Scaling eCGA Model Building via Data-Intensive Computing

Abhishek Verma, Xavier Llorà, Shivaram Venkataraman, David E. Goldberg, and Roy H. Campbell

***Abstract*— This paper shows how the extended compact genetic algorithm can be scaled using data-intensive computing techniques such as MapReduce. Two different frameworks (Hadoop and MongoDB) are used to deploy MapReduce implementations of the compact and extended compact genetic algorithms. Results show that both are good choices to deal with large-scale problems as they can scale with the number of commodity machines, as opposed to previous efforts with other techniques that either required specialized high-performance hardware or shared memory environments.**

I. Introduction

Efficiency enhancement efforts are key for improving the scalability and practical usage of genetic algorithms and estimation of distribution algorithms on many real-world problems. The extended compact genetic algorithm (eCGA) [1], is based on a key idea that the choice of a good probability distribution is equivalent to linkage learning. The minimum description length (MDL) restriction penalizes both inaccurate and complex models, thereby leading to an optimal probability distribution. However, the model-building algorithm based on the MDL metric used which enable linkage learning is also a hindrance to eCGA scalability. The model-building algorithm presents an asymptotic of $O(\ell^3)$, where ℓ is the number of genes.

Recent improvements to the efficiency of eCGA model building focus on clustering techniques to help reduce the model building complexity and local-search hybridization [2]. Such methodological efforts can be divided in four main categories: Parallelization, hybridization, time continuation and evaluation relaxation. In this paper we will focus on how parallelization could help palliate model-building complexity. In fact, if enough computation resources are available, the best asymptotic time complexity would be $\theta(\ell)$. Another key element in this scalability equation is the need of eCGA to maintain a population. For instance, algorithms like the compact genetic algorithm (CGA) [3] are able to scale and solve billion bit optimization problems [4] thanks to compact population representations based on probability distributions, getting rid of the need to maintain a population. However, model-building mechanisms like the one used in eCGA require that a population be maintained. Hence, when scaling the algorithm to large problem sizes, the memory requirements can be render the approach infeasible.

However, data richness can also be seen as an opportunity. Data-intensive computing exploits data parallelism as a form to facilitate scalability. Recent advances have shown that these models like MapReduce [5] may present an alternative approach that can help scale model-building approaches like eCGA. Results have shown that traditional genetic algorithm schemes, even simple estimation of distribution algorithms like CGA, can be parallelized and scaled easily with such techniques [6], [7], [8]. In this paper, we introduce data-intensive computing frameworks that when applied to eCGA can help scale its model-building algorithm to achieve the best asymptotic time complexity.

We start by reviewing some of the traditional approaches used in the research community to parallelize genetic algorithms in Section II. Then, in Section III we present a quick overview of data-intensive computing frameworks that we use throughout the paper. Before introducing the proposed parallelization of eCGA using data-intensive computing, we present a quick overview of CGA and eCGA in Sections IV and VI. Section V and VII present and discuss the results achieved by using data-intensive computing technique to parallelize both of the previously described algorithms. Finally, the paper concludes by evaluating the results obtained in Section VIII and conclude in Section IX.

II. Parallel Programming, GAs, and MPI

The methodology proposed by Goldberg [9] allows a principled design of competent GAs, which can solve hard problems in polynomial time. The same methodology can be used to design efficiency enhancements for GAs [10], which can be divided in four main categories: Parallelization, hybridization, time continuation and evaluation relaxation. A very superficial overview would describe the role of each of them as follows. Parallelization deals with the division of the GA in several processors. Hybridization deals with the integration of GAs with other search procedures. Time continuation considers the trade-off among using a larger population for short time or smaller population for long time. Finally, evaluation relaxation involves with the trade-off among having a noisy and cheap evaluation against an accurate and expensive one.

Work on parallel GAs have been twofold. On one hand, several parallel models have been proposed trying to take advantage of intrinsic parallelism of the process. Models

Abhishek Verma is with the Department of Computer Science, University of Illinois at Urbana-Champaign, 201 N. Goodwin Ave., Urbana, IL 61801 (phone: 1 217 333 0176; email: verma7@illinois.edu).

