Scaling Genetic Algorithms using MapReduce

Abhishek Verma†, Xavier Llorà∗, David E. Goldberg# and Roy H. Campbell†
{verma7, xllora, deg, rhc}@illinois.edu

†Department of Computer Science
∗National Center for Supercomputing Applications (NCSA)
#Department of Industrial and Enterprise Systems Engineering
University of Illinois at Urbana-Champaign, IL, US 61801

Abstract—Genetic algorithms (GAs) are increasingly being applied to large scale problems. The traditional MPI-based parallel GAs require detailed knowledge about machine architecture. On the other hand, MapReduce is a powerful abstraction proposed by Google for making scalable and fault tolerant applications. In this paper, we show how genetic algorithms can be modeled into the MapReduce model. We describe the algorithm design and implementation of GAs on Hadoop, an open source implementation of MapReduce. Our experiments demonstrate the convergence and scalability up to $10^5$ variable problems. Adding more resources would enable us to solve even larger problems without any changes in the algorithms and implementation since we do not introduce any performance bottlenecks.

Keywords—Genetic Algorithms, MapReduce, Scalability

I. INTRODUCTION

The growth of the internet has pushed researchers from all disciplines to deal with volumes of information where the only viable path is to utilize data-intensive frameworks [29], [1], [5], [22]. Genetic algorithms are increasingly being used for large scale problems like non-linear optimization [7], clustering [6] and job scheduling [24]. The inherent parallel nature of evolutionary algorithms makes them optimal candidates for parallelization [2]. Although large bodies of research on parallelizing evolutionary computation algorithms are available [2], there has been little work done in exploring the usage of data-intensive computing [19].

The main contributions of the paper are as follows:

- We demonstrate a transformation of genetic algorithms into the map and reduce primitives
- We implement the MapReduce program and demonstrate its scalability to large problem sizes.

The organization of the paper is as follows: We introduce the MapReduce model and its execution overview in Section II. Then, we discuss how genetic algorithms can be modeled using the MapReduce model in Section III and report our experiments in Section IV. In Section V, we discuss and compare with the related work and finally conclude with Section VI.

II. MAPREDUCE

Inspired by the map and reduce primitives present in functional languages, Google proposed the MapReduce [3] abstraction that enables users to easily develop large-scale distributed applications. The associated implementation parallelizes large computations easily as each map function invocation is independent and uses re-execution as the primary mechanism of fault tolerance.

In this model, the computation inputs a set of key/value pairs, and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: Map and Reduce. Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce framework then groups together all intermediate values associated with the same intermediate key $I$ and passes them to the Reduce function. The Reduce function, also written by the user, accepts an intermediate key $I$ and a set of values for that key. It merges together these values to form a possibly smaller set of values. The intermediate values are supplied to the user’s reduce function via an iterator. This allows the model to handle lists of values that are too large to fit in main memory.

Conceptually, the map and reduce functions supplied by the user have the following types:

$$map(k_1, v_1) \rightarrow list(k_2, v_2)$$
$$reduce(k_2, list(v_2)) \rightarrow list(v_3)$$

i.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

The Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of $M$ splits. The input splits can be processed in parallel by different machines. Reduce invocations are distributed by partitioning the intermediate key space into $R$ pieces using a partitioning function, which is $hash(key)\%R$ according to the default Hadoop configuration (which we later override for our needs). The number of partitions ($R$) and the
partitioning function are specified by the user. Figure 1 shows the high level data flow of a MapReduce operation. Interested readers may refer to [3] and Hadoop\(^1\) for other implementation details. An accompanying distributed file system like GFS \([8]\) makes the data management scalable and fault tolerant.

### III. MAPREDUCING GAS

In this section, we start with a simple model of genetic algorithms and then transform and implement it using MapReduce along with a discussion of some of the elements that need to be taken into account. We encapsulate each iteration of the GA as a separate MapReduce job. The client accepts the commandline parameters, creates the population and submits the MapReduce job.

