Acceleration of the Geostatistical Software Library (GSLIB) by Code Optimization and Hybrid Parallel Programming

Oscar Peredo [*†] Julián M. Ortiz[† ‡] José R. Herrero [§]

December 2015

Abstract

The Geostatistical Software Library (GSLIB) has been used in the geostatistical community for more than thirty years. It was designed as a bundle of sequential Fortran codes, and today it is still in use by many practitioners and researchers. Despite its widespread use, few attempts have been reported in order to bring this package to the multi-core era. Using all CPU resources, GSLIB algorithms can handle large datasets and grids, where tasks are compute and memory intensive. In this work, a methodology is presented to accelerate GSLIB applications using code optimization and hybrid parallel processing, specifically for compute-intensive applications. Minimal code modifications are added decreasing as much as possible the elapsed time of execution of the studied routines. If multi-core processing is available, the user can activate OpenMP directives to speed up the execution using all resources of the CPU. If multi-node processing is available, the execution is enhanced using MPI messages between the compute nodes. Four case studies are presented: experimental variogram calculation, kriging estimation, sequential gaussian and indicator simulation. For each application, three scenarios (small, large and extra large) are tested using a desktop environment with 4 CPU-cores and a multi-node server with 128 CPU-nodes. Elapsed times, speedup and efficiency results are shown.

1 Introduction

The Geostatistical Software Library (GSLIB), originally presented by Deutsch and Journel [1], has been used in the geostatistical community for more than thirty years. It contains plotting utilities (histograms, probability plots, Q-Q/P-P plots, scatter plots, location maps), 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 their 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 [2], SGeMS [3] and HPGL [4] being the most relevant efforts. SGeMS and HPGL moves away from Fortran and implements Python and C/C++ code in conventional and new algorithms. Although there is a significant gain with this change, 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 package.

According to the authors' knowledge, few efforts have been reported in order to accelerate the GSLIB package by itself: analyzing, optimizing and accelerating the original Fortran routines. In this work we present case studies of accelerations performed on original GSLIB routines (in their Fortran 90 versions), using code optimization and multi-core programming techniques. We explain our methodology, in which a performance profile is obtained from the original routine, with the aim of identifying overhead sources in

[*]Advanced Laboratory for Geostatistical Computing (ALGES), University of Chile., (Chile.) operedo@alges.cl
[†]Advanced Mining Technology Center (AMTC), University of Chile., (Chile)
[‡]Department of Mining Engineering, University of Chile., (Chile) jortiz@ing.uchile.cl
[§]Computer Architecture Dept., Universitat Politècnica de Catalunya (UPC) - BarcelonaTech, (Spain). josepr@ac.upc.edu.

the code. After that, incremental modifications are applied to the code in order to accelerate the execution. OpenMP [5] directives and MPI [6] instructions are added in the most time consuming parts of the routines. Similar experiences in other geostatistical codes have been reported in [7] and [8].

2 GSLIB structure

According to GSLIB documentation [1], 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* (Fig. 1). 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.

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 [9] 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 the first step into its journey.

With these concepts in mind, in the next sections we show how to apply a methodology to accelerate GSLIB applications and utilities.

3 Methodology

Re-design: 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.

Profiling and code optimization: After re-design, we have to study the run-time behavior of the application using a profiler tool. In our case we choose the Linux-based tools `gprof` [10], `strace/ltrace` [11] and `Oprofile` [12]. These tools can deliver several statistics, among the most important are: elapsed time per routine or line of code, number of system/library calls, number of calls per routine, number of L1/L2 cache misses per line of code and number of misspredicted branches per line of code. Each of these statistics will be described in further sections. 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. In this work, only hardware-independent optimizations will be included, because of portability issues.

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, for example in simulation-based loops or grid-based loops. 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. If original GSLIB unmodified routines are parallelized, this analysis can be very tedious and error prone. For this reason, the first step of the methodology is fundamental in order to facilitate this analysis. Further examples of common problems if the re-design is not applied will be detailed in the next sections.

MPI parallelization: After single-node multi-thread processing has been defined, multi-node MPI processing can be added in a straightforward way once the data-flow of variables has been studied. As for OpenMP directives, the most time consuming parts of the code can be even more parallelized among distributed parallel processes.

At last, we must re-optimize the application in order to exploit efficiently all resources (memory, I/O, CPU and network) running multiple threads of execution distributed among multiple compute nodes. When using several threads in a multi-node system, new bottlenecks and sources of overhead can arise. For this reason we must repeat the second step of the methodology, using multi-node multi-thread profiling tools ([13], [14]) whenever is possible.

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      stop
12   end
```

Figure 1: Original main program for `gamv`, `kt3d`, `sgsim` and `sisim` applications.

4 Case study

The proposed methodology was applied to accelerate four GSLIB applications: `gamv`, `kt3d`, `sgsim` and `sisim`. We tested the final versions of the applications in two Linux-based systems: the *Server*, running SUSE operating system with multiple nodes of 2x8-cores Intel Xeon CPU E52670 2.60GHz interconnected through a fast Infiniband FDR10 network, and the *Desktop*, running openSUSE operating system with a single node of 1x4-cores Intel Xeon CPU E31225 3.10GHz. All programs were compiled using GCC `gfortran` version 4.7 and Open MPI `mpif90` version 1.8.1, supporting OpenMP version 3.0, with the option `-O3` 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.

In the first application, `gamv`, synthetic 2D and 3D datasets of normal scored Gaussian random fields are used as base dataset. In the other three applications a real mining 3D dataset of copper grades (2376 samples) is used as base dataset. In this 3D dataset, the lithology of the samples corresponds to Granodiorite (15%), Diorite (69%) and Breccia (16%).

Through all this section, we use the notation defined in [1], where the basic entities are random functions or fields (RF) which are sets of continuous location-dependent random variables (RV), such as $\{Z(\mathbf{u}) : \mathbf{u} \in \Omega\}$. For each application we include a brief description of the main mathematical formulas required by the underlying algorithm.

4.1 `gamv`

The measurement of the spatial variability/continuity of a variable in the geographic region of study is a key tool in any geostatistical analysis. The `gamv` application calculates several of these 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 [1] (III.1). Among the most used, we can mention the semivariogram, which

is defined as

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

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) and $Z(\mathbf{u}_i+\mathbf{h})$ is the corresponding end (head). In Algorithm 1 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 of the measure to be calculated. For example, to calculate the semivariogram (eq. 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 1), 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  β ← zeros(nvar × nlag × ndir × nvarg);
2  for i ∈ {1, ..., |Ω|} do
3  |  for j ∈ {i, ..., |Ω|} do
4  |  |  for (id, iv, il) ∈ {1, ..., ndir} × {1, ..., nvarg} × {1, ..., nlag} do
5  |  |  |  (p_i, p_j) ← ((x_i, y_i, z_i), (x_j, y_j, z_j)) ∈ Ω × Ω;
6  |  |  |  if (p_i, p_j) satisfy tolerances τ_id ∧ ||(p_i − p_j) − h_id × il × h|| ≈ 0 then
7  |  |  |  |  β ←save_statistics(V_{i,h_iv}, V_{i,t_iv}, V_{j,h_iv}, V_{j,t_iv}, type_iv);
8  |  |  |  end
9  |  |  end
10 |  end
11 end
12 γ ← build_variogram(β);
13 write(output.txt, γ);
```

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

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

The small test scenario consists in calculating semivariogram values of 30 lags with separation $h = 4.0$ using the north direction in the XY plane ($\mathbf{h} = (0, 1, 0)$), with a regularly spaced 2D dataset of $256 \times 256 \times 1$ (65,536) nodes. The large and extra large test scenarios use 20 lags with separation $h = 2.0$, the east direction in the XY plane ($\mathbf{h} = (1, 0, 0)$) and a regularly spaced 3D dataset of $50 \times 50 \times 50$ (125,000) and $100 \times 100 \times 100$ (1,000,000) nodes respectively. All datasets correspond to a Gaussian random field with exponential covariance model. Even though a special version of `gamv` is available for regularly spaced datasets, called `gam`, we have decided to test the scalability of the optimization and parallelization using the general application `gamv`, which can also be applied without loss of generality in regular or irregular grids. In Figures 2 and 3 we can see the small and large datasets with their corresponding semivariogram plots.

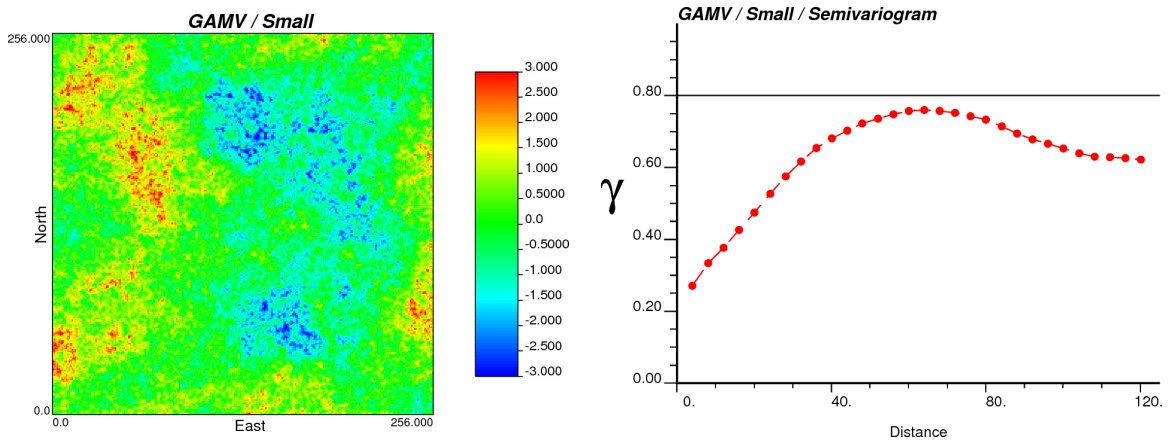

Figure 2: Input dataset (left) and output (right) of `gamv` application using a small scenario.

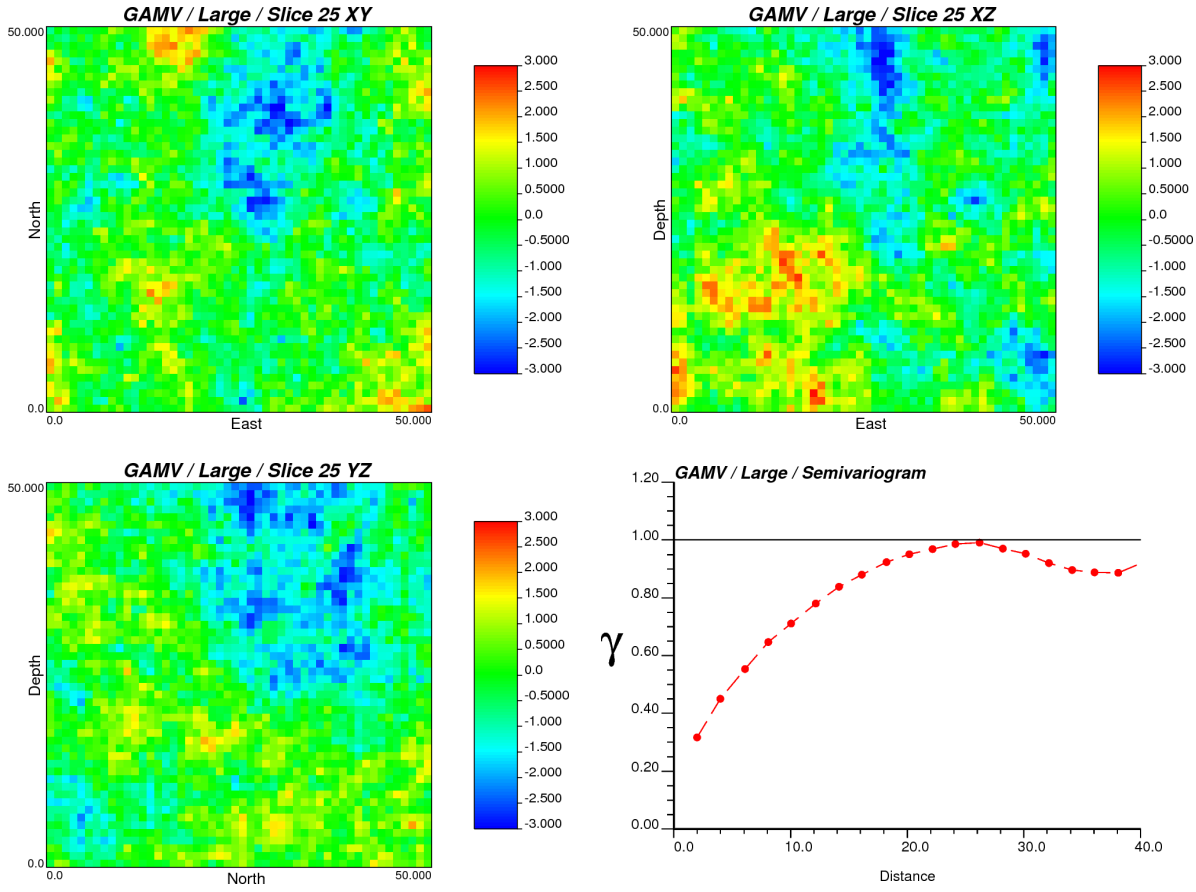

Figure 3: Input dataset (top-left, top-right and bottom-left) and output (bottom-right) of `gamv` application using a large scenario.

