Parallel Genetic Algorithm for Solving Non-Convex Optimization Problems

Eashan Singh
B.S. Computer Science – In Progress
University of Central Florida
Orlando, USA
eashan.singh1@knights.ucf.edu

Amelia Castilla
B.S. Computer Science – In Progress
University of Central Florida
Orlando, USA
accrazed@knights.ucf.edu

Samu Wallace
B.S. Computer Science – In Progress
University of Central Florida
Orlando, USA
samu.wallace@knights.ucf.edu

***Abstract*—Non-convex optimization problems refer to the optimization of an objective function which is non-convex – that is, a function which curves up and down, not fitting either of the clear definitions for convex or concave functions – whose global extrema we are trying to find. In this way, a non-convex function is expected to have several local minima/maxima, which makes it challenging for optimization algorithms to find the true global extremes. As such, there are many classes of algorithms for solving these problems, one of which being Genetic Algorithms (GA).**

Genetic Algorithms are inspired by and modeled after phenomena observed in biology, namely natural selection and genetics. They work by generating a population of 'candidate solutions' (set of variables to be input into the objective function) and then use evaluation, selection, crossover, and mutation operators in order to evolve the population over multiple generations into better and better candidate solutions, eventually settling on an optimized solution. The generation of these candidate solutions through the above steps can be computationally taxing, with limited scalability when using a single processor. This is where parallelization improves the performance of GAs, by splitting up the work among multiple processes.

In this paper, we build a Parallel Genetic Algorithm (PGA) using the *master-slave model* for solving non-convex optimization problems, specifically the Rastrigin function. We then benchmark the resulting performance of the PGA against its serial GA counterpart. Performance is measured in both runtime and solution accuracy, although solution accuracy is expected to remain statistically equivalent. As we observe performance differences, we also discuss possible causes for them. Through analysis of our PGA performance, we aim to answer our main research question – that is, whether our parallelized genetic algorithm produces improvements in runtime that is worth the extra implementation complexity compared to a simple GA.

Our benchmarks showed that the runtime of our PGA did indeed improve upon that of the serial GA, with an unaffected solution accuracy. We observed that the runtime decreased linearly as population size increased. Runtime also decreased as the number of threads increased, although the performance began to decrease after a certain amount of threads. However, runtime improvements were sizable but not enormous, and so future parallelization designs – such as the *island model* may be implemented and benchmarked in future works. This may be because the computation the fitness and selection of individuals is not overwhelmingly computationally taxing compared to the overhead when creating and joining threads.

***Index Terms*—parallel genetic algorithm, master-slave model, island model, coarse-grained, fine-grained, parent**

I. Introduction

A. Motivation

Non-convex functions have far-reaching applications in many fields, including mathematics, computer science (machine learning, image processing, signal processing, robotics motion, etc.), finance, material science, and more [1]. As such, it is imperative to find the optimizations to these complex functions in as little time as possible through the use of increasingly powerful algorithms. Genetic Algorithms have already shown great performance in this area. This paper explores the power in the parallelization of the already powerful GA.

B. Background

1) Genetic Algorithms: Genetic Algorithms are inspired by evolution and biology, which is reflected in their implementation. The fundamental idea of a GA is to spawn a population of randomly generated potential solutions, and then to evolve this population into iteratively better solutions through the use of techniques borrowed from biology – these being fitness, selection, crossover, and mutation. The implementation is as follows:

1) Randomly generate the initial population.
2) Evaluate the fitness of each individual. Fitness is defined as the accuracy of an individual's solution, determined by inputting their solution into the objective function. These solutions are represented by a chromosome, made up of individual bits (genes).
3) Select new parents. The parents (also called survivors) are selected based on their fitness – following the logic of natural selection. Fitter individuals are thus more likely to be selected as parents for the next step. This is often referred to as tournament selection.
4) Crossover and Mutate. These two techniques are used to create the next generation of potential solutions. Crossover involves taking two parent individuals and splitting their chromosomes at a point(s) and combining them as a new individual. This crossover point along the chromosome is usually selected at random, as is done in our implementation. The crossover technique shown in Figure 1 is applied to every pair of selected parents, until the population size for the next generation

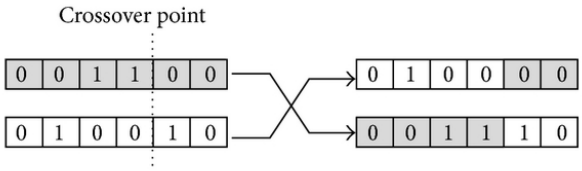


Fig. 1: One-point crossover. Left shows the parent chromosomes combining to create two offspring on the right.

is met. Mutations are applied after crossover, involving randomly selecting individuals for random gene modification. In the case of a binary chromosome, this could mean simply flipping a bit.

