J. Comp. Int. Sci. (2017) 8(3):143-152
http://epacis.net/jcis/PDF_JCIS/JCIS-0131.pdf
jcis@epacis.net

Pan-american Association of Computational Interdisciplinary Sciences

A parallel implementation of the ordinary kriging algorithm for heterogeneous computing environments

Thales Luis Rodrigues Sabino[a1] Gisele Goulart Tavares[a], Leonardo Goliatt[a]
Marcelo Lobosco[a], Rodrigo Weber dos Santos[a] and Filipe de Oliveira Chaves[b]
[a]Federal University of Juiz de Fora, Juiz de Fora, Brazil
[b]State University of Rio de Janeiro, Rio de Janeiro, Brazil

Received on September 05, 2016 / accepted on October 19, 2017

Abstract

The Ordinary Kriging (OK) algorithm is an important method used in geostatistics applications for data interpolation usually based on field measurements of a variable of interest. The OK algorithm is used to estimate the value of a variable based on the variance of previous measurements. The OK algorithm has applications in areas such as environmental science, hydrogeology, mining, weather prediction and remote sensing. However, the OK algorithm is many times not suitable for large volumes of data due to its computational cost. In this work we present a study of a parallel implementation of the OK algorithm that takes advantage of heterogeneous computing environments through the usage of the Open Computing Language (OpenCL) framework. We use the OK algorithm to estimate a Digital Terrain Model (DTM) from a point cloud of a mangrove forest obtained via a terrestrial laser 3D scanning device. A downsampling procedure is applied to the raw data in order to extract the points that belong to the ground and then the OK algorithm is applied to obtain the DTM. The objective of our implementation is to effectively use all the computational resources available to accelerate the OK procedure. We show that our implementation can scale across multiple CPUs and GPUs devices and it is able to handle volumes of data that are typically produced by real world field measurements. The interpolation is a crucial step to make predictions of a variable at unknown locations, it is important to use the maximum amount of measurements possible in order to increase the performance of the prediction algorithm. Reducing the time to generate the interpolation model is, therefore, extremely important. We also compare the run times of a reference sequential implementation with our parallel one to show that the usage of parallel programming techniques can accelerate the fundamental steps of many important scientific researches.

Keywords: kriging, geostatistics, interpolation, GPGPU, parallelization, heterogeneous computing

[1]E-mail Corresponding Author: tluis@ice.ufjf.br

doi: 10.6062/jcis.2017.08.03.0131

1. Introduction

The Kriging process, also known as Gaussian Regression Process, is an interpolation method in which the interpolated values are modeled by a Gaussian process governed by prior co-variances. Under a set of suitable assumptions on the input data, the Kriging process gives the best linear unbiased prediction (BLUP). From its original design to predict ore grades from spatially correlated sample data in goldmines of South Africa, the Kriging interpolation method has, nowadays, applications in many fields such as petroleum engineering, geology, meteorology, hydrology, soil science, precision agriculture, pollution control, public health, fishery, plant and animal ecology, and remote sensing [1]. Ordinary Kriging (OK) is the most popular of its variants. The only requirements to apply OK is the knowledge of a variogram function and samples of data for its implementation. Therefore, OK is the default Kriging algorithm offered by many geographical information system (GIS) packages.

The constant increase in the amount of data that is generated by remote sensing devices can potentially make application of OK prohibitive in terms of computational time. The OK is used to find a Digital Terrain Model (DTM) from a mangrove forest obtained using a Terrestrial Laser Scanning (TLS) device. The final goal is to perform a point cloud normalization, as presented in [2], to further apply a tree segmentation algorithm in order to automate the point cloud processing of a forest captured using TLS devices. Being a fundamental step to DTM determination, the Kriging algorithm is an excellent candidate for parallelization in heterogeneous computing environments due to its parallel nature.

Nevertheless, there are various works where a parallelization of Kriging is presented and discussed. References [3–6] presented parallel implementations of kriging and other geostatistical procedures through the usage of the Message Passing Interface (MPI), OpenMP, Parallel Virtual Machines (PVMs) and/or Graphics Processing Units (GPUs). Despite all efforts on kriging's parallelization there is still room for improvements on the parallelization of this algorithm.

In summary, the OK algorithm needs to fit a function to an empirical semivariogram that is obtained through inspection in the spatial variance in the input data and then use this function to interpolate the variable value throughout the entire field. Manually sampling a field for some variable of interest can produce a couple hundred or even a thousand samples to interpolate data from. Even a sequential implementation can handle data

in this order of magnitude. However, kriging is being increasingly used on data collected using remote sensing devices. These devices can easily produce data in the order of tens of thousands samples and it should be possible to use kriging to process this data in a reasonable amount of time.

