Towards Billion-Bit Optimization via a Parallel Estimation of Distribution Algorithm

Kumara Sastry[1,2], David E. Goldberg[1], and Xavier Llorà[1,3]

[1] Illinois Genetic Algorithms Laboratory (IlliGAL), Industrial and Enterprise Systems Engineering

[2]Materials Computation Center

[3]National Center for Super Computing Applications (NCSA)

University of Illinois at Urbana-Champaign, Urbana IL 61801

ksastry@uiuc.edu, deg@uiuc.edu, xllora@uiuc.edu

ABSTRACT

This paper presents a highly efficient, fully parallelized implementation of the compact genetic algorithm (cGA) to solve very large scale problems with millions to billions of variables. The paper presents principled results demonstrating the scalable solution of a difficult test function on instances over a billion variables using a parallel implementation of cGA. The problem addressed is a noisy, blind problem over a vector of binary decision variables. Noise is added equaling up to a tenth of the deterministic objective function variance of the problem, thereby making it difficult for simple hillclimbers to find the optimal solution. The compact GA, on the other hand, is able to find the optimum in the presence of noise quickly, reliably, and accurately, and the solution scalability follows known convergence theories. These results on noisy problem together with other results on problems involving varying modularity, hierarchy, and overlap foreshadow routine solution of billion-variable problems across the landscape of search problems.

Categories and Subject Descriptors

G.1.6 [**Numerical Analysis**]: Optimization; G.4 [**Mathematics of Computing**]: Mathematical Software; D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Algorithms, Performance

Keywords

compact genetic algorithm, efficiency enhancement, parallelization, vectorization, population sizing, convergence time, scalability analysis, large-scale optimization, billion-variable optimization

GECCO'07, July 7–11, 2007, London, England, United Kingdom.

1. INTRODUCTION

Since the mid-1980s, genetic algorithms (GAs) [13, 6]—search procedures based on mechanics of natural selection and genetics—have been used to solve problems across the spectrum of human endeavor. Increasingly, GAs find answers to important scientific problems, but doubts remain about solution size, speed, and effectiveness despite efforts that demonstrate GA scalability in a principled manner [7]. GAs have often been criticized as being slow, suitable for optimizing problems with only a few variables, and that they do not scale to larger and more complex problems. Moreover, the push for scientific understanding formulates an array of challenging optimization problems with staggering number of decision variables. Despite these an other compelling needs, optimization today is generally limited to problems well under a million decision variables.

In this paper we show that the criticisms are somewhat unfounded and clearly demonstrate that GA scalability by presenting principled solutions to a representative, difficult problem over a billion variables. Specifically, we take the first step of designing a highly-efficient competent GA that can solve boundedly-difficult large scale problems with millions to billions of variables. Specifically, in this paper we develop a fully parallelized, highly-efficient compact genetic algorithm (cGA) [12, 1, 18]. The purpose of this paper is also to show that by utilizing a number of memory and computational efficiencies, we can design a GA that can potentially solve problems with millions to a billion binary variables. Moreover, we show that the as predicted cGA scales as $\Theta(\ell \log \ell)$—where ℓ is the number of binary variables—on a class of additively-separable problems with and without additive noise, and local search methods fail to do so in the presence of even a modest amount of noise.

This paper is organized as follows. We provide a brief literature review on studies involving large-scale optimization problems in the next section. In section 3, we provide details of the GA implementation that incorporates efficient memory utilization, and a number of computational efficiencies. A brief description of the test problems and the key results are discussed in section 5. Finally we summarize and present key conclusions in section 6.

2. LITERATURE REVIEW

Over the recent years, there has been a push to develop optimization procedures to tackle large-scale problems [2, 3, 5, 4, 10, 14, 19, 20, 25, 24]. Many of the aforementioned large-scale optimization studies have relied on linear programming solvers such as simplex methods and interior point methods. While these methods are very efficient in solving linear programming problems, they fall quite short of solving nonlinear, noisy, and deceptive problems. To the best of our knowledge, the largest known GA in the literature solves a problem with 4-million binary variables [5, 4, 24], but its applicability to other optimization problems remains in doubt because of problem-specific operators, small population used, and the lack of theory.

Our purpose is to develop a scalable and efficient GA that can adaptively solve boundedly-difficult large-scale problems with millions to billions of variables. To that effect, we take the first step of designing a highly-efficient compact genetic algorithm [12] to solve search problems with millions to billions of variables, where linear solvers fail. Doing so strongly suggests that extremely large practical scientific problems can be tackled using similar methods.

3. EFFICIENT COMPACT GENETIC ALGORITHM IMPLEMENTATION

In order to solve search problems with millions to billions of variables, we need a GA that is not only computationally efficient, but also efficient in terms of memory usage. Therefore, we use of the compact genetic algorithm [12, 1, 18] which is efficient in terms of its memory usage. Additionally, we make use of a number of memory and computational efficiencies in our implementation of the cGA. In this section we provide details of these efficiency enhancements and begin with a description of the algorithmic implementation of cGA [12, 1, 18].

