

# Trabajo Práctico N° 2

## Programación en memoria compartida

---

### Características del hardware y del software usados:

- *Hardware*
  - *Procesador: Intel (R) Core (TM) i5- 2310 CPU*
  - *Núcleos: 4*
  - *Arquitectura: x86\_64*
  - *Memoria RAM: 7.7 GB*
- *Software*
  - *Sistema Operativo: Ubuntu 22.04.1*
  - *Compilador: GCC version 11.3.0*
- *Cluster*
  - *Multicore (partición Blade) conformado por 16 nodos.*
  - *Cada nodo posee 8GB de RAM*
  - *2 procesadores Intel Xeon E5405 de cuatro 4 cores cada uno que operan a 2.0GHz.*

### Enunciado

**Dada la siguiente expresión:**

$$R = \frac{(MaxA \times MaxB - MinA \times MinB)}{PromA \times PromB} \times [A \times B] + [C \times B^T]$$

- **Donde A, B, C y R son matrices cuadradas de NxN con elementos de tipo double.**
- **MaxA, MinA y PromA son los valores máximo, mínimo y promedio de la matriz A, respectivamente.**
- **MaxB, MinB y PromB son los valores máximo, mínimo y promedio de la matriz B, respectivamente.**
- **$B^T$  Es la matriz transpuesta de B.**

**Desarrolle 3 algoritmos que computen la expresión dada:**

- 1. Algoritmo secuencial optimizado**
  - 2. Algoritmo paralelo empleando Pthreads**
-

---

### **3. Algoritmo paralelo empleando OpenMP**

***Mida el tiempo de ejecución de los algoritmos en el clúster remoto. Las pruebas deben considerar la variación del tamaño de problema ( $N=\{512, 1024, 2048, 4096\}$ ) y, en el caso de los algoritmos paralelos, también la cantidad de hilos ( $T=\{2,4,8\}$ ). Por último, recuerde aplicar las técnicas de programación y optimización vistas en clase.***

Para compilar los códigos desarrollados mantuvimos el uso del flag -O3 de optimización, de la entrega pasada, el cual nos dio buenos resultados.

En cuanto a los tamaños de bloque (BS) utilizamos el valor 64 que fue el que mejor resultado dio.

*1. Algoritmo secuencial optimizado:* Para el algoritmo secuencial usamos el código con sus tiempos promedios de ejecución de la entrega 1, pero con algunas modificaciones. Agregamos variables y pasamos a la variable bs como una constante simbólica con el valor 64.

Se usarán los nuevos tiempos de ejecución para comparar las siguientes alternativas y poder calcular el SpeedUp y la Eficiencia de las otras opciones.

*2. Algoritmo paralelo empleando Pthreads:*

Para realizar el algoritmo paralelo empleando Pthreads lo que hicimos fue identificar las regiones de código que podían ejecutarse en paralelo, es decir, que tarea puede ejecutar cada hilo de forma independiente a otro hilo y en los casos de necesitar recursos compartidos hacer los bloqueos necesarios. En nuestro caso los bloqueos a recursos compartidos los evitamos haciendo que cada hilo acceda a una posición diferente dentro de un vector global y así no generar condiciones de carrera ni demoras innecesarias.

En el código desarrollado lo que podemos ver es como cada hilo ejecuta de forma independiente el bloque "calcular" dónde se obtienen los valores máximos, mínimos y promedios de las matrices A y B. Para realizar estas operaciones cada hilo va guardando sus resultados dentro de unos vectores compartidos, de forma tal que el hilo 0 (id = 0) después itere sobre estos vectores y no sobre la matriz completa. También se paraleliza la función transpuesta.

---

Por otro lado, también se modificó “multiplicar\_matrices\_bloques” para que cada hilo solo recorra la parte que le corresponda y no toda la estructura.

### *3. Algoritmo paralelo empleando OpenMP*

Para realizar el algoritmo paralelo empleando OpenMP partimos de una versión secuencial donde identificamos las partes del código que pueden ejecutarse concurrentemente. Luego definimos una región paralela con la directiva `#pragma omp parallel private(..)`, donde se crea un equipo de hilos siguiendo el modelo Fork-Join.

El trabajo dentro de la región paralela se distribuye entre los hilos del equipo existente utilizando constructores de trabajo compartido, como la directiva `#pragma omp for`. Esta directiva divide las iteraciones de un bucle entre los hilos del equipo.

