Tuning and Hybrid Parallelization of a Genetic-based Multi-Point Statistics Simulation Code

Oscar Peredo[a,b], Julián M. Ortiz[b,c], José R. Herrero[d], Cristóbal Samaniego[a]

[a] *Barcelona Supercomputing Center (BSC-CNS), Department of Computer Applications in Science and Engineering, Edificio NEXUS I, Campus Nord UPC, Gran Capitán 2-4, 08034 Barcelona, Catalunya, Spain.*
[b] *Advanced Laboratory for Geostatistical Supercomputing, Advanced Mining Technology Center, University of Chile.*
[c] *Department of Mining Engineering, University of Chile. Av. Tupper 2069, Santiago, 837-0451, Chile.*
[d] *Computer Architecture Department, Universitat Politècnica de Catalunya (UPC-BarcelonaTech), Campus Nord, Mòdul C6, Desp. 206, C/ Jordi Girona 1-3, 08034 Barcelona, Catalunya, Spain.*

Abstract

One of the main difficulties using multi-point statistical (MPS) simulation based on annealing techniques or genetic algorithms concerns the excessive amount of time and memory that must be spent in order to achieve convergence. In this work we propose code optimizations and parallelization schemes over a genetic-based MPS code with the aim of speeding up the execution time. The code optimizations involve the reduction of cache misses in the array accesses, avoid branching instructions and increase the locality of the accessed data. The hybrid parallelization scheme involves a fine-grain parallelization of loops using a shared-memory programming model (OpenMP) and a coarse-grain distribution of load among several computational nodes using a distributed-memory programming model (MPI). Convergence, execution time and speed-up results are presented using 2D training images of sizes 100×100×1 and 1000×1000×1 on a distributed-shared memory supercomputing facility.

Keywords: Geostatistics, Stochastic simulation, Multi-point statistics, Code optimization, Parallel computing, Genetic algorithms

1. Multi-point statistics simulation

Numerical modeling with geostatistical techniques aims at characterizing natural phenomena by summarizing and using the spatial correlation of collected data in order to measure the uncertainty at unsampled locations in space. As explained by Deutsch (2002), in simulation techniques, this spatial correlation is imposed into a model commonly constructed on a regular lattice. The models must reproduce the statistical (histogram) and spatial distribution (variogram or other spatial statistics) and their quality is often judged in terms of the reproduction of geological features.

Conventional techniques in geostatistics address the modeling using statistical measures of spatial correlation that quantify the expected dissimilarity (transition to a different category) between locations separated by a given vector distance, in reference to a given attribute, such as the facies, rock type, porosity, grade of an element of interest, among others. This is done using the variogram. Limitations of these techniques have been pointed out in that they only account for two locations at a time when defining the spatial structure (Krishnan and Journel (2003)). Much richer features can be captured by using multi-point statistics (MPS) that consider the simultaneous arrangement of the attribute of interest at several locations, providing the possibility to account for complex features, such as hierarchy between facies, delay effects, superposition or curvilinearity.

MPS simulation aims at generating realizations that reproduce pattern statistics inferred from some training source, usually a training image. For example, in figure 1, left, we can see a training image based on sinuous channels with a simulated realization. These training images are used as a pattern database to generate simulations of the underlying image, as shown in figure 1, right. The simulations use those patterns with the aim that the training and simulated images share the same pattern histogram.

There are several approaches to simulate accounting for MPS. Modifications of conventional methods to impose local directions of continuity using the variogram is a simple approach to impose some of the complex geological

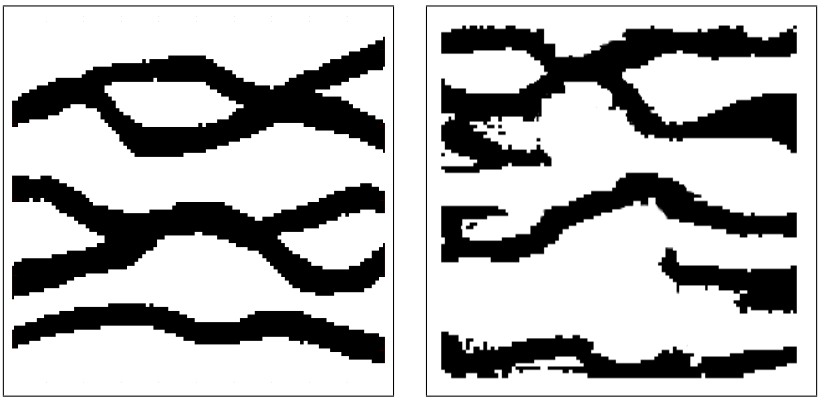

Figure 1: Training image (left) and simulated realization (right)

features (Xu (1996); Zanon (2004)). Object based methods and methods inspired in the genetic rules and physics of the deposition of sediments in different environments also seek to overcome the limitations of conventional categorical simulation techniques, with significant progress (Deutsch and Wang (1996); Tjelmeland (1996); Pyrcz and Strebelle (2008)). Presently, the most popular method is a sequential approach based on Bayes' postulate to infer the conditional distribution from the frequencies of multi-point arrangements obtained from a training image. This method, originally proposed by Guardiano and Srivastava (1993), and later efficiently implemented by Strebelle and Journel (2000), is called single normal equation simulation (snesim) (see also Strebelle (2002)). This method has been the foundation for many variants such as simulating directly full patterns (Arpat and Caers (2007); Eskandari and Srinivasan (2007)) and using filters to approximate the patterns (Zhang et al. (2006)). The use of a Gibbs Sampling algorithm to account directly for patterns has also been proposed (Boisvert et al. (2007); Lyster and Deutsch (2008)). A sequential method using a fixed search pattern and a 'unilateral path' also provides good results (Daly (2005); Daly and Knudby (2007); Parra and Ortiz (2009)). Other approaches available consider the use of neural networks (Caers and Journel (1998); Caers and Ma (2002)), updating conditional distributions with multi-point statistics as auxiliary information (Ortiz (2003); Ortiz and Deutsch (2004); Ortiz and Emery (2005)) or secondary variable (Hong et al. (2008)). Recently, a couple of new approaches focused on patching patterns directly to reduce computing time and impose larger scale structures, have been presented (Rezaee et al. (2013); Faucher et al. (2013)). These methods have a significant potential for practical applications. Alternatively, the problem can be addressed as an optimization one, using simulated annealing (Deutsch (1992)) or genetic algorithms (Peredo and Ortiz (2012)). The genetic approach is still under development, but essentially follows the same stochastic strategy as the annealing scheme. This work focuses on code optimizations and parallelization of a genetic-based sequential code that simulates categorical variables to reproduce multi-point statistics. However, many of the techniques and ideas proposed here can be applied to other codes implementing similar simulation algorithms.

In section 2 we explain the basic ideas about genetic algorithms, parallel architectures and programming models. After that, the main bottlenecks of the genetic-based simulation are detailed in section 3. A brief explanation of the actual implementation is presented in section 4, together with the proposed code optimizations and parallelization schemes, in sections 5 and 6 respectively. Finally, in the last sections we include the results obtained and final conclusions.

2. Genetic algorithms and Parallel computing

Genetic algorithms (GA) were developed in the 1970s with the work of Holland (1975) and in subsequent decades with De Jong (1980) and Goldberg (1989). Initially used to find good feasible solutions for combinatorial optimization problems, today they are used in various industrial applications, and recent advances in parallel computing have allowed their development and continuing expansion.