3.1 Compact Genetic Algorithms

In the compact genetic algorithm [12], a population of candidate solutions is represented by a vector of probabilities. Specifically, each element in the vector represents the proportion of ones 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 GA 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. That is, 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. Therefore, cGA is operationally equivalent to the order-one behavior of simple genetic algorithm with steady state tournament selection and uniform crossover [12].

The compact GA 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 set to 0.5. However, other initialization procedures can also be used in a straightforward manner.

2. *Model sampling:* Generate two candidate solutions by sampling the probability vector.

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 [9] in cGA.

5. *Probability 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,

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

   Where, $\mathbf{x}_{w,i}$ is the i^{th} gene of the winning chromosome, $\mathbf{x}_{c,i}$ is the i^{th} gene of the competing chromosome, and p_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 population size, n, for cGA is given by the following approximate form of the gambler's ruin model [11, 8]:

$$n = \frac{\sqrt{\pi}}{2}\sqrt{\ell}\log\ell\sqrt{\left(1+\frac{\sigma_N^2}{\sigma_f^2}\right)}, \tag{2}$$

where ℓ is the problem size, σ_N^2 is the variance of of exogenous noise, and σ_f^2 is the fitness variance. The above equation assumes a failure probability, $\alpha = 1/\ell$.

The convergence time of the cGA is given by the following approximate form of a selection-intensity based convergence-time model [16, 7, 21]:

$$t_c = \frac{1}{2}\pi\sqrt{\pi}\sqrt{\ell}\sqrt{1+\frac{\sigma_N^2}{\sigma_f^2}}. \tag{3}$$

The population-sizing and convergence-time models indicate that the exogenous noise increases the population size and elongates the convergence time. Using equations 2 and 3, we can now predict the scalability, or the number of function evaluations required for successful convergence, of cGA as follows:

$$n_{\text{fe,cGA}} = \frac{\pi^2}{4}\ell\log\ell\left(1+\frac{\sigma_N^2}{\sigma_f^2}\right) \tag{4}$$

3.2 Efficient Memory Utilization via Compact Genetic Algorithm

With the use cGA, we eliminate the need to store the population, and thus significantly reduces the memory requirements when compared to traditional implementations of GAs. While the simple GA needs to store $n \cdot \ell$ bits to represent a population of n chromosomes with ℓ binary variables, cGA only needs to keep the proportion of ones, a

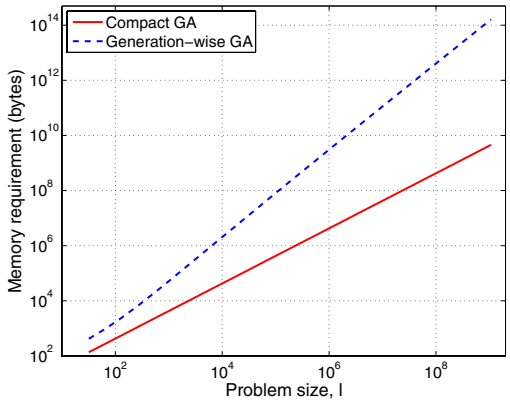


Figure 1: Comparison of memory-requirement estimation for compact genetic algorithm and simple generation-wise genetic algorithm. Memory requirement of cGA scales as $\Theta(\ell)$ as opposed to a simple GA which scales as $\Theta(\ell^{1.5})$.

finite set of n numbers that can be stored in $\log_2 n$ for each of the ℓ genes. That is, using cGA we reduce the memory requirements from $(\Theta(n \cdot \ell))$ to $(\Theta(\ell \log_2 n))$ to represent a population.

With cGA, when the population size is less than four billion, the probability of each variable can be represented by an unsigned integer (four bytes). Neglecting the memory required to generate random numbers, fitness values of two sampled individuals, etc., the two largest memory requirements for cGA are: (1) Probability vector: $4 * \ell$ bytes, and (2) sampling two individuals: $2*\ell/8 = \ell/4$ bytes. Therefore, the total memory required to store the probability vector and to sample two candidate solutions is given by

$$\mu_{\text{cGA}} = 4 * \ell + \ell/4 = \frac{17}{4} \cdot \ell. \tag{5}$$

In contrast, simple generation-wise GA implementations would require: (1) Parent population: $n \cdot \ell/8$ bytes, (2) Parent fitness: $4n$ bytes, (3) Mating pool: $4n$ bytes, (4) Offspring population: $n \cdot \ell/8$ bytes, and (4) Offspring fitness: $4n$ bytes. The above estimation assumes that parent and offspring population and their fitness values are stored. The selection procedure requires only the index of the selected parents to be stored. Therefore, the simple GA implementation would require a total memory of

$$\mu_{\text{GA}} = \frac{n \cdot \ell}{4} + 16n \tag{6}$$

From the population-sizing model (Equation 2), assuming no exogenous noise, the total memory requirement for the simple GA would be

$$\mu_{\text{GA}} = \frac{\sqrt{\pi}}{8} \ell^{1.5} \log \ell + 8\sqrt{\pi \cdot \ell} \log \ell. \tag{7}$$

The memory-requirement estimates for cGA and simple GA are compared in Figure 1, which shows that cGA is highly memory-efficient than a simple GA, especially for large problems. For example, the memory requirements for solving a 2^{25} (32-million) bit OneMax problem with a population size of 2^{17} (131,072) cGA is a little over 128 megabytes in contrast to about 700 gigabytes for simple GA.

