Using Spark and GraphX to Parallelize Large-Scale Simulations of Bacterial Populations over Host Contact Networks

Andreia Sofia Teixeira[1,2(✉)], Pedro T. Monteiro[1,2], João A. Carriço[3], Francisco C. Santos[1,2], and Alexandre P. Francisco[1,2]

[1] INESC-ID Lisboa, Lisboa, Portugal
[2] Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal
sofia.teixeira@tecnico.ulisboa.pt
[3] Faculdade de Medicina, Instituto de Microbiologia
and Instituto de Medicina Molecular, Universidade de Lisboa, Lisboa, Portugal

Abstract. Large-scale population genetics studies are fundamental for phylogenetic and epidemiology analysis of pathogens. And the validation of both evolutionary models and methods used in such studies depend on large data analysis. It is, however, unrealistic to work with large datasets as only rather small samples of the real pathogen population are available. On the other hand, given model complexity and required population sizes, large-scale simulations are the only way to address this issue. In this paper we study how to efficiently parallelize such extensive simulations on top of Apache Spark, making use of both the MapReduce programming model and the GraphX API. We propose a simulation framework for large bacterial populations, over host contact networks, implementing the Wright-Fisher model. The experimental evaluation shows that we can effectively speedup simulations. We also evaluate inherent parallelism limits, drawing conclusions on the relation between cluster computing power and simulations speedup.

Keywords: Population genetics · Large-scale simulations · Graph-parallel computations · Spark · GraphX

1 Introduction

Understanding bacterial population genetics is vital for interpreting the response of bacterial populations to selection pressures, as antibiotic treatment or vaccines targeted at only a subset of strains [15]. In this context, large-scale studies are fundamental not only for such understanding, but also for validating both models and methods, such as phylogenetic inference algorithms. But we cannot validate them with real pathogen populations as only rather small population samples are available and accessible. Hence, given model complexity and required population sizes, large-scale simulations are the only way to address this issue.

© Springer International Publishing AG 2017
S. Ibrahim et al. (Eds.): ICA3PP 2017, LNCS 10393, pp. 591–600, 2017.
DOI: 10.1007/978-3-319-65482-9_44

Previous studies have shown that observed population genetic structure of several important human pathogens, such as *Streptococcus pneumoniae* and *Neisseria meningitidis*, can be explained using a simple evolutionary model [6–9]. This model is based on neutral mutational drift and modulated by recombination, but incorporating the impact of epidemic transmission only for panmictic populations. Although this simple evolutionary model works well for local populations, at a "microepidemic" level, its predictions no longer seem to fit observed genetic relationships of large and widely distributed bacterial populations. With the increasing volume of data obtained with sequence based typing methods, namely by Multi-Locus Sequence Typing (MLST) [12], currently the gold standard for epidemiological surveillance, a much more complex pattern emerges, that cannot be explained solely by the simple "microepidemic" assumption.

The evolution of transmissible bacteria occurs by mutation and recombination, and is influenced by epidemiological as well as molecular processes. These aspects are fundamental in the process of strain diversification [16], and as a mechanism by which strains acquire virulence factors or resistance determinants [13]. On the other hand, bacterial population evolution is also influenced by the environment and by host contact networks, through which bacterial populations spread. The study of the impact of host contact network topologies, and associated transmission ratios, on bacterial population evolution and genetic diversity becomes then relevant, increasing the model complexity.

Large-scale simulations are, however, computationally demanding, in particular when model complexity increases. In this work, considering an extension of the above simple evolutionary model by incorporating the underlying host contact network, we propose a simulation framework for large bacterial populations, implementing the Wright-Fisher model [17], on top of Apache Spark [21] and making use of both MapReduce programming model and GraphX API [19]. Then we discuss how such large simulations can benefit from parallelization. We conducted several experiments on Google Cloud Platform and results show that we can effectively speedup simulations. We also evaluate inherent parallelism limits, drawing conclusions on the relation between cluster computing power and simulations speedup.

The paper is organized as follows. We describe the simulation and computational models in Sect. 2. Implementations details are discussed in Sect. 3. Finally, we present and discuss experimental evaluation results in Sect. 4.

