


# Submit V

Juan Diego Collazos y Juan Sebastián Garizao

5 de octubre de 2025

## 1. Challenges in Parallel Problem Decomposition

Los programas diseñados para la solución de los problemas enunciados se encuentran aquí:

 `submit-v/` in GitHub

## 2. The component labeling problem

### 2.1. Model PCAM

Para el diseño de este algoritmo se seguirá el enfoque del modelo PCAM (Partitioning, Communication, Agglomeration, Mapping)

#### 2.1.1. Partitioning (Descomposición)

El objetivo de la etapa de descomposición dentro del modelo PCAM es identificar las unidades fundamentales de trabajo que pueden ser ejecutadas de manera concurrente, preservando la estructura lógica del problema original. En el contexto del etiquetado de componentes conexos en una imagen binaria, el problema consiste en propagar etiquetas entre píxeles adyacentes con valor uno, hasta que todos los elementos pertenecientes a una misma región conexa compartan una misma etiqueta.

Una primera observación es que el procesamiento de los píxeles con valor cero es irrelevante para el cómputo, por lo que la descomposición puede centrarse exclusivamente en los píxeles activos. De este modo, el espacio de trabajo se restringe al conjunto:

$$L = \{(i, j) \mid M[i][j] = 1\}$$

donde  $M$  es la matriz de entrada de dimensiones  $n \times m$ . Cada elemento de  $L$  representa una unidad de trabajo independiente: la operación de actualización de la etiqueta de un píxel a partir de sus vecinos inmediatos.

El siguiente paso es agrupar estas unidades en subconjuntos que puedan procesarse de forma paralela. Existen varias estrategias de descomposición posibles:

- **Descomposición espacial:** dividir la matriz en bloques o subregiones contiguas, asignando cada bloque a un proceso o hilo.
- **Descomposición por elementos activos:** considerar únicamente los píxeles con valor uno y distribuirlos en conjuntos de trabajo de tamaño similar.

El segundo enfoque resulta más eficiente en este caso, ya que la densidad de píxeles activos puede variar significativamente dentro de la imagen. Al basar la descomposición en los elementos activos, se evita asignar regiones con gran cantidad de ceros, mejorando el equilibrio de carga entre las unidades de procesamiento.

Cada unidad de trabajo ejecuta de manera independiente una operación de actualización local, en la cual un píxel adopta la etiqueta mínima entre la suya y la de sus vecinos directos:

$$M'(i, j) = \min\{ M(i, j), M(i - 1, j), M(i + 1, j), M(i, j - 1), M(i, j + 1) \}$$

Este tipo de operación es inherentemente local y depende únicamente de la vecindad inmediata, lo que facilita una descomposición paralela con bajo acoplamiento entre tareas.

### 2.1.2. Communication (Comunicación)

En esta fase se define el flujo de información necesario entre las unidades de cómputo para mantener la coherencia global del proceso de etiquetado. Aunque la descomposición divide el trabajo entre píxeles o bloques independientes, las conexiones entre componentes adyacentes requieren intercambio de información sobre las etiquetas vecinas.

El propósito de la comunicación es garantizar que los píxeles pertenecientes a una misma componente conexa —pero asignados a diferentes hilos— converjan hacia una etiqueta común. Esto implica que, tras cada fase de actualización local, los hilos deben compartir los cambios ocurridos en las regiones limítrofes o en los píxeles fronterizos con otros bloques.

Desde una perspectiva teórica, la comunicación puede clasificarse como:

- **Local:** intercambio de etiquetas entre píxeles vecinos dentro de un mismo bloque o región asignada a un hilo.
- **Global:** propagación de actualizaciones entre bloques adyacentes o entre diferentes hilos, necesaria para asegurar la consistencia del etiquetado en toda la matriz.

El modelo de comunicación adoptado es *síncrono*, en el sentido de que los hilos avanzan de forma coordinada por iteraciones o fases globales. Después de cada fase, se produce un punto de sincronización en el que se garantiza que todos los hilos han completado sus actualizaciones locales antes de continuar con la siguiente iteración. Este enfoque facilita la convergencia global del algoritmo y evita inconsistencias en la propagación de etiquetas.