In the canonical approach of GA, typically there is an initial population of individuals, where each individual is

represented by a string of bits, as $indiv_k$ = 000110101, and a fitness function $fitness(indiv_k)$ which represents the performance of each individual. The fitness function, or objective function, is the objective that must be minimized through the generations over all the individuals. A termination criteria must be defined in order to achieve the desired level of decrement in the fitness function. The main steps and operations performed in a canonical GA can be viewed in algorithm 1. *Selection* and *restart* are operations performed over the entire population. *Selection* extracts the best individuals and *restart* modifies part or the entire population in order to jump from local optimal values. *Crossover* and *mutation* are operations performed over particular individuals of the population. *Crossover* mixes the bits of two individuals according to a predefined set of cut points and *mutation* modifies specific random bits from one individual. Other operators can be found in the mentioned literature.

Algorithm 1 Canonical genetic algorithm

```
INPUT: N individuals (population)
Evaluate a fitness function fitness in each individual
while termination criteria is not achieved do
    {Breeding a new generation}
    Sort the individuals by their fitness function value
    if no improvement is measured in the population then
        Restart: select some individuals and restart their bits
        Sort the individuals by their fitness function value
    end if
    Selection: select the best individuals based on their fitness function
    Crossover: breed new individuals crossing bits of individuals from the selection
    Mutation: breed new individuals mutating some bits of individuals from the selection
    Replace old individuals with new ones
    Evaluate a fitness function fitness in each individual
end while
OUTPUT: best individual in the population
```

In parallel computing architectures, as described by Culler et al. (1998), the two main models are distributed-memory and shared-memory, with their respective best known programming models MPI (Snir et al., 1998) and OpenMP (Chandra et al., 2001). In the first model, each processor has its own private memory and the data interchanged between processors travels through a network in chunks of messages. The speed of this communication depends on the speed of the interconnection network. In the second model, each processor has access to a common memory through data coherence and data consistency methods.

In order to use efficiently all the resources of parallel architectures, we need to explore algorithms that can exploit the parallelism and be able to adapt to future trends.

Genetic algorithms receive the classification of *embarrassingly parallel* technique to solve problems. This classification comes from the fact that separating the workload of the problem into several parallel tasks is trivial. This property motivates its investigation and application in the field of geostatistics, and particularly in MPS simulation.

3. Bottlenecks of genetic-based MPS simulation

We can see in algorithm 1 that the evaluation of the function $fitness$ is performed $\#generations \times \#individuals$, and strong evidence indicates that this function is the most time consuming routine (a profiling using the `gprof` tool, Graham et al. (2004), tells us that for sufficiently large training images, more than 96% of the execution time is spent in this routine). Its calculation is based on an object called *template*. A *template* T consists in a set of coordinates that defines cell positions. Another interpretation is that a template is basically a pattern of memory accesses. $Patterns(T)$ represents all the possible patterns that can be generated from a template T given k possible categories. If a template is defined as T = {(1,1), (2,1), (1,2), (2,2)} (4 nodes) and the number of categories is $k = 2$, the pattern database $Patterns(T)$ will have 2^4 elements. Complex geometries can be used to define the template and its corresponding pattern database, for example, in figure 2 we have a template defined as T = { (1,1), (5,1), (9,1), (4,4), (5,4), (6,4),

(1,5), (4,5), (5,5), (6,5), (9,5), (4,6), (5,6), (6,6), (1,9), (5,9), (9,9) }. This template is disconnected and its memory-access pattern is very irregular. This irregularity induces a slowdown in the overall performance when we need to traverse all its nodes. In contrast, the template of figure 3 has a regular memory-access pattern, if we access to the first element $T(1,1)$, additionally the CPU will bring to the cache memory contiguous elements for free (each CPU has its own cache line size, for example 64 bytes, which means that each cache line can store 16 consecutive integers of 4 bytes). In Fortran, the CPU will bring a column line, in C/C++, a row line (see Hennessy and Patterson (1990) for more details about the CPU memory hierarchy).

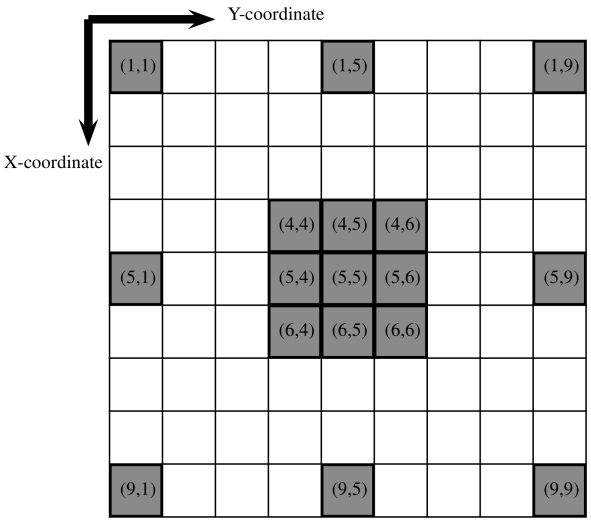

Figure 2: Template of 17 nodes with a complex geometry (irregular memory accesses)

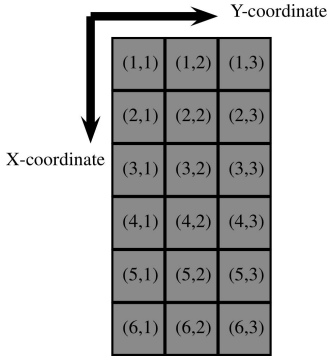

Figure 3: Template of 18 nodes with a simple geometry (regular memory accesses)

Given the pattern database $Pattern(T)$, two main tasks must be performed:

- First we have to count the frequency of appearances of each pattern in the training image and store them in an appropriate structure.
- After that, for each individual in the population, and in each generation, we have to count the frequency of appearances of those patterns and calculate the following equation (other possible equations can be viewed in Peredo and Ortiz (2011)):

$$fitness(indiv_k) = \sum_{p \in Patterns(T)} O_p \left(freq_{TI}(p) - freq_{indiv_k}(p) \right)^2 \quad (1)$$

with O_p a weight factor for each pattern, $freq_{TI}(p)$ and $freq_{indiv_k}(p)$ the number of appearances of pattern p in a training image and individual $indiv_k$ respectively.

In both tasks, we have to handle with patterns located at boundary nodes. A buffer zone of halo nodes, with size equal to $h = \max\{width(T), height(T)\}$, is added at the boundaries of the training image and realizations. For the training image, an extension of the original image is added keeping the geological continuity and statistical properties (alternative, for sufficiently large images, a reduction of size h in each side of the images can be applied, keeping the removed space as buffer zone). For each individual in the population, the buffer zone is filled with random categorical values between 0 and $k-1$.

For the first task, following ideas from Straubhaar et al. (2011), we store the frequencies of the patterns that appear in the training image in a list $\mathcal{L}$. In this list each element is a pair (d, f) where $d = (s_1, \ldots, s_{|T|})$ is the pattern (data event), stored as an array of integers of length $|T|$ with $s_i \in \{0, \ldots, k-1\}$ (k categories), and f is the frequency of appearance, stored as an integer. With this structures defined, the algorithm used in this task corresponds to algorithm 2 using an empty frequency list $\mathcal{L}$.

