

# ANÁLISIS DEL RENDIMIENTO DE UN ALGORITMO GENÉTICO PARALELO BASADO EN SPARK Y UN ALGORITMO GENÉTICO PARALELO BASADO EN THREADING PARA LA GENERACIÓN DE PAIRWISE TEST SUITE

Melvin Salcedo A.<sup>1</sup>, Ruth Huilca R.<sup>1</sup>, Gabriela Rojas C.<sup>1</sup>

<sup>1</sup>Universidad Nacional de San Agustín

El pairwise es una técnica efectiva para obtener las situaciones de pruebas requeridas para llegar a cierta cobertura, el objetivo del pairwise es cubrir todas las posibles combinaciones de dos factores. Los algoritmos genéticos han sido la herramienta de uso que muchos investigadores han usado para la generación de conjuntos de prueba por pares. Pero a pesar de la optimización que ofrecen los algoritmos genéticos este es un proceso que aún lleva mucho tiempo, y esto conduce a limitaciones para el uso de algoritmos genéticos hacia problemas de prueba a gran escala. Aplicar paralelismo nos permitirá mejorar el rendimiento de cómputo y así también mejorar la calidad de soluciones. En este documento realizamos el análisis del algoritmo genético paralelo basado en spark y pthreads, para la plataforma spark utilizamos una técnica que aborda dos fases una es la paralelización evaluación de aptitud y la paralelización de operación genética y la librería pthreads para la otra forma de paralelización del algoritmo genético. Como resultados de nuestros experimentos efectivamente comprobamos que el algoritmo genético paralelo basado en spark es mejor respecto a algoritmo genético secuencial y también respecto al algoritmo basado en pthreads, el algoritmo basado en spark muestra mejor rendimiento computacional.

## I. INTRODUCCIÓN

Las pruebas de software consisten en la dinámica de la verificación del comportamiento de un programa en un conjunto finito de casos de prueba, debidamente seleccionados de por lo general infinitas ejecuciones de dominio, contra la del comportamiento esperado. Los sistemas de software reales se convierten más complejos y tienden a tener demasiados parámetros, muchas fallas se desencadenan por las interacciones entre los parámetros del software. Esto originará una gran cantidad de combinaciones posibles de todos los valores de parámetros. Las pruebas exhaustivas prueban todas las posibles combinaciones de valores de parámetros para detectar fallas desencadenadas por interacción. Pese a esto las pruebas exhaustivas no son las adecuadas para situaciones reales.

Por ejemplo, si un sistema tiene 10 parámetros y se le puede asignar a cada parámetro uno de 10 valores diferentes, generaremos 1010 posibles combinaciones. Si cada combinación se ejecuta en un minuto, tomará alrededor de 20 mil años completar la prueba completa. Un mejor enfoque a este problema nos lo da Pairwise. El Pairwise Testing es una técnica básica propuesta por TMap Next para obtener las situaciones de prueba requeridas para llegar a una cierta cobertura. Su objetivo es probar todas las posibles combinaciones de dos factores. Esto redundará en una gran reducción en el número de casos de prueba, obteniéndose aún así buenos resultados en la detección de defectos. Esta técnica se basa en el hecho de que la mayor parte de los errores se producen como consecuencia de un factor concreto o de la combinación de dos factores. Debido a esto, en lugar de probar todas las posibles combinaciones de todos los factores, resulta muy efectivo probar cada combinación

de dos factores. Los investigadores del área han intentado varios métodos para generar un conjunto de pruebas casi mínimas. Nie y Leung presentaron cuatro grupos principales de enfoques: algoritmos codiciosos, algoritmos de búsqueda heurística, métodos matemáticos y métodos aleatorios. Entre estos métodos, los algoritmos codiciosos son los más ampliamente utilizados para la generación de pruebas combinatorias. Por lo general, son más rápidos que los algoritmos metaheurísticos, pero no siempre producen los conjuntos de pruebas más pequeños. Los algoritmos de búsqueda heurística formulan un problema de generación de prueba combinatoria como un problema de búsqueda y aplican técnicas de búsqueda como Hill Climbing (HC), Annealing simulado (SA), Algoritmo genético (GA), Algoritmo de colonia de hormigas (ACA) y así sucesivamente para resolver el problema. Estos algoritmos a menudo pueden producir un conjunto de pruebas más pequeño que los algoritmos codiciosos, pero normalmente requieren un cálculo más largo.