2 Simulation and Computational Models

The simulation framework proposed in this paper allows us to observe the evolution of a bacterial population through a host contact network, while parametrizing different simulation aspects. We focus on bacterial population genetics where isolates are represented as MLST profiles. MLST is a technique in which DNA sequences are obtained for a set of housekeeping loci, and different sequences at each locus are assigned as different alleles [12]. From an abstract point of view, each isolate/pathogen is just characterized through a profile that may be subject

to transformations along time, under the influence of genetic events, environment and host contact networks.

Let each strain in a bacterial population be then characterized by a MLST profile, where each sequence type (ST), or MLST profile, is defined by the combination of its alleles, a vector of labels, where different labels mean different alleles. Given an underlying host contact network, we simulate a bacterial population at each host, or vertex, with a neutral evolutionary model [5]. This model is based on a previous null model for evolutionary change, the neutral infinite alleles model (IAM) [10]. Under IAM, mutation always generates a new allele, leading to new STs. Recombination, on the other hand, introduces an existing allele randomly selected from the isolates present in the previous generation, which may lead to novel allelic profiles, or to the reappearance of existing ones. Mutation or recombination occur independently, with each event being rare and mutation taking precedence over recombination. When a new ST is produced, it is given a new ST number, and the parental ST is recorded. For recombination, the allele donor is also recorded.

We assume non-overlapping generations and, at each step of evolution, a new generation is obtained. The probability of an ST to occur in the next generation is proportional to its frequency in the current generation after migration among hosts. The interactions between hosts, over the network, takes place by allowing pathogens to migrate from one vertex to another accordingly to some frequency and edge transmission probabilities, defined by the user, followed by selection at each host through sampling with replacement from the current host generation.

The model just described, based on the IAM, is also known as the Wright-Fisher model [17]. Neutral evolution means that all individuals have the same fitness. Fitness, in population genetics, is a measure of the expected number of offspring. In the neutral Wright-Fisher model, equal fitness is implemented by equal probabilities for all individuals to be picked as a parent.

We rely on Apache Spark [21] and GraphX [19] to parallelize our simulations. Apache Spark came up with an extended MapReduce model that enables the creation of iterative programs, maintaining the scalability and fault tolerance of MapReduce, through its Resilient Distributed Datasets (RDD) [20]. MapReduce [11] is a high-level programming model originally proposed by Google [4] and it was designed to address embarrassingly parallel data processing problems using cheap commodity hardware. Apache Hadoop is probably the best well known MapReduce open source implementation, on top of which Apache Spark is built. Apache Spark provides new levels of abstraction exceeding some limitations of Apache Hadoop and a high-level API, usable through several programming languages such as Scala, Java or Python. It provides also the GraphX API [19] to address graph-parallel problems, relying also on RDDs. We will rely on both the MapReduce programming model and the GraphX API.

The MapReduce programming model has two main steps: map and reduce. The map phase processes the input as key-value pairs and applies a user defined function to each one, generating a set of intermediate key-value pairs. The reduce phase takes those intermediate key-value pairs, aggregate them by key and then

apply a user defined function to the values, merging them, and generating new key-value pairs.

When thinking about applying the MapReduce programming model for solving a problem, since many mappers and reducers run in parallel, and the distributed file system is a shared global resource, special care must be taken to ensure that such operations avoid synchronization conflicts. It is then important to analyze some basic requirements. Taking a careful look into the model described above, we observe that the problem is composed by two main tasks: (1) the evolution of each population at each node where mutation and recombination take place, and (2) the exchanges between nodes and the replacement of each population taking into account the samples of the populations from the neighbourhoods.