Re-design: Applying the first step of the methodology to the original `gamv` code of Figure 1, a slight acceleration is observed, with a speedup of 1.09x/1.11x/1.12x for the small/large/extra large scenarios. All variable declarations inside of `geostat` module were relocated at the beginning of the main program. After that, the routines `readparm`, `gamv` and `writeout` were manually inlined. Finally all internal routine calls were redesigned allowing only argument-based communication between the caller and the callee. No global variables which are modified at run-time are allowed through the usage of include files or module variables. Additionally, whenever is possible, the usage of the statement `implicit none` is mandatory.

Profiling and code optimization: In the second step of the methodology we are interested in observing the outputs of all tools that can extract information during execution. In Table 1 we can see the output of `ltrace` for the small scenario (the large and extra large scenarios deliver the same information). The tools `gprof` and `strace` do not give us any useful information about possible bottlenecks or overhead sources during execution in any scenario. `gprof` accounts for subroutines execution timing, but in this case all the code is self contained in `gamv`. `strace` accounts for system calls during the execution and since only small scale I/O operations are performed at the begining and end of the execution, no significant information is collected. The output of `ltrace` is reporting a large usage of the internal library function `_gfortran_transfer_real`. This function belongs to the native GCC Fortran library `libgfortran`, and performs casting functions among single and double precision floating-point numbers. Although a major source of overhead is added by this native function, optimizing this library function is not straightforward and requires critical modifications of internal library routines. Examining `gamv` code we infer that most of this type casting is used in critical parts where numerical accuracy is mandatory, like double-precision square roots and trigonometric functions. For this reason we decided to skip it and focus our efforts in other high-level overhead sources. If we inspect the implemented algorithm, we can observe that almost all work is done in loops of lines 2 and 3 of Algorithm 1. Instrumenting the re-designed code with `system_clock` calls before and after these loops, approximately 99.767% of the elapsed time is spent on them.

% time (`ltrace`, small)	seconds	calls	name
77.87	112.12	2400020	`_gfortran_transfer_real`
12.40	17.85	480031	`_gfortran_st_read`
9.69	13.95	480031	`_gfortran_st_read_done`
0.04	-	-	other routines

Table 1: Output of `ltrace` tool applied to `gamv` application (small scenario) running in *Desktop* system. Large and extra large scenarios deliver the same information. Timing column ("seconds") of `ltrace` may not show accurate measurements and must be taken only as an approximation reference.

Other profiling tool at our disposal is `Oprofile`. Different hardware events can be queried with this tool, such as the number of branch instructions, number of miss-predicted branches, number of unhalted CPU cycles and number of L1/L2 data cache events (loads, stores, cache misses and evictions). For a complete description of each hardware event and architecture details about performance-monitoring counters in Intel based architectures, see [15]. After extracting several line-by-line profiles with this tool, the most influential event in the overall execution time is the number of miss-predicted branch instructions. More that 50% of this event is measured in a few lines of code. In Figure 4 we can see the original and optimized lines of code. After the proposed optimization, a speedup of 6.01x/2.86x/2.76x was reached (Table 2). In addition, the results delivered by the optimized application match exactly with the results of the original application.

```
1   ...
2   do ilag=2,nlag+2
3      if(h.ge.(xlag*real(ilag-2)-xltol).and.
4  +      h.le.(xlag*real(ilag-2)+xltol)) then
5            if(lagbeg.lt.0) lagbeg = ilag
6            lagend = ilag
7      end if
8   end do
9   ...
```

```
1   ...
2   xlaginv=1.0/xlag
3   liminf=(ceiling((h-xltol)*xlaginv)+2)
4   limsup=(floor((h+xltol)*xlaginv)+2)
5   if(lagbeg.lt.0) then
6      do ilag=liminf,limsup
7         lagbeg = ilag
8         lagend = ilag
9      end do
10  else
11     do ilag=liminf,limsup
12        lagend = ilag
13     end do
14  end if
15  ...
```

Figure 4: Original (left) and optimized (right) `gamv` code. In the original code, we can see a source of miss-predicted branches in lines 3-4. Pre-computing the limits of the loop in line 2 and hoisting out a static condition in line 5 of original code, we can obtain an optimized version with 50% less miss-predicted branches.

Optimization	Small	Large	Extra large
Baseline	1.00x	1.00x	1.00x
Re-design	1.09x	1.11x	1.12x
Reduction of miss-predicted branches	6.01x	2.86x	2.76x

Table 2: Summary of code optimizations in `gamv` application running in *Desktop* system.

OpenMP parallelization: For the third step of the methodology, we add the OpenMP `parallel for` pragma before the loop of line 2 in Algorithm 1, using a run-time defined schedule for loop splitting.

MPI parallelization: The fourth step consists in a blocked partition of the same loop of line 2, using a block size defined by the user. The main steps of the multi-thread multi-node algorithm are depicted in Algorithm 2. The parallelization of the optimized application still delivers similar results to the original `gamv` application, allowing some tolerance up to the 5th or 6th decimal number (non-commutativity of floating-point operations).

In line 8 of this algorithm, we define a thread-local statistics container $\boldsymbol{\beta}_{ithread}$, with the same dimensions as the global container $\boldsymbol{\beta}$. This local container stores the statistics measured by each thread. After finishing the process of each local chunk of iterations from i_{start} to i_{end}, we merge the statistics into the global container, as shown in line 21. Finally, in lines 26 and 27 the master thread builds the variogram values with the global statistics collected by all threads. As mentioned before, the fourth step of the methodology is applied splitting the same loop of line 2 from Algorithm 1 using a block size B defined by the user. In line 3 of Algorithm 2 the value K indicates the number of blocks in which the total number of iterations $|\Omega|$ can be divided. Additionally in Algorithm 2 an outer block-based loop is defined in line 4, where each MPI task handles a block of iterations that will be divided among the local threads, as shown in lines 5 and 6. In order to collect all results among different nodes, the MPI instruction `MPI_Reduce` is launched by all MPI tasks in line 24 of Algorithm 2. With this instruction, all tasks send their local statistics $\boldsymbol{\beta}$ to the master task (`itask` $= 0$) which adds all of them in the local container $\boldsymbol{\beta}^{global}$.

A key step is the multi-thread loop splitting. As we can observe, if we split the loop of line 2 of the single-thread algorithm (Algorithm 1) using the default "`static`" schedule, an unbalanced workload will be performed by all threads (Figure 5-left), since the outer and inner loops in lines 2 and 3 form a triangular iteration space [16]. For this reason, we allow the user to set the runtime schedule, depending on the size of the input dataset. In these particular test scenarios, small and large, we have inferred that the optimal schedules are "`static,32`" and "`static,1024`" respectively, evaluating all possible schedules in the form "`{static,dynamic,guided},{8,16,...,512,1024}`".

Results: In table 3 we can see the single-node average elapsed time of five executions using the optimal

schedules and the corresponding speedup compared against the original GSLIB routine. Using the *Server* system, we have achieved a maximum speedup of 69.56x, 60.99x and 37.46x for the small, large and extra large respectively, using 16 CPU-cores. Using the *Desktop* system, the maximum achieved speedups with 4 CPU-cores are 22.14x, 11.23x and 9.59x respectively. Using the optimized sequential time as baseline, the maximum achieved speedups are 10.89x, 13.66x and 13.81x for the small, large and extra large scenarios in the *Server* system and 3.67x, 3.91x and 3.47x respectively in the *Desktop* system. In terms of efficiency, defined as speedup/#processes, these values represent 68%, 85% and 86% in the *Server* system and 91%, 97% and 86% in *Desktop*, for the small, large and extra large scenarios respectively.

The multi-node results can be viewed in table 4, where the maximum achieved speedup is 232.07x for the extra large scenario with 16 CPU-cores and 8 compute nodes, a total of 128 distributed CPU-cores. Using the optimized sequential time with one node as baseline, the maximum achieved speedup is 85.54x using the same distributed 128 CPU-cores. This speedup accomplishes a 66% of efficiency using 128 CPU-cores. Comparatively, using 64 and 32 distributed CPU-cores and the optimized baseline, the average efficiency obtained is 79% and 86% respectively. The efficiencies obtained using 64 CPU-cores are 77% with 4 nodes and 16 threads each, and 81% with 8 nodes and 8 threads each. Using 32 CPU-cores, the efficiencies are 81% with 2 nodes and 16 threads each, 88% with 4 nodes and 8 threads each, and 90% with 8 nodes, 4 threads each.

The decay in efficiency observed in these results is related with the overhead added by thread synchronization and contention when the threads are writing to the local-node global container $\boldsymbol{\beta}$. This container is implemented as a two dimensional shared array with size $(nvar \times nlag \times ndir \times nvarg, T)$ where each thread uses the space $\boldsymbol{\beta}(:, \texttt{ithread})$ to save their local results. With this implementation, contention may appear if $nvar \times nlag \times ndir \times nvarg$ is smaller than the cache line size divided by the size in bytes of $\boldsymbol{\beta}$ data type (double-precision float).

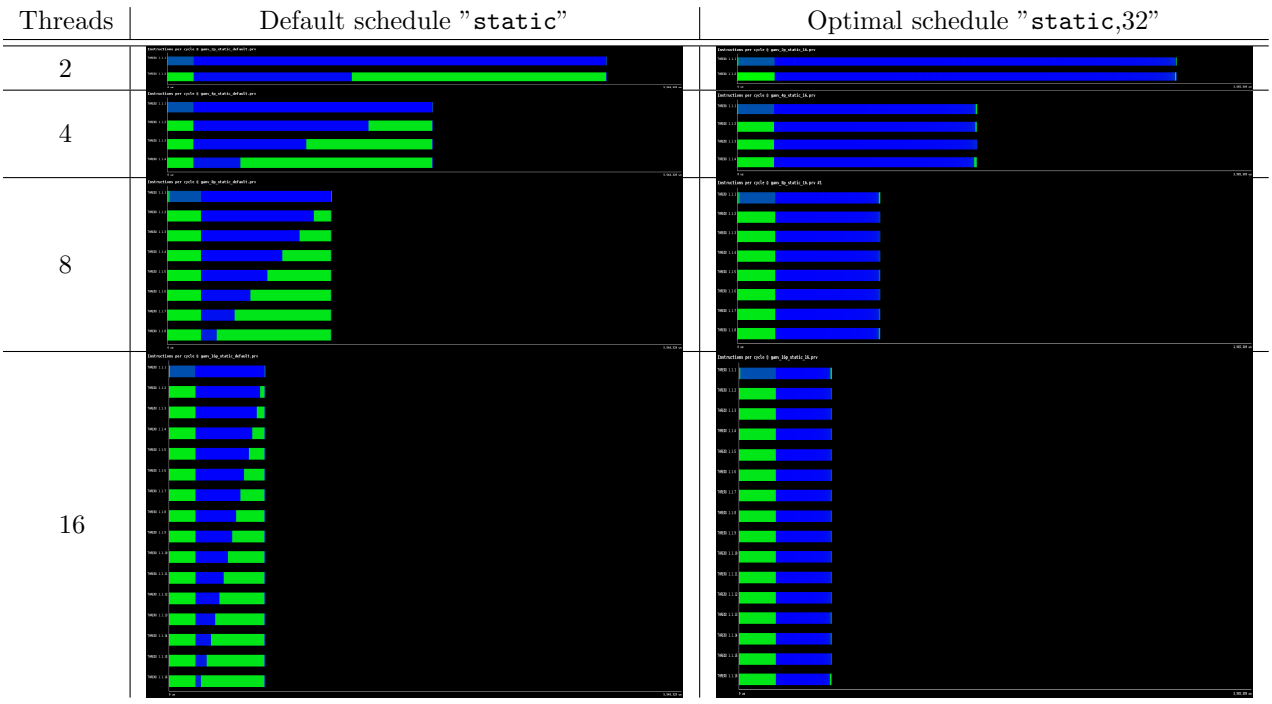

Figure 5: Trace views of number of instructions per cycle (IPC) generated with `extrae/paraver` applied on multi-thread version of `gamv`, small test scenario. Left column: using default schedule. Right column: using `"static,32"` schedule. Green color indicates low IPC values, blue color indicates high IPC values.