3.3 Computational Efficiency via Parallelization

We parallelized cGA using a master-slave process with message passing interface (MPI). As illustrated in figure 2, the probability vector is distributed across a number of processors and each processor—including the master—is responsible for sampling and updating a part of the probability vector. That is, with n_p processors, each processor contains ℓ/n_p probability vectors, with the master containing probabilities of variables 1 through ℓ/n_p, the first slave processor containing probabilities of variables $\ell/n_p + 1$ through $2\ell/n_p$, and the $(n_p - 1)^{\text{th}}$ slave containing probabilities of variables $(n_p - 1) \cdot \ell/n_p + 1$ through ℓ. Along with computational efficiencies, distributing the probability vector across multiple processors reduces the memory requirements per processor further to $\mu_{\text{cGA}} = \frac{17}{4} \cdot \frac{\ell}{n_p}$.

Each processor samples a part of the candidate solution by sampling the probabilities of variables and send the sampled solutions to the master processor. Therefore, at each cGA iteration there is one synchronization step where the partial candidate solutions are collected from all slave processors and combined in the master. The sampled solutions are then evaluated and the results of the evaluations are broadcasted to all the processors to be used for updating the probabilities. The probability updates are performed in parallel across different processors, with each processor updating probabilities of those variables that it contains. Distributing sampling and updating of probability vectors as described above, we minimize the communication between processors.

With the parallelization, we obtain linear speedup with the number of processors. Moreover, since the sampling and updating steps of cGA scales linearly with the problem size, and the total number of cGA iterations also scales linearly with the problem size, the computational time of cGA scales quadratically with the problem size. That is, when the problem size is doubled, we need to four-fold increase in the number of processors to solve the problem in the same time.

While the parallelization significantly enhances the efficiency of cGA, the computational efficiency can be further enhanced with the use of a number of efficiency enhancements including vectorization. Specifically, we concentrate on the following computationally costly steps of cGA: (1) the pseudo random number generation, (2) sampling two individuals from the probabilities, and (3) updating of the probabilities. Among these, as verified with profiling tools, most of the time is spent on the random number generation. For example, for solving 2^{20} (million) binary variable problem with a population size of 2^{15}, requires about 2^{47} calls to the pseudo random number generator. Therefore, we first optimized the pseudo-random number generator and developed a vectorized version of mersenne twister [15] using vector (single input multiple data, or SIMD) instruction sets, specifically Altivec for IBM PowerPC, G4, and G5 processors, and Intel's SSE2 for Intel and AMD processors.[1] Using vectorization, instead of one random number stream per processor, we generate four independent streams of ran-

[1]More information can be found at `http://developer.apple.com/hardware/ve/tutorial.html`, and `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24592.pdf`

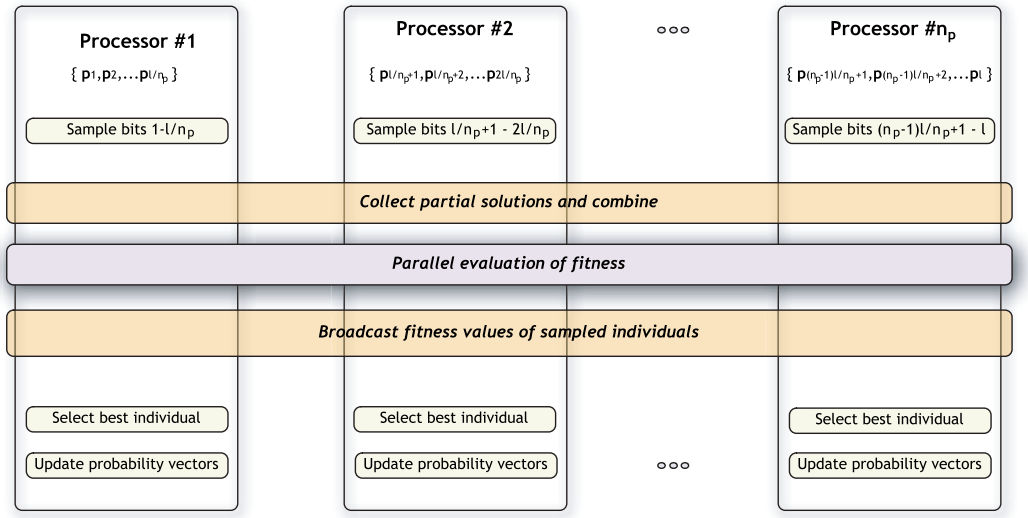


Figure 2: Schematic illustration of the parallelized SIMD-based compact genetic algorithm.

dom numbers per processor. Similarly, we also vectorize the sampling and updating of the probabilities and thus sample four bits at a time and update four probabilities at a time. Vectorizing the random number generation, probability sampling and updating provides a speedup of 4 over the non-vectorized implementation as shown in Figure 3.

