. Addison-Wesley Professional, 2004.

JORC. Australasian code for reporting of exploration results, mineral resources and ore reserves (the JORC Code, 2012 Edition): Report prepared by the Joint Ore Reserve Committee of the Australasian Institute of Mining and Metallurgy, Australian Institute of Geoscientists and Minerals Council of Australia, 2012. URL http://www.jorc.org/.

A. Journel and F. Alabert. Non-gaussian data expansion in the earth sciences. *Terra Nova*, 1:123 134, 1989.

A. G. Journel and C. J. Huijbregts. *Mining geostatistics / [by] A. G. Journel and Ch. J. Huijbregts*. Academic Press London ; New York, 1978. ISBN 0123910501.

A. G. Journel and E. H. Isaaks. Conditional indicator simulation: Application to a saskatchewan uranium deposit. *Journal of the International Association for Mathematical Geology*, 16(7):685–718, 1984.

M. B. Kennel. Kdtree 2: Fortran 95 and c++ software to efficiently search for near neighbors in a multi-dimensional euclidean space, 2004.

J. Manchuk, O. Leuangthong, and C. V. Deutsch. *Direct Geostatistical Simulation on Unstructured Grids*, pages 85–94. Springer Netherlands, Dordrecht, 2005.

K. V. Mardia, J. T. Kent, and J. M. Bibby. *Multivariate Analysis*. Academic Press, 1979.

G. Mariethoz. A general parallelization strategy for random path based geostatistical simulation methods. *Computers & Geosciences*, 36(7):953 – 958, July 2010. ISSN 0098-3004. doi: 10.1016/j.cageo.2009.11.001. URL http://dx.doi.org/10.1016/j.cageo.2009.11.001.

G. Mariethoz and J. Caers. *Multiple-point Geostatistics: Stochastic Modeling with Training Images*. Wiley-Blackwell, 2014.

P. Martinsson, G. Quintana Ortí, N. Heavner, and R. van de Geijn. Householder qr factorization with randomization for column pivoting (hqrrp). *SIAM Journal on Scientific Computing*, 39(2):C96–C115, 2017. doi: 10.1137/16M1081270. URL https://doi.org/10.1137/16M1081270.

P. G. Martinsson, G. Quintana-Ortí, and N. Heavner. randutv: A blocked randomized algorithm for computing a rank-revealing utv factorization. *ACM Trans. Math. Softw.*, 45(1):4:1–4:26, Mar. 2019. ISSN 0098-3500. doi: 10.1145/3242670. URL http://doi.acm.org/10.1145/3242670.

B. Matérn. *Spatial variation (2nd Edition)*. Springer, New York, 1986.

G. Matheron, H. Beucher, C. de Fouquet, A. Galli, D. Guerillot, and C. Ravenne. *Conditional Simulation of the Geometry of Fluvio-Deltaic Reservoirs*, pages 123–131. Society of Petroleum Engineers, 1987.

Microsoft Azure. Microsoft Azure Price Calculator (December 2021), Sept. 2021. URL https://azure.microsoft.com/en-us/pricing/calculator/.

P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

R. Nunes and J. A. Almeida. Parallelization of sequential gaussian, indicator and direct simulation algorithms. *Computers & Geosciences*, 36(8):1042 – 1052, 2010.

R. Nussbaumer, G. Mariethoz, M. Gravey, E. Gloaguen, and K. Holliger. Accelerating sequential gaussian simulation with a constant path. *Comput. Geosci.*, 112:121–132, 2018. doi: 10.1016/j.cageo.2017.12.006. URL https://doi.org/10.1016/j.cageo.2017.12.006.

NVIDIA Corporation. CUDA C/C++ Streams and Concurrency (December 2021). https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf, 2017.

G. Pan. Conditional simulation as a tool for measuring uncertainties in petroleum exploration. *Nonrenewable Resources*, 6(4):285–293, 1997.

O. Peredo and J. M. Ortiz. Parallel implementation of simulated annealing to reproduce multiple-point statistics. *Computers & Geosciences*, 37(8):1110–1121, 2011.

O. Peredo, J. M. Ortiz, J. R. Herrero, and C. Samaniego. Tuning and hybrid parallelization of a genetic-based multi-point statistics simulation code. *Parallel Computing*, 40 (5):144–158, 2014.