Xavier Llorà is with the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, 1205 W. Clark Street, Urbana, IL 61801 (phone: 1 217 265 0894; email: xllora@illinois.edu).

Shivaram Venkataraman is with the Department of Computer Science, University of Illinois at Urbana-Champaign, 201 N. Goodwin Ave., Urbana, IL 61801 (phone: 1 2117 333 0176; email: venkata4@illinois.edu).

David E. Goldberg is with the Department of Industrial and Enterprise Systems Engineering, University of Illinois at Urbana-Champaign, 104 S. Mathews Avenue, Urbana, IL 61801 (phone: 1 217 333 0897; email: deg@illinois.edu).

Roy H. Campbell is with the Department of Computer Science, University of Illinois at Urbana-Champaign, 201 N. Goodwin Ave., Urbana, IL 61801 (phone: 1 217 333 0215; email: rhc@illinois.edu).

ranging from master/slave configuration, island based models, of cellular automata inspired models [11], [12], [13] have been quite common in the literature. Most of these take advantage of the high-performance computing (HPC) resources available. Models like the one proposed by Sastry, Goldberg and Llorà [4] were able to solve billion bit variable problems relying on a MPI-based implementation running on supercomputers with high-speed interconnection networks. The authors also started exploring how hardware acceleration techniques could provide an extra boost. This trend has gone mainstream in the last five year in the evolutionary computation community, where researchers focus on hardware accelerations techniques to help maximize the hardware resources available [14], [15], [16], [17].

However, when solving large-scale optimization or machine learning problems using evolutionary computation techniques, researchers have realized that the population requirements may become infeasible if approached with traditional high-performance computing techniques [4]. Most models required maintaining the entire or at least a sample of large populations during the evolutionary process. However, the data abundance provided by such large populations have enabled data-intensive computing techniques to become a viable alternative parallelization scheme for evolutionary computation techniques [18], [7], [6], [8], as we will illustrate in the next three sections. Moreover, such approaches also provide three key advantages when compared to their traditional HPC counterparts:

1) They do not require detailed knowledge of the underlying hardware architecture and their complex programming techniques, which are hard to debug.
2) They do not require intensive check-pointing to tolerate failures quite common on large jobs than may run for days.
3) They scale well on commodity clusters; usually the efficiency of MPI techniques rely on expensive high quality interconnection networks.

III. Data-Intensive Computing Using MapReduce

This section presents a quick overview of three data-intensive frameworks that we will use or refer throughout the rest of the paper. The first one is Hadoop[1], Yahoo!'s open source MapReduce framework. Modeled after Google's MapReduce paper [5], Hadoop builds on the *map* and *reduce* primitives present in functional languages. Hadoop relies on these two abstractions to enable the easily development of large-scale distributed applications as long as your application can be modeled around these two phases. The second framework is MongoDB[2], a scalable, high-performance, open source, schema-free, document-oriented database. Among others, MongoDB provides MapReduce tasks as a primitive of the query interface. When documents are stored in sharded collections (collections of documents broken in to shards distributed across different servers),

[1] `http://hadoop.apache.org`
[2] `http://www.mongodb.org/`

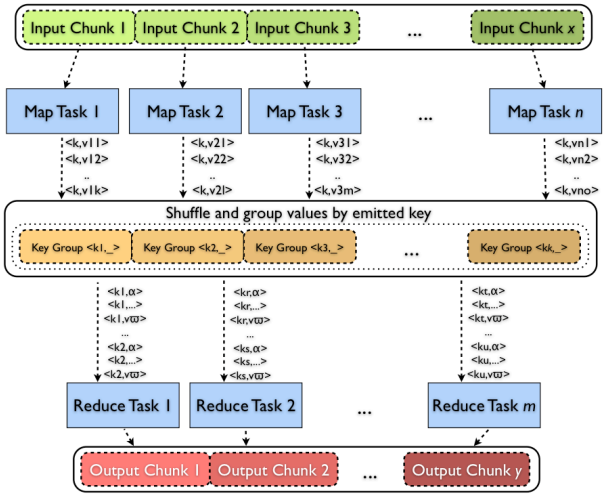

Fig. 1. Basic tasks and dataflow of a MapReduce job.