For the second task, the accounting process is performed using algorithm 2. In this algorithm, we compare the

Algorithm 2 Fitness function calculation: $fitness(indiv_k)$

INPUT: individual $indiv_k$, training image frequency list $\mathcal{L}$, template T
$sum \leftarrow 0$
$\mathcal{L}_{aux} \leftarrow \mathcal{L}$
for each node (i, j) from $indiv_k$ **do**
  Extract pattern located in node (i, j) using template T and store it in an array *localPattern* of length $|T|$
  Search *localPattern* in the list $\mathcal{L}_{aux}$
  if *localPattern* exists in $\mathcal{L}_{aux}$ **then**
    $\mathcal{L}_{aux}[localPattern] \leftarrow \mathcal{L}_{aux}[localPattern] - 1$
  end if
end for
for each pattern p in $\mathcal{L}_{aux}$ **do**
  $sum \leftarrow sum + \mathcal{L}_{aux}[p] * \mathcal{L}_{aux}[p]$
end for
OUTPUT: *sum*

histograms of the individual and the training image. A considerable bottle-neck for this calculation is the access to the list $\mathcal{L}$ which stores the training image histogram, because for each extracted pattern from an individual, a search must be performed over it in order to see if this pattern exists in the histogram of the training image or not. A proposed solution to this problem is to store the elements (d, f) of the list using a lexicographical order in the patterns d. This order allows to search the existence of a pattern in the list using a binary search with an average and worst case performance of order $O(\log_2 n)$ comparisons, with n the length of $\mathcal{L}$.

In the next section we will explain the implementation issues for this two tasks using code examples in Fortran 90 as programming language, in order to see the data structures that are used and the proposed optimizations.

4. Implementation

4.1. Storage of pattern frequencies from training image

The routines involved in the storage and management of the list $\mathcal{L}$ are encapsulated in a Fortran module called `patternOperations`. This module consists of a set of global variables and routines performed over arrays of integers. Part of its code structure is depicted in code 1.

```
module patternOperations
  implicit none
  integer(4) :: npatterns
```

```
  type patternType
     integer(4), pointer :: pattern(:)
     integer(4)          :: frequency
  end type patternType
  type(patternType), pointer :: patternList(:)

contains
  subroutine patternInsertion(...)
  subroutine patternSearch(...)
  subroutine patternComparison(...)
  subroutine printPatternList(...)
end module patternOperations
```

Code 1: Module `patternOperations.f90`

Initially, when we store the pattern histogram of the training image, we use the global array `patternList` to keep track of the different patterns `patternList(i)%pattern(1:tem_nodes)` with their respective frequencies `patternList(i)%frequency`. A first scan to the training image must be performed in order to fill `patternList`. Using the routine `patternInsertion`, each time a new pattern is found, the memory space used by `patternList` is re-allocated (adding one new element with frequency equal to 1) and all the elements previously inserted together with the new pattern are re-ordered according to a lexicographical order. If an existing pattern is found, the frequencies are updated. The lexicographical order is as follows: given two arrays of integers of the same length $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$ we will say that A is greater than B if and only if $\exists k \in \{1, \ldots, n\}$ such that $a_k > b_k$ and $\forall i$ satisfying $i < k$, $a_i = b_i$ holds. For example, given this arrays $A = (0, 0, 0, 1, 0, 0)$ and $B = (0, 0, 0, 0, 0, 0)$, this order will indicate that A is greater than B.

After the filling step, `patternList` will be scanned each time we call the routine `patternSearch`. This routine gives the index position in the global array `patternList` where the searched pattern is located, if there is a match. If there is no match, it returns -1. This routine is implemented through a simple iterative binary search, which uses the routine `patternComparison` to compare patterns.

The routine `patternComparison` basically traverses each pair of pattern's nodes until they have different values and keeps track of which pattern has the greater one, returning 0 if they are equal, 1 if the first array is *larger* than the second one, and -1 otherwise. The parameters of this routine are two arrays with the pattern values and the length of those arrays (must be equal on both).

As we explained before `patternSearch` and `patternComparison` are intensively used by the routine that calculates the fitness function. In the worst case, if the training image pattern list $\mathcal{L}$ has size n and the template has t nodes, a search in the list will perform $t \times O(\log_2 n)$ comparisons. For very large templates, this pattern search is very time consuming and different data structures must be used in order to get a reasonable execution time in the search task.

4.2. Calculate $fitness(indiv_k)$

Following algorithm 2, which explains all the steps involved in the calculation of the fitness function, in this subsection we explain its associated routine presented in code 2. In this routine, the input `indivk` is a 2D integer array with an image loaded (an *individual* in the genetic algorithm terminology). The inputs `tem_nodes`, `tem_rows`, `tem_cols`, `tem_coord_rows` and `tem_coord_cols` correspond to the specific *template* that we are using. For example, in figure 2, `tem_nodes` = 17, `tem_rows` = `tem_cols` = 9 and `tem_coord_rows` and `tem_coord_cols` are arrays that store respectively the row and column coordinates of the nodes in template T.

```
subroutine fitnessFunction(indivk,rows,cols,
       tem_rows, tem_cols, tem_nodes,
       tem_coord_rows,tem_coord_cols,value)
  use patternOperations  !! contains global variables npatterns and patternList

  integer(4), intent(in) :: indivk(rows,cols)
  integer(4), intent(in) :: rows, cols
  integer(4), intent(in) :: tem_nodes
  integer(4), intent(in) :: tem_rows, tem_cols
  integer(4), intent(in) :: tem_coord_rows(tem_nodes)
```

```
  integer(4), intent(in) :: tem_coord_cols(tem_nodes)
  integer(4), intent(out):: value
  integer(4)             :: ii,irow,icol,inode,pos
  integer(4)             :: freq_aux(npatterns)
  integer(4)             :: localPattern(tem_nodes)

  do ii=1,npatterns
     freq_aux(ii)=patternList(ii)%frequency
  end do
  value=0
  do icol = 0,cols-tem_cols
     do irow = 0,rows-tem_rows
        do inode = 1, tem_nodes
              localPattern(inode)=indivk(
                                  irow+tem_coord_rows(inode),
                                  icol+tem_coord_cols(inode)
                                  )
        end do
        call patternSearch(tem_nodes,localPattern,pos)
        if(pos/=-1) then
           freq_aux(pos)=freq_aux(pos)-1
        end if
     end do
  end do
  do ii=1,npatterns
     value=value+freq_aux(ii)*freq_aux(ii)
  end do
end subroutine fitnessFunction
```

Code 2: Subroutine `fitnessFunction`

We can see that in this implementation, the values of O_p, the weight factors described in equation (1), are equal to 0 if $freq_{TI}(p) = 0$ and equal to 1 otherwise (we only take into account patterns that are present in the training image).

5. Code optimization

We have applied several code optimization techniques to the previous described routines, in order to better exploit the CPU resources of the sequential execution. These optimizations can be grouped as: increase data locality, improve stack memory usage, code specialization of fitness routine, branch and load reductions.

5.1. Increasing data locality of the main data structures

The routines `patternSearch` and `patternComparison` are based on the global data structure `patternList`. The first modification consists in adapting this structure to the column-major order of the Fortran language in order to exploit the data and temporal locality.

The column-major order in Fortran is related to the way in which the CPU accesses the data stored in memory. In this order, the matrices are accessed using the address $row + (col - 1) * numrows$ with $numrows$ fixed. The cache lines that are moved from the main memory to the cache memory consist in contiguous memory addresses of fixed size. Leaving col fixed and traversing first all values of row, we can minimize the accesses to non-contiguous memory addresses (Hennessy and Patterson (1990)). Modifying the data structures in order to increase this kind of accesses reduces the cache data misses, reducing the overall execution time.

