Verificación de Covering Arrays

Armando Isaac Hernández Muñiz

Cinvestav Unidad Tamaulipas
Cómputo paralelo
Dr. Mario Garza Fabre

Abstract. Los Covering Arrays (Matrices de Cobertura) son objetos combinatorios que han sido utilizados para automatizar la generación de casos de prueba para pruebas de software. Las características de un CA es que son de cardinalidad mínima, y de máxima cobertura. En pocos casos es conocido una solución óptima para construir CAs, pero en general, el problema de construcción de CAs es un difícil problema de optimización combinatoria. En este documento se describe la impremeditación un método de construcción de Covering Arrays con el uso de los GTPs (Greather Than Polynomials) y así poder analizar si es un CA valido teniendo que cumplir con todas sus características especificas [1].

1 Introducción

Los *covering arrays* derivaron de los diseños conocidos como *Orthogonal Arrays* (Matriz Ortogonal). Un *Ortogonal array* $OA\gamma=(N;t;k;v)$ es una matriz de $N * k$ dentro del conjunto $Z_v = \{0,1,..., v-1\}$ con la propiedad de que cada sub-matriz de t distintas columnas cubren exactamente $\gamma \geq 1$ veces cada tupla del conjunto Z_v^t. En una matriz ortogonal $OA\gamma=(N;t;k;v)$ los parámetros N y k son las dimensiones de la matriz, el parámetro v es el orden o el numero de símbolos en cada columna; el parámetro t es la fuerza de cobertura de interacciones, y por ultimo el parámetro γ es el numero de veces de cada tupla del Z_v^t aparece en cada combinación de t columnas distintas. Los covering arrays son diseños combinatorios muy similares a los *orthogonal arrays*; la diferencia es que en los covering arrays cada combinación de t columnas distintas cubren cada tupla del conjunto Z_v^t al menos una vez.
Dados los valores de t, k y v el problema de construcción de covering arrays es el problema de generación de $CA(N;t;k;v)$ con el numero mínimo de filas N. Este problema es muy difícil por lo general en os valores de t, k y v. La N mas pequeña para el cual un covering array es el *covering array number (CAN)* para los parámetros t, k y v, y es denotado por: $CAN(t,k,v) = min\{ N| \exists\ CA(N;t;k;v)\}$. También es presentado un sistema de numeracion que asocia a un vector $V = (v_1,...,v_m)$ con un numero natural α donde las entradas satisfacen $v_i < v_{i+1}$, $v_1 \geq 0$. Los valores v_i son usados en coeficientes binomiales resumido en $\alpha = \sum_{i=1}^{m} \binom{v_i}{i}$ para definir unicamente un entero $\alpha \in N$. Este sistema de numeración es llamado *Greater-Than Polynomials* (GTP) porque la expansion de los coeficientes binomiales produce un polinomio en $(v_1,...,v_m)$ y los componentes del vector V estan incrementando monotónicamente [2].

2 Descripción del Problema

Estudios empíricos de pruebas de software han mostrado que las pruebas de interaccion combinatoria son un enfoque util que garantiza la funcionalidad de los componentes de software. El objetivo matemático que da apoyo a las pruebas de interaccion combinatoria es el *Covering Array*. Un CA , denotado por *CA(N;t;k;v)*, es una matriz de $N * k$, donde cada entrada de la matriz toma valores de un conjunto de simbolos de tamaño v, tal que cada $N * t$ sub-matriz contiene todos los posibles v^t *t-tuplas* al menos una vez [2].

$$\mathrm{CA}(12;2,7,3) = \begin{pmatrix} 0 & 0 & 1 & 2 & 0 & 0 & 0 \\ 0 & 2 & 2 & 1 & 1 & 2 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 0 & 0 & 1 & 0 \\ 2 & 2 & 0 & 1 & 0 & 1 & 1 \\ 2 & 0 & 0 & 0 & 2 & 2 & 0 \\ 1 & 2 & 1 & 2 & 2 & 2 & 1 \\ 2 & 1 & 2 & 2 & 1 & 0 & 1 \\ 1 & 0 & 2 & 1 & 2 & 0 & 2 \\ 1 & 2 & 0 & 0 & 1 & 0 & 2 \\ 0 & 1 & 0 & 2 & 2 & 1 & 2 \\ 2 & 1 & 1 & 1 & 0 & 2 & 2 \end{pmatrix}$$

Fig. 1.

