SCALING SIMPLE, COMPACT AND EXTENDED COMPACT GENETIC ALGORITHMS USING MAPREDUCE

BY

ABHISHEK VERMA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Master's Committee:

Professor Roy H. Campbell

Abstract

Data-intensive computing has emerged as a key player for processing large volumes of data exploiting massive parallelism. Data-intensive computing frameworks have shown that terabytes and petabytes of data can be routinely processed. However, there has been little effort to explore how data-intensive computing can help scale evolutionary computation. We present a detailed step-by-step description of how three different evolutionary computation algorithms, having different execution profiles, can be translated into the MapReduce paradigm. Results show that (1) Hadoop is an excellent choice to push evolutionary computation boundaries on very large problems, and (2) that transparent linear speedups are possible without changing the underlying data-intensive flow thanks to its inherent parallel processing.

To Father and Mother.

Acknowledgments

This project would not have been possible without the support of many people. Many thanks to my adviser, Prof. Roy H. Campbell, who gave me direction and helped make some sense of the confusion. Also thanks to Prof. David Goldberg for rejuvenating my interest in Genetic algorithms. This work would not have been possible without the initiative of Xavier Llorà. Also, thanks to Shivaram Venkataramani for helping in the design and implementation of these algorithms. And finally, thanks to my parents and numerous friends for always offering support and love.

Table of Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

The current data deluge is happening across different domains and is forcing a rethinking of how large volumes of data are processed. Most data-intensive computing frameworks [30, 7] share a common underlying characteristic: data-flow oriented processing. Availability of data drives, not only the execution, but also the parallel nature of such processing. The growth of the internet and its easy communication medium has pushed researchers from all disciplines to deal with volumes of information where the only viable path is to utilize data-intensive frameworks [41, 4, 10, 29]. Although large bodies of research on parallelizing evolutionary computation algorithms are available [5, 1], there has been little work done in exploring the usage of data-intensive computing [25].

The inherent parallel nature of evolutionary algorithms makes them optimal candidates for parallelization [5]. Moreover, as we will layout in this thesis, evolutionary algorithms and their inherent need to deal with large volumes of data, regardless if it takes the form of populations of individuals or samples out of a probabilistic distribution, can greatly benefit from a data-intensive computing modelling. In this thesis, we will explore the usage Yahoo!'s Hadoop model and its MapReduce implementation.

Inspired by the *map* and *reduce* primitives present in functional languages, Google proposed the MapReduce [8] 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:

$$\begin{aligned} map(k_1, v_1) &\rightarrow list(k_2, v_2) \\ reduce(k_2, list(v_2)) &\rightarrow list(v_3) \end{aligned}$$

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 1.1 shows the high level data flow of a MapReduce operation. Interested readers may refer to [8] and Hadoop[1] for other implementation details. An accompanying distributed file system like GFS [11] makes the data management scalable and fault tolerant.

To illustrate the benefits for the evolutionary computation community of adopting such approaches we selected three representative algorithms and developed their equivalent MapReduce implementations. It is important to note here that we paid special attention to guarantee that the underlying mechanics were not altered and the properties of these algorithms maintained. The three algorithms transformed were: a simple selecto-recombinative genetic algorithm [13, 14], the compact genetic algorithm [18], and the extended compact genetic algorithm [19]. We will show how a simple selecto-recombinative genetic algorithm [13, 14] can be modelled using the data-intensive computing via Hadoop's MapReduce approach. We will review (1) some of the basic steps of the transformation process required to achieve its data-intensive computing counterparts, (2) how to design components that can maximally benefit from a data-driven execution, and (3) analyse the results obtained. The second example, the compact genetic algorithm [18], we focus on how Hadoop's MapReduce modelling can help scale being a clear competitor of traditional high performance computing version [37]. The third example addresses the parallelization of the model building of estimation of distribution algorithms. We will show how MapReduce's data-driven implementation of the extended compact classifier system (eCGA) [19] produces, *de facto*, a parallelized implementation of the costly model building stage.

It is important to note here, that each of these algorithms has different profiles. For instance, the simple selecto-recombinative genetic algorithm requires dealing with large populations as you tackle large

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

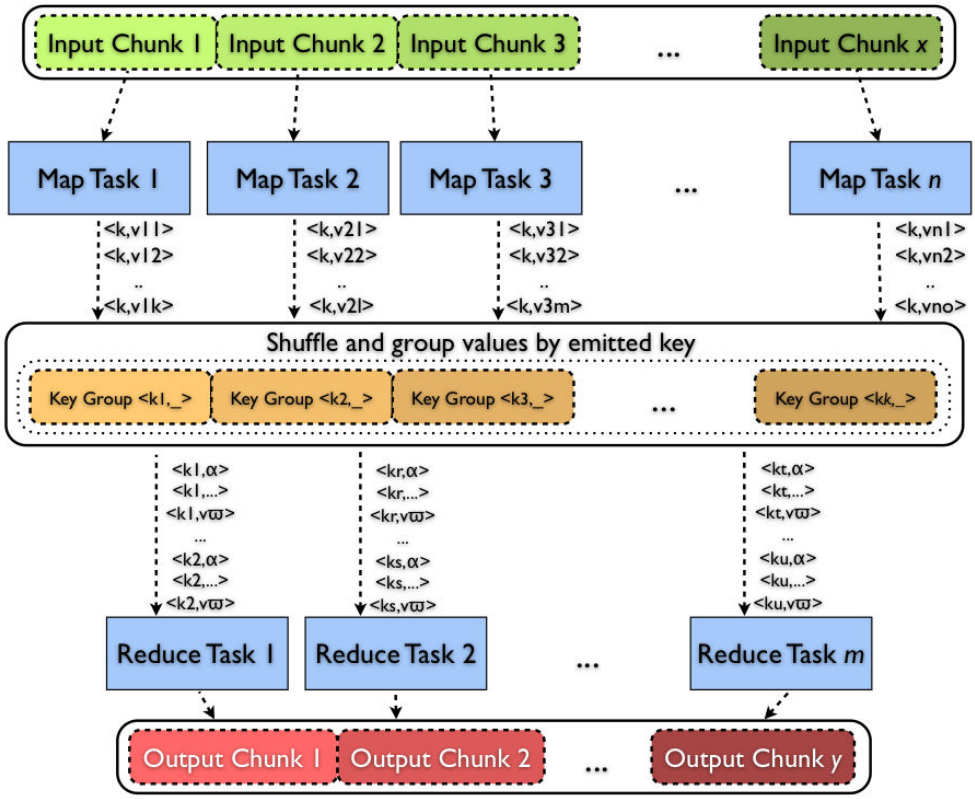

Figure 1.1: MapReduce Data flow overview

problems, but the operators are straight forward. The compact genetic algorithm instead is memory efficient, but requires the proper updating of a simple probability distributions. Finally the extended compact genetic algorithms requires to deal with large populations as you scale your problem size, and also requires an elaborated model building process to induce the probability distribution required. We will focus on the massive parallel data-driven execution that allows users to automatically benefit from the advances of the current multi-core era — which has opened the door to peta-scale computing — without having to modify the underlying algorithm.