Even with each node having its own population, and hence being able to evolve independently from one another, one must take into consideration the fact that each mutation requires the creation of a new allele, because of the IAM, demanding a unique global identifier. Thus, for the first task we find two challenges: (a) mutation process assumes that there is a central memory/database for generating/requesting new alleles; (b) recombination demands that the populations of each node must be in the same memory space—if we are to recombine an allele we must choose, randomly, other ST to provide an already existing allele, i.e., to be its parent. To avoid implementing a shared database, one solution is modifying how a new allele is generated and identified. If we guarantee that at each mutation a new allele is created and the individual gets a unique identifier, then we can have different populations evolving at the same time in independent nodes. For solving both (a) and (b) problems, we had to create a new way to identify uniquely each ST and also to make sure that after the first mapping task we group each population at the same memory space, at the cost of losing some efficiency of the MapReduce model.

For the second task, each node just has to receive a sample of the current population of its neighbours, mix with its own population, and create a new population through sampling. Besides being a simple problem when thinking about the MapReduce model, we can identify some similarities between this process and PageRank [14] or Label Propagation [22] problems. Both rely on exchanging information among neighbours to update their state, and both have already been implemented using MapReduce and, in particular, making use of GraphX. Hence we will use GraphX and a similar approach for this second task.

3 Implementation

Let $G = (V, E)$ be a connected and weighted graph, with $n = |V|$ vertices and $m = |E|$ edges, and with an edge transmission probability function $w : E \to \mathbb{R}$. Let t be the number of times that the sequence of number of evolutions followed by the number of exchanges happens.

The simulator takes eight parameters: (1) the population size of each node; (2) the file containing the populations; (3) the file containing the network;

(4) mutation rate; (5) recombination rate; (6) number of evolutions; (7) number of exchanges; (8) number of times for the cycle (6) followed by (7) to happen; (9) frequency of writing on disk. Each iteration of evolution is considered one generation.

The general workflow consists in iterating t times the following two steps: (i) perform the sequence of evolutions for each population, and (ii) perform the sequence of exchanges between nodes.

The interaction between host populations is as follows: each host copies a given proportion of its population, corresponding to the edge transmission probability $w(u, v)$, and sends it to each corresponding neighbour; each host receives the amount of population sent by the neighbours and create a pool with those bacteria mixed with its own population; a new population is built, with the same size as the previous one, but the individuals are chosen randomly from the pool.

3.1 Using MapReduce with Spark and GraphX

Adapting an algorithm into a MapReduce programming model implies some modifications in the way the input is processed and how the data is managed through the *Map* and *Reduce* phases. Frameworks based on this model, as Hadoop or Apache Spark, read the input from a file in HDFS, where each line is a pair $\langle key, value \rangle$.

We rely on Apache Spark not only because of its flexibility and easiness of use, but also because RDDs allow to develop iterative programs in a light manner. These RDDs are fault-tolerant, parallel data structures that make it possible to persist intermediate results in memory, manage how they are partitioned to optimize data placement among workers, and provide a rich set of operators to apply on data processing. Apache Spark has also an embedded API, the GraphX API, that allows the user to work with graphs in a transparent manner. With GraphX, graph-parallel and data-parallel computations are possible with a single composable API. Graphs can be viewed as collections (RDDs) without data duplication and one can attribute properties to the vertices or edges through the PropertyGraph. It also allows to access vertices, edges, or both (triplets) separately.

Let us not discuss how the two main tasks defined before can be parallelized. The first input file contains the bacterial population of all the host contact network. This file has an individual per line in which the *key* corresponds to a vertex identifier to which it belongs, and the *value* corresponds to the sequence of its alleles, and also a global unique identifier. This global unique identifier is needed to guarantee that, at each mutation event, a completely new individual is generated, as demanded in IAM. We must also modify the characterization of each allele. Although in traditional MLST data we have an array of integers, in our approach each individual is characterized by an array of strings in the form $X.Y.Z$ where X is the id of the node where the mutation occurs, Y is the generation id, and Z is the individual unique identifier. With this change we guarantee the requirements of the IAM regarding allele uniqueness on mutation events. The second input file is the network file. This file contains an edge list

Listing 1.1. Evolutionary process.

```
val nextpop = populationRDD.map{ i =>
    val idpop = i._1
    val population = i._2
    for (each indidivual in population){
        //Recombintation | //Mutation
        population.update(i, individual)}
    (idpop, population)}
populationRDD = nextpop
```

with the transmission probabilities: source, destination and the probability of transmission $w(source, destination)$. If the network is undirected the edge list must contain edges in both directions. Another requirement is that each node must have an edge to itself with a probability of transmission 1.0. The reason is explained later on. We load the input files with the *SparkContext.textFile* method, which maps the inputs into RDDs. With this method we are able to explicitly define how data should be partitioned.

