




Land consolidation through parcel exchange among landowners using a distributed Spark-based genetic algorithm

Diego Teijeiro¹  · Margarita Amor¹ · Ramón Doallo¹ · Eduardo Corbelle² · Juan Porta¹ · Jorge Parapar¹

Accepted: 23 May 2022 / Published online: 24 June 2022
© The Author(s) 2022

Abstract

Land consolidation is an essential tool for public administrations to reduce the fragmentation of land ownership. In particular, parcel exchange shows promising potential for restructuring parcel holdings, even more when the number of parcels and owners involved is large. Unfortunately, the number of possible exchange combinations grows very quickly with the number of participating landowners and parcels, with the associated challenge of finding an acceptable solution. In this paper, we present a high-performance solution for parcel exchange based on genetic algorithms. Our proposal, using Apache Spark framework, is based on the exploiting of distributed-memory systems with effortless access in order to reduce the execution time. This also allows increasing the search width through multiple populations that share their advances. This can be achieved without compromising the search depth thanks to the higher amount of resources available from using distributed-memory systems. Our proposal is capable of achieving better solutions in lower amounts of time compared to previous works, showing that genetic algorithms on a high performance system can be used to propose fair parcel exchanges under strict time constraints, even in complex scenarios. The performance achieved allows for fast trial of several options, reducing the time usually needed to perform administrative procedures associated with land fragmentation problems. Specifically, our proposal is capable of combining the benefits of both depth-focused and width-focused multithreaded parallelization. It matches the speedup gains of depth-focused multithreaded parallelization. The width-focused parallelization provides local minimum resilience and fitness value reduction potential. In this paper, multithreading solutions and Spark-based solutions are tested.

Margarita Amor, Ramón Doallo, Eduardo Corbelle, Juan Porta and Jorge Parapar authors have contributed equally to this work.

Extended author information available on the last page of the article

Keywords Land fragmentation · Geographic information systems · Global optimization · Genetic algorithms · Parcel exchange · Apache Spark

1 Introduction

High fragmentation of land ownership is generally regarded as a threat to the profitability of farms and forest holdings [1, 2]. Actual fragmentation can be the result of quite different situations: a high number of landowners, a high number of holdings (land users), lack of overlap between landowners and land users, a high number of land parcels per landowner, or high average distances between the parcels of the same landowner [3, 4]. More usually, real cases result from a combination of these situations.

Public and private initiatives aimed at reducing land fragmentation exist in most countries and, very often, these focus on reducing the average number of parcels per landowner and/or on relocating parcels of each landowner as close as possible to each other [5–7] and, as such, may include the exchange of parcels between landowners. The potential benefits of parcel exchange increase with the total number of landowners and the total number of parcels involved in the exchange, as the number of possible solutions increases very quickly. Obviously, a higher number of possible solutions also make the search of a (sub)optimal solution a harder task to fulfill and the use of heuristics becomes suitable for these cases.

Compared to exchanging parcels one-by-one between two landowners, multiple parcel exchange has logically greater potential benefits, increasing with the number of owners and parcels involved in the exchange process. At this point, the problem can be simplified to a combinatorial problem, and as such, the number of possible exchange combinations grows very quickly as both the number of landowners and parcels involved increase [8]. Combinatorial or trial-and-error approaches, even performed by computers, need large amounts of time that increases exponentially with the number of owners and parcels. Due to the size of the problem, in general, these approaches are not viable and we must look for other options to solve them. A common viable option for complex problems with large numbers of possible solutions is the use of heuristic algorithms. These algorithms trade the optimality of the solution (they are not guaranteed to find the best solution) for performance (the speed of the algorithm), reducing the time needed to achieve good solutions, and as such, they may be a viable technique in this case.

From a general perspective, heuristic algorithms search for the solution by attempting to improve a candidate solution iteratively, according to a measure of quality. They are used to search for solutions of complex problems, but not necessarily the best one, in much shorter times. This speed allows them to solve complex problems such as those commonly found in land management or land administration. Among the several subtypes of heuristic algorithms, genetic algorithms (GA) operate with the basic principles of natural evolution: The best individuals on a population produce the next generations more suited for the environment. This translates to an algorithm that maintains a group of candidate solutions (population) from which the best individuals according to a fitness function (FF) are selected to create

new individuals. Every new generation of the population is formed with potentially better solutions, repeating this procedure until the stopping criteria are met.

Previous publications have proven that evolutionary algorithms are well suited to support multiobjective spatial decision-making [9, 10]. Several applications can be found in literature, from traditional land consolidation [11] to space planning [12, 13], automatic delimitation of population settlements [14] and land-use allocation [15–18]. While there are several cases of genetic algorithms proposed to support decisions concerning land reallocation in traditional land consolidation processes [19–22], a different approach is raised in [23], where the results of the proposed GA are analyzed from an agroforestry standpoint. The effectiveness of this approach in real cases was verified with satisfactory results. The performance of GAs can be improved with several optimizations and techniques already known and tested in the literature [15].

Recently, big data technologies, such as Hadoop, Spark or Flink, have emerged as efficient solutions for large applications on distributed-memory systems. These technologies provide easier access to high amounts of computational resources, abstracting the underlying hardware structure. Some of them are flexible enough to use them outside of the problem scales usually related to Big Data, enabling their use on traditional problems. Other uses of these big data technologies for genetic algorithms can be found on the literature [24–26], some using clusters [27], others in public clouds [26].

In this paper, we analyze the utilization of a big data framework, specifically Spark, to not only increase the performance of the algorithm but also improve the quality of the results, enabling more complex configurations or support larger use cases. Spark has proven its suitability for this type of algorithms in the literature [27].

The rest of the paper is structured as follows: Sect. 2 describes the application of genetic algorithms for parcel exchange. Section 3 describes some details of the proposed system using a multithread approach, and Sect. 4 presents the proposal to perform parcel exchange using Apache Spark. Section 5 presents the results obtained and Section 6 summarizes the conclusions and outlines future work.

2 Genetic algorithm for parcel exchange: GeAPaE

Genetic algorithms are among the several search heuristics usually applied to optimization, in this case, inspired by natural processes like natural selection, and therefore, most of the terminology and mechanisms come from those fields.

The base element of the algorithm is an *individual*, and in itself is a possible solution to the optimization problem. This means that an individual encodes all the data needed to represent a possible solution to the problem, and that is the assignment of an owner to each of the N parcels involved. These can be also referred to as *ownership patterns* and can be abstracted as an ordered sequence of owner identifiers. An individual contains one owner identifier, one of the L owners of the particular use case for each parcel involved, each of those identifiers can be called a *gene* in common genetic algorithm terminology. The algorithm

tries to find an individual that has the lowest value according to a specific set of rules usually represented as a mathematical function known as *fitness function* (FF) that factors the owner of each parcel in the computation of the value.

The basic algorithm maintains Q different groups of individuals called *populations*, denoted as $P^1, \dots, P^j, \dots, P^Q$, and it creates new individuals using a combination of *crossover* and *mutation* processes. Each population could have a potential different number of individuals, denoted as M_j . Each individual can be identified by their population and element index, denoted as P_i^j (the individual i of population j). $P_{i,k}^j$ denotes the owner of the k -th parcel according to the i -th individual of the j -th population, one for each of the N parcels involved. These populations can also be referred as *generations* when one group is created from the members of the other to introduce the sense of parenthood or precedence: Individuals of a population generate a new group of individuals. These terms will be used interchangeably during this work.