#### A. Genetic Algorithms

Selecto-recombinative genetic algorithms \([10], [11]\), one of the simplest forms of GAs, mainly rely on the use of selection and recombination. We chose to start with them because they present a minimal set of operators that help us illustrate the creation of a data-intensive flow counterpart. The basic algorithm that we target to implement as a data-intensive flow can be summarized as follows:

1. Initialize the population with random individuals.
2. Evaluate the fitness value of the individuals.
3. Select good solutions by using s-wise tournament selection without replacement \([12]\).
4. Create new individuals by recombining the selected population using uniform crossover\(^2\)[28].
5. Evaluate the fitness value of all offspring.
6. Repeat steps 3–5 until some convergence criteria are met.

\[^1\]http://hadoop.apache.org

\[^2\]We assume a crossover probability \(p_c=1.0\).

#### B. Map

Evaluation of the fitness function for the population (Steps 2 and 5) matches the MAP function, which has to be computed independent of other instances. As shown in the algorithm in Algorithm 1, the MAP evaluates the fitness of the given individual. Also, it keeps track of the the best individual and finally, writes it to a global file in the Distributed File System (HDFS). The client, which has initiated the job, reads these values from all the mappers at the end of the MapReduce and checks if the convergence criteria has been satisfied.

**Algorithm 1** Map phase of each iteration of the GA

\[
\begin{align*}
\text{MAP}(\text{key}, \text{value}): \\
\text{individual} & \leftarrow \text{INDIVIDUAL REPRESENTATION(key)} \\
\text{fitness} & \leftarrow \text{CALCULATE FITNESS(individual)} \\
\text{EMIT} (\text{individual, fitness}) \\
\{\text{Keep track of the current best}\} & \\
\text{if} \text{ fitness} > \text{max} \text{ then} \\
\text{max} & \leftarrow \text{fitness} \\
\text{maxInd} & \leftarrow \text{individual} \\
\text{end if} \\
\text{if all individuals have been processed then} & \\
\text{Write best individual to global file in DFS} & \text{end if}
\end{align*}
\]

#### C. Partitioner

If the selection operation in a GA (Step 3) is performed locally on each node, spatial constraints are artificially introduced and reduces the selection pressure \([25]\) and can lead to increase in the convergence time. Hence, decentralized and distributed selection algorithms \([16]\) are preferred. The only point in the MapReduce model at which there is a global communication is in the shuffle between the Map and Reduce. At the end of the Map phase, the MapReduce framework shuffles the key/value pairs to the reducers using the partitioner. The partitioner splits the intermediate key/value pairs among the reducers. The function GETPARTITION() returns the reducer to which the given \((\text{key, value})\) should be sent to. In the default implementation, it uses HASH(\text{key}) \% numReducers so that all the values corresponding to a given key end up at the same reducer which can then apply the REDUCE function. However, this does not suit the needs of genetic algorithms because of two reasons: Firstly, the HASH function partitions the namespace of the individuals \(N\) into \(r\) distinct classes : \(N_0, N_1, \ldots, N_{r-1}\) where \(N_i = \{n : \text{HASH}(n) = i\}\). The individuals within each partition are isolated from all other partitions. Thus, the HASHPARTITIONER introduces an artificial spatial constraint based on the lower order bits. Because of this,
the convergence of the genetic algorithm may take more iterations or it may never converge at all.

Secondly, as the genetic algorithm progresses, the same (close to optimal) individual begins to dominate the population. All copies of this individual will be sent to a single reducer which will get overloaded. Thus, the distribution progressively becomes more skewed, deviating from the uniform distribution (that would have maximized the usage of parallel processing). Finally, when the GA converges, all the individuals will be processed by that single reducer. Thus, the parallelism decreases as the GA converges and hence, it will take more iterations.

For these reasons, we override the default partitioner by providing our own partitioner, which shuffles individuals randomly across the different reducers as shown in Algorithm 2.