Formalmente, un covering array *CA(N;t;k;v)* con fuerza t y orden v es una matriz de tamaño *N*k* sobre los simbolos Z_v = {*1,....,v-1*} tal que cada sub-matriz de tamaño $N * t$ contiene en la fila cada t-tupla en Z_v al menos una vez. Un CA de fuerza t asegura la cobertura de todas las posibles combinaciones de los valores entre cualquier t columnas. Un ejemplo de CA se muestra en la **Figura 1**. En este CA cada sub-matriz de $t = 2$ columnas cubren al menos una vez cada posible t-tupla sobre Z_3 = {*0,1,2*} de las cuales estan las tuplas (0,0), (0,1),(0,2), (1,0), (1,1), (1,2), (2,0), (2,1) y (2,2). En la sub-matriz formada por las primeras dos columnas la primera ocurrencia de estas nueve tuplas estan coloreadas. Algunas tuplas pueden ocurrir mas de una vez, pero el requisito es que todas ellas ocurran al menos una vez por cada t distintas columnas [3].

3 Implementación del Algoritmo

3.1 Versión Secuencial

Para generar un CA son se necesitan parametros de entrada como lo son N(filas), t(fuerza), k(columnas) y v(alfabeto). En el programa solo se tiene como referencia el valor de k, t y v, por lo que falta calcular N, para ello se hace referencia a la siguiente formula:

$$k \leq \binom{N-1}{\frac{N}{2}}$$

El **Algoritmo 1** muestra el procedimiento para calcular el valor de N dado el valor de k:

Algorithm 1 N_value(k)

```
N := 4
while true do
    if (N-1 choose N/2) ≥ k then
        break
    end if
    N := N + 1
end while
return N
```

Primero se inicializa la variable N en 4 como mínimo valor posible, después se evalúa que $\binom{N-1}{\frac{N}{2}} \geq k$, si esta condición se cumple entonces se devuelve el valor de N, de lo contrario incrementa para ser evaluado en la siguiente iteración. Este proceso se repite hasta que se cumpla la condición y obtener el valor final de N.

El **Algoritmo 2** muestra la implementación del método el cual se encarga de realizar la construcción de un covering array dependiendo de los valores de k, t y v.
Primero se declaran las variables dependientes v que es el alfabeto, N es el numero de filas del CA, k numero de columnas y m longitud del GTP. Los GTPs se utilizan para saber los índices de las columnas del CA en las que pueden existir las v^t combinaciones. La variable *num* es el numero total de GTPs que se pueden hacer con el alfabeto v los cuales se guardan en *c_array_alfabeto* y por cada uno se genera un covering array. Cada una de las filas de *c_array_alfabeto* genera dos valores llamando a la función *inverseGTP*, ésta función convierte un numero natural a un vector GTP y tiene como primer parámetro el numero, y como segundo parámetro la longitud del GTP que se desea generar. por cada fila se genera un CA de N filas utilizando los dos valores como el alfabeto. Después, la variable *filas_ca* que guarda el numero total de filas del CA, ya que se calculan

num covering arrays de tamaño *N*, cada uno utilizando los valores del alfabeto. se declara el covering array en la variable *CA*, en el ciclo for se inicializa la *N*-ésima parte del CA de *filas_ca* renglones en cada iteración, con el primer índice del gtp correspondiente de longitud 2 que se almacena en *c_array_alfabeto*[c][0], donde *c* es el numero de fila con el gtp con el que se construye el *N*-ésimo CA. Por ultimo se rellena cada columna del CA con el valor *c_array_alfabeto*[c][1] desde la fila 0 hasta *N-1* tomando como índices los valores del GTP generado por cada columna.