In [23], different fitness functions that were used are explained in detail. In some of them, a *reference point of the owner* is mentioned, this point being a location indicated by the owner. This reference point usually is located at their farmyard and indicates the place around which the owner prefers their parcels to be located. A short description of each fitness function is as follows:

- *Parcel Distances Total (PDT)*: adds all the distances between each pair of parcels assigned to the same landowner, for all landowners.
- *Parcel Distances Average (PDA)*: similar to the previous one but uses the average for each landowner instead of just adding them.
- *Reference point Distance Total (RDT)*: adds all the distances between each parcel and the reference point of the owner.
- *Reference point Distance Average (RDA)*: similar to the previous one but the value for each landowner is the average instead of the sum.
- *Total Distances Combined (TDC)*: combines the *PDT* and *RDT* fitness functions, with potentially different weights.
- *Average Distances Combined (ADC)*: combines *PDA* and *RDA*, again with independent weights.
- *4 Distances Combined (4DC)*: combines the first four fitness functions, that is *PDT*, *PDA*, *RDT* and *RDA*, giving different weights to each one.
- *Number Of ParcelS (NPL)*: number of parcels assigned to the owner.

All of these fitness functions except the last one, *NPL*, calculate the value for each landowner and average the values of each owner to give the final fitness value of the individual. The *NPL* fitness functions instead add the values of every owner, and the final value for an individual is the total number of parcels.

Table 1 summarizes the main equations for each fitness function. N_o is the number of parcels assigned to the owner o in the individual being evaluated. $PD_{i,j,o}$ is the distance between parcel i and j of the owner o . $RD_{i,o}$ is the distance from parcel i to the reference point of the owner o . Finally, W_{PDT} , W_{PDA} , W_{RDT} and W_{RDA} are the weights given to the first, second, third and fourth evaluation methods, respectively.

Table 1 Equations for fitness functions used in [23]

Fitness function	Equation
PDT	$\frac{\sum_{o=1}^{o=L} \sum_{i=1}^{i=N_o} \sum_{j=i}^{j=N_o} PD_{i,j,o}}{L}$
PDA	$\frac{\sum_{o=1}^{o=L} \frac{\sum_{i=1}^{i=N_o} \sum_{j=i}^{j=N_o} PD_{i,j,o}}{N_o}}{L}$
RDT	$\frac{\sum_{o=1}^{o=L} \sum_{i=1}^{i=N_o} RD_{i,o}}{L}$
RDA	$\frac{\sum_{o=1}^{o=L} \frac{\sum_{i=1}^{i=N_o} RD_{i,o}}{N_o}}{L}$
TDC	$\frac{\sum_{o=1}^{o=L} \left(\left(W_{PDT} * \sum_{i=1}^{i=N_o} \sum_{j=i}^{j=N_o} PD_{i,j,o} \right) + \left(W_{RDT} * \sum_{i=1}^{i=N_o} RD_{i,o} \right) \right)}{L}$
ADC	$\frac{\sum_{o=1}^{o=L} \left(\left(W_{PDA} * \frac{\sum_{i=1}^{i=N_o} \sum_{j=i}^{j=N_o} PD_{i,j,o}}{N_o} \right) + \left(W_{RDA} * \frac{\sum_{i=1}^{i=N_o} RD_{i,o}}{N_o} \right) \right)}{L}$
4DC	$\frac{\sum_{o=1}^{o=L} \left(\left(W_{PDT} * \sum_{i=1}^{i=N_o} \sum_{j=i}^{j=N_o} PD_{i,j,o} \right) + \left(W_{PDA} * \frac{\sum_{i=1}^{i=N_o} \sum_{j=i}^{j=N_o} PD_{i,j,o}}{N_o} \right) + \left(W_{RDT} * \sum_{i=1}^{i=N_o} RD_{i,o} \right) + \left(W_{RDA} * \frac{\sum_{i=1}^{i=N_o} RD_{i,o}}{N_o} \right) \right)}{L}$
NPL	$\sum_{o=1}^{o=L} N_o$

The fitness function to use can be configured in each execution according to the needs of the users. In some cases, the goal is to have the parcels closest to their main farmyard (RDA or RDT would be the best ones to use). Sometimes, they don't need to be so close to their farmyard but prefer parcels grouped together (PDA or PDT in that case). Other times they want to have the lowest amount of separated pieces of land with no regards to the shape itself (NPL performing the parcel union, only cares about the amount of parcels after fusing the ones assigned to each landowner wherever possible).

3 Description of GeAPae implementation using multithreading

This section describes some internal peculiarities of previous implementations of GeAPae [23]. GeAPae is implemented using the Java programming language and parallelized using threads. It maintains all the modes of execution (command-line, integrated web application or GeoServer WPS operation). In this paper, we use command-line mode exclusively as it is the most adequate for the Spark environment used (HPC cluster with job scheduling). Input, output and configuration of the execution remain unchanged, as well as the distance calculation and caching done at the start.

In this section, we detail how the genetic algorithm works, focusing on the parallelization techniques that exploit the multithreading capabilities of modern systems. The algorithm can use multiple processing threads in two different ways: parallelize the creation of new individuals to create the next generation quicker for a

given population, or maintain several populations evolving independently that share individuals periodically. These two approaches have different effects on the overall algorithm, increasing the thread count working in the same population increases the amount of generations completed in the same time, which can be understood as increasing the *depth of the search* or *search depth* since it can stack mutations on the individuals. Increasing the number of populations, on the other hand, does not increase the amount of generations achieved, but since each population evolves independently, they can test different mutation paths in parallel, increasing the *width of the search* or *search width* when taking into account all of the populations.

3.1 Overall structure

For a single population, a more detailed description of the algorithm is presented in [23]. Algorithm 1 shows the pseudocode of the sequential algorithm, to explain the evolution process.

The algorithm starts with an initial phase of population creation (lines 1-6). In this phase, M_k individuals are created to fill the first generation, checking that the ownership pattern is valid, that is, each landowner keeps the total property value within a prefixed margin with respect to his property value in the original distribution. When the population is complete, the algorithm enters the second and main phase, the evolution loop.

The evolution is performed for a predefined amount of time (line 7). For each individual in the population, another random individual is selected, and the crossover operation is performed (lines 15-19), generating two children. Those new individuals are mutated one or several times, and if the resulting individual is valid, it is added to the next generation's population (lines 20-37). The best of the children and the original elements is selected as the individual to be included into the next generation (line 38). If the best fitness in the population does not change during several generations, the algorithm is considered stagnated and the loop stops before the maximum time is reached. This is done to avoid wasting execution time when the optimum is reached, either a local or global optimum, and, as such, this stagnation detection can be seen as another stopping criterion.

Algorithm 1 GeAPae pseudocode for a single population P^k .

```

1: for  $i = 1$  to  $M_k$  do ▷ Population creation
2:   Create  $P_i^k$ 
3:   while  $P_i^k$  is not valid do
4:      $\tilde{P}_i^k =$  Create  $P_i^k$ 
5:   end while
6: end for
7: while  $t \leq T\_max$  do ▷ Evolution loop
8:   if  $T\_since\_last\_export \geq T\_export$  then ▷ Export step
9:     Export best individuals, adding to import queues of other populations
10:  end if
11:  if Import queue is not empty then ▷ Import step
12:    Add individuals from import queue to  $P^k$ , sort by fitness, truncate to  $M_k$ 
13:  end if
14:  for  $i = 1$  to  $M_k$  do
15:    Select  $P_j^k$  where  $1 \leq i, j \leq M_k$  and  $i \neq j$ 
16:    Add  $P_i^k$  and  $P_j^k$  to offspring candidates
17:    Randomly select  $l_0$  and  $l_1$  where  $1 \leq l_0 \leq l_1 \leq N$ 
18:     $\tilde{P}_i^k = P_{i,1}^k, \dots, P_{i,l_0}^k, P_{j,l_0+1}^k, \dots, P_{j,l_1}^k, P_{i,l_1+1}^k, \dots, P_{i,N}^k$ 
19:     $\tilde{P}_j^k = P_{j,1}^k, \dots, P_{j,l_0}^k, P_{i,l_0+1}^k, \dots, P_{i,l_1}^k, P_{j,l_1+1}^k, \dots, P_{j,N}^k$ 
20:    Mutate  $\tilde{P}_i^k$ 
21:     $mutations\_made = 1$ 
22:    while  $\tilde{P}_i^k$  is not valid and  $mutations\_made \leq Max\_mutations$  do
23:      Mutate  $\tilde{P}_i^k$ 
24:       $mutations\_made = mutations\_made + 1$ 
25:    end while
26:    if  $\tilde{P}_i^k$  is valid then
27:      Add  $\tilde{P}_i^k$  to offspring candidates
28:    end if
29:    Mutate  $\tilde{P}_j^k$ 
30:     $mutations\_made = 1$ 
31:    while  $\tilde{P}_j^k$  is not valid and  $mutations\_made \leq Max\_mutations$  do
32:      Mutate  $\tilde{P}_j^k$ 
33:       $mutations\_made = mutations\_made + 1$ 
34:    end while
35:    if  $\tilde{P}_j^k$  is valid then
36:      Add  $\tilde{P}_j^k$  to offspring candidates
37:    end if
38:     $P_i^{k'} =$  individual with best fitness value from offspring candidates
39:  end for
40:   $P^k = P^{k'}$ 
41: end while
42: return  $\hat{P}_\gamma^k$  where  $F(\hat{P}_\gamma^k) = \min\{F(P_0^k), \dots, F(P_M^k)\}$ 