### 2.1.3. Agglomeration (Agregación)

La aglomeración se realiza al agrupar múltiples píxeles activos dentro de una misma partición lineal, en lugar de tratar cada píxel como una tarea independiente. Es decir, en vez de asignar una tarea por cada elemento de la lista  $L = (i, j)M[i][j] = 1$ , se agrupan los píxeles en subconjuntos contiguos  $L_k$  que constituyen unidades de trabajo más grandes.

Esta forma de aglomeración reduce el número de tareas paralelas desde  $L$  hasta  $p$ , donde  $p$  es el número de hilos o procesadores disponibles. Cada hilo procesa un bloque completo de píxeles activos, realizando internamente todas las actualizaciones locales de etiquetas antes de sincronizarse con el resto.

La decisión de aplicar esta aglomeración lineal tiene varias justificaciones teóricas:

1. Minimiza la sobrecarga de coordinación, ya que se disminuye el número de entidades que requieren sincronización en cada iteración.

2. Mejora la localidad de datos, al mantener el procesamiento de píxeles consecutivos dentro del mismo hilo, lo que reduce la latencia por acceso a memoria.
3. Equilibra el paralelismo con la eficiencia, evitando la creación de miles de tareas extremadamente pequeñas que no compensarían el costo de gestión paralela.

La aglomeración adoptada transforma una descomposición extremadamente fina (por píxel) en una descomposición por bloques lineales de píxeles, manteniendo un grado de paralelismo adecuado, pero con una granularidad más eficiente para entornos de memoria compartida.


#### 2.1.4. Mapping (Mapeo)

En la etapa de mapeo se define cómo las unidades de trabajo agregadas se asignan a los recursos de procesamiento disponibles. En el diseño de este algoritmo, se adopta un mapeo estático de bloques: cada hilo del sistema se asocia de manera fija a uno de los subconjuntos  $L_k$  obtenidos en la fase de aglomeración. De este modo, cada procesador ejecuta de manera independiente las actualizaciones sobre su bloque de píxeles activos, sin reasignación de tareas durante la ejecución.

Esta decisión presenta varias ventajas teóricas:

1. Simplicidad y bajo costo de gestión: al realizar la asignación de tareas de forma estática, se evita la sobrecarga asociada a esquemas dinámicos o balanceo en tiempo de ejecución.
2. Determinismo en el rendimiento: dado que la carga se reparte equitativamente al inicio, el comportamiento del algoritmo resulta más predecible y fácil de analizar.
3. Compatibilidad con la arquitectura de memoria compartida: los hilos operan sobre regiones de datos accesibles en un mismo espacio de direcciones, lo que elimina la necesidad de comunicación explícita entre procesadores.

## 2.2. Implementación

 [labeling/](#) in GitHub

La implementación del algoritmo se realizó en **C++** utilizando la biblioteca `pthread.h` de **POSIX Threads**, lo cual permite explotar el paralelismo de memoria compartida descrito en el diseño PCAM. Cada hilo ejecuta un conjunto de operaciones sobre un subconjunto de píxeles activos, siguiendo una estructura estrechamente alineada con las fases teóricas del modelo.

### Relación con el diseño PCAM

**Partitioning.** En el código, la descomposición se materializa en la función `labeling()`. Durante la inicialización, se recorre la matriz binaria `mat` y se construye el vector `ones`, que almacena las coordenadas de todos los píxeles con valor uno:

```
vector<pair<int,int>> ones;
if (mat[i][j] == 1) ones.push_back({i, j});
```

Este vector representa la lista  $L$  descrita en el modelo teórico. Cada elemento de `ones` corresponde a una unidad de cómputo independiente, lo cual traduce la fase de partición del diseño conceptual a una estructura de datos concreta.

**Communication.** La comunicación entre hilos se maneja mediante memoria compartida, aprovechando el entorno de ejecución de POSIX Threads. Las matrices `current` y `next` son estructuras globales accesibles por todos los hilos. Cada hilo lee los valores de `current` y escribe en `next`, lo que constituye una forma de comunicación implícita entre procesos concurrentes. La variable booleana compartida `all_converged` actúa como mecanismo de sincronización lógica, indicando si alguna actualización produjo cambios:

```
if (mini != (*data->current)[u.first][u.second])
    (*data->all_converged) = false;
```

Este esquema refleja el modelo de sincronización por fases planteado en el diseño, donde todos los hilos deben completar una iteración antes de pasar a la siguiente.