En este documento, desarrollaremos el análisis en cuanto a rendimiento de un algoritmo genético paralelo basado en Spark al cual llamaremos (GA), y un algoritmo genético paralelo basado en threads (python) para así acelerar el proceso de generación de pairwise al aplicar paralelismo podemos mejorar el rendimiento y la calidad de las soluciones, como resultado se presentará un análisis del performance de cada algoritmo.

Threading de Python es una biblioteca relacionada con Python para la programación de soluciones que emplean múltiples CPU o CPU multinúcleo en un entorno de multiprocesamiento simétrico (SMP) o memoria compartida, o potencialmente un gran número de computadoras en un clúster o entorno de grillas. En este artículo se trabajó bajo el principio de multiprocesamiento de memoria com-

partida. Spark es un framework de clúster rápida , es adecuada para manejar la paralelización de AG, en el siguiente documento la paralelización basada en Spark se dara en dos fases, como primera fase evaluamos el valor de la aptitud de cada individuo en paralelo, la segunda fase dividimos la población en diferentes sectores que pueden mutarse por separado. A continuación resumiremos las principales fases del documento:

- 1) Se implementará la paralelización en un algoritmo genético basado en Spark [1] para la generación de suite de prueba por pares proporcionando la evaluación de este en cuanto a efectividad.
- 2) Seguidamente se presentará la evaluación en cuanto a rendimiento del algoritmo genético paralelizado basado en spark sobre el algoritmo genético lineal y posteriormente se darán las conclusiones del trabajo realizado.

## II. SPARK

Spark proporciona una interfaz para programar clusters completos con paralelismo de datos implícitos y tolerancia a fallas. Tiene como fundamento arquitectónico el conjunto de datos resiliente distribuida (RDD), una de sólo lectura multiset de elementos de datos distribuidos sobre un grupo de máquinas.

Spark y sus RDD se desarrollaron en respuesta a las limitaciones del paradigma de computación de clústeres MapReduce , que fuerza una estructura de flujo de datos lineal particular en los programas distribuidos. Los RDD de Spark funcionan como un conjunto de trabajo para programas distribuidos que ofrece una forma restringida de memoria compartida distribuida.

## III. THREADING DE PYTHON

Es una biblioteca relacionada con Python para la programación de soluciones que emplean múltiples CPU o CPU multinúcleo en un entorno de multiprocesamiento simétrico (SMP) o memoria compartida, o potencialmente un gran número de computadoras en un clúster o entorno de grillas. En este artículo se trabajo bajo el principio multiprocesamiento memoria compartida.

## IV. PAIRWISE TESTING

Al generar un conjunto de pruebas con pruebas por pares , el espacio de entrada del software bajo prueba (SBP), se modela como una colección de parámetros , cada uno de los cuales asume uno o más valores. La prueba prueba por pares tiene como objetivo seleccionar un subconjunto completo de combinaciones de valores de parámetros de modo que todos los pares de valores de parámetros estén en el subconjunto seleccionado.

Cada combinación de valor de parámetro seleccionada generará al menos un caso de prueba para SBP. El conjunto de casos de prueba a comunmente se denomina conjunto de prueba que se representa mediante el array de cobertura (AC) definido a continuación :

**Definición 1 (Array de Cobertura)** Dejamos que SBP tenga  $k$  parámetros y que cada parámetro  $P_i$  tenga valores  $v_i (1 \leq i \leq k)$ .

Un array de cobertura  $AC (N; v_1^{p_1} v_2^{p_2} \dots v_k^{p_k}, t)$  es una matriz de  $N \times k$ . Cada fila de esta matriz es un caso de prueba ,  $N$  es la cantidad de casos de prueba y  $t$  es la fuerza del array de cobertura. Cada subarray contiene al menos una ocurrencia de prueba de cada tupla  $t$ -tuple correspondiente a las  $t$  columnas.