```

Two different approaches to increase performance and reach better fitness values faster are proposed. One is based on increasing the *search depth* by parallelizing the creation of new individuals. The other approach is based on increasing the *search width*, creating several populations that evolve independently but share some individuals periodically.

The first parallelization strategy uses the available threads to accelerate the creation of new generations. This is achieved by parallelizing the for loop in lines 14-39. Each iteration of the loop is considered an independent task, the i -th iteration is the creation of the individual that will replace the i -th individual in the next generation, and the available threads complete those tasks. The tasks in a generation are independent with each other, while there is a dependency between generations. There is no static scheduling of those tasks, since the computational cost of each one is not fixed, each thread proceeds with the next tasks when it completes the previous one, balancing the load of each thread and making efficient use of the resources. Increasing the speed at which new generations are completed increases the number of mutations that are applied to the same individual, since the mutations only occur between generations. The context of the algorithm increases the relevance of the mutation process, since it is the step that introduces randomness to the individuals, necessary to explore new parcel-landowner associations. Therefore, the depth of the genealogy of an individual, also seen as the *search depth*, has a notable role in the performance of the algorithm.

With regards to the width focused approach, a single process handles each population in a different thread in order to be able to use shared memory to perform that cooperation, instead of communication between different processes. The cooperation between populations is performed in the export/import step of the algorithm (lines 8-13), using lists of individuals as import queues in a producer-consumer pattern. At set intervals, each population exports its best individuals, adding them to the import queues of every other population (lines 8-10). Each population is independent of the others, and the only interaction is the asynchronous communication through the import queues that are managed at a higher level of abstraction.

When a population imports individuals of other populations, all incoming individuals are evaluated with the configuration of the accepting population, in case they originate from a population that uses a different fitness function, and they are appended to the current population, temporarily increasing the population size. In order to avoid uncontrolled population size increases, the inflated population is sorted by fitness value from best to worst, and the list is truncated to the correct population size, removing the worst individuals from it (lines 11-13). If the imported individuals are better than some of the ones already in the population, the old ones will be removed. Otherwise, the imported individuals are removed when the list is truncated, and the population will keep its original individuals.

Figure 1 shows a representation of this process when executed in a single node using the multithread versions of the two approaches detailed. In the shown case, there are three populations with 4, 3 and 4 individuals, respectively, where the second population exports the best individual to the other populations and they integrate that individual in their individual lists. For reasons of clarity, only population 2 shares individuals with the other two populations. In the case shown, P^2 performs the export, sending its best individual, P_1^2 to the other two populations. Those populations import that individuals, adding it to their individuals list, sort them by increasing fitness value and they truncate the individual list to the population size, four in both cases. In P^1 , the imported individual is better than the worst of the existing individuals, so the truncation removes P_4^1 and the imported individual becomes

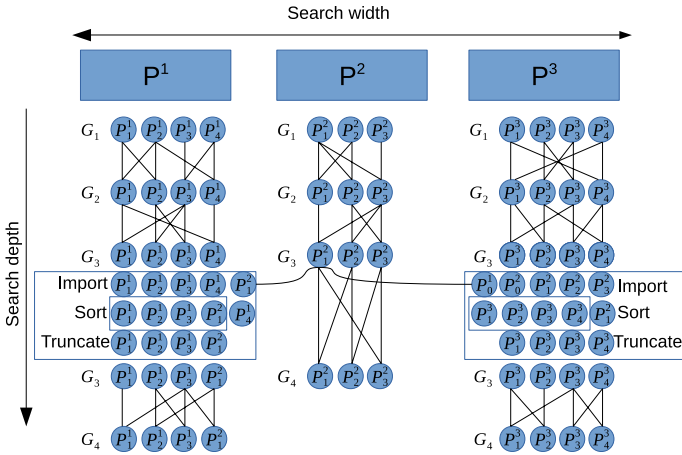


Fig. 1 Scheme of the evolution of multiple populations exploiting both parallelization strategies, search width and depth

part of P^1 . In P^3 , the imported individual is worse than all of the existing ones, so the truncation removes the imported individual and the population remains unchanged.

The increase in search width by itself does not provide big improvements, search depth tends to be more important than search width, in general, more so in this case due to the mutation step being the main source of fitness improvement. However, since the whole algorithm’s evolution is performed isolated in each population, a different configuration can be used in each population. This heterogeneous configuration opens new possibilities, an advancement in one population that is focused on one fitness function can have a positive impact in another population that is searching with other fitness function. This has a large impact when the computational cost of several fitness functions is very different, populations with fast fitness functions will complete new generations faster, and individuals from those deeper generations will be exported to slower populations, jumpstarting it or introducing big leaps in the evolution of the slower populations.

4 Efficient Spark-based GeAPaeSp implementation

With the approaches explained in Sect. 3, a trade-off must be sought between depth and search width, choosing how to dedicate the computational resources of the system that runs the algorithm. The possibility of not having to compromise one of the two levels of parallelism, increasing the amount of computational resources available, is appealing. There is a large number of tools and techniques to utilize distributed memory systems effectively, one big example being MPI. The rise of Big Data leads to the creation of new commercially oriented and more flexible software tools to utilize those systems in different applications, in contrast to the more research or academic focused uses or traditional approaches. An alternative approach of the multiple population parallelization has been proposed, using the Apache Spark

framework to allow simple access to distributed memory systems like clusters or cloud infrastructure, obtaining other features like fault tolerance without extra effort.

There are other alternatives to Spark, such as Flink or Hadoop. The main reason for choosing Spark is ease of use and correct support for iterative algorithms. Hadoop is not adequate for iterative algorithms like a genetic algorithm as it has been tested and compared with Spark in previous works [27]. One of the main problems with Hadoop is the requirement to use HDFS for input, output an intermediate results, introducing a forced overhead of writing intermediate results to disk, which can be too high depending on the configuration as explored in previous works [27]. Flink, on the other hand, is not adequate for our proposal due to its design focus on stream processing.

Our Spark approach allows the utilization of multiple populations to increase the search width while using all available resources in each Spark worker node to maximize the search depth in each population. Each population can be distributed to a different worker node and use all of its available threads for the parallelization to maximize the search depth.

Apache Spark uses the concept of *Resilient Distributed Dataset* (RDD) as the main tool for efficient fault tolerance and distributed memory computation. An RDD is a collection of data distributed across the underlying infrastructure that can be operated in parallel and cached in memory when possible. In our case, we create an RDD from data already in memory, integrating Spark with the existing algorithm. Spark uses a master/worker architecture, with the *Driver* program running on the master node and *Executors* running on the worker nodes using resources allocated through a cluster manager.