To maximize the efficiency of cGA, we also optimized cGA operations in each processor using mostly bitwise operators, table lookup (for example, by pre-counting 16-bit unitation values), avoiding divisions, avoiding function calls, and mostly integer operations. For example, we precompute the unitation values (sums of bits) by pre-computing 16-bit unitation values. We also precompute the mapping the masks required to retrieve a specific bit from a word (4 bytes). In order to avoid divisions and to ensure integer-only operations, we restrict the population size to be a power of 2. Ensuring this restriction enables us to represent the probability vector as an integer and also allows us to use integer operations for sampling the probabilities. The resultant fully-parallelized, vectorized cGA is highly efficient—both computationally and memory-requirement wise—and therefore enable routine solutions to problems with millions to billions of variables.

4. TEST PROBLEM

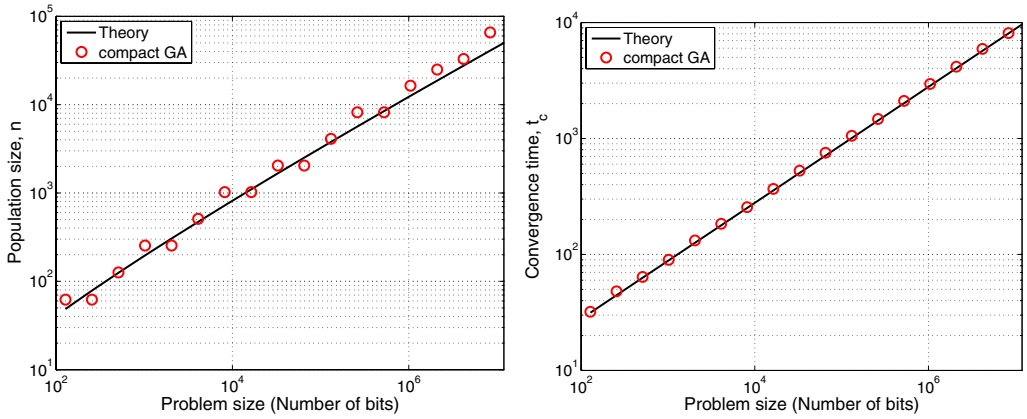
Our approach in testing cGA and other search methods is to consider problems from a *design envelope* perspective and to follow a Cartesian decomposition of different facets of problem difficulty [7]. Since cGA treats the variables to be independent of each other, we consider a class of additively-separable problems where the variables are independent of each other. The particular facet of problem difficulty we consider is the presence of exogenous noise in fitness evaluation. Specifically we consider the blind, noisy OneMax problem, where the objective is to maximize an outwardly unknown function of a binary string of fixed length ℓ—inwardly evaluated as the number of ones in a binary string—in the presence of additive Gaussian noise of specified variance, σ_N^2.

$$f(x) = \sum_{i=1}^{\ell} x_i + \mathcal{N}\left(0, \sigma_N^2\right). \tag{8}$$

As the function is unknown to the solver, successive samples are required for subsequent improvement, but even so, without noise, the OneMax is a problem that can be solved easily by a variety of GAs and simple hillclimbers; however, with noise set as a significant fraction of the deterministic variance of the problem, the blind, noisy OneMax poses a significant challenge to GAs and hillclimbers alike, because the noise makes it difficult to decide correctly between different samples. More difficult problems from the standpoint of exogenous noise will have higher variance values, and in this study we have considered exogenous noise that varies from 10^{-5} to 16 times the fitness variance.

Note that limiting ourselves to a class of problems where the variables are independent of each other does not mean that they are particularly easy. Indeed, in the presence of exogenous noise which equals 10% of the deterministic fitness variance, we make mistakes on roughly 10% of the variables. For example, an exogenous-noise variance set to 10% of the deterministic fitness variance for a 2^{30} (1.1 billion) variable problem, leads to mistakes on 27 million variables at each sampling. Moreover, elsewhere it has been demonstrated that certain class of variable interactions (also known as epistasis) manifest themselves as exogenous noise [7, 23].

In order to efficiently evaluate sampled solutions we parallelize the function evaluations across processors. That is each processor evaluates a part of the fitness of an individual and send the partial evaluation to the master. The master collects the partial evaluations for all the processors and combines them into a complete fitness values of the sampled solutions. These fitness values are then broadcast to all the processors. In each processor, we further enhance the efficiency of the fitness computation by precomputing 16-bit unitation values and using table-lookup for adding summation of 16 bits at a time.