Algorithm 2 CoveringArray()

Require v := numero de simbolos (alfabeto)
Require k := numero de columnas del CA
Require N := *N_Value(k)*
Require m := longitud de GTP
num := $\binom{v}{2}$
c_array_alfabeto = matriz de *num* filas
for i := 0 hasta *num* **do**
    c_array_alfabeto[i] = ***inverseGTP****(i, 2)*
end for
filas_ca := *num* * *N*
CA := matriz de *filas_ca* * *k*
fila := 0
for c := 0 hasta *num* **do**
    Se inicializa la matriz *CA* con el valor de *c_array_alfabeto*[c][0]
    for i := 0 hasta *k* **do**
        gtp := ***inverseGTP****(i, m)*
        CA[gtp][i] = c_array_alfabeto[c][1]
    end for
    fila := *fila* + *n*
end for

El **Algoritmo 3** muestra el metodo *Comparations* que verifica si el CA es valido o no. La variable *c* guarda el valor de $\binom{k}{t}$ que es el numero de combinaciones de longitud t de los índices de columna de un CA. Se crea el vector de *gtps* que guarda el orden de los indices de las columnas en donde se compara cada tupla de las combinaciones de longitud *t*. El ciclo de *c* iteraciones recorre cada gtp generado el cual es usado para comparar cada t columna para verificar que existan todas las posibles combinaciones. El contador *exist* lleva el conteo del total de combinaciones que van ocurriendo en el covering array. Para que el CA sea correcto, se tiene que cumplir que $exist = v^t$, de lo contrario sera incorrecto.

Algorithm 3 Comparations()

```
Require v := numero de simbolos (alfabeto)
Require n := numero de filas del CA
Require k := numero de columnas del CA
Require m := longitud de GTP
Require t := fuerza
c := (k t)
gtps := vector de c posiciones
for i := 0 hasta c do
    gtps[i] := inverseGTP(i, t)
    exist := 0
    for j := 0 hasta v^t do
        for n := 0 hasta total de filas del CA do
            if existe la combinacion j en CA[n][gtps] then
                exist := exist + 1
                break
            end if
        end for
    end for
    if exist < v^t then
        CA incorrecto
    else
        CA correcto
    end if
end for
```

3.2 Versión Paralela (Pthreads)

Para construir el programa paralelo se identificaron los pasos para diseñarlo.

- **Particionamiento:** En el algoritmo se identificaron dos partes que pueden ser paralelizadas ya que son las que mas tiempo de ejecución ocupan para procesar toda la tarea. Se seleccionaron los métodos de construcción (*Algoritmo 2*) y comparación (*Algoritmo 3*) de CA.

- **Comunicación:** La comunicación que hay entre estas tareas son independientes una de la otra, pero primero se debe ejecutar la tarea de construcción del CA antes de poder compararla, por lo que tendría primero que ejecutarse una y luego la otra.

- **Agregación:** Para el caso del algoritmo de construcción, cada procesador ejecuta $\frac{filas\_ca}{p}$ tareas, donde *filas_ca* es el numero total de filas del CA, y p es el numero de procesos. En el caso del algoritmo de comparación también se hace la misma división de tareas, solo que en este caso a cada proceso le tocan $\frac{c}{p}$, donde c es el numero de combinaciones de longitud t de los índices de columna de un CA.

– **Mapeo:** Finalmente se distribuyen las tareas al numero de procesos que se requieren de forma equitativa, en caso de que el numero total de tareas no sea divisible entre el numero de procesos, entonces las tareas restantes se asignan a distintos procesos o hilos.

