

# 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 implementación 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 específicas [1].

## 1 Introducción

Los *covering arrays* derivaron de los diseños conocidos como *Orthogonal Arrays* (Matriz Ortogonal). Un *Orthogonal array*  $OA_{\gamma}(N;t;k;v)$  es una matriz de  $N * k$  dentro del conjunto  $Z_v = \{0, 1, \dots, v-1\}$  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  $\gamma$  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 los 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 numeración que asocia a un vector  $V = (v_1, \dots, v_m)$  con un numero natural  $\alpha$  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  $(v_1, \dots, v_m)$  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 interacción combinatoria son un enfoque útil que garantiza la funcionalidad de los componentes de software. El objetivo matemático que da apoyo a las pruebas de interacción 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 símbolos de tamaño  $v$ , tal que cada  $N * t$  sub-matriz contiene todos los posibles  $v^t$   $t$ -tuplas al menos una vez [2].

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

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 símbolos  $Z_v = \{1, \dots, v-1\}$  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 están 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 están coloreadas. Algunas tuplas pueden ocurrir más 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  $\binom{N-1}{\frac{N}{2}} \geq k$  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**  
     $c\_array\_alfabeto[i] = \text{inverseGTP}(i, 2)$   
**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**  
         $gtp := \text{inverseGTP}(i, m)$   
         $CA[gtp][i] = c\_array\_alfabeto[c][1]$   
    **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 índices 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
Global  $CA$  := Covering Array ya generado
 $c := \binom{k}{t}$ 
 $gtps$  := 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.

- **Maapeo:** 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)

---

**Require**  $v$  := numero de simbolos (alfabeto)  
**Require**  $k$  := numero de columnas del CA  
**Require**  $N$  :=  $N\_Value(k)$   
**Require**  $m$  := longitud de GTP  
 $num := \binom{v}{2}$   
 $tareas := filas\_ca/p$   
 $inicio := tareas * id\_hilo$   
 $fin := inicio + tareas$

**for**  $i := inicio$  hasta  $fin$  **do**  
     $c := 0$   
    **for**  $j := 1$  hasta  $num$  **do**  
        **if**  $i \geq N\_value(k)*j$  **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 **Algoritmo 4** muestra la versión paralela del **Algoritmo 2**. En esta caso el método *CoveringArray* sera llamado por cada hilo o proceso para ejecutar las tareas a la vez, para eso la función 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 versión 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)

---

```

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
Global  $exist\_all$  := 0
 $c := \binom{k}{t}$ 
 $tareas := c/p$ 
 $inicio := tareas * id\_hilo$ 
 $fin := inicio + tareas$ 

 $exist := 0$ 
 $gtps :=$  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 función, 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 método **pthread\_mutex\_lock** para

que cada proceso afecte la variable y bloquee la entrada a los demás 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 verificación se debe hacer ya que todos los procesos/hilos terminen de ejecutar todas sus tareas, para ello se usa el método **pthread\_barrier\_wait**.

### Metodo Main con Pthreads

A continuación se muestra el llamado de las funciones en paralelo en el método main descrito en el **Algoritmo 6** utilizando también los mecanismos de paralelización *mutex* y *barrieras* antes descritos.

---

#### Algorithm 6 Main()

---

```

Global  $v$  := numero de simbolos (alfabeto)
Global  $k$  := numero de columnas del CA
Global  $N$  :=  $N\_Value(k)$ 
Global  $m$  := longitud de GTP
Global  $num\_hilos$  := numero de procesos/hilos
 $num = \binom{v}{2}$ 
 $filas\_ca = num * N$ 
 $c\_array\_alfabeto$  = matriz de  $num$  filas
for  $i := 0$  hasta  $num$  do
     $c\_array\_alfabeto[i] = \text{inverseGTP}(i, 2)$ 
end for
Global  $CA$  := se inicializa el CA de  $(filas\_ca$  filas) *  $(k$  columnas)

pthread_t  $*hilos$  := vector de longitud  $num\_hilos$ 
for  $i := 0$  hasta  $num\_hilos$  do
    Crea el proceso del  $i$ -ésimo hilo para el metodo CoveringArray
end for
Se limpia la memoria utilizada de los hilos con pthread_join

Se inicializa una barrera con pthread_barrier_init
Se inicializa un mutex con pthread_mutex_init
Se inicializa nuevamente la variable  $*hilos$  con la misma longitud
for  $i := 0$  hasta  $num\_hilos$  do
    Crea el proceso del  $i$ -ésimo hilo para el metodo Comparations
end for
Se limpia la memoria utilizada de los hilos con pthread_join
Se destruye el mutex con pthread_mutex_destroy
Se destruye la barrera con pthread_barrier_destroy

```

---



### 3.3 Versión con paralela (MPI)

Para la implementación del algoritmo paralelo con MPI también se usaron los mismos pasos esenciales para la fabricación de un programa paralelo: **Particionamiento, Comunicación, Aglomeración, Maqueo** .

---

**Algorithm 7** CoveringArray(*local\_CA*, *count*, *inicio*, *id*)

---

```

Require  $v$  := numero de simbolos (alfabeto)
Require  $k$  := numero de columnas del CA
Require  $N$  :=  $N\_Value(k)$ 
Require  $m$  := longitud de GTP
 $fila\_global$  := inicio
 $c = 0$ 
for  $i := 0$  hasta count do
   $c := 0$ 
  for  $j := 1$  hasta num do
    if  $fila\_global \geq N\_value(k)*j$  then
       $c := c + 1$ 
    end if
  end for
  for  $i := 0$  hasta  $k$  do
    Se inicializa la submatriz local_CA con los valores de  $c\_array\_alfabeto[c][0]$ 
     $*gtp := inverseGTP(i, m)$ 
     $f := fila\_global \% N$ 
    for  $x := 0$  hasta  $m$  do
      if  $gtp_x + 1 == f$  then
         $local\_CA_{i,j} = c\_array\_alfabeto[c][1]$ 
      end if
    end for
  end for
   $fila\_global = fila\_global + 1$ 
end for

```

---

El **Algoritmo 7** muestra un ligero cambio en comparación del algoritmo secuencial descrito en el **Algoritmo 2**. En este caso la función *CoveringArray* recibe tres parámetros: *local\_CA* corresponde a un pedazo de la matriz que forma el CA completo, la cual contiene el mismo numero de  $k$  columnas, pero difiere en el numero de filas, ya que estas se dividen dependiendo el numero de procesos utilizados. El objetivo del método es el mismo ya descrito, donde cada proceso se encarga de rellenar la submatriz *local\_CA* que le fue asignada como parámetro.

**Algorithm 8** Comparations(CA, tareas, inicio, id)

---