(a) Population size (b) Convergence time

Figure 4: Population size and convergence time required by cGA as a function of the problem size. The results demonstrate the validity of the facetwise models for population sizing and convergence time (Equations 2 and 3). In both cases, results follow theoretical predictions where population size scales as $\Theta(\ell^{0.5}\log\ell)$ and the convergence time scales as $\Theta(\ell^{0.5})$. The number of experimental trials of each run was reduced with increased problem size with up to $\ell = 2^{18}$ receiving 50 trials, up to $\ell = 2^{23}$ receiving at least 10 trials.

5. RESULTS AND DISCUSSION

Before we present the results, we provide details of two local-search methods used to compare the performance of cGA. We then present the scalability of population size and run duration required by cGA as a function of problem size. We also compare the scalability of cGA with those of the local-search methods on both deterministic and noisy One-Max problems.

5.1 Sequential and Random Hillclimbers

To compare the scalability of cGA, we consider two local search methods: (1) Sequential hillclimber (sHC), and (2) random hillclimber (rHC). In sHC, we start with a randomly generated solution and then the variables of the search problem are flipped sequentially and the mutation that yields the best improvement in fitness is retained [22]. In the presence of exogenous noise, after each mutation we use the average of multiple samples of the fitness in deciding if the mutation should be retained or not. The number of samples of fitness required is given by [8, 22]:

$$n_s = \min\{1, 2c\sigma_N^2\}, \tag{9}$$

where n_s is the number of independent fitness samples, and c is the square of the ordinate of a one-sided standard Gaussian deviate at a specified error probability α. As with cGA, for sHC we use $\alpha = 1/\ell$.

Since the initial solution is evaluated n_s times and after that for each of the ℓ variables, an individual created after flipping the binary variable is evaluated n_s times, the total number of function evaluations required for the sHC is given by

$$n_{\text{fe,sHC}} = n_s\,(\ell+1) \approx \frac{c}{2}\left(1+\frac{\sigma_N^2}{\sigma_f^2}\right)\ell\,(\ell+1)\,. \tag{10}$$

Similar to sHC, in random hillclimber (rHC), we start with a randomly generated solution. However, unlike sHC, in rHC a variable is selected according to uniform random distribution and mutated. If the mutated solution has a higher fitness than the original one, the mutation is retained. If not, the mutation is discarded and the process continues. For the OneMax problem, the first-hitting time of the global optimum via rHC is given by $\ell\log(\ell)$ [17]. However, the presence of exogenous noise introduces error in the decision making during every mutation step. The probability of correctly deciding, p_{dm}, during a mutation step is given by the decision-making model [8]:

$$p_{dm} = \phi\left(\frac{1}{\sigma_N}\right). \tag{11}$$

The probability of making correct decisions on majority of the ℓ bits, which is given by

$$P_s = \sum_{i=\ell(1-\epsilon)}^{\ell} \binom{\ell}{i} p_{dm}^{i}\,(1-p_{dm})^{\ell-i} \tag{12}$$

where $1-\epsilon$ represents the minimum proportion of variables for which the decision-making has to be accurate. Using the above equation, we can obtain the first-hitting time to obtain the global optimum with rHC as:

$$n_{\text{fe,rHC}} = \ell\log\ell\cdot\frac{1}{P_s}. \tag{13}$$

5.2 Empirical Results

We ran the cGA experiments on a 128- and 256- processor partition of 1280-processor Turing cluster which consists of 1,280 Apple G5 Xserve. Since we restricted the population size to be a power of 2, in order to determine the minimum population size, we doubled the population size till the cGA converged to a population with least $\ell-1$ out of ℓ variables set to their optimal values. For problems with less than or equal to 2^{18} (over 256,000) bits, the results are averaged over 50 independent runs. For problems with greater than 2^{18} bits, but less than or equal 2^{25} (over 32 million) bits, the results are averaged over 10 independent runs. For problems with more than a million variables, where its very expensive to run multiple runs with different population sizes, we fixed

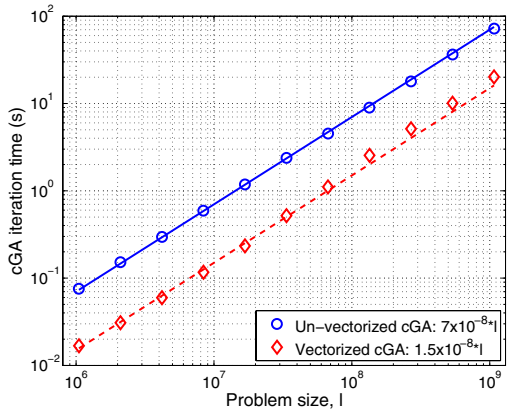


Figure 3: Comparison of execution times for both non-vectorized and vectorized parallel cGA implementations. We used 32 processors and ran cGA for one hour of wall-clock time. The number of cGA iterations were roughly halved as the problem size increased. That is, with 2^{20} variables, we used 6 million cGA iterations for the vectorized implementation, and with 2^{21} variables we used 3 million cGA iterations. The results show that as the problem size is increased, we need to increase the computational time four times to perform an equal number of cGA iterations, validating our earlier assessment on the scalability of parallelized cGA. The results are averaged over 10 independent runs.

