Parallelism and Partitioning in Large-Scale GAs using Spark

Laila Alterkawi
School of Computing
University of Kent
Management of information System Department
Gulf University for Science and Technology
alterkawi.l@gust.edu.kw

Matteo Migliavacca
School of Computing
University of Kent
m.migliavacca@kent.ac.uk

ABSTRACT

Big Data promises new scientific discovery and economic value. Genetic algorithms (GAs) have proven their flexibility in many application areas and substantial research effort has been dedicated to improving their performance through parallelisation. In contrast with most previous efforts we reject approaches that are based on the centralisation of data in the main memory of a single node or that require remote access to shared/distributed memory. We focus instead on scenarios where data is partitioned across machines.

In this partitioned scenario, we explore two parallelisation models: PDMS, inspired by the traditional master-slave model, and PDMD, based on island models; we compare their performance in large-scale classification problems. We implement two distributed versions of Bio-HEL, a popular large-scale single-node GA classifier, using the Spark distributed data processing platform. In contrast to existing GA based on MapReduce, Spark allows a more efficient implementation of parallel GAs thanks to its simple, efficient iterative processing of partitioned datasets.

We study the accuracy, efficiency and scalability of the proposed models. Our results show that PDMS provides the same accuracy of traditional BioHEL and exhibit good scalability up to 64 cores, while PDMD provides substantial reduction of execution timeat a minor loss of accuracy.

CCS CONCEPTS

• **Computing methodologies** → **Genetic algorithms**; **Parallel algorithms**;

KEYWORDS

Genetic Algorithms, Big Data, Spark, Distributed Learning Classifier System, Distributed Data Mining

ACM Reference Format:
Laila Alterkawi and Matteo Migliavacca. 2019. Parallelism and Partitioning in Large-Scale GAs using Spark. In *Genetic and Evolutionary Computation Conference (GECCO '19), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3321707.3321775

GECCO '19, July 13–17, 2019, Prague, Czech Republic

1 INTRODUCTION

Big Data analytics is becoming increasingly important in the academic and business sectors as it helps to understand scientific phenomena and supports effective decision making. Extracting knowledge from such volumes of data is, however, challenging: as the size of data produced increases rapidly, storage and processing requirements quickly grow beyond the capability of centralised solutions, which become prohibitively expensive, requiring distributed infrastructure. This fostered the development of large-scale cloud computing platforms such as those provided by Amazon and Google, which promise affordable, reliable and scalable storage capacity and processing power.

In addition, scalable analytics requires algorithmic techniques that could benefit from those parallel infrastructures. Evolutionary algorithms, and genetic algorithms (GAs) in particular, are an attractive solution for big data analysis: first they are a flexible search technique to address general optimisation tasks and have been applied successfully to problems in many different disciplines; second their exploration of the solution space is inherently parallel, making them good candidates for execution on parallel architectures.

GAs can find acceptable solutions in a reasonable time, however they may need an exceedingly long time when dealing with large or complex problems. Therefore, there has been substantial effort in enhancing their speed, with GA parallelisation being studied extensively since the 80's leading to many successful implementations capable of reducing the time to obtain good results.

This research work, starting from the 80s, laid down the fundamentals approaches to GA parallelisation, e.g. see surveys by Alba et Al. [1] and Cantu-Paz [4]; however most of these proposals focus on time and quality improvements without considering the case of large data volumes. The frequent assumption is that the training dataset is readily available at each worker node, either through replication or by making use of shared/distributed memory, which is either impractical or inefficient for very large training datasets. A few recent proposals address GA training in the case of distributed datasets, but they are either based on ensemble approaches, such as FlexGP [18] and DXCS [6], or assume frequent changes to the set of worker nodes, typical of volunteer computing such as EC-Star [15].

In this paper, we focus instead on the case where data is partitioned over multiple nodes in a cluster, i.e., that the data is physically split into distinct parts, and all data access for fitness evaluation are restricted inside a partition, while data movement is tightly controlled by the system.

In terms of implementation, most distributed GA systems are examples of specialised solutions, which are either not readily available [15], or not proven to scale to a large number of distributed

nodes. In this paper we exploit Spark, a general-purpose large-scale distributed processing paradigm which can process high data volumes in parallel. With respect to MapReduce, another popular distributed processing framework that has been used to implement parallel GAs, Spark stands out for its more general programming paradigm and efficient parallelisation of iterative jobs such as fitness computation, thus being a good match for a parallel distributed implementation of evolutionary algorithms.

We propose two parallel GA models which target partitioned data processing: Partitioned-Data Master-slave (PDMS) and Partitioned-Data Multiple-deme (PDMD) models, based the traditional Master-slave and Multiple-deme models. In order to test our design, we have selected BioHEL, a centralised GA-classifier and parallelised it using our partitioned-data models. We implemented the resulting classifiers on the Apache Spark platform and compared the two according to performance and accuracy over multiple cluster sizes. Our tests shows that the master-slave model noticeably reduces the training time as more nodes are used, while holding the expected accuracy of the original BioHEL. We also show that multiple-deme model greatly outperforms the master-slave model with respect to speed while maintaining good accuracy as more nodes are used.

The remainder of the paper is organised as follows: Section 2 covers the background in GA parallelisation, Section 3 introduces our GA models, PDMS and PDMD and presents the basics of the Spark system. Section 4 covers BioHEL and its parallel implementations using PDMS and PDMD on Spark, which are then evaluated in Section 5. Finally, Section 6 covers related works and Section 7 concludes the paper, highlighting future research directions.