Input: Same inputs as Algorithm 1; P: number of MPI tasks; T: number of execution threads per MPI task; S: runtime schedule of thread synchronization; B: block size for each MPI task;

$\boldsymbol{\beta} \leftarrow$ `zeros`$(nvar \times nlag \times ndir \times nvarg)$;
$itask \leftarrow$ `MPI_COMM_RANK(...)`;
$K \leftarrow \left\lceil \frac{|\Omega|-B+1}{B} \right\rceil$; `// Count the number of blocks of size` B `that fit in` $|\Omega|$
for $k \in \{1, \ldots, K\}$ **do**
  if `itask` $== (k \ mod \ P)$ **then**
    $(k_{start}, k_{end}) \leftarrow (k * B, \min\{(k+1) * B, |\Omega|\})$;
    `/* omp parallel default(firstprivate) shared(`$\mathbf{V}, \Omega$`) */`
    foreach $ithread \in \{1, \ldots, T\}$ **do**
      $\boldsymbol{\beta}_{ithread} \leftarrow$ `zeros`$(nvar \times nlag \times ndir \times nvarg)$;
      $(i_{start}, i_{end}) \leftarrow$ `loop_split`$(ithread, k_{start}, k_{end}, S, T)$;
      for $i \in \{i_{start}, \ldots, i_{end}\}$ **do**
        for $j \in \{i, \ldots, |\Omega|\}$ **do**
          for $(id, iv, il) \in \{1, \ldots, ndir\} \times \{1, \ldots, nvarg\} \times \{1, \ldots, nlag\}$ **do**
            $(\mathbf{p}_i, \mathbf{p}_j) \leftarrow ((x_i, y_i, z_i), (x_j, y_j, z_j)) \in \Omega \times \Omega$;
            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**
              $\boldsymbol{\beta}_{ithread} \leftarrow$ `save_statistics`$(\mathbf{V}_{i,h_{iv}}, \mathbf{V}_{i,t_{iv}}, \mathbf{V}_{j,h_{iv}}, \mathbf{V}_{j,t_{iv}}, type_{iv})$;
            end
          end
        end
      end
    end
    $\boldsymbol{\beta} \leftarrow$ `merge_statistics`$(\boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_T)$;
  end
end
$\boldsymbol{\beta}^{global} \leftarrow$ `MPI_REDUCE(`$\boldsymbol{\beta}$`,target=0)`;
if $itask == 0$ **then**
  $\boldsymbol{\gamma} \leftarrow$ `build_variogram`$(\boldsymbol{\beta}^{global})$;
  `write(output.txt,`$\boldsymbol{\gamma}$`)`;
end

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

Algorithm 2: Pseudo-code of `gamv`, measurement of spatial variability/continuity (multi-thread multi-node algorithm)

4.2 `kt3d`

The kriging algorithm 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 [1] (IV.1). Different versions of the algorithm are available in the `kt3d` application. Simple Kriging (SK), in its stationary version, aims to obtain a linear regression estimator $Z^*_{SK}(\mathbf{u})$ defined as

$$Z^*_{SK}(\mathbf{u}) = \sum_{\alpha=1}^{n} \lambda_\alpha^{(SK)}(\mathbf{u}) Z(\mathbf{u}_\alpha) + \left[1 - \sum_{\alpha=1}^{n} \lambda_\alpha^{(SK)}(\mathbf{u})\right] m \tag{2}$$

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

$$\sum_{\beta=1}^{n} \lambda_\beta^{(SK)}(\mathbf{u}) C(\mathbf{u}_\beta, \mathbf{u}_\alpha) = C(\mathbf{u}, \mathbf{u}_\alpha), \ \alpha = 1, \ldots, n \tag{3}$$

#Threads	Time[s] (Speedup) *Server* Small 1 node	*Server* Large 1 node	*Server* Extra large 1 node	*Desktop* Small 1 node	*Desktop* Large 1 node	*Desktop* Extra large 1 node
1 (base gslib)	121.04 (1.0x)	535.52 (1.0x)	9956.0 (1.0x)	166.28 (1.0x)	508.66 (1.0x)	9159.0 (1.0x)
1 (optimized)	18.96 (6.38x)	119.95 (4.46x)	3670.0 (2.71x)	27.63 (6.01x)	177.35 (2.86x)	3318.0 (2.76x)
2 (optimized)	9.60 (12.60x)	60.12 (8.9x)	1888.35 (5.27x)	13.86 (11.99x)	87.90 (5.78x)	1901.2 (4.81x)
4 (optimized)	5.04 (24.01x)	30.61 (17.49x)	960.96 (10.36x)	7.51 (22.14x)	45.28 (11.23x)	954.88 (9.59x)
8 (optimized)	3.06 (39.55x)	16.03 (33.40x)	497.60 (20.00x)	-	-	-
16 (optimized)	1.74 (69.56x)	8.78 (60.99x)	265.72 (37.46x)	-	-	-

Table 3: Single-node time/Speedup results for `gamv`. Small scenario: 2D, 30 lags, east direction, 65536 data points, schedule "`static,32`". Large scenario: 3D, 20 lags, north direction, 125000 data points, schedule "`static,1024`". Extra large scenario: 3D, 20 lags, north direction, 1000000 data points, schedule "`static,1024`"

#Threads	Time[s] (Speedup) *Server* Extra Large 2 nodes	*Server* Extra Large 4 nodes	*Server* Extra Large 8 nodes
1 (optimized)	1854.73 (5.36x)	947.14 (10.51x)	478.54 (20.80x)
2 (optimized)	948.30 (10.49x)	475.70 (20.92x)	243.37 (40.90x)
4 (optimized)	479.91 (20.74x)	244.62 (40.69x)	127.18 (78.28x)
8 (optimized)	251.57 (39.57x)	130.27 (76.42x)	70.05 (142.12x)
16 (optimized)	140.52 (70.85x)	73.82 (134.86x)	42.90 (232.07x)

Table 4: Multi-node time/speedup results for `gamv` with extra large scenario

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), imposing that the sum of weights is equal to one. The resulting OK estimator is as

$$Z^*_{OK}(\mathbf{u}) = \sum_{\alpha=1}^{n} \lambda_\alpha^{(OK)}(\mathbf{u}) Z(\mathbf{u}_\alpha) \tag{4}$$

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

$$\sum_{\beta=1}^{n} \lambda_\beta^{(OK)}(\mathbf{u}) C(\mathbf{u}_\beta - \mathbf{u}_\alpha) + \mu(\mathbf{u}) = C(\mathbf{u} - \mathbf{u}_\alpha),\ \alpha = 1, \ldots, n \tag{5}$$

$$\sum_{\beta=1}^{n} \lambda_\beta^{(OK)} = 1 \tag{6}$$

with $\mu(\mathbf{u})$ a Lagrange parameter associated with the second constraint (6). 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). Other kriging algorithms available add different trend models in 1, 2 or 3D. These last algorithms are not supported yet by the multi-thread proposed version of `kt3d`. The main steps of the kriging algorithms can be viewed in Algorithm 3. The super block spatial search strategy ([1], II.4) is implemented in order to accelerate the search of neighbor values in cases where large datasets are available (lines 1 and 3). Once the neighbor values $\mathbf{N}$ are available, the covariance matrix values $C(\mathbf{u}_\beta - \mathbf{u}_\alpha)$ must be assembled into the matrix $\mathbf{A}$ (line 4). Additionally, the right-hand side is composed by the covariance values $C(\mathbf{u} - \mathbf{u}_\alpha)$ and is stored in the array $\mathbf{b}$ (line 5). With the left and right-hand sides $\mathbf{A}$ and $\mathbf{b}$ already computed, the weight vector $\boldsymbol{\lambda}$ can be calculated solving the SK linear system (2) or the OK linear system (5)-(6) (line 6). Finally the estimated value is computed using the weights $\boldsymbol{\lambda}$ (line 7) and stored in a file (line 8).

Input: $(\mathbf{V}, \Omega)$: sample data base values defined in a 3D domain Ω; r: radius of search (neighbour of node to estimate); β: super block strategy parameters; γ: structural variographic models;

```
1 H ←set_spatial_hash(V,Ω,β,r);
2 for i ∈ {1,...,|Ω|} do
3 |   N ←search_spatial_hash(i,H,V);
  |   /* N ⊂ V, neighbourhood values of node i */
4 |   A ←assemble_covariance_matrix(N,γ);
5 |   b ←assemble_rhs_vector(N,γ);
6 |   λ ←solve_system(A,b);
7 |   y ←compute_estimation(λ,N);
8 |   write(output.txt,y);
9 end
```

Output: Output file `output.txt` with estimated values for all nodes

Algorithm 3: Pseudo-code of `kt3d`, kriging 3D estimation (single-thread algorithm)

The small test scenario consists in performing an Ordinary Kriging estimate in a grid of $40 \times 60 \times 12$ nodes (28,800), representing a 3D volume of $400 \times 600 \times 120$[m^3] with a search radius of 100[m] and a maximum of 50 neighbours to include in the local estimation. The large and extra large test scenarios have the same parameters as the small scenario but using finer grids of $120 \times 180 \times 24$ and $120 \times 180 \times 48$ nodes (518,400 and 1,036,800 respectively) in the same 3D volume. All estimation grids use the same variographic inputs, which are a spherical plus an exponential structure, and the same input dataset (2376 copper grade 3D samples). In Figure 6 we can see slices of the small and large kriging outputs.

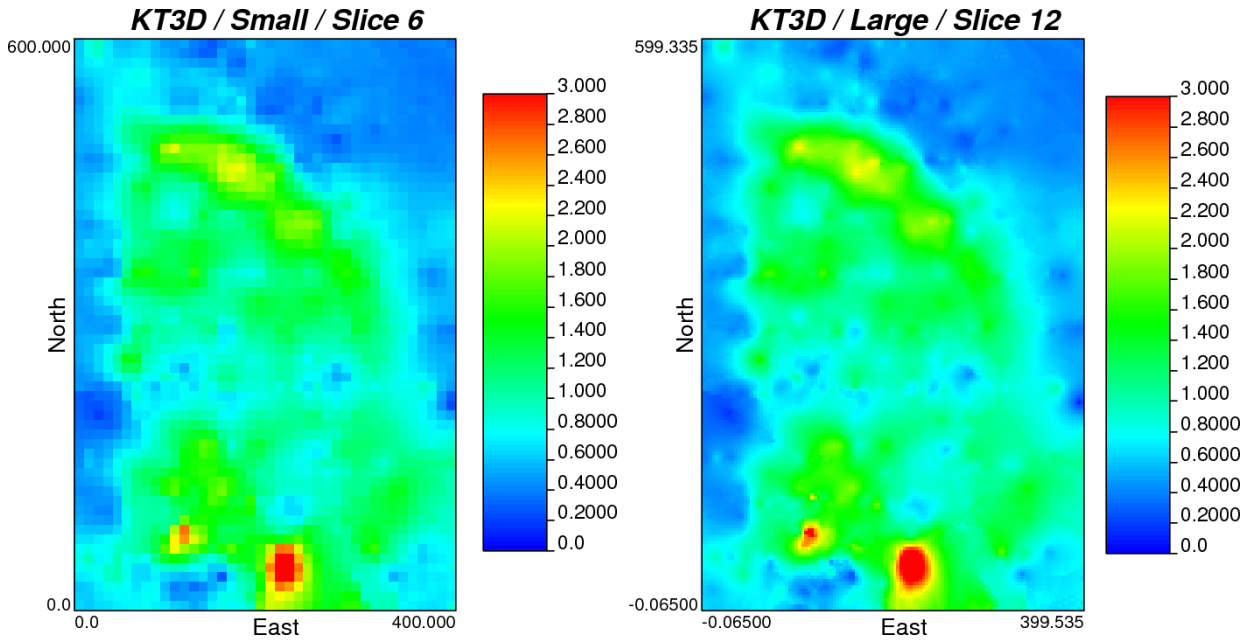

Figure 6: Output of `kt3d` application using a small (left) and large (right) scenarios.