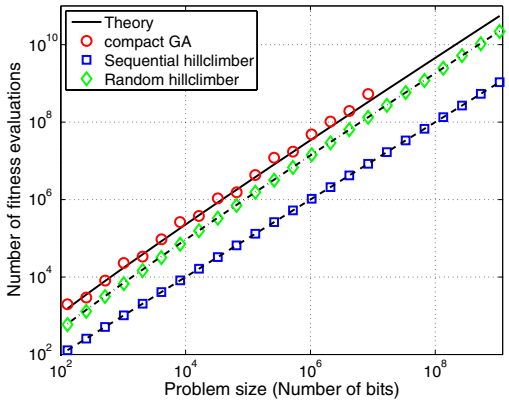


Figure 5: Scalability of compact genetic algorithm (cGA), sequential hillclimber (sHC), and random hillclimber (rHC) on the OneMax problem with no exogenous noise. The results follow theoretical predictions and the cGA and rHC scale as $\Theta(\ell \log \ell)$ and the sHC scales as $\Theta(\ell)$. As expected, both sHC and rHC are more efficient than cGA in solving the deterministic OneMax problem. The number of experimental trials of each run of the cGA and rHC were reduced with increased problem size with up to $\ell = 2^{18}$ receiving 50 trials, up to $\ell = 2^{23}$ receiving at least 10 trials.

the population size as specified by the model (Equation 2). We then determine the total number of function evaluations required by cGA with that population size to converge to the optimal solution.

We start by demonstrating the population-sizing and convergence-time scaling of cGA for the OneMax problem in Figure 4. As shown in the figure, the results follow theoretical predictions (see Equations 2 and 3) and that the population size indeed scales as $\Theta(\ell^{0.5} \log \ell)$ and the convergence time scales as $\Theta(\ell^{0.5})$. We then compare the total number of function evaluations required by the compact GA, sequential hillclimber, and the random hillclimber to successfully solve the OneMax problems in Figure 5. Again the results follow theoretical predictions and the cGA and rHC scale as $\Theta(\ell \log \ell)$ and sHC scales as $\Theta(\ell)$. More importantly, the memory and computational efficiencies enables us to use cGA to successfully solve problems with over 8 million variables to optimality.

Now we consider the scalability of cGA, sHC, and rHC on the noisy OneMax problem with the exogenous noise variance of 10^{-5}th of the deterministic variance. That is, $\sigma_N^2 = 10^{-5}\sigma_f^2 = 0.25 \times 10^{-5} \cdot \ell$. The results comparing the scalability are shown in Figure 6. Once again, the results follow theoretical predictions (see Equations 4, 10, and 13). Moreover the results clearly show that even with the addition of a very small exogenous noise, the rHC is not able to solve problems with greater than 2^{15} (32,766) variables. On the other hand, sHC while being more efficient than cGA for smaller problems, for problems larger than 2^{19} (half a million) variables, it becomes increasingly less efficient. Moreover, the memory and computational efficiencies enables us to use cGA to successfully solve the blind, noisy OneMax problem with over 8 million variables to optimality.

Finally we demonstrate the ability of the cGA to solve problems with several million to billions of variables. Specifically we consider the blind, noisy OneMax problem with the variance of the additive exogenous noise set to a tenth of the deterministic variance. That is, $\sigma_N^2 = 0.1\sigma_f^2 = 0.025\ell$. Note that this noise level is significantly high to disrupt local-search and other non-population oriented methods. For example, the exogenous noise variance for a 2^{25} (over 32 million) variable problem is about 8.4×10^5. As in earlier cases, the population sizes of cGA were first verified for problems with 2^{20} and less variables and for larger problems set according to theoretical predictions of the gambler's ruin model (Equation 2). For sHC, the number of fitness samples used to estimate solution quality is empirically determined using bisection method and follows theoretical predictions (Equation 9) The number of experimental trials of each run of the cGA were reduced with increased problem size with up to $\ell = 2^{18}$ receiving 50 trials, up to $\ell = 2^{23}$ receiving at least 2 trials, and larger problem instances receiving one trial.