MongoDB is able to run MapReduce tasks in parallel making it an appealing alternative to Hadoop. Finally, we will also present a quick overview of Meandre [19], which is NCSA's data-intensive computing infrastructure for science, engineering, and humanities. Meandre provides a more flexible programming model that allows to create complex data flows, which could be regarded as complex and possible iterating MapReduce stages. Meandre can also benefit of some Hadoop tools, such as Hadoop's distributed file system.

A. The MapReduce Model

Inspired by the *map* and *reduce* primitives present in functional languages, Google popularized the MapReduce [5] 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 takes a set of input 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:

$$map(k_1, v_1) \rightarrow list(k_2, v_2) \quad (1)$$
$$reduce(k_2, list(v_2)) \rightarrow list(v_3) \quad (2)$$

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. The number of partitions (R) and the partitioning function are specified by the user. The overall execution is thus orchestrated in two steps: first all Mappers are executed in parallel, then the Reducers process the generated key value pairs by the Reducers. A detailed explanation of this framework is beyond the scope of this paper and can be found elsewhere [5]. We will also use Hadoop, Yahoo!'s open source MapReduce framework through this paper.

B. MapReduce in MongoDB

MongoDB [20] is a scalable, high-performance, open source, schema-free, document-oriented database. MongoDB provides mechanisms to store, retrieve, and query data stored as BSON documents. A BSON[3] document is a binary-encoded serialization of JSON-like documents, and, like JSON, supports the embedding of objects and arrays within other objects and arrays. MongoDB understands natively objects stored as BSON and, hence it is able to access key/values and index keys shared among documents and nested documents. A collection groups documents that share some common characteristic, which is similar to tables on a traditional relational data based managements system, and collections are grouped into a database. A MongoDB server can handle several databases. The basic interaction with MongoDB is via JavaScript. You can write simple snippets of JavaScript code to create, store, and retrieve objects. A complete description of the differences between MongoDB's NoSQL approach is beyond the scope of this paper and can be found elsewhere [20].

MongoDB's query language provides MapReduce jobs as a primitive. Thus, MapReduce tasks execute against an stored collection. Moreover, if the targeted collection is a sharded one, the execution of the MapReduce task is parallelized across the different servers that host shards of the collection. In another words, you can easily add more servers to a sharded MongoDB server and get the benefits of: (1) scattering shards across machines with lower disk space requirements, (2) replicating shards across machines for fault tolerance, and (3) easily scale MapReduce performance by parallel execution.

[3] `http://www.mongodb.org/display/DOCS/BSON`

The MapReduce model implemented by MongoDB slightly differs from Hadoop's implementation. The user can provide three different functions to a MapReduce task:

$$map(doc_i) \rightarrow list(k_j, doc_k)$$
$$reduce(k_i, list(doc_j)) \rightarrow doc_k$$
$$finalize(k_i, list(doc_j)) \rightarrow doc_k$$

the input keys and documents are drawn from a different domain than the output keys and values as done in the Hadoop implementation. The emitted keys/value pairs emitted on the *task* are stored and indexed on a temporary collection than then is used as the input of the reduce step. A key difference between the *reduce* operation in MongoDB is that the user-supplied function must be idempotent.

$$reduce(k, [reduce(k, v)]) = reduce(k, v) | \forall k, v$$

MongoDB may also issue multiple calls to the *reduce* function with a subset of the values available for the key to minimize the memory footprint. This technique is widely used when on MapReduce job against sharded collections. MongoDB also allows the user to supply a *finalize* function, that basically behaves as Hadoop *reduce* function. If a user provides a *finalize* function, MongoDB guarantees that it will only invoke this function only once for each key providing all the emitted values during the *map* step, which exhibits the same behavior as Hadoop's *reduce* function.