El código especifica la cláusula `schedule(static)` para definir cómo se distribuyen las iteraciones del bucle, en este caso utilizamos una asignación estática de bloques de iteraciones a los hilos.

El código utiliza barreras implícitas al final de cada bucle `#pragma omp for` (a menos que se use `nowait`). Se emplea la directiva `#pragma omp single`, la cual garantiza que el bloque de código asociado sea ejecutado por un único hilo dentro de la región paralela. Esta directiva tiene una barrera implícita al final, lo que asegura que todos los hilos esperen a que la sección `single` (donde se calcula el escalar) se complete antes de continuar.

La cláusula `nowait`, usada en el `#pragma omp for` para la multiplicación de matrices, evita la barrera implícita al final de ese bucle específico, permitiendo que los hilos que terminan esa tarea pasen a la siguiente si hay trabajo disponible, mejorando potencialmente el rendimiento.

---

### Tiempos de ejecución

Lo que podemos ver en las tablas son los tiempos en segundos de ejecución de los algoritmos en el cluster remoto blade. Donde **N** = tamaño matrices y **T** = cantidad hilos.

- Algoritmo secuencial optimizado

	<b><i>N = 512</i></b>	<b><i>N = 1024</i></b>	<b><i>N = 2048</i></b>	<b><i>N = 4096</i></b>
<b><i>T=1</i></b>	0.428034	3.501337	28.008172	226.518864

- Algoritmo paralelo empleando PThreads

	<b><i>N = 512</i></b>	<b><i>N = 1024</i></b>	<b><i>N = 2048</i></b>	<b><i>N = 4096</i></b>
<b><i>T=2</i></b>	0.23233	1.86795	14.81877	117.43627
<b><i>T=4</i></b>	0.11923	0.96951	7.50275	59.21544
<b><i>T=8</i></b>	0.06087	0.50905	3.87257	30.91349

- Algoritmo paralelo empleando OpenMP

	<b><i>N = 512</i></b>	<b><i>N = 1024</i></b>	<b><i>N = 2048</i></b>	<b><i>N = 4096</i></b>
<b><i>T=2</i></b>	0.218058	1.777731	13.970170	112.604606
<b><i>T=4</i></b>	0.112818	0.933191	7.249562	56.246380
<b><i>T=8</i></b>	0.060604	0.511298	3.871394	29.558310

---



### Speedup y eficiencia- Resultados obtenidos

- El speedup muestra el beneficio de usar procesamiento paralelo para resolver un problema comparado con la ejecución secuencial, es decir, muestra cuántas veces más rápida es una solución paralela respecto a su versión secuencial. Un Speedup lineal o perfecto sería igual al número de procesadores (p). Un Speedup menor a 1 indica que el algoritmo paralelo es más lento que el secuencial.

- La eficiencia complementa al Speedup y muestra qué tan bien se están utilizando los procesadores. La eficiencia ideal es 1 (o 100%), lo que ocurre en un sistema paralelo ideal donde el Speedup es igual a p. En la práctica, la eficiencia suele ser menor a 1 ( $0 < E_p(n) \leq 1$ ) debido a fuentes de overhead como el ocio, la interacción entre procesos y cómputo adicional. Generalmente, una mayor eficiencia se relaciona con un menor overhead.



A continuación están las tablas donde se ve que hay un speedup para cada par (tamaño de problema (N) y cantidad de hilos (T) → unidades de procesamiento). Lo mismo ocurre para la eficiencia, no hay un único valor de eficiencia sino que depende del par.

- Algoritmo paralelo empleando Pthreads - Speedup →  $T_{sec} / T_p$



<i>T</i>	<i>N = 512</i>	<i>N = 1024</i>	<i>N = 2048</i>	<i>N = 4096</i>
<i>2</i>	1.84234	1.87442	1.89005	1.92887
<i>4</i>	3.58993	3.61146	3.73306	3.82533
<i>8</i>	7.03251	6.87823	7.23245	7.32753