```

Global  $v$  := numero de simbolos (alfabeto)
Global  $n$  := numero de filas del CA
Global  $k$  := numero de columnas del CA
Global  $m$  := longitud de GTP
Global  $t$  := fuerza
Global  $gtps$  := combinaciones de columnas  $\binom{k}{t}$  a comparar del CA
fin := inicio + tareas

exist := 0
for  $i$  := inicio hasta fin do
   $exist$  := 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
    break
  end if
end for
return  $exist$ 

```

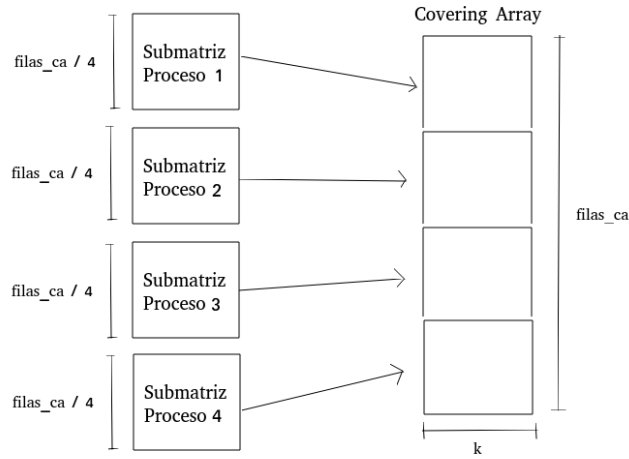
---

El **Algoritmo 8** describe el segundo método con implementación paralela también utilizado para la versión de *pthread*s, realizando el mismo procedimiento descrito. Lo que hace diferencia ésta implementación con la de *pthread*s es que en este método se le pasan directamente los parámetros de *CA*, que es el covering array ya generado con el primer método, que es usado en esta función ya que realiza la verificación de un *CA* correcto en un rango de filas de acuerdo a las tareas de cada proceso. Como la matriz *CA* es dividida en procesos, puede que la submatriz que trabaja un proceso sea valida, pero también puede que la de otro proceso no lo sea, por lo que al final se requiere juntar los resultados de cada proceso y en caso de que todos los resultados sean exitosos, significa que el *CA* completo es correcto, en caso de que al menos un proceso invalide una submatriz del *CA*, basta con ese para definir que el *CA* completo no es válido.

### Implementación de funciones de comunicación de MPI

En esta versión paralela del algoritmo de Verificación de Covering Arrays, aplicado con MPI se hicieron uso de alguna de las funciones de comunicación colectiva. Ya que el algoritmo trabaja principalmente con *CAs* utiliza una representación con matrices bidimensionales. El trabajo paralelo consta que cada uno de los procesos que se utilizan se les asignara una submatriz de *CA*. Primero

se debe generar la matriz con el método *CoveringArray* ya descrito, el cual se genera de forma paralela ya que cada proceso genera una parte del CA, para al final juntarlas. Para el proceso de juntar cada submatriz a una matriz global se utiliza el método **MPI\_Gatherv**. La **Figura 2** muestra el proceso de captura de cada submatriz para formar el CA final. Una vez que se tiene el CA completo, ahora falta hacer la verificación, el cual es el trabajo de la función *Comparations* que también se implemento en paralelo. Al igual, a cada proceso se le asigno un pedazo del CA para verificar que exista el alfabeto dado en todas las combinaciones de las columnas de cada submatriz del CA que le toca a cada proceso. La función *Comparations* ya descrita, devuelve un valor entero indicando los valores existentes. Al final estos valores devueltos a cada proceso deben ser sumados y su resultado debe ser igual a todas las combinaciones del alfabeto el cual es  $v^t$ . Para sumar los valores de cada proceso fue utilizada la función **MPI\_Reduce** la cual acumula los valores a una variable global.



**Fig. 2.** La figura muestra un ejemplo de lo que hace la función **MPI\_Gatherv** con 4 procesos los cuales generan una parte del Covering Array y después recolecta cada uno de ellos en bloques de  $filas\_ca/4$  obteniendo finalmente el CA completo.

### Metodo Main con MPI

A continuación se muestra el llamado de las funciones en paralelo en el método main descrito en el **Algoritmo 9** utilizando las funciones de MPI: **MPI\_Gather**, **MPI\_Reduce**, **MPI\_Barrier** antes descritos.

---

**Algorithm 9** Main()

---

**Global**  $v$  := numero de simbolos (alfabeto)  
**Global**  $k$  := numero de columnas del CA  
**Global**  $N$  :=  $N\_Value(k)$   
**Global**  $m$  := longitud de GTP  
**Global**  $procesos$  := numero de procesos

$id$  := identificador de proceso  
**MPI.Init**(NULL, NULL);  
**MPI.Comm\_size**(MPI\_COMM\_WORLD, & $procesos$ );  
**MPI.Comm\_rank**(MPI\_COMM\_WORLD, & $id$ );

$num = \binom{v}{2}$   
 $filas\_ca = num * N$   
 $c\_array\_alfabeto$  = matriz de  $num$  filas  
**for**  $i$  := 0 hasta  $num$  **do**  
      $c\_array\_alfabeto[i]$  = *inverseGTP*( $i$ , 2)  
**end for**

**if** proceso 0 **then**  
     Se inicializa el CA global de ( $filas\_ca * k$ )  
**end if**  
 $tareas[procesos]$  := filas de la submatriz que tocan por proceso  
 $displs[procesos]$  := filas de inicio por proceso de acuerdo al CA global

$*local\_ca$  := Se inicializa la submatriz de  $counts[id]*k$  de cada proceso  
*CoveringArray*( $*local\_ca$ ,  $tareas[id]$ ,  $displs[id]$ ,  $id$ )  
 Se llama a **MPI.Gatherv** para unir las submatrices  $*local\_ca$  a la matriz global CA  
**MPI.Barrier**(MPI\_COMM\_WORLD)

Se usa **MPI.Bcast** para compartir el CA generado a todos los procesos

$c := \binom{k}{t}$  combinaciones de columnas del CA a comparar  
 $gtps$  := vector de  $c$  posiciones  
**for**  $i$  := 0 hasta  $c$  **do**  
      $gtps[i]$  := *inverseGTP*( $i$ ,  $t$ )  
**end for**  
 $tareas[procesos]$  :=  $\frac{c}{procesos}$  combinaciones a verificar por proceso  
 $displs[procesos]$  := puntos de inicio de cada proceso  
 $exist\_all := 0$   
 $local\_exist$  := *Comparations*(CA,  $tareas[id]$ ,  $displs[id]$ ,  $id$ )  
 Se usa **MPI.Reduce** para sumar todos los  $local\_exist$  de cada proceso y almacenarlos en  $exist\_all$

**if** proceso 0 **then**  
     **if**  $exist\_all == procesos * v^t$  **then**  
         CA correcto  
     **else**  
         CA incorrecto  
     **end if**  
**end if**

**MPI.Finalize**

---

## 4 Versión Paralela (OpenMP)

OpenMP es un API de programación paralela de memoria compartida. A diferencia de la versión con Pthreads, en éste es mas sencillo codificar comportamientos paralelos ya que OpenMP simplemente indica que bloque debe ejecutarse en paralelo. Para esta versión se implementaron al igual que Pthreads, los pasos para la fabricación de un programa paralelo: **Particionamiento, Comunicación, Aglomeración, Mapeo**. El paso de Particionamiento sigue manteniendo las mismas partes del programa a paralelizar, que es la tarea de construcción del CA y la tarea de verificación o comparación. A continuación se muestra el algoritmo paralelo con OpenMP para la construcción de un CA.

---

**Algorithm 10** CoveringArray(*CA, counts, starts*)

---