Evolutions in each population and exchanges between the nodes can be implemented as independent *Map* and *Reduce* tasks as follows. For the evolution process, after loading the input into an RDD, we use the ($groupByKey$) transformation to guarantee that each population is in the same memory space. As explained before, recombination demands elements from the same population to recombine. This transformation, when called on a dataset of (K, V) pairs, returns a dataset of $(K, Iterable\langle V\rangle)$ pairs. With this we have each population gathered in the same data structure. This is the major drawback of approach, we can not have each individual evolving independently and *groupByKey* uses shuffle operations that have some costs. After this, the approach is straightforward. Having the populations RDD, we do the *Map* transformation to make each population evolve in parallel (Listing 1.1).

In what concerns the exchange process we have an explicit notion of graph structure. For the nodes (populations) to communicate with their neighbours we use the GraphX API. By making use of the GraphX API and its PropertyGraph, we can use both the populations RDDs and the network (edge list) RDD to create the graph.

After the PropertyGraph is created, we use the *triplets* view to access all $EdgeTriplet[V, E]$ that contain information about the source and destination vertices, with their properties (populations), and also information about the edge and its property (transmission probability). With this we have all the information needed to generate the samples from each source node to each destination node. Regarding the fact that each node must have an edge to itself with a probability of 1.0 associated, this is necessary for the *reduceByKey* operation. After the samples are generated and emitted, the *reduceByKey* function is applied on the values with the concatenation operator (++) that joins all the collections with the same key, providing a collection with all the elements to be sampled for the

Listing 1.2. Exchanging process.

```
val result = graphpopulation.triplets.map{ t =>
    //generate a collection with the proportion
    //of population to send
    (dest, fractionofpopulation)
}.reduceByKey(_ ++ _).map{ i =>
    //create the new populations by sampling
    (key, newpopulation)}
```

new population: its own population and the samples sent from the connected populations. Each new population is then generated through sampling with a *Map* transformation (Listing 1.2).

4 Results and Discussion

Apache Spark can run either in local mode or in a cluster. We relied on both for our experiments to compare how running time scales, parametrizing simulations with different population sizes and different graphs. Experiments were conducted in a cluster hosted at Google Cloud Platform[1], configured with different number of workers: 2, 4, 8 and 16. Each worker is an Intel(R) Xeon(R) CPU @ 2.50 GHz with 4 cores and 16 GB of RAM, where only 2 cores were usable by Apapche Spark. To achieve the best results possible, in each experiment we partitioned the input in as many partitions as the number of the cores available. The local setup is equivalent to a worker node.

Given the evolution model coupled with a host contact network, leading to both evolutions and interactions between populations, the operations in local mode, with each step being executed sequentially, take considerable time. And it scales reasonably well when in cluster mode. For a precise comparison, in what concerns time scale, we fixed almost all parameters of the simulator: the size of the population per node is 1000; the mutation rate is 0.001; the recombination rate is 0.01; the number of evolutions per time step is 25; the number of exchanges per time step is 1; the number of time steps is 10; the frequency for writing on disk is 10 generations; and the transmission probability is 0.01. This means that each dataset will evolve in a cycle of 25 evolutions followed by one exchange between nodes 10 times. We also write on disk every time we have exchanges to track how STs are transversing the network. This information is relevant in most biological analyses, which are however out of the scope of this paper.

The first observation is that writing on disk is the operation that takes an higher cost. We address this issue by using the *saveAsTextFile* method available in Apache Spark, exploiting the ability of parallelize write operations on HDFS.

[1] https://cloud.google.com/.

Table 1. Running time in seconds for different topologies and network sizes.