O. Peredo, J. M. Ortiz, and J. R. Herrero. Acceleration of the Geostatistical Software Library (GSLIB) by code optimization and hybrid parallel programming. *Computers & Geosciences*, 85, Part A:210 – 233, 2015a.

O. Peredo, J. M. Ortiz, and O. Leuangthong. Inverse modeling of moving average isotropic kernels for non-parametric three-dimensional gaussian simulation. *Mathematical Geosciences*, 48(5):559–579, 2016.

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

O. F. Peredo, F. Navarro, M. Garrido, and J. M. Ortiz. Incorporating distributed dijkstras algorithm into variogram calculation with locally varying anisotropy. In *37th International Symposium APCOM 2015*, pages 1162–1170. S. Bandopadhyay, 2015b.

D. Pollard and R. Fletcher. *Fundamentals of structural geology*. Cambrige University Press, 2005.

M. Pyrcz, J. H., A. Kupenko, W. Liu, A. Gigliotti, T. Salomaki, and J. Santos. Geostatspy python package. https://github.com/GeostatsGuy/GeostatsPy. last checked: 29.11.2021.

L. G. Rasera, P. L. Machado, and J. F. C. Costa. A conflict-free, path-level parallelization approach for sequential simulation algorithms. *Computers & Geosciences*, 80:49 – 61, 2015.

N. Remy, A. Boucher, and J. Wu. *Applied Geostatistics with SGeMS: A User's Guide*. Cambridge University Press, 2009.

S. T. Roweis and L. K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *SCIENCE*, 290:2323–2326, 2000.

V. Savichev, A. Bezrukov, A. Muharlyamov, K. Barskiy, D. Nasibullina, and R. Safin. High performance geostatistics library. https://github.com/hpgl/hpgl. last checked: 29.11.2021.

L. Serrano, R. Vargas, V. Stambuk, C. Aguilar, M. Galeb, C. Holmgren, A. Contreras, S. Godoy, I. Vela, M. A. Skewes, and C. R. Stern. The Late Miocene to Early Pliocene Rio Blanco-Los Bronces Copper Deposit, Central Chilean Andes. In *Andean Copper Deposits: New Discoveries, Mineralization, Styles and Metallogeny*. Society of Economic Geologists, 01 1998. doi: 10.5382/SP.05.09. URL https://doi.org/10.5382/SP.05.09.

Statios LLC. WinGslib Installation and Getting Started Guide. Website, 2001. URL http://www.statios.com/WinGslib/GettingStarted.pdf. last checked: 29.11.2021.

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

P. Tahmasebi, M. Sahimi, G. Mariethoz, and A. Hezarkhani. Accelerating geostatistical simulations using graphics processing units (gpu). *Computers & Geosciences*, 46:51 – 59, 2012.

J. B. Tenenbaum, V. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319, 2000.

F. van der Meer. Sequential indicator conditional simulation and indicator kriging applied to discrimination of dolomitization in ger 63-channel imaging spectrometer data. *Nonrenewable Resources*, 3(2):146–164, 1994.

H. S. Vargas, H. Caetano, and M. Filipe. Parallelization of sequential simulation procedures. In *Proceedings of the Petroleum Geostatistics. EAGE (European Association of Geoscientists and Engineers)*. EAGE, 2007.

R. Versteeg. The marmousi experience: Velocity model determination on a synthetic complex data set. *The Leading Edge*, 13:927–936, 1994.

B. Wilkinson and C. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. An Alan R. Apt book. Pearson-/Prentice Hall, 2005.

J. M. Yarus, R. L. Chambers, M. Maucec, and G. Shi. Facies simulation in practice: Lithotype proportion mapping and plurigaussian simulation, a powerful combination. In *Geostatistics Oslo 2012*. Springer Netherlands, 2012.

V. Zaytsev, P. Biver, H. Wackernagel, and D. Allard. Change-of-support models on irregular grids for geostatistical simulation. *Mathematical Geosciences*, 48(4):353–369, May 2016.

Z. Zhang and H. Zha. Principal manifolds and nonlinear dimension reduction via local tangent space alignment. *SIAM Journal of Scientific Computing*, 26:313–338, 2002.