C. Data-Intensive Flow Computing with Meandre

Meandre [19] is a semantic-enabled web-driven, dataflow execution environment. It provides the machinery for assembling and executing data flows. Flows are software applications composed by components that process data. Each flow represents as a directed multigraph of executable components, nodes, which are linked through their input and output ports. Based on the inputs, properties, and its internal state, an executable component may produce output data. Meandre also provides component and flow publishing capabilities enabling users to assemble a repository of components by reusing and sharing. Users can discover and reuse components and flows previously published by other researchers. It is important to mention here, that component and flow can act as self-contained elements. Other approaches like Chimera still rely on external information [21]. Meandre builds on three main concepts: (1) dataflow-driven execution, (2) semantic-web metadata manipulation, and (3) metadata publishing. A detailed description of the Meandre data-intensive computing architecture is beyond the scope of this paper and can be found elsewhere [19].

The main difference between Meandre and the Hadoop and MongDB approaches relies on the data-driven execution nature of Meandre. Components with input and output ports can be connected to describe a complex task, commonly referred as flow. Dataflow execution engines provide a scheduler that

determines the firing (execution) sequence of components[4]. Due to its different approach, in the rest of this paper we will review some previous results obtained using Meandre [6], [8], but will mainly focus on MapReduce results obtained using Hadoop and MongoDB.

IV. The Compact Genetic Algorithm

The compact genetic algorithm [3] is one of the simplest estimation distribution algorithms (EDAs) [22], [23]. 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 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 [24] 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,

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

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.

[4]Meandre uses a *decentralized scheduling policy* designed to maximize the use of multicore architectures. Also groups of components can be placed across different machines and hence also scale by distributed execution. Meandre also allows works with processes that require directed cyclic graphs, thus extending beyond the traditional MapReduce directed acyclic graphs.

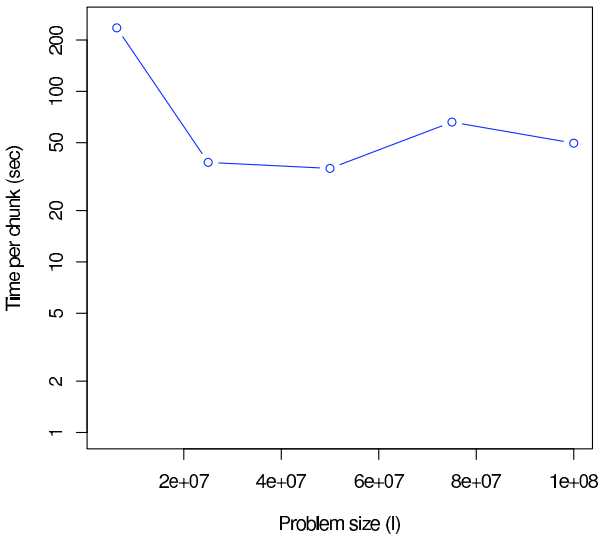

Fig. 2. CGA results obtained using MongoDB.

The probabilistic model of CGA is similar to those used in population-based incremental learning (PBIL) [25], [26] and the univariate marginal distribution algorithm (UMDA) [27], [28]. 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.

V. MapReducing CGA

CGA has been a main candidate for parallelization and large-scale optimization because of its small memory footprint [4] mainly using MPI techniques. Previous work has also shown that Hadoop and Meandre versions of CGA could be developed and scaled easily [6], [7], [8]. We revisit them and also provide a version build on MongoDB to help finalize the viability argument of data-intensive computing as a useful tool for evolutionary computation.

Previous results [7], [6], [8] showed compelling arguments in favor of the new path data-intensive technologies offer. A recent newcomer in this arena is MongoDB [20]. As introduced in section III, MongoDB scales by sharding document collections, hence when issuing a MapReduce task on sharded collections the execution is parallelized based on

the shards available. The implementation of CGA on top of MongoDB clearly resembles the one implemented using Hadoop. The CGA probability vector was divided on disjoint, equally sized, model fragments, an array of 20 probabilities. Each of these arrays was part of a MongoDB document via a key `g`. Each document also contained the unique fragment identifier `c` and the overall population size value `ps`. Far from being an exhaustive experimentation, we only wanted to validate that MongoDB and its lighter approach to MapReduce when compared to Hadoop. They follow the same performance trends as results previously obtained [7], [6], [8].