Si  $v_1 = v_2 = \dots = v_k = v$ , se dice que el array de cobertura es uniforme y se denota como  $AC (N; v^k, t)$

Cuando  $t = 2$ , llama al array de cobertura bidireccional. La prueba con un array de cobertura bidireccional se denomina prueba de pares **pairwise**. El objetivo del pairwise es reducir el número de casos de prueba. Por ejemplo, suponemos que SBP tiene tres parámetros:  $p_0$ ,  $p_1$  y  $p_2$ . Los valores posibles para cada parámetro son  $\{a_0, a_1\}$ ,  $\{b_0, b_1\}$  y  $\{c_0, c_1\}$  respectivamente. El número total de combinaciones posibles de todos los valores de parámetros es  $2^3$  :

$(a_0, b_0, c_0), (a_0, b_0, c_1), (a_0, b_1, c_0), (a_0, b_1, c_1),$   
 $(a_1, b_0, c_0), (a_1, b_0, c_1), (a_1, b_1, c_0), (a_1, b_1, c_1).$

Al probar este SBP con pairwise, los casos de prueba generados cubrirán cada par de valores de parámetros .

Existen 12 de esos pares:

$\{a_0, b_0\}, \{a_0, b_1\}, \{a_0, c_0\}, \{a_0, c_1\},$   
 $\{a_1, b_0\}, \{a_1, b_1\}, \{a_1, c_0\}, \{a_1, c_1\},$   
 $\{b_0, c_0\}, \{b_0, c_1\}, \{b_1, c_0\}, \{b_1, c_1\}.$

En este caso, el siguiente conjunto de cuatro combinaciones es suficiente:

$(a_0, b_0, c_1), (a_0, b_1, c_0), (a_1, b_0, c_0), (a_1, b_1, c_1).$

La Tabla 1 muestra el array de cobertura AC  $(4; 2^3, 2)$  para el SBP.

	p0	p1	p2
t <sub>1</sub>	a <sub>0</sub>	b <sub>0</sub>	c <sub>1</sub>
t <sub>2</sub>	a <sub>0</sub>	b <sub>1</sub>	c <sub>0</sub>
t <sub>3</sub>	a <sub>1</sub>	b <sub>0</sub>	c <sub>0</sub>
t <sub>4</sub>	a <sub>1</sub>	b <sub>0</sub>	c <sub>1</sub>

Cuadro I. Array de cobertura  $4; 2^3, 2$

En el ejemplo anterior, se puede ver que las pruebas por parejas pueden reducir el número requerido de casos de prueba de 8 a 4, una reducción del 50 %. Con combinaciones de valores de parámetros más grandes, el resultado será mejor. En este artículo, se tratará de generar un array de cobertura casi mínima (conjunto de pruebas) para alcanzar una cobertura del 100 % por pares para el SBP.

## V. DISEÑO DEL ALGORITMO GENÉTICO (GA)

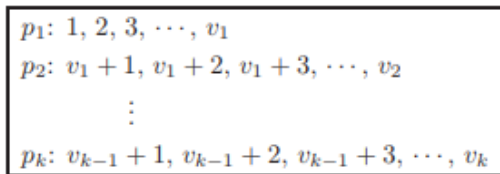
GA es una técnica de búsqueda metaheurística que simula la evolución de los sistemas naturales. Es de uso común para resolver problemas de búsqueda y optimización. En general, no es factible evaluar exhaustivamente el espacio de entrada completo y, por lo tanto, GA se utiliza para producir buenas soluciones en un tiempo razonable al evaluar solo una pequeña porción del espacio de entrada. El GA básico primero construye una población inicial al azar y luego itera a través de los siguientes procedimientos hasta que se cumplen los criterios de detención. Evalúa el valor de aptitud de todos los individuos en la población. Los individuos con valores altos de aptitud tienen una mejor oportunidad de evolucionar hacia la próxima generación mediante la aplicación de operadores genéticos como la selección, el cruce y la mutación. Cuando se usa GA para resolver un problema de generación de prueba por pares, se deben tomar las siguientes decisiones de diseño: codificación cromosómica, función de aptitud física (fitness) y operadores genéticos.

### A. Codificación cromosómica

La codificación cromosómica es la representación de un individuo que es la solución candidata al problema. En el escenario de generación de prueba por pares, la solución suele ser un conjunto de prueba adecuado de SBP. Existen diversos métodos de codificación en este artículo se hará uso de la codificación de enteros para representar al individuo. Este método de codificación codifica un conjunto de casos de prueba como un array de valores enteros. Cada entero corresponde a un posible valor de un parámetro de SBP. Por lo tanto un individuo es un array de listas de enteros y cada lista representa un caso de prueba, la longitud de cada lista de prueba es igual a la cantidad de parámetros, el tamaño de un individuo esta denotado por  $m$ , es el número de casos de prueba.