Topology	Size (N)	Mode				
		Local	2 workers	4 workers	8 workers	16 workers
Clique	100	170	60	53	48	43
	200	411	109	74	59	50
	500	1052	338	202	118	97
	1000	2538	796	485	267	169
Scale-free	1000	916	480	403	221	152
	2000	1807	897	519	341	292
	8000	6293	3020	1902	1300	912

We consider two different network topologies, cliques (or fully connected networks) and scale-free networks [2]. Clique topology leads to highest computational cost while performing exchanges. Scale-free networks are more realistic for host contact networks, but being very sparse lead to much less work during exchanges. We consider random scale-free networks generated with a partial duplication model, with parameter 0.5 and different number of vertices [3]. Tests were run for both topologies and different network sizes. The results averaged over 10 runs are presented in Table 1 and in Fig. 1.

We note that we used a graph partitioning schema available in GraphX designed specifically for scale-free networks. The *PartitionStrategy.EdgePartition2D* was proposed by Verma *et al.* [18] as the best for these

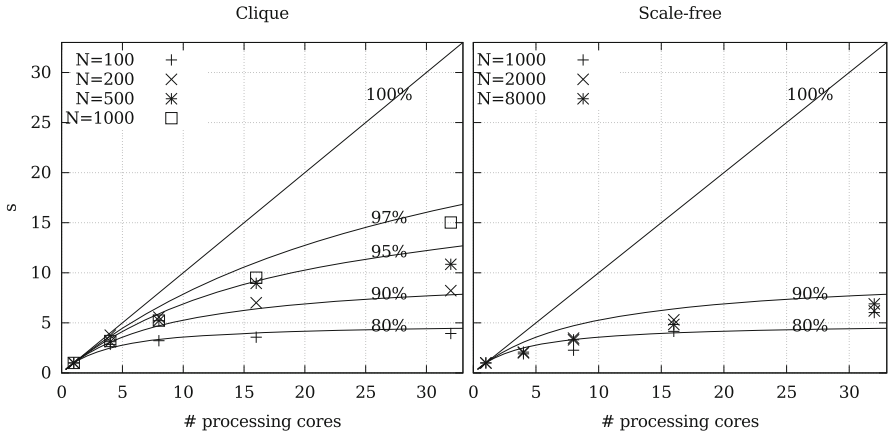

Fig. 1. Speedup as a function of the number of available cores for cliques (left) and scale-free networks (right), with different network sizes (N). The curves are provided by Amdahl's law [1], where the percentage corresponds to the fraction that is infinitely parallelizable.

networks. And we confirmed their results while running our experiments and comparing with other partitioning strategies.

We can observe that the speedup becomes more evident as networks grow in size. Running in cluster mode, and increasing the number of workers, results in a significantly boost in the running time. The fact that we can compute the population evolution at each node independently seems to be exploited as expected. The same happens with the exchange process among nodes populations.

When designing parallel algorithms, one of the analyses that should be done is to estimate the relation between achievable speedups and the number of workers. Amdahl's law is crucial here to both interpret the results and project expected speedups [1]. On one hand it points out that one should only optimize if the fraction that can be optimized constitutes a large portion of the overall time. On the other hand, if the optimization is effective the speedup we obtain is largely determined by the strictly sequential fraction, which cannot be optimized and initially constituted only a small fraction of the time. Given k workers (cores) and a program that spends a fraction f of time on operations that are infinitely parallelizable, and the remaining fraction $1-f$ on strictly sequential operations, the overall speedup is given by $1/((1-f)+f/m)$. In Fig. 1 we can observe that, as we increase the size of the clique networks, we obtain a higher speedup as we increase the number of workers. According to Amdahl's law, we are observing about $f = 97\%$ for a clique of 1000 nodes. For the scale-free networks, because they are much sparser than cliques, with less work performed on exchange, we observe about $f = 90\%$ and speedups are small above 16 processing cores. This seems to point out that the exchange process is highly parallelizable, independently of the network size, which is an important observation if simulations with much larger networks are desirable. We should also note that the running time grows almost linearly as we increase the number of nodes (see Table 1). This is expected as the evolution of each population can be done independently and, even not being fully parallelized due to recombinations as explained before, we can still benefit from parallelization in simulations over large networks.