**Agglomeration.** El agrupamiento se realiza al distribuir la lista de píxeles `ones` en bloques continuos, definidos por índices `start` y `end`. Cada bloque se encapsula en una estructura `tdata`, que agrupa los datos necesarios para el trabajo del hilo:

```
thr_data[i] = tdata(&current, &next, &all_converged, &ones, act, to);
```

Esta decisión reduce la sobrecarga de comunicación, ya que cada hilo trabaja sobre un rango de píxeles predefinido y sólo sincroniza al final de cada iteración. La clase `tdata` implementa el concepto de *tarea agregada* del modelo PCAM, al encapsular tanto los datos locales como los punteros compartidos necesarios para la operación.

**Mapping.** Finalmente, el mapeo entre tareas y recursos se realiza mediante la creación explícita de hilos POSIX en la función `update()`. Cada hilo se asocia a un bloque de datos, manteniendo un mapeo estático de carga:

```
pthread_create(&threads[i], NULL, improve_label, (void*) &thr_data[i]);
```

El número de hilos activos se limita al mínimo entre el número de píxeles y el parámetro `N_THREADS`, lo cual evita la creación innecesaria de hilos cuando el trabajo es escaso:

```
int num_threads = min(n, N_THREADS);
```

Esta decisión refleja el mapeo estático propuesto en la fase teórica, garantizando simplicidad y predictibilidad en el rendimiento.

## Estructura general del algoritmo

La función principal `labeling()` coordina el proceso de iteración y convergencia. Durante cada iteración:

1. Se llama a la función `update()` para distribuir el trabajo entre los hilos.
2. Cada hilo ejecuta `improve_label()`, calculando el mínimo de etiquetas entre el píxel actual y sus cuatro vecinos.
3. Tras la sincronización, las matrices `current` y `next` se intercambian.

El proceso continúa hasta que `all_converged` permanece verdadero, indicando que todas las etiquetas son estables.

### 3. Algorithms Performance I (Amdahl's law)

#### 1) Caso: $s = 0,01$ , $N = 61$

Dado que el 1 % no es paralelizable:

$$s = 0,01, \quad p = 1 - s = 0,99.$$

Aplicamos Amdahl:

$$T(61) = s + \frac{p}{61} = 0,01 + \frac{0,99}{61}.$$

Calculemos  $\frac{0,99}{61}$  con pasos:

$$\frac{0,99}{61} = 0,99 \div 61.$$

Podemos transformar  $0,99 = 99/100$ , entonces

$$\frac{0,99}{61} = \frac{99}{100} \cdot \frac{1}{61} = \frac{99}{6100}.$$

División decimal:

$$\frac{99}{6100} \approx 0,016229508196721313.$$

Sumando con  $s$ :

$$T(61) = 0,01 + 0,016229508196721313 = 0,026229508196721313.$$

El speedup es

$$S(61) = \frac{1}{T(61)} = \frac{1}{0,026229508196721313} \approx 38,132282.$$

**Respuesta:**  $\boxed{S(61) \approx 38,13}$ .

Aunque se usan 61 núcleos, el speedup está limitado por la fracción secuencial; 38.13 es mucho menor que 61.

#### 2) Caso: $s = 0,001$ (0.1 % secuencial), varias $N$

Ahora:

$$s = 0,001, \quad p = 0,999.$$

La fórmula general:

$$S(N) = \frac{1}{0,001 + \frac{0,999}{N}}.$$

##### a) $N = 30$

Calculemos  $\frac{0,999}{30}$ :

$$0,999 = \frac{999}{1000}, \quad \frac{0,999}{30} = \frac{999}{1000} \cdot \frac{1}{30} = \frac{999}{30000} = \frac{333}{10000} = 0,0333.$$

Sumamos con  $s$ :

$$T(30) = 0,001 + 0,0333 = 0,0343.$$

Speedup:

$$S(30) = \frac{1}{0,0343} = \frac{1}{343/10000} = \frac{10000}{343} \approx 29,154518.$$

$$\boxed{S(30) \approx 29,1545}.$$

**b)**  $N = 30,000$

$$\frac{0,999}{30000} = \frac{999}{30000000} = \frac{333}{10000000} = 0,0000333.$$