Re-design: Applying the first step of the methodology to the original `kt3d` code as in the `gamv` application, a slight acceleration is observed, with a speedup of 1.03x/1.03x/1.02x for the small/large/extra large scenarios.

Profiling and code optimization: For the second step of the methodology, `gprof` tool delivers information depicted in Table 5 for the small (top), large (medium) and extra large (bottom) scenarios. All these routines

are utilities from the `gslib.a` library. In the three scenarios, the most called routines are `cova3` and `sqdist`, where `cova3` calculates the covariance value $C(\mathbf{u}, \mathbf{u}+\mathbf{h})$ according to the variographic model passed as input, and `sqdist` calculates a squared anisotropic distance between two points. `sortem` implements a sorting algorithm for floating-point single precision arrays in ascending order. `srchsupr` implements a spatial search using a super block strategy. `ktsol` solves a floating-point double precision linear system by gaussian elimination with partial pivoting. In terms of elapsed time, a predominance of `cova3` is observed in the three scenarios, with 35%, 30% and 27% in each case. The routine `srchsupr` shows a steady increase of participation from 4% to 26% of the elapsed time, which is proportional to the scenario size. In Table 6 we can observe the output of `ltrace` for the three scenarios. In all of them, the library function `expf` is executed by the application several times, however its low-level optimization is out of the scope of this work. The tool `strace` does not report any useful information about possible bottlenecks or overhead sources during execution in all scenarios.

% time (`gprof`, small)	seconds	calls	name
35.12	1.77	36946936	`cova3`
30.95	1.56	28798	`ktsol`
19.25	0.97	85615247	`sqdist`
5.75	0.29	1	`MAIN`
4.56	0.23	28800	`srchsupr`
4.17	0.21	28801	`sortem`
0.02	-	-	other routines

% time (`gprof`, large)	seconds	calls	name
30.82	32.00	665354374	`cova3`
26.79	27.82	518358	`ktsol`
16.43	17.06	1485946159	`sqdist`
16.07	16.69	518400	`srchsupr`
5.21	5.41	518401	`sortem`
4.61	4.79	1	`MAIN`
0.06	-	-	other routines

% time (`gprof`, extra large)	seconds	calls	name
27.07	63.96	1330982767	`cova3`
26.05	61.54	1036800	`srchsupr`
23.84	56.32	1036728	`ktsol`
14.18	33.50	2968320694	`sqdist`
4.52	10.69	1036801	`sortem`
4.27	10.08	1	`MAIN`
0.07	-	-	other routines

Table 5: Output of `gprof` tool applied to `kt3d` application (small, large and extra large scenarios) running in *Desktop* system. Timing column ("seconds") may not show accurate measurements and must be taken only as an approximation reference.

Analyzing `Oprofile` outputs, we can observe a large amount of branch instructions executed in `cova3` routine. A basic optimization applied to the code is to specialize this routine for the input parameters. In this case, the variographic structures passed as inputs define which branch instructions will be executed. The first structure is a spherical model and the second is an exponential model. In Figure 7 we can observe the original and specialized routine `cova3`, using these two variographic structures as input. A speedup of 1.06x/1.05x/1.05x is obtained after this optimization for the small/large/extra large scenarios. Additionally, removing dead code not used by the multi-thread version of the application can speed up the execution time. Using preprocessor directives as `#ifdef DEBUG` or `#ifdef _OPENMP` allow us to remove all unnecessary code at compilation time and reduce the number of instruction cache misses. A speedup of 1.11x/1.08x/1.07x is obtained after this optimization (Table 7). Another easy modification involves the use of a buffer array to store the estimated results in each grid node, followed by one single `write` instruction. In line 8 of Algorithm 3 we can observe that a `write` call is performed for each grid node. A slight acceleration is obtained using the buffering technique (1.11x/1.10x/1.09x), however this feature slows down the execution

% time (`ltrace`, small)	seconds	calls	name
97.54	125.14	1992512	`expf`
1.44	1.84	19046	`_gfortran_transfer_real`
1.02	-	-	other routines

% time (`ltrace`, large)	seconds	calls	name
97.93	125.81	2912965	`expf`
1.79	2.30	19046	`_gfortran_transfer_real`
0.28	-	-	other routines

% time (`ltrace`, extra large)	seconds	calls	name
99.65	125.51	2675566	`expf`
0.14	0.17	19046	`_gfortran_transfer_real`
0.21	-	-	other routines

Table 6: Output of `ltrace` tool applied to `kt3d` application (small, large and extra large scenarios) running in *Desktop* system. Timing column ("seconds") may not show accurate measurements and must be taken only as an approximation reference. In this case it shows the measurements running the application approximately by two minutes

using multiple threads. A possible reason for this behaviour might be contention generated by false sharing [17]. False sharing can happen because the buffer array must be shared among all threads, even if they write in different memory locations of it. The usage of the shared array adds contention in the CPU-memory bus, in order to synchronize and manage all threads, for this reason we decide to skip this optimization and leave the code as it is. In order to accelerate the I/O operations, a binary format in the output file is adopted, adding the option `form='UNFORMATTED'` in the `open` instruction. A speedup of 1.12x/1.10x/1.06x is observed using this modification. Regarding the number of cache misses and evictions in the L1 and L2 cache levels, the routine `srchsupr` shows the largest number of these events, which is in concordance with the size of scenario used in the tests. As shown in the previous paragraph with the tool `gprof`, the elapsed time of the routine `srchsupr` becomes larger if the size of the scenario increases. If the estimation grid size increases, the number of cache misses and eviction events increases proportionally. In order to optimize these features the access pattern can be improved by using a single container for the contiguous super block offsets stored in the arrays `ixsbtosr(1:N)`, `iysbtosr(1:N)` and `izsbtosr(1:N)`. The container can be denoted `ixyzsbtosr(1:3N)` and it can store the three values for a particular index i contiguously, following the access pattern implemented in `srchsupr`. Although this modification reduces the number of cache misses and evictions approximately 10% to 15% in the large and extra large scenarios, its effect in the total elapsed time is negligible. For this reason we decided to skip this optimization and leave the code of the original routine. Significant effects of this optimization probably can be observed using even larger scenarios, where the number of cache misses and evictions dominates the bottleneck factors of the execution. As for the `gamv` application, the optimized `kt3d` application delivers the same results as the original `kt3d` application.

Optimization	Small	Large	Extra large
Baseline	1.00x	1.00x	1.00x
Re-design	1.03x	1.03x	1.02x
Specialization of `cova3`	1.06x	1.05x	1.05x
Remove dead code	1.11x	1.08x	1.07x
Unbuffered binary writing	1.12x	1.10x	1.06x

Table 7: Summary of code optimizations in `kt3d` application running in *Desktop* system.

OpenMP parallelization: For the third step of the methodology, the OpenMP `parallel for` pragma is added in the loop of line 2 from Algorithm 3. The default schedule of loop splitting is used.

MPI parallelization: The fourth step of the methodology consists in using MPI to split even more the same loop. With MPI task-based combined with OpenMP thread-based loop splittings, multi-node processing is available using all CPU-cores in each compute node. The parallelization of the optimized application still

```
 1   ...
 2   cova = 0.0
 3   do is=1,nst(ivarg)
 4     ist = istart + is - 1
 5     if(ist.ne.1) then
 6       ir = min((irot+is-1),MAXROT)
 7        hsqd=sqdist(x1,y1,z1,x2,y2,z2,
 8  +                 ir,MAXROT,rmat)
 9     end if
10     h = real(dsqrt(hsqd))
11     if(it(ist).eq.1) then
12       hr = h/aa(ist)
13       if(hr.lt.1.) then
14         cova=cova +
15  +      cc(ist)*(1.-hr*(1.5-.5*hr*hr))
16       end if
17     else if(it(ist).eq.2) then
18       cova = cova +
19  +    cc(ist)*exp(-3.0*h/aa(ist))
20     else if(it(ist).eq.3) then
21       ...
22     else if(it(ist).eq.4) then
23       ...
24     else if(it(ist).eq.5) then
25       ...
26     endif
27   end do
28   return
29   end
30   ...
```

```
 1   ...
 2   cova = 0.0
 3   is=1
 4   ist = istart + is - 1
 5   h = real(dsqrt(hsqd))
 6   hr = h/aa(ist)
 7   if(hr.lt.1.) then
 8      cova=cova+cc(ist)*(1.-hr*(1.5-.5*hr*hr))
 9   end if
10   is=2
11   ist = istart + is - 1
12   ir = min((irot+is-1),MAXROT)
13   hsqd=sqdist(x1,y1,z1,x2,y2,z2,ir,MAXROT,rmat)
14   h = real(dsqrt(hsqd))
15   cova = cova +
16  +       cc(ist)*exp(-3.0*h/aa(ist))
17   return
18   end
19   ...
```

Figure 7: Original (left) and specialized (right) `cova3` utility. The specialized code works for two additive variographic structures (`nst(ivarg)=2`), spherical (`it(1)=1`) and exponential (`it(2)=1`), and performs a full unroll of the main loop.

delivers similar results to the original GSLIB application, allowing some tolerance up to the 5th or 6th decimal number (non-commutativity of floating-point operations).

The main steps of the multi-thread multi-node algorithm are depicted in Algorithm 4. In line 3, each MPI task selects a balanced share of the iterations of loop 2 from Algorithm 3. After that, in line 6 of Algorithm 4 each local thread opens a file `output-`$itag$`.txt` where it will store the estimated values of each local chunk of iterations from j_{start} to j_{end} in the loop of lines 8-15. The value of $itag$ is a unique identifier of each thread in each compute node. The rest of the algorithm is the same as the single-thread version.

Results: Analyzing the single-node results of table 8, the maximum achieved speedup is 13.87x for the small scenario, 14.48x for the large scenario and 13.88x for the extra large scenario using 16 CPU-cores in the *Server* system. Using the *Desktop* system, the maximum achieved speedups with 4 CPU-cores are 3.95x, 3.96x and 3.79x respectively. Using the optimized sequential time as baseline, the maximum achieved speedups are 12.43x, 13.72x and 12.92x for the small, large and extra large scenarios in the *Server* system and 3.55x, 3.60x and 3.57x respectively in the *Desktop* system. In terms of efficiency (speedup/#CPU-cores), these values represent 77%, 85% and 81% in the *Server* system and 88%, 90% and 89% in *Desktop*, for the small, large and extra large scenarios respectively.

Regarding the multi-node results of table 9, the maximum achieved speedup is 56.78x for the extra large scenario with 16 CPU-cores and 8 compute nodes, a total of 128 distributed CPU-cores. Using the optimized sequential single-node time as baseline, the maximum achieved speedup is 52.84x using the same distributed 128 CPU-cores. This speedup accomplishes a 41% of efficiency using 128 CPU-cores. Comparatively, using 64 and 32 distributed CPU-cores and the optimized baseline, the average efficiency obtained is 56% and 71% respectively. The efficiencies obtained using 64 CPU-cores are 55% with 4 nodes and 16 threads each, and 57% with 8 nodes and 8 threads each. Using 32 CPU-cores, the efficiencies are 69% with 2 nodes and 16 threads each, 71% with 4 nodes and 8 threads each, and 72% with 8 nodes, 4 threads each.