De acuerdo con la definición 1, el SBP tiene  $k$  parámetros y cada parámetro  $p_i$  tiene valores  $v_i$  ( $1 \leq i \leq k$ ).

Estos parámetros y valores se expresan mediante un archivo de texto como se ilustra en la Fig.1. Este archivo de texto se usará como entrada para nuestro algoritmo.



```
p1: 1, 2, 3, ..., v1
p2: v1 + 1, v1 + 2, v1 + 3, ..., v2
    ⋮
pk: vk-1 + 1, vk-1 + 2, vk-1 + 3, ..., vk
```

Figura 1. Archivo de texto de entrada para SBP

### B. Función Fitness

Una función fitness para pair wise es a menudo un criterio de cobertura determinado que mide cuan bueno es un individuo. Por literatura se definió que el 100 % de cobertura por pares requiere que cada posible par de valores interesantes de cualquiera de los dos parámetros este incluido en algún caso de prueba. En este artículo se hará uso de esta cobertura al 100 % por pares como nuestra función fitness. Por lo tanto, la función fitness es el número total de pares diferentes cubiertos por todos los casos de pruebas en un individuo. Si un individuo cubre mas parejas diferentes que otros, es mejor que otros. Un individuo se convierte en una solución cuando cubre todos los pares.

### C. Operadores Genéticos

Los operadores genéticos incluyen la *selección*, el *cruce* y la *mutación*.

En cuanto al operador de *selección* se emplea una selección proporcional de aptitud para determinar que individuos se elegirán como padres para la reproducción. En la selección proporcional de aptitud, los individuos se seleccionan en proporción a sus valores de aptitud mas altos. Tenemos  $S =$

$$\sum_i f_i$$

que representa la suma de los valores de condición de aptitud de todos los individuos. Se escoge un número aleatorio de 0 a  $S$ , si  $n$  cae dentro del rango de algún individuo en el array de rangos individuales, este individuo es seleccionado.

Después de la selección, los padres seleccionados se copian y luego el cruce mezcla y combina partes de estos dos padres copiados para formar mejores hijos. El mecanismo de cruce utiliza cruce aleatorio de punto único y punto múltiple. El cruce aleatorio de punto único elige un número  $c$  al azar de 0 a la longitud de un individuo e intercambia todos los índices menores que  $c$ . En el cruce de puntos múltiples, los individuos se consideran como un anillo. Se seleccionan varios puntos únicos al azar, dividiendo el anillo en varios segmentos. Un segmento de un anillo se intercambia con uno de otro anillo para producir nuevos descendientes. Después del cruce, utilizamos la mutación aleatoria para cambiar los genes de nuevos hijos generados con una tasa de mutación mas alta para encontrar una solución mas rápida. Para encontrar la solución mas rápida, el algoritmo muta el mejor individuo al final de cada generación para generar un nuevo mutante que reemplaza al individuo con el valor de fitness mas bajo. Este mejor individuo aún se mantiene en la población futura.

## VI. ALGORITMO GENÉTICO PARALELO

Existen cuatro modelos paralelos: modelo global, modelo distribuido, modelo celular y modelo híbrido. *Spark* se basa en el modelo híbrido. *Spark* se basa en el modelo de computación distribuida maestro-esclavo y es adecuado para el modelo global y distribuido de paralelización de AG. Se implementa un algoritmo de paralelización en dos fases: evaluación del fitness y operación genética, ambos con el mismo objetivo de mejorar el rendimiento y efectividad en la búsqueda de un conjunto de pruebas casi mínimas.

### A. Arquitectura

El algoritmo de paralelización en dos fases se basa en el conjunto de datos distribuidos (RDD) propio de *Spark*. Toda la población se almacena como RDD y se almacena en memoria caché, lo que permite que las acciones futuras sean mucho más rápidas. La arquitectura se muestra en la Fig. 2. La evaluación del fitness es la primera fase de paralelización que incluye las etapas 1 a 3. La etapa 1 genera la población inicial que luego se paraleliza en diferentes particiones de RDD en la etapa 2. El valor del fitness de cada individuo se evalúa en diferentes particiones desde la etapa 2 hasta la etapa 3. La paralelización de la primera fase se ajusta al modelo global.