Algorithm 4 CoveringArray(id_hilo)

```
tareas := filas_ca/p
inicio := tareas * id_hilo
fin := inicio + tareas

for i := inicio hasta fin do
    c := 0
    for i := 1 hasta num do
        if i ≥ N_value(k)*a then
            c := c + 1
        end if
    end for
    if i % N_value(k) is 0 then
        Se inicializa la matriz CA con el valor de c_array_alfabeto[c][0]
        for i := 0 hasta k do
            gtp := inverseGTP(i, m)
            CA[gtp][i] = c_array_alfabeto[c][1]
        end for
    end if
end for
```

El **Algortimo 4** muestra la versión paralela del **Algoritmo 2**. En esta caso el metodo *CoveringArray* sera llamado por cada hilo o proceso para ejecutar las tareas a la vez, para eso la funcion debe recibir el identificador del hilo que lo esta trabajando, para así saber el rango de trabajo que debe realizar cada hilo. En si el algoritmo realiza el mismo procedimiento que el del primer algoritmo, solo que ahora en lugar de que se construya el CA en un solo ciclo, ahora cada proceso rellenara el CA al mismo tiempo pero diferentes rangos de la matriz completa. Para ello se agregaron las variables *tareas*, que indica la cantidad de filas que le toca rellenar de CA a cada hilo o proceso, el *inicio* indica desde que fila inicia cada hilo su tarea y *fin* indica donde termina, estas dos ultimas variables sirven de manera que diferentes hilos no sobreescriban valores ya asignados al CA. El **Algoritmo 5** trabaja de la misma manera, ya que es la version paralela del **Algoritmo 3**, a diferencia que este realiza la tarea de comparar un conjunto de $\binom{k}{t}$ combinaciones en las columnas del CA.

Algorithm 5 Comparations(id_hilo)

```
Global exist_all := 0
c := (k t)
tareas := c/p
inicio := tareas * id_hilo
fin := inicio + tareas

exist := 0
gtps := vector de c posiciones
for i := inicio hasta fin do
    gtps[i] := inverseGTP(i, t)
    exist := 0
    for j := 0 hasta v^t do
        for n := 0 hasta total de filas del CA do
            if existe la combinacion j en CA[n][gtps] then
                exist := exist + 1
                break
            end if
        end for
    end for
end for

pthread_mutex_lock
exist_all := exist_all + exist
pthread_mutex_unlock

pthread_barrier_wait
if exist < v^t *p then
    CA válido
else
    CA inválido
end if
```

Implementacion de Mutex y Barreras

El detalle del algoritmo paralelo de *Comparations* es que tiene una variable que cuanta el numero de combinaciones que se encuentran para determinar que todas existan y el CA declare como válido. Esa variable debe ser actualizada por cada proceso que ejecute la funcion, pero no al mismo tiempo ya que puede que el programa requiera mas tiempo y corre el riesgo de que obtenga valores equivocados debido a que todos los procesos acceden a la misma variable al mismo tiempo. Para ello se implemento el metodo **pthread_mutex_lock** para que cada proceso afecte la variable y bloquee la entrada a los demas procesos a la variable, una vez que actualiza desbloquea el paso con **pthread_mutex_unlock**. Ya que se obtenga la suma total de existencias de las combinaciones, este valor se verifica con el numero real de combinaciones para verificar que se cumplen.

Pero esta verificacion se debe hacer ya que todos los procesos/hilos terminen de ejecutar todas sus tareas, para ello se usa el metodo **pthread_barrier_wait**.

4 Resultados del algoritmo secuencial

A continuación se muestran los tiempos de ejecución del algoritmo secuencial comparando cada una de las instancias. Cada instancia se denota como *CA(N;t;k;v)* que es la forma en que se ve un Covering array, donde N es el numero de filas, t es la fuerza (longitud de tupla), k es el numero de columnas y v es el alfabeto (conjunto de símbolos).