The decay in efficiency observed in these results is related with overhead generated by thread synchro-

nization and bus contention when the threads are writing to disk. No further overhead is added by network contention since no MPI messages are shared between tasks during the main computations. Further tests are needed in order to check if multiple threads can use the I/O library buffer in a contention-free way, and also to explore if external OS factors can affect the behaviour of the threads when writing to disk.

Input: Same inputs as Algorithm 3; P: number of MPI tasks; T: number of execution threads per MPI task; S: runtime schedule of thread synchronization;

```
1  itask ← MPI_COMM_RANK(...);
2  H ←set_spatial_hash(V,Ω,β,r);
3  (i_start, i_end) ← loop_split(itask, 1, |Ω|, "static", P);
   /* omp parallel default(firstprivate) shared(H,V,Ω) */
   /* omp for schedule(S) , S="static"*/
4  foreach ithread ∈ {1,...,T} do
5      itag ← ithread + itask × T;
6      open(output-itag.txt);
7      (j_start, j_end) ← loop_split(ithread, i_start, i_end, S, T);
8      for j ∈ {j_start,...,j_end} do
9          N ←search_spatial_hash(j,H,V);
           /* N ⊂ V, neighbourhood values of node j */
10         A ←assemble_covariance_matrix(N,γ);
11         b ←assemble_rhs_vector(N,γ);
12         λ ←solve_system(A,b);
13         y ←compute_estimation(λ,N);
14         write(output-itag.txt,y);
15     end
16 end
```

Output: Output files `output-itag.txt` with estimate values for all nodes

Algorithm 4: Pseudo-code of `kt3d`, kriging 3D estimation (multi-thread multi-node algorithm)

#Threads	Time[s] (Speedup)					
	Server Small 1 node	*Server* Large 1 node	*Server* Extra large 1 node	*Desktop* Small 1 node	*Desktop* Large 1 node	*Desktop* Extra large 1 node
1 (base gslib)	5.69 (1.0x)	108.62 (1.0x)	244.74 (1.0x)	5.66(1.0x)	112.27 (1.0x)	246.76 (1.0x)
1 (optimized)	5.10 (1.11x)	102.94 (1.05x)	227.78 (1.07x)	5.08 (1.12x)	102.06 (1.10x)	232.79 (1.06x)
2 (optimized)	2.59 (2.19x)	52.18 (2.08x)	118.03 (2.07x)	2.76 (2.05x)	52.85 (2.12x)	124.03 (1.98x)
4 (optimized)	1.33 (4.27x)	26.64 (4.07x)	60.67 (4.03x)	1.43 (3.95x)	28.30 (3.96x)	65.06 (3.79x)
8 (optimized)	0.71 (8.01x)	13.90 (7.81x)	32.10 (7.62x)	-	-	-
16 (optimized)	0.41 (13.87x)	7.50 (14.48x)	17.62 (13.88x)	-	-	-

Table 8: Single-node time/speedup results for `kt3d`. Small scenario: $40 \times 60 \times 12$ grid nodes. Large scenario: $120 \times 180 \times 24$ grid nodes. Extra large scenario: $120 \times 180 \times 48$ grid nodes

#Threads	Time[s] (Speedup) *Server* Extra Large 2 nodes	*Server* Extra Large 4 nodes	*Server* Extra Large 8 nodes
1 (optimized)	123.10 (1.98x)	65.49 (3.73x)	33.55 (7.29x)
2 (optimized)	61.21 (3.99x)	31.99 (7.65x)	17.24 (14.19x)
4 (optimized)	31.75 (7.71x)	17.24 (14.19x)	9.80 (24.97x)
8 (optimized)	17.49 (13.99x)	9.99 (24.49x)	6.15 (39.79x)
16 (optimized)	10.28 (23.80x)	6.47 (37.82x)	4.31 (56.78x)

Table 9: Multi-node time/speedup results for `kt3d`

4.3 `sgsim`

The application `sgsim` implements the sequential gaussian simulation algorithm, as described in [1] (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 5. 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 conditional cumulative distribution function (ccdf) must be estimated by simple or ordinary kriging (eqs. (2) and (5)-(6)), 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  Y ← normal_score(V);
2  for isim ∈ {1, ..., N} do
3  |  P ← create_random_path(Ω, τ);
4  |  Y^tmp ← zeros(Y);
5  |  for ixyz ∈ {1, ..., |Ω|} do
6  |  |  index ← P_ixyz;
7  |  |  p ← kriging(index, γ, κ);
8  |  |  Y^tmp_index ← simulate(index, p, τ);
9  |  |  V^tmp ← back_transform(Y^tmp_index);
10 |  |  write(output.txt, V^tmp_index);
11 |  end
12 end
```

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

Algorithm 5: Pseudo-code of `sgsim`, sequential gaussian simulation program (single-thread algorithm)

The small test scenario consists in generating 96 realizations using a regular grid of $40 \times 60 \times 12$ (28800) nodes, simulating a single variable with three variographic structures (one spherical plus two different exponentials). The large and extra large test scenarios consist in 96 and 128 realizations respectively, using a regular grid of $400 \times 600 \times 12$ (2880000) nodes with the same variographic features. In Figure 8 we can see a slice in the plane XY at level $z = 6$ of a simulated realization using the small and large grids. Regarding the spatial search strategy, in the three scenarios the parameter `sstrat` is set to 1, which means that the data are relocated to grid nodes and a spiral search is used, as described in [1] (II.4).

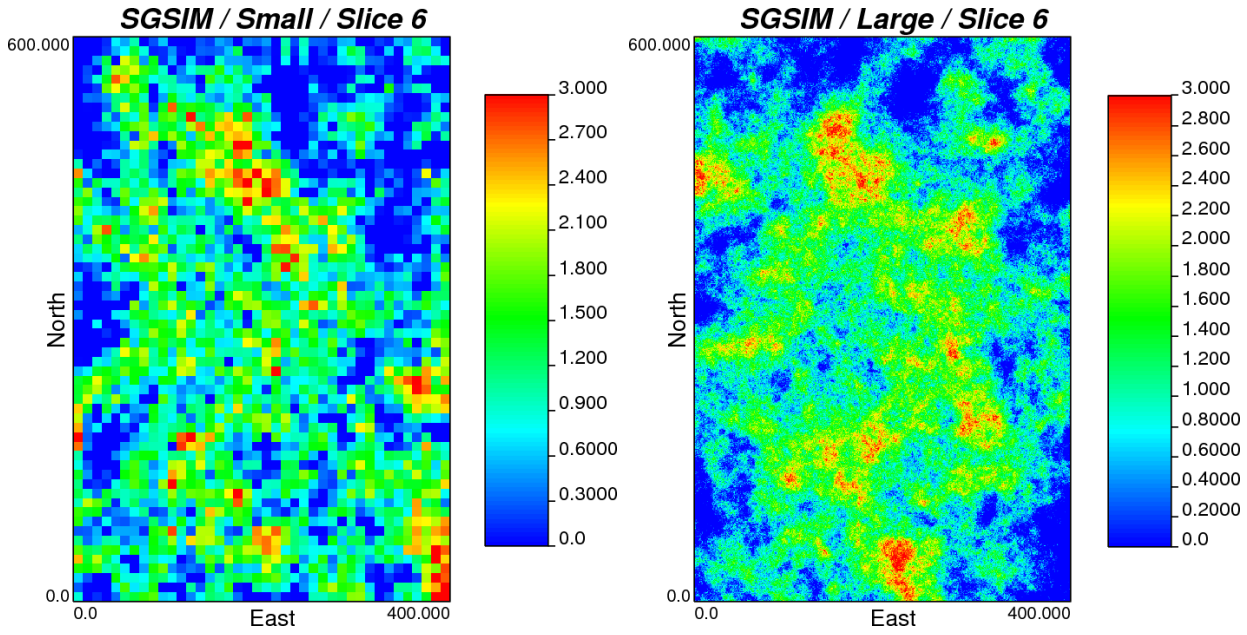

Figure 8: Output of `sgsim` application using a small (left) and large (right) scenarios.

Re-design: Applying the first step of the methodology as in the `gamv` application, no significant acceleration is observed, only a speedup of 1.01x/1.01x/1.02x for the small/large/extra large scenarios is obtained.

Profiling and code optimization: Regarding the second step, the output report of `gprof` can be viewed in Table 10. We can observe that the routines `krige` (implemented as subroutine in `sgsim`, inside the pseudo-routine `kriging` from Algorithm 5), `ksol` (utility contained in `gslib.a` library, launched inside of `krige`) and `srchnd` (utility contained in `gslib.a`) together consist in approximately 89%, 90% and 91% of the total elapsed time for the small, large and extra large scenarios. `krige` assembles the covariance matrix and right-hand side of the kriging linear system, `ksol` solves it using a standard gaussian elimination algorithm without pivot search and `srchnd` search for nearby simulated grid nodes using the spiral search strategy. Although not considered in the three test scenarios, the super block search can be activated by setting `sstrat=0`. This parameter makes the hard data and previously simulated nodes been searched separately (the hard data are searched with the super block strategy and previously simulated nodes are searched with the spiral search). In this case, the average profile in the three scenarios is dominated by the routine `srchsupr`, as described in the `kt3d` application, reaching levels of 27% of the total elapsed time, followed by `cova3`, `sqdist`, `ksol` and `krige`. The optimization of `sgsim` using the parameter `sstrat=0`, which implies the optimization of the routine `srchsupr`, is left as a future work in further versions of the code. In Table 11, the output of `ltrace` is depicted. We can see a large number of library write calls (`_gfortran_st_write`, `_gfortran_st_write_done` and `_gfortran_transfer_real_write`), executed inside the grid node loop. Additionally, there is a large number of library calls to mathematical (`expf` and `log`). The tool `strace` does not report any useful information about possible bottlenecks or overhead sources during execution in all scenarios.

A first optimization is related with code hoisting and dead code removal. A simple inspection in the code allow us to perform this simple optimization, achieving a speedup of 1.03x/1.02/1.03x. Profiles generated with `Oprofile` tool indicate that a large number of branch instructions are executed in `ksol`. For this reason we specialize this routine in order to reduce the number of branches executed. In Figure 9 we can observe the original and specialized `ksol` routine. This optimization achieves a speedup of 1.05x/1.03x/1.05x. In order to reduce the number of write calls, we use a buffer array to store each simulation results ($\mathbf{V}^{tmp}$, Algorithm 5), and at the end of each grid node loop only one write call is executed. In contrast to `kt3d` application, in this case there is no contention by false sharing when each thread stores the simulated values in the buffer, since each thread has its own local buffer array. Additionally, in order to accelerate the write call, we use a binary format in the output file, adding the option `form='UNFORMATTED'` in the `open`

instruction. With this modification, a considerable speedup of 1.49x/1.32x/1.36x is achieved. Two further optimizations are added to accelerate the multi-thread execution. The first is a specialization of the GSLIB utility `sortem`, as described previously for `kt3d` application. The second is a local definition of an array, denoted `ixv`, which is stored in a common block used by the GSLIB utility routine `acorni`. This utility calculates pseudo-random numbers using a seed as input and storing it in the first element of the array `ixv` at the beginning of the execution [18]. In the single-thread version, these optimizations deliver a speedup of 1.53x/1.32x/1.38x, matching the numerical results of the original `sgsim` application. A summary of all the optimizations performed can be viewed in Table 12.

% time (`gprof`, small)	seconds	calls	name
36.05	2.43	2558429	`krige`
35.61	2.40	2558009	`ksol`
17.21	1.16	2558592	`srchnd`
6.23	0.42	5324392	`acorni`
2.37	0.16	2558592	`gauinv`
1.63	0.11	1	`MAIN`
0.90	-	-	other routines

% time (`gprof`, large)	seconds	calls	name
34.33	319.07	276255462	`krige`
29.24	271.73	276258240	`srchnd`
26.58	247.02	276250404	`ksol`
3.12	29.00	97	`sortem`
3.08	28.61	552739240	`acorni`
2.43	22.56	1	`MAIN`
1.22	-	-	other routines

% time (`gprof`, extra large)	seconds	calls	name
35.36	453.53	368340629	`krige`
29.37	376.79	368344320	`srchnd`
25.83	331.36	368333882	`ksol`
3.01	38.62	129	`sortem`
3.01	38.56	736985320	`acorni`
2.31	29.65	1	`MAIN`
1.11	-	-	other routines

Table 10: Output of `gprof` tool applied to `sgsim` application (small, large and extra large scenario) running in *Desktop* system. Timing column ("seconds") may not show accurate measurements and must be taken only as an approximation reference.

% time (`ltrace`, small)	seconds	calls	name
24.02	21.36	374902	`_gfortran_st_write_done`
20.05	17.83	374903	`_gfortran_st_write`
18.60	16.54	374868	`_gfortran_transfer_real_write`
17.30	15.39	373128	`log`
17.11	15.21	367032	`expf`
2.76	2.45	28552	`_gfortran_transfer_real`
0.16	-	-	other routines

% time (`ltrace`, large)	seconds	calls	name
46.42	128.82	2877690	`log`
19.57	54.31	1056004	`_gfortran_transfer_real_write`
17.87	49.58	1056019	`_gfortran_st_write_done`
15.93	44.19	1056020	`_gfortran_st_write`
0.17	0.46	28538	`_gfortran_transfer_real`
0.04	-	-	other routines

% time (`ltrace`, extra large)	seconds	calls	name
37.48	108.51	2877690	`log`
24.32	70.41	1333265	`_gfortran_st_write`
21.06	60.97	1333249	`_gfortran_transfer_real_write`
16.92	48.97	1333265	`_gfortran_st_write_done`
0.16	0.46	28538	`_gfortran_transfer_real`
0.06	-	-	other routines

Table 11: Output of `ltrace` tool applied to `sgsim` application (small, large and extra large scenario) running in *Desktop* system. Timing column ("seconds") may not show accurate measurements and must be taken only as an approximation reference. In large and extra large scenarios, it shows the measurements running the application approximately by two minutes