La operación genética es la segunda fase de paralelización que incluye las etapas 4 a 6. La etapa 4 divide la población con un valor de fitness en números de subpoblaciones que se almacenan en caché en diferentes particiones de RDD. Las operaciones genéticas se realizan en paralelo desde la etapa 4 a la etapa 5. La etapa 6 recoge los mejores individuos de diferentes particiones. La segunda fase de paralelización se ajusta al modelo distribuido.

### B. Evaluación paralela del Fitness

La idea principal de la paralelización del fitness es paralelizar la población inicial en un RDD y evaluar el valor del fitness en diferentes áreas de trabajo del clúster. Luego se recopilan los resultados y aplica operadores genéticos. El gráfico de origen de la evaluación del fitness paralelo se muestra en la fig. 3. Primero la población inicial se paraleliza en la población RDD mediante el método *parallelize()* de *Spark*. Luego se aplica una transformación de mapa *map(\_assessFitness())* para la transformación la población RDD en un RDD de fitness que contiene pares (individuo, fitness). La función *assessFitness()* evalúa el fitness del individuo y pasa este valor al programa del controlador para ejecutarse en el clúster. Para terminar la función *collect()* comienza a recopilar estos pares para el controlador.

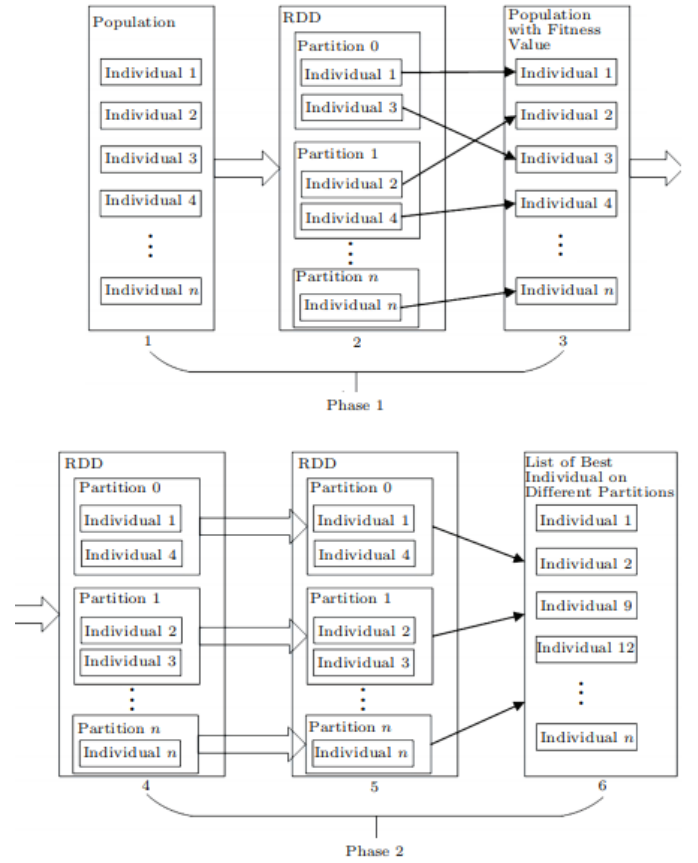


Figura 2. Arquitectura de las dos fases de paralelización

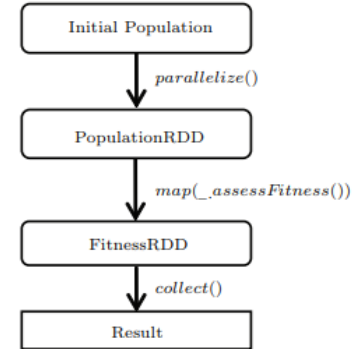


Figura 3. Origen de RDD para la fase 1

### C. Operación genética paralela

Una vez que el resultado de la evaluación paralela del Fitness retorna al controlador, si el resultado no contiene la solución el algoritmo sigue aplicando una operación genética paralela. El gráfico del origen de la operación paralela se muestra en la fig. 4. Como primer paso, la población con valor de fitness se paraleliza con

la `populationRDD` mediante el método `parallelize()` de Spark. Luego se aplica una transformación de mapa `map(_assessFitness())` divide la población en números de subpoblaciones que se almacenan en el caché en diferentes particiones de `populationRDD` y transforman `populationRDD` en `evolutionRDD`. La función `evolution()` que aplica los operadores genéticos se pasa al programa del controlador para ejecutarse en el clúster. Seguido de esto, se aplica otra transformación de mapa `map(_getBest())` para transformar `evolutionRDD` en `bestIndividualRDD` (mejor individuo) que contiene el par (clave,fitness) y el fitness es la cantidad necesaria para encontrar una solución. La función `map(_getBest())` que obtiene el mejor individuo de cada subpoblación pasa al programa del controlador para ejecutarse en el clúster. Para culminar la función `collect()` recolecta los mejores individuos de cada estación de trabajo para encontrar la mejor solución.