The majority of related works presented here have focused on the parallelization of the interpolation step, which tends to be the most expensive one. Although this might be true for manually acquired samples, when dealing with remote sensing devices, determine the empirical semivariogram can be as expensive as the interpolation step.

In this paper we present the design and implementation of a parallel version of the OK algorithm and we demonstrate its performance and scalability across multiple compute devices.

One important contribution from our work is the the parallelization of the semivariogram processing step. This avoids data exchange between host and devices and also leverages the processing power of both CPUs and GPUs available.

3. Kriging Formulation

Kriging is similar to Inverse Distance Weighted (IDW) method in that it weights the surrounding measured values to derive a prediction for an unmeasured location minimizing the variance of the estimator

$$\sigma_E^2(x) = Var\{\hat{Z}(x) - Z(x)\}, \quad \text{with } \hat{Z}(x_0) = \sum_{i=1}^{n} \lambda_i Z(x_i) \tag{1}$$

by modeling its variogram through the following equation

$$\gamma(h) = \frac{1}{2} E\big[(Z(x_i) - Z(x_j))^2\big], \forall i, j \tag{2}$$

which employs the variogram model $\gamma(h)$ and the weights λ_i in the following way

$$\sum_{i=0}^{N} \lambda_i \gamma(x_i, x_j) + \phi = \gamma(x_j, x_0), \forall j \tag{3}$$

where $\gamma(x_i, x_j)$ is the covariance between samples at location i and j respectively, $\hat{Z}(x_0)$ is the variable response value that needs to be interpolated at position x_0 and λ_i is the weighted coefficient of point i, $Z(x_i)$ is the response value measured at location x_i.

Writing Equation 3 into matrix notation, we can deduce the vector of weights λ:

$$\begin{bmatrix} \gamma_{11} & \gamma_{12} & \cdots & \gamma_{1n} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \gamma_{n1} & \gamma_{n2} & \ddots & \gamma_{nn} & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \vdots \\ \lambda_n \\ \phi \end{bmatrix} = \begin{bmatrix} \gamma_{10} \\ \vdots \\ \gamma_{n0} \\ 1 \end{bmatrix} \quad \Rightarrow \quad \mathbf{K}\lambda = k \quad \Rightarrow \quad \lambda = \mathbf{K}^{-1}k. \tag{4}$$

The γ function is defined as the covariance between two samples according to a covariance model. This can be one of many models, for this work we use the spherical covariance model which is defined as

$$y(x) = \begin{cases} c_0 + c\left[\frac{3h}{2a} - \left(\frac{h}{2a}\right)^3\right], & 0 < h \leq a \\ c_0 + c, & h > a \end{cases} \tag{5}$$

with $y(0) = 0$.

With this set of equations, the OK algorithm can be summarized in the following steps:

1. Using Equation 2, calculate the empirical semivariogram for the input dataset;
2. Fit a covariance model into the empirical semivariogram (we use the spherical covariance model shown in Equation 5;
3. Calculate the covariance matrix $\mathbf{K}$ with the function covariance function fitted into the empirical semivariogram;
4. Calculate the inverse covariance matrix $\mathbf{K}^{-1}$;
5. Determine the boundaries of the region where points must be interpolated;
6. Split the region in a grid of pixels for interpolation;
7. For each pixel in the region, solve the linear system shown in Equation 4 to find the weights vector λ;
8. Finally, for each pixel, solve Equation 1 to predict value $\hat{Z}(x)$ at location x.

4. Parallel Implementation

OpenCL, an acronym for Open Computing Language, was originally developed by Apple Incorporated and it is perhaps the first general programming standard aimed to tackle heterogeneous computing environments. It is free, cross-platform and has good interoperability [7].

As previously stated, we were able to write OpenCL kernels for all steps of the kriging algorithm except Step 4. Here, we provide implementation details regarding the algorithm's steps.

In both Steps 5 and 8 there is a common operation that need to be performed: Reduction. The straightforward way to implement Step 5 is to find the minimum and maximum points from the input dataset. This operation can be performed by a reduction operation using the *min* and *max* operators. When solving Equation 1 in Step 8 we perform a matrix vector multiplication followed by a dot product between λ and $Z(x)$. The multiplication part of the dot product if performed in parallel using an OpenCL kernel dedicated to the task and a parallel reduction with the *sum* operator is used to sum all the values producing a scalar value that represents the prediction $\hat{Z}(x_0)$. For more details on using OpenCL to perform parallel reductions refer to [8].

In order to determine the empirical semivariogram, Equation 2 must be solved. This equation involves calculating the difference squared between the values of paired locations. Often, each pair of locations have a unique distance so, to avoid an over saturated variogram and to reduce the number of calculations in Step 1, instead of generating an entry for each pair, they are grouped into lag bins. For example, compute the average semivariance for all pair of samples in the range 40m to 50m. The empirical semivariogram, therefore, is a graph of the averaged semivariogram values on the y-axis and lag distance on the x-axis.