The rest of this thesis is organized as follows: Chapter 2 introduces the three evolutionary computation algorithms that we will use in our experimentation with the two introduced frameworks, a simple selecto-recombinative genetic algorithm, the compact genetic algorithm, and the extended compact genetic algorithm. These algorithms are transformed and implemented using data-intensive computing techniques, and the proposed implementations are discussed on Chapter 3. Chapter 4 presents the results achieved and using the data-intensive implementations showing that scalability is only bounded by the available resources, and linear speed-ups are easily achievable. Finally we review some related work in Chapter 5 and present some conclusions and possible further work in Chapter 6.

The work presented in this thesis has been published in these papers [42, 27, 43].

Chapter 2

Genetic Algorithms

In this chapter, we describe the simple genetic algorithm, followed by compact and extended compact genetic algorithm.

2.1 Genetic Algorithms

Selecto-recombinative genetic algorithms [13, 14], one of the simplest forms of GAs, mainly rely on the use of selection and recombination. 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 [16]. In this step, s random individuals from the population are chosen, and the best individual is chosen as the winner and used in the next step.
4. Create new individuals by recombining the selected population using uniform crossover[1] [40].
5. Evaluate the fitness value of all offspring.
6. Repeat steps 3–5 until some convergence criteria are met.

2.2 The Compact Genetic Algorithm

The compact genetic algorithm [18], is one of the simplest estimation distribution algorithms (EDAs) [33, 22]. Similar to other EDAs, CGA replaces traditional variation operators of genetic algorithms by building a probabilistic model of promising solutions and sampling the model to generate new candidate solutions. The

[1] We assume a crossover probability p_c=1.0.

probabilistic model used to represent the population is a vector of probabilities, and therefore implicitly assumes each gene (or variable) to be independent of the other. Specifically, each element in the vector represents the proportion of ones (and consequently zeros) in each gene position. The probability vectors are used to guide further search by generating new candidate solutions variable by variable according to the frequency values.

The compact genetic algorithm consists of the following steps:

1. *Initialization:* As in simple GAs, where the population is usually initialized with random individuals, in CGA we start with a probability vector where the probabilities are initially set to 0.5. However, other initialization procedures can also be used in a straightforward manner.

2. *Model sampling:* We generate two candidate solutions by sampling the probability vector. The model sampling procedure is equivalent to uniform crossover in simple GAs.

3. *Evaluation:* The fitness or the quality-measure of the individuals are computed.

4. *Selection:* Like traditional genetic algorithms, CGA is a selectionist scheme, because only the better individual is permitted to influence the subsequent generation of candidate solutions. The key idea is that a "survival-of-the-fittest" mechanism is used to *bias* the generation of new individuals. We usually use tournament selection [16] in CGA.

5. *Probabilistic model update:* After selection, the proportion of winning alleles is increased by $1/n$. Note that only the probabilities of those genes that are different between the two competitors are updated. That is,

$$p_{x_i}^{t+1} = \begin{cases} p_{x_i}^t + 1/n & \text{If } x_{w,i} \neq x_{c,i} \text{ and } x_{w,i} = 1, \\ p_{x_i}^t - 1/n & \text{If } x_{w,i} \neq x_{c,i} \text{ and } x_{w,i} = 0, \\ p_{x_i}^t & \text{Otherwise.} \end{cases} \tag{2.1}$$

   Where, $\mathbf{x}_{w,i}$ is the i^{th} gene of the winning chromosome, $\mathbf{x}_{c,i}$ is the i^{th} gene of the competing chromosome, and $p_{x_i}^t$ is the i^{th} element of the probability vector—representing the proportion of i^{th} gene being one—at generation t. This updating procedure of CGA is equivalent to the behavior of a GA with a population size of n and steady-state binary tournament selection.

6. Repeat steps 2–5 until one or more termination criteria are met.

The probabilistic model of CGA is similar to those used in population-based incremental learning (PBIL) [2, 3] and the univariate marginal distribution algorithm (UMDA) [32, 31]. However, unlike PBIL and UMDA, CGA can simulate a genetic algorithm with a given population size. That is, unlike the PBIL and

UMDA, CGA modifies the probability vector so that there is direct correspondence between the population that is represented by the probability vector and the probability vector itself. Instead of shifting the vector components proportionally to the distance from either 0 or 1, each component of the vector is updated by shifting its value by the contribution of a single individual to the total frequency assuming a particular population size.

Additionally, CGA significantly reduces the memory requirements when compared with simple genetic algorithms and PBIL. While the simple GA needs to store n bits, CGA only needs to keep the proportion of ones, a finite set of n numbers that can be stored in $\log_2 n$ for each of the ℓ gene positions. With PBIL's update rule, an element of the probability vector can have any arbitrary precision, and the number of values that can be stored in an element of the vector is not finite.

Elsewhere, it has been shown that CGA is operationally equivalent to the order-one behavior of simple genetic algorithm with steady state selection and uniform crossover [18]. Therefore, the theory of simple genetic algorithms can be directly used in order to estimate the parameters and behavior of the CGA. For determining the parameter n that is used in the update rule, we can use an approximate form of the gambler's ruin population-sizing[2] model proposed by Harik et al. [17]:

$$n = -log\alpha \cdot \frac{\sigma_{BB}}{d} \cdot 2^{k-1}\sqrt{\pi \cdot m}, \tag{2.2}$$

where k is the BB size, m is the number of building blocks (BBs)—note that the problem size $\ell = k \cdot m$,—d is the size signal between the competing BBs, and σ_{BB} is the fitness variance of a building block, and α is the failure probability.

2.3 The Extended Compact Genetic Algorithm

The extended compact genetic algorithm (eCGA), is based on a key idea that the choice of a good probability distribution is equivalent to linkage learning [19]. The measure of a good distribution is quantified based on minimum description length (MDL) models. The key concept behind MDL models is that given all things are equal, simpler distributions are better than the complex ones. The MDL restriction penalizes both inaccurate and complex models, thereby leading to an optimal probability distribution. The probability distribution used in eCGA is a class of probability models known as marginal product models (MPMs). MPMs are formed as a product of marginal distributions on a partition of the genes. MPMs also facilitate a direct linkage map with each partition separating tightly linked genes.

[2]The experiments conducted in this thesis used $n = 3\ell$.

The eCGA, later extended to deal with n-ary alphabets in χ-eCGA [6], can be algorithmically outlined 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 [16].
4. Build the probabilistic model: In χ-eCGA, both the structure of the model as well as the parameters of the models are searched. A greedy search is used to search for the model of the selected individuals in the population.
5. Create new individuals by sampling the probabilistic model.
6. Evaluate the fitness value of all offspring.
7. Repeat steps 3–6 until some convergence criteria are met.

Two things need further explanation: (1) the identification of MPM using MDL, and (2) the creation of a new population based on MPM.

The identification of MPM in every generation is formulated as a constrained optimization problem,

$$\text{Minimize} \qquad C_m + C_p \tag{2.3}$$

$$\text{Subject to}$$

$$\chi^{k_i} \leq n \quad \forall i \in [1, m] \tag{2.4}$$

where χ is the alphabet cardinality—$\chi = 2$ for the binary strings—C_m is the model complexity which represents the cost of a complex model and is given by

$$C_m = \log_\chi(n+1) \sum_{i=1}^{m} \left(\chi^{k_i} - 1\right) \tag{2.5}$$

and C_p is the compressed population complexity which represents the cost of using a simple model as against a complex one and is evaluated as

$$C_p = \sum_{i=1}^{m} \sum_{j=1}^{\chi^{k_i}} N_{ij} \log_\chi \left(\frac{n}{N_{ij}}\right) \tag{2.6}$$

where m in the equations represent the number of BBs, k_i is the length of BB $i \in [1, m]$, and N_{ij} is

the number of chromosomes in the current population possessing bit-sequence $j \in [1, \chi^{k_i}]$[3] for BB i. The constraint (Equation 2.4) arises due to finite population size.