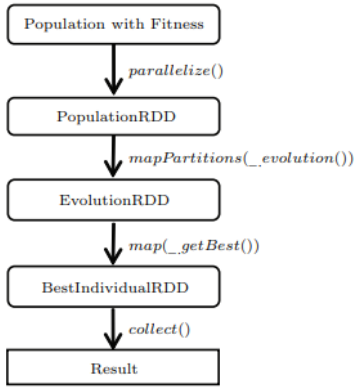


Figura 4. Origen de RDD para la fase 2

#### D. Algoritmos

El algoritmo 1 esboza el pseudocódigo de paralelización en dos fases. Toma como entradas el archivo de texto de valores de parámetros, el número de parámetros, el número de valores para cada parámetro, el tamaño del conjunto de pruebas y el tamaño de población deseado. El resultado es un conjunto de pruebas casi mínimo con una cobertura del 100 por ciento por pares. Al principio, se genera el número de todos los pares a cubrir. La población inicial consiste en individuos de tamaño natural, cada uno de los cuales se compone de  $m$  casos de prueba que se crean aleatoriamente seleccionando cada espacio de manera uniforme entre todos los valores posibles. Luego el algoritmo entra en la paralelización de la primera fase. Luego, los pares clave-valor recopilados por el conductor se ordenan por el fitness. Si el valor del primer par es AP, su clave es el mejor individuo que se devolverá. De lo contrario, el algoritmo entra en la paralelización de la segunda fase. Luego, la lista de pares(clave, valor), donde la clave es un par (individuo, fitness) y el valor es

la cantidad de generación necesaria para encontrar una solución, se ordena por el fitness y el número de generación. Si el valor del primer par es AP, su clave es el mejor individuo que se devolverá.

---

**Input:** *pv.txt*: parameter-values text file  
*k*: number of parameters in SUT  
*v<sub>i</sub>*: number of values for each parameter  
*m*: test suite size  
*popsiz*: desired population size

**Output:** a pairwise test suite that covers all pairs of values at least once

```

1: AP ← getNumOfAllPairs(k, vi)
2: Population ← initializePop(m, popsiz, "pv.txt")
3: populationRDD ← parallelize(Population)
4: For each individual Ij ∈ populationRDD do
5:   fitnessRDD ← Ij.map(_assessFitness())
6: End for
7: result ← fitnessRDD.collect()
8: sortByValue(result)
9: If result.top = AP then return top
10: populationRDD ← parallelize(PopWithFitness)
11: For each subPopulation Ij ∈ populationRDD do
12:   evolutionRDD ← Ij.mapPartitions(_evolution())
13:   bestIndividualRDD ← evolutionRDD.map(_getBest())
14: End for
15: result ← bestIndividualRDD.collect()
16: sortByValue(result)
17: If result.top = AP then return top
  
```

---

Figura 5. Algoritmo genético paralelo

En el algoritmo 1, se pasan tres funciones en el programa del controlador para ejecutar en el clúster: `assessFitness()`, `evolution()` y `getBest()`.

El algoritmo 2 describe el proceso de evolución. Toma como entradas la población, el número máximo de generación y el tamaño de población deseado. Las poblaciones evolucionadas en cada trabajador se generan. El bucle externo conduce todo el proceso de evolución. En cada iteración del bucle externo, el algoritmo ingresa en un bucle interno que aplica operadores genéticos que incluyen la selección, el cruce, la mutación y el reemplazo. Después de abandonar el ciclo interno, el algoritmo muta el mejor individuo. Luego, el algoritmo ingresa al bucle externo para comenzar nuevamente la evolución. El proceso de evolución continúa hasta que alcanza el número máximo de generaciones o se ha encontrado la solución. Finalmente, devuelve las poblaciones evolucionadas al Algoritmo 1.

#### VII. EVALUACIÓN PRELIMINAR

Estudiamos el comportamiento de PGAS al resolver un problema de generación de suite de prueba por pares. Realizamos dos categorías de experimentos: la comparación con el algoritmo genético secuencial (SGA), la comparación con la paralelizada.