2 PARALLEL GENETIC ALGORITHMS

Genetic algorithms are a search and optimisation method inspired by natural evolution: a set of candidate solutions (or part thereof) reproduce, evolve and compete for survival similarly to individuals in a population. Success in competition and reproduction is modelled as a fitness function, a measure of the solution quality. Individuals characteristics are encoded in chromosome-like data structures and used for both fitness computation and for the application of genetic operators: crossover, which mixes genetic material between individuals and mutation which randomly mutate individuals.

Parallelisation is key to the success of genetic algorithms: as GAs are applied to problems of increasingly larger scale in business, science and engineering domains, there is a strong drive to reduce the execution time required to obtain good quality solutions. As a consequence, there are several well-established strategies, together with some more recent techniques, to parallelise GAs.

There are three main traditional approaches to GA parallelisation [1]: master-slave (or global), coarse-grained (multiple demes) and fine grained parallelisation. Each model distributes data and computation tasks differently across different workers.

Master-slave model. In the master-slave model all individuals belong to the same population, which is managed by a master node. The master node is responsible for handling the selection and genetic operators, however, it delegates to the workers (slaves) the fitness computation task, i.e. the most expensive operation of the GA. In particular, the master distributes each individual to a worker for the evaluation of the fitness function, thus parallelising its computation: the slaves determine the fitness for the assigned individuals on the training instances. Results are sent back to the master node, and used to select individuals to pass to the next generation. This approach leads to solutions of the same quality of the sequential approach as the fitness evaluation is equivalent to the sequential case and the selection and crossover operate across all individuals in the (panmictic) population.

Multiple-deme model. The multiple-deme model, also known as coarse-grained or island-model, takes GA parallelism a step further: similarly to the master-slave model, individuals are assigned to different nodes for fitness computation, but in addition, all evolutionary steps (selection, crossover, and mutation) are also performed among individuals assigned to the same node, i.e. within the same *sub-population* (*island*). Since the (sub-)population is a subset of the global population, this parallel GA approach converges faster then a serial GA. To promote evolution, individuals are migrated among islands according to a pre-defined policy. Since most of the computation is parallel, multiple-deme models have less synchronisation overhead, resulting in increased performance; however, since evolution is not panmictic the quality of the final solution could differ from sequential GAs.

Other parallelisation models. In the fine-grained model the population is distributed over a large topological mesh where each node hosts one or a few individuals. While fine-grained approaches have been successfully implemented on massively parallel hardware using e.g., MPI or GPGPU techniques, it is challenging to adapt these techniques to a large-data scenario, given the high degree of communication required between the different nodes. As a consequence we do not consider such models here and we leave the exploration of such techniques as a possible venue for future work.

3 PARTITIONED-DATA PARALLEL GA MODELS

Parallel GAs were firstly introduced in the 80s-90s with a focus on exploiting hardware parallelism to improve runtime performance, before very large datasets became common. Parallelism helps in achieving the performance necessary to process large datasets, however, there has been little focus on the performance challenges in accessing large data, with some notable exceptions, e.g., [12]. In this section we first discuss the impact of large data on these models, then we present two simple adaptations to large data scenarios, finally, we introduce Spark, a large scale processing framework that we used in the implementation.

The master-slave and multiple-deme models can achieve substantial speedups with respect to a sequential GA, however, they were designed under the assumption of relatively small datasets. In particular, in both models, each node would require access to the complete dataset to perform fitness evaluations. This can be either implemented by remote data access techniques such as distributed shared memory (DSM) [14] or replication of the training dataset on all nodes [11]. Unfortunately, in the case of very large datasets, the first approach would result in substantial overhead, given that each processor would need to repeatedly access all instances, and the second approach is unfeasible as one machine cannot store the complete dataset.

Our proposed approach for GA parallelisation stems from the following principles: (i) data must be kept in memory for fast access required by GA iterative processing; (ii) data must be distributed, to scale memory storage and access bandwidth, in addition to harness distributed processing power, (iii) processors should be restricted to local data access during GA iterations, and (iv) remote access should be tightly controlled to reduce network contention.

The key strategy for large-scale GA parallelisation is to partition the training dataset in the main memory of the distributed nodes, and structure the computation and data access according to this partitioning. This will result intwo parallelism schemes:

PDMS. The Partitioned-Data Master-Slave model is a typical master-slave model where the master handles the core GA algorithm and the slaves are responsible to compute the fitness for the population's individuals. In contrast to the typical master-slave model, where the master sends different subsets of the population individuals to different workers, in PDMS the master node sends a copy of the complete population to all the slaves and then each slave computes a *partial fitness* for the individuals on the local data partition. Then, slaves send their results to the master which, in turn, aggregates all partial fitness values and use them in the following GA algorithm steps. As the master will be sending the population and collecting the fitness values in every iteration, communication overhead can be a drawback for this model.

It is worth mentioning that in this scenario it must be possible to compute the global fitness by efficiently combining the partial fitness computed on the different partitions, i.e. the fitness function is required to be associative and commutative. In this way by co-locating the computation of the partial fitness with the data in a partition, and later combining the results to compute the global fitness, PDMS would require less communication overhead than providing remote access to the full dataset by each processor to directly compute the global fitness.