Instancia	Tiempo de ejecución
CA(130;2;500;5)	8.740956
CA(210;2;800;6)	52.2733
CA(294;2;1000;7)	157.491165
CA(392;2;1500;8)	705.430359
CA(540;2;2000;9)	1934.234987

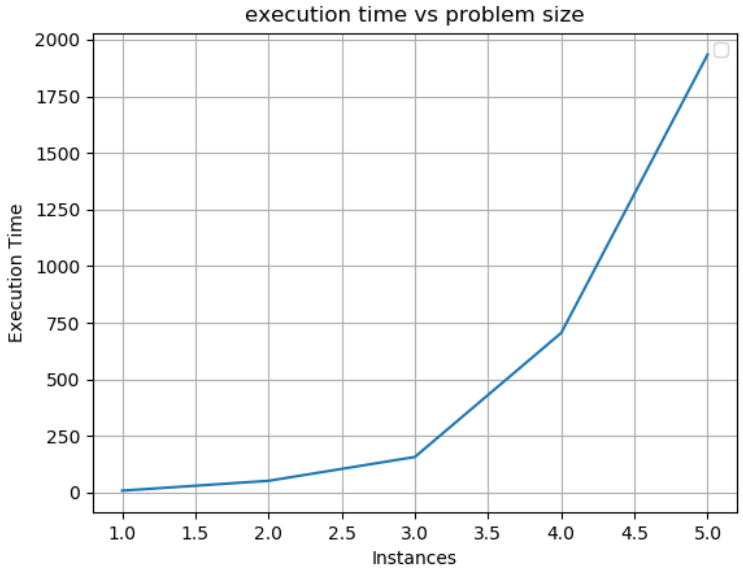

Fig. 2. En esta grafica se puede ver los tiempos de ejecucion de cada una de las cinco instancias. La instancia 5 es la de mayor tamaño y se ve a simple vista ya que toma un tiempo de casi 2000 segundos a diferencia de las demas. Esto dice que a mayor tamaño del problema, mayor tomara tiempo en procesar las tareas del algoritmo.

4.1 Ejemplo de salida del programa

```
./VCA_serial {argument1} {argument2} {argument3}
{argument1} = K value (columns)
{argument2} = t value (force)
{argument2} = v value (alphabet)
```

Exec:

```
./VCA_serial 4 2 3
K=4
N=5
M=3
Covering Array:
0 0 0 0
1 1 1 0
1 1 0 1
1 0 1 1
0 1 1 1
0 0 0 0
2 2 2 0
2 2 0 2
2 0 2 2
0 2 2 2
1 1 1 1
2 2 2 1
2 2 1 2
2 1 2 2
1 2 2 2
Alfabeto:
0, 0,
0, 1,
0, 2,
1, 0,
1, 1,
1, 2,
2, 0,
2, 1,
2, 2,
Covering Array Correcto: si
```

5 Resultados del algoritmo paralelo

A continuación se muestran los resultados del algoritmo paralelo para cada uno de los casos de prueba (tamaño del problema) comparando cada uno de ellos con el tiempo de ejecución, la aceleración y la eficiencia usando diferentes valores de los hilos de procesamiento en cada ejecución.

5.1 Tiempo de ejecución

La siguiente tabla muestra los tiempos de ejecución de cada una de las instancias usando diferentes valores de hilos.

Instancia	1 hilo	2 hilos	4 hilos	8 hilos	16 hilos	32 hilos	64 hilos	32 hilos
CA(130;2;500;5)	8.46	4.28	2.15	1.08	0.55	0.45	0.47	0.45
CA(210;2;800;6)	51.38	25.93	12.98	6.53	3.29	2.76	2.81	3.0066
CA(294;2;1000;7)	156.58	78.65	39.67	19.79	11.25	9.27	9.09	8.87
CA(392;2;1500;8)	622.81	310.12	156.55	78.75	49.28	38.11	37.83	35.70
CA(540;2;2000;9)	1938.50	977.61	490.73	245.88	123.007	102.06	102.31	106.28