4.1 Spark distributed structure

Our proposal takes advantage of the resources available through Spark by distributing the populations to different worker nodes and uses all the resources on each worker node to increase the search depth of the population assigned to it. The basic abstraction is based on the execution of the application on the worker nodes, each worker node performs the evolution of one population, with no interaction with the others. After the evolution is completed, some individuals are copied to other populations to allow cooperation, and the evolution is performed again repeating it until the criteria of time elapsed or stagnation are achieved.

Figure 2 shows a simple scheme of how the computation is distributed in a traditional Spark environment. In contrast to other parallelization techniques seen in the literature, this work is focused on the quality improvement that is possible with this approach, while maintaining the benefits on reduction in execution time. With our proposal, we can use the distributed memory paradigm to increase search width, running several populations in parallel in different nodes, and the shared memory to maximize the search depth in each population using all of the available threads in each node to increase the search depth of the population evolving in that node.

The driver program creates a RDD of *key-value* pairs, where the *value* is the individual list that constitutes a population, and the *key* is the *ID* (or index) of the

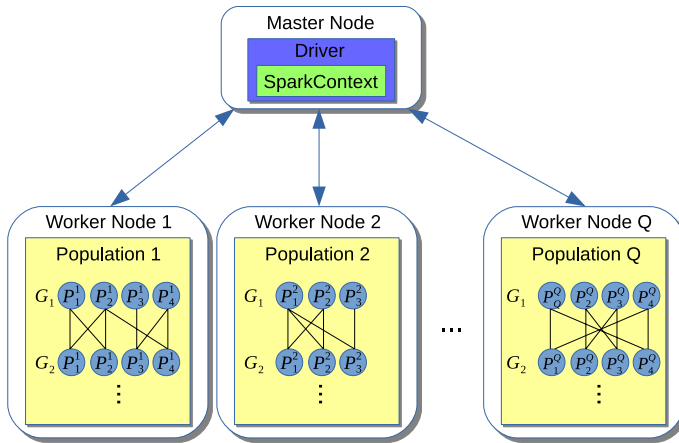


Fig. 2 Scheme of the evolution of multiple populations using our Spark approach

population. At this point, the RDD contains each population associated with its *ID*. A *flatmap* operation is executed on the RDD, and this operation performs the evolution of the population, and a *reduce* operation is performed to group the output of the *flatmap* by the key of each element. Then, the process is repeated with the reformed populations until the maximum allowed time is reached. Each group of *flatmap-reduce* operations is called a *stage* for future reference. We describe each of these operations in detail below.

The *flatmap* operation receives each key-value pair, runs the evolution on the population and creates new key-value pairs with the population index as key and each individual in its own list. The execution of the flatmap operation in each key-value pair is independent of the others in the same stage, while there is a dependency with consecutive stages. For one population of M_k individuals, M_k pairs are created, each one with the index of the population as key and a list containing the *i*-th individual. Additionally, new pairs for the *b* best individuals of the population are also generated (the precise amount can be configured). In an execution with Q populations, the *b* best elements of the population create $Q - 1$ new pairs, each one with the index of another population, and a list with one of the best individuals. At the end, there are M_k pairs with lists of one individual containing the full population in total, and $b * (Q - 1)$ with the best individuals copied with the indexes of the other populations.

The *reduce* operation joins the lists that share the same key. This operation applies an associative and commutative binary operator to all the key-value pairs sharing the same key. All the pairs with the same key are sent to one worker node. Spark should do this in a way that reduces the data transfers keeping each population in the same node and only moving the copied individuals, where the *reduce* operation is executed over those pairs. This binary operator receives two pairs and creates a new and single pair with a list that contains the individuals of the original pairs. After this operation is completed, the RDD returns to the original

size, with one pair per population with the population *ID* as key and the list of individuals that constitute the population as value, ready to repeat the process.

The process of dividing the populations into lists of one individual at the end of the *flatmap* operation, and regrouping them with the *reduce* operation, allows the cooperation between different populations. Spark does not allow direct communication between worker nodes, so the best way of moving data from one worker to another is to complete the task which the worker is currently doing, rearrange the data in the RDD and launch another computation task. The movement of data between worker nodes is normally discouraged as it carries a relevant overhead, but in this case, the amount of data that needs to move through the network is not large, only some individuals of the populations.

After the first stage, the size of the lists (size of each population) increases, because on top of the individuals of the population itself, the best individuals of other populations are also added. To avoid increasing the size of the populations continually, only the M_k best individuals of each populations create new pairs at the end of the *flatmap* operation, with M_k being the original population size. The evolution may be with more individuals, but only the best M_k make it to the next stage, keeping the population size under control.

Algorithm 2 shows a representation of the RDD through several stages, using three populations of 3, 2 and 4 individuals and exporting one individual to other populations.

An important performance parameter is the duration of each stage, the time each population evolves isolated, as it controls the level of cooperation. On the one hand, short stages, more of them, are desired to increase the cooperation between populations, but on the other hand, each stage has an associated overhead due to data movement, so less stages are also desired. These two conflicting factors need to be balanced, so the progress achieved in the evolution is worth the overhead time introduced for communications. The duration of each stage can be adjusted in the configuration file to balance these two factors in each execution.

Algorithm 2 Evolution of the RDD through two stages

- 1: $\langle (0, \langle P_0^0, P_1^0, P_2^0 \rangle), (1, \langle P_0^1, P_1^1 \rangle), (2, \langle P_0^2, P_1^2, P_2^2, P_3^2 \rangle) \rangle$
 - 2: *First stage start*
 - 3: flatmap execution;
 - 4: $\langle (0, \langle P_0^0 \rangle), (0, \langle P_1^0 \rangle), (0, \langle P_2^0 \rangle), (1, \langle P_0^0 \rangle), (2, \langle P_0^0 \rangle), (1, \langle P_1^1 \rangle), (1, \langle P_1^1 \rangle), (0, \langle P_0^1 \rangle), (2, \langle P_0^1 \rangle), (2, \langle P_0^2 \rangle), (2, \langle P_1^2 \rangle), (2, \langle P_2^2 \rangle), (2, \langle P_3^2 \rangle), (0, \langle P_0^2 \rangle), (1, \langle P_0^2 \rangle) \rangle$;
 - 5: reduce operation;
 - 6: $\langle (0, \langle P_0^0, P_1^0, P_2^0, P_0^1, P_0^2 \rangle), (1, \langle P_0^1, P_1^1, P_0^0, P_0^2 \rangle), (2, \langle P_0^2, P_1^2, P_2^2, P_3^2, P_0^0, P_0^1 \rangle) \rangle$;
 - 7: *First stage end, second stage start*
 - 8: flatmap execution;
 - 9: $\langle (0, \langle P_1^0 \rangle), (0, \langle P_0^0 \rangle), (0, \langle P_0^0 \rangle), (1, \langle P_1^0 \rangle), (2, \langle P_1^0 \rangle), (1, \langle P_1^1 \rangle), (1, \langle P_2^2 \rangle), (0, \langle P_1^1 \rangle), (2, \langle P_1^1 \rangle), (2, \langle P_2^2 \rangle), (2, \langle P_3^2 \rangle), (2, \langle P_0^0 \rangle), (2, \langle P_1^2 \rangle), (0, \langle P_2^2 \rangle), (1, \langle P_2^2 \rangle) \rangle$;
 - 10: reduce operation;
 - 11: $\langle (0, \langle P_1^0, P_0^0, P_0^1, P_1^1, P_2^2 \rangle), (1, \langle P_1^0, P_1^1, P_2^2, P_2^2 \rangle), (2, \langle P_1^0, P_1^1, P_2^2, P_3^2, P_0^1, P_1^2 \rangle) \rangle$;
 - 12: *Second stage end, third stage start*
 - 13: ...
-