```
 1   subroutine ksol(nright,neq,nsb,a,r,s,ising)
 2      ...
 3      nm    = nsb*neq
 4      ...
 5      do k=1,m1
 6         ...
 7         do iv=1,nright
 8            nm1=nm*(iv-1)
 9            ii=kk+nn*(iv-1)
10            ...
11         end do
12      end do
13      ijm=ij-nn*(nright-1)
14      ...
15      do iv=1,nright
16         nm1=nm*(iv-1)
17         ij=ijm+nn*(iv-1)
18         ...
19      end do
20      return
21   end
```

```
 1   subroutine ksol01(nright,neq,nsb,a,r,s,ising)
 2      ...
 3      nm    = neq
 4      ...
 5      do k=1,m1
 6         ...
 7
 8         nm1=0
 9         ii=kk
10         ...
11
12      end do
13      ijm=ij
14      ...
15
16      nm1=0
17      ij=ijm
18      ...
19
20      return
21   end
```

Figure 9: Original (left) and specialized (right) `ksol` utility. The original `ksol` routine accepts as arguments several right-hand side vectors (`nright`>1 and `nsb`>1). However, the `sgsim` application uses it only with one right-hand side vector (`nright`$=1$ and `nsb`$=1$). We can specialize this routine if each appearance of `nright` and `nsb` is replaced by 1.

Optimization	Small	Large	Extra large
Baseline	1.00x	1.00x	1.00x
Re-design	1.01x	1.01x	1.02x
Code hoisting + remove dead code	1.03x	1.02x	1.03x
Specialization of `ksol`	1.05x	1.03x	1.05x
Buffered binary writing	1.49x	1.32x	1.36x
Specialization of `sortem`	1.51x	1.33x	1.39x
Local arrays of `acorni`	1.53x	1.32x	1.38x

Table 12: Summary of code optimizations in `sgsim` application running in *Desktop* system.

OpenMP parallelization: The next step is to add OpenMP pragmas into the source code. According to [19], there are three levels of parallelization in sequential stochastic simulation algorithms: realization, path and node level. We choose to parallelize at realization level because it introduces minimal modifications in the code, which is one of our main objectives. Although the node level is the finest-grained scheme, we leave this task as a future work to explore with other technologies (accelerators and co-processors), nested-parallelism with threads or task-based parallelism [20]. The `parallel` pragma is added in the loop of line 2 from Algorithm 5. The default static block scheduling is used for loop splitting, since each thread will perform the same number of iterations, $iters(N,T) = \lceil (N-T+1)/T \rceil$, being N the total number of simulations and T the number of threads.

MPI parallelization: The distribution of load using MPI is performed in a similar way, splitting the loop of line 2 from Algorithm 5, where each MPI task will perform the same number of iterations $iters(N,P)$, with P the total number of MPI tasks. In this way, using multiple threads per task, each thread will perform $iters(iters(N,P),T)$ iterations. As in the previous applications, the parallelization of the optimized application still delivers similar results to the original GSLIB application, allowing some tolerance up to the 5th or 6th decimal number (non-commutativity of floating-point operations), and also allowing different random seeds for each set of realizations obtained by each thread in each node.

The main steps of the multi-thread multi-node algorithm are depicted in Algorithm 6. The random seed τ can be modified differently in each thread and node, using a function f depending on the thread and task identifier (line 6). In terms of reproducibility of results, the function f can be defined using a lookup table, where the values defined for each thread and node correspond the historical values of `ixv(1)` extracted from the utility `acorni` just before the time where the first pseudo-random number was generated in the corresponding realization of the target reproducible set of simulations. Additionally, the rest of historical values of the array `ixv`, namely `ixv(2)` to `ixv(KORDEI+1)`, must be loaded in the current `ixv` array in order to maintain the reproducibility of pseudo-random numbers. In this work, the reproducibility features were not implemented, leaving this topic for future versions of the code. After the random seed generation step, each thread opens a binary local file `output-`$itag$`.bin` where it will store the simulated back-transformed values $\mathbf{V}^{tmp}$ (line 8). The next steps are the same as in the single-thread version, with the only difference that each thread in each node uses its own random seed τ_{itag} and the result is saved to a file after all nodes have been simulated (line 19).

Results: In table 13 we can see the maximum achieved speedup for the small, large and extra large scenarios with 16 CPU-cores in the *Server* system, 15.51x, 15.01x and 14.17x respectively. Using the *Desktop* system, the maximum achieved speedups with 4 CPU-cores are 5.66x, 4.70x and 5.02x respectively. Using the optimized sequential time as baseline, the maximum achieved speedups are 10.13x, 11.16x and 11.38x for the small, large and extra large scenarios in the *Server* system and 3.70x, 3.55x and 3.63x respectively in the *Desktop* system. In terms of efficiency, these values represent a 63%, 69% and 71% in the *Server* system and 92%, 88% and 90% in *Desktop*, for the small, large and extra large scenarios respectively.

Multi-node results are shown in table 14, with a maximum achieved speedup of 100.57x for the extra large scenario with 16 CPU-cores per node and 8 compute nodes. Using the optimized sequential time as baseline, the maximum achieved speedup is 80.76x using the same distributed CPU-cores. This speedup accomplishes a 63% of efficiency using 128 CPU-cores.Using 64 and 32 distributed CPU-cores and the optimized baseline, the average efficiency obtained is 72% and 80% respectively. The efficiencies obtained using 64 CPU-cores are 65% with 4 nodes and 16 threads each, and 79% with 8 nodes and 8 threads each. Using 32 CPU-cores, the efficiencies are 76% with 2 nodes and 16 threads each, 82% with 4 nodes and 8 threads each, and 83% with 8 nodes, 4 threads each.

As in `kt3d`, the decay in efficiency observed in these results is related with overhead added by bus contention when the threads are writing to disk. No additional overhead is added since no contention is generated by false sharing, thread synchronization or MPI message sharing.

Input: Same inputs as Algorithm 5; P: number of MPI tasks; T: number of execution threads;

```
1  itask ← MPI_COMM_RANK(...);
2  N_itask ← ⌈(N−P+1)/P⌉;
3  Y ← normal_score(V);
   /* omp parallel default(firstprivate) shared(Y, Ω) */
4  foreach ithread ∈ {1, ..., T} do
5      itag ← ithread + itask × T;
6      τ_itag ← τ * f(ithread, itask);
7      N_itag ← ⌈(N_itask−T+1)/T⌉;
8      open(output-itag.bin);
9      V^tmp ←zeros(|Ω|);
10     for isim ∈ {1, ..., N_itag} do
11         P ← create_random_path(Ω, τ_itag);
12         Y^tmp ← zeros(Y);
13         for ixyz ∈ {1, ..., |Ω|} do
14             index ← P_ixyz;
15             p ← kriging(index, γ, κ);
16             Y^tmp_index ← simulate(index, p, τ_itag);
17             V^tmp_index ← back_transform(Y^tmp_index);
18         end
19         write(output-itag.bin, V^tmp);
20     end
21 end
```

Output: N stochastic simulations stored in binary files `output-1.bin`,...,`output-`$T \times P$`.bin`

Algorithm 6: Pseudo-code of `sgsim`, sequential gaussian simulation program (multi-thread multi-node algorithm)

#Threads	Time[s] (Speedup) *Server* Small 1 node	*Server* Large 1 node	*Server* Extra large 1 node	*Desktop* Small 1 nodee	*Desktop* Large 1 node	*Desktop* Extra large 1 node
1 (base gslib)	10.24 (1.0x)	1177.52 (1.0x)	1528.76 (1.0x)	10.25 (1.0x)	1268.98 (1.0x)	1807.52 (1.0x)
1 (optimized)	6.69 (1.53x)	875.17 (1.34x)	1227.67 (1.24x)	6.70 (1.53x)	959.73 (1.32x)	1307.87 (1.38x)
2 (optimized)	3.49 (2.93x)	456.48 (2.57x)	640.38 (2.38x)	3.51 (2.92x)	511.9 (2.47x)	685.41 (2.63x)
4 (optimized)	2.08 (4.92x)	245.01 (4.80x)	351.47 (4.34x)	1.81 (5.66x)	269.95 (4.70x)	359.78 (5.02x)
8 (optimized)	0.92 (11.13x)	134.5 (8.75x)	178.18 (8.57x)	-	-	-
16 (optimized)	0.66 (15.51x)	78.42 (15.01x)	107.82 (14.17x)	-	-	-

Table 13: Single-node time/speedup results for `sgsim`. Small scenario: $40\times60\times12$ grid nodes, 96 realizations. Large scenario: $400 \times 600 \times 12$ grid nodes, 96 realizations. Extra large scenario: $400 \times 600 \times 12$ grid nodes, 128 realizations

#Threads	Time[s] (Speedup) *Server* Extra Large 2 nodes	*Server* Extra Large 4 nodes	*Server* Extra Large 8 nodes
1 (optimized)	611.77 (2.49x)	309.9 (4.93x)	156.54 (9.76x)
2 (optimized)	309.10 (4.94x)	161.48 (9.46x)	83.05 (18.40x)
4 (optimized)	170.65 (8.95x)	88.55 (17.26x)	45.87 (33.32x)
8 (optimized)	90.73 (16.84x)	46.57 (32.82x)	24.15 (63.30x)
16 (optimized)	50.09 (30.52x)	29.24 (52.28x)	15.20 (100.57x)

Table 14: Multi-node time/speedup results for `sgsim`

4.4 `sisim`

The application `sisim` implements the sequential indicator simulation algorithm, as described in [1] (V.3.1). Its main steps can be viewed in Algorithm 7. The main steps are analogous to the sequential gaussian simulation algorithm, implemented by the `sgsim` application. The differences with Algorithm 5 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). Indicator kriging (IK) is a variant of the traditional simple and ordinary kriging from eqs. (2) and (5)-(6), which uses a nonlinear transformation of the data $Z(\mathbf{u})$, based in the indicator function $I(Z(\mathbf{u})) = 1$ if $Z(\mathbf{u}) \leq z$ and $I(Z(\mathbf{u})) = 0$ if $Z(\mathbf{u}) > z$. The algorithm provides a least-squares estimate on the conditional cumulative density function (ccdf) at cutoff z_k, defined as

$$[i(\mathbf{u}; z_k)]^* = \mathbb{E}\{I(\mathbf{u}; z_k|(n))\}^* = \mathbb{P}\{Z(\mathbf{u}) \leq z_k|(n)\} \tag{7}$$

where (n) is the conditional information available in the neighbourhood of location $\mathbf{u}$. The Simple Kriging estimator of the indicator transform $i(\mathbf{u}; z)$ can be written as

$$[i(\mathbf{u}; z_k)]^*_{SK} = \sum_{\alpha=1}^{n} \lambda_\alpha^{(ISK)}(\mathbf{u}; z) i(\mathbf{u}_\alpha; z) + \left[1 - \sum_{\alpha=1}^{n} \lambda_\alpha^{(ISK)}(\mathbf{u}; z)\right] \mathbb{P}\{Z(\mathbf{u}) \leq z\} \tag{8}$$

where $\lambda_\alpha^{(ISK)}(\mathbf{u}; z)$ are the SK weights corresponding to cutoff z. These weights are calculated as the solution of the following linear system:

$$\sum_{\beta=1}^{n} \lambda_\beta^{(ISK)}(\mathbf{u}; z) C_I(\mathbf{u}_\beta - \mathbf{u}_\alpha; z) = C_I(\mathbf{u} - \mathbf{u}_\alpha; z),\ \alpha = 1, \ldots, n \tag{9}$$

with $C_I(\mathbf{h}; z) = C\{I(\mathbf{u}; z), I(\mathbf{u}+\mathbf{h}; z)\}$ the indicator covariance at cutoff z. Ordinary Kriging estimator aims at re-estimating locally the prior cumulative density function $\mathbb{P}\{Z(\mathbf{u}) \leq z\}$.

The small test scenario consists in generating 96 realizations using a regular grid of $40 \times 60 \times 12$ (28800) nodes in a 3D volume of $400 \times 600 \times 120[\mathrm{m}^3]$, simulating three categories (lithologies) with one variographic

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: local interpolation parameters; 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  |  |  for icut ∈ {1, ..., C} do
7  |  |  |  p_icut ← indicator_kriging(index, γ_icut, κ);
8  |  |  end
9  |  |  V^tmp_index ← simulate(index, p_1, ..., p_C, τ);
10 |  |  write(output.txt, V^tmp_index);
11 |  end
12 end
```