Algorithm 2 Random partitioner for GA

```java
int GET_PARTITION(key, value, numReducers):
    return RANDOMINT(0, numReducers - 1)
```

D. Reduce

We implement Tournament selection without replacement [9]. A tournament is conducted among $S$ randomly chosen individuals and the winner is selected. This process is repeated population number of times. Since randomly selecting individuals is equivalent to randomly shuffling all individuals and then processing them sequentially, our reduce function goes through the individuals sequentially. Initially the individuals are buffered for the last rounds, and when the tournament window is full, SELECTION AND CROSSOVER is carried out as shown in the Algorithm 3. When the crossover window is full, we use the Uniform Crossover operator. For our implementation, we set the $S$ to 5 and crossover is performed using two consecutively selected parents.

E. Optimizations

After initial experimentation, we noticed that for larger problem sizes, the serial initialization of the population takes a long time. According to Amdahl’s law, the speedup is bounded because of this serial component. Hence, we create the initial population in a separate MapReduce phase, in which the MAP generates random individuals and the REDUCE is the Identity Reducer. We seed the pseudorandom number generator for each mapper with $mapperid \times current\_time$. The bits of the variables in the individual are compactly represented in an array of long long ints and we use efficient bit operations for crossover and fitness calculations. Due to the inability of expressing loops in the MapReduce model, each iteration consisting of a Map and Reduce, has to executed till the convergence criteria is satisfied.

IV. RESULTS

The ONE MAX Problem [27] (or BitCounting) is a simple problem consisting in maximizing the number of ones of a bitstring. Formally, this problem can be described as finding an string $\vec{x} = \{x_1, x_2, \ldots, x_N\}$, with $x_i \in \{0, 1\}$, that maximizes the following equation:

$$F(\vec{x}) = \sum_{i=1}^{N} x_i \quad (1)$$

We implemented this simple problem on Hadoop (0.19) and ran it on our 416 core (52 nodes) Hadoop cluster. Each

3Setting the number of reducers to 0 in Hadoop removes the extra overhead of shuffling and identity reduction.

4http://hadoop.apache.org
node runs a two dual Intel Quad cores, 16GB RAM and 2TB hard disks. The nodes are integrated into a Distributed File System (HDFS) yielding a potential single image storage space of $2 \cdot 52/3 = 34.6TB$ (since the replication factor of HDFS is set to 3). A detailed description of the cluster setup can be found elsewhere$^5$. Each node can run 5 mappers and 3 reducers in parallel. Some of the nodes, despite being fully functional, may be slowed down due to disk contention, network traffic, or extreme computation loads. Speculative execution is used to run the jobs assigned to these slow nodes, on idle nodes in parallel. Whichever node finishes first, writes the output and the other speculated jobs are killed. For each experiment, the population for the GA is set to $n \log n$ where $n$ is the number of variables.

We perform the following experiments:

1) **Convergence Analysis:** In this experiment, we monitor the progress in terms of the number of bits set to 1 by the GA for a $10^4$ variable ONE MAX problem. As shown in Figure 2, the GA converges in 220 iterations taking an average of 149 seconds per iteration.

2) **Scalability with constant load per node:** In this experiment, we keep the load set to 1,000 variables per mapper. As shown in Figure 3, the time per iteration increases initially and then stabilizes around 75 seconds. Thus, increasing the problem size as more resources are added does not change the iteration time. Since, each node can run a maximum of 5 mappers, the overall map capacity is $5 \cdot 52(nodes) = 260$. Hence, around 250 mappers, the time per iteration increases due to the lack of resources to accommodate so many mappers.

3) **Scalability with constant overall load:** In this experiment, we keep the problem size fixed to 50,000 variables and increase the number of mappers. As shown in Figure 4, the time per iteration decreases

$^5$http://cloud.cs.illinois.edu
as more and more mappers are added. Thus, adding more resources keeping the problem size fixed decreases the time per iteration. Again, saturation of the map capacity causes a slight increase in the time per iteration after 250 mappers. However, the overall speedup gets bounded by Amdahl’s law introduced by Hadoop’s overhead (around 10s of seconds to initiate and terminate a MapReduce job). However, as seen in the previous experiment, the MapReduce model is extremely useful to process large problems size, where extremely large populations are required.