```

Require  $v$  := numero de simbolos (alfabeto)
Require  $k$  := numero de columnas del CA
Require  $N$  :=  $N\_Value(k)$ 
Require  $m$  := longitud de GTP

 $id\_thread$  := omp_get_thread_num()
 $procesos$  := omp_get_num_threads()
 $tareas$  :=  $counts[id\_thread]$ 
 $inicio$  :=  $starts[id\_thread]$ 
 $fin$  :=  $inicio + tareas$ 
 $c = 0$ 
for  $i$  :=  $inicio$  hasta  $count$  do
   $c := 0$ 
  for  $j$  := 1 hasta  $num$  do
    if  $i \geq N\_value(k)*j$  then
       $c := c + 1$ 
    end if
  end for
  for  $i$  := 0 hasta  $k$  do
    Se inicializa la submatriz  $local\_CA$  con los valores de  $c\_array\_alfabeto[c][0]$ 
     $*gtp$  := inverseGTP( $i, m$ )
     $f := i \% N$ 
    for  $x$  := 0 hasta  $m$  do
      if  $gtp_x + 1 == f$  then
         $CA_{i,j} = c\_array\_alfabeto[c][1]$ 
      end if
    end for
  end for
end for

```

---

El **Algoritmo 10** muestra el método de construcción de CA ahora para la versión con OpenMP. Esta implementación es casi idéntica a la de Pthreads, ya que cada uno de los hilos se encarga de llenar partes de una matriz global (CA completo) correspondiendo a cada uno un determinado numero de filas para procesar pero siempre accediendo a la matriz global al mismo tiempo cada hilo, pero procesando diferentes secciones de ella. Debido a esto no es necesario recurrir a secciones criticas ya que los hilos no acuden una variable al mismo tiempo. En este caso, el método *CoveringArray* obtiene tres parámetros, *CA* es la matriz global inicializada en el método main la cual sera accedida por todos los hilos, *counts* es un vector de longitud cuantos hilos existan, el cual cada indice contiene las tareas que le corresponden a cada hilo, y *starts* es un vector de longitud igual a *counts*, pero este contiene los índices de filas de CA, y en cada índice del vector guarda la fila inicial de cada submatriz que le corresponde al hilo. Después se obtiene el id del hilo actual con el metodo **omp\_get\_thread\_num()** que se almacena en *id.thread*, y el numero de hilos totales que se obtienen con el método **omp\_get\_num\_threads()** almacenado en *procesos*. Después se selecciona el numero de tareas correspondientes a cada hilo obtenidas con el vector *counts* en la posicion *id.thread*. Al igual se obtiene el punto inicial en la misma posicion pero ahora del vector *starts*. Y de esta forma se obtiene el rango de proceso en la matriz global CA para cada hilo. La porcion restante del código trabaja de la misma manera que la versión de MPI a partir del primer ciclo **for**.

---

**Algorithm 11** Comparations(CA, counts, starts)

---

**Global** *v* := numero de simbolos (alfabeto)  
**Global** *n* := numero de filas del CA  
**Global** *k* := numero de columnas del CA  
**Global** *m* := longitud de GTP  
**Global** *t* := fuerza  
**Global** *gtps* := combinaciones de columnas  $\binom{k}{t}$  a comparar del CA

*id.thread* := **omp\_get\_thread\_num()**  
*procesos* := **omp\_get\_num\_threads()**  
*tareas* := *counts*[*id.thread*]  
*inicio* := *starts*[*id.thread*]  
*fin* := *inicio* + *tareas*  
exist := 0

---

---

```

for i := inicio hasta fin do
  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
    break
  end if
end for
return exist

```

---

En el **Algoritmo 11** muestra el segundo método a paralelizar el cual es el de verificar las columnas del CA para determinar que sea válido, esto en el método de *Comparations*. Al igual que en el metodo anterior, tambien se obtienen los mismos parametros de entrada para obtener las tareas y los puntos de partida que le corresponden a cada hilo y el codigo restante es el mismo procedimiento explicado en el **Algoritmo 8** de MPI.

### Metodo Main con OpenMP

A continuación se muestra el llamado de las funciones en paralelo en el método main descrito en el **Algoritmo 12** utilizando OpenMP:

---

#### Algorithm 12 Main()

---

```

Global v := numero de simbolos (alfabeto)
Global k := numero de columnas del CA
Global N := N_Value(k)
Global m := longitud de GTP
Global procesos := numero de procesos

```

```

num =  $\binom{v}{2}$ 
filas_ca = num * N
c_array_alfabeto = matriz de num filas
for i := 0 hasta num do
  c_array_alfabeto[i] = inverseGTP(i, 2)
end for

```

```

Se inicializa el CA global de (filas_ca * k)
counts[procesos] := filas de la submatriz que tocan por proceso
starts[procesos] := filas de inicio por proceso de acuerdo al CA global

```

---

---

```

#pragma omp parallel num_threads(procesos)
  CoveringArray(CA, counts, starts)