An observation can be made regarding the parallelism that could be extracted in this operation. Each lag interval can be processed independently from the others and, within each lag, all pair-wise semivariance calculations can also be performed in parallel. To calculate the average distance and semivariance values, a reduction *sum* operation can be applied leaving only two divisions and a multiplication (Equation 2) to be performed by the host application. Performing a parallel reduction avoids unnecessary memory transfers and greatly reduces the host overhead with expensive computations.

The number of lag bins is a parameter for the algorithm and it is usually

a value between 10 and 30. The number of lags correspond to the number of points in the empirical semivariogram graph. Thus, fitting a function using a covariance model (Equation 5) into the semivariogram is an almost inexpensive operation.

Step 8 is performed on-the-fly with two nested loops where the position (x, y) of each pixel to be interpolated is computed. The values of these pixels are sent to a kernel for calculating the covariance between the location to be interpolated with respect to its distance to neighboring samples using the empirical semivariogram fitted model. This, in fact, calculates weights vector λ. Following, we finish solving Equation 1 as described earlier.

As previously stated, each lag interval can be processed independently. In order to maximize the usage of a platform resources, an OpenCL command queue is created for each available device. Each command queue can be used by a host thread to simultaneously dispatch kernels execution and memory operations to the associated devices. In the beginning of these parallel regions, a command queue is assigned to each OpenMP thread through a *rounding robin* scheme. This ensures that our implementation can scale across any number of available compute devices.

5. Experiments

In this paper the OK algorithm is used to perform a terrain interpolation with the final goal to do a point cloud normalization. The point cloud used in the experiments comes from scanning a portion of the Guaratiba Biological Reserve, a conservation unit that encompass a mangrove forests complex.

The dataset used in this study is comprised of text files in the simple XYZ format, where each line represents a location (x, y, z) in a 3D space. Prior to perform the terrain's interpolation, a data downsampling is performed by finding the points with the lowest z-value in a grid of 0.25 m^2 block size. This has the objective of segmenting the points that belongs to the ground eliminating any tree data in the point cloud. The file that was exported by the scanner software has 15 070 181 points that were down-sampled to 7176 points using the grid minimum filter. This is the size of data used in the kriging interpolation.

In order To interpolate the terrain, points are treated as measured samples at location (x, y) with response z. The idea is to predict new z-values from the "sampled" data.

The experiments were performed on a machine that has 64 AMD Opteron(TM) Processor 6272 cores grouped into 8 physical chips with 128GB of RAM memory and 4 NVIDIA Tesla M2090 GPUs each with 6GB of GDDR5 RAM

384-bit RAM memory. This machine was used for the experiments to show that our implementation can scale across multiple computing devices.

To test the performance of the algorithm we performed an Ordinary Kriging terrain interpolation using a 300x300 grid with 10 lags. All experiments use the same input dataset of 7176 points but with a varying number of computing devices, also, each experiment was performed 5 times and the results represents average runtimes.

6. Results

In order to show that the OK algorithm can gain a lot of performance with parallelism, we did a rough comparison with a reference implementation made with the R language. Using the same dataset as the other experiments, a simple serial implementation in C++, we achieve a speedup of 7.1x. Using OpenCL the speedup increases to 8.0x when comparing the total execution time. Although this may not seen like a great improvement, when comparing the total execution time, the vast majority of time is spent inverting the covariance matrix (see Table 2). The other experiments shows the scalability of our implementation.

The first measurement correspond to the total runtime of the program, including input/output (IO) operations and OpenCL setup time. In Table 1 we show this measurement varying the number of GPU devices used in the execution.

# GPUs	**Runtime (s)**
1	170.9764
2	149.8212
3	143.5154
4	140.2316

Table 1: Total runtime of the kriging algorithm using multiple GPU devices

It is easily noticeable that we achieve a poorly speedup value even with four very powerful GPUs. In this execution the speedup was only 1.2 with 4 GPUs. However, this is explained by the fact that the vast majority of the runtime is spent inverting the covariance matrix. As previously explained, Step 4 is performed by the host application (CPU). This inverse matrix operation is performed by the Eigen, a free C++ template library for linear algebra and related algorithms [9]. Despite the possibility to use OpenMP as a processing backend for Eigen, this is still the most time consuming part of the algorithm. This is shown in Table 2, 69% of the execution time is spend

inverting the covariance matrix. Although this is a considerably amount, it only needs to be performed once. With the increase in the number of points to be interpolated, the inverse matrix time could be outperformed leading to a better speedup.