Figure 2 presents the results obtained running one iteration of MongoDB's CGA implementation. A constant load per shard, obtained by increasing the problem size, was maintained as we increased the number of processors used. As we increased the problem size (6.5M, 25M, 50M, 75M, and 100M bits) more shards (1,4,8,12 and 14 processors) were added. Under this assumptions, MongoDB should maintain a constant iteration time, as shown in CGA Hadoop's implementation. Figure 2 shows how MongoDB, under constant shard load, was able to deliver similar times per iteration. However, when comparing these results to Hadoop ones, MongoDB was not as stable. MongoDB sharding implementation is still on an alpha stage [20] and the consistent hashing mechanics used require a fair amount of documents to evenly distribute them across all the available shards. When inspecting MongoDB document distribution based on key range partitioning, significant fluctuations existed on the number of chunks assigned to shards. Some over allocated, some under allocated, which directly translated to the time fluctuations identified.

VI. The Extended Compact Genetic Algorithm

The extended compact genetic algorithm (eCGA) [1], is based on a key idea that the choice of a good probability distribution is equivalent to linkage learning. 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.

The eCGA, later extended to deal with n-ary alphabets in χ-eCGA [29], 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 [24].
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,

$$\text{Minimize} \qquad C_m + C_p \tag{3}$$

$$\text{Subject to} \qquad \chi^{k_i} \leq n \quad \forall i \in [1, m] \tag{4}$$

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

$$C_m = \log_\chi(n+1) \sum_{i=1}^{m} \left(\chi^{k_i} - 1\right) \tag{5}$$

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

$$C_p = \sum_{i=1}^{m} \sum_{j=1}^{\chi^{k_i}} N_{ij} \log_\chi \left(\frac{n}{N_{ij}}\right) \tag{6}$$

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 $j \in [1, \chi^{k_i}]$[5] for BB i. The constraint (Equation 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 [30]. 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

$$n \propto \chi^k \left(\frac{\sigma_{BB}^2}{d^2}\right) m \log m, \tag{7}$$

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

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 [31], and m is the number of building blocks. For the experiments presented in this paper 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).

VII. MapReducing eCGA

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 into two MapReduces, which also inspired by our previous work on eCGA using Meandre [7]. 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.

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 inter-dependence 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, it uses 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.

Algorithm 1 Building the model in eCGA:
Initially, each bit is a separate building block b,
P[b] $\leftarrow$ 0, $\forall$ building blocks b

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

PickAndMerge:
// Find the best merge of building blocks
$b_i \leftarrow -1, b_j \leftarrow -1, b_{comp} \leftarrow 1$
while $b_{comp} > 0$ **do**
  $b_{comp} \leftarrow -1$
  for $i \leftarrow 0$ to number of building blocks **do**
    for $j \leftarrow i+1$ to number of building blocks **do**
      $c_i \leftarrow$ Combined complexity of b_i
      $c_j \leftarrow$ Combined complexity of b_j
      $c_{ij} \leftarrow$ Combined complexity of blocks b_i and b_j combined together
      $\delta_{ij} \leftarrow c_i + c_j - c_{ij}$
      if $\delta_{ij} \geq b_{comp}$ **then**
        $b_i \leftarrow i, b_j \leftarrow j, b_{comp} \leftarrow \delta_{ij}$
      end if
    end for
  end for
  if $b_{comp} \neq -1$ **then**
    // Perform the merge and recompute
    Merge building blocks i and j
    Recompute the marginal probability of each building block
  end if
end while