5) If termination criteria is met, stop the algorithm; otherwise, go back to the fitness evaluation step and loop.

These techniques are clearly inspired by the real life biology of genetics. Just as in evolution, the crossing over (combination) of successful parent genes serves to converge the population towards fitter individuals, and mutation serves to introduce new genetic information and prevent premature convergence. Fitness evaluation and selection of parents represent the mechanisms of natural selection. It is also true – in nature and in computing – that these mechanisms take time, which is what we aim to minimize through parallelization.

2) Rastrigin Function: Since the goal of our paper is heavily tied to benchmarking, we have chosen the Rastrigin function as our objective function to evaluate. The Rastrigin function is more popular in its ability to benchmark optimization algorithms than its practical applications; however, an optimization algorithm which performs well on the Rastrigin function can be expected to perform well with most other non-convex functions. Our PGA is also able to optimize any other non-convex (or convex) function.

$$f(\mathbf{x}) = An + \sum_{i=1}^{n} [x_i^2 - A\cos(2\pi x_i)] \tag{1}$$

Equation (1) shows the Rastrigin function, where A is a positive constant that scales the function's curvature and n is the number of dimensions. However, the equation alone does not fully convey the difficulty in optimizing such a function.

The graph in Fig. 2 effectively illustrates the clustering of local extrema and degree of ruggedness which makes the Rastrigin function so difficult to optimize. Unsuitable optimization algorithms may get trapped in one of numerous local minima, or may have trouble navigating the tight spacing of extrema and overshoot peaks. All of these challenges are what makes the Rastrigin function a fitting benchmark for our PGA.

C. Objectives and Research Questions

This paper aims to resolve several questions and meet several objectives. In regards to the former, we hope to answer

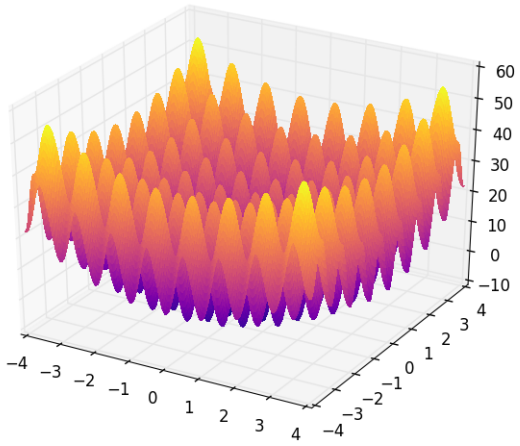


Fig. 2: Graph of the Rastrigin function.

whether any decreases in runtime are worth the extra programming complexity. We also hope to address the following questions:

- What is the optimal number of threads, depending on user CPU specifications?
- Does the performance increase from parallelization warrant the multi-core power draw which may slow down other processes on the computer?
- How does thread creation overhead affect our runtimes?
- Is our implementation easily compatible with other non-convex functions?
- What effect does changing the constant A in the Rastrigin function have on our PGA performance?

Furthermore, we have an additional objective of providing a PGA that is easier to implement than popular alternatives, which is why we have chosen the popular language of Java. Java is especially popular in the realm of algorithm analysis, and is also prone to notoriously long runtimes as a result of its interpreted nature. So, we implement a PGA in Java with the simple objective of reducing these runtimes.

II. Related Work

One of the earliest studies on PGAs was conducted by Goldberg and Lingle, where they used a parallel implementation of a GA to solve a traveling salesman problem. They found that the parallel GA outperformed the serial implementation in terms of both solution quality and runtime [2].

Other papers have explored the many different types of PGAs, classifying them into three main categories: global single-population master-slave PGAs, single population fine-grained PGAs, and multiple-population coarse-grained PGAs [3] [4]. This paper will implement a global single-population master-slave PGA.

III. Problem Statement

The specific problem which this paper aims to address is the high runtime for genetic algorithms of sufficiently high

population sizes. This paper introduces a parallelized genetic algorithm in order to reduce this runtime, built to solve non-convex optimization problems.