The new structure is simply a 2D array in which the row size is the number of nodes in the template, `tem_nodes`, and the column size is the number of patterns found in the training image.

```
module patternOperations
  implicit none
  integer(4) :: npatterns
  integer(4), pointer :: patternList(:,:)  !! (tem_nodes) X (patterns in training image)
  integer(4), pointer :: frequency(:)      !! (patterns in training image)

```

```
contains
  subroutine patternInsertion(...)
  subroutine patternComparison(...)
  subroutine patternSearch(...)
  subroutine printPatternList(...)
end module patternOperations
```

Code 3: Module `patternOperations.f90` with re-designed data structures

5.2. Using the stack memory to store local arrays

In Fortran, using the Intel's compiler `ifort`, we can use the option `-auto`, which causes all local, non-SAVED variables to be allocated on the run-time stack, including fixed-length arrays. The default is `-auto-scalar`, saving all the scalar variables in the run-time stack. The main advantage is that the access time to the run-time stack is faster than the time access to the run-time heap (which stores the dynamically allocated memory), which decreases the execution time (Intel Corporation (2006)). A negative side of this option is related with the stack size. The stack has a maximum size fixed before the execution and if that size is exceeded, a stack overflow error can be obtained. Also, if we use several threads with OpenMP, each thread has its own stack memory space, so the amount of global stack space is increased proportionally to the number of threads. In order to avoid a stack overflow, we can calculate exactly how much data we need to allocate before compile time and see if it can fit in the stack. If it is too big, we can add more space to the stack, for example using the `ulimit` command in Linux operating systems or setting the environment variable `OMP_STACK_SIZE` with an appropriate value.

5.3. Specialization of fitness function to an input template

Given an input template, we can specialize our fitness function routine in order to exploit the data accesses provided by the template. For example, in the fitness function of code 2 using the template of figure 2, the variables `icol+1,...,icol+9` are calculated several times, but in reality we only need to calculate them one time per row-iteration and keep their values stored in an auxiliary variable. This allows for avoiding the accesses to the coordinate arrays `tem_coord_rows` and `tem_coord_cols` (see code 4) and keeping the cache memory *clean* for other data.

If we denote by t = `tem_nodes`, n = `cols` − `tem_cols` and m = `rows` − `tem_rows`, the total number of memory accesses performed by the fitness routine (only taking into account the arrays `tem_coord_rows`, `tem_coord_cols` and `indivk`) is $3 \times t \times n \times m$. Using this optimization the total number of memory accesses for the same arrays is $t \times n \times m$ with a reduction of 3x less memory accesses than the original scenario. In the modified code depicted in 4, using the template described in figure 2, loop unrolling and common subexpression eliminations are included in order to eliminate the accesses of the arrays `tem_coord_rows` and `tem_coord_cols` and re-utilize the values of `icol+1,...,icol+9`.

```
subroutine fitnessFunction(...)
  ...
  integer(4)::rowplus1,rowplus4,rowplus5,rowplus6,rowplus9
  integer(4)::colplus1,colplus4,colplus5,colplus6,colplus9
  ...
  do icol = 0,cols-tem_cols
     colplus1=icol+1
     colplus4=icol+4
     colplus5=icol+5
     colplus6=icol+6
     colplus9=icol+9
     do irow = 0,rows-tem_rows
        rowplus1=irow+1
        rowplus4=irow+4
        rowplus5=irow+5
        rowplus6=irow+6
        rowplus9=irow+9
        localPattern(1)=indivk(rowplus1,colplus1)
        localPattern(2)=indivk(rowplus5,colplus1)
        localPattern(3)=indivk(rowplus9,colplus1)
```

```
        ...
        localPattern(15)=indivk(rowplus1,colplus9)
        localPattern(16)=indivk(rowplus5,colplus9)
        localPattern(17)=indivk(rowplus9,colplus9)
        call patternSearch(tem_nodes,localPattern,pos)
        if(pos/=-1) then
           freq_aux(pos)=freq_aux(pos)-1
        end if
     end do
  end do
  ...
end subroutine fitnessFunction
```

Code 4: Subroutine `fitnessFunction` specialized to the input template of figure 2

5.4. Branch reduction

The routine `patternComparison`, described in code 5, which is the most intensive in terms of execution time, can be re-designed in order to reduce the number of branch instructions (control flow, for example if-then-else or while loops) executed inside a loop.

```
subroutine patternComparison(length,onePattern1,onePattern2,value)
  !
  ! Compare two patterns.
  ! If value = 0, both patterns are equals
  ! If value = 1, the first pattern is bigger
  ! If value = -1, the second pattern is bigger
  !
  integer(4), intent(in)   :: length
  integer(4), intent(in)   :: onePattern1(length)
  integer(4), intent(in)   :: onePattern2(length)
  integer(4), intent(out)  :: value
  integer(4)               :: ii

  value = 0
  ii = 0
  do while ( value == 0 .and. ii < length )
     ii = ii + 1
     if (     onePattern1(ii) > onePattern2(ii) ) then
        value = 1
     elseif ( onePattern1(ii) < onePattern2(ii) ) then
        value = -1
     end if
  end do
end subroutine patternComparison
```

Code 5: Subroutine `patternComparison`

When a branch instruction is processed by the CPU, some cycles may be lost due to an incorrect branch prediction or an expensive condition evaluation (Hennessy and Patterson (1990)). For that reason, we know that reducing the number of branches and relaxing their boolean conditions are good practices in order to reduce the overall execution time. A first version of the modified routine can be viewed in code 6.

```
subroutine patternComparison(length,onePattern1,onePattern2,value)
  ! Compare two patterns.
  ! If value = 0, both patterns are equals
  ! If value = 1, the first pattern is bigger
  ! If value = 2, the second pattern is bigger
  integer(4), intent(in)   :: length
  integer(4), intent(in)   :: onePattern1(length)
  integer(4), intent(in)   :: onePattern2(length)
  integer(4), intent(out)  :: value
```

```
  integer(4)               :: ii
  value = 0
  ii = 0
  do while ( value == 0 .and. ii <length )
     ii = ii + 1
     value=onePattern1(ii) - onePattern2(ii)
  end do
end subroutine patternComparison
```

Code 6: First re-design of subroutine `patternComparison` with reduced number of branch instructions

In this re-design, almost all the branch instructions were eliminated from the original routine. Only the ones injected in the boolean conditions of the `while` loop remain. One possible way to get rid of the evaluation of those conditions is to unroll the loop and use explicit evaluations of each statement incrementing the value of the variable `ii`. Using this technique we are avoiding conditional evaluations and reducing the number of branch instructions performed by the CPU. The re-designed routine can be viewed in code 7. This modified routine is specialized and depends on the number of nodes of the template. In this case we are using the template described in figure 2.

```
subroutine patternComparison(length,onePattern1,
                             onePattern2,value)
  integer(4), intent(in)   :: length
  integer(4), intent(in)   :: onePattern1(length)
  integer(4), intent(in)   :: onePattern2(length)

  value=onePattern1(1) - onePattern2(1)
  if(value/=0)return
  value=onePattern1(2) - onePattern2(2)
  if(value/=0)return
  ...
  value=onePattern1(16) - onePattern2(16)
  if(value/=0)return
  value=onePattern1(17) - onePattern2(17)
  if(value/=0)return
end subroutine patternComparison
```

Code 7: Second re-design of subroutine `patternComparison` with reduced number of branch instructions and minimal boolean conditionals

5.5. Load reduction

In the previous optimized routine `patternComparison` (code 7) the value of the array `onePattern2` is invariant through the entire execution of the caller routine `patternSearch`, implemented as depicted in code 8.

```
subroutine patternSearch(length,onePattern,pos)
  !
  ! Search a pattern in the pattern
  ! database of training image
  ! If pos!=-1, found
  ! If pos==-1, not found
  !
  use patternOperations       !! contains global variables npatterns and patternList
  integer(4), intent(in)   :: length
  integer(4), intent(in)   :: onePattern(length)
  integer(4), intent(out)  :: pos
  integer(4)               :: isFound,minn,maxx
  integer(4)               :: value,ii,jj

  minn = 1
  maxx = npatterns
  isFound=0
  do while ( minn <= maxx .and. isFound == 0 )
     pos = int(real(minn+maxx)* 0.5)
     call patternComparison(
```

```
                        length,
                        patternList(pos)%pattern,
                        onePattern,
                        value
                        )
     if (value == 0) then
        isFound = 1
     elseif (value == 1) then
        maxx = pos - 1
     else
        minn = pos + 1
     end if
  end do
  if (isFound == 0) pos = -1
end subroutine patternSearch
```

