

When Huge is Routine: Scaling Genetic Algorithms and Estimation of Distribution Algorithms via Data-Intensive Computing

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

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. In this book chapter we explore how evolutionary computation algorithms can be modeled using two different data-intensive frameworks—Yahoo!’s Hadoop and NCSA’s Meandre. We present a detailed step-by-step description of how three different evolutionary computation algorithms, having different execution profiles, can be translated into the data-intensive computing paradigms. Results show that (1) Hadoop is an excellent choice to push evolutionary computation boundaries on very large problems, and (2) that transparent Meandre linear speedups are possible without changing the underlying data-intensive flow thanks to its inherent parallel processing.

Xavier Llorà

National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign,
1205 W. Clark Street, Urbana, IL 61801 e-mail: xllora@illinois.edu

Abhishek Verma

Department of Computer Science, University of Illinois at Urbana-Champaign, 201 N Goodwin
Ave, Urbana, IL 61801 e-mail: verma7@illinois.edu

Roy H. Campbell

Department of Computer Science, University of Illinois at Urbana-Champaign, 201 N Goodwin
Ave, Urbana, IL 61801 e-mail: rhc@illinois.edu

David E. Goldberg

Department of Industrial and Enterprise Systems Engineering, University of Illinois at Urbana-
Champaign, 104 S. Mathews Ave, Urbana, IL 61801 e-mail: deg@illinois.edu

1 Introduction

Data-intensive computing branding is relatively recent, however data flow started to get traction back in the mid 90's with the appearance of frameworks such as IBM promoted FBP[32] or NCSA's D2K [44], and later simplified and popularized by Google's MapReduce model [10] and Yahoo!'s open source Hadoop project¹. Recent advances on data-intensive computing have lead to frameworks that are now able to exploit massive parallelism to efficiently process petabytes of data. These frameworks, due to their data-flow nature, provide specialized programming environments tailored for developing flow applications that, up to a certain degree, transparently benefit from the available parallelism.

The current data deluge is happening across different domains and is forcing a rethinking of how large volumes of data are processed. Most of these data-intensive computing frameworks 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 [42, 6, 13, 31]. Although large bodies of research on parallelizing evolutionary computation algorithms are available [8, 1], there has been little work done in exploring the usage of data-intensive computing [22, 27].

The inherent parallel nature of evolutionary algorithms makes them optimal candidates for parallelization [8]. Moreover, as we will layout in this paper, 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 modeling. In this book chapter we will explore the usage of two frameworks: Yahoo!'s Hadoop model and its MapReduce implementation, and NCSA's semantic-driven data-intensive computing framework – Meandre² [29]. Hadoop provides a simple scalable programming model based on the implementation of two basic functions: the *map* and the *reduce* functions. The *map* function provides uniform and parallel process to large volumes of data in forms of chunks, whereas the *reduce* function aggregates the results produced by mappers. On the other hand, Meandre allows explicit descriptions of complex, and possibly iterating, data flows via a directed multigraph of components describing the data flow processing. Two main benefits of such a modeling are: (1) favoring encapsulation, reutilization, and sharing via Lego-like component modeling, and (2) massive parallel data-driven execution. The first benefit targets improving software engineering best practices and a detailed discussion is beyond the scope of this paper and can be find elsewhere [29].

To illustrate the benefits for the evolutionary computation community of adopting such approaches we selected three representative algorithms and developed their equivalent data-intensive computing equivalents. It is important to note here that

¹ <http://hadoop.apache.org/>

² Catalan spelling of the word *meander*. <http://seasr.org/meandre>

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 selectorecombinative genetic algorithm [15, 16], the compact genetic algorithm [20], and the extended compact genetic algorithm [21]. We will show how a simple selectorecombinative genetic algorithm [15, 16] can be modeled using the data-intensive computing via Hadoop's MapReduce approach and Meandre data-intensive flow modeling. 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) analyze the results obtained. The second example, the compact genetic algorithm [20], we focus on how Hadoop's MapReduce modeling can help scale being a clear competitor of traditional high performance computing version [40]. The third example addresses the parallelization of the model building of estimation of distribution algorithms. We will show how Meandre's data-driven implementation of the extended compact classifier system (eCGA) [21] produces, *de facto*, a parallelized implementation of the costly model building stage. Experiments show that speedups linear to the number of cores or processors are possible without any further modification.

It is important to note here, that each of these algorithms has different profiles. For instance, the simple selectorecombinative genetic algorithm requires dealing with large populations as you tackle large 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. In this book chapter, we will focus on the massive parallel data-driven execution that allows users to automatically benefit from the advances of the current multicore era — which has opened the door to petascale computing — without having to modify the underlying algorithm.

The rest of this chapter is organized as follows. Section 2 presents a quick introductory overview of the two data-intensive frameworks we will use through the chapter, Hadoop and Meandre. Then, Section 3 introduces the three evolutionary computation algorithms that we will use in our experimentation with the two introduced frameworks, a simple selectorecombinative 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 Section 4. Section 5 presents the results achieved and using the data-intensive implementations showing that scalability is only bounded by the available resources, and linear speedups are easily achievable. Finally we review some related work in section 6 and present some conclusions and possible further work on section 7.

2 Data-Intensive Computing

This section presents a quick overview of the two data-intensive frameworks we will use throughout the rest of this book chapter. The first one is Hadoop³ — Yahoo!’s open source MapReduce framework. Modeled after Google’s MapReduce paper [10], 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 Meandre [29]—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.

2.1 MapReduce and The Hadoop Model

Inspired by the *map* and *reduce* primitives present in functional languages, Google popularized the MapReduce[10] 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:

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

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.