Table 2 Test cases information

Test cases	Municipality	N	L	Characteristics
Synthetic	–	351	10	Identical parcels Full land coverage Full adjacency
Real case	Ribadeo (Spain)	329	12	Low size variation Low land coverage High parcel dispersion High reference points dispersion

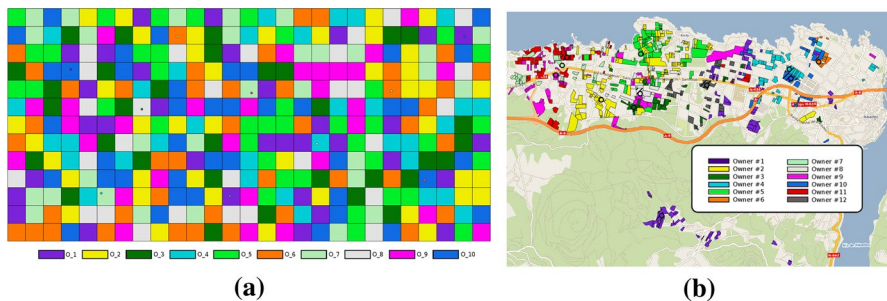


Fig. 3 Testing parcel holdings. Each color represents an owner. **a** Synthetic parcel holding. **b** Real case holding, Ribadeo, Spain

Lastly, to maintain the stagnation stop criteria, the driver program has to keep track of the evolution changes in each stage, since it has no access to the data the worker nodes kept during the evolution. Our Spark approach detects the stagnation when the fitness value does not change in any population after several stages.

5 Experimental results

An analysis of our proposal is presented. This analysis is split into two studies: an analysis for the multithread approach and a performance study for the GeAPaeSp approach.

During the testing procedure, several parcel holdings are used. One of them is synthetic with uniform, adjacent parcels with random landowner assignments for a controlled environment. The other parcel holding is a real test case. Table 2 contains a brief description of the parcel holdings and Fig. 3 shows a graphical representation of the initial distribution where each parcel is color coded by the assigned landowner. Both parcel holdings are color coded by the landowner of each parcel.

The configurations throughout this section (see Figs. 5–8) are named following the format (Q, T) . Q is the number of populations evolving in parallel. T is the total

number of threads in use, each population has an equal amount from the total. The best results are marked with bold numbers in the tables in the following sections.

5.1 Multithreading parallelism

For this analysis, we use a cluster where each node is equipped with two Intel Xeon E5-2660 Sandy Bridge-EP, each one with 8 cores at 3.0 GHz for a total of 16 cores, 64 GB of memory and one 1TB local hard disk drive.

Each configuration was executed ten times, and the average fitness value of the ten executions is displayed in the graphs, measuring progress in a 15-second interval. The population size M is not the same in all cases, Table 3 shows the value used in each configuration. The fitness function used is Average Distances Combined (ADC) with equal importance to parcel distances and reference point distances ($W_{PDA} = W_{RDA} = 0.5$). This fitness function is chosen because it can produce good results using under an hour for the parcel holding used, even in the sequential configuration.

Figure 4 shows the effect on the algorithm results when increasing the number of threads processing a single population. In this graph, the time axis is in logarithmic scale to improve graph visibility, which amplifies the representation of the 15 seconds between datapoints at the start of the graph. We consider a step-before graph over a straight line to never overrepresent the fitness improvement. A reference value called *90% improvement* is showed. The difference between the maximum starting fitness value and the minimum final fitness value is calculated, and it represents the maximum fitness improvement possible. Then, the *90% improvement* value is calculated as the higher starting fitness value minus 90% of the improvement just calculated. This value represents the point where fitness value reduction starts to provide diminishing returns. The user could stop

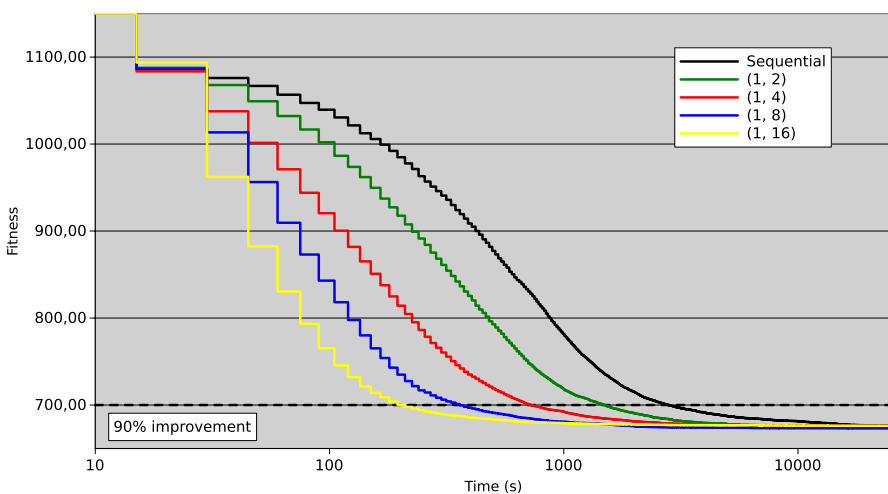


Fig. 4 Multiple threads on one population and a single node

the algorithm once this value is reached, since most of the improvement will have already been made. The conclusions based on speed or time are based on the time to reach this value.

As can be seen, there is a clear reduction in time needed to achieve good fitness values. The efficiency of the parallelization (seen as the ratio of time reduction versus the increase in resources used) goes down as the number of threads increases due to having low computational load for the amount of threads available. Our GeAPae proposal is up to 13.73x faster compared with the sequential version, and a slight improvement in the fitness value is achieved.

Figure 5 shows results when increasing the number of populations using one thread. In this case, as it was expected, a small benefit has been obtained in execution times, because the same amount of work to do has to be done for each population, while the resources used for processing each population are the same as for sequential algorithm. A slightly reduction in execution time is still achieved due to the cooperation between populations; if one population found a good element, it will be shared with the other populations, contributing to a faster achievement of good fitness values. Anyway, this approach is up to 1.71x faster with respect to the utilization of a single population.

Figure 6 shows different configurations varying the number of populations using the maximum number of available threads, 16. To ensure a fair comparison, all the configurations have the same computational load. In those configurations where less number of populations is processed, the size of each one is increased in order to keep the total number of individuals equal to 512 elements, maintaining the overall search width to reduce a variable from the equation. As can be seen on the graph, the first configuration to cross the 90% improvement line, therefore the fastest, is the one with one population of 512 individuals and 16 threads, but it stagnates on slightly higher fitness values. The configuration with four populations of 128 individuals