IV. Methodology

A. Choosing an Algorithm

In order for our Genetic Algorithm to be parallelized, the work must be divided into parts which are solved concurrently through the use of multiple processors. Of course, there are several ways to accomplish this when it comes to GAs. Some algorithm designs consist of an undivided population, wherein steps such as selection and crossover are performed on this one population, and parallelization is only utilized in the calculation of fitness. This is often classified as a master-slave model PGA. Other algorithm designs consist of subdivided populations which are*fine-grained* (i.e., numerous populations) or *coarse-grained* (i.e., sparse populations). These PGAs which are implemented with populations which evolve separately are often referred to as *island model* PGAs.

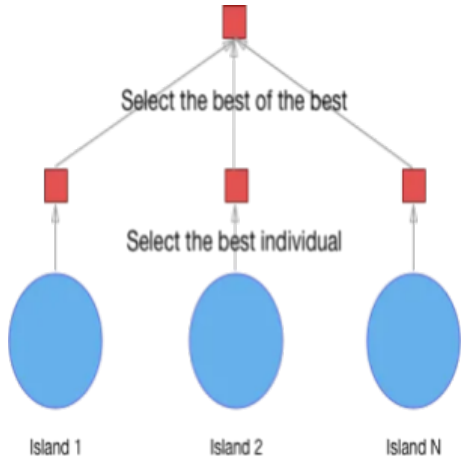


Fig. 3: Basic representation of the island model PGA.

In an island model design, every island represents an individual genetic algorithm, each with their own populations and their own divergent mutations and crossover history. The GAs in this design are referred to as islands because of their isolation; they do not depend on each other. Figure 3 helps illustrate some of the benefits of this design, as it is clear to see that the pool of selected *parent* individuals increases N-fold. However, with this design, it is also important that each island produces varied results, so that parallelization is not wasted on computing extremely similar Genetic Algorithms in separate processes. This can be done by varying the following between the individual subpopulations:

- How mutations are performed.
- How crossovers are performed.
- How many individuals are selected for crossover, as well as for parenting (generation size).
- The total number of generations to compute.

The one aspect of the algorithm which stays constant is the fitness function, as all algorithms need to know what makes an individual successful.

Parallelization in combination with these variations helps to address one of the main issues with genetic algorithms, in that they tend towards preliminary convergence to a subset of individuals that dominate others. Having GAs with differing parameters run in parallel and selecting from those should greatly mediate this issue.

Nevertheless, seeing as the aim of this paper is to evaluate the performance pertaining to runtime of parallelization in particular, we decided to choose a PGA algorithm which derives its benefit chiefly from parallelization of computation – for example, in the computation of the objective function for evaluating fitness – rather than from the parallelization of multiple populations. We felt that the island model techniques such as migration and hyperparameter tweaking between populations would add complexity to the PGA and only serve to improve solution accuracy – not runtime – which we had already observed as wholly satisfactory in our serial GA testing, The island model and other PGA algorithms which subdivide the population inevitably change the behavior of the GA, and this is beyond the scope of our paper.

Therefore, we implement our PGA using a single population master-slave model as our parallelization algorithm.

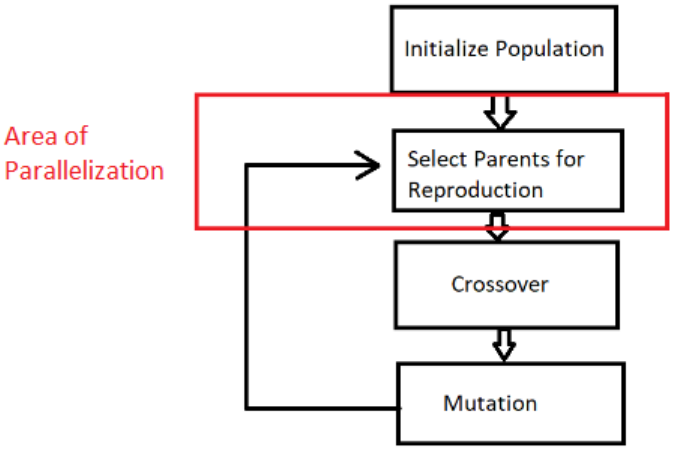


Fig. 4: Basic flowchart of our PGA implementation.

B. Our Algorithm

For this paper, we decided to employ a global single-population master-slave PGA.