PDMD. In the Partitioned-Data Multiple-Deme Model, each node runs the complete GA on the local data partition, i.e. each sub-population is initialised, evaluated and evolved only with respect to the local data of that node. Occasionally, nodes exchange their best individuals with randomly selected nodes to allow interaction among individuals in different populations. Finally, a solution is selected after an individual is globally nominated as the best individual. It is worth to mention that in such implementation as individuals are not evaluated over the complete dataset, their fitness values may not reflect their global quality. We will return to this aspect when discussing the implementation (§ 4) and in the evaluation where we examine PDMD solution quality (§ 5.2).

3.1 Spark

To support the implementation of PDMS and PDMD model in large-scale scenarios, we build on top of Spark [20], a popular cluster computing framework. With respect to MapReduce [8], Spark is a better fit for in-memory iterative computations like those required for GA training. MapReduce has been designed to support massively parallel computations on disk-based datasets which could be efficiently computed with a single pass over the input (e.g. web indexing). Iterative jobs can be implemented as a pipeline of MapReduce jobs which are repeated until convergence, but that would require transferring input/intermediate results from and to an external storage layer (typically a distributed filesystem such as GFS/HDFS) at every iteration. As GAs typically require a high number of iterations to converge this would result in substantial overhead due to disk access latency and throughput.

Spark can provide faster iteration on datasets that fit in the memory of distributed machines. Resilient Distributed Datasets (RDD) are indeed Spark's core abstraction: immutable, distributed, collection of objects. RDDs can be created from stored datasets or from other RDDs by applying a rich set of functional transformations such as `map`, `aggregate`, or `join`. RDDs can then be created to represent input datasets, and intermediate results at different iteration steps can be represented by different RDDs. Developers can control the persistence of RDDs e.g., by caching them in memory thus allowing fast computation of RDDs in the next iteration.

Internally each RDD is partitioned and RDD transformations are designed for efficient parallel computation across partitions: e.g. `filter(pred)` can be executed in parallel in each partition to filter RDD elements, and `aggregate(start, seqOp, combOp)` aggregates elements in each partition first using a `seqOp` function (starting from `start`) and then aggregates partial results from each partition using function `combOp`.

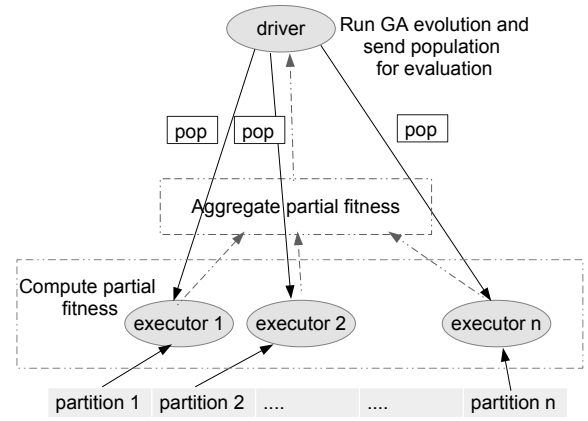
Spark programs are executed in a distributed fashion by a driver and a set of executor processes. The driver executes the serial part of the code and distributes RDD operations tasks to the executors. Partitions of cached RDDs are stored in the executors' memory.

Finally Spark keeps track of RDD lineages: a fault during the computation of an RDD operation would allow to recompute the content of RDD in case of the failure of an executor. Spark allows to checkpoint RDDs to disk to shorten the amount of recomputation necessary to recover from a fault.

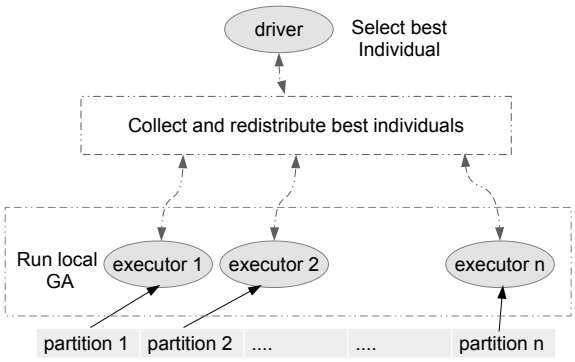
4 IMPLEMENTATION

Like the original master-slave and multiple-deme parallelisation models, the principles of PDMS and PDMD can be applied to any GA algorithm, however the resulting benefits would depend on their reification in a specific parallel GA. To test both accuracy and runtime performance of the data-partitioned approach, we selected BioHEL, a single-node classifier for large datasets, and we turned it into a distributed GA according to the PDMS and PDMD models. In this section we first introduce BioHEL and explain its main workflow, then we show its adaptation according to the two proposed models and their implementation using Spark's primitives.

BioHEL. BioHEL (Bioinformatics-oriented Hierarchical Evolutionary Learning) [2] is a classifier designed to handle large and complex data classification tasks such as protein structure prediction. Algorithm 1 shows its general workflow: BioHEL adopts an iterative rule learning algorithm (IRL) approach [7], which creates a classifier by iteratively learning one rule at a time (lines 1–10) and adding it to a rule list (line 8). Each rule is evolved through a generational GA (`findRules` lines 11–20) evolved from a sample of the training set (line 12). Several learning attempts are repeated with different seeds (lines 4–5) and only the best candidate rule is selected (`bestRule`, lines 21–27). The selected rule is then penalised to induce niche formation in the search space. A common way to penalize the obtained



(a) PDMS



(b) PDMD

Figure 1: Partitioned-Data GA Models.

rules is to delete the training examples that have been covered by the selected rules (line 25). The iterative search for new rules stops when it is no longer possible to find any rule where the associated class is the majority class of the matched examples (lines 23–24).