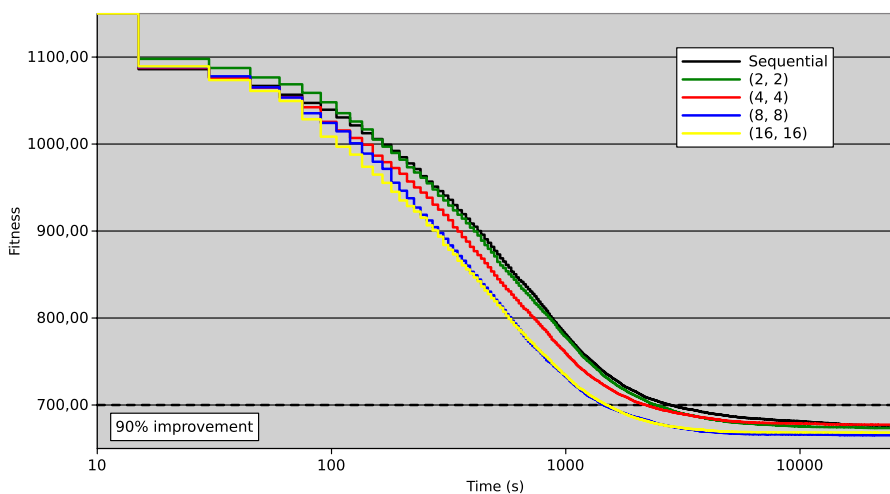


Fig. 5 Multiple populations with one thread

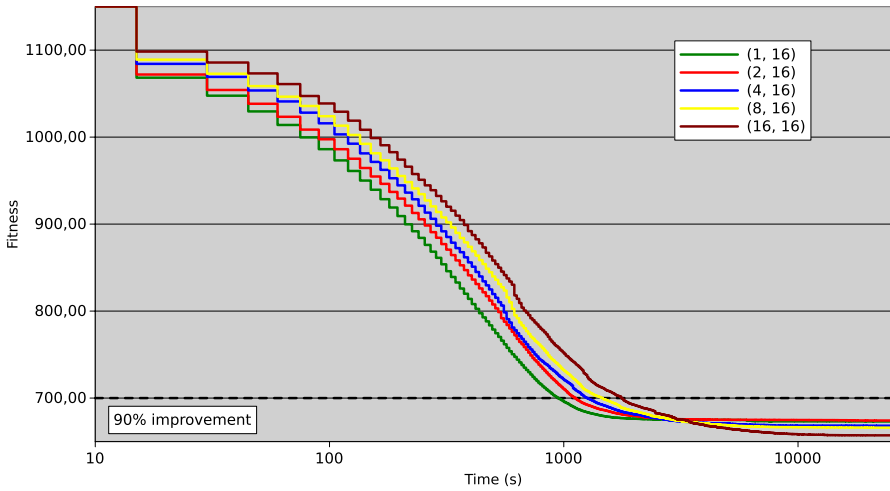


Fig. 6 Different configurations increasing populations using 16 threads

processed each one by four threads presents an average behavior regarding execution time until stagnation, but achieves the best fitness value. As a result, this configuration is a good compromise between speedup and fitness values achieved.

Table 3 shows numeric data for all configurations. Q indicates the number of populations, T is the number of threads processing each population and M is the amount of individuals in each population. For each configuration, the starting and final fitness values, the time needed to achieve 90% of the whole potential improvement and the speedups are displayed. The speedup is calculated with

Table 3 Time to achieve 90% of the potential whole improvement and speedups

Configuration			Final value (100%)	90% improvement time (s)	Speedup
Q	T	M			
1	1	32	676.19	2805	–
1	2	32	673.53	1500	1.87
1	4	32	676.10	720	3.90
1	8	32	673.22	360	7.79
1	16	32	673.22	210	13.36
2	1	32	673.51	2415	1.16
4	1	32	677.39	2205	1.27
8	1	32	665.19	1455	1.93
16	1	32	657.16	1770	1.58
8	2	64	666.01	1455	1.93
4	4	128	668.22	1260	2.23
2	8	256	674.20	1125	2.49
1	16	512	673.10	945	2.97

respect to the $P = 1$ and $T = 1$ configuration. Compared to that configuration, fitness values reductions are up to 4% better and speedups of 13.36 times faster can be achieved.

5.2 Spark distributed parallelism

The following results were taken in a working Spark environment with multiple computation nodes, the testing platform is the same cluster with 17 nodes, in this case, running Spark 3.0.1. Each node consists of two Intel Xeon E5-2660 Sandy Bridge-EP with 8 cores each at 3.0 GHz for a total of 16 cores, 64 GB of memory and one 1TB local hard disk drive.

For this analysis, we use a synthetic parcel holding to have a more controlled environment, with the initial parcel distribution shown in Fig. 3a. The configuration used consists of 128 individuals for population size and fitness function of average parcel distances (PDA) after performing parcel union. This combined with the synthetic parcel holding allows to reach a situation where each owner has one contiguous set of parcels that, after performing the geometry union, will result in one parcel per landowner, resulting in a fitness value of 0. Using that target to measure time is prone to executions not being able to reach it due to being stuck at local minimum, so we will use the 90% improvement like the previous section. In this case, that value is 40, all populations start at values slightly below 400, so the speedup is calculated with the time to reach a value of 40. The fitness value of the original parcel distribution (completely random) is 1918, but the initial individual generation to fill the population is capable of creating better elements with a fitness value around 350, so in reality, the evolution itself starts at that point, not the 1918 of the initial configuration. This generation process is described in more depth in [23]. To increase

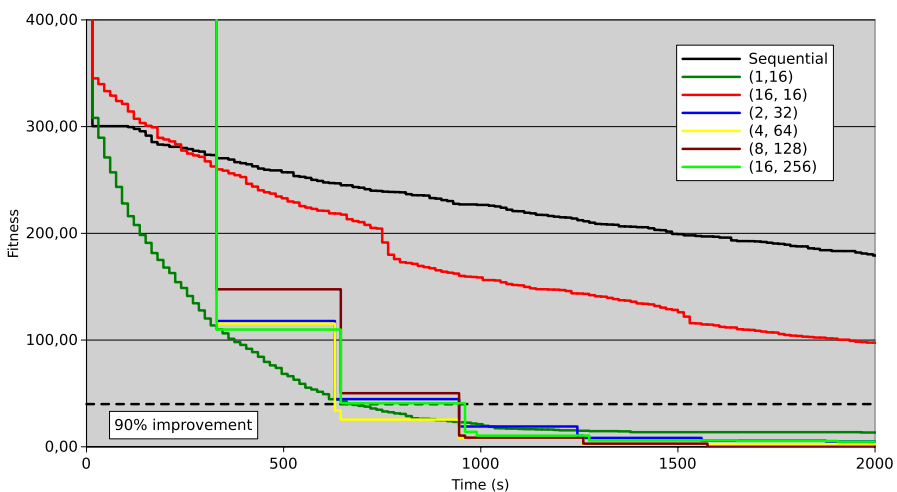


Fig. 7 GeAPaeSp proposal performance using a synthetic parcel holding

the clarity of the graph, the fitness value axis is focused on the $[0, 400]$ range, where most of the data are located.

Figure 7 shows the fitness curves during the evolution (if multiple populations are used, the value of the first population is used) for the synthetic parcel holding. The time axis displayed stops at 2000 seconds to increase readability, only two series have not reached the 90% value by that time, Table 4 has the time they need to reach it.

We test one population with one thread to get a reference, that would be the sequential behavior of the algorithm.

We include configurations (1, 16) and (16, 16) to show the two extremes of the depth-width balance that the Spark implementation solves. Both of those configurations are limited to one node, the user has to choose to increase depth or width with the resources of that node. With the Spark implementation, each node dedicates its resources to maximize search depth, and more nodes process different populations to increase search width maintaining search depth. The configuration (1, 16) uses all available resources for search depth, so it can evolve faster but is prone to being stuck on local minimum, which results in less progress at the end part of the evolution. The configuration (16, 16) only uses one thread per population, all resources are dedicated to width which causes the evolution to be similarly fast as the sequential, but being able to reduce stagnation due to local minimum, achieving the target value of 0 more reliably.

The rest of the configurations show the first advantage of the Spark implementation. We can increase the number of populations (search width) and benefit from the stagnation resistance without losing search depth or speed to reach lower values. The total amount of resources available grows with each population added, executing each one in different nodes of the cluster. The Spark implementation has less granularity with the time measurements. Only after a stage is completed, we can know the evolution progress in the *Driver* program. In this case, the duration of each stage is 300 seconds so there are only around six stages, the end of each corresponds with the big reduction in fitness for the series with more than 16 threads in total.