In this nomenclature, the master is the main thread, which loops for several runs through several generations, holds all of the data regarding individuals, and performs the creation of new individuals through crossover and mutation. The master (main) thread distributes the fitness evaluation across several processors, which are the slaves that run the objective function and find the fitness of each individual in their subdivided search area. As a result, each generation of the PGA should

execute faster. We parallelized the evaluation of fitness (seen in Figure 5) as this is the usual bottleneck in GAs, requiring that every individual have its chromosome input into an often complicated fitness function.

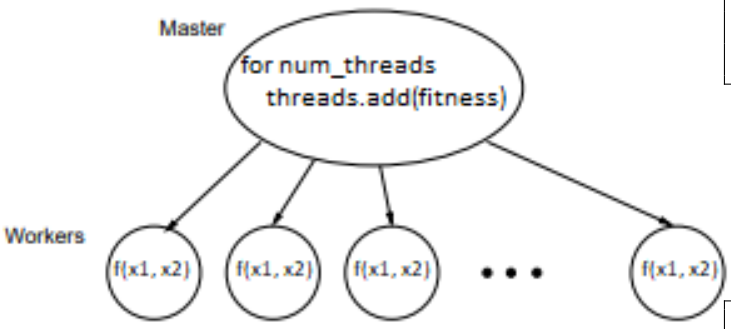


Fig. 5: Flowchart representation of our master-slave PGA. The master evenly distributes the total population divided by the number of threads to each worker/slave for fitness evaluation.

Beyond choosing a global population parallelization model, we also made decisions regarding the genetic algorithm. First, we decided not to implement complicated algorithms for choosing the different hyperparameters for each parallel genetic algorithm, as we wanted to stick to the idea of a simple implementation. We also on our termination criteria to be a maximum number of generations, because this allowed for a fairer comparison between the GA and PGA. Finally, we decided to implement only single crossover for this paper, as our main research objective lies in runtime and not solution accuracy. More complex crossover strategies would only benefit the latter.

Since one of our objectives in this paper is to provide easy implementation, important Java code snippets are provided for reference, showcasing crucial elements of the PGA.

```
// Generate binary candidate solution
for (int i = 0; i < 20; i++) {
    gene = Math.random() > 0.5 ? '0':'1';
    this.chromo = chromo + gene;
}

this.fitness = -1;
```

Listing 1: Chromosome initialization.

Listing 1 is the constructor function for the chromosome class, This is part of the first step in any GA implementation, wherein each individual of a population is initialized and their chromosome randomly generated bit by bit. The for loop has 20 iterations, which represent the 10 + 10 bits that hold the data for the x_1 and x_2 candidate solutions. This number may seem arbitrary, but it was chosen as it is small enough to speed up calculations while retaining enough precision. These chromosomes are stored as strings. The next step is to evaluate the fitness.

```
// Convert 0-9 bits of chromo to float
for (int i = 9; i >= 0; i--) {
    gene = chromo.chromo.charAt(i);
    if (gene == '1') {
        x1 = x1 + (int)Math.pow(2.0, 9-i);
    }
}
```

Listing 2: Chromosome bit conversion.

Before plugging into our objective function though, we need to transform our data from the binary chromosome to float representation, as we can see in Listing 2, Then, the x1 and x2 floats are transformed further and plugged into the objective function.

```
x1 = x1/100.0;
x2 = x2/100.0;

x1 = x1 - 5.12;
x2 = x2 - 5.12;

double rast = 0;

rast += (x1*x1)-(10*Math.cos(2*PI*x1))+10;
rast += (x2*x2)-(10*Math.cos(2*PI*x2))+10;

chromo.fitness = rast;
```

Listing 3: Fitness/objective function.

```
int blkSize = (int)(POP_SIZE/NUM_THREADS);
int total = 0;
List<threadFitness> threads =
    new ArrayList<threadFitness>();
for (int q = 0; q < NUM_THREADS; q++) {
    if(q!=NUM_THREADS-1) {
        threadFitness myThing =
            new threadFitness(problem,
                total,total+blockSize-1,
                POP_SIZE, member);
        myThing.start();
        threads.add(myThing);
    }
    else {
        threadFitness myThing =
            new threadFitness(problem,
                total, POP_SIZE-1,
                POP_SIZE, member);
        myThing.start();
        threads.add(myThing);
    }
    total += blockSize;
}
```

Listing 4: Multithreaded fitness evaluation.

In our implementation, the Rastrigin function is hardcoded in for viewing simplicity, but this can easily be substituted for a generic *problem()* function. Listing 3 is the fitness function which each slave thread will be applying to each of its individuals.