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

Algorithm 7: Pseudo-code of `sisim`, sequential indicator simulation program (single-thread algorithm)

structure each (spherical). The large and extra large test scenarios consist in generating 16 and 128 realizations respectively, using a regular grid of $400 \times 600 \times 12$ (2880000) nodes with the same 3D volume and the same variographic features. In Figure 10 we can see a slice in the plane XY at level $z = 6$ of a simulated realization using the small and large grid. Regarding the spatial search strategy, in the three scenarios the parameter `sstrat` is set to 1, the same parameter used by the `sgsim` application.

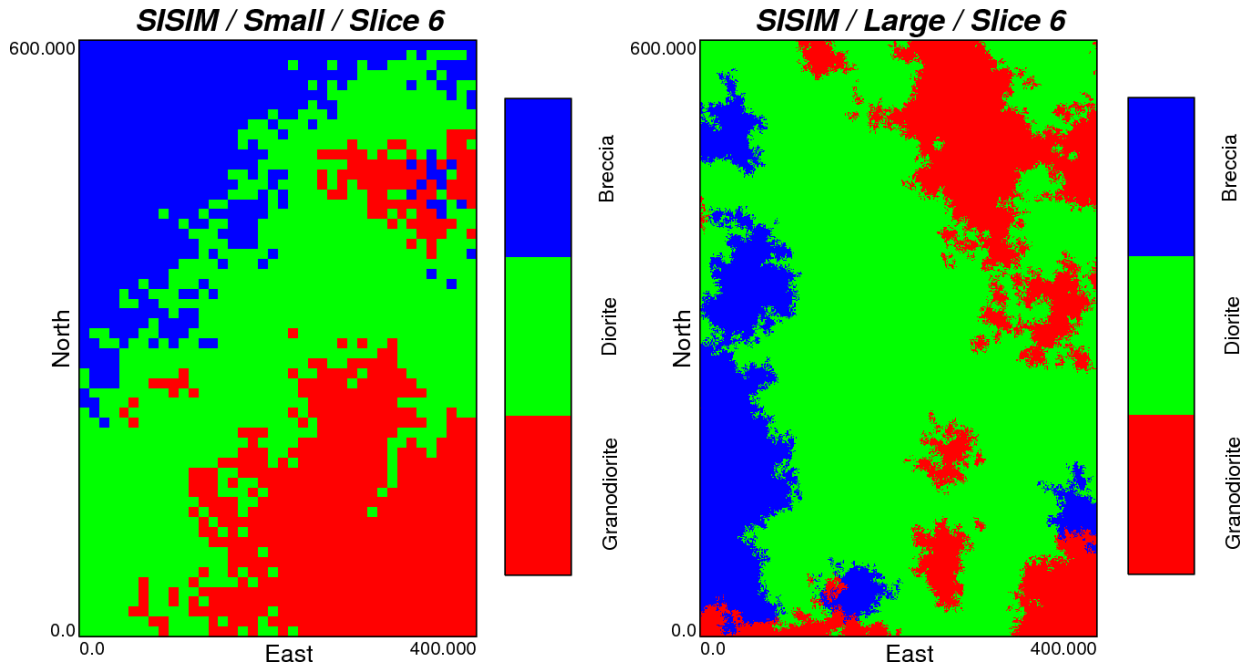

Figure 10: Output of `sisim` application using a small (left) and large (right) scenarios.

Re-design: Applying the first step of the methodology as in the `gamv` application, we can achieve a speedup of 1.05x/1.03x/1.01x for the small/large/extra large scenarios.

Profiling and code optimization: In the second step, the `gprof` output report can be viewed in Table 15. We can see in this report that the routines `krige` and `ksol` together consist in approximately 93%, 89% and 88% of the total elapsed time in the small, large and extra large scenarios. As in the `sgsim` application, the three test scenarios can activate the super block strategy for hard data by setting `sstrat=0`. By changing this parameter, the average profile in the three scenarios is dominated by the routines `ksol` and `cova3`, together with more than 50% of the total elapsed time, followed by `sqdist` and `krige`, all of them with comparatively balanced weights. The effect of the super block search routine `srchsupr` is not dominant, reaching levels between 3% and 7% in the three scenarios. In order to optimize `sisim` using the parameter `sstrat=0`, a specialization of the routine `cova3` can be applied, as shown for the `kt3d` application. However, the analysis and performance measurement of this task is left as a future work in further versions of the code.

Additionally, in Table 16 we can see a large number of library write calls (`_gfortran_st_write` and `_gfortran_transfer_real_write`) performed inside of the grid node loop. The tool `strace` does not report any useful information about possible bottlenecks or overhead sources during execution in all scenarios.

% time (`gprof`, small)	seconds	calls	name
56.96	38.05	7673982	`ksol`
36.47	24.36	7675245	`krige`
4.25	2.84	2558592	`srchnd`
1.26	0.84	1	`MAIN`
1.06	-	-	other routines

% time (`gprof`, large)	seconds	calls	name
55.24	711.83	138125112	`ksol`
33.73	434.68	138127767	`krige`
8.73	112.47	46043040	`srchnd`
1.20	15.50	1	`MAIN`
1.10	-	-	other routines

% time (`gprof`, extra large)	seconds	calls	name
51.69	4896.94	1105001571	`ksol`
36.64	3471.48	1105022088	`krige`
9.12	863.74	368344320	`srchnd`
1.33	126.40	1	`MAIN`
1.211	-	-	other routines

Table 15: Output of `gprof` tool applied to `sisim` application (small, large and extra large scenarios) running in *Desktop* system. Timing column ("seconds") may not show accurate measurements and must be taken only as an approximation reference.

% time (`ltrace`, small)	seconds	calls	name
36.97	114.90	1931004	`_gfortran_transfer_real_write`
32.15	99.90	1931126	`_gfortran_st_write`
30.02	93.28	1931125	`_gfortran_st_write_done`
0.86	-	-	other routines

% time (`ltrace`, large)	seconds	calls	name
42.98	102.56	2017680	`_gfortran_transfer_real_write`
29.60	70.64	2017669	`_gfortran_st_write`
26.28	62.71	2017668	`_gfortran_st_write_done`
1.14	-	-	other routines

% time (`ltrace`, extra large)	seconds	calls	name
35.42	96.16	2024550	`_gfortran_st_write`
33.26	90.31	2024562	`_gfortran_transfer_real_write`
31.07	84.36	2024550	`_gfortran_st_write_done`
0.25	-	-	other routines

Table 16: Output of `ltrace` tool applied to `sisim` application (small, large and extra large scenarios) running in *Desktop* system. Timing column ("seconds") may not show accurate measurements and must be taken only as an approximation reference. In this case it shows the measurements running the application approximately by two minutes

Further information can be obtained with `Oprofile`. After exploring different profiles obtained with this tool and studying the source code, we inferred that the two main optimizations to be performed in this application are related with branch instructions reductions of the routines `krige` and `ksol`, and library call reduction using buffered and unformatted writes to disk. Inspecting the source code, we can see several loop-invariant expressions (if-then-else type) that can be hoisted out of the internal loops of both routines. If we modify first the `krige` routine, the optimized code can achieve a speedup of 1.09x/1.04x/1.03x. After that, the routine `ksol` is specialized to use fixed arguments (in the original code those arguments never change), reducing the number of branch instructions executed, allowing us to achieve a speedup of 1.11x/1.06x/1.05x. In order to reduce the number of library calls, we store each nodal value in a buffer array, and after all nodes have been simulated successfully, just one `write` call is performed, writing the buffer array in an unformatted file previously opened with the option `form='UNFORMATTED'` (same optimization applied before in `kt3d` and `sgsim`). The resulting speedup on the *Desktop* system is 1.16x/1.12x/1.11x for each scenario, with numerical results matching the original `sisim` application.

A summary of the speedups obtained can be viewed in Table 17.

Optimization	Small	Large	Extra large
Baseline	1.00x	1.00x	1.00x
Re-design	1.05x	1.03x	1.01x
Code hoisting + remove dead code	1.09x	1.04x	1.03x
Specialization of `ksol`	1.11x	1.06x	1.05x
Buffered binary writing	1.16x	1.12x	1.11x

Table 17: Summary of code optimizations in `sisim` application running in *Desktop* system.

OpenMP and MPI parallelization: Following the same strategy defined for `sgsim`, the proposed parallel implementation is depicted in Algorithm 8. A single `parallel` pragma was added before the simulation loop. As for the multi-thread multi-node sequential gaussian simulation, each thread in each node initializes its own random seed τ_{itag} and writes the simulated values in a local binary file.

Input: Same inputs as Algorithm 7; P: number of MPI tasks; T: number of execution threads;

```
1  itask ← MPI_COMM_RANK(...);
2  N_itask ← ⌈(N−P+1)/P⌉;
   /* omp parallel default(firstprivate) shared(V, Ω) */
3  foreach ithread ∈ {1,...,T} do
4      τ_itag ← τ * f(ithread, itask);
5      N_itag ← ⌈(N_itask−T+1)/T⌉;
6      open(output-itag.bin);
7      for isim ∈ {1,...,N_itag} do
8          P ← create_random_path(Ω, τ_itag);
9          V^tmp ← zeros(V);
10         for ixyz ∈ {1,...,|Ω|} do
11             index ← P_ixyz;
12             for icut ∈ {1,...,C} do
13                 p_icut ← indicator_kriging(index, γ_icut, κ);
14             end
15             V^tmp_index ← simulate(index, p_1,...,p_C, τ_itag);
16         end
17         write(output-itag.bin, V^tmp);
18     end
19 end
```

Output: N stochastic simulations stored in binary files `output-1.bin`,...,`output-`$T \times P$`.bin`

Algorithm 8: Pseudo-code of `sisim`, sequential indicator simulation program (multi-thread multi-node algorithm)

Results: Analyzing the single-node results of table 18, the maximum achieved speedup is 15.13x for the small scenario, 14.34x for the large scenario and 15.07x for the extra large scenario using 16 CPU-cores in the *Server* system. Using the *Desktop* system, the maximum achieved speedups with 4 CPU-cores are 4.28x, 4.07x and 4.04x respectively. Using the optimized sequential time as baseline, the maximum achieved speedups are 13.16x, 13.17x and 13.81x for the small, large and extra large scenarios in the *Server* system and 3.66x, 3.60x and 3.62x respectively in the *Desktop* system. In terms of efficiency, these values represent 82%, 82% and 86% in the *Server* system and 91%, 90% and 90% in *Desktop*, for the small, large and extra large scenarios respectively.