³ <http://hadoop.apache.org>

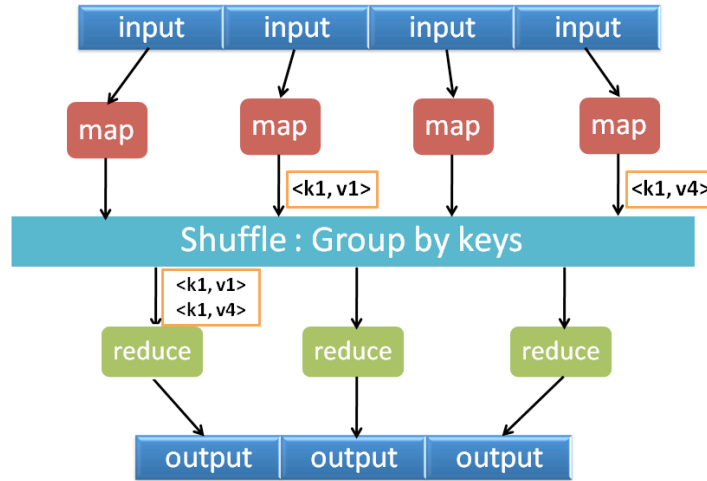


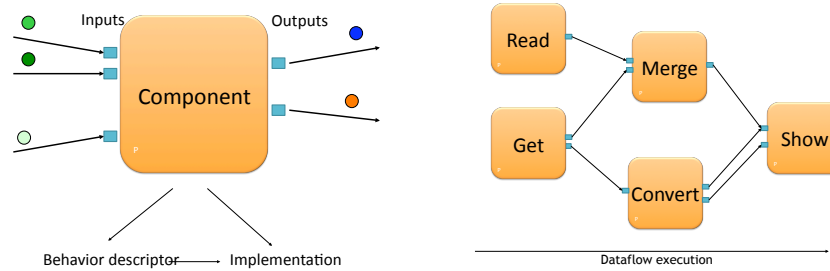
Fig. 1 MapReduce data flow overview

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 article and can be found elsewhere [10]. We will also use the Yahoo!’s open source MapReduce framework through this article.

2.2 Data-Intensive Flow Computing with Meandre

Meandre [29] 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—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 by querying and reuse components and flows previously published by other researchers. It is important to mention here, that

component and flow abstract can act as self-contained elements—other approaches like Chimera still rely on external information [13]. 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 [29].



(a) A component is described by several input and output ports where data flows through. Also, each component have a set of properties which govern its behavior in the presence of data.

(b) A flow is a directed graph where multiple components are connected together via input/output ports. A flow represents a complex task to solve.

Fig. 2 A data-intensive flow is characterized by the components it uses (basic process units) and their interconnection (a directed multigraph). Grouping several components together describes a complex task. It also emphasize rapid development by component reutilization.

2.2.1 Dataflow Execution Engines

Conventional programs perform their computational tasks by executing a sequence of instructions. One after another, each code instruction is fetched and executed. All data manipulation is performed by these basic units of execution. In a broad sense, this approach can be termed “code-driven execution.” Any computation task is regarded as a sequence of code instructions that ultimately manipulates data. However, data-driven execution (or dataflow execution) revolves around the idea of applying transformational operations to a flow or stream of data. In a data-driven model, data availability determines the sequence of code instructions to execute.

An analogy of the dataflow execution model is the black box operand approach. That is, any operand (operator) may have zero or more data inputs. It may also produce zero or more data through its data outputs. The operand behavior may be controlled by properties (behavior controls). Each operand performs its operations based on the availability of its inputs. For instance, an operand may require that data is available in all its inputs to perform its operations. Others may only need some, or

none. A simple example of a black box operand could be the arithmetic ‘+’ operand. This operand can be modeled as follows:

1. It requires two inputs.
2. When two inputs are available, it performs the addition.
3. It then pushes the result to an output.

Such a simple operand may have two possible implementations. The first one defines a executable component (Meandre terminology for a black box operator) with two inputs. When data is present on both inputs, then the operator is executed—*fired*. The operator produces one piece of data to output, which may become the input of another operator. Another possible implementation is to create a component with a single input that adds together two consecutive data pieces received. The component requires an internal variable (or state) which stores the first data piece of a pair. When the second data piece arrives, it would be added to the first and an output is produced. The internal variable would then be cleared and the component will treat the next data piece received as the first of a new pair. As we will see later in this paper, both implementations have merit, but in certain conditions we will choose one over the other based on clarity and efficiency requirements.

Meandre uses the following terminology:

1. *Executable component*: A basic unit of processing.
2. *Input port*: Input data required by a component.
3. *Firing policy*: The policy of component execution (e.g. when all/any input ports contain data).
4. *Output port*: Outputs data produced by component execution.
5. *Properties*: Component read-only variables used to modify component behavior.
6. *Internal state*: The collection of data structures designed to manage data between component firings.

Figure 2 presents a schema of the component and flow anatomy. 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 components⁴.

2.2.2 Components

Meandre components serve as the basic *building block* of any computational task. There are two kinds of Meandre components: (1) *executable components* and (2) *flow components*. Regardless of type, all Meandre components are described using metadata expressed in RDF. Executable components also require an executable implementation form that can be understood by the Meandre execution engine⁵.

⁴ Meandre uses a *decentralized scheduling policy* designed to maximize the use of multicore architectures. Meandre also allows works with processes that require directed cyclic graphs—extending beyond the traditional MapReduce directed acyclic graphs.

⁵ Java, Python, and Lisp are the current languages supported by Meandre to implement a component.

Meandre’s metadata relies on three ontologies: (1) the RDF ontology [5, 7] serves as a base for defining Meandre components; (2) the Dublin Core elements ontology [43] provides basic publishing and descriptive capabilities in the description of Meandre components; and (3) the Meandre ontology describes a set of relationships that model valid components, as understood by the Meandre architecture—refer to [29] for a more detailed description.

2.2.3 Programming Paradigm

The programming paradigm creates complex tasks by linking together a bunch of specialized components—see Figure 2. Meandre’s publishing mechanism allows components developed by third parties to be assembled in a new flow. There are two ways to develop flows on Meandre: (1) using visual programming tools, or (2) using Meandre’s ZigZag scripting language—see [29]. For simplicity purposes, throughout the rest of this paper flows will be presented as ZigZag scripts.

3 Three Diverse Genetic Algorithms Models

Evolutionary computing encompass a large diversity of algorithms and implementations. In order to illustrate the usefulness of data-intensive computing, we will focus on three widely used models: selectorecombinative genetic algorithms [15, 16], the compact genetic algorithm [20], and the extended compact genetic algorithm [21]. As we will describe in the rest of this section, each of these algorithms posses different profiles. Ranging from purely population-based to model-based algorithms, to create their data-intensive computing counterparts—as we will show in the next section—will require to pay close attention to their basic needs.

3.1 A Simple Selectorecombinative Genetic Algorithm

Selectorecombinative genetic algorithms [15, 16] mainly rely on the use of selection and recombination. We chose to start with them because they present a minimal set of operators that will help us illustrate the creation of a data-intensive flow counterpart. As we will see, the addition of mutation operators will be trivial after the setting up the proper data-intensive flow. The rest of this section will present a quick description of the algorithm we transformed and implemented it using Meandre, a discussion of some of the elements that need to be taken into account, and finally review the execution profile of the final implementation.