The greedy search heuristic used in χ-eCGA starts with a simplest model assuming all the variables to be independent and sequentially merges subsets until the MDL metric no longer improves. Once the model is built and the marginal probabilities are computed, a new population is generated based on the optimal MPM as follows, population of size $n(1 - p_c)$ where p_c is the crossover probability, is filled by the best individuals in the current population. The rest $n \cdot p_c$ individuals are generated by randomly choosing subsets from the current individuals according to the probabilities of the subsets as calculated in the model.

One of the critical parameters that determines the success of eCGA is the population size. Analytical models have been developed for predicting the population-sizing and the scalability of eCGA [36]. The models predict that the population size required to solve a problem with m building blocks of size k with a failure rate of $\alpha = 1/m$ is given by

$$n \propto \chi^k \left(\frac{\sigma_{BB}^2}{d^2} \right) m \log m, \tag{2.7}$$

where n is the population size, χ is the alphabet cardinality (here, $\chi = 3$), k is the building block size, $\frac{\sigma_{BB}^2}{d^2}$ is the noise-to-signal ratio [15], and m is the number of building blocks. For the experiments presented in this thesis we used $k = |a| + 1$ (where $|a|$ is the number of address inputs), $\frac{\sigma_{BB}^2}{d^2}$=1.5, and $m = \frac{\ell}{|I|}$ (where ℓ is the rule size).

[3]Note that a BB of length k has χ^k possible sequences where the first sequence denotes be $00\cdots0$ and the last sequence $(\chi-1)(\chi-1)\cdots(\chi-1)$

Chapter 3

Using MapReduce

In this chapter, 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. This is followed by the MapReduce algorithms for compact and extended compact genetic algorithm.

3.1 MapReducing SGAs

We encapsulate each iteration of the GA as a separate MapReduce job. The client accepts the command-line parameters, creates the population and submits the MapReduce job.

Selecto-recombinative genetic algorithms [13, 14], 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.

3.1.1 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.

3.1.2 Partitioner

If the selection operation in a GA (Step 3) is performed locally on each node, spatial constraints are artificially introduced and leading to reduced the selection pressure [35]. This can lead to increase in the convergence time. Hence, decentralized and distributed selection algorithms [21] 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.

Algorithm 1 Map phase of each iteration of the GA

```
1: MAP(key, value):
2: individual ← INDIVIDUALREPRESENTATION(key)
3: fitness ← CALCULATEFITNESS(individual)
4: EMIT (individual, fitness)

5: {Keep track of the current best}
6: if fitness > max then
7:         max ← fitness
8:         maxInd ← individual
9: end if

10: if all individuals have been processed then
11:         Write best individual to global file in DFS
12: end if
```

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 responsible for processing the given $(key, value)$. The default implementation uses HASH(key) % $numReducers$ so that all the values corresponding to a given key end up at the same reducer.

However, there are two reasons why this does not suit the needs of genetic algorithms: First, the HASH function partitions the name space 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

```
1: int GETPARTITION(key, value, numReducers):
2: return RANDOMINT(0, numReducers - 1)
```

3.1.3 Reduce

We implement Tournament selection without replacement [12]. 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, SelectionAndCrossover 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.

Algorithm 3 Reduce phase of each iteration of the GA

```
1: Initialize processed ← 0, tournArray [2· tSize], crossArray [cSize]

2: Reduce(key, values):
3: while values.hasNext() do
4:         individual ← IndividualRepresentation(key)
5:         fitness ← values.getValue()

6:         if processed < tSize then
7:                 {Wait for individuals to join in the tournament and put them for the last rounds}
8:                 tournArray [tSize + processed%tSize] ← individual
9:         else
10:                {Conduct tournament over past window}
11:                SelectionAndCrossover()
12:        end if
13:        processed ← processed + 1

14:        if all individuals have been processed then
15:                {Cleanup for the last tournament windows}
16:                for k ←1 to tSize do
17:                        SelectionAndCrossover()
18:                        processed ← processed + 1
19:                end for
20:        end if
21: end while

22: SelectionAndCrossover:
23: crossArray[processed%cSize] ← Tourn(tournArray)
24: if (processed - tSize) % cSize = cSize - 1 then
25:        newIndividuals ← Crossover(crossArray)
26:        for individual in newIndividuals do
27:                Emit (individual, dummyFitness)
28:        end for
29: end if
```

3.1.4 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 speed-up is bounded because of this serial component. Amdahl's law states: if p is the proportion of a program that can be made parallel (i.e. benefit from parallelization), and $(1-p)$ is the proportion that cannot be parallelized (remains serial), then the maximum speed-up that can be achieved by using n processors in the limit, as n tends to infinity tends to $1/(1-p)$. Thus, the speed up is bound by fraction of time of 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[1]. We seed the pseudo-random number generator for each mapper with $mapper\_id \cdot current\_time$. 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.

3.2 MapReducing Compact Genetic Algorithms

We encapsulate each iteration of the CGA as a separate single MapReduce job. The client accepts the command-line parameters, creates the initial probability vector splits and submits the MapReduce job. Let the probability vector be $P = \{p_i : p_i = Probability\_of\_the\_variable(i) = 1\}$. Such an approach would allow us to scale in terms of the number of variables, if P is partitioned into m different partitions $P_1, P_2, \ldots, P_m$ where m is the number of mappers.

3.2.1 Map

Generation of the two individuals matches the MAP function, which has to be computed independent of other instances. As shown in the algorithm in Algorithm 3.2.1, the MAP takes a probability split P_i as input and outputs the $tournamentSize$ individuals splits, as well as the probability split. Also, it keeps track of the number of ones in both the individuals and writes it to a global file in the Distributed File System (HDFS). All the reducers later read these values.

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

Algorithm 4 Map phase of each iteration of the CGA

```
1: MAP(key, value):
2: splitNo ← key
3: probSplitArray ← value
4: EMIT(splitNo, [0, probSplitArray])
5: for k ←1 to tournamentSize do
6:         SELECTIONANDCROSSOVER()
7:         processed ← processed + 1
8:         individual ←
9:         ones ← 0
10:        for prob in probSplitArray do
11:                if RANDOM(0,1) < prob then
12:                        individual ← 1
13:                        ones ← ones + 1
14:                else
15:                        individual ← 0
16:                end if
17:                EMIT(splitNo, [k, individual])
18:                WRITETODFS(k, ones)
19:        end for
20: end for
```

3.2.2 Reduce

We implement Tournament selection without replacement. A tournament is conducted among $tournamentSize$ generated individuals and the winner and the loser is selected. Then, the probability vector split is updated accordingly. A detailed description of the reduce step can be found on Algorithm 3.2.2.

3.2.3 Optimizations

We use optimizations similar to the simple GA. After initial experimentation, we noticed that for larger problem sizes, the serial initialization of the population takes a long time. Similar to the optimizations used while MapReducing SGAs, we create the initial population in a separate MapReduce phase, in which the MAP generates the initial probability vector and the REDUCE is the Identity Reducer.