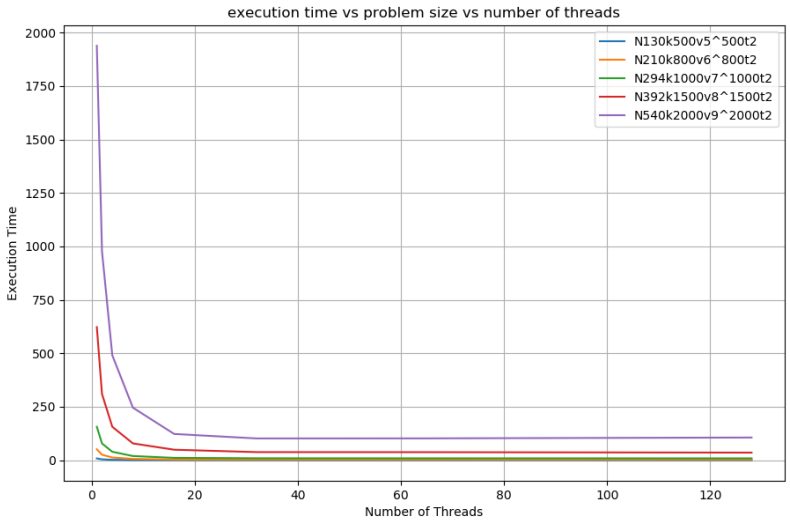

Fig. 3. La gráfica muestra los la comparación con el tiempo de ejecución, el tamaño del problema y el numero de hilos. En la tabla se puede ver que los tiempos de ejecución se parten por la mitad mientras el numero de hilos aumenta, hasta llegar al caso de 32 hilos, los tiempos se mantienen casi iguales a partir de ese valor.

5.2 Aceleración

La siguiente tabla muestra los valores de la aceleración de cada una de las instancias usando diferentes valores de hilos. Para obtener la aceleracion se toma en cuenta el tiempo del algoritmo secuencial y el paralelo [4]:

$$S = \frac{T_{serial}}{T_{paralelo}}$$

Instancia	1 hilo	2 hilos	4 hilos	8 hilos	16 hilos	32 hilos	64 hilos	32 hilos
CA(130;2;500;5)	1.03	2.04	4.04	8.02	15.86	19.05	18.29	19.15
CA(210;2;800;6)	1.01	2.01	4.02	8.002	15.84	18.93	18.57	17.38
CA(294;2;1000;7)	1.005	2.002	3.97	7.95	13.99	16.97	17.31	17.74
CA(392;2;1500;8)	1.13	2.27	4.50	8.95	14.31	18.50	18.64	19.75
CA(540;2;2000;9)	0.99	1.97	3.94	7.86	15.72	18.95	18.90	18.19

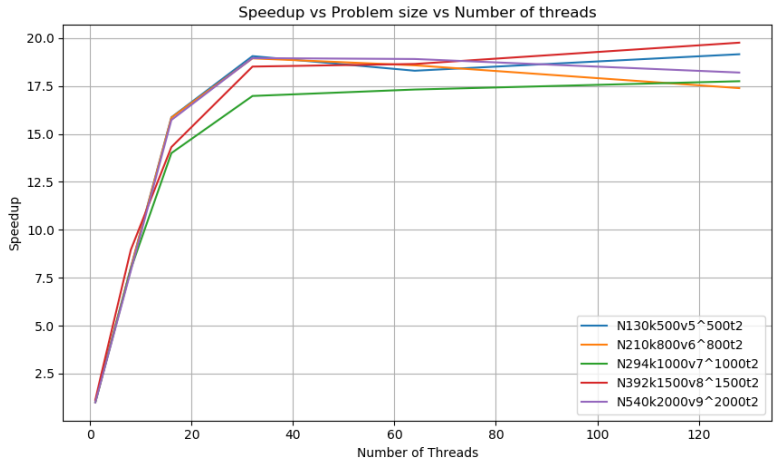

Fig. 4. La gráfica muestra los la comparación con la aceleración, el tamaño del problema y el numero de hilos. Se puede ver que la instancia que con poca frecuencia es *CA(294;2;1000,7) (linea verde)*, y la instancia *CA(392;2;1500;8) (linea roja)* acelera mas rápido a partir de los 64 hilos de ejecución.