The basic algorithm we will 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 [18].
4. Create new individuals by recombining the selected population using uniform crossover⁶[41].
5. Evaluate the fitness value of all offspring.
6. Repeat steps 3–5 until some convergence criteria are met.

3.2 The Compact Genetic Algorithm

The compact genetic algorithm [20], is one of the simplest estimation distribution algorithms (EDAs) [35, 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 [18] 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,

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

⁶ For this particular exercise we have assumed a crossover probability $p_\chi=1.0$.

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) [3, 4] and the univariate marginal distribution algorithm (UMDA) [34, 33]. 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 [20]. 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-sizing⁷ model proposed by Harik et al. [19]:

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

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.

3.3 The Extended Compact Genetic Algorithm

The extended compact genetic algorithm (eCGA) [21], 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,

⁷ The experiments conducted in this paper used $n = 3\ell$.

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 [?], 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 [18].
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,

$$\text{Minimize } C_m + C_p \quad (3)$$

Subject to

$$\chi^{k_i} \leq n \quad \forall i \in [1, m] \quad (4)$$

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

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

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

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

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 $j \in [1, \chi^{k_i}]^8$ 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 [39]. 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

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

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 [17], 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).

4 Data-Intensive Computing in Action

The previous section described the three algorithms we will target to create their data-intensive computing counterparts. This section we will take a stab at designing efficient and scalable version of these algorithms to show the benefits of banking on either MapReduce or Meandre approaches.

4.1 A Simple Selectorecombinative Genetic Algorithm

4.1.1 MapReducing SGAs

In this section, we start with a simple model of Genetic algorithms and then transform and implement it using MapReduce. We encapsulate each iteration of the GA as a

⁸ Note that a BB of length k has χ^k possible sequences where the first sequence denotes be $00 \dots 0$ and the last sequence $(\chi - 1)(\chi - 1) \dots (\chi - 1)$

Listing 1 Map phase of each iteration of the Genetic Algorithm

```

procedure Initialization :
begin
    max := -1
end

procedure Map(key, value) :
begin
    individual := IndividualRepresentation(key)
    fitness := CalculateFitness(individual)
    Emit(individual, fitness)
    {Keep track of the current best}
    if fitness > max then
        max := fitness
        maxInd := individual
    {Finished all local maps}
    if processed_all_individuals then
        Write best individual to global file in DFS
end

```

separate MapReduce job. The client accepts the commandline parameters, creates the population and submits the MapReduce job.

Map Evaluation of the fitness function for the population matches the MAP function, which has to be computed independent of other instances. As shown in the algorithm in Listing 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)⁹. The client reads these values from all the mappers at the end of the MapReduce and determines whether to start the next iteration.

Partitioner If the selection operation in a GA (Step 3) is performed locally on each node, it reduces the selection pressure [37] and can lead to increase in the time taken to converge. Hence, decentralized and distributed selection algorithms [9] are preferred. The only point at which there is a global communication is in the shuffle between the Map and Reduce. At the end of the Map phase, the MapReduce framework shuffles the key/value pairs to the reducers using the partitioner. The partitioner splits the intermediate key/value pairs among the reducers. The function GETPARTITION() returns the reducer to which the given (*key, value*) should be sent to. In the default implementation, it uses $\text{HASH}(\textit{key}) \% \textit{numReducers}$ so that all the values corresponding to a given *key* end up at the same reducer which can then apply the REDUCE function. However, this does not suit the needs of Genetic algorithms because of two reasons: Firstly, the HASH function partitions the namespace of the individuals N into r distinct classes : $\{N_0, N_1, \dots, N_{r-1}\}$

⁹ This cleanup functionality can be implemented by overriding the *close()* function, but it's overlaps with *reduce()* function and hence sometimes throws *FileNotFoundException*

Listing 2 Random partitioner for the Genetic Algorithm

```
int getPartition(key, value, numReducers):
    return RandomInt(0, numReducers - 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 one single reducers 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 Listing 2.

Reducer We implement Tournament selection without replacement[18]. A tournament is conducted among $tSize$ 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 Listing 3. When the crossover window is full, we use the Uniform Crossover operator. For our implementation, we set the $tSize$ to 5 and the $cSize$ to 2.

Optimizations After initial experimentation, we noticed that for larger problem sizes, the serial initialization of the population takes a long time. According to Amdahl's law, the speedup is bounded because of this serial component. Hence, we create the initial population in a separate MapReduce phase, in which the `MAP` generates random individuals and the `REDUCE` is the Identity Reducer.¹⁰ We seed the pseudo-random number generator for each mapper with $mapperId \cdot currentTime$. 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 be executed till the convergence criteria is satisfied.

¹⁰ Setting the number of reducers to 0 in Hadoop removes the extra overhead of shuffling and identity reduction.

Listing 3 Reduce phase of each iteration of the Genetic Algorithm

```

procedure Initialization :
begin
    processed := 0
    Allocate tournamentArray [1 ... 2*tSize]
    Allocate crossoverArray [cSize]
end

procedure Reduce(key, values):
begin
    while values.hasNext()
    begin
        individual := Individual_representation(key)
        fitness := values.getValue()
        if processed < tSize
        then
            {Wait for individuals to join in the tournament and put
             them for the last rounds}
            tournamentArray [tSize + processed*tSize] := individual
        else
            {Conduct a tournament over the past window}
            SelectionAndCrossover()
            processed := processed + 1

            {Finished all reduces}
            if processed_all_individuals
            then
                {Cleanup for the last tournament windows}
                for k=1 to tSize
                begin
                    SelectionAndCrossover()
                    processed := processed + 1
                end
            end
        end
    end