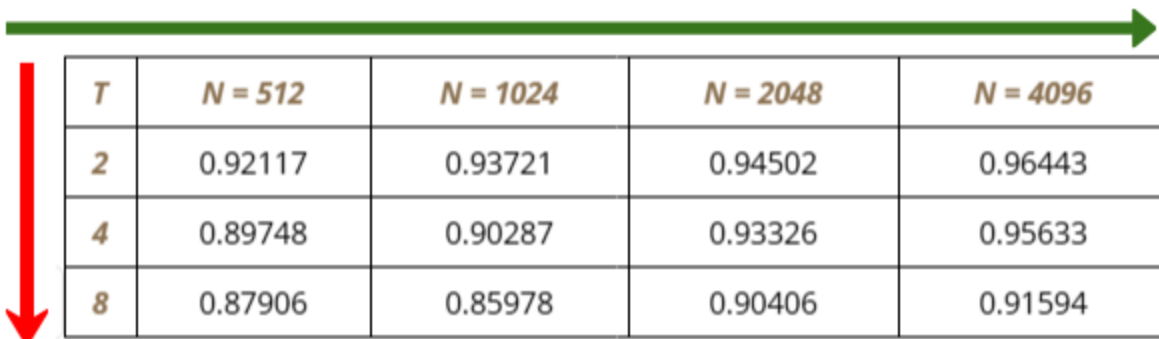
Algoritmo paralelo empleando OpenMP - Speedup



<i>T</i>	<i>N = 512</i>	<i>N = 1024</i>	<i>N = 2048</i>	<i>N = 4096</i>
<i>2</i>	1.96294	1.96955	2.00486	2.01163
<i>4</i>	3.79402	3.75200	3.86343	4.02726
<i>8</i>	7.06280	6.84794	7.23465	7.66346

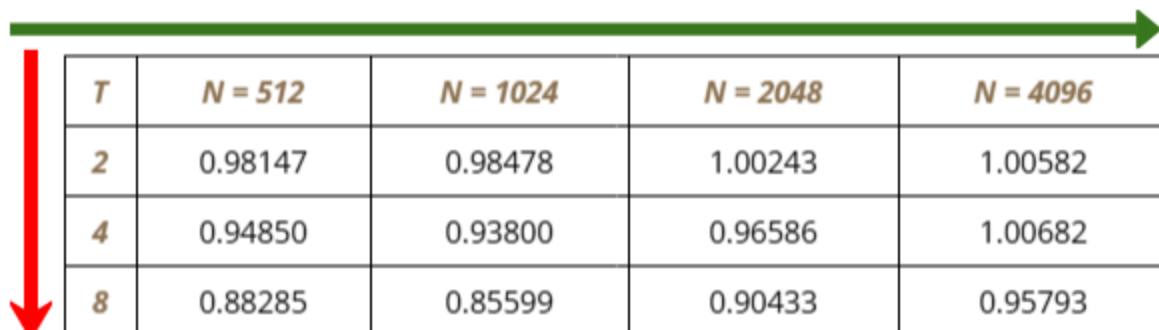
Los resultados obtenidos nos muestran que para un tamaño de N dado al aumentar el número de hilos el speedup mejora, esto se relaciona a que si utilizamos mayor cantidad de unidades de procesamiento (hilos) la solución se obtiene en un tiempo menor. Pasa lo mismo si mantenemos fija la cantidad de procesadores (T), y aumentamos el tamaño del problema (N) también vemos una mejora en casi todos los casos en el speedup, esto tiene que ver con que cuando uno incrementa el tamaño del problema pero mantiene fija la cantidad de procesadores (T) la proporción de sincronización o comunicación de la solución no crece en la misma proporción de cómputo, al contrario, la proporción de cómputo suele crecer más que la proporción de sincronización o comunicación, eso lleva a que se genera una menor cantidad de overhead y hay una mejora en el rendimiento.

- Algoritmo paralelo empleando Pthreads - Eficiencia  $\rightarrow$  Speedup / P (cant hilos)



<i>T</i>	<i>N = 512</i>	<i>N = 1024</i>	<i>N = 2048</i>	<i>N = 4096</i>
<i>2</i>	0.92117	0.93721	0.94502	0.96443
<i>4</i>	0.89748	0.90287	0.93326	0.95633
<i>8</i>	0.87906	0.85978	0.90406	0.91594