BioHEL PDMS Implementation. The main goal of the PDMS model is to speed-up the most costly step that is the fitness computation for the population's individuals. All the evolutionary work is handled by the Spark driver which sends the individuals across the executors to compute the fitness. Each executor computes a partial fitness based on the individuals in the local partition, while the master node perform the aggregation.

The PDMS BioHEL implementation follows the main structure of BioHEL, but with specialised `findRulesPDMS` and `bestRulePDMS` functions. The training set is stored as an RDD (tSet, underlined, in Algorithm 2) with instances equally partitioned across executors. In `findRulesPDMS` (lines 1–10) a `sample` primitive is used to derive a sample RDD from tSet, then `collect` is used to retrieve the sample from the executors and make it available to the driver (line 2). The fitness computation is then parallelised using the `aggregate` function (lines 3,8): a copy of the population is transferred to each executor and used in the *partialFitness* function to compute the fitness of the population on the local tSet partition. In BioHEL the

```
1  BioHEL(tSet):
2      ruleList = ∅; stop = false;
3      while stop is false do
4          for repetition=1 to numRepetitions do
5              candidates += findRules(tSet);
6          (bestRule, tSet) = bestRule(candidates, tSet);
7          if bestRule not null then
8              ruleList += bestRule; candidates = ∅;
9          else stop = true;
10     return ruleList;
11 findRules(tSet):
12     pop = tSet.sample(N);
13     pop.fitness = doFitness(pop,tSet);
14     for iter = 1 to numIter do
15         offsp = pop.selection();
16         offsp.crossover();
17         offsp.mutation();
18         offsp.fitness = doFitness(offsp,tSet);
19         pop.replacement(offsp);
20     return pop.best();
21 bestRule(candidates,tSet):
22     bestRule = candidates.bestFitness();
23     matched = tSet.matchedBy(bestRule);
24     if bestRule.class = matched.majorityClass() then
25         tSet = tSet.removeMatched(bestRule);
26         return (bestRule, tSet);
27     else return (null, tSet);
```

Algorithm 1: BioHEL general workflow.

```
1  findRulesPDMS(tSet):
2      pop = tSet.sample(...).collect();
3      pop.fitness = tSet.aggregate(zeroes, partialFitness(pop),
         mergeFitness);
4      for iter = 1 to numIter do
5          offsp = pop.selection();
6          offsp.crossover();
7          offsp.mutation();
8          offsp.fitness=tSet.aggregate(zeroes,
             partialFitness(offsp), mergeFitness);
9          pop.replacement(offsp);
10     return pop.best();
11 bestRulePDMS(candidates,tSet):
12     bestRule = candidates.bestFitness();
13     matchedCl = tSet.filter(_.matches(bestRule))
         .map(_.class).countByValue();
14     if bestRule.class = majorityClass(matchedCl) then
15         tSet = tSet.filter(!_.matches(bestRule));
16         return (bestRule, tSet);
17     else return (null, tSet);
```

Algorithm 2: PDMS BioHEL.

```
1  findRulesPDMD(tSet):
2      pop = ∅; mig = ∅;
3      for migration=1 to numMigrations do
4          pop = tSet.zipPartitions(pop, mig)(
5          (tSetPart, popPart, migPart)⇒
6              if popPart = ∅ then
7                  popPart = tSetPart.sample(n);
8              popPart.replacement(migPart);
9              popPart.fitness=doFitness(popPart, tSetPart);
10             for iter = 1 to numIter/numMig do
11                 offsp = popPart.selection();
12                 offsp.crossover();
13                 offsp.mutation();
14                 offsp.fitness=doFitness(offsp,popPart);
15                 popPart.replacement(offsp);
16             return popPart;
17         );
18         mig =pop.mapPartitions(best(n/2))
            .repartition();
19     return pop.mapPartitions(best(2)).collect();
20 bestRulePDMD(candidates,tSet):
21     candidates.fitness=tSet.aggregate(zeroes,
        partialFitness(candidates), mergeFitness);
22     bestRule = candidates.bestFitness();
23     matchedCl = tSet.filter(_.matches(bestRule))
        .map(_.class).countByValue();
24     if bestRule.class = majorityClass(matchedCl) then
25         tSet = tSet.filter( !_.matches(bestRule));
26         return (bestRule, tSet);
27     else return (null, tSet);
```

Algorithm 3: PDMD BioHEL.

data-dependent part of the fitness function consists in computing the confusion matrix of the rule to be evaluated, thus partialFitness computes four integers for each rule: true-positives, false-positives, true-negatives and false-negatives. Then a *mergeFitness* function is used to aggregate the counters of each rule from each partition, thus obtaining the total fitness.

`bestRulePDMS` (lines 11–17) is then used to select the best rule among repetitions (line 12); and the stopping condition is evaluated by comparing the class of the rule and the most frequent (*majorityClass* line 14) of the classes of the matched instances (*matchedCl*, computed using `countByValue`, `map` and `filter`, line 13). If the rule is accepted a new tSet RDD is created by removing the instances not matched by the selected rule (line 15).