The bits of the variables in the individual are compactly represented in an array of **long long int**s and we use efficient bit operations for crossover and fitness calculations. Also, we use **long long int**s to represent probabilities instead of floating point numbers and use the more efficient integer operations.

3.3 MapReducing Extended Compact Genetic Algorithms

All the steps in eCGA as described in the previous section, except step 4 are very similar to a simple genetic algorithm. We modify our technique of scaling simple genetic algorithms by breaking the eCGA algorithm

Algorithm 5 Reduce phase of each iteration of the CGA

```
1: Initialize:
2: AllocateAndInitialize(OnesArray[tournamentSize])
3: winner ← -1
4: loser ← -1
5: processed ← 0
6: n ← 0
7: for k ← 1 to tournamentSize do
8:         for r ← 1 to numReducers do
9:                 Ones[k] ← Ones[k] + ReadFromDFS(r, k)
10:                if Ones[k] > winner then
11:                        winnerIndex ← k
12:                else
13:                        if Ones[k] < loser then
14:                                loserIndex ← k
15:                        end if
16:                end if
17:        end for
18: end for

19: Reduce(key, values):
20: while values.hasNext() do
21:        splitNo ← key
22:        value[processed] ← values.getValue()
23:        processed ← processed + 1
24: end while
25: for prob in value[0] do
26:        if value[winner].bit[n] ≠ value[winner][n] then
27:                if value[winner].bit[n] = 1 then
28:                        newProbSplit [n] ← value[0] + 1/population
29:                else
30:                        newProbSplit [n] ← value[0] - 1/population
31:                end if
32:        end if
33:        Emit(splitNo, [0, newProbSplit])
34: end for
```

into two MapReduces, which also inspired by our previous work on eCGA using Meandre [42]. The first MapReduce computes the fitness of the individuals in the Map phase and performs a tournament selection in the Reduce phase. After this MapReduce, we develop a MapReduce algorithm for building the model. After this model is built, we perform a second MapReduce to perform the crossover according to the model that has been built.

Algorithm 6 Building the model in eCGA

```
Initially, each bit is a separate building block b, P[b] ← 0, ∀ building blocks b

ComputeMarginalProbabilites:
// Compute the marginal probability of building blocks
for all building blocks b do
    for all individuals i do
        value ← decimal value of b in i
        P(b)[value] ← P(b)[value] + 1
    end for
end for

PickAndMerge:
// Find the best merge of building blocks
b_i ← −1, b_j ← −1, b_comp ← 1
while b_comp > 0 do
    b_comp ← −1
    for i ← 0 to number of building blocks do
        for j ← i + 1 to number of building blocks do
            c_i ← Combined complexity of b_i
            c_j ← Combined complexity of b_j
            c_ij ← Combined complexity of blocks b_i and b_j combined together
            δ_ij ← c_i + c_j − c_ij
            if δ_ij ≥ b_comp then
                b_i ← i, b_j ← j, b_comp ← δ_ij
            end if
        end for
    end for
    if b_comp ≠ −1 then
        // Perform the merge and recompute
        Merge building blocks i and j
        Recompute the marginal probability of each building block
    end if
end while
```

The model building is an important step in eCGA and can become the bottleneck if implemented sequentially. However, it is also difficult to parallelize this step because of the interdependence of these steps. We split the population among different mappers. Each mapper could calculate the local C_m and C_p values. However, the global values cannot be calculated from these local values because of the operations involved in their calculation. Specifically, it is difficult to express $\log(x+y)$ as any independent function $h(f(x), g(y))$ where h, f and g can be any arbitrary functions.

We could partition the different building blocks among multiple machines. However, in this case, every mapper would have to read in the entire population. As the required population scales as $n \log n$, where n is the number of variables; this would be infeasible.

We partition the population among multiple mappers, which count the marginal probability of each building block in the individuals it processes. Then we have a single reducer which aggregates these marginal probabilities and computes the global C_m and C_p values. As a part of the greedy heuristic for building the model, the reducer picks the best building blocks to merge and sends the merged partition to the mappers. Since we have a single reducer, we try to offload as much work as possible to the multiple mappers. Hence, the mappers also pre-compute the local C_m and C_p values of every possible two-way merge of the building blocks as shown in Algorithm 1.

We decided to partition the individuals among multiple mappers. These mappers compute the marginal probabilities of each building block according to the ComputeMarginalProbabilities function and also compute the marginal probabilities for every possible pair-wise merge of the building blocks and emit these values to the reducer. We use a single to reducer to aggregate all these marginal probabilities for each building block. Then, we use the PickAndMerge function to go over pair-wise merge and pick the best possible merge. It writes this changed building block index to a file, which is later read by the next round of mappers. If the compressed value cannot be decreased, the model building is complete and the client starts the next MapReduce.

Chapter 4

Evaluation

In this chapter, we report the results of the experiments to evaluate the algorithms presented in the previous chapter.

We implemented the MapReduce algorithms on Hadoop (0.19)[1] and ran it on our 416 core (52 nodes) Hadoop cluster. Each 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[2]. 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.

4.1 Simple Genetic Algorithm

The OneMax Problem [39] (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:

$$F(\vec{x}) = \sum_{i=1}^{N} x_i \tag{4.1}$$

We use the OneMax problem to evaluate our implementation of simple genetic algorithms and perform the following experiments:

[1] http://hadoop.apache.org
[2] http://cloud.cs.illinois.edu

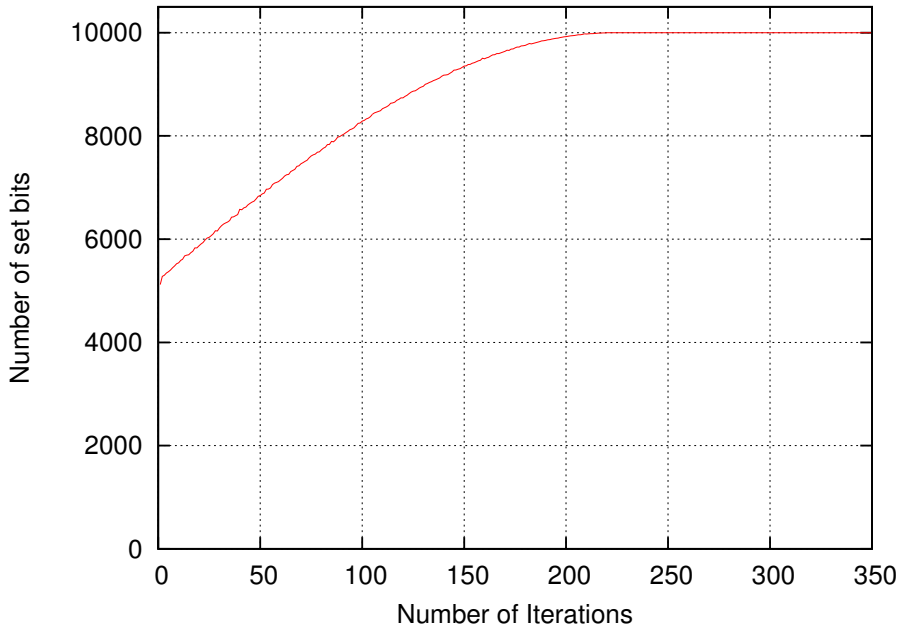

Figure 4.1: Convergence of GA for 10^4 variable OneMAX problem

4.1.1 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 OneMAX problem. As shown in Figure 4.1, the GA converges in 220 iterations taking an average of 149 seconds per iteration.