   $c := \binom{k}{t}$  combinaciones de columnas del CA a comparar
  gtps := vector de  $c$  posiciones
  for i := 0 hasta  $c$  do
    gtps[i] := inverseGTP(i, t)
  end for
  tareas[procesos] :=  $\frac{c}{procesos}$  combinaciones a verificar por proceso
  displs[procesos] := puntos de inicio de cada proceso
  exist_all := 0

  #pragma omp parallel num_threads(procesos)
    reduction(+:exist_all)
  exist_all := Comparations(CA, counts, starts)

  if exist_all == procesos *  $v^t$  then
    CA correcto
  else
    CA incorrecto
  end if

```

---

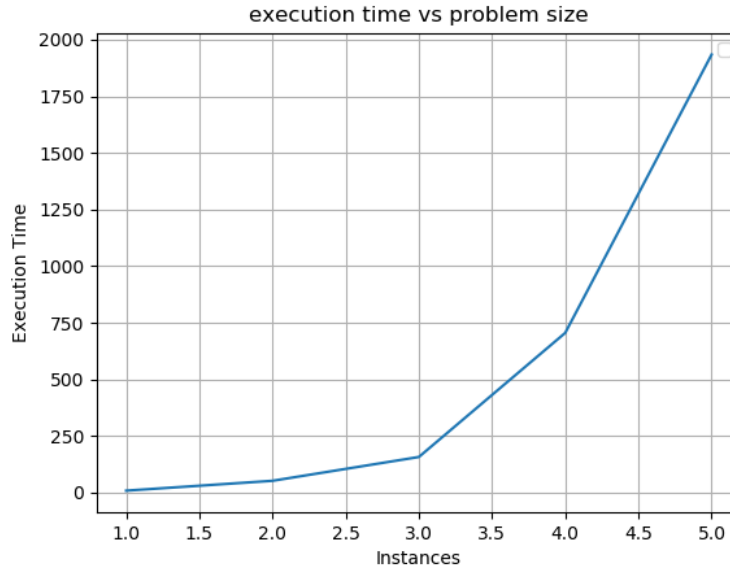
Primero, como en los algoritmos anteriores, se deben dar como entrada los valores globales para el problema, que ya han sido descritos, dentro de ellos también se da el número de hilos/procesos que se requieren. Para el método de construcción de CA primero se calculan las combinaciones de  $t$  símbolos para generar un CA, y se almacenan en el vector *c.array\_alfabeto*. De acuerdo al total de combinaciones, cada una genera un CA de  $N$  filas por lo que el CA final sería de  $num * N$  filas, donde  $num$  es el número de combinaciones y el resultado se almacena en *filas\_ca*. Después se debe inicializar la matriz CA que almacena las submatrices de  $N$  filas, cada una por cada combinación de símbolos, por lo que sus dimensiones serían de  $filas_ca * k$  donde  $k$  son las columnas. Seguido se generan los vectores *counts* y *starts*, que almacenan el número de tareas por cada hilo y los puntos de inicio de cada hilo respectivamente. Después se implementa un bloque paralelo de openmp, el cual se define con la instrucción *#pragma*, aquí se definen los números de procesos que se utilizan. Una vez puesto esta línea de código, la instrucción siguiente es la que se ejecuta de forma paralela, que en primer caso es el método *CoveringArray* con sus respectivos parámetros ya descritos en la sección anterior. Para el método de *Comparations* también se definieron las tareas de cada hilo, en este caso ya no se particionaban las filas del CA, sino que ahora se calcula el total de combinaciones de  $t$  columnas en un total de  $k$ , y a cada hilo le corresponde verificar tantas combinaciones como le toquen de un total de  $\binom{k}{t}$ . Para este método se usa la función *reduction(+:)* ya que el método *Comparations* regresa un entero que representa el total de combinaciones existentes y la tarea es sumar esos totales que calcula cada hilo.



## 5 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 número de filas,  $t$  es la fuerza (longitud de tupla),  $k$  es el número 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. 3.** 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.

### 5.1 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
```

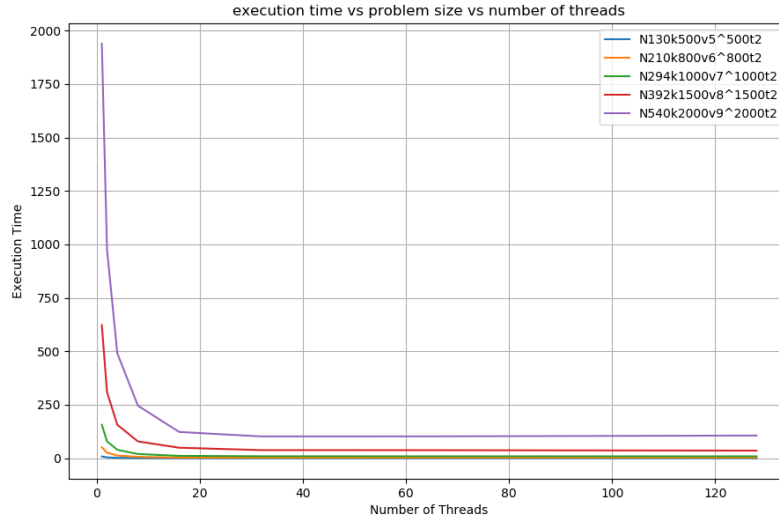
## 6 Resultados del algoritmo paralelo (Pthreads)

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.

### 6.1 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



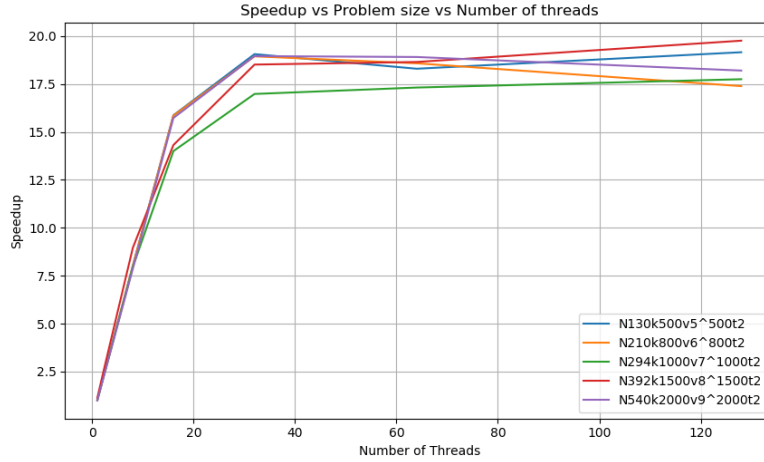
**Fig. 4.** La gráfica muestra la comparación con el tiempo de ejecución, el tamaño del problema y el número de hilos. En la tabla se puede ver que los tiempos de ejecución se parten por la mitad mientras el número de hilos aumenta, hasta llegar al caso de 32 hilos, los tiempos se mantienen casi iguales a partir de ese valor.