BioHEL PDMD Implementation. In the multiple-deme model, each island is an executor. In addition to the training instances tSet, also the population pop and the incoming migrants mig are represented by RDDs, partitioned across executors. The driver instructs each executor to evolve each sub-population in parallel by invoking the `zipPartitions` function (line 4 in Algorithm 3). `zipPartitions` allows to combine partitions of several RDDs that are colocated on the same executor by providing a user-provided function. This (anonymous) function (lines 5–16) first initialises the population in the local partition (popPart) by taking a sample of size $n = N/\text{numPartitions}$ from the local training set (tSetPart, line 7), then repeats the main evolutionary loop until migration. Finally the population in the local partition popPart is returned (line 16) becoming a partition of pop referenced by the driver (line 4). The driver randomly redistributes the best half of the population in each partition as migrants (line 18) which will be merged with the local population at the next migration (line 8). After *numMigrations* the best two individuals from each partition are collected by the driver as candidate rules (line 19). Candidate rules for every repetition are then evaluated globally (`bestRulePDMD`, line 22), the best is selected and used for termination detection and filtering as before.

Parameter	Value
crossover prob. Rule sets per ensemble	0.6
Selection algorithm	tournament
Tournament size	4
Population size	500
Individual-wise mutation prob.	0.6
Default class policy	MAJOR
Iterations	50
Expected value of #expressed att. in init.	15
Repetitions of rule learning process	2
Prob. generalize	0.1
Prob. specialise	0.1

Table 1: General Parameters of BioHEL.

5 EVALUATION

In our experiments, we aim to investigate the influence of increasing the cluster size on the efficiency of both PDMS and PDMD models in terms of accuracy and runtime performance.

Next, we report our experimental settings, we discuss the results for both PDMS and PDMD scalability, then we report the effect of migration on PDMD accuracy and performance.

5.1 Experimental Setting

In order to train and test the PDMS and PDMD models, we used the HEPMASS dataset (10M instances, 28 attributes, 2 classes) obtained from https://archive.ics.uci.edu/ml/datasets.html, and KDD-cup99 (full version with 4.9M instances, 42 attributes, 23 classes) obtained from https://www.openml.org/d/1110. For the BioHEL configuration, we set the algorithm parameters using the same configuration provided in [3], summarised in Table 1.

We run the experiments on a cluster with 17 servers, each with 2x Intel Xeon E5520 CPUs running at 2.27GHz. Each server has 8 cores and 12GB RAM. In order to test the scalability of the two models, we ran configurations with different numbers of executors with data equally partitioned between them (1 partition per executor), thus resulting in an increasing parallelism level. We report the results for both models using a cluster of 8, 16, 32, 64, and 96 executors, which have been selected according to both memory limitations and the number of processing units available. All results are averaged over

Data	Model	P	R	Accuracy	Time
Hep.	PDMS	8	68 ±5	81.63 ±0.1	279572 ±993
		16	69 ±4	81.62 ±0.1	140285 ±989
		32	72 ±5	81.44 ±0.2	70142 ±494
		64	71 ±5	81.49 ±0.1	32176 ±530
		96	72 ±5	81.40 ±0.2	26553 ±243
KDD		8	29 ±2	99.74 ±0.0	38522 ±700
		16	27 ±2	99.70 ±0.0	16677 ±500
		32	27 ±3	99.70 ±0.0	11494 ±273
		64	26 ±3	99.67 ±0.1	6076 ±130
		96	26 ±3	99.67 ±0.1	5625 ± 70
Hep.	PDMD	8	45 ±1	80.13 ±0.0	15577 ±494
		16	40 ±2	79.92 ±0.2	4403 ±212
		32	40 ±2	79.75 ±0.2	2528 ±144
		64	39 ±3	78.87 ±0.3	1094 ± 94
		96	38 ±3	78.63 ±0.2	788 ± 60
KDD		8	25 ±4	99.69 ±0.0	4558 ±500
		16	21 ±3	99.67 ±0.2	1278 ± 50
		32	20 ±6	99.59 ±0.0	783 ± 39
		64	19 ±3	99.49 ±0.1	310 ± 19
		96	13 ±5	93.26 ±1.0	226 ± 13

Table 2: PDMS and PDMD Scalability. P is number of partitions, R is number of rules.

7 runs using 10-fold cross validation. We report average and 95% confidence intervals.

5.2 Scalability

PDMS. Table 2 shows PDMS results as the parallelism level p is increased. As expected, PDMS maintains a constant accuracy 81% for Hepmass and 99.7% for KDD-cup which is in line with centralised BioHEL. This is due to the fact that, in PDMS, increasing the number of partitions does not impact the accuracy of the fitness computation, thus leading to the same solution quality. Figure 2 shows the speedup with respect to the execution time with 8 partitions, i.e. T_8/T_p. PDMS shows a good scalability in general where the processing time decreases as the cluster size increase. For Hepmass scalability is linear up to 64 cores, and sublinear at 96 cores while for the smaller KDD-Cup it is sublinear starting from 32 cores. The maximum speedup is 11× for 12× cores for Hepmass and 7× for 12× cores for KDD-Cup. This drop in scalability is due to thread contention and synchronisation overhead between the driver and the executors to collect the fitness results.