Table 4 shows the execution time needed for each configuration together with the speedup achieved with respect to the sequential execution. For the configurations

Table 4 Time in seconds to achieve a fitness value of 40 and speedup in the synthetic parcel holding

Configuration		Final value	Execution time	Speedup
Q	T			
1	1	15.31	8985	–
1	16	6.67	675	13.31
16	16	5.05	3015	2.98
2	32	3.14	945	9.51
4	64	0	630	14.26
8	128	0	945	9.51
16	256	0	960	9.36

Table 5 Time in seconds to achieve 90% improvement in fitness value and speedup in the real test case parcel holding

Configuration		Final value	Execution time	Speedup
Q	T			
1	1	730.27	239475	–
1	16	695.23	19575	12.23
16	16	861.30	>87000	<2.75
2	32	677.76	17760	13.48
4	64	676.47	17610	13.60
8	128	681.22	17220	13.91
16	256	668.00	17520	13.67

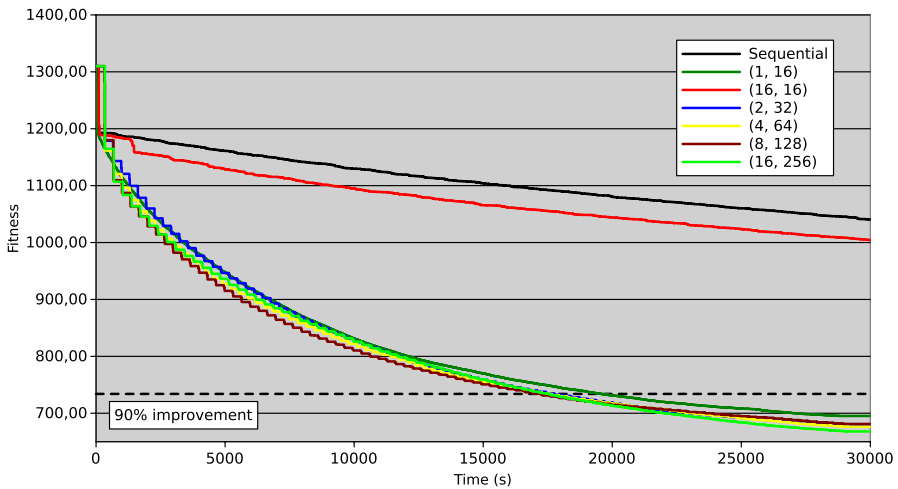


Fig. 8 GeAPaeSp proposal performance using a real case

that use Spark, the time used is the time in which the stage that reached 40 or below finishes. The driver does not know exactly when the 90% threshold value actually was surpassed, but it is the time the results are available to check stopping criteria. Increasing the number of populations allows to reach lower values more reliably, with 4, 8 and 16 populations, all executions reach a value of 0. With less populations, that value is not always reached, and the average is higher than 0. This is evidence for the resilience to local minimum that wider configurations provide. The sequential case reached 0 in 3 out of 10 executions, (1, 16) reached 0 in 5 out of 10 executions, (16, 16) does it in 6 of 10 executions and (2, 32) reached the value 0 in 7 out of 10 executions. The configurations (4, 64), (8, 128) and (16, 256) reach 0 in all executions. Our GeAPaeSp is up to 14.26x faster compared to the sequential version, and the fitness reductions achieved are up to 1.69% better compared to the configuration (1, 16).

Figure 8 shows the fitness evolution of each configuration using the real parcel holding. As with the previous section, we will use the 90% improvement as the target for time measurement for speedup calculations. Table 5 shows the time needed for each configuration together with the speedup achieved with respect to the sequential execution. The configuration used in this test is similar to the one used on Sect. 5.1 but it is not the same, the fitness function in use is different, although the fitness values are very similar, they cannot be directly compared.

This use case requires more execution time, and the stage duration of 300 seconds is more adequate relative to the total execution time. The fitness curves for the Spark configurations differentiate from each other better than in the synthetic parcel holding. In this case, the best configuration is the (8, 128), being 13.91 times faster than the sequential version. Similar to the synthetic parcel holdings results, increasing the number of populations helps to reach lower fitness values, higher search width has that effect, and with this implementation, the search depth is not compromised. The best fitness value reduction was achieved by the configuration (16, 256), outperforming the (1, 16) configuration by 4.43%. The speedup is mostly achieved from dedicating 16 threads to each populations, and the search width helps to reach lower fitness values, combining the two parallelization techniques advantages while mitigating the disadvantages of each one.

To close this section, Fig. 9 displays the result of one execution using the Spark distributed approach with 16 populations on each parcel holding. The synthetic parcel holding (see Fig. 9a) behaves uniquely when using the fitness function PDA together with parcel union. Since all of the parcels are adjacent to each other, one contiguous group of parcels (considering the eight neighbors of each parcel, sharing a corner is considered adjacency) is joined into one parcel, resulting in a fitness value of 0 for the landowner, with no regards to its shape. On real parcel holdings (see Fig. 9b) where there are roads and other parcels to separate the involved parcels, this situation does not happen. Nevertheless, it is easy to see that the new parcel holdings represent an improvement with regards to ownership fragmentation. The synthetic parcel holding execution managed to form one contiguous block of land (touching corners at worst) for each landowner, even with the aforementioned peculiarity. The real parcel holding also

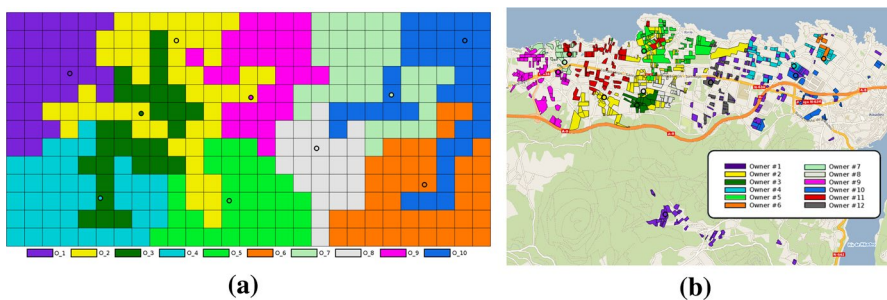


Fig. 9 Result parcel holdings. Each color represents an owner. **a** Synthetic parcel holding. **b** Real case, Ribadeo, Spain

shows a notable improvement in this regard, all the parcels of each landowner are confined to smaller areas in general (see owners #3, #7, #9 and #11 for notable improvements).

6 Conclusions

This paper provides a parallel proposal of a genetic algorithm for parcel exchange using Apache Spark, GeAPaeSp. It should be observed that this proposal is especially well suited to increase the quality of the final results and minimize the execution time. Our proposal improves the width of search increasing the number of populations; and the depth of search increasing the number of generations in a suitable time. GeAPaeSp outperforms a sequential and multithreaded proposal, combining speedups of up to 14.26 with respect to the sequential version and, at the same time, better fitness improvements by up to 10.74% (4.43% comparing against the depth-focused multithreaded proposal), reaching those better fitness values more reliably.

Distributing the computation across multiple computation nodes allows the possibility of using broadly different configurations that cater to the multiple needs of the landowners involved, without hampering any of the other configurations.

There is still work to be done on this front. There are still known improvements to genetic algorithms not implemented, like local searches on selected individuals. Another possibility is dynamic configuration of each population independently to adapt to the progress of the evolution process, using fast methods during the initial stages and changing to more intensive options when those fast methods become stagnant.

Acknowledgements We would like to thank the anonymous reviewers, whose insightful comments greatly improved the quality of this paper. This research was funded by the Ministry of Science and Innovation of Spain (PID2019-104184RB-I00 / AEI / 10.13039/501100011033), by Xunta de Galicia and FEDER funds of the EU under the Centro de Investigación de Galicia accreditation 2019-2022 (ED431G 2019/01), as well as under the Consolidation Program of Competitive Reference Groups (UDC/GI-000265, ref ED431C/2021/30) and one of the authors received financial support from Xunta de Galicia and the European Social Fund (ESF) of the European Union (predoctoral fellowship ref. ED481A-2019/231).

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Data availability The datasets used during the current study were either procedurally generated or created as part of a regional government project and so are not publicly available. Data are however available from the authors upon reasonable request and with permission of the members of the project.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.