4) **Scalability with increasing the problem size:** Here, we utilize the maximum resources and increase the number of variables. As shown in Figure 5, our implementation scales to \( n = 10^5 \) variables, keeping the population set to \( n \log n \). Adding more nodes would enable us to scale to larger problem sizes. The time per iteration increases sharply as the number of variables is increased to \( n = 10^5 \) as the population increases super-linearly \((n \log n)\), which is more than 16 million individuals.

V. DISCUSSION OF RELATED WORK

Several different models like fine grained [21], coarse grained [18] and distributed models [17] have been proposed for implementing parallel GAs. Traditionally, Message Passing Interface (MPI) has been used for implementing parallel GAs. However, MPIs do not scale well on commodity clusters where failure is the norm, not the exception. Generally, if a node in an MPI cluster fails, the whole program is restarted. In a large cluster, a machine is likely to fail during the execution of a long running program, and hence efficient fault tolerance is necessary. This forces the user to handle failures by using complex checkpointing techniques.

MapReduce [3] is a programming model that enables the users to easily develop large-scale distributed applications. Hadoop is an open source implementation of the MapReduce model. Several different implementations of MapReduce have been developed for other architectures like Phoenix [23] for multicore and CGL-MapReduce [4] for streaming applications.

To the best of our knowledge, MRPGA [15] is the only attempt at combining MapReduce and GAs. However, they claim that GAs cannot be directly expressed by MapReduce, extend the model to MapReduceReduce and offer their own implementation. We point out several shortcomings: Firstly, the Map function performs the fitness evaluation and the “ReduceReduce” does the local and global selection. However, the bulk of the work - mutation, crossover, evaluation of the convergence criteria and scheduling is carried out by a single co-ordinator. As shown by their results, this approach does not scale above 32 nodes due to the inherent serial component. Secondly, the “extension” that they propose can readily be implemented within the traditional MapReduce model. The local reduce is equivalent to and can be implemented within a Combiner [3]. Finally, in their mapper, reducer and final reducer functions, they emit “default_key” and 1 as their values. Thus, they do not use any characteristic of the MapReduce model - the grouping by keys or the shuffling. The Mappers and Reducers might as well be independently executing processes only communicating with the co-ordinator.

We take a different approach, trying to hammer the GAs to fit into the MapReduce model, rather than change the MapReduce model itself. We implement GAs in Hadoop, which is increasingly becoming the de-facto standard MapReduce implementation and used in several production environments in the industry. Meandref[20], [19] extends beyond some limitations of the MapReduce model while maintaining a data-intensive nature. It shows linear scalability of simple GAs and EDAs on multicore architectures. For very large problems (> \( 10^9 \) variables), other models like compact genetic algorithms(cGA) and Extended cGA(eCGA) have been explored[26].

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we have mainly addressed the challenge of using the MapReduce model to scale genetic algorithms. We described the algorithm design and implementation of GAs on Hadoop. The convergence and scalability of the implementation has been investigated. Adding more resources would enable us to solve even larger problems without any changes in the algorithm implementation.

MapReducing more scalable GA models like compact GAs [14] and extended compact GAs [13] will be investigated in future. We also plan to compare the performance with existing MPI-based implementations. General Purpose GPUs are an exciting addition to the heterogeneity of clusters. The compute intensive Map phase and the random number generation can be scheduled on the GPUs, which can be performed in parallel with the Reduce on the CPUs. We would also like to demonstrate the importance of scalable GAs in practical applications.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their valuable feedback. This research was funded, in part, by NSF IIS Grant #0841765. The views expressed are those of the authors only.

REFERENCES