## 6.2 Aceleración

La siguiente tabla muestra los valores de la aceleración de cada una de las instancias usando diferentes valores de hilos. Para obtener la aceleración 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



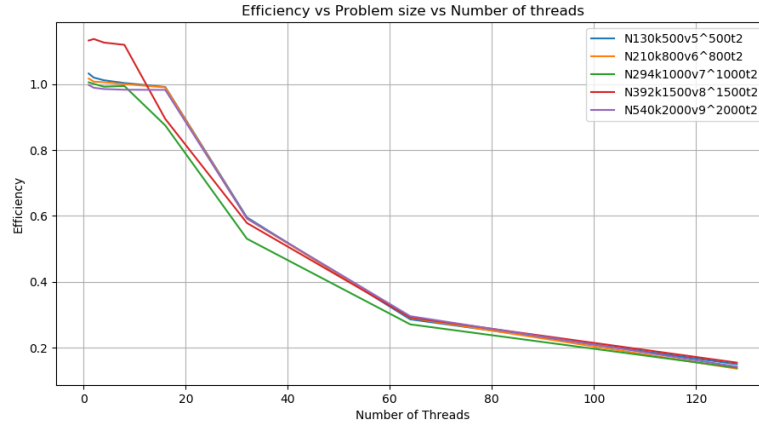
**Fig. 5.** La gráfica muestra los la comparación con la aceleración, el tamaño del problema y el numero de hilos. Se puede ver que la instancia que con poca frecuencia es  $CA(294;2;1000,7)$  (línea verde), y la instancia  $CA(392;2;1500;8)$  (línea roja) acelera mas rápido a partir de los 64 hilos de ejecución.

### 6.3 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 aceleración y el número 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. 6.** La gráfica muestra la comparación con la eficiencia, el tamaño del problema y el número de hilos. Se puede ver que cada instancia va tomando una eficiencia similar por cada valor de los hilos de procesamiento.

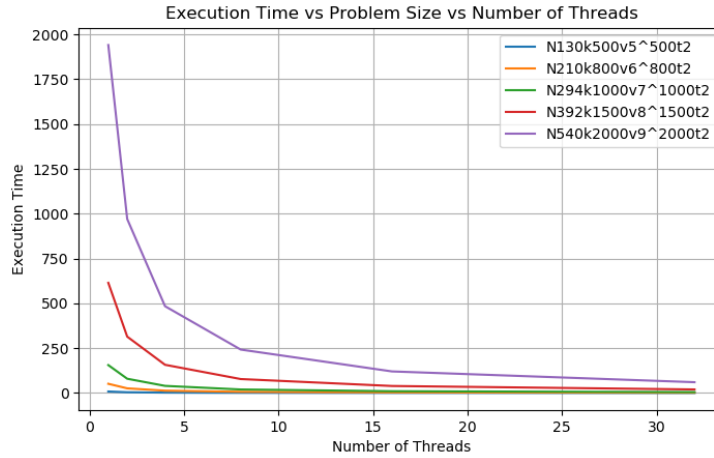
## 7 Resultados del algoritmo paralelo (MPI)

A continuación se muestran los resultados del algoritmo paralelo con implementación de MPI 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 procesos en cada ejecución.

### 7.1 Tiempo de ejecución

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

Instancia	1 proc	2 proc	4 proc	8 proc	16 proc	32 proc
CA(130;2;500;5)	8.38	4.22	2.14	1.11	0.6	0.34
CA(210;2;800;6)	51.35	25.91	12.97	6.55	3.36	1.77
CA(294;2;1000;7)	155.62	79.53	39.7	19.64	9.95	5.09
CA(392;2;1500;8)	614.43	314.40	157.34	77.81	39.13	19.79
CA(540;2;2000;9)	1941.15	970.5	484.1	242.3	120.3	60.25



**Fig. 7.** La gráfica muestra la comparación con el tiempo de ejecución, el tamaño del problema y el número de procesos. En la tabla se puede ver que los tiempos de ejecución se parten por la mitad mientras el número de procesos aumenta para cada una de las instancias.

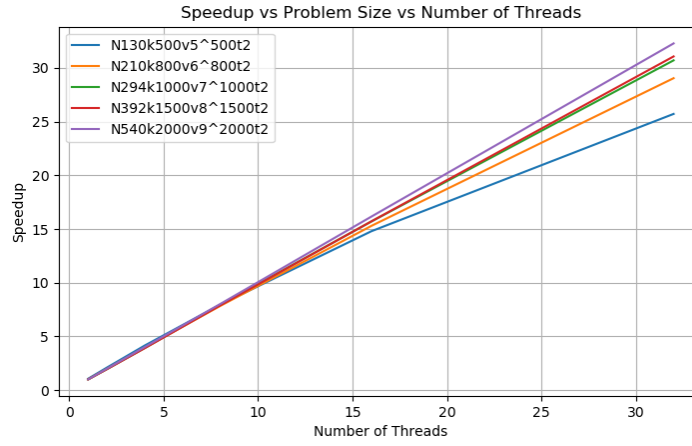
### 7.2 Aceleración

La siguiente tabla muestra los valores de la aceleración de cada una de las instancias usando diferentes valores de hilos. Para obtener la aceleración 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)	1.06	2.10	4.1	7.96	14.7	25.72		
CA(210;2;800;6)	1.00	1.98	3.96	7.8	15.2	29.04		
CA(294;2;1000;7)	1.00	1.96	3.93	7.9	15.7	30.71		
CA(392;2;1500;8)	1.00	1.9	3.9	7.9	15.7	31.07		
CA(540;2;2000;9)	1.00	2.00	4.01	8.02	16.16	32.28		



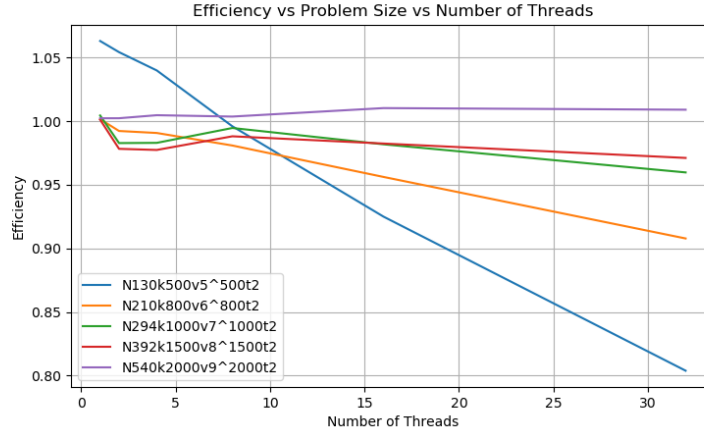
**Fig. 8.** La gráfica muestra la comparación con la aceleración, el tamaño del problema y el número de hilos. Se puede ver que la instancia que acelera con poca frecuencia es  $CA(130;2;500;5)$  (línea verde), y la instancia  $CA(540;2;2000;9)$  (línea roja) acelera más rápido a partir de los 64 hilos de ejecución.