Code 8: Subroutine `patternSearch`

Its values correspond to the pattern that is going to be searched in the pattern database stored in `patternList`. Storing each of the array cells in register variables (up to the maximum number of integer registers available in the CPU) is an alternative that reduces the number of loads performed by the CPU. The variables `op1` to `op17` store the values of `onePattern(1)` to `onePattern(17)`, using the template of figure 2. Using a pseudo-inlined version of the routine `patternComparison` together with the registers `op1` to `op17` allows us to reduce by a half the number of loads executed and also to reduce completely the number of routine calls to `patternComparison`. The code 9 depicts this modifications.

```
subroutine patternSearch(length,onePattern,pos)

  use patternOperations   !! contains global variables npatterns and patternList

  integer(4), intent(in)  :: length
  integer(4), intent(in)  :: onePattern(length)
  integer(4), intent(out) :: pos
  integer(4)              :: isFound
  integer(4)              :: minn,maxx,value
  integer(4)              :: ii,jj,pos
  integer(4)              :: op1,op2,op3,op4
  ...
  integer(4)              :: op15,op16,op17

  op1=onePattern(1)
  ...
  op17=onePattern(17)
  minn = 1
  maxx = npatterns
  pos  = -1
  do while ( minn <= maxx )
     pos = int(real(minn+maxx)* 0.5)
     do
        value=patternList(1,pos) - op1
        if(value/=0)exit
        ...
        value=patternList(17,pos) - op17
        if(value/=0)exit
        exit !! value==0
     end do
     if (value == 0) then
        minn = maxx + 1
     elseif (value == 1) then
        maxx = pos - 1
     else
        minn = pos + 1
     end if
```

```
  end do
end subroutine patternSearch
```

Code 9: Re-design of subroutine `patternSearch` with reduced number of loads

6. Parallelization

6.1. Fine grained parallelization with OpenMP

A parallelization scheme using OpenMP over a genetic-based MPS code was described in Peredo and Ortiz (2012). In that work, the proposed parallelization was based on parallel do-loops over the fitness function routine. The parallelization presented in this work is essentially the same, based on parallel do-loops, but taking in consideration the previously described code optimizations.

```
subroutine fitnessFunction(...)
  use patternOperations
  ...
  idthread=omp_get_thread_num()+1
  numthreads=omp_get_num_threads()
  ...
  do ii=1,npatterns
     freq_aux(ii)=frequency(ii)      !! global
     freq_aux_by_id(ii,idthread)=0   !! local
  end do
  value=0

  !$OMP PARALLEL PRIVATE( localPattern,pos,icol,irow,  &
  !$OMP colplus1,colplus4,colplus5,colplus6,colplus9,  &
  !$OMP rowplus1,rowplus4,rowplus5,rowplus6,rowplus9,  &
  !$OMP freq_aux_by_id )
  !$OMP DO
  do icol = 0,cols-tem_cols
     colplus1=icol+1
     colplus4=icol+4
     colplus5=icol+5
     colplus6=icol+6
     colplus9=icol+9
     do irow = 0,rows-tem_rows
        localPattern(1)=indivk(rowplus1,colplus1)
        ...
        localPattern(17)=indivk(rowplus9,colplus9)
        call patternSearch(tem_nodes,localPattern,pos)
        if(pos/=-1) then
           freq_aux_by_id(pos,idthread)=freq_aux_by_id(pos,idthread) + 1
        end if
     end do
  end do
  !$OMP END DO
  !$OMP END PARALLEL

  do jj=1,numthreads
     do ii=1,npatterns
        freq_aux(ii)=freq_aux(ii)-freq_aux_by_id(ii,jj)
     end do
  end do

  do ii=1,npatterns
     value=value+freq_aux(ii)*freq_aux(ii)
  end do
end subroutine fitnessFunction
```

Code 10: Parallelization of specialized subroutine `fitnessFunction` with OpenMP

The main difference with the sequential code 2, besides the code optimizations, is the utilization of a local array `freq_aux_by_id` which stores the frequencies of the patterns calculated locally by all threads. After each thread finishes their corresponding loops (an implicit wait is placed at the end of the loop for thread synchronization) all those frequencies are gathered into the global array `freq_aux` and this array is used to calculate the final value of the fitness function.

Several schedules (`static`, `dynamic` and `guided` with different chunk sizes) were tested but none of them was considerably faster than the others, so we choose to stay using the `static` schedule in all of our tests. No profit was obtained after tuning the chunk size, because no false sharing (a review of this topic can be found in Culler et al. (1998)) was introduced in the calculations since no writing is done in the matrix `indivk`. The default value of chunk size was used in the execution of the tests.

6.2. Coarse grained parallelization with MPI+OpenMP

If the initial population is distributed in several processes, the amount of work that each process does decreases with a reduction in the execution time. Following ideas from Cantú-Paz (1998), we implement an *island-based* parallelization scheme in which each process waits for the best individuals calculated by the others (other *islands*), and asynchronously each process sends to everybody the best individual calculated by itself. In this way, all processes share the same best individuals and each one can combine them in order to breed a new generation. The steps are depicted in algorithm 3. Using this distribution of load, each process can use several threads with OpenMP as in the previous subsection and the code optimizations explained in the previous section. The result of this integration is a hybrid distributed-shared memory parallelization based on the MPI and OpenMP programming models.

Algorithm 3 Parallel canonical genetic algorithm (based on island model)

INPUT: P processes, $\frac{N}{P}$ individuals per process (population)
for each process p **do**
  Evaluate a fitness function $fitness$ in each individual
  while termination criteria is not achieved **do**
    {Breed a new generation}
    Sort the individuals by their fitness function value
    if no improvement is measured in the local population **then**
      Restart: select some individuals and restart their bits
      Sort the individuals by their fitness function value
    end if
    Selection: select the best individuals based on its fitness functions
    Crossover: breed new individuals crossing bits of individuals from the selection
    Mutation: breed new individuals mutating some bits of individuals from the selection
    Replace old individuals by new ones
    Send (asynchronously) the best individual to the other processes
    Receive (asynchronously) the best individuals from other processes and copy them in the population (using the memory space of the worst individuals)
    Evaluate a fitness function $fitness$ in each individual
  end while
end for
OUTPUT: best individual in population of process master.

The only drawback of this parallel implementation is the reduction of the population size in each island. Each process will have a population of $\frac{N}{P}$ individuals, and if N is not sufficiently large, local minimum will be achieved earlier in the convergence process, restricting the search for better solutions. A solution to this problem is to set large population sizes according to the number of processes involved in the executions and the size of the search space, in our case, the number of template nodes and training image nodes. Several tests must be done in order to get a good estimation of the optimal population size. In this work, however, our focus was to get reasonable values of performance and convergence, leaving this topic for future research.

The key part of the parallel strategy is the communication between the processes, which is equivalent to an all-to-all broadcast of the best individuals stored in each island. This communication can be implemented with MPI using asynchronous `MPI_Isend`/`MPI_Irecv`/`MPI_Wait` or with the intrinsic routine `MPI_Allgather`.

A performance analysis tool called Paraver, described in Labarta et al. (1995), was used to generate trace views associated to an execution of the proposed implementation using 16 processes with 1 and 12 threads each. In these traces we can see the different states of execution described in algorithm 3. In the X-axis we have the execution time in microseconds and in the Y-axis we have the states of processes and threads. In figure 4 we can see the overall execution time with both processors sets using the same time scale.