procedure SelectionAndCrossover :
begin
    crossoverArray[processed*cSize] := Tournament(tournamentArray)
    if (processed-tSize)%cSize = cSize - 1
    then
        {Perform crossover whenever the crossover window is full}
        newIndividuals := Crossover(crossoverArray)
        for individual in newIndividuals
            Emit(individual, dummyFitness)
    end

```

4.1.2 SGAs as Data-Intensive Flows

The first step in designing a data-intensive flow implementation of the algorithm presented in the previous section is to identify what data will be processed. This decision is similar to the *partition* step of the methodology proposed by Foster [12], to design general purpose parallel programs, where data is maximally partitioned to maximize parallelization. The two possible options here are to deal with populations or individuals. E2K [26]—a data-flow extension for D2K [44]—chose to use populations. Such a decision only allows parallelizing the concurrent evolution of distinct populations. In this paper we will choose the second option. Our data-flow implementation is going to be built around processing individuals. In other words, a population will be a stream of individuals—a *stream initiator* and a *stream terminator* will enclose each stream defining a population. Making the decision of processing streams of individuals will allow creating components that perform the genetic manipulation as the stream goes by. This approach may be regarded as an analogy of pipeline segmentation on central processing units.

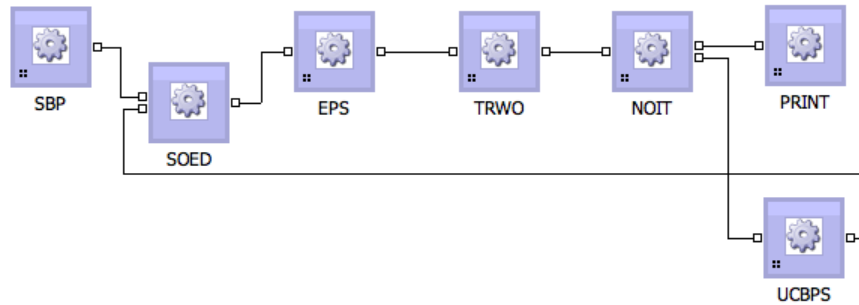


Fig. 3 Meandre's flow for the proposed selectorecombinative genetic algorithm flow.

Inspecting the algorithm presented in section 3.1, we need to create components that implement each of the operation required: *initialization*, *evaluation*, *selection*, and *recombination*. *Initialization* and *evaluation* are straight forward; the first one creates the stream of individuals that forms a population, and the second one updates the fitness of the individuals as they are streamed by. The *recombination* components—as introduced in section 2.2—will require an internal state. As individuals stream by, it requires two individuals in order to perform the uniform crossover operation.

The *selection* component requires a bit more thinking. One may think that a similar implementation to the one used for the *recombination* component may work approximately accurate enough. However, such an implementation would be equivalent to implement a spatial based selection method instead of a tournament one. Spatially constraint selection methods have been shown to elongate the takeover time, and thus reduce the selection pressure when compared to tournament selection without replacement [9, 37, 38, 28, 14]. Also, following on the temptation of accumulate all

Listing 4 Portion of the ZigZag script implementing a selectorecombinative genetic algorithm using Meandre components.

```

#
# Main import and aliases
#

# Omitted ...

#
# Component instances
#
sbp, soed, eps = SBP(), SOED(), EPS()
noit, twrops, ucbps = NOIT(), TWROPS(), UCBPS()
print = PRINT()
#
# The flow
#
@new_pop, @cross_pop = sbp(), ucbps()
@pop_ed = soed(
    initial_stream : new_pop.population ;
    stream : cross_pop.population
)
@eval_pop = eps(population : pop_ed.stream)
@sel_pop = twrops(population : eval_pop.population)
@cnt = noit(population : sel_pop.population)
ucbps(population : cnt.population)
print(population : cnt.final_population)

```

the individuals and then recreate the stream as we conduct tournaments against the accumulated population also seems prone to introduce a large sequential bottleneck and, thus, leaving the execution profile prone to the Amdahl's law [2]. The answer is simpler. Create all the required tournaments when you get the *stream initiator*. Then as individuals are streamed in perform the possible tournaments and start streaming the new selected population. Thus, we will guarantee that as individuals are still streamed in, we are already streaming out of the component a newly selected population, minimizing Amdahl's law impact.

Figure 3 presents the components discussed above and how they get assembled to for the final data-intensive flow. The `sbp` (stream binary population) streams a population of individuals to start the flow. Individuals get streamed into the `soed` (single open entry door) component. This is a special component. Its only goal is to make sure that all the individuals on the initial population are streamed into the evaluation `eps` component before the next streamed population arrives. The goal of this component is to avoid having individuals from the next population mixed with the previous population ones. It is important to note here that individuals may still be streamed in from the initialization when new individuals are already being streamed out the recombination `ucbps` component, and hence, we must guarantee that the two populations do not get mixed. Evaluated individuals are then streamed

into the tournament selection `twrops` component, and the selected individuals are streamed into the `noit` (number of iterations) component which allows a population stream to go by a given number of iterations and then diverts the population to a secondary output port to print the final output to the console. If the finalization criteria is not met, the selected individuals are streamed into the recombination `ucbps` component. New offspring will be sent for evaluation passing through the `soed` safe gate. Program 4 presents the ZigZag script implementing this flow. A detailed description on how to implement Meandre components and write ZigZag scripts can be found elsewhere—see [29] and <http://seasr.org/meandre>.

4.2 The Compact Genetic Algorithm and Hadoop

We encapsulate each iteration of the CGA as a separate single MapReduce job. The client accepts the commandline parameters, creates the initial probability vector splits and submits the MapReduce job. Let the probability vector be $P = \{p_i : p_i = \text{Probability_of_the_variable}(i) = 1\}$. Such an approach would allow us to scale over a billion variables, if P is partitioned into m different partitions P_1, P_2, \dots, P_m where m is the number of mappers.

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 Listing 5, 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.

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 Listing 6.

Optimizations 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 ints** and we use efficient bit operations for crossover and fitness calculations. Also, we use **long long ints** to represent probabilities instead of floating point numbers and use the more efficient integer operations.

Listing 5 Map phase of each iteration of the CGA

```

procedure Map(key, value):
begin
  splitNo := key
  probSplitArray := value
  Emit(splitNo, [0, probSplitArray])
  for k := 1$ to tournamentSize
  begin
    individual := nil
    ones := 0
    for prob in probSplitArray
    begin
      if Random(0,1) < prob
      then
        individual := 1
        ones := ones + 1
      else
        individual := 0
      Emit(splitNo, [k, individual])
      WritetoDFS(k, ones)
    end
  end
end

```

4.3 The Extended Compact Genetic Algorithm and Meandre

As we did with the selectorecombinative genetic algorithm, and loosely following Foster’s methodology [12], we will identify what data is going to drive our execution. In this particular case, the relevant pieces of information used by eCGA’s model building are the gene partitions used to compute the MPM. The greedy model-building algorithm requires exploring a large number of possible partition merges while building the model—being $\mathcal{O}(\ell^3)$ the worst case scenario. Thus, this would suggest that the partitions of genes should be the basic piece of data to stream. At each step of the model building process, a stream of partitions will need to be evaluated to compute each combined complexity score. The evaluation of each partition is also independent of each other, further simplifying the overall design.

Figure 4 presents the four components we will use to implement a data-intensive version of eCGA model builder. The `init_ecga` component creates a new population, evaluates the individuals (using and MK deceptive trap [16] where $k = 4$ and $d = 0.25$), pushes the selected population obtained using tournament selection without replacement ($s = 6$), and starts streaming the initial set of gene partitions that require evaluation. Then, the `update_partition` component computes the combine complexity of the partition and streams that information to the `greedy_ecga_mb` component. This component implements the greedy algorithm that receives the evaluated partitions and decides which ones to merge. In case that a partition merge is possible, the new set of partitions to be evaluated are

Listing 6 Reduce phase of each iteration of the CGA