PDMD. PDMD maintains a relatively good accuracy compared to PDMS. For Hepmass, accuracy decrease of 1% up to 32 partitions, and then settles at around −2.5% at 64 and 96 partitions. The reduction in accuracy is due to the fact that the fitness is computed locally in each partition, thus leading to the discovery of fewer rules and of potential lower quality. For KDD-Cup the accuracy is close to the significance threshold up to 64 partitions but then drops of 6.5% at 96. This increased drop is likely due to the nature of the dataset, which has fewer instances, higher dimensionality and higher number of classes. As seen in Figure 2, PDMD shows good scalability up to 96 cores for both datasets, with a 20× speedup

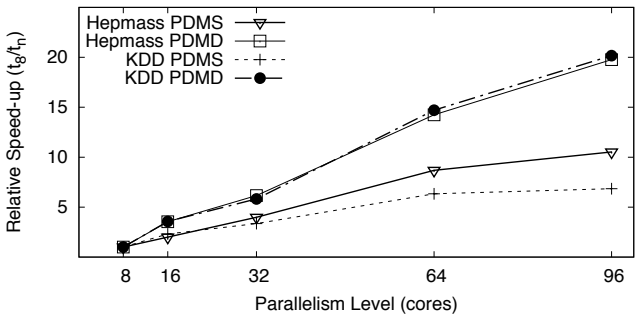
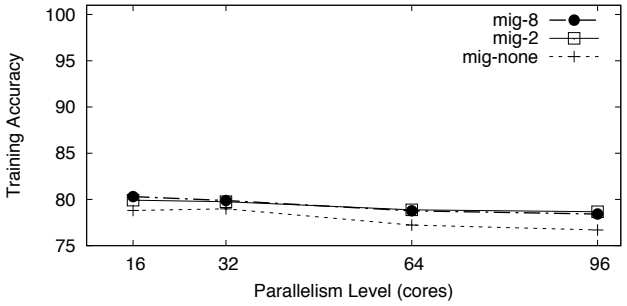
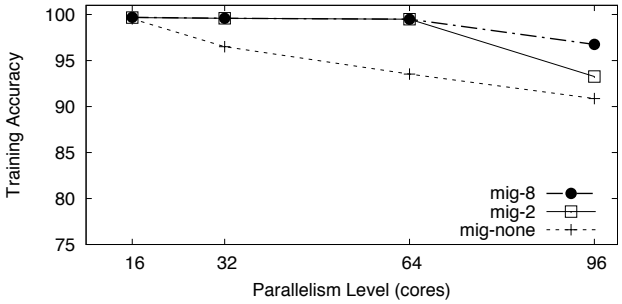


Figure 2: PDMS and PDMD Scalability.



(a) Hepmass



(b) KDD-cup

Figure 3: Impact of migration on PDMD accuracy.

for 12× cores. This super-linear scalability is due to two factors: the reduction in the fitness computation time and the reduction in synchronisation overhead. The fitness computation time decreases quadratically: the size of both the local population and the local training dataset decrease linearly with an increase in p, an effect that is dominant at small p resulting in a quadratic time reduction from 8 to 16 cores; as p increases however other linear factors start to dominate, such as the filtering of the training set characteristic of IRL which improves only linearly with the number of cores. Finally, PDMD is also less affected by synchronisation overhead as islands do not need to synchronise at every fitness evaluation, but only at migration intervals.

Dataset	P	mig-8		mig-2		no-mig	
Hepmass	16	80.30 ±0.2	5851 ±700	79.92 ±0.2	4403 ±420	78.81 ±0.2	2792 ±450
	32	79.88 ±0.2	3454 ±260	79.75 ±0.2	2528 ±144	78.98 ±0.2	1720 ±250
	64	78.77 ±0.3	1453 ±220	78.87 ±0.3	1094 ± 94	77.23 ±0.5	777 ± 70
	96	78.42 ±0.2	1174 ±200	78.67 ±0.2	788 ± 60	76.70 ±0.5	603 ± 30
KDD	16	99.70 ±0.0	2033 ±170	99.67 ±0.2	1278 ±100	99.53 ±0.2	554 ±120
	32	99.59 ±0.2	1126 ±150	99.59 ±0.0	783 ± 70	96.50 ±0.8	499 ± 69
	64	99.49 ±0.2	512 ± 70	99.49 ±0.1	310 ± 35	93.53 ±0.8	173 ± 40
	96	96.75 ±0.8	313 ± 14	93.26 ±1.0	226 ± 26	90.86 ±1.0	91 ± 22

Table 3: Impact of migration on PDMD accuracy and execution time.

5.3 Migration

To study the influence of migration on PDMD accuracy and execution time we tested the impact of a change in the frequency of migrations (*numMigrations*) while performing GA iterations. In addition to the default case, 2 migrations per 50 iterations (`mig-2`), we tested with 8 migrations (`mig-8`) and with no migrations (`no-mig`). We performed experiments for 16, 32, 64, and 96 parallelism levels for both Hepmass and KDD-cup datasets (Table 3).

Accuracy results are shown in Figures 3a and 3b. For Hepmass `mig-2` improves accuracy of 1% for 16 and 32 partitions with respect to `no-mig`, and of 2% for 64 and 96 partitions, while `mig-8` does not improve accuracy further. For KDD-Cup there are no differences at 16 partitions, but at 32 and 64 partitions `mig-2` is able to keep the accuracy close to the maximum, improving accuracy by $4-6\%$ effectively negating the accuracy loss due to increased partitioning; finally at 96 partitions `mig-2` is no longer enough and `mig-8` starts providing benefits improving accuracy of 6% over `no-mig`.