We show the scalability of cGA on the blind, noisy OneMax in Figure 7. The first set of runs (full convergence) compares cGA and a sequential hillclimber (sHC) where the number of function evaluations required to solve the noisy OneMax problem to optimality is shown for different problem sizes. For cGA and sHC, the theoretical and empirical results both track nicely, but with the resources available, we were able to run the cGA to 2^{25} (over 32-million) variables, but only 2^{14} (16,384) variables for sHC. With available resources rHC was not able to solve even the smallest problem instances. Again as predicted by facetwise models, the cGA scales as $\Theta(\ell \log \ell)$. To the best of our knowledge, this is

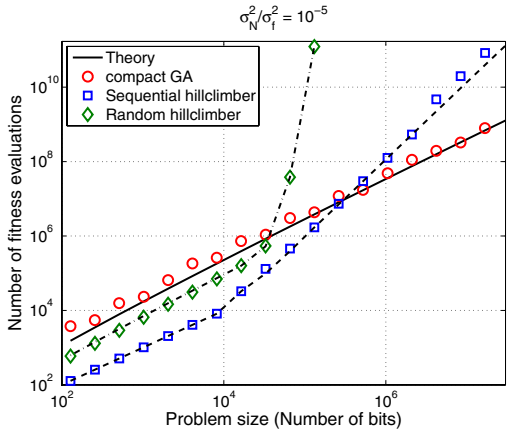


Figure 6: Scalability of cGA, sHC, and rHC on the noisy OneMax problem with exogenous noise variance equal to 10^{-5}th of the deterministic variance. That is $\sigma_N^2 = 10^{-5}\sigma_f^2 = 0.25 \times 10^{-5} \cdot \ell$. Even with a very small noise, the rHC is not able to solve problems with greater than 2^{15} (32,766) variables. On the other hand, sHC while being more efficient than cGA for smaller problems, for problems larger than 2^{19} (half a million) variables, it becomes increasingly less efficient. The number of experimental trials of each run of the cGA and rHC were reduced with increased problem size with up to $\ell = 2^{18}$ receiving 50 trials, up to $\ell = 2^{23}$ receiving at least 10 trials.

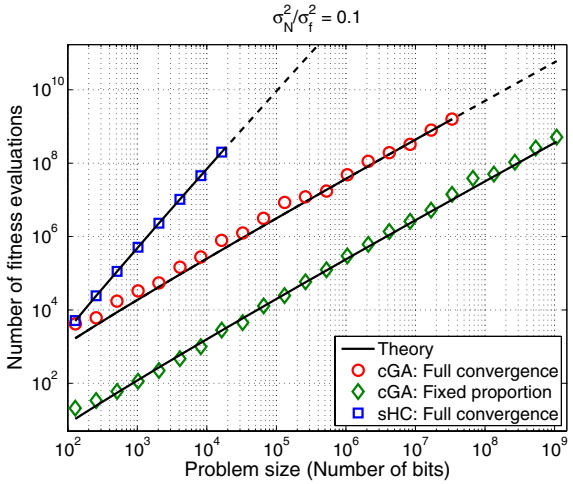


Figure 7: The figure presents computational results from two sets of experiments on a blind, noisy OneMax problem. In the results labeled full convergence, average total number of fitness evaluations is plotted versus the problem size ℓ for runs with sHC and cGA. In both cases, results follow theoretical predictions (see Equations 4 and 10) with the sHC and cGA requiring $\Theta(\ell^2)$ and $\Theta(\ell \log \ell)$ number of function evaluations, respectively. The rHC on this problem failed to converge for even the smallest problem instance. The number of experimental trials of each run was reduced with increased problem size with up to $\ell = 2^{18}$ receiving 50 trials, up to $\ell = 2^{23}$ receiving at least 2 trials, and larger problems receiving one trial. In a set of fixed proportion runs the cGA is run until all probability values in the vector are greater than 0.501. Like the full convergence CGA result, the fixed proportion result scales as $\Theta(\ell \log \ell)$ as predicted theoretically, and this level of convergence strongly predicts accurate full convergence. As such, these results represent the first time any genetic algorithm has ever been run successfully at a problem size over a billion bits.

the first time any GA has been used to successfully solve a class of boundedly-difficult problems with over 32-million variables.

To move on to a billion bits, we performed a set of fixed-proportion tests on cGA up to 2^{30} (1.1-billion) variables. Here the criterion of convergence was relaxed, requiring that each variable reach a fixed-proportion correct ($p_i = 0.501$). We note that for the fixed-proportion runs, the population size was sized according to the gambler's ruin model. For example, for the 2^{30}-bit problem, the population size used was 2^{20} (over 1 million). We stress that the reason for relaxing the convergence criteria was the limited computational resources available, and not the algorithmic limitation. Nevertheless, the fixed-proportion runs required $\Theta(\ell \log \ell)$ number of function evaluations as did the full-convergence results, thereby demonstrating scalability and the first successful billion-bit genetic algorithm ever run.

6. SUMMARY AND CONCLUSIONS

This paper presents an highly-efficient, fully parallelized compact genetic algorithm for solving large-scale problems with millions to billions of variables. A number of memory and computational efficiencies, including vectorization using SIMD instructions are used to enable rapid solutions to very large-scale problems. The scalability of the parallel cGA is tested on a noisy OneMax problem with exogenous noise variance set up to a tenth of the deterministic variance. We show that using a modest 128- and 256- processors, cGA can solve problems with over 33 million variables to optimality and with a relaxed convergence, cGA can successfully solve problems with 1.1 billion variables.