```

procedure Initialize :
begin
  Allocate_and_initialize ( OnesArray[ tournamentSize ] )
  winner := -1
  loser := -1
  processed := 0
  n := 0
  for k:=1 to tournamentSize
  begin
    for r=1 to numReducers
    do
      Ones[k] := Ones[k] + ReadFromDFS(r, k)
      if Ones[k] > winner
      then
        winnerIndex := k
      else
        if Ones[k] < loser
        then
          loserIndex := k
      end
    end
  end

procedure Reduce(key, values):
  while values.hasNext()
  begin
    splitNo := $key
    value[processed] := values.getValue()
    processed := processed + 1
  end
  for prob in value[0]
  begin
    if value[winner].bit[n] != value[winner][n]
    then
      if value[winner].bit[n] = 1
      then
        newProbSplit [n] := value[0] + 1/population
      else
        newProbSplit [n] := value[0] - 1/population
      end
    end
    Emit(splitNo, [0, newProbSplit])
  end

```

streamed into the `update_partition` component. If no merger is possible, the `greedy_ecga_mb` component pushes the final MPM model to the `print_model` component. Program 7 present the ZigZag script implementing the data-intensive computing version of eCGA.

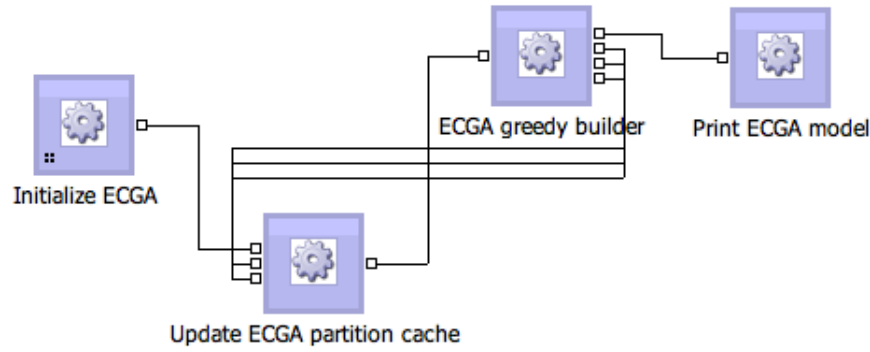


Fig. 4 Meandre's flow for eCGA.

5 Experiments

This section presents the results obtained using the Hadoop and Meandre frameworks to scale the proposed GAs and EDAs. The section begins presenting the results achieved using both frameworks to speedup traditional genetic algorithms. Then it reviews the results obtained using Hadoop to speedup cGA—more fitted to tackle large data-volume problems with relatively easy to implement algorithms. Finally the section concludes presenting the promising scalability results achieved using Meandre on eCGA being, to the best of our knowledge, one of the first attempts that has succeeded in showing that efficient parallelization of eCGA model building is possible.

5.1 Selectorecombinative Genetic Algorithms

To illustrate the benefits of both frameworks, we implemented and tested the selectorecombinative genetic algorithm following the descriptions presented above.

Listing 7 ZigZag script implementing the extended compact genetic algorithm using Meandre components.

```

#
# Main import and aliases
#

# Omitted ...

#
# Component Instances
#
init_ecga , greedy_ecga_mb = INIT_ECGA() , GREEDY_ECGA_MB()
update_partitions = UPDATE_ECGA_PARTITIONS()
print_model , print_pop = PRINT_MODEL() , PRINT_POP_MATRIX()
#
# The flow
#
@init_ecga = init_ecga()
@update_part = update_partitions(
    ecga_partition_cache :
        init_ecga.ecga_partition_cache ;
    ecga_partition_to_update_i :
        init_ecga.ecga_partition_to_update_i ;
    ecga_partition_to_update_j :
        init_ecga.ecga_partition_to_update_j
)
@greedy_mb = greedy_ecga_mb(
    ecga_partition_cache :
        update_part.ecga_partition_cache
)
update_partitions(
    ecga_partition_cache :
        greedy_mb.ecga_partition_cache ;
    ecga_partition_to_update_i :
        greedy_mb.ecga_partition_to_update_i ;
    ecga_partition_to_update_j :
        greedy_mb.ecga_partition_to_update_j
)
print_model(ecga_model : greedy_mb.ecga_model)