As a final remark, we must note that since most real host contact networks are scale-free, according to Amdahl's law, using more than 16 processing cores does not lead to significant improvements for realistic simulations.

Acknowledgments. This work was partly supported by DEI, IST, Universidade de Lisboa, and national funds through FCT – Fundação para a Ciência e Tecnologia, under projects TUBITACK/0004/2014, LISBOA-01-0145-FEDER-016394, PTDC/EEISII/5081/2014, PTDC/MAT/STA/3358/2014, and UID/CE C/500021/2013.

References

1. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the Spring Joint Computer Conference, AFIPS 1967 (Spring), pp. 483–485. ACM, 18–20, April 1967
2. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science **286**(5439), 509–512 (1999)

3. Chung, F., Lu, L., Dewey, T.G., Galas, D.J.: Duplication models for biological networks. J. Comput. Biol. **10**(5), 677–687 (2003)
4. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
5. Fraser, C., Hanage, W., Spratt, B.: Neutral microepidemic evolution of bacterial pathogens. PNAS **102**(6), 1968–1973 (2005)
6. Fraser, C., Alm, E.J., Polz, M.F., Spratt, B.G., Hanage, W.P.: The bacterial species challenge: making sense of genetic and ecological diversity. Science **323**(5915), 741–746 (2009)
7. Fraser, C., Hanage, W.P., Spratt, B.G.: Neutral microepidemic evolution of bacterial pathogens. Proc. Natl. Acad. Sci. U.S.A. **102**(6), 1968–1973 (2005)
8. Fraser, C., Hanage, W.P., Spratt, B.G.: Recombination and the nature of bacterial speciation. Science **315**(5811), 476–480 (2007)
9. Hanage, W.P., Spratt, B.G., Turner, K.M., Fraser, C.: Modelling bacterial speciation. Philos. Trans. Roy. Soc. Lond. B: Biol. Sci. **361**(1475), 2039–2044 (2006)
10. Kimura, M.: Evolutionary rate at the molecular level. Nature **217**, 624–626 (1968)
11. Lin, J., Dyer, C.: Data-Intensive Text Processing with MapReduce. Morgan and Claypool Publishers (2010)
12. Maiden, M., Bygraves, J., Feil, E., Morelli, G., Russell, J., Urwin, R., Zhang, Q., Zhou, J., Zurth, K., Caugant, D., et al.: Multilocus sequence typing: a portable approach to the identification of clones within populations of pathogenic microorganisms. PNAS **95**(6), 3140–3145 (1998)
13. Ochman, H., Lawrence, J.G., Groisman, E.A.: Lateral gene transfer and the nature of bacterial innovation. Nature **405**, 299–304 (2000)
14. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999–66, Stanford InfoLab (1999)
15. Robinson, D.A., Falush, D., Feil, E.J.: Bacterial Population Genetics in Infectious Disease. John Wiley & Sons, Hoboken (2010)
16. Spratt, B.G., Hanage, W.P., Feil, E.J.: The relative contributions of recombination and point mutation to the diversification of bacterial clones. Curr. Opin. Microbiol. **4**(5), 602–606 (2001)
17. Tran, T.D., Hofrichter, J., Jost, J.: An introduction to the mathematical structure of the Wright-Fisher model of population genetics. Theory Biosci. **132**(2), 73–82 (2013)
18. Verma, S., Leslie, L.M., Shin, Y., Gupta, I.: An experimental comparison of partitioning strategies in distributed graph processing. Proc. VLDB Endow. **10**(5), 493–504 (2017)
19. Xin, R.S., Gonzalez, J.E., Franklin, M.J., Stoica, I.: Graphx: a resilient distributed graph system on spark. In: First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, pp. 2:1–2:6. ACM (2013)
20. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI 2012, p. 2. USENIX Association (2012)
21. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud 2010, p. 10. USENIX Association (2010)
22. Zhu, X., Ghahramani, Z.: Learning from labeled and unlabeled data with label propagation (2002)

View publication stats