4.1.2 Scalability with constant load per node

In this experiment, we keep the load set to 1,000 variables per mapper. As shown in Figure 4.2, 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.

4.1.3 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 4.3, the time per iteration decreases 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

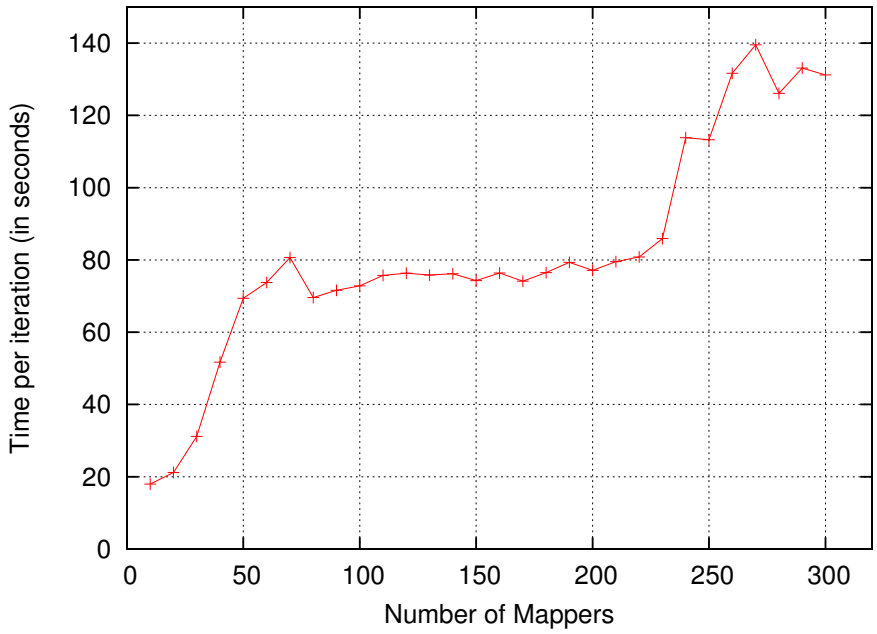

Figure 4.2: Scalability of GA with constant load per node for OneMAX problem

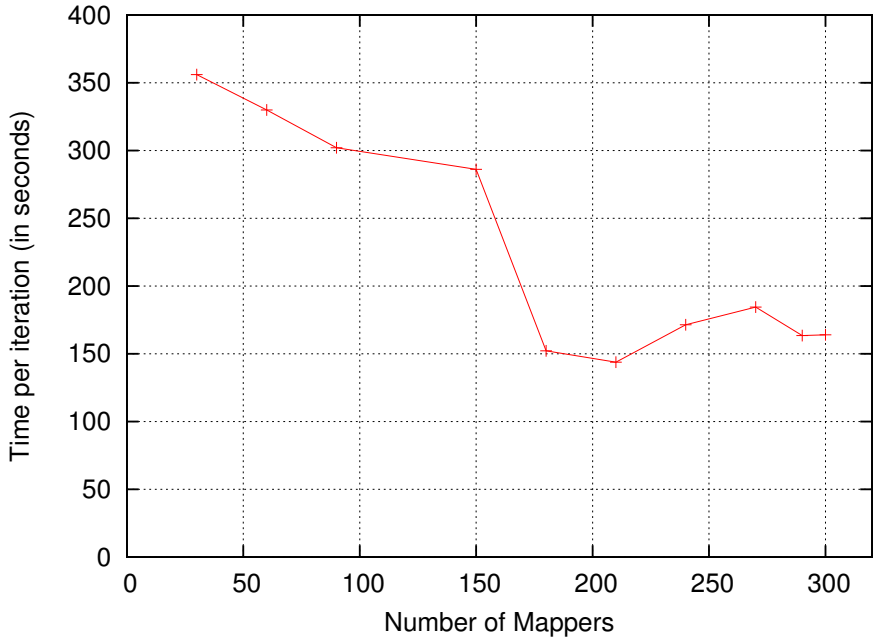

Figure 4.3: Scalability of GA for $50,000$ variable OneMAX problem with increasing number of mappers

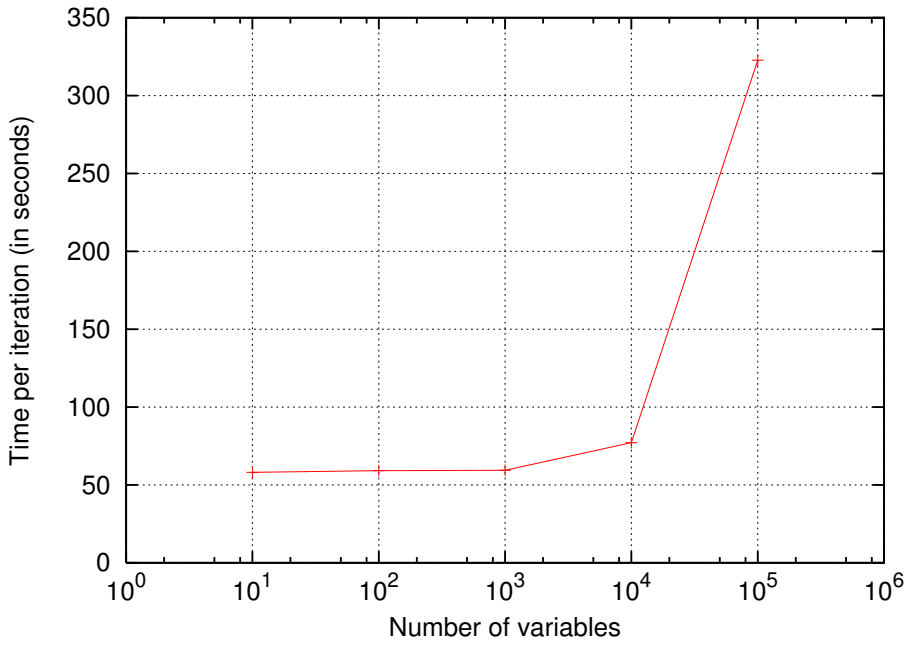

Figure 4.4: Scalability of GA for OneMAX problem with increasing number of variables

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.

4.1.4 Scalability with increasing the problem size

Here, we utilize the maximum resources and increase the number of variables. As shown in Figure 4.4, 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.