### 7.3 Eficiencia

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

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

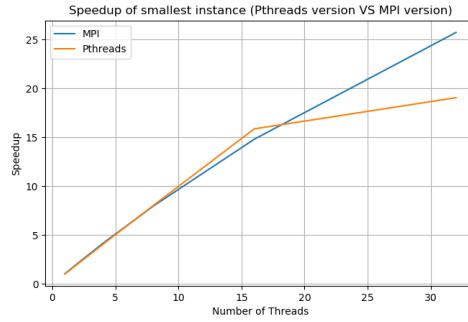
Instancia	1 hilo	2 hilos	4 hilos	8 hilos	16 hilos	32 hilos	64 hilos	32 hilos
CA(130;2;500;5)	1.06	1.05	1.03	0.99	0.9	0.8		
CA(210;2;800;6)	1.00	0.99	0.99	0.98	0.95	0.90		
CA(294;2;1000;7)	1.00	0.98	0.98	0.99	0.98	0.95		
CA(392;2;1500;8)	1.00	0.97	0.97	0.98	0.98	0.97		
CA(540;2;2000;9)	1.00	1.00	1.00	1.00	1.0	1.008		



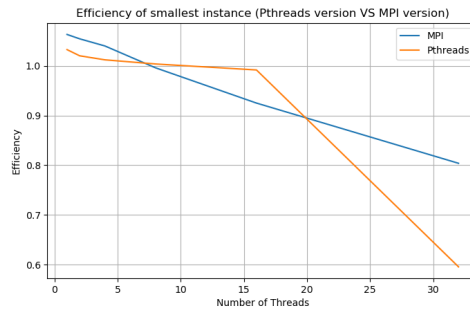
**Fig. 9.** La gráfica muestra los la comparación con la eficiencia, el tamaño del problema y el numero de hilos. Se puede ver que cada instancia va tomando una eficiencia similar por cada valor de los hilos de procesamiento.



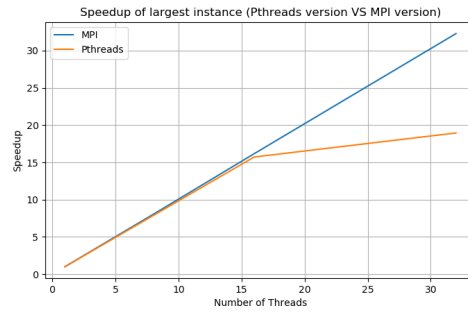
#### 7.4 Comparación de MPI con Pthreads



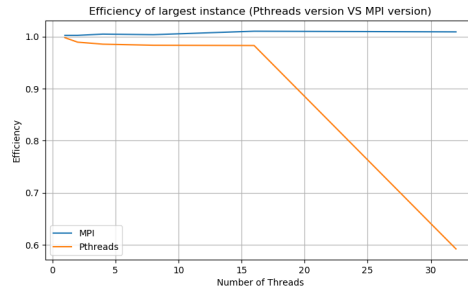
**Fig. 10.** Aquí se compara la velocidad de la instancia  $CA(130;2;500;5)$  en la versión paralela de Pthreads (línea naranja) y MPI (línea azul) donde se puede ver que la implementación mpi tiene una mayor aceleración



**Fig. 11.** Aquí se compara la eficiencia de la instancia  $CA(130;2;500;5)$  en la versión paralela de Pthreads (línea naranja) y MPI (línea azul) donde se puede ver que la implementación Pthreads es menos eficiente que MPI



**Fig. 12.** Aquí se compara la velocidad de la instancia CA(540;2;2000;9) en la versión paralela de Pthreads(linea naranja) y MPI(linea azul) donde se puede ver que la implementación mpi tiene una mayor aceleración



**Fig. 13.** Aquí se compara la eficiencia de la instancia CA(130;2;500;5) en la versión paralela de Pthreads(linea naranja) y MPI(linea azul) donde se puede ver que la implementación Pthreads es mucho menos eficiente que MPI

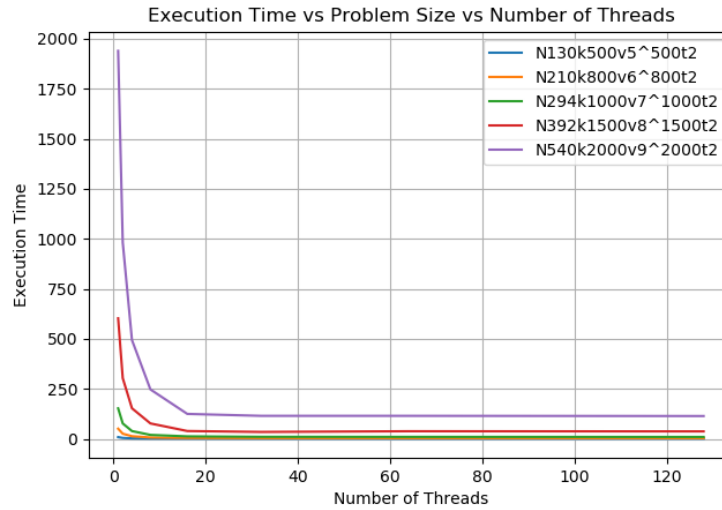
## 8 Resultados del algoritmo paralelo (OpenMP)

A continuación se muestran los resultados del algoritmo paralelo con implementación de OpenMP 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 procesos en cada ejecución.