VIII. Evaluation

We performed our experiments on 62 nodes from the Cloud Computing Testbed (CCT)[6]. The nodes are connected together with a Gigabit ethernet switch running 64 bit Cent OS 5.4 operating system. Each node has dual Intel Quad cores, 16GB RAM and 2TB hard disks. A single node was configured to be the JobTracker, another one was configured as the NameNode and the other 60 nodes were used as slaves. The replication factor of the distributed file system was set to 3 and the default chunksize was 128MB. The

[6]http://cloud.cs.illinois.edu

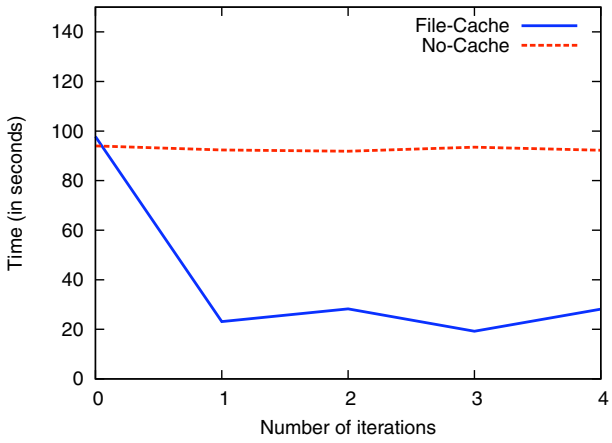

Fig. 3. Effect of caching on the iteration time

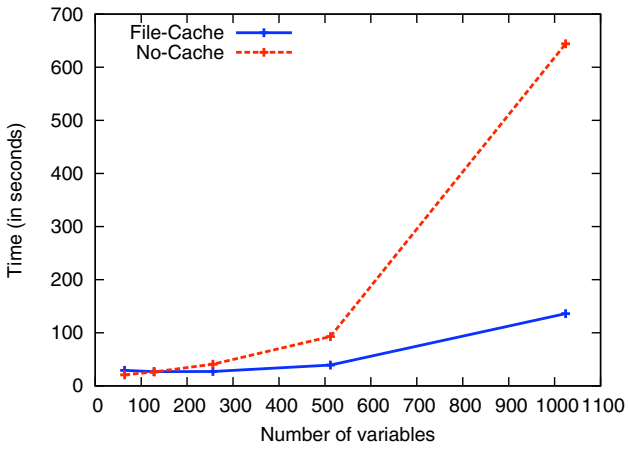

Fig. 4. Scaling the model building with problem size

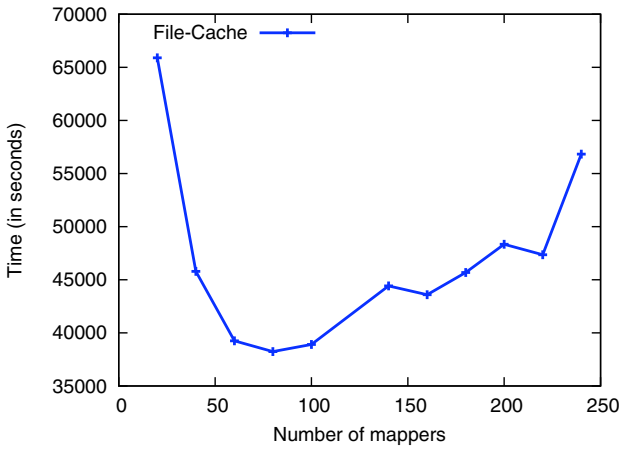

Fig. 5. Scaling the model building with number of mappers

number of Mappers and Reducers per node was set to 6 and 2 respectively, in order to utilize all the 8 cores on each node.

We tested our eCGA implementation on the MK deceptive trap function [9], where $k = 4$ and $d = 0.25$. We performed selection of the individuals using using tournament selection without replacement with a window size of 5.

A. 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.

B. 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 3. 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.

C. 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. 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.

D. Scaling the model building with number of mappers

This experiment shows how our implementation scales with increasing number of mappers. Figure 5 shows the average time per iterations 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.