5.3 Eficiencia

La siguiente tabla muestra los valores de la eficiencia de cada una de las instancias usando diferentes valores de hilos. Para obtener la eficiencia se toma en cuenta la aceleracion y el numero de procesos [4]:

$$E = \frac{S}{p} = \frac{\frac{T_{serial}}{T_{paralelo}}}{p} = \frac{T_{serial}}{p * T_{paralelo}}$$

Instancia	1 hilo	2 hilos	4 hilos	8 hilos	16 hilos	32 hilos	64 hilos	32 hilos
CA(130;2;500;5)	1.03	1.02	1.01	1.003	0.99	0.59	0.28	0.14
CA(210;2;800;6)	1.01	1.007	1.006	1.000	0.99	0.59	0.29	0.13
CA(294;2;1000;7)	1.005	1.001	0.99	0.994	0.87	0.53	0.27	0.13
CA(392;2;1500;8)	1.132	1.137	1.12	1.11	0.89	0.57	0.29	0.15
CA(540;2;2000;9)	0.99	0.989	0.985	0.983	0.982	0.59	0.29	0.14

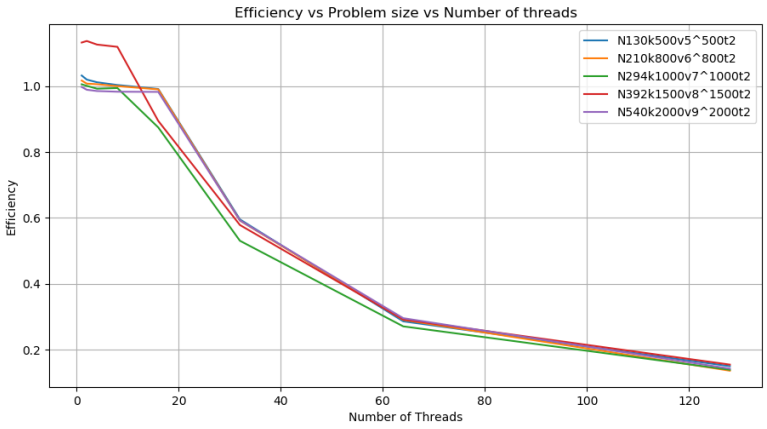

Fig. 5. La gráfica muestra los la comparación con la eficiencia, el tamaño del problema y el numero de hilos. Se puede ver que cada instancia va tomando una eficiencia similar por cada valor de los hilos de procesamiento.

6 Conclusiones

En este documento se implementaron los algoritmos secuencial y paralelo para el problema de construcción y Verificación de Covering Arrays con la ayuda de los GTPs (Greater-Than Polynomials), el cual el objetivo es que dado los valores de *k (columnas), t (fuerza)* y *v(alfabeto)* de un CA, verificar que el conjunto de v^t tuplas aparezcan al menos una vez en cualquier columna de todas las filas del Covering Array. Para la versión del algoritmo paralelo se identificaron los procesos mas costosos del algoritmo para así hacer una modificación a los métodos los cuales pueden ser paralelizados y de esta forma reducir el tiempo de ejecución dividiendo el trabajo en distintos procesos a la vez. Finalmente se comparan los resultados del algoritmo secuencial con el paralelo logrando reducir razonablemente el tiempo del algoritmo de acuerdo a un determinado tamaño de problema usando diferentes valores de hilos de procesamiento.

References

[1] Jose Torres-Jimenez and Idelfonso Izquierdo-Marquez. "Survey of covering arrays". In: *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE. 2013, pp. 20–27.

[2] Pepe Torres Jimenez et al. "Combinatorial Analysis of Diagonal, Box and Greater-Than Polynomials as Packing Functions". In: *Applied Mathematics & Information Sciences* 9.Applied Mathematics & Information Sciences (2015).

[3] Jose Torres-Jimenez, Idelfonso Izquierdo-Marquez, and Himer Avila-George. "Methods to Construct Uniform Covering Arrays". In: *IEEE Access* 7 (2019), pp. 42774–42797.

[4] Peter Pacheco. *An introduction to parallel programming*. Elsevier, 2011.