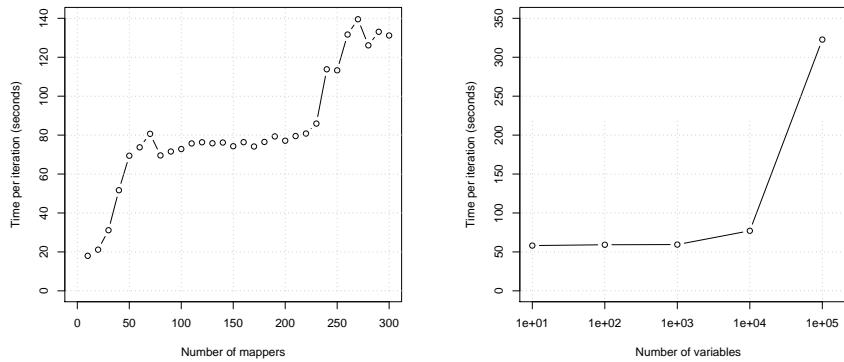
```

5.1.1 Hadoop and SGAs

We implemented the simple ONEMAX problem on Hadoop (0.19)¹¹ 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 can be found

¹¹ <http://hadoop.apache.org>

elsewhere¹². 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 finished 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.



(a) Scalability of selectorecombinative genetic algorithm with constant load per node for the ONEMAX problem.

(b) Scalability of selectorecombinative genetic algorithm for ONEMAX problem with increasing number of variables.

Fig. 5 Results obtained using Hadoop when implementing a simple selectorecombinative genetic algorithm

We ran two sets of experiments. In the first one, we kept the load set to 1,000 variables per mapper. As shown in Figure 5(a), 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 $5 \cdot 52(\text{nodes}) = 260$. Hence, around 250 mappers, the time per iteration increases due to the lack of resources to accommodate so many mappers. In the second set of experiments, we utilize the maximum resources and increase the number of variables. As shown in Figure 7(b), 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.

¹² <http://cloud.cs.illinois.edu>

5.1.2 Meandre and SGAs

We run some experiments to illustrate the properties of data-intensive computing modeling. Unless noted otherwise, the experiments were run on an Intel 2.8GHz Quad Core equipped with 4Gb of RAM, running Ubuntu Linux 8.0.4, and Sun JVM 1.5.0_15. The problem we solved was a relatively small OneMax [15, 16], for 5,000 bits and a population size of 10,000—details on the population sizing can be found elsewhere [16]. The goal of the experiment was to reveal the execution profile of the converted algorithm. Figure 6(a) presents the average time spent by each component. Times are averaged over 20 runs, all evolving the optimal solution. The first thing to point out is that evaluation is not the most expensive part of the execution. OneMax is so simple, that the cost of selection and crossover dominates the execution.

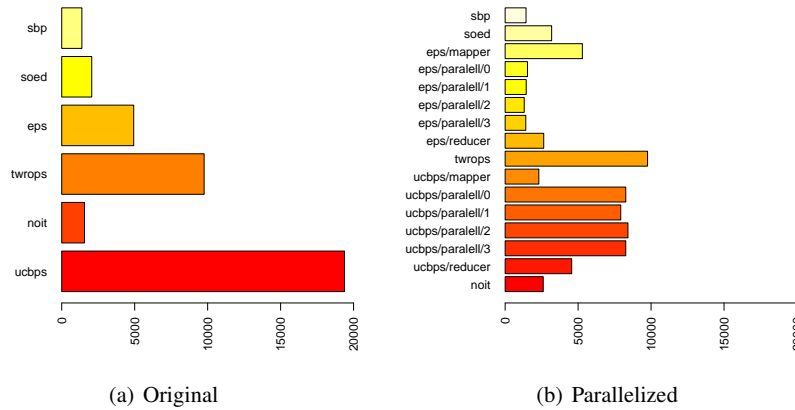


Fig. 6 Execution profile of the original data-intensive flow implementing a selectorecombinative genetic algorithm and its automatically parallelized version of `epb` and `ucbps` components (parallelization degree equal to the number of available cores, 4). Times are in milliseconds and are averages over twenty runs.

Such counter intuitive profile would be a problem if we took a traditional parallelization route based on master/slave configurations delegating evaluations [8]—which works its best on the presence on costly evaluation functions. Thanks to choosing a data-intensive computing—and Meandre’s ability to automatically parallelize components¹³—we can also automatically parallelize the costly part of the execution: the uniform crossover¹⁴. Also, we can, at the same time parallelize the evaluation, which in this situation may have little effect. However, the key property to highlight is that either in this cases, or in the case of having a costly evaluation func-

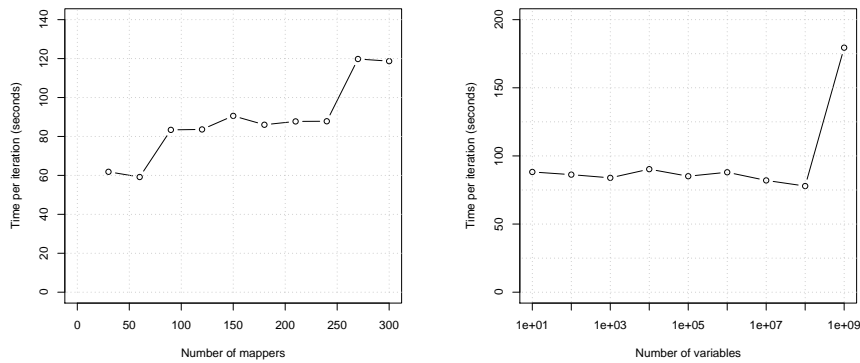
¹³ Multiple instance of a component can be spawned to process in parallel incoming individuals.

¹⁴ Same considerations would apply in case of having a mutation component.

tion, the underlying data-intensive flow algorithm does not need to be changed, and component parallelization will help, for a given problem, parallelize the costly parts of the execution profile—see Figure 6(b). Hence, the inherent nature of data-intensive computing can help focus attention where is really needed. Also, parallelization, can introduce new bottlenecks—see `twrops` times on Figure 6(b)—, which now we could also parallelize to make such bottleneck disappear. Next section will show the scalability benefits that this data-intensive approach can help unleash.

5.2 The Compact Genetic Algorithm and Hadoop

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 7(a), 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 $5 \cdot 52(\text{nodes}) = 260$. 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.



(a) Scalability of compact genetic algorithm with constant load per node for the ONEMAX problem.

(b) Scalability of compact genetic algorithm for ONEMAX problem with increasing number of variables.

Fig. 7 Results obtained using Hadoop when implementing a the compact genetic algorithm.

In the second set of experiments, we utilized the maximum resources and increase the number of variables. As shown in Figure 7(b), our implementation scales to $n = 10^8$ variables, keeping the population set to $n \log n$.

5.3 The Extended Compact Genetic Algorithm and Meandre

We ran three different experiments. First we measure the executions profile of the implement data-intensive eCGA. Figure 9(a) presents the average time spend in each component over 20 runs— $\ell = 256$ and $n = 100,000$. All the runs lead to learning the optimal MPM model thanks to the oversized population. Figure 9(a) highlights the already known fact that most of the execution time of the model building process is spent evaluating the gene partitions. Also, the initialization is being negatively affected by the partition updates, since it is being held back since it produces partitions much faster than they can be evaluated. This fact can be observed on Figure 9(b). After providing four parallelized partition evaluation components, not only the overall wall clock time drop, but also the initialization time too, since now, the update input queues can keep up with the initial set of partitions generated.

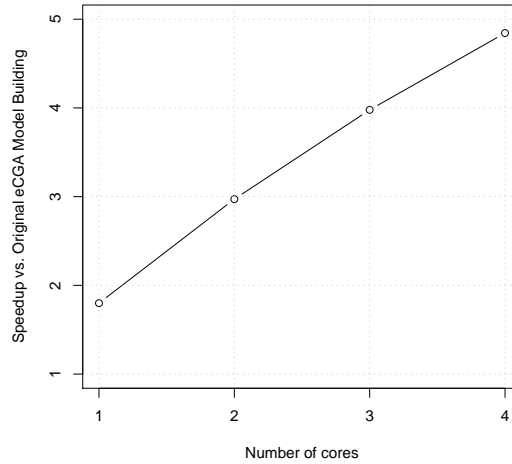


Fig. 8 eCGA speedup compared to the original non data-intensive computing implementation. Figure show the speedup as a function of the number of `update_partitions` parallelized components available.

We repeated these experiments providing $\{1,2,3, \text{ and } 4\}$ parallelized `update_partition` components and measured the overall speedup against the traditional eCGA model building implementation. Figure 8 presents the speedup results