Sumando  $s$ :

$$T(30000) = 0,001 + 0,0000333 = 0,0010333.$$

Speedup:

$$S(30000) = \frac{1}{0,0010333} \approx 967,741935.$$

(Observación: como  $s = 0,001$ , el speedup máximo teórico cuando  $N \rightarrow \infty$  es  $1/s = 1000$ ; con  $N = 30\,000$  nos acercamos mucho:  $\approx 967,74$ .)

$$\boxed{S(30,000) \approx 967,74}.$$

**c)**  $N = 3,000,000$

$$\frac{0,999}{3,000,000} = \frac{999}{3,000,000,000} = \frac{333}{1,000,000,000} = 0,000000333.$$

Sumando  $s$ :

$$T(3,000,000) = 0,001 + 0,000000333 = 0,001000333.$$

Speedup:

$$S(3,000,000) = \frac{1}{0,001000333} \approx 999,667.$$

$$\boxed{S(3,000,000) \approx 999,667}.$$

### 3) Broadcast con overhead: modelos y óptimos

Planteamiento: además del término  $s + p/N$  añadimos un término de overhead que depende de  $n$  (número de núcleos). Usamos  $s = 0,001$  (0.1 %) como caso de referencia, y suponemos que el overhead se suma al término total de tiempo normalizado (esto es una modelación razonable para costos paralelos adicionales). Entonces el denominador es:

$$D(n) = s + \frac{p}{n} + \text{overhead}(n).$$

Buscamos  $n$  que *maximice* el speedup  $S(n) = 1/D(n)$ , equivalente a *minimizar*  $D(n)$  respecto a  $n$  (tratamos  $n$  como continua para obtener óptimos; después tomamos entero).

Dos implementaciones de broadcast:

1. Overhead lineal:  $\text{overhead}_1(n) = a \cdot n$  con  $a = 0,0001$ .
2. Overhead logarítmico:  $\text{overhead}_2(n) = b \cdot \log(n)$  con  $b = 0,0005$ .

### Caso A: overhead linear $an$

$$D_1(n) = s + \frac{1-s}{n} + an.$$

Derivada (en  $n$ , continua):

$$\frac{dD_1}{dn} = -\frac{1-s}{n^2} + a.$$

Punto crítico cuando se anula la derivada:

$$-\frac{1-s}{n^2} + a = 0 \quad \Rightarrow \quad a = \frac{1-s}{n^2}.$$

Despejando  $n$ :

$$n = \sqrt{\frac{1-s}{a}}.$$

Sustituyendo  $s = 0,001$ ,  $1-s = 0,999$ ,  $a = 0,0001$ :

$$n = \sqrt{\frac{0,999}{0,0001}} = \sqrt{9990} \approx 99,94998749 \approx 100.$$

Tomando  $n \approx 100$  núcleos obtenemos el óptimo (entero cercano).

Calculemos el speedup en  $n = 100$ :

$$\frac{1-s}{n} = \frac{0,999}{100} = 0,00999, \quad an = 0,0001 \cdot 100 = 0,01.$$

Denominador:

$$D_1(100) = 0,001 + 0,00999 + 0,01 = 0,02099.$$

Speedup:

$$S_1(100) = \frac{1}{0,02099} \approx 47,64.$$

Óptimo approx para overhead lineal:  $n \approx 100$ ,  $S \approx 47,6$ .

Interpretación: el overhead lineal crece tan rápido que añadir más núcleos después de  $\sim 100$  empeora el resultado.

### Caso B: overhead logarítmico $b \log(n)$

$$D_2(n) = s + \frac{1-s}{n} + b \log(n).$$

Derivada (asumiendo log como logaritmo natural  $\ln$ ; si fuera base 2 eso cambia por un factor constante):

$$\frac{dD_2}{dn} = -\frac{1-s}{n^2} + \frac{b}{n}.$$

Igualando a cero:

$$-\frac{1-s}{n^2} + \frac{b}{n} = 0 \quad \Rightarrow \quad \frac{b}{n} = \frac{1-s}{n^2}.$$

Multiplicando por  $n^2$ :

$$bn = 1-s \quad \Rightarrow \quad n = \frac{1-s}{b}.$$

Sustituyendo  $1 - s = 0,999$ ,  $b = 0,0005$ :

$$n = \frac{0,999}{0,0005} = 1998.$$

Así, con log natural el óptimo continuo es  $n \approx 1998$  núcleos.