Multi-node results are shown in table 19, with a maximum achieved speedup of 109.40x for the extra large scenario with 16 CPU-cores per node and 8 compute nodes. Using the optimized sequential time as baseline, the maximum achieved speedup is 100.29x using the same distributed CPU-cores. This speedup accomplishes a 78% of efficiency using 128 CPU-cores.Using 64 and 32 distributed CPU-cores and the optimized baseline, the average efficiency obtained is 85% and 89% respectively. The efficiencies obtained using 64 CPU-cores are 80% with 4 nodes and 16 threads each, and 89% with 8 nodes and 8 threads each. Using 32 CPU-cores, the efficiencies are 83% with 2 nodes and 16 threads each, 91% with 4 nodes and 8 threads each, and 95% with 8 nodes, 4 threads each.

Although this application is similar to `sgsim`, the main difference is the amount of computations performed. In `sgsim`, a small kriging system is solved for each node in the grid. In `sisim`, C kriging systems are solved for each node in the grid (in these examples $C = 3$). For this reason, the effect of contention due to disk writing has less impact in the parallel efficiency obtained.

#Threads	Time[s] (Speedup) *Server* Small 1 node	*Server* Large 1 node	*Server* Extra large 1 node	*Desktop* Small 1 node	*Desktop* Large 1 node	*Desktop* Extra large 1 node
1 (base gslib)	74.46 (1.0x)	1378.92 (1.0x)	11060 (1.0x)	73.68 (1.0x)	1369.97 (1.0x)	10995.0 (1.0x)
1 (optimized)	64.77 (1.14x)	1266.31 (1.08x)	10130 (1.09x)	63.14 (1.16x)	1213.48 (1.12x)	9854.0 (1.11x)
2 (optimized)	33.93 (2.19x)	641.79 (2.14x)	5100 (2.16x)	33.16 (2.22x)	649.46 (2.11x)	5235.0 (2.10x)
4 (optimized)	16.96 (4.39x)	327.95 (4.20x)	2597 (4.25x)	17.21 (4.28x)	336.76 (4.07x)	2716.72 (4.04x)
8 (optimized)	8.89 (8.37x)	173.03 (7.96x)	1358 (8.14x)	-	-	-
16 (optimized)	4.92 (15.13x)	96.15 (14.34x)	733 (15.07x)	-	-	-

Table 18: Time/Speedup results for `sisim`. Small scenario: $40 \times 60 \times 12$ grid nodes, 96 realizations. Large scenario: $400 \times 600 \times 12$ grid nodes, 16 realizations. Extra large scenario: $400 \times 600 \times 12$ grid nodes, 128 realizations

#Threads	Time[s] (Speedup) *Server* Extra Large 2 nodes	*Server* Extra Large 4 nodes	*Server* Extra Large 8 nodes
1 (optimized)	5242 (2.10x)	2560 (4.31x)	1273 (8.68x)
2 (optimized)	2556 (4.32x)	1288 (8.58x)	640 (17.27x)
4 (optimized)	1308 (8.45x)	653 (16.91x)	332 (33.29x)
8 (optimized)	697 (15.86x)	345 (31.98x)	176 (62.80x)
16 (optimized)	377 (29.33x)	196 (56.42x)	101 (109.40x)

Table 19: Multi-node time/speedup results for `sisim`

5 Conclusions and future work

We have shown a methodology to accelerate GSLIB applications and utilities based on code optimizations and hybrid parallel programming using multi-core and multi-node execution with OpenMP directives and MPI task distribution. The methodology was tested in four well-known GSLIB applications: `gamv`, `kt3d`, `sgsim` and `sisim`. All tests were performed in Linux-based systems. However, no additional external libraries or intrinsic operating system routines were used, so the code could be compiled and tested in other distributed systems (Windows and Mac) without further modifications.

Hardware independent code optimizations are applied to the four applications, in order to maintain the basic portability of the original code. The most critical optimizations are related to reductions of branch instruction execution, avoidance of miss-predicted branch instructions, specialization of generic routines and buffering I/O calls. OpenMP parallel loops and regions, as well as MPI instructions, are added to the optimized codes, with minimal modifications, delivering reasonable speedup results compared to the performance of original GSLIB applications.

The variogram application `gamv` reaches a speedup of 69.56x, 60.99x and 37.46x with 16 non-distributed CPU-cores in the small, large and extra large test scenarios, and 232.07x with 128 distributed CPU-cores in the extra large scenario. This high speedup value was obtained after applying a small code optimization which reduces the number of miss-predicted branches during execution. Additionally, a static optimal chunk-size strategy was set for the run-time scheduling of parallelization. The kriging application `kt3d` reaches a speedup of 13.87x, 14.48x and 13.88x with 16 non-distributed CPU-cores in the small, large and extra large test scenarios, and 56.78x with 128 distributed CPU-cores in the extra large scenario. Code optimizations were difficult to apply in this application, and some features of the single-thread version have to be eliminated in the multi-thread version, in order to accelerate the parallel execution. The default schedule was the best alternative for the run-time scheduling of parallelization. Both sequential simulation applications, `sgsim` and `sisim`, reach speedup levels of 15.51x and 15.13x in the small scenario, 15.01x and 14.34x in the large scenario, and 14.17x and 15.07x in the extra large scenario using a single node of execution. In the extra large scenario using 128 distributed CPU-cores, both applications reach a speedup of 100.57x and 109.40x respectively. The code optimizations that showed successful results were the specialization of routines and buffering I/O calls. As shown in Algorithms 6 and 8, a parallel region was defined for both applications

without using a parallel loop to distribute the workload, reducing the thread synchronization overhead.

Using the current multi-thread multi-node versions of the applications, we can conclude that parallel `sisim` is the best candidate to scale and keep the efficiency of the parallel execution using a large number of CPU-cores. Results showed that using a small and large dataset/grid, `sisim` delivers a similar efficiency (82% both) using 16 CPU-cores in a single node. Using the extra large dataset/grid, it improves its efficiency (86%) using 16 CPU-cores in a single node, without degrading it considerably using up to 128 distributed CPU-cores (89%, 85% and 78% with 32, 64 and 128 CPU-cores). `kt3d` showed an increment in efficiency from the small to large scenarios (77% to 85%) using 16 CPU-cores in a single node. However, with the extra large scenario a considerable decay is observed (80%). This behaviour is accentuated in the multi-node tests, reaching even lower levels of average efficiency up to 128 distributed CPU-cores (71%, 56% and 41% with 32, 64 and 128 CPU-cores). `gamv` showed an increment in efficiency proportional to the size of dataset, from small to extra large scenarios (68%, 85% and 86% respectively) using 16 CPU-cores in a single node. In the multi-node tests, the average efficiency decreases steadily using up to 128 distributed CPU-cores (86%, 79% and 66% with 32, 64 and 128 CPU-cores), without reaching lower levels as in the `kt3d` application. `sgsim` showed the lowest efficiency percentages in the small and large scenarios (63% and 69% respectively) using 16 CPU-cores in a single node. However, in the extra large scenario the efficiency improved using 16 CPU-cores in a single node (71%) and slightly decreased using up to 128 distributed CPU-cores (80%, 72% and 63% with 32, 64 and 128 CPU-cores).

According to the single-node results, the efficiency increases as the size of the input dataset and/or estimation/simulation grid gets larger, which is in concordance with Gustafson's law [21], a counterpart of the well-known Amdahl's law [22]. Gustafson's law says that computations involving arbitrarily large data sets can be efficiently parallelized. If larger scenarios are considered in future versions of the applications, we can expect higher efficiency values. Additionally, extremely large scenarios may benefit from memory-aware optimizations, such as the improved memory access pattern proposed for the super block strategy `srchsupr` in the context of `kt3d` application.

Our final goal is to apply this methodology to the most important applications and utilities from GSLIB, setting the compilation scripts as general as possible, so other scientists and practitioners could benefit from this effort. The addition of new acceleration technologies, like Graphics Processing Units (GPU) [23] and Many Integrated Cores (MIC) [24], and advanced parallelization techniques, like task-based OpenMP pragmas [25], are being explored using the present work as base.

6 Acknowledgements

The authors thankfully acknowledge the computer resources, technical expertise and assistance provided by the Barcelona Supercomputing Center - Centro Nacional de Supercomputación (Spain) which supports the Marenostrum supercomputer, and the National Laboratory for High Performance Computing (Chile), which supports the Leftraru supercomputer. Additional thanks are owed to industrial supporters of ALGES laboratory, in particular Yamana Gold, as well as the Advanced Mining Technology Center (AMTC) and the whole ALGES team.

7 Source code

The current version of the modified codes can be downloaded from

`https://github.com/operedo/gslib-alges`.

References

[1] C. V. Deutsch and A. G. Journel, *GSLIB: Geostatistical Software Library and User's Guide*. Oxford University Press, 1998.

[2] Statios LLC, "WinGslib Installation and Getting Started Guide." Website, 2001. last checked: 04.08.2015.

[3] N. Remy, A. Boucher, and J. Wu, *Applied Geostatistics with SGeMS: A User's Guide*. Cambridge University Press, 2009.

[4] "High performance geostatistics library, version 0.9.9." `http://hpgl.mit-ufa.com`.

[5] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Pub. Inc., 2001.

[6] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI-The Complete Reference, Volume 1: The MPI Core*. Cambridge, MA, USA: MIT Press, 2nd. (revised) ed., 1998.

[7] J. Straubhaar, A. Walgenwitz, and P. Renard, "Parallel Multiple-Point Statistics Algorithm Based on List and Tree Structures," *Mathematical Geosciences*, vol. 45, no. 2, pp. 131–147, 2013.

[8] 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*, vol. 40, no. 56, pp. 144 – 158, 2014.

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

[10] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "`gprof`: A Call Graph Execution Profiler," *SIGPLAN Not.*, vol. 17, pp. 120–126, June 1982.

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

[12] Oprofile community, "Oprofile, system-wide profiler for Linux systems." Website, 2013. last checked: 04.08.2015.

[13] Barcelona Supercomputing Center, "Paraver/Extrae Performance Analysis Tools, Computer Science Department." Website, 2013. last checked: 04.08.2015.

[14] ZIH - TU Dresden, "VampirTrace 5.14.4 User Manual." Website, 2015. last checked: 04.08.2015.

[15] Intel Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3 (3A, 3B & 3C): System Programming Guide," September 2014.

[16] C. A. Navarro and N. Hitschfeld, "Improving the GPU space of computation under triangular domain problems," *Computing Research Repository, arXiv.org*, vol. abs/1308.1419, 2013.

[17] D. Culler, J. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1st ed., 1998. The Morgan Kaufmann Series in Computer Architecture and Design.

[18] R. S. Wikramaratna, "ACORN: A New Method for Generating Sequences of Uniformly Distributed Pseudo-random Numbers," *J. Comput. Phys.*, vol. 83, pp. 16–31, July 1989.

[19] G. Mariethoz, "A general parallelization strategy for random path based geostatistical simulation methods," *Computers & Geosciences*, vol. 36, no. 7, pp. 953 – 958, 2010.

[20] L. G. Rasera, P. Lopes Machado, and J. F. C. L. Costa, "A conflict-free, path-level parallelization approach for sequential simulation algorithms," *Computers & Geosciences*, vol. 80, pp. 49 – 61, 2015.

[21] J. L. Gustafson, "Reevaluating Amdahl's Law," *Communications of the ACM*, vol. 31, pp. 532–533, 1988.

[22] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[23] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2010.

[24] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2013.

[25] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The Design of OpenMP Tasks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, pp. 404–418, Mar. 2009.