---

**Input:**  $P$ : population  
 $max$ : maximum number of generation  
 $popsiz$ : desired population size

**Output:** evolved populations on every worker

```

1:  $it \leftarrow 1$ 
2: While ( $it \leq max$  && the ideal solution not found)
3:   For  $popsiz/2$  times do
4:     Parent  $P_1 \leftarrow Selection(P)$ 
5:     Parent  $P_2 \leftarrow Selection(P)$ 
6:     Children  $C_1, C_2 \leftarrow Crossover(Copy(P_1), Copy(P_2))$ 
7:      $Mutate(C_1, C_2)$ 
8:      $ReplaceWorstIndividual(P)$ 
9:   End for
10:  Mutate the best individual
11:   $it \leftarrow it + 1$ 
12: End While
13: Return the evolved populations

```

---

Figura 6. Evolución de procesos

### A. Diseño experimental

En los experimentos, implementamos PGAS en Spark usando python y ejecutamos PGAS en un pequeño grupo compuesto por cinco nodos, donde cada nodo tiene un procesador Intel Core i5 750 Quad-Core a 2.66 GHz CPU, 4 GB de RAM. Un nodo es el namenode y los otros cuatro nodos son los nodos de datos que tienen 16 núcleos en total. Cada nodo se ejecuta en Windows 10, Python 3.5 y Spark 1.1.0.

Parameter name	setting1	setting2	setting3
Population size	4800	8000	12800
Crossover	1-point	3-point	5-point
Mutation	2-point	5-point	5-point
Maximum number of generations	100000	200000	500000

Cuadro II. Configuración de parámetros

Se hizo una comparación de tiempos en modo paralelo y serial probando que la forma paralela es mucho más veloz dando más eficiencia al algoritmo

Población	4800	8000	12800
Tiempo Spark	38.20s	67.28s	122.820s
Tiempo lineal	57.65s	109.46s	264.89s
Tiempo threading(Python)	21.9s	42.5s	102.67s

Cuadro III. Configuración de parámetros

en la Fig.7 de muestra el gráfico de barras de los resultados:

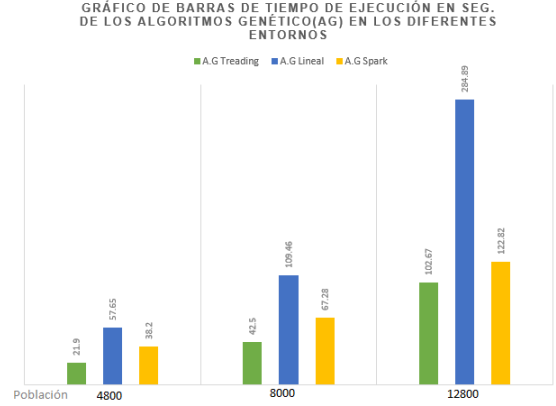


Figura 7. Gráfico de barras

### B. Conclusiones

Se propuso un algoritmo genético paralelo basado en threading y se hizo la réplica y un análisis de un algoritmo genético paralelo usando Spark llamado PGAS, para la generación de conjuntos de prueba de pares.

Según nuestros experimentos las pruebas realizadas Spark no muestra una diferencia alta en cuanto a eficiencia debido a que la data no es sobrecargada, sin embargo quedó demostrado que cuando la data es considerablemente alta entonces Spark respectivamente muestra mejores resultados en cuanto tiempo de ejecución.

Respecto a la eficiencia de Spark frente al algoritmo lineal si es de considerarse. Observamos que el tiempo de ejecución con los valores mínimos de parámetros tanto en el Spark y Treading(Python) no varían mucho. Realizamos una comparación donde resulta ser más eficiente en Threading(Python).

Demostramos también que la forma paralelizada es más eficiente que la versión serial. Debido a factores externos a la implementación del algoritmo los tiempos varían.

[1] A Parallel Genetic Algorithm Based on Spark for Pairwise Test Suite Generation- Rong-Zhi Qi Member, CCF, Zhi-Jian Wang 1, Member, IEEE, and Shui-Yan Li 2

[2] <http://ampcamp.berkeley.edu/wp-content/uploads/2012/06/matei-zaharia-part-1-ampcamp-2012-spark-intro.pdf>