### 8.1 Tiempo de ejecución

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

Instancia	1 proc	2 proc	4 proc	8 proc	16 proc	32 proc	64 proc	128 proc
CA(130;2;500;5)	8.32	4.32	2.27	1.38	1.08	0.91	0.59	1.18
CA(210;2;800;6)	50.45	25.35	12.83	6.54	4.22	3.29	3.18	3.11
CA(294;2;1000;7)	152.25	76.58	38.51	19.41	12.07	9.81	9.46	9.13
CA(392;2;1500;8)	602.52	303.18	152.47	76.79	38.81	34.24	37.79	37.01
CA(540;2;2000;9)	1939.88	981.72	493.61	246.83	124.39	114.76	114.76	113.76



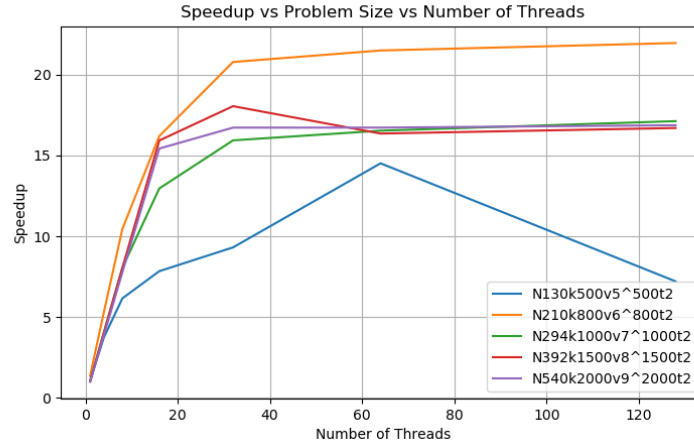
**Fig. 14.** La gráfica muestra la comparación con el tiempo de ejecución, el tamaño del problema y el número de procesos. En la tabla se puede apreciar que los tiempos se parten más o menos a la mitad mientras más procesos haya. Pero a partir de los 32 procesos el tiempo ya no disminuye favorablemente.

## 8.2 Aceleración

La siguiente tabla muestra los valores de la aceleración de cada una de las instancias usando diferentes valores de hilos. Para obtener la aceleración 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	128 hilos
CA(130;2;500;5)	1.02	1.97	3.75	6.16	7.84	9.31	14.52	7.21
CA(210;2;800;6)	1.35	2.70	5.33	10.45	16.20	20.79	21.50	21.96
CA(294;2;1000;7)	1.03	2.04	4.06	8.06	12.96	15.94	16.54	17.13
CA(392;2;1500;8)	1.03	2.04	4.05	8.05	15.93	18.05	16.36	16.70
CA(540;2;2000;9)	0.99	1.96	3.89	7.78	15.43	16.73	16.73	16.87



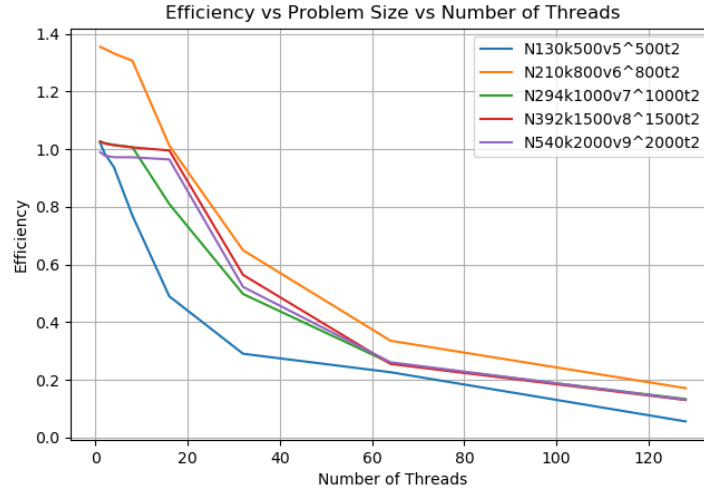
**Fig. 15.** La gráfica muestra los la comparación con la aceleración, el tamaño del problema y el numero de hilos. En este caso, la instancia mas pequeña (linea azul) acelera favorablemente hasta los 32 procesos, pero con mas el ritmo es mas lento. Mientras que la instancia 2 (linea naranja) es la mas rápida mientras mas procesos utilice.

### 8.3 Eficiencia

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

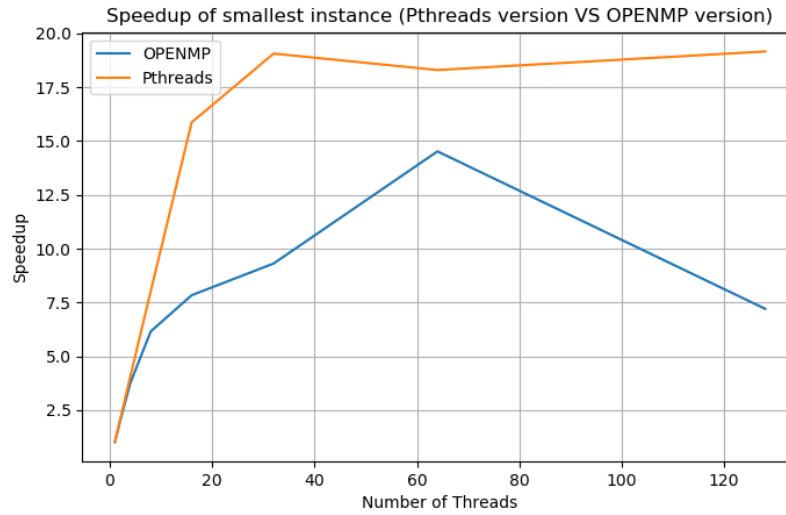
$$E = \frac{S}{p} = \frac{\frac{T_{serial}}{T_{paralelo}}}{p} = \frac{T_{serial}}{p * T_{paralelo}}$$

Instancia	1 hilo	2 hilos	4 hilos	8 hilos	16 hilos	32 hilos	64 hilos	128 hilos
CA(130;2;500;5)	1.02	0.98	0.94	0.77	0.49	0.29	0.23	0.06
CA(210;2;800;6)	1.35	1.35	1.33	1.31	1.01	0.65	0.34	0.17
CA(294;2;1000;7)	1.03	1.02	1.02	1.01	0.81	0.50	0.26	0.13
CA(392;2;1500;8)	1.03	1.02	1.01	1.01	1.00	0.56	0.26	0.13
CA(540;2;2000;9)	0.99	0.98	0.97	0.97	0.96	0.52	0.26	0.13