Operation	**Percentage**
Distances Matrix	0.01%
Covariance Matrix Matrix	0.00%
Inverse Covariance Matrix	69.00%
Semivariogram Fitting	4.97%
Kriging Prediction	21.47%
Total	**95.46%**

Table 2: Table showing the percentage of execution time spent by some steps of the kriging algorithm. The remaining 4.54% correspond to IO and OpenCL initialization times.

As state in Section 3, we were able to implement a OpenCL kernel for all steps except Step 4. The second most expensive portion of the algorithm is the Step 8. According to Table 2 it is responsible for 21.47% of the execution time so we choose this step to show that our implementation can scale with the number of available computing devices. In this step there is one thread per compute device operating independently using different command queues. In Figure 1 (left) one can see that the prediction runtime greatly reduces with the increase of the number of GPUs. In order to show that the algorithm can scale well, we present the graph of Figure 1 (right) We can achieve a quasi-linear speedup.

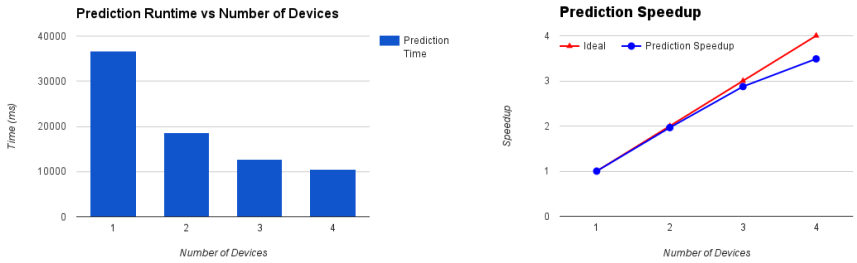

Figure 1: Kriging prediction (Step 8): runtime in milliseconds vs the number of GPU devices (left) and speedup increase where the *ideal* line can be used as reference (right).

Another important measurement when evaluating a parallel implementation is efficiency.

The efficiency is a measure that summarize how efficient the computational resources are being used. The efficiency tends to drop a little even in a good implementation due to the overhead of a parallel program. Nevertheless, Table 3 shows the efficiency with the number of compute devices for the prediction step. It is clear that we were able to achieve a very good efficiency value even with 4 devices.

# GPUs	**Efficiency**
1	1.00
2	0.98
3	0.95
4	0.87

Table 3: Efficiency of the kriging prediction step with the increase of the number of compute devices.

7. Conclusion

In this work we present a parallel implementation of the Ordinary Kriging interpolation algorithm that takes advantage of heterogeneous computing environments through the usage of the OpenCL framework and the OpenMP parallel extensions. We were able to achieve a quasi-linear speedup in the second most time consuming step of applying the kriging algorithm, the prediction step. We also show that our implementation can achieve a high efficiency (87%) even when the number of computing devices increases.

We also provide the source code used in the experiments at `https://bitbucket.org/pgmc-ufjf/parallelordinarykriging`.

Acknowledgments

The authors would like to thanks the The Graduate Program in Computational Modeling of the Federal University of Juiz de Fora (PGMC-UFJF), the NEMA Group of State University of Rio de Janeiro (NEMA-UERJ) for providing the dataset used in this study and FAPEMIG for the financial support.

References

[1] M. A. Oliver and R. Webster. A tutorial guide to geostatistics: Computing and modelling variograms and kriging. *Catena*, 113:56–69, 2014.

[2] Shengli Tao, Fangfang Wu, Qinghua Guo, Yongcai Wang, Wenkai Li, Baolin Xue, Xueyang Hu, Peng Li, Di Tian, Chao Li, Hui Yao, Yumei Li, Guangcai Xu, and Jingyun Fang. Segmenting tree crowns from terrestrial and mobile LiDAR data by exploring ecological theories. *ISPRS Journal of Photogrammetry and Remote Sensing*, 110(March 2016):66–76, 2015.

[3] Lluís Pesquer, Ana Cortés, and Xavier Pons. Parallel ordinary kriging interpolation incorporating automatic variogram fitting. *Computers and Geosciences*, 37(4):464–473, 2011.

[4] A. J Rossini, Luke Tierney, and Na Li. Simple Parallel Statistical Computing in R. *Journal of Computational and Graphical Statistics*, 16(2):399–420, 2007.

[5] Accelerating geostatistical simulations using graphics processing units (GPU). *Computers & Geosciences*, 46:51–59, 2012.

[6] Tangpei Cheng. Accelerating universal Kriging interpolation algorithm using CUDA-enabled GPU. *Computers and Geosciences*, 54:178–183, 2013.

[7] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12(3):66–72, 2010.

[8] Bryan Catanzaro. Opencl optimization case study: Simple reductions. http://developer.amd.com/resources/articles-whitepapers/opencl-optimization-case-study-simple-reductions, 2010.

[9] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.