graphically. The first element to highlight is that, only using one `update_partitions` component instance we obtained a superlinear speedup. This is the result of few improvements on the implementation of the components, which allowed to remove extra layers of unnecessary function calls present in the original code. Also, the fact that partitions results are streamed into the greedy model builder adds and extra improvement—similar to pipeline segmentation as discussed earlier and the availability of idle cores¹⁵—by advancing computations instead of waiting for the last partition to be calculated. The final speedup graph shows a clear linear increase in performance as more cores are efficiently used despite resources contention when using all the cores available. Finally, we ran the same experiment on a SGI Altix machine with a multiprocessor NUMA architecture at the National Center for Supercomputing Applications (NCSA)¹⁶ and requested 16 and 32 nodes. The averaged speedup was computed over 20 independent runs. Again, the speedup showed a linear speedup of 14.01 and 27.96 of 16 and 32 processors. The slight drop on performance is the results of memory contention of the NUMA interconnection architecture of the SGI Altix machine.

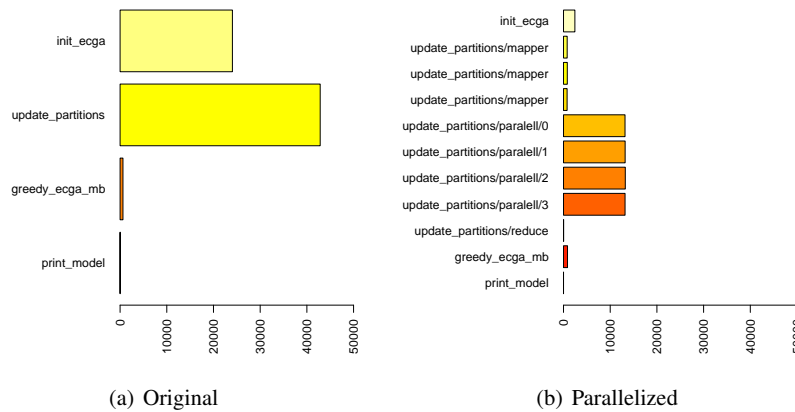


Fig. 9 Execution profile of the original data-intensive flow implementing eCGA and its automatically parallelized version of its `update_partitions` component (parallelization degree equal to the number of available cores, 4). Times are in milliseconds and are averages of twenty runs.

¹⁵ The hardware used for this experiment did not provide a fair way to execute the data-intensive flow using only one core. If that could have been possible, a normal linear speedup curve would have been obtained when extra cores were added and the time of executing on one core used to compute the speedup instead of the time of the original sequential implementation.

¹⁶ <http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/SGIAltix/TechSummary/>

6 Related Work

Several different models like fine grained [30], coarse grained [25] and distributed [24] models have been proposed for implementing parallel GAs. Traditionally, 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 fault tolerance is necessary. MapReduce [10] is a programming model that enables the users to easily develop large-scale distributed applications. Hadoop¹⁷ is an open source implementation of the MapReduce model. Several different implementations of MapReduce have been developed for other architectures: Phoenix [36] for multicores, CGL-MapReduce [11] for streaming applications.

To the best of our knowledge, MRPGA [22] 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 [10]. 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.

7 Conclusion

In this paper we have shown that implementing evolutionary computation algorithms using a data-intensive computing paradigms is possible. We have presented step-by-step transformations for three illustrative cases—selectorecombinative genetic algorithms and estimation of distribution algorithms—and reviewed some best practices during the process. Transformations have shown that either Hadoop’s MapReduce model, or Meandre’s semantic-driven data-intensive flows can help

¹⁷ <http://hadoop.apache.org>

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 using Meandre 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.

We are current exploring how the extended compact genetic algorithm could be implemented using a MapReduce paradigm, as well as finishing Meandre's implementation of the compact genetic algorithm. Our future work is focused on analyzing other evolutionary computation algorithms that may display different execution profiles than the ones used in this book chapter, and what challenges they may face when developing their data-intensive computing counterparts.

Acknowledgments

This work was funded, in part, by NSF IIS Grant #0841765. We would also like to thank The Andrew W. Mellon Foundation and the National Center for Supercomputing Applications for their funding and support to the SEASR project that has made possible the development of the Meandre infrastructure.

References

1. Alba, E. (ed.): *Parallel Metaheuristics*. Wiley (2007)
2. Amdahl, G.: Validity of the single processor approach to achieving large-scale computing capabilities. In: *AFIPS Conference Proceedings*, pp. 483–485 (1967)
3. Baluja, S.: 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)
4. Baluja, S., Caruana, R.: Removing the genetics from the standard genetic algorithm. Tech. Rep. CMU-CS-95-141, Carnegie Mellon University (1995)
5. Beckett, D.: *RDF/XML Syntax Specification (Revised)*. W3C Recommendation 10 February 2004, The World Wide Web Consortium (2004)
6. Beynon, M.D., Kurc, T., Sussman, A., Saltz, J.: Design of a framework for data-intensive wide-area applications. In: *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, p. 116. IEEE Computer Society, Washington, DC, USA (2000)
7. Brickley, D., Guha, R.: *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C Recommendation 10 February 2004, The World Wide Web Consortium (2004)
8. Cantú-Paz, E.: *Efficient and Accurate Parallel Genetic Algorithms*. Springer (2000)

9. De Jong, K., Sarma, J.: On decentralizing selection algorithms. In: In Proceedings of the Sixth International Conference on Genetic Algorithms, pp. 17–23. Morgan Kaufmann (1995)
10. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI'04: Sixth Symposium on Operating System Design and Implementation (2004)
11. Ekanayake, J., Pallickara, S., Fox, G.: Mapreduce for data intensive scientific analyses. In: ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience, pp. 277–284. IEEE Computer Society, Washington, DC, USA (2008). DOI <http://dx.doi.org/10.1109/eScience.2008.59>
12. Foster, I.: Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Addison Wesley (1995)