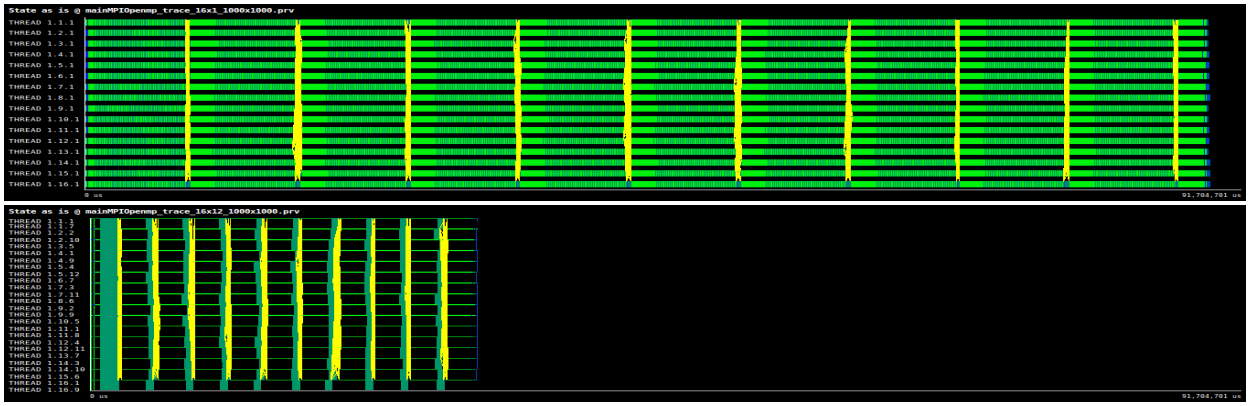

Figure 4: Execution trace (using the same time scale) with two processor sets, 16×1 (top) and 16×12 (bottom) processes×threads, using a training image of size 1000×1000 running 10 generations of the algorithm 3

7. Results

7.1. Timing

In order to study the effectiveness of the optimizations and parallelizations, we use a test routine which loads two 2D images, using one as training image and the other one as individual. After that, it calculates the fitness function of the individual using the previous routines. All measurements are average of 100 executions of this test routine. Two images sizes were used, 100×100 and 1000×1000. They are shown in figure 5. Additionally, full iterations were measured running 30 generations of the genetic algorithm.

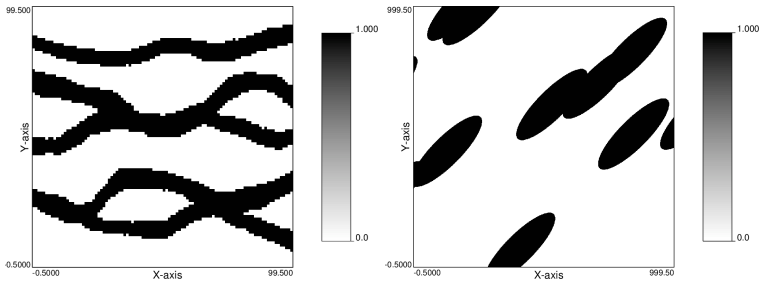

Figure 5: Training images of size 100×100 (left) and 1000×1000 (right)

The code optimizations were tested incrementally: if an optimization X reduces the execution time, all the subsequent optimizations are calculated over the baseline time using the optimization X. The code compilation was done using `ifort` (Intel Fortran Compiler) version 12.0.4 (20110427) (Intel Corporation (2011)), without any added options or flags. The default level of compiler optimization is `-O2`. The tests were executed in a cluster of compute nodes

running Linux operating system where every node has two processors Intel Xeon E5649 6-Core (12 CPUs per node) at 2.53 GHz with 24 GB of RAM memory, 12MB of cache memory and 250 GB local disk storage. The execution times and speed-up values obtained is detailed in tables 1 to 5.

	100×100		1000×1000	
Optimization	Time (seconds)	Speed up	Time (seconds)	Speed up
Baseline	0.00319	1	0.402347	1
Increase data locality	0.00213	1.49x	0.26368	1.53x
Use Stack memory	0.00208	1.53x	0.257903	1.56x
Specialization of fitness	0.00181	1.76x	0.243201	1.65x
Branch reduction	0.00160	1.99x	0.222511	1.80x
Load reduction	0.00157	2.03x	0.214222	1.87x

Table 1: Code optimization: fitness function calculation of a 100×100 and 1000×1000 images using the template in figure 2

	100×100		1000×1000	
Threads	Time (seconds)	Speed up	Time (seconds)	Speed up
1	0.003190	1x	0.402347	1x
2	0.001800	1.77x	0.186401	2.15x
4	0.031770	0.10x	0.094410	4.26x
6	0.020684	0.15x	0.065002	6.18x
8	0.027191	0.11x	0.055411	7.26x
10	0.046754	0.06x	0.049403	8.14x
12	0.027156	0.11x	0.037000	10.87x

Table 2: Fine-grained parallelization without code-optimizations: fitness function calculation of a 100×100 and 1000×1000 images using the template in figure 2

	100×100		1000×1000	
Threads	Time (seconds)	Speed up	Time (seconds)	Speed up
1	0.003190	1x	0.402347	1x
1+code-opt	0.001573	2.03x	0.222202	1.80x
2+code-opt	0.002028	1.57x	0.109205	3.68x
4+code-opt	0.012321	0.25x	0.056436	7.12x
6+code-opt	0.015411	0.20x	0.041220	9.76x
8+code-opt	0.027145	0.11x	0.036421	11.04x
10+code-opt	0.016694	0.19x	0.030886	13.02x
12+code-opt	0.017766	0.17x	0.023531	17.09x

Table 3: Fine-grained parallelization with code-optimizations: fitness function calculation of a 100×100 and 1000×1000 images using the template in figure 2

Based on the results from table 1, the code optimization accelerates considerably the sequential execution of the fitness function calculation. A speedup of 2.03x and 1.87x was obtained for images of 100×100 and 1000×1000 respectively. The best results were obtained after re-designing the data structures that handle the pattern list, allowing it to reduce the execution time by improving the locality of the memory accesses.

	100×100			1000×1000	
Processes×Threads	Time (seconds)	Speed up	Processes×Threads	Time (seconds)	Speed up
1×1	3.466	1x	1×1	312.133	1x
1×2	1.852	1.87x	1×12	55.62	5.61x
2×2	0.934	3.71x	2×12	27.53	11.33x
4×2	0.556	6.23x	4×12	14.03	22.24x
8×2	0.295	11.74x	8×12	7.26	42.99x
16×2	0.213	16.27x	16×12	4.02	77.64x

Table 4: Coarse-grained parallelization without code optimizations: average generation step (30 generations) of the genetic algorithm using a population of 1000 individuals, each one a 100×100 and 1000×1000 images respectively, using the template in figure 2

	100×100			1000×1000	
Processes×Threads	Time (seconds)	Speed up	Processes×Threads	Time (seconds)	Speed up
1×1	3.466	1x	1×1	312.133	1x
1×1+code-opt	2.032	1.70x	1×12+code-opt	42.911	7.10x
2×1+code-opt	1.027	3.37x	2×12+code-opt	22.102	14.12x
4×1+code-opt	0.561	6.17x	4×12+code-opt	11.354	27.49x
8×1+code-opt	0.310	11.18x	8×12+code-opt	5.916	52.76x
16×1+code-opt	0.221	15.68x	16×12+code-opt	3.085	101.17x

Table 5: Coarse-grained parallelization with code optimizations: average generation step (30 generations) of the genetic algorithm using a population of 1000 individuals, each one a 100×100 and 1000×1000 images respectively, using the template in figure 2

According to the results from tables 2 and 3, the fine-grain parallelization allows to accelerate the execution considerably only when the size of the training image and realizations are large, in our case a size of 1000×1000 reaching a speedup of 17.09x with 12 threads in the best case. With a size of 100×100 no profit from the parallelization was observed (except in the non-optimized code with 2 threads), due to the small amount of work that each thread performs compared with the overhead introduced by thread management. The best speedup result, 2.03x, was obtained using the optimized code with only one thread.