Calculemos el speedup en  $n = 1998$  (usando  $\ln$ ):

$$\frac{1-s}{n} = \frac{0,999}{1998} = 0,0005 \quad (\text{exacto, ya que } 1998 \times 0,0005 = 0,999).$$

Calculamos  $\ln(1998) \approx 7,60090246$  (aprox.) y luego  $b \ln(n) = 0,0005 \times 7,60090246 \approx 0,00380045123$ . Denominador:

$$D_2(1998) = 0,001 + 0,0005 + 0,00380045123 = 0,00530045123.$$

Speedup:

$$S_2(1998) \approx \frac{1}{0,00530045123} \approx 188,676.$$

Óptimo approx con  $\ln$  :  $n \approx 1998$ ,  $S \approx 188,7$ .

#### 4) ¿Por qué construir sistemas con millones de núcleos a pesar de Amdahl?

Aunque Amdahl señala límites fuertes para el speedup de *un único programa* con fracción secuencial fija, existen múltiples razones por las que se diseñan y construyen supercomputadores gigantes:

- **Escalado débil (weak scaling) y throughput:** muchos problemas crecen con el tamaño de datos; al aumentar la cantidad de trabajo proporcionalmente al número de núcleos (weak scaling) se mantiene la eficiencia y se resuelven problemas mayores en tiempo razonable.
- **Aplicaciones embarrassingly parallel:** multitud de cargas de trabajo son triviales de paralelizar (simulaciones Monte Carlo, búsquedas independientes, renderizado), y se benefician linealmente de más núcleos.
- **Reducción del tiempo hasta la solución (time-to-solution):** para problemas que requieren más memoria que la que cabe en una sola máquina, o para reducir tiempos de análisis, usar muchos núcleos reduce el tiempo real a resultado.
- **Capacidad de memoria y I/O agregado:** grandes sistemas agregan mucha memoria y ancho de banda de I/O distribuido, necesario para grandes datasets.
- **Investigación y resiliencia:** permiten investigar nuevas arquitecturas, tolerancia a fallos, y ejecutar grandes trabajos científicos que simplemente no cabrían en sistemas pequeños.
- **Multiplicidad de trabajos:** centros HPC no ejecutan un solo trabajo; muchos usuarios ejecutan jobs concurrentes. Más núcleos significan mayor *throughput* del centro.
- **Modelos de programación y algoritmos escalables:** con nuevas técnicas (reducciones eficientes, algoritmos con menor sección crítica) se puede mitigar el efecto de la fracción secuencial.



## 5) Mínimo por reducción en árbol binario: work y depth

Algoritmo: Crear parejas de elementos y comparar; el menor sigue a la siguiente ronda. Si  $n$  no es potencia de dos, se pueden emparejar con cuidando “byes”, pero análisis asintótico es mismo.

**Work (trabajo total):** En la primera ronda se hacen  $\lfloor n/2 \rfloor$  comparaciones, en la segunda  $\lfloor n/4 \rfloor$ , etc. Sumando hasta quedar uno:

$$W = \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{4} \right\rfloor + \left\lfloor \frac{n}{8} \right\rfloor + \dots$$

Si  $n$  es potencia de 2, la suma exacta es:

$$W = \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots + 1 = n - 1.$$

Por tanto,  $\boxed{W = n - 1 = \Theta(n)}$  comparaciones totales.

**Depth (tiempo crítico paralelo):** La cantidad de rondas es el número de niveles del árbol binario hasta reducir a uno:

$$\text{depth} = \log_2 n$$

si  $n$  es potencia de 2; en general  $\lceil \log_2 n \rceil$ . Así,

$$\boxed{\text{depth} = \lceil \log_2 n \rceil = \Theta(\log n).}$$

**Comentarios sobre asignación de procesadores:** Si se dispone de  $n/2$  procesadores en la primera ronda, cada uno hace una comparación en paralelo; en rondas posteriores la cantidad de procesadores activos disminuye. El tiempo paralelo ideal es  $\Theta(\log n)$  con suficiente paralelismo; el trabajo total ( $\Theta(n)$ ) no aumenta respecto a la versión secuencial más que en una constante.