4.2 Compact Genetic Algorithms

To better understand the behavior of the Hadoop implementation of cGA, we repeated the two experiment sets done in the case of the Hadoop SGA implementation. For each experiment, the population for the cGA is set to $n \log n$ where n is the number of variables. As done previously, first we keep the load set to 200,000 variables per mapper. As shown in Figure 4.5, the time per iteration increases initially and

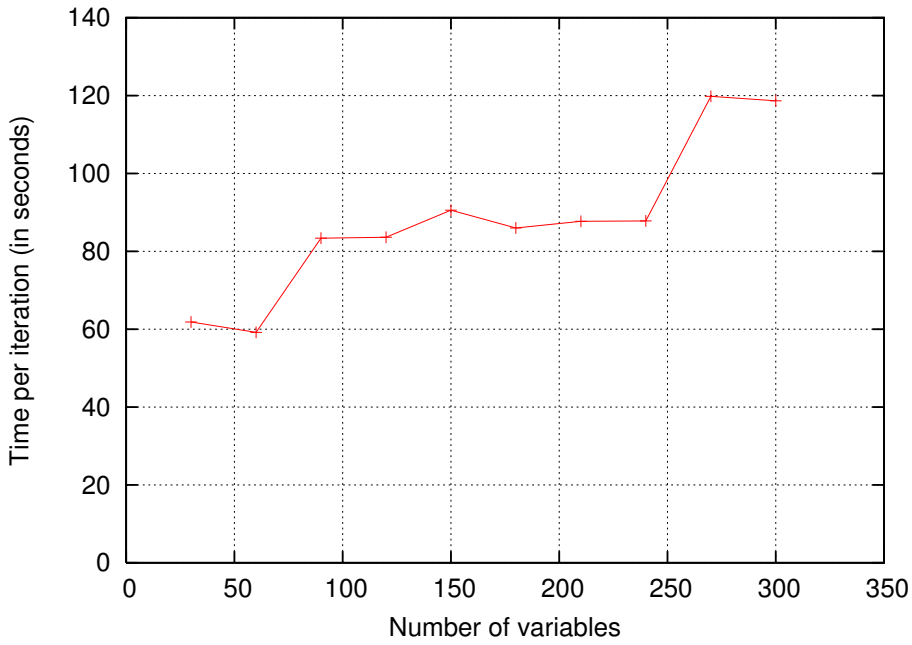

Figure 4.5: Scalability of compact genetic algorithm with constant load per node for the OneMAX problem.

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 fact that no available resources (mapper slots) in the Hadoop framework are available. Thus, the execution must wait till mapper slots are released and the remaining portions can be executed, and the whole execution completed.

In the second set of experiments, we utilized the maximum resources and increase the number of variables. As shown in Figure 4.6, our implementation scales to $n = 10^8$ variables, keeping the population set to $n \log n$.

4.3 Extended Compact Genetic Algorithm

We report our results for the experiments with MapReducing eCGA algorithms here.

4.3.1 Convergence

In order to ensure the correctness of our parallel implementation of the eCGA algorithm, we ran an experiment on a problem with 16 bit variables and it converged in three iterations, achieving the best possible fitness. We demonstrate the model building process that ensued in the last iteration in the following listing:

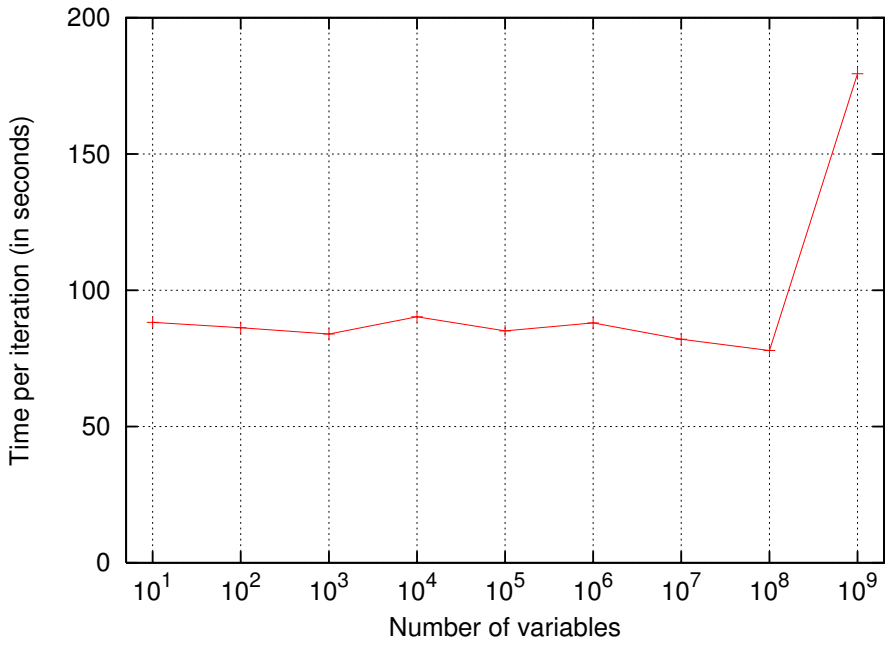

Figure 4.6: Scalability of compact genetic algorithm for OneMAX problem with increasing number of variables.

(0) (1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13) (14 15)

(0) (1) (2 3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13) (14 15)

(0) (1 2 3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13) (14 15)

(0) (1 2 3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13 14 15)

(0 1 2 3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13 14 15)

(0 1 2 3) (4) (5) (6) (7) (8) (9) (10) (11) (12 13 14 15)

(0 1 2 3) (4) (5 7) (6) (8) (9) (10) (11) (12 13 14 15)

(0 1 2 3) (4) (5 6 7) (8) (9) (10) (11) (12 13 14 15)

(0 1 2 3) (4) (5 6 7) (8) (9 11) (10) (12 13 14 15)

(0 1 2 3) (4) (5 6 7) (8) (9 10 11) (12 13 14 15)

(0 1 2 3) (4) (5 6 7) (8 9 10 11) (12 13 14 15)

(0 1 2 3) (4) (5 6 7) (8 9 10 11 12 13 14 15)

(0 1 2 3) (4 5 6 7) (8 9 10 11 12 13 14 15)

(0 1 2 3 4 5 6 7) (8 9 10 11 12 13 14 15)

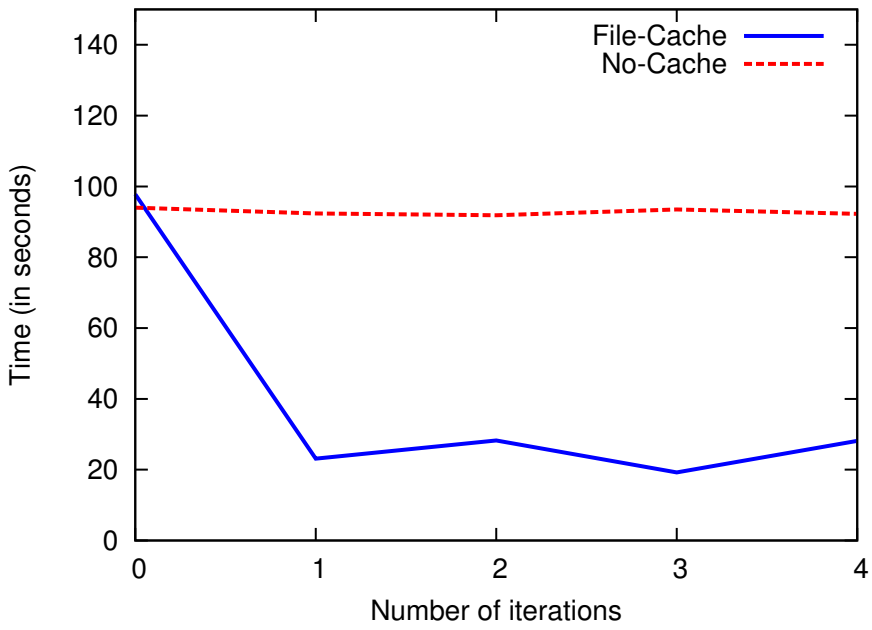

Figure 4.7: Effect of caching on the iteration time.