The coarse-grain parallelization accelerates the execution proportionally to the number of processes involved in it. This feature is obtained after distributing the workload evenly among the processes. In our case, the workload is represented by the population, which is divided among the processes. According to tables 4 and 5, the best results achieved in terms of speedup were obtained using the hybrid parallelization scheme with optimized code, MPI and OpenMP on large training and realization images of size 1000×1000. The thread selection policy used to defined how many threads will run in each test was based in the best results obtained in the corresponding fine-grain tests from tables 2 and 3, namely, 2 and 12 threads for small and large images respectively, using non-optimized code, and 1 and 12 threads for small and large images respectively, using optimized-code.

7.2. Convergence

In order to test the convergence of the method using the parallel optimized code, we choose a fixed set of parameters for the genetic algorithm and with those parameters we observe the behaviour of the fitness function and the realization obtained, starting from a randomly generated population of individuals. Those parameters are:

- population size: 640 individuals
- mutation rate: described in the next paragraph
- crossover percentage: 50%, corresponding to the number of individuals selected to perform a crossover with other individual
- restart percentage: 10%, corresponding to the percentage of individuals that will be restarted, in each restart step
- number of cut-points: 10%, corresponding to the number of cut-points in the crossover operation
- number of mutation nodes: described in the next paragraph

In order to accelerate the convergence, a multi-point mutation strategy was implemented. In this strategy, a random node and a random category were selected (in our tests, we have only 2 categories). Using an influence radius r calculated a priori (in our case $r = 0.02 \times \min(\#\text{rows}, \#\text{columns})$) all nodes that fall within the circle centered in the random node with radius r are *mutated* (changed) into a new selected category (leaving the conditionant nodes without modification). Additionally, a cyclic-cooling scheme was implemented in order to control the variability of the population, by means of reducing the mutation ratio by a factor $\lambda \in (0, 1)$ each 1000 generations. Starting from a mutation rate $m = 1$, if λm goes below a threshold, the ratio is restarted from $m = 1$ and a new mutation cycle begins.

Samples of realization images and convergence plots of function 1 are included in figures 6-7. The convergence plots show the relative decrease in percentage of the fitness function with respect to the initial value (the best fitness value obtained in the first population of individuals). As we can see in these figures, the simulated images are not equal to the training image, which is not bad, because one of the objectives of the simulation process using MPS methods is to obtain simulated images that fits the underlying statistics of the training image but not being necessarily equal. If conditional data is used (nodes with information from the training image which are not modified in the simulation process) the resulting simulated image will be more similar to the training image with the corresponding match of the underlying statistics. The tests that we are considering do not use any conditional data. Therefore they must be considered as a worst case scenario in terms of convergence.

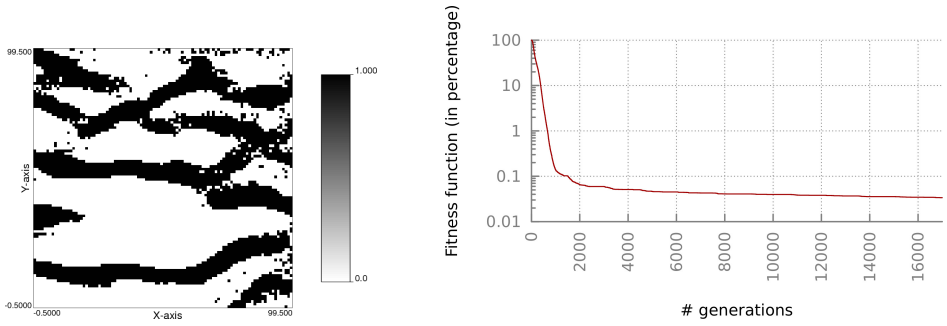

Figure 6: Simulated realization of size 100×100 (left) and convergence plot showing 17000 generations

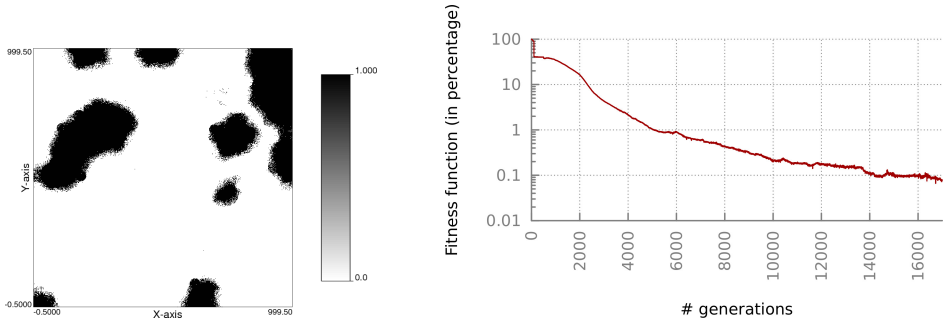

Figure 7: Simulated realization of size 1000×1000 (left) and convergence plot showing 17000 generations

8. Conclusions

The proposed hybrid parallelization using an optimized code has shown reasonable speedup results, according to the time measurements reported in section 7. The differences between non-optimized and optimized code execution are considerable and justify our research in this recent topic. A negative aspect has to do with the loss of generality of some of the proposed optimizations. Applying routine specialization or branch/load reduction introduces several

modifications in the corresponding code, were specific values of the size of the template are used. If the size or geometry of the template changes, we need to modify those routines in order to adapt them to the new template's values. If the complete code has to be used in an agile framework allowing the utilization of user-defined templates, this behaviour can generate inconveniences because each new template needs its own optimized routines, with their corresponding creation, compilation and inclusion in the main application. A possible solution to this problem can be the adoption of auto-tuning techniques to automatically deploy new kernels according to the geometry of the template in use. Examples of auto-tuning techniques applied to stencil optimization for Finite Differences PDE solvers can be reviewed in Datta et al. (2008, 2009).

The strategies proposed to accelerate the convergence allow us to get fast realizations, using reasonable small populations (40 individuals per process) and a small restart percentage (10% of the population is restarted). However, a further research on the acceleration of convergence by tuning the genetic parameters (population size, mutation and crossover rates, percentage of selection, percentage of restarted population, number of cross points and mutation points, among others) is left open for future research. Also, several other research topics can be explored, among them we can mention: dynamic mutation rate (using the full annealing scheme), non-linear crossovers (using external information to mix two realizations with some physical or geological interpretation), or selective restart (allowing to reset only individuals that meet certain properties).

This work focuses on 2D training images and realizations. However 3D models are used in real geostatistical scenarios. In order to adapt our code to the 3D scenario, several further optimizations and modifications must be done, but these are left as future work. Among the most relevant ones we can mention the specialization of the fitness evaluation to 3D templates, use of efficient data structures to manage large 3D images and 1D individuals, new methods to perform crossover and mutations in these individuals, and modifications of the fine and coarse grain strategies to the new 3D scenario. Another related future work could be the application of the fitness calculation optimized routines into the simulated annealing scheme of simulation as described in Deutsch (1992), using the MPI implementation described in Peredo and Ortiz (2011) or the standard routines implemented in the GSLIB library from Deutsch and Journel (1992).

Finally, another possible research area is related to explore new computer architectures with this algorithm. Among the possible alternatives are NVIDIA's GPUs (using two programming models, CUDA and OpenACC), Intel's MICs (several cores in one chip and accelerators working toghether in Intel architectures) or energy-efficient new supercomputers that will be available in the next years.