## 4. Algorithms Performance (Speedup)

### 1. Revisión de Implementaciones

- **Implementación A (Versión Secuencial):** Corresponde a la versión base del algoritmo, ejecutada de forma completamente secuencial. Cada celda del tablero se actualiza iterativamente sin ningún tipo de concurrencia, lo que implica un procesamiento estrictamente dependiente del recorrido de los bucles. Aunque garantiza una ejecución determinista y sencilla de depurar, su desempeño se ve limitado por la falta de aprovechamiento de los núcleos disponibles. Esta versión sirve como punto de referencia para evaluar el impacto de las estrategias de optimización y paralelización implementadas en las versiones posteriores.
- **Implementación B (Versión Multithreading):** Esta versión introduce paralelismo a nivel de hilos utilizando el módulo `threading` de Python. El tablero se divide en bloques de filas que son procesados de manera concurrente por múltiples hilos, cada uno ejecutando una porción independiente de la función de actualización. La estrategia de descomposición espacial permite aprovechar varios núcleos del procesador, reduciendo el tiempo total de ejecución respecto a la versión secuencial.

- **Implementación C (Versión Multiprocessing):** Implementa paralelismo real mediante la creación de múltiples procesos independientes, cada uno encargado de actualizar una subregión del tablero. La comunicación entre procesos se gestiona mediante arreglos compartidos (`RawArray`), lo que permite el acceso concurrente a la memoria sin duplicar datos. Esta descomposición del dominio elimina las restricciones impuestas por el *Global Interpreter Lock* (GIL) y permite aprovechar de forma efectiva varios núcleos del procesador. Esta versión representa la implementación más eficiente dentro del conjunto, demostrando el impacto directo de la paralelización en el rendimiento computacional.

## 2. Análisis de Rendimiento

Cuadro 1: Comparativa de rendimiento entre las implementaciones del Juego de la Vida.

Implementación	Configuración	Tiempo (s)	Speedup
A	Secuencial	12.84	1.00
B	Multithreading	5.21	2.46
C	Multiprocessing (4 procesos)	1.46	8.79
C	Multiprocessing (8 procesos)	0.79	16.25

Los resultados muestran una mejora progresiva en el rendimiento a medida que se incorporan mecanismos de paralelismo. La versión secuencial establece la línea base, mientras que la versión con *multithreading* logra un incremento moderado debido a las limitaciones del *GIL*. La implementación con *multiprocessing* evidencia un escalamiento casi lineal hasta ocho procesos, lo que confirma la efectividad del paralelismo a nivel de procesos para tareas computacionalmente intensivas.

## 3. Discusión y Reporte

El análisis comparativo de las tres versiones evidencia la influencia directa del grado de paralelismo sobre el rendimiento global del algoritmo. La **Implementación A** representa la versión base secuencial, caracterizada por su simplicidad y ejecución determinista, pero limitada al procesamiento en un solo núcleo. Esta falta de concurrencia impide aprovechar los recursos de hardware disponibles, resultando en el mayor tiempo de ejecución.

La **Implementación B**, basada en multithreading, introduce una primera forma de paralelismo lógico mediante la división del dominio en subregiones procesadas por hilos concurrentes. Sin embargo, el impacto del *Global Interpreter Lock* (GIL) en Python restringe el verdadero paralelismo a nivel de CPU, limitando las ganancias de rendimiento observadas. Aun así, esta versión constituye un avance conceptual hacia la ejecución concurrente y permite validar la correcta descomposición del problema.

Finalmente, la **Implementación C**, construida con el módulo `multiprocessing`, logra paralelismo real al ejecutar procesos independientes que operan sobre memoria compartida. Esto elimina las limitaciones del GIL y permite una escalabilidad casi lineal hasta ocho procesos, alcanzando una mejora de más de 16 veces respecto a la versión secuencial. No obstante, se observa una tendencia a la saturación del rendimiento a partir de cierto número de procesos, producto de la sobrecarga en la creación de procesos y el acceso compartido a memoria.

**Conclusión:** La paralelización resulta decisiva para mejorar el desempeño del algoritmo. Mientras que el multithreading demuestra las limitaciones del paralelismo lógico en Python, el modelo basado en multiprocessing aprovecha efectivamente los recursos multinúcleo, evidenciando una reducción significativa en el tiempo de ejecución y un escalamiento eficiente hasta cierto punto de saturación.

## Referencias

- [1] Ian T. Foster, “Designing and Building Parallel Programs (PCAM Model)”, Argonne National Laboratory, 1995.  
Disponible en: <https://www.mcs.anl.gov/~itf/dbpp/text/node14.html>