References

1. Lu H, Xie H, He Y, Wu Z, Zhang X (2018) Assessing the impacts of land fragmentation and plot size on yields and costs: a translog production model and cost function approach. *Agric Syst* 161:81–88. <https://doi.org/10.1016/j.agry.2018.01.001>
2. Kilgore MA, Snyder SA (2016) Exploring the relationship between parcelization metrics and natural resource managers' perceptions of forest land parcelization intensity. *Landsc Urban Plan* 149:43–48. <https://doi.org/10.1016/j.landurbplan.2016.02.003>
3. van Dijk T (2003) Scenarios of Central European land fragmentation. *Land Use Policy* 20(2):149–158. [https://doi.org/10.1016/S0264-8377\(02\)00082-0](https://doi.org/10.1016/S0264-8377(02)00082-0)
4. Hartvigsen M (2014) Land reform and land fragmentation in Central and Eastern Europe. *Land Use Policy* 36:330–341. <https://doi.org/10.1016/j.landusepol.2013.08.016>
5. Pašakarnis G, Morley D, Maliene V (2013) Rural development and challenges establishing sustainable land use in Eastern European countries. *Land Use Policy* 30(1):703–710. <https://doi.org/10.1016/j.landusepol.2012.05.011>
6. Vranken L, Swinnen J (2006) Land rental markets in transition: Theory and evidence from Hungary. *World Dev* 34(3):481–500. <https://doi.org/10.1016/j.worlddev.2005.07.017>
7. Sklenicka P, Janovska V, Salek M, Vlasak J, Molnarova K (2014) The farmland rental paradox: Extreme land ownership fragmentation as a new form of land degradation. *Land Use Policy* 38:587–593. <https://doi.org/10.1016/j.landusepol.2014.01.006>
8. Borgwardt S, Brieden A, Gritzmann P (2014) Geometric clustering for the consolidation of farmland and woodland. *Math Intel* 36(2):37–44. <https://doi.org/10.1007/s00283-014-9448-2>
9. Bennett DA, Xiao N, Armstrong MP (2004) Exploring the geographic consequences of public policies using evolutionary algorithms. *Ann Assoc Am Geograph* 94(4):827–847. <https://doi.org/10.1111/j.1467-8306.2004.00437.x>
10. Xiao N, Bennett DA, Armstrong MP (2007) Interactive evolutionary approaches to multiobjective spatial decision making: A synthetic review. *Comput Environ Urban Syst* 31(3):232–252. <https://doi.org/10.1016/j.compenvurbsys.2006.08.001>
11. Touriño J, Rivera FF, Álvarez C, Dans CM, Parapar J, Doallo R, Boullón M, Bruguera JD, Crecente R, González XP (2001) COPA: a GIS-based tool for land consolidation projects. In: *ACM-GIS 2001, Proceedings of the Ninth ACM International Symposium on Advances in Geographic Information Systems*, pp. 53–58. <https://doi.org/10.1145/512161.512174>
12. Xin H, Zhi-xia Z (2008) Application of genetic algorithm to spatial distribution in urban planning. In: *IEEE international symposium on knowledge acquisition and modeling workshop*, pp. 1026–1029
13. Vallejo M, Rieser V, Corne DW (2015) Genetic algorithm evaluation of green search allocation policies in multilevel complex urban scenarios. *J Comput Sci* 9:57–63. <https://doi.org/10.1016/j.jocs.2015.04.004>
14. Porta J, Parapar J, Doallo R, Barbosa V, Santé I, Crecente R, Díaz C (2013) A population-based iterated greedy algorithm for the delimitation and zoning of rural settlements. *Comput Environ Urban Syst* 39:12–26. <https://doi.org/10.1016/j.compenvurbsys.2013.01.006>
15. Porta J, Parapar J, Doallo R, Rivera FF, Santé I, Crecente R (2013) High performance genetic algorithm for land use planning. *Comput Environ Urban Syst* 37:45–58. <https://doi.org/10.1016/j.compenvurbsys.2012.05.003>
16. Stewart TJ, Janssen R (2014) A multiobjective GIS-based land use planning algorithm. *Comput Environ Urban Syst* 46:25–34. <https://doi.org/10.1016/j.compenvurbsys.2014.04.002>
17. Liu Y, Tang W, He J, Liu Y, Ai T, Liu D (2015) A land-use spatial optimization model based on genetic optimization and game theory. *Comput Environ Urban Syst* 49:1–14. <https://doi.org/10.1016/j.compenvurbsys.2014.09.002>
18. Santé I, Rivera FF, Crecente R, Boullón M, Suárez M, Porta J, Parapar J, Doallo R (2016) A simulated annealing algorithm for zoning in planning using parallel computing. *Comput Environ Urban Syst* 59:95–106. <https://doi.org/10.1016/j.compenvurbsys.2016.05.005>
19. Akkus MA, Karagoz O, Dulger O (2012) Automated land reallocation using genetic algorithm. *INI-STA 2012 - International Symposium on INnovations in Intelligent Systems and Applications*, 1–5. <https://doi.org/10.1109/INISTA.2012.6247018>

20. Demetriou D, Stillwell J, See L (2012) Land consolidation in Cyprus: Why is an Integrated Planning and Decision Support System required? *Land Use Policy* 29(1):131–142. <https://doi.org/10.1016/j.landusepol.2011.05.012>
21. Uyan M, Cay T, Inceyol Y, Hakli H (2015) Comparison of designed different land reallocation models in land consolidation: A case study in konya/turkey. *Comput Electron Agric* 110:249–258. <https://doi.org/10.1016/j.compag.2014.11.022>
22. Ertunç E, Çay T, Haklı H (2018) Modeling of reallocation in land consolidation with a hybrid method. *Land Use Policy* 76:754–761. <https://doi.org/10.1016/j.landusepol.2018.03.003>
23. Teijeiro D, Rico EC, Porta J, Parapar J, Doallo R (2020) Optimizing parcel exchange among land-owners: A soft alternative to land consolidation. *Comput Environ Urban Syst* 79:101422. <https://doi.org/10.1016/j.compenvurbsys.2019.101422>
24. Zhou C (2010) Fast parallelization of differential evolution algorithm using mapreduce. In: *Proceedings of the 12th Annual Genetic and Evolutionary Computation Conference, GECCO '10*, pp. 1113–1114. <https://doi.org/10.1145/1830483.1830689>
25. Daoudi M, Hamena S, Benmounah Z, Batouche M (2014) Parallel differential evolution clustering algorithm based on mapreduce. In: *2014 6th International Conference of Soft Computing and Pattern Recognition (SoCPar)*, pp. 337–341. <https://doi.org/10.1109/SOCPAR.2014.7008029>
26. Teijeiro D, Pardo XC, González P, Banga JR, Doallo R (2016) Implementing parallel differential evolution on spark. *Applications of Evolutionary Computation*. Springer, Cham, pp 75–90
27. Teijeiro D, Pardo XC, Penas DR, González P, Banga JR, Doallo R (2017) Evaluation of parallel differential evolution implementations on mapreduce and spark. *Euro-Par 2016: Parallel Processing Workshops*. Springer, Cham, pp 397–408

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Diego Teijeiro¹  · Margarita Amor¹ · Ramón Doallo¹ · Eduardo Corbelle² · Juan Porta¹ · Jorge Parapar¹

✉ Diego Teijeiro
diego.teijeiro@udc.es

Margarita Amor
margarita.amor@udc.es

Ramón Doallo
ramon.doallo@udc.es

Eduardo Corbelle
eduardo.corbelle@usc.es

Juan Porta
juan.porta@udc.es

Jorge Parapar
jorge.parapar@udc.es

¹ CITIC, Computer Architecture Group, Universidade da Coruña, Campus de Elviña, 15007 A Coruña, Spain

² Department of Agriculture & Forest Engineering, Universidade de Santiago de Compostela, Santiago, Spain