References

Arpat, B., Caers, J., 2007. Stochastic simulation with patterns. Math. Geology 39, 177–203.

Boisvert, J.B., Lyster, S., Deutsch, C.V., 2007. Constructing training images for veins and using them in multiple-point geostatistical simulation. 33rd International Symposium on Application of Computers and Operations Research in the Mineral Industry, APCOM 2007, E. J. Magri (ed.) , 113–120.

Caers, J., Journel, A.G., 1998. Stochastic reservoir simulation using neural networks trained on outcrop data. SPE Annual Technical Conference and Exhibition, New Orleans, LA, September 1998. Society of Petroleum Engineers. SPE paper # 49026 , 321–336.

Caers, J., Ma, X., 2002. Modeling conditional distributions of facies from seismic using neural nets. Mathematical Geology 34, 143–167.

Cantú-Paz, E., 1998. A survey of parallel genetic algorithms. Calculateurs paralleles, Reseaux et Systems Repartis 10.

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

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

Daly, C., 2005. Higher order models using entropy, markov random fields and sequential simulation. Geostatistics Banff 2004, Leuangthong, O. and Deutsch, C.V., eds., , Springer , 215–224.

Daly, C., Knudby, C., 2007. Multipoint statistics in reservoir modelling and in computer vision. Petroleum Geostatistics A.

Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K., 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures, in: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, IEEE Press, Piscataway, NJ, USA.

Datta, K., Williams, S., Volkov, V., Carter, J., Oliker, L., Shalf, J., Yelick, K., 2009. Auto-tuning the 27-point stencil for multicore, in: In Proc. iWAPT2009: The Fourth International Workshop on Automatic Performance Tuning.

De Jong, K.A., 1980. Adaptive system design: a genetic approach. IEEE Transactions on Systems, Man, and Cybernetics SMC-10, 566–574.

Deutsch, C., Wang, L., 1996. Hierarchical object-based stochastic modeling of fluvial reservoirs. Mathematical Geology 28, 857–880.

Deutsch, C.V., 1992. Annealing techniques applied to reservoir modeling and the integration of geological and engineering (well test) data. Doctoral dissertation. Ph.D. thesis. Stanford University.

Deutsch, C.V., 2002. Geostatistical Reservoir Modeling. Oxford University Press, New York.

Deutsch, C.V., Journel, A.G., 1992. GSLIB : geostatistical software library and user's guide. Oxford University Press, New York, Oxford.
Eskandari, K., Srinivasan, S., 2007. Growthsim - a multiple point framework for pattern simulation. Petroleum Geostatistics A.
Faucher, C., Saucier, A., Marcotte, D., 2013. A new patchwork simulation method with control of the local-mean histogram. Stochastic Environmental Research and Risk Assessment 27, 1–21.
Goldberg, D.E., 1989. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley.
Graham, S.L., Kessler, P.B., McKusick, M.K., 2004. gprof: a call graph execution profiler. SIGPLAN Not. 39, 49–57.
Guardiano, F., Srivastava, M., 1993. Multivariate geostatistics: beyond bivariate moments. Geostatistics Troia 1, 133–144.
Hennessy, J.L., Patterson, D.A., 1990. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, San Mateo, CA.
Holland, J., 1975. Adaptation in natural and artificial systems. U. of Michigan Press.
Hong, S., Ortiz, J.M., Deutsch, C.V., 2008. Multivariate density estimation as an alternative to probabilistic combination schemes for data integration. Geostats 2008 - Proceedings of the Eighth International Geostatistics Congress, J.M. Ortiz and X. Emery (eds.), Gecamin Ltda., Santiago, Chile 1, 197–206.
Intel Corporation, 2006. Fortran Compiler Use of Temporaries. `http://software.intel.com/file/20415` (Aug. 2012).
Intel Corporation, 2011. Intel© Fortran Composer XE 2011. `http://software.intel.com/en-us/articles/intel-composer-xe/` (Sep. 2011).
Krishnan, S., Journel, A.G., 2003. Spatial connectivity: From variograms to multiple-point measures. Mathematical Geology 35, 915–925.
Labarta, J., Girona, S., Pillet, V., Cortes, T., Cela, J.M., 1995. A parallel program development environment, in: In proceedings of the 2th. Int. Euro-Par Conference, Springer. pp. 665–674.
Lyster, S., Deutsch, C.V., 2008. Mps simulation in a gibbs sampler algorithm. Geostats 2008 - Proceedings of the Eighth International Geostatistics Congress, J.M. Ortiz and X. Emery (eds.), Gecamin Ltda., Santiago, Chile 1, 79–88.
Ortiz, J.M., 2003. Characterization of high order correlation for enhanced indicator simulation, Unpublished doctoral dissertation. Ph.D. thesis. University of Alberta.
Ortiz, J.M., Deutsch, C.V., 2004. Indicator simulation accounting for multiple-point statistics. Mathematical Geology 36, 545–565.
Ortiz, J.M., Emery, X., 2005. Integrating multiple point statistics into sequential simulation algorithms. Geostatistics Banff 2004, Leuangthong, O., and Deutsch, C.V., eds., Springer , 969–978.
Parra, A., Ortiz, J.M., 2009. Conditional multiple-point simulation with a texture synthesis algorithm. IAMG 09 Conference, Stanford University .
Peredo, O., Ortiz, J.M., 2011. Parallel implementation of simulated annealing to reproduce multiple-point statistics. Computers & Geosciences 37, 1110 – 1121.
Peredo, O., Ortiz, J.M., 2012. Multiple-point geostatistical simulation based on genetic algorithms implemented in a shared-memory supercomputer, in: Abrahamsen, P., Hauge, R., Kolbjrnsen, O. (Eds.), Geostatistics Oslo 2012. Springer Netherlands. volume 17 of *Quantitative Geology and Geostatistics*, pp. 103–114.
Pyrcz, M.J., Strebelle, S., 2008. A showcase of event-based geostatistical models, in: Ortiz, J.M., Emery, X. (Eds.), Geostas Oslo 2012. Gecamin Ltda., Santiago, Chile. volume 2, pp. 1143–1148.
Rezaee, H., Mariethoz, G., Koneshloo, M., Omid Asghari, O., 2013. Multiple-point geostatistical simulation using the bunch-pasting direct sampling method. Computers & Geosciences 54, 293 – 308.
Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J., 1998. MPI-The Complete Reference, Volume 1: The MPI Core. MIT Press, Cambridge, MA, USA. 2nd. (revised) edition.
Straubhaar, J., Renard, P., Mariethoz, G., Froidevaux, R., Besson, O., 2011. An improved parallel multiple-point algorithm using a list approach. Mathematical Geosciences , 1–24.
Strebelle, S., 2002. Conditional simulation of complex geological structures using multiple-point statistics. Mathematical Geology 34, 1–21.
Strebelle, S., Journel, A.G., 2000. Sequential simulation drawing structures from training images. 6th International Geostatistics Congress, Cape Town, South Africa. Geostatistical Association of Southern Africa. .
Tjelmeland, H., 1996. Stochastic models in reservoir characterization and Markov random fields for compact objects, Unpublished doctoral dissertation. Ph.D. thesis. Norwegian University of Science and Technology.
Xu, W., 1996. Conditional curvilinear stochastic simulation using pixel-based algorithms. Mathematical Geology 28, 937–949.
Zanon, S., 2004. Advanced aspects of sequential Gaussian simulation. Master's thesis. University of Alberta.
Zhang, T., Switzer, P., Journel, A., 2006. Filter-based classification of training image patterns for spatial simulation. Mathematical Geology 38, 63–80.