4.3.2 Caching

In this experiment, we measure the benefit of caching in the model building phase of the eCGA algorithm. In the first iteration, we compute the marginal probabilities of each building block in the map phase and the the marginal probabilities of each pair-wise combination of the building block. If we don't cache these marginal probabilities, they are computed in every iteration of the model building process. This is demonstrated in the "No-Cache" line in Figure 4.7. We can cache most of this information for the next iteration, as only the merged building block will have different marginal probabilities. This results in upto 80% lesser time per iteration, as is demonstrated in the "File-cache" line in the same figure.

4.3.3 Scaling the model building with problem size

In this experiment, we analyze the average time per iteration in the model building process for different problem sizes. Our results show that for problem sizes upto 128, the start-up overhead of the MapReduce results in similar execution times for the no-cache and file-cache versions as shown in Figure 4.8. The difference becomes more prominent for larger problem sizes. Our implementation scales up to 1024 bit variable problems. We found that beyond this value, the memory overhead of maintaining marginal probabilities for each pair-wise merge of building blocks becomes the bottleneck.

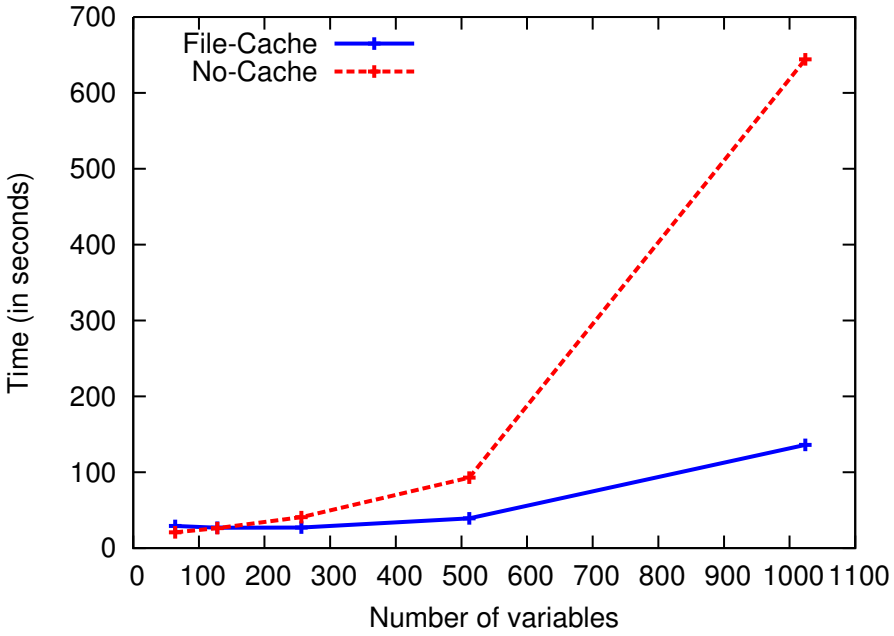

Figure 4.8: Scaling the model building with problem size.

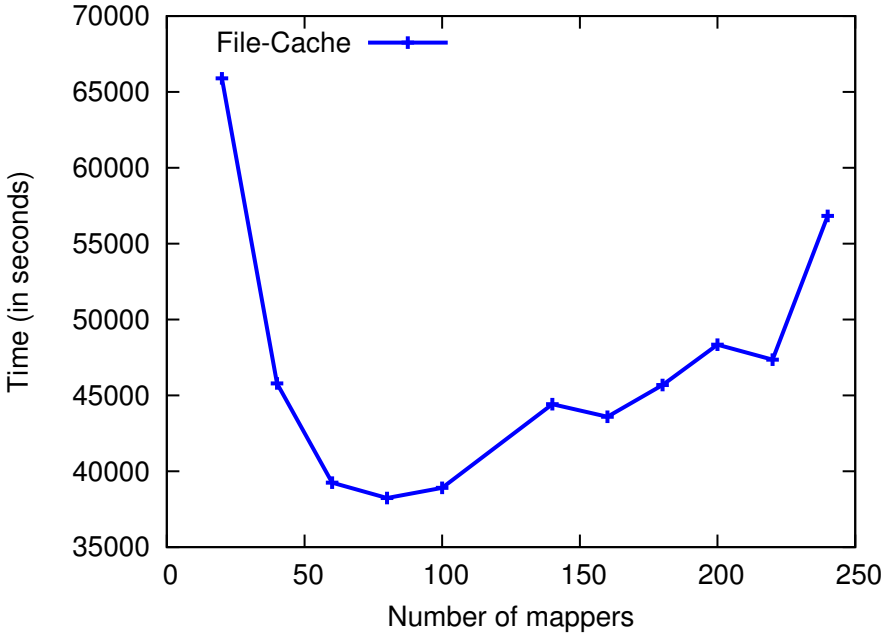

Figure 4.9: Scaling the model building with number of mappers.

4.3.4 Scaling the model building with number of mappers

This experiment shows how our implementation scales with increasing number of mappers. Figure 4.9 shows the average time per iteration as the number of mappers is increased. When the number of mappers is small, the mappers have too much load and the time per iteration is high. As this work is distributed among more machines, as the number of mappers is increased, the time decreases. However, as the number of mappers is increased beyond a limit (120), then the overhead of reading from so many mappers by the single reducer increases and the time increases.

Chapter 5

Related Work

Several different models like fine grained [28], coarse grained [24] and distributed models [23] 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 [8] 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 [34] for multi-cores and CGL-MapReduce [9] for streaming applications.

To the best of our knowledge, MRPGA [20] 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 [8]. 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. Meandre[26, 25] 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[38].

Chapter 6

Conclusions

We have shown that implementing evolutionary computation algorithms using a data-intensive computing paradigm is possible. We have presented step-by-step transformations for three illustrative cases—selecto-recombinative genetic algorithms and estimation of distribution algorithms—and reviewed some best practices during the process. Transformations have shown that Hadoop's MapReduce model can help scale easily and transparently evolutionary computation algorithms. Moreover, our results have also shown the inherent benefits of the underlying usage of data-intensive computing frameworks and how, when properly engineered, these algorithms can directly benefit from the current race on increasing the number of cores per chips without having to change the original data-intensive flow.

Results have shown that Hadoop is an excellent choice when we have to deal with large problems, as long as resources are available, being able to maintain iteration times relatively constant despite the problem size. We have also shown that linear speedups are possible without changing the underlying algorithms based on data-intensive computing thanks to the its inherent parallel processing. We have also shown that such results hold for multicore architectures, but also for multiprocessor NUMA architectures.

Our future work is focused on 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.

References

[1] E. Alba, editor. *Parallel Metaheuristics*. Wiley, 2007.

[2] S. Baluja. Population-based incremental learning: A method of integrating genetic search based function optimization and competitive learning. Technical Report CMU-CS-94-163, Carnegie Mellon University, 1994.

[3] S. Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. Technical Report CMU-CS-95-141, Carnegie Mellon University, 1995.

[4] M. D. Beynon, T. Kurc, A. Sussman, and J. Saltz. Design of a framework for data-intensive wide-area applications. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, page 116, Washington, DC, USA, 2000. IEEE Computer Society.