Migration has a significant impact on runtime performance due to synchronisation effects: `mig-8` total execution time is $33-65\%$ higher than the default case, while `no-mig`, results in $23-60\%$ lower execution time.

6 RELATED WORK

In this section, we review proposals for parallel genetic-based machine learning (GBML) and in particular recent proposal that approach parallel GMBL from the perspective of parallel data-processing frameworks such as MapReduce and Spark.

EC-Star is a parallel computing framework which uses distributed Genetic Programming (**GP**) model upon commercial volunteer resources [15]. The model consists of Evolution Coordinators, Evolutionary Engines, and Fitness Case Servers. Volunteer *evolutionary engines* can independently enter or leave the framework at any time, under the supervision of *coordinators*. Engines do not communicate with each other but receive random data packages from *fitness case servers* for fitness evaluation. Individuals are evolved locally and exchanged with the coordinator for further mixing with individuals evolved by other engines. EC-Star is designed for a totally asynchronous scenario, where engines can join or depart from the network in an unpredictable way and is thus difficult to achieve optimal data partitioning across engines, which avoid multiple evaluations of a candidate solutions over a data partition.

DXCS as an instance of distributed XCS data mining system [6]. XCS is a genetic based machine learning algorithm that applies reinforcement learning (**RL**) techniques for rule learning. The DXCS system consists of a number of clients and a single server. Each client runs a complete XCS that is trained independently on its local database. Then clients forward their XCS models with misclassified and untrained instances to the server. The server holds copies of all clients' models and applies a knowledge probing approach [13] to combine the local models. The server combines misclassified and untrained instances and uses them as inputs for all copies of XCS local models available at the server, and then the server trains an XCS to learn the mapping between the output of these local models and the target class. The paper reports experiments with two clients and a server, which shows accuracy that is competitive with a centralised XCS. DXCS uses a fused model approach where several ensemble models are combined into a global model, however, such approach tends to generate models which are more complex to understand than those produced from a direct approach.

Model fusion is also used in Flex-GP, a large-scale genetic programming cloud computing system [18]. It uses EC2 Amazon cloud service as its infrastructure based on an island model implemented on top of ECJ, an EC system written in Java. The experiments are set up to keep a fixed number of individuals per island, thus increasing the global population linearly as islands increase. The goal is to obtain a better accuracy by increasing the use of computation resources obtaining a 40% increase in accuracy for an increase computation cost of $256x$ and a $3x$ increase in latency. The system has been tested up to 350 nodes has been later redesigned to improve scalability [9].

Parallelised GA using MapReduce/Spark. While the above systems are examples of custom parallel GA implementations other recent parallel GA implementations are built on top of large-scale data processing frameworks. MR-GEP [10] is a scalable parallel evolutionary algorithm model based on MapReduce. Authors propose a hybrid model which consists of two layers. The upper layer uses a coarse-grained computational model, while the lower layer uses fine-grained and master-slave model. The population is divided into a number of sub-population equivalent to the number of processors "islands". Each node computes the fitness within a map function. Then the processor applies the genetic operations on its sub-population within a reduce function. The processors use a global shared memory to share intermediate results (population). MR-GEP proved to improve the speed up to 16 nodes.

Verma has addressed the challenge of using the MapReduce model to scale simple and compact genetic algorithms [19]. The algorithms' fitness process is computed using map function and

then the selection of best individuals is done using tournament selection within a reduce function. At the end of every step, the population's individuals are stored on HDFS and two disk accesses are needed for each iteration, impacting performance.

MRPGA extends the MapReduce model with a hierarchical reduction phase [5]. The proposed work uses MapReduce as a distributed parallelised model to parallelise the GA algorithm. Each worker within the system sub-evaluates a set of individuals and applies the GA operations. Each individual represents a complete solution. The system uses three MapReduce phases: i) a local map within the worker is used to partially-evaluate individuals locally; 2) a local reduce select the local optimum individual for this worker; and 3) a global reduce selects the best individuals from all workers. The complete process is repeated until the solution meets the requirement set by the user. The results show that such approach can scale-up the GA up to 20 workers after which the running time settle and adding more workers doesn't speedup the system.

Spark-based solutions benefits from utilising the memory of the cluster to speed up iterative computations. Spark is used in [16] to parallelise a GA for Pairwise Test Suite Generation. The parallelisation process is applied in two-phases: fitness evaluation and genetic operations. The system mainly highlights the advantage of using RDD to partition and apply the parallel spark operations on the partitioned data. The followed procedure uses a map and collect operations for each phase. The model is an example of parallel control model where individuals are partitioned and the fitness and genetic operations are applied in parallel on these partitions. However, in this case the computation of the fitness function does not depend on a large training set and thus this strategy is not appropriate for our target scenario.

GenRBFN [17] is a Spark implementation of an Evolutionary Computation algorithm to develop Radial Basis Function Networks (RBFN). The goal of the work is to find the optimal network which minimizes classification error and complexity. The implementation uses map and reduce operations to compute the fitness for the individuals. In the map phase every instance is evaluated by the model and the result is compared with the real output, return 1 if they are similar or 0 if they are different and in the reduce steps results are added. GenRBFN was tested with up to four partitions, with linear speedup.