IX. Conclusions

Regardless of the implementation—Hadoop, MongoDB or Meandre—this paper has shown that data-intensive computing can play a principal role in the scalability of estimation of distribution algorithms. Exploiting the massive data richness available on population-based methods as eCGA, or even

on compact memory models like CGA, has shown that can help scale as long as proper resource provisioning is available. Also, enhancement techniques based on computation caching has shown that they can greatly speed up eCGA model building, as they also do on the original sequential implementations.

ACKNOWLEDGMENTS

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

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

[2] T. S. Duque, D. E. Goldberg, and K. Sastry, "Enhancing the efficiency of the ecga," in *Proceedings of the 10th international conference on Parallel Problem Solving from Nature*, (Berlin, Heidelberg), pp. 165–174, Springer-Verlag, 2008.

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

[4] 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*, (New York, NY, USA), pp. 577–584, ACM, 2007.

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

[6] 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*, (New York, NY, USA), pp. 1387–1394, ACM, 2009.

[7] 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*, (Washington, DC, USA), pp. 13–18, IEEE Computer Society, 2009.

[8] 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 *Parallel and Distributed Computational Intelligence* (F. Fernndez de Vega and E. Cant-Paz, eds.), ch. 1, p. 1141, Berlin Heidelberg: Springer-Verlag, 2010.

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

[10] K. Sastry, D. E. Goldberg, and M. Pelikan, "Efficiency enhancement of probabilistic model building genetic algorithms," tech. rep., University of Illinois at Urbana-Champaign), 2004. IlliGAL TR No. 2004020.

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

[12] X. Llorà, *Genetic Based Machine Learning using Fine-grained Parallelism for Data Mining*. PhD thesis, Enginyeria i Arquitectura La Salle. Ramon Llull University, Barcelona, February, 2002.

[13] E. Alba, ed., *Parallel Metaheuristics*. Wiley, 2007.

[14] X. Llorà and K. Sastry, "Fast rule matching for learning classifier systems via vector instructions," in *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, (New York, NY, USA), pp. 1513–1520, ACM, 2006.

[15] J.-M. Li, X.-J. Wang, R.-S. He, and Z.-X. Chi, "An efficient fine-grained parallel genetic algorithm based on gpu-accelerated," in *NPC '07: Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing Workshops*, (Washington, DC, USA), pp. 855–862, IEEE Computer Society, 2007.

[16] A. Munawar, M. Wahib, M. Munetomo, and K. Akama, "Hybrid of genetic algorithm and local search to solve max-sat problem using nvidia cuda framework," *Genetic Programming and Evolvable Machines*, vol. 10, no. 4, pp. 391–415, 2009.

[17] M. L. Wong, "Parallel multi-objective evolutionary algorithms on graphics processing units," in *GECCO '09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, (New York, NY, USA), pp. 2515–2522, ACM, 2009.

[18] C. Jin, C. Vecchiola, and R. Buyya, "MRPGA: An extension of mapreduce for parellelizing genetic algorithms," in *IEEE Fouth International Conference on eScience 2008* (I. Press, ed.), pp. 214–221, 2008.

[19] 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*, pp. 238–245, IEEE press, 2008.

[20] G. Magnusson, R. Murphy, D. Merriman, M. Dirolf, E. Horowitz, K. Chodorow, J. Merriman, and K. Banker, "The Humongous Database: MongoDB." Online reference manual and driver's documentation at `http://www.mongodb.org`, 2010.

[21] 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*, pp. 11–, 2003.

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

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

[24] D. E. Goldberg, B. Korb, and K. Deb, "Messy genetic algorithms: Motivation, analysis, and first results," *Complex Systems*, vol. 3, no. 5, pp. 493–530, 1989.

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

[26] S. Baluja and R. Caruana, "Removing the genetics from the standard genetic algorithm," Tech. Rep. CMU-CS-95-141, Carnegie Mellon University, 1995.

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

[28] H. Mühlenbein, "The equation for response to selection and its use for prediction," *Evolutionary Computation*, vol. 5, no. 3, pp. 303–346, 1997.

[29] 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.

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

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