Listing 4 shows the actual parallelization of the fitness evaluation. Each thread is assigned a block of individuals evenly distributed among them by the master process. This parallelization is able to be done without the use of atomic data types or mutual exclusion primitives because each thread is operating on its own subset of the population. Because of this, race-conditions between threads writing to the same area of memory are not possible.

After fitness evaluation comes parent selection, which is called by the master process as it iterates through the entire population two at at time. As it iterates, it calls a *tournament selector* to decide the indices of two parents representing two selected child-bearers for every two individuals.

```
int tourneySize = 2;
double minFitness = Double.MAX_VALUE;
int bestMember = -1;

for (int i = 0; i < tourneySize; i++) {
    randnum = Search.r.nextDouble();
    int x = (int)(randnum * 1000);
    if (Search.member[x].fitness
                    < minFitness) {
        minFitness =
            Search.member[x].fitness;
        bestMember = x;
    }
}

return bestMember;
```

Listing 5: Selection tournament (formatting edited to fit column-width).

The tournament selector in Listing 5 compares a random individual from the entire population to lowest fitness seen at that point in the tournament, and returns index of the fitter individual of the two that were randomly selected. The more fit individual is the one with the lower fitness, as Rastrigin is being treated as a minimization problem. In the master process, this is called until a number of parents equal to the population size are selected. However, parents can reproduce more than once, so the actual number of unique parents is still a subset of the total population of each previous generation. In this way, poor fitness individuals may never reproduce, thus eliminating less desirable genes, while an entire population size of more desirable children is still produced with each generation.

Next, the chromosomes of these selected parents get crossed over and combined, creating two children. You can see in Listing 6 that this crossing over results in genetic info from both parents, expanding the solution space and inheriting high fitness genes.

```
//  Select crossover point
xoverPoint1 = 1 +
    (int)(Search.r.nextDouble() * (19));

//  Create child chromosome
child1.chromo =
parent1.chromo.substring(0,xoverPoint1)
+ parent2.chromo.substring(xoverPoint1);

child2.chromo =
parent2.chromo.substring(0,xoverPoint1)
+ parent1.chromo.substring(xoverPoint1);
```

Listing 6: Single point crossover function.

Finally, the last step before going back to fitness evaluation and selection is mutation.

```
for (int j=0; j<20; j++) {
    x = this.chromo.charAt(j);
    randnum = Search.r.nextDouble();
    if (randnum < 0.05){
        if (x == '1') x = '0';
        else x = '1';
    }
    mutatedChromosome =
        mutatedChromosome + x;
}

this.chromo = mutatedChromosome;
```

Listing 7: Mutation function.

Listing 7 shows how the mutation function works. A for loop with 20 iterations (the size of the chromosome) reads every bit from the individual's chromosome. There is a 5% chance that any one bit is flipped. This randomness expands the solution space and introduces diversity, thus deterring premature convergence.

If the termination criteria of a certain number of generations has not been met, enter the loop again and evaluate the fitness of the next generation.

V. Results

The results show a sizeable decrease in runtime as the number of threads increase.

The solution accuracy (seen in Figure 9) also remained statistically the same between serial and various thread counts, as the behavior of the GA was not altered by our parallelization. This is good news, as it is indicative of our implementation avoiding race-conditions.

VI. Discussion

After analyzing the results of our PGA benchmarks, we can address the questions brought forward in Subsection 1.C.

First and foremost – yes, it was worth the extra programming complexity. This is due to the fact that the vast

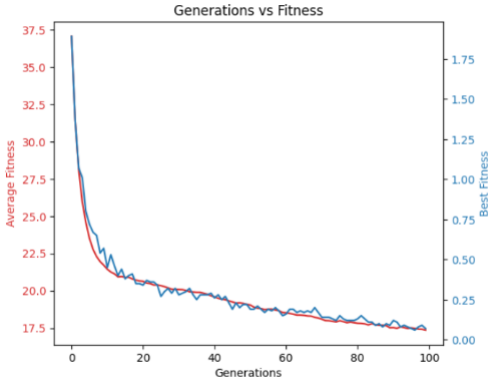


Fig. 6: Convergence plots.