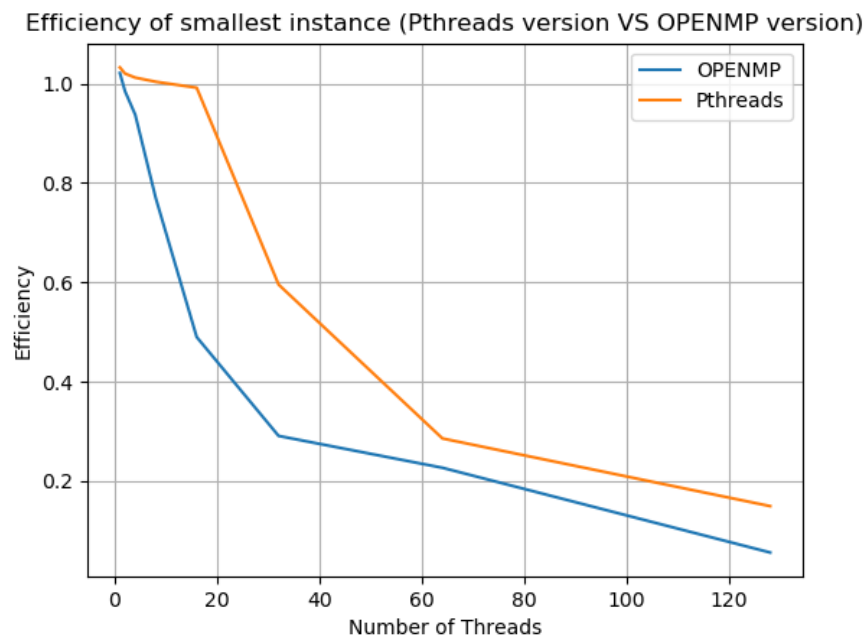


**Fig. 16.** La gráfica muestra la comparación con la eficiencia, el tamaño del problema y el número de hilos. Se puede ver que cada instancia va tomando una eficiencia similar por cada valor de los hilos de procesamiento.

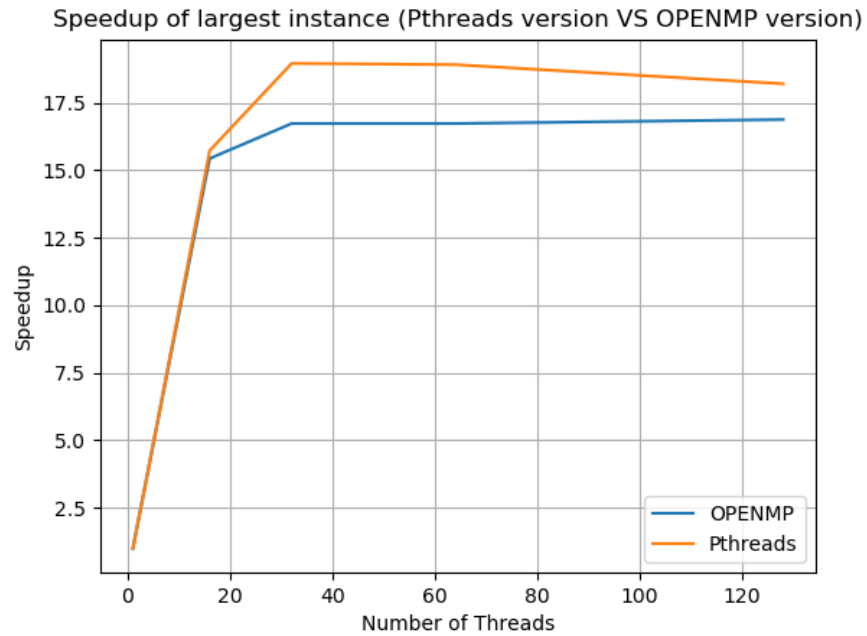
#### 8.4 Comparación de OpenMP con Pthreads



**Fig. 17.** Aquí se compara la velocidad de la instancia CA(130;2;500;5) en la versión paralela de Pthreads (línea naranja) y OpenMP (línea azul). En este caso la versión con OpenMP es más lenta a mayor número de procesos, por lo que es mejor Pthreads

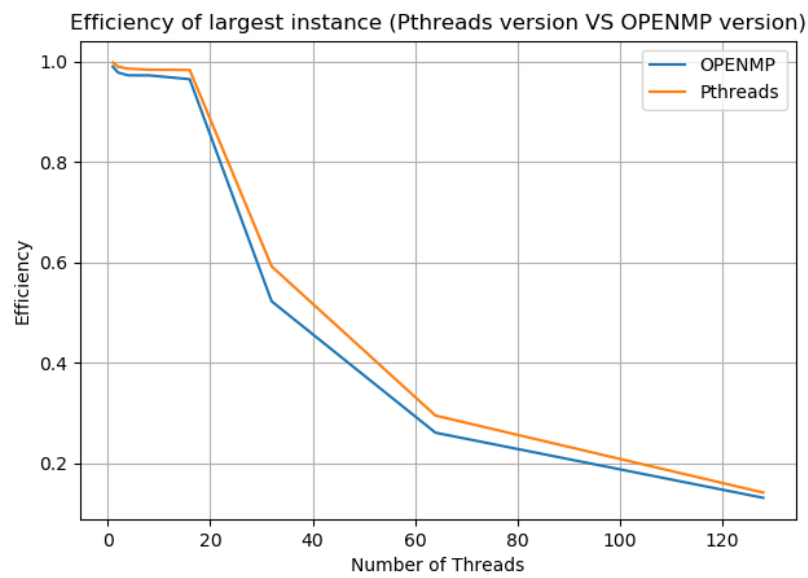


**Fig. 18.** Aquí se compara la eficiencia de la instancia CA(130;2;500;5) en la versión paralela de Pthreads(linea naranja) y OpenMP(linea azul) donde se puede ver que la implementación Pthreads es mas eficiente que OpenMP



**Fig. 19.** Aquí se compara la velocidad de la instancia CA(540;2;2000;9) en la versión paralela de Pthreads(linea naranja) y OpenMP(linea azul) donde se puede ver que la implementación Pthreads tiene una mayor aceleración hasta 32 procesos y después empieza a disminuir levemente pero aun así OpenMP es mas lento.





**Fig. 20.** Aquí se compara la eficiencia de la instancia CA(130;2;500;5) en la versión paralela de Pthreads (línea naranja) y OpenMP (línea azul) donde se puede ver que la implementación OpenMP es un poco menos eficiente que Pthreads a mayor procesos utilizados

## 9 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.