13. Foster, I.: The virtual data grid: A new model and architecture for data-intensive collaboration. In: in the 15th International Conference on Scientific and Statistical Database Management, pp. 11– (2003)
14. Giacobini, M., Tomassini, M., Tettamanzi, A.: Takeover time curves in random and small-world structured populations. In: GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation, pp. 1333–1340. ACM, New York, NY, USA (2005). DOI <http://doi.acm.org/10.1145/1068009.1068224>
15. Goldberg, D.E.: Genetic algorithms in search, optimization, and machine learning. Addison-Wesley, Reading, MA (1989)
16. Goldberg, D.E.: The Design of Innovation: Lessons from and for Competent Genetic Algorithms. Kluwer Academic Publishers, Norwell, MA (2002)
17. Goldberg, D.E., Deb, K., Clark, J.H.: Genetic algorithms, noise, and the sizing of populations. *Complex Systems* **6**, 333–362 (1992). (Also IlliGAL Report No. 91010)
18. Goldberg, D.E., Korb, B., Deb, K.: Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems* **3**(5), 493–530 (1989)
19. Harik, G., Cantú-Paz, E., Goldberg, D.E., Miller, B.L.: The gambler's ruin problem, genetic algorithms, and the sizing of populations. *Evolutionary Computation* **7**(3), 231–253 (1999). (Also IlliGAL Report No. 96004)
20. Harik, G., Lobo, F., Goldberg, D.E.: The compact genetic algorithm. Proceedings of the IEEE International Conference on Evolutionary Computation pp. 523–528 (1998). (Also IlliGAL Report No. 97006)
21. Harik, G.R., Lobo, F.G., Sastry, K.: Linkage learning via probabilistic modeling in the ECGA. In: M. Pelikan, K. Sastry, E. Cantú-Paz (eds.) *Scalable Optimization via Probabilistic Modeling: From Algorithms to Applications*, chap. 3. Springer, Berlin (in press). (Also IlliGAL Report No. 99010)
22. Jin, C., Vecchiola, C., Buyya, R.: MRPGA: An extension of mapreduce for parallelizing genetic algorithms. In: I. Press (ed.) *IEEE Fourth International Conference on eScience 2008*, pp. 214–221 (2008)
23. Larrañaga, P., Lozano, J.A. (eds.): *Estimation of Distribution Algorithms*. Kluwer Academic Publishers, Boston, MA (2002)
24. Lim, D., Ong, Y.S., Jin, Y., Sendhoff, B., Lee, B.S.: Efficient hierarchical parallel genetic algorithms using grid computing. *Future Gener. Comput. Syst.* **23**(4), 658–670 (2007). DOI <http://dx.doi.org/10.1016/j.future.2006.10.008>
25. Lin, S.C., Punch, W.F., Goodman, E.D.: Coarse-grain parallel genetic algorithms: Categorization and new approach. In: *Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing*, pp. 28–37 (1994)
26. Llorà, X.: E2K: evolution to knowledge. *SIGEVOLUTION* **1**(3), 10–17 (2006). DOI <http://doi.acm.org/10.1145/1181964.1181966>
27. Llorà, X.: Data-intensive computing for competent genetic algorithms: A pilot study using meandre. In: *Proceedings of the 2009 conference on Genetic and evolutionary computation (GECCO 2009)*. ACM Press, Montreal, Canada (2009, in press)
28. Llorà, X.: *Genetic Based Machine Learning using Fine-grained Parallelism for Data Mining*. Ph.D. thesis, Enginyeria i Arquitectura La Salle. Ramon Llull University, Barcelona (February, 2002)

29. Llorà, X., Ács, B., Auvil, L., Capitanu, B., Welge, M., Goldberg, D.E.: 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)
30. Maruyama, T., Hirose, T., Konagaya, A.: A fine-grained parallel genetic algorithm for distributed parallel systems. In: Proceedings of the 5th International Conference on Genetic Algorithms, pp. 184–190. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1993)
31. Mattmann, C.A., Crichton, D.J., Medvidovic, N., Hughes, S.: 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, pp. 721–730. ACM, New York, NY, USA (2006). DOI <http://doi.acm.org/10.1145/1134285.1134400>
32. Morrison, J.P.: Flow-Based Programming: A New Approach to Application Development. Van Nostrand Reinhold (1994)
33. Mühlenbein, H.: The equation for response to selection and its use for prediction. *Evolutionary Computation* **5**(3), 303–346 (1997)
34. Mühlenbein, H., Paaß, G.: From recombination of genes to the estimation of distributions I. Binary parameters. *Parallel Problem Solving from Nature, PPSN IV* pp. 178–187 (1996)
35. Pelikan, M., Lobo, F., Goldberg, D.E.: A survey of optimization by building and using probabilistic models. *Computational Optimization and Applications* **21**, 5–20 (2002). (Also IlliGAL Report No. 99018)
36. Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating mapreduce for multicore and multiprocessors systems. In: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (2007)
37. Sarma, J., De Jong, K.: An analysis of local selection algorithms in a spatially structured evolutionary algorithm. In: Proceedings of the Seventh International Conference on Genetic Algorithms, pp. 181–186. Morgan Kaufmann (1997)
38. Sarma, J., De Jong, K.: Selection pressure and performance in spatially distributed evolutionary algorithms. In: In Proceedings of the World Congress on Computational Intelligence, pp. 553–557. IEEE Press (1998)
39. Sastry, K., Goldberg, D.E.: 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
40. Sastry, K., Goldberg, D.E., Llorà, X.: 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, pp. 577–584. ACM, New York, NY, USA (2007). DOI <http://doi.acm.org/10.1145/1276958.1277077>
41. Sywerda, G.: Uniform crossover in genetic algorithms. In: Proceedings of the third international conference on Genetic algorithms, pp. 2–9. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1989)
42. Uysal, M., Kurc, T.M., Sussman, A., Saltz, J.: 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, pp. 243–258. Springer-Verlag (1998)
43. Weibel, S., Kunze, J., Lagoze, C., Wolf, M.: Dublin Core Metadata for Resource Discovery. Tech. Rep. RFC2413, The Dublin Core Metadata Initiative (2008)
44. Welge, M., Auvil, L., Shirk, A., Bushell, C., Bajcsy, P., Cai, D., Redman, T., Clutter, D., Ayt, R., Tchong, D.: Data to Knowledge (D2K). Tech. rep., Technical Report Automated Learning Group, National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign (2003)