7 CONCLUSION

We presented two partitioned data models for parallel GA, PDMD and PDMS, to address big data classification problems. We used the two proposed models to reimplement BioHEL, a popular large-scale single-node GA classifier, using the Spark distributed data processing platform under the assumption of data partitioning. Our results show that the training time reduces as parallelisation level increase. The PDMS scales-up linearly up to 64 nodes while overhead starts to negatively affect the training speed when the cluster size increases to 96 nodes. The PDMD model is substantially faster than PDMS while maintaining relatively good accuracy. In our experiments we found that two-migrations per rule enhance the PDMD accuracy while maintaining good time performance.

In the future research, we plan to study the enhancement of PDMD model using different migration methods. Moreover, we plan to study the influence of data filtering on accuracy and time performance and therefore we will consider implementing other GA strategies using the same environment settings. We also plan to investigate the applicability of data-partitioned models beyond classification tasks, e.g., to general GAs, GP and other machine learning techniques.

REFERENCES

[1] Enrique Alba and José M. Troya. 1999. A Survey of Parallel Distributed Genetic Algorithms. *Complex.* 4, 4 (March 1999), 31–52. https://doi.org/10.1002/(SICI)1099-0526(199903/04)4:4<31::AID-CPLX5>3.0.CO;2-4

[2] Jaume Bacardit, Edmund K. Burke, and Natalio Krasnogor. 2009. Improving the scalability of rule-based evolutionary learning. *Memetic Computing* 1, 1 (01 Mar 2009), 55–67. https://doi.org/10.1007/s12293-008-0005-4

[3] Jaume Bacardit and Natalio Krasnogor. 2006. BioHEL : Bioinformatics-oriented Hierarchical Evolutionary Learning.

[4] Erick Cantú-Paz. 1998. A Survey of Parallel Genetic Algorithms. *CALCULATEURS PARALLELES* 10 (1998).

[5] Rajkumar Buyya Chao Jin, Christian Vecchiola. 2008. MRPGA: an extension of MapReduce for parallelizing genetic algorithms. In *2008 IEEE Fourth International Conference on eScience*. Indianapolis, IN, USA, 214–221. https://doi.org/10.1109/eScience.2008.78

[6] Hai Huong Dam, Hussein A. Abbass, and Christopher J. Lokan. 2005. DXCS: an XCS system for distributed data mining. In *GECCO*.

[7] Kenneth A. De Jong and William M. Spears. 1991. Learning Concept Classification Rules Using Genetic Algorithms. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 2 (IJCAI'91)*. 651–656. http://dl.acm.org/citation.cfm?id=1631552.1631559

[8] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[9] Owen Derby, Kalyan Veeramachaneni, and Una-May O'Reilly. 2013. Cloud Driven Design of a Distributed Genetic Programming Platform. In *Applications of Evolutionary Computation*, Anna I. Esparcia-Alcázar (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 509–518.

[10] Xin Du, Youcong Ni, Zhiqiang Yao, Ruliang Xiao, and Datong Xie. 2013. High performance parallel evolutionary algorithm model based on MapReduce framework. *IJCAT* 46 (2013), 290–295.

[11] Henri E. Bal, M Frans Kaashoek, Andrew Tanenbaum, and Jack Jansen. 1992. Replication Techniques For Speeding Up Parallel Applications On Distributed Systems. *Concurrency: Practice and Experience* 4 (08 1992). https://doi.org/10.1002/cpe.4330040502

[12] Alex A. Freitas and S. H. Lavington. 1997. *Mining Very Large Databases with Parallel Processing* (1st ed.). Kluwer Academic Publishers, Norwell, MA, USA.

[13] Yike Guo, Stefan M. Rüger, Janjao Sutiwaraphun, and Jodie Forbes-millott. 1997. Meta-Learning for Parallel Data Mining. In *In Proceedings of the Seventh Parallel Computing Workshop*. 1–2.

[14] Alan Judge, Paddy Nixon, Brendan Tangney, and Stefan Weber. 1998. *Overview of Distributed Shared Memory*. Technical Report.

[15] Una-May O'Reilly, Mark Wagy, and Babak Hodjat. 2013. *EC-Star: A Massive-Scale, Hub and Spoke, Distributed Genetic Programming System*. Springer New York, New York, NY, 73–85. https://doi.org/10.1007/978-1-4614-6846-2_6

[16] Rongzhi Qi, Zhi-Jian Wang, and Shui-Yan Li. 2016. A Parallel Genetic Algorithm Based on Spark for Pairwise Test Suite Generation. *Journal of Computer Science and Technology* 31 (03 2016), 417–427.

[17] AJ Rivera, MD Pérez-Godoy, F Pulgar, and MJ del Jesus. [n. d.]. GenRBFNSpark: A first implementation in Spark of a genetic algorithm to RBFN design. ([n. d.]).

[18] Dylan Sherry, Kalyan Veeramachaneni, James McDermott, and Una-May O'Reilly. 2012. Flex-GP: Genetic Programming on the Cloud. In *Applications of Evolutionary Computation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 477–486.

[19] Abhishek Verma, Xavier Llorà, David E. Goldberg, and Roy H. Campbell. 2009. Scaling Genetic Algorithms Using MapReduce. In *Proceedings of the 2009 Ninth International Conference on Intelligent Systems Design and Applications (ISDA '09)*. IEEE Computer Society, Washington, DC, USA, 13–18. https://doi.org/10.1109/ISDA.2009.181

[20] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX, San Jose, CA, 15–28.