This effort is the first step towards developing a scalable genetic algorithm that can routinely solve boundedly-difficult problems. Existing theory confirmed by bounding experiment already tells us how to extend these results in which noise is the primary difficulty to those where modularity, overlap, or hierarchy come into play [26] and efforts are currently underway towards this goal. In this way, these results are a building block to practical and effective billion-variable optimization in the increasingly large, complex problems of science. As science pushes to the frontiers of the small, the large, the living, and the societal, the need to tackle extraordinarily large search and optimization problems will become increasingly acute. These results foretell a time when such problems can be tackled routinely, confidently, and well.

Acknowledgments

The empirical runs for this paper were conducted on the Turing cluster at Computational Science and Engineering at the University of Illinois.

This work was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant FA9550-06-1-0096, the National Science Foundation under ITR grant DMR-03-25939. The U.S. Government is authorized to reproduce and distribute reprints for government purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFOSR, NSF, or the U.S. Government.

7. REFERENCES

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

[2] S. J. Benson, L. C. McInnes, and J. J. More. A case study in the performance and scalability of optimization algorithms. *ACM Transactions on Mathematical Software*, 27(3):361–376, 2001.

[3] B. Carter and K. Park. Scalability problems of genetic search. *Proceedings of the 1994 IEEE International Conference on Systems, Man, and Cybernetics*, 2:1591–1596, 2004.

[4] K. Deb and K. Pal. Efficiently solving: A large-scale integer linear program using a customized genetic algorithm. *Proceedings of the 2004 Genetic and Evolutionary Computation Conference*, pages 1054–1065, 2004.

[5] K. Deb, A. R. Reddy, and G. Singh. Optimal scheduling of casting sequence using genetic algorithms. *Materials and Manufacturing Processes*, 18(3):409–432, 2003.

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

[7] D. E. Goldberg. *Design of innovation: Lessons from and for competent genetic algorithms*. Kluwer Academic Publishers, Boston, MA, 2002.

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

[9] D. E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3(5):493–530, 1989. (Also IlliGAL Report No. 89003).

[10] J. Gondzio and A. Grothey. Direct solution of linear systems of size 10^9 arising in optimization with interior point methods. *Proceedings of the Parallel Processing and Applied Mathematics (PPAM 2005)*, pages 513–525, 2006.

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

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

[13] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975.

[14] L.-D. Lang and L. T. Biegler. Large-scale nonlinear programming with cape-open compliant interface. *Chemical Engineering Research and Design*, 83(A6):718–723, 2005.

[15] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998. `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`.

[16] B. L. Miller and D. E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9(3):193–212, 1995. (Also IlliGAL Report No. 95006).

[17] H. Mühlenbein. How genetic algorithms really work: Mutation and hillclimbing. *Parallel Problem Solving from Nature II*, pages 15–26, 1992.

[18] H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions I. Binary parameters. *Parallel Problem Solving from Nature*, 4:178–187, 1996.

[19] S. S. Nielsen and S. A. Zenios. Scalable parallel benders decomposition for stochastic linear programming. *Parallel Computing*, 23:1069–1088, 1997.

[20] S. Oh and S. Y. Shin. A parallel algorithm for large-scale linear programs with a special structure. *Proceedings of IEEE Scalable High Performance Computing Conference*, pages 749–755, 1994.

[21] K. Sastry. Evaluation-relaxation schemes for genetic and evolutionary algorithms. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 2001. (Also IlliGAL Report No. 2002004).

[22] K. Sastry and D. E. Goldberg. Let's get ready to rumble: Crossover versus mutation head to head. *Proceedings of the 2004 Genetic and Evolutionary Computation Conference*, 2:126–137, 2004. Also IlliGAL Report No. 2004005.

[23] K. Sastry, P. Winward, D. E. Goldberg, and C. F. Lima. Fluctuating crosstalk as a source of deterministic noise and its effects on ga scalability. *Applications of Evolutionary Computing EvoWorkshops2006: EvoBIO, EvoCOMNET, EvoHOT, EvoIASP, EvoInteraction, EvoMUSART, EvoSTOCK*, pages 740–751, 2006. (Also IlliGAL Report No. 2005025).

[24] Y. Semet and M. Schoenauer. An efficient memetic, permutation-based evolutionary algorithm for real-world train timetabling. *Proceedings of the 2005 Congress on Evolutionary Computation*, pages 661–667, 2005.

[25] D. Yang and S. A. Zenios. A scalable parallel interior point algorithm for stochastic linear programming and robust optimization. *Computational Optimization and Applications*, 7:143–158, 1997.

[26] T.-L. Yu. *A matrix approach for finding extrema: Problems with modularity, hierarchy, and overlap*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 2006.