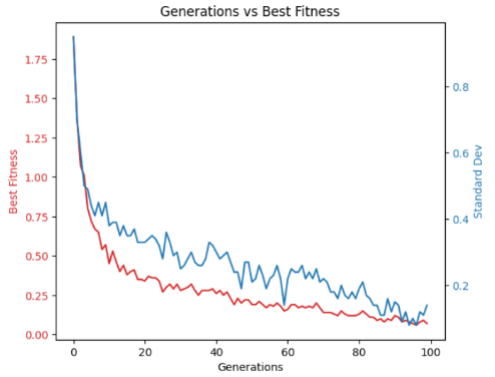
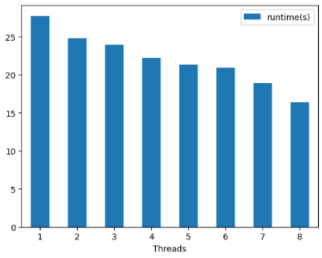


Fig. 7: Best fitness plot.

majority of complexity came solely from the initial serial GA implementation. From that point, parallelizing the fitness evaluation was more or less trivial, as no shared memory, mutexes, or inter-process communication was required. For just a little added effort, we observed a 41% decrease in runtime at best, as the number of threads increased.

We found that the optimal amount of threads was usually 8. Lower than this seemed to result in potential runtime decreases are not being met, and more than this seemed to introduce too much overhead in thread creation or in CPU traffic.

The performance increase from parallelization does warrant



Thread #	Runtime(s)
1	27.728
2	24.827
3	23.892
4	22.186
5	21.351
6	20.959
7	18.869
8	16.333*

41% speed up !

Fig. 8: Runtime vs. thread count.

84	17.55	0.65	0.09	0.12
85	17.53	0.64	0.07	0.08
86	17.53	0.72	0.1	0.16
87	17.5	0.68	0.09	0.12
88	17.42	0.7	0.08	0.14
89	17.39	0.64	0.08	0.11
90	17.38	0.65	0.12	0.21
91	17.36	0.74	0.1	0.16
92	17.42	0.74	0.08	0.11
93	17.36	0.7	0.07	0.08
94	17.37	0.69	0.07	0.11
95	17.44	0.7	0.06	0.09
96	17.41	0.65	0.07	0.07
97	17.37	0.62	0.07	0.07
98	17.29	0.69	0.09	0.12
99	17.31	0.75	0.1	0.16

Fig. 9: Snippet from Rastrigin_summary.txt. This output file shows the statistics of each generation, from left to right: generation number, average fit, StdAvg, best fit, and StdBest.

the multi-core power draw, as those who are running GAs with performance in mind are typically not worried about multi-tasking. Rather, they are worried with getting their optimization as soon and as accurate as possible.

Our implementation could easily be compatible with other non-convex functions, though this was not particularly pressing to implement for this paper. As such, that functionality is left out at the moment, but can easily be added within 30 minutes, provided it takes in two inputs that can fit in 10 bits each.

We observed that modifying the constant A (which controls the ruggedness of the Rastrigin function) had little to no effect on our runtime, although it did have a marginal effect on our solution accuracy.

VII. Conclusion

In conclusion, this paper has presented a parallel genetic algorithm implementation that parallelizes the fitness evaluation process of individuals. Our results showed that increasing the number of threads up to 8 resulted in a linear decrease in the runtime for optimizing the Rastrigin function, which is a promising outcome for the optimization of other computationally intensive problems. The parallelization of the fitness evaluation process markedly reduces the time required for the algorithm to complete, leading to faster convergence to equally accurate solutions as a GA – which is only marginally easier to implement. Overall, the results suggest that parallelization is a viable strategy for improving the performance of genetic algorithms, and it has the potential to be applied to a wide range of optimization problems in various fields. Future papers could explore the differences in solution accuracy when using an island model PGA, as well as with the tuning of each island's hyperparameters.

References

[1] Z. Konfrst, "Parallel genetic algorithms: advances, computing trends, applications and perspectives," 18th International Parallel and Distributed Processing Symposium, 2004

[2] D.E. Goldberg and R. Lingle, Jr., Alleles, loci, and the traveling salesman problem, in:Proc. Int. Conf. on Genetic Algorithms and Their Applications, 1985, 154–159

[3] Cantú-Paz, E. ”A survey of parallel genetic algorithms.” Calculateurs paralleles, reseaux et systems repartis, 1998
[4] R K Nayak, B S P Mishra and Jnyanaranjan Mohanty, “An overview of GA and PGA,” International Journal of Computer Applications 178(6):7-9, November 2017