[5] E. Cantú-Paz. *Efficient and Accurate Parallel Genetic Algorithms*. Springer, 2000.

[6] L. de la Ossa, K. Sastry, and F. G. Lobo. Extended compact genetic algorithm in C++: Version 1.1. IlliGAL Report No. 2006013, University of Illinois at Urbana-Champaign, Urbana, IL, March 2006.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[9] J. Ekanayake, S. Pallickara, and G. Fox. Mapreduce for data intensive scientific analyses. In *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 277–284, Washington, DC, USA, 2008. IEEE Computer Society.

[10] I. Foster. The virtual data grid: A new model and architecture for data-intensive collaboration. In *in the 15^{th} International Conference on Scientific and Statistical Database Management*, pages 11–, 2003.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[12] D. Goldberg, K. Deb, and B. Korb. Messy genetic algorithms: motivation, analysis, and first results. *Complex Systems*, (3):493–530, 1989.

[13] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, Reading, MA, 1989.

[14] D. E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Kluwer Academic Publishers, Norwell, MA, 2002.

[15] D. E. Goldberg, K. Deb, and J. H. Clark. Genetic algorithms, noise, and the sizing of populations. *Complex Systems*, 6:333–362, 1992. (Also IlliGAL Report No. 91010).

[16] D. E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3(5):493–530, 1989.

[17] G. Harik, E. Cantú-Paz, D. E. Goldberg, and B. L. Miller. The gambler's ruin problem, genetic algorithms, and the sizing of populations. *Evolutionary Computation*, 7(3):231–253, 1999. (Also IlliGAL Report No. 96004).

[18] G. Harik, F. Lobo, and D. E. Goldberg. The compact genetic algorithm. *Proceedings of the IEEE International Conference on Evolutionary Computation*, pages 523–528, 1998. (Also IlliGAL Report No. 97006).

[19] G. R. Harik, F. G. Lobo, and K. Sastry. Linkage learning via probabilistic modeling in the ECGA. In M. Pelikan, K. Sastry, and E. Cantú-Paz, editors, *Scalable Optimization via Probabilistic Modeling: From Algorithms to Applications*, chapter 3. Springer, Berlin, in press. (Also IlliGAL Report No. 99010).

[20] C. Jin, C. Vecchiola, and R. Buyya. Mrpga: An extension of mapreduce for parallelizing genetic algorithms. *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 214–221, 2008.

[21] K. D. Jong and J. Sarma. On decentralizing selection algorithms. In *In Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 17–23. Morgan Kaufmann, 1995.

[22] P. Larrañaga and J. A. Lozano, editors. *Estimation of Distribution Algorithms*. Kluwer Academic Publishers, Boston, MA, 2002.

[23] D. Lim, Y.-S. Ong, Y. Jin, B. Sendhoff, and B.-S. Lee. Efficient hierarchical parallel genetic algorithms using grid computing. *Future Gener. Comput. Syst.*, 23(4):658–670, 2007.

[24] S.-C. Lin, W. F. Punch, and E. D. Goodman. Coarse-grain parallel genetic algorithms: Categorization and new approach. In *Proceeedings of the Sixth IEEE Symposium on Parallel and Distributed Processing*, pages 28–37, 1994.

[25] X. Llorà. Data-intensive computing for competent genetic algorithms: a pilot study using meandre. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1387–1394, New York, NY, USA, 2009. ACM.

[26] X. Llorà, B. Ács, L. Auvil, B. Capitanu, M. Welge, and D. E. Goldberg. Meandre: Semantic-driven data-intensive flows in the clouds. In *Proceedings of the 4th IEEE International Conference on e-Science*, pages 238–245. IEEE press, 2008.

[27] X. Llorà, A. Verma, R. H. Campbell, and D. E. Goldberg. When Huge Is Routine: Scaling Genetic Algorithms and Estimation of Distribution Algorithms via Data-Intensive Computing. In F. Fernndez de Vega and E. Cant-Paz, editors, *Parallel and Distributed Computational Intelligence*, chapter 1, page 1141. Springer-Verlag, Berlin Heidelberg, 2010.

[28] T. Maruyama, T. Hirose, and A. Konagaya. A fine-grained parallel genetic algorithm for distributed parallel systems. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 184–190, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[29] C. A. Mattmann, D. J. Crichton, N. Medvidovic, and S. Hughes. A software architecture-based framework for highly distributed and data intensive scientific applications. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 721–730, New York, NY, USA, 2006. ACM.

[30] J. P. Morrison. *Flow-Based Programming: A New Approach to Application Development*. Van Nostrand Reinhold, 1994.

[31] H. Mühlenbein. The equation for response to selection and its use for prediction. *Evolutionary Computation*, 5(3):303–346, 1997.

[32] H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions I. Binary parameters. *Parallel Problem Solving from Nature, PPSN IV*, pages 178–187, 1996.

[33] M. Pelikan, F. Lobo, and D. E. Goldberg. A survey of optimization by building and using probabilistic models. *Computational Optimization and Applications*, 21:5–20, 2002. (Also IlliGAL Report No. 99018).

[34] R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, Jan 2007.

[35] A. J. Sarma, J. Sarma, and K. D. Jong. Selection pressure and performance in spatially distributed evolutionary. In *In Proceedings of the World Congress on Computatinal Intelligence*, pages 553–557. IEEE Press, 1998.

[36] K. Sastry and D. E. Goldberg. Designing competent mutation operators via probabilistic model building of neighborhoods. *Proceedings of the Genetic and Evolutionary Computation Conference*, 2:114–125, 2004. Also IlliGAL Report No. 2004006.

[37] K. Sastry, D. E. Goldberg, and X. Llorà. Towards billion-bit optimization via a parallel estimation of distribution algorithm. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 577–584, New York, NY, USA, 2007. ACM.

[38] K. Sastry, D. E. Goldberg, and X. Llora. Towards billion-bit optimization via a parallel estimation of distribution algorithm. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 577–584, New York, NY, USA, 2007. ACM.

[39] J. Schaffer and L. Eshelman. On Crossover as an Evolutionary Viable Strategy. In R. Belew and L. Booker, editors, *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 61–68. Morgan Kaufmann, 1991.

[40] G. Sywerda. Uniform crossover in genetic algorithms. In *Proceedings of the third international conference on Genetic algorithms*, pages 2–9, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[41] M. Uysal, T. M. Kurc, A. Sussman, and J. Saltz. A performance prediction framework for data intensive applications on large scale parallel machines. In *In Proceedings of the Fourth Workshop on Languages, Compilers and Run-time Systems for Scalable Computers, number 1511 in Lecture Notes in Computer Science*, pages 243–258. Springer-Verlag, 1998.

[42] A. Verma, X. Llorà, D. E. Goldberg, and R. H. Campbell. Scaling genetic algorithms using mapreduce. In *ISDA '09: Proceedings of the 2009 Ninth International Conference on Intelligent Systems Design and Applications*, pages 13–18, Washington, DC, USA, 2009. IEEE Computer Society.

[43] A. Verma, X. Llorà, S. Venkataraman, D. E. Goldberg, and R. H. Campbell. Scaling ecga model building via data-intensive computing. In *CEC '10: To appear in Proceedings of the IEEE Congress on Evolutionary Computation*, Barcelona, Spain, 2010. IEEE Computer Society.