- Algoritmo paralelo empleando OpenMP - Eficiencia



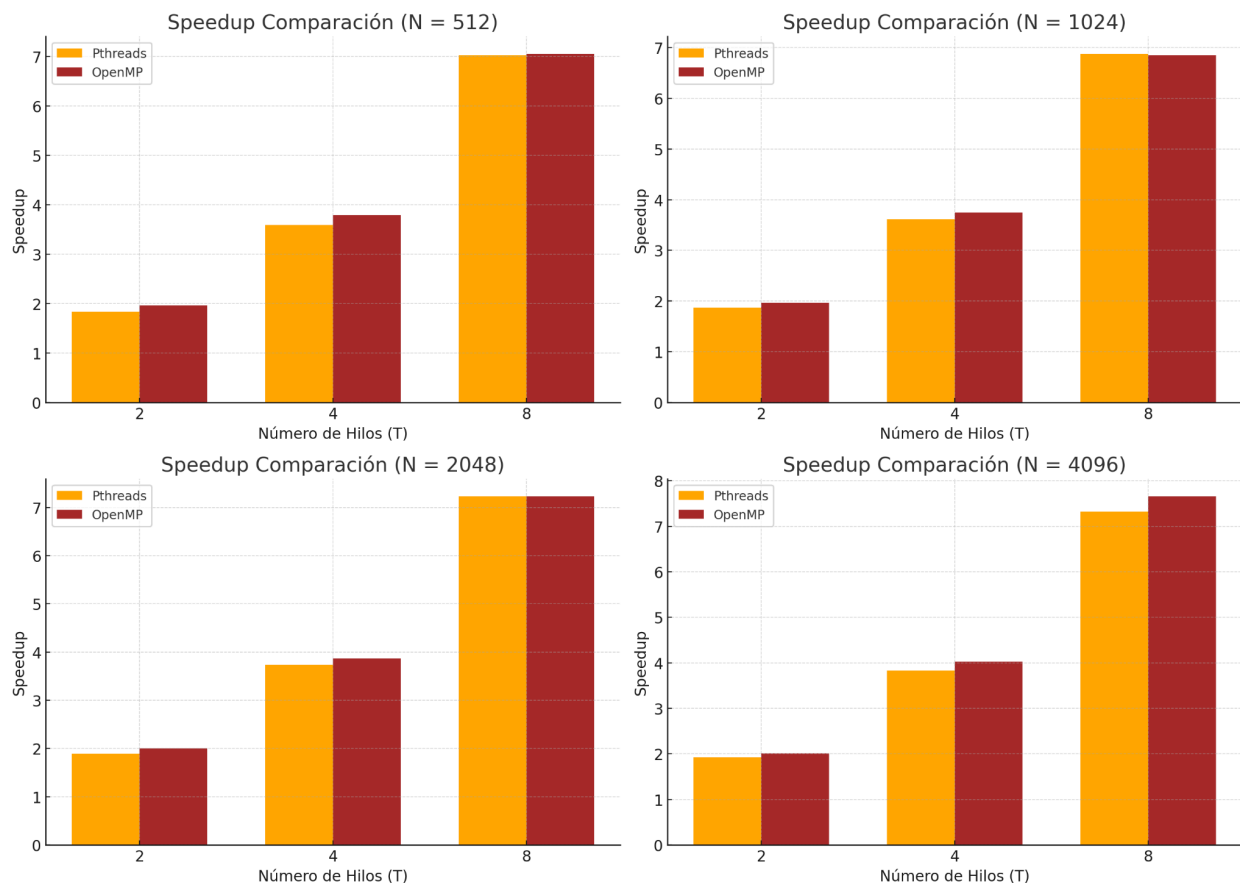
<i>T</i>	<i>N = 512</i>	<i>N = 1024</i>	<i>N = 2048</i>	<i>N = 4096</i>
<i>2</i>	0.98147	0.98478	1.00243	1.00582
<i>4</i>	0.94850	0.93800	0.96586	1.00682
<i>8</i>	0.88285	0.85599	0.90433	0.95793

Como podemos ver en las tablas de eficiencia, los resultados muestran que cuando mantenemos fija la cantidad de procesadores (T) y aumentamos el tamaño del problema

(N), la eficiencia mejora y esto se relaciona con que el overhead va disminuyendo a medida que el tamaño del problema crece.

En sentido opuesto, las flechas rojas muestra cómo dejando un tamaño de problema fijo (N) y aumentando la cantidad de procesadores (T) por más que el speedup mejore es mayor la cantidad de procesos que tienen que sincronizar o comunicarse, es por eso que la proporción de overhead crece más que la proporción de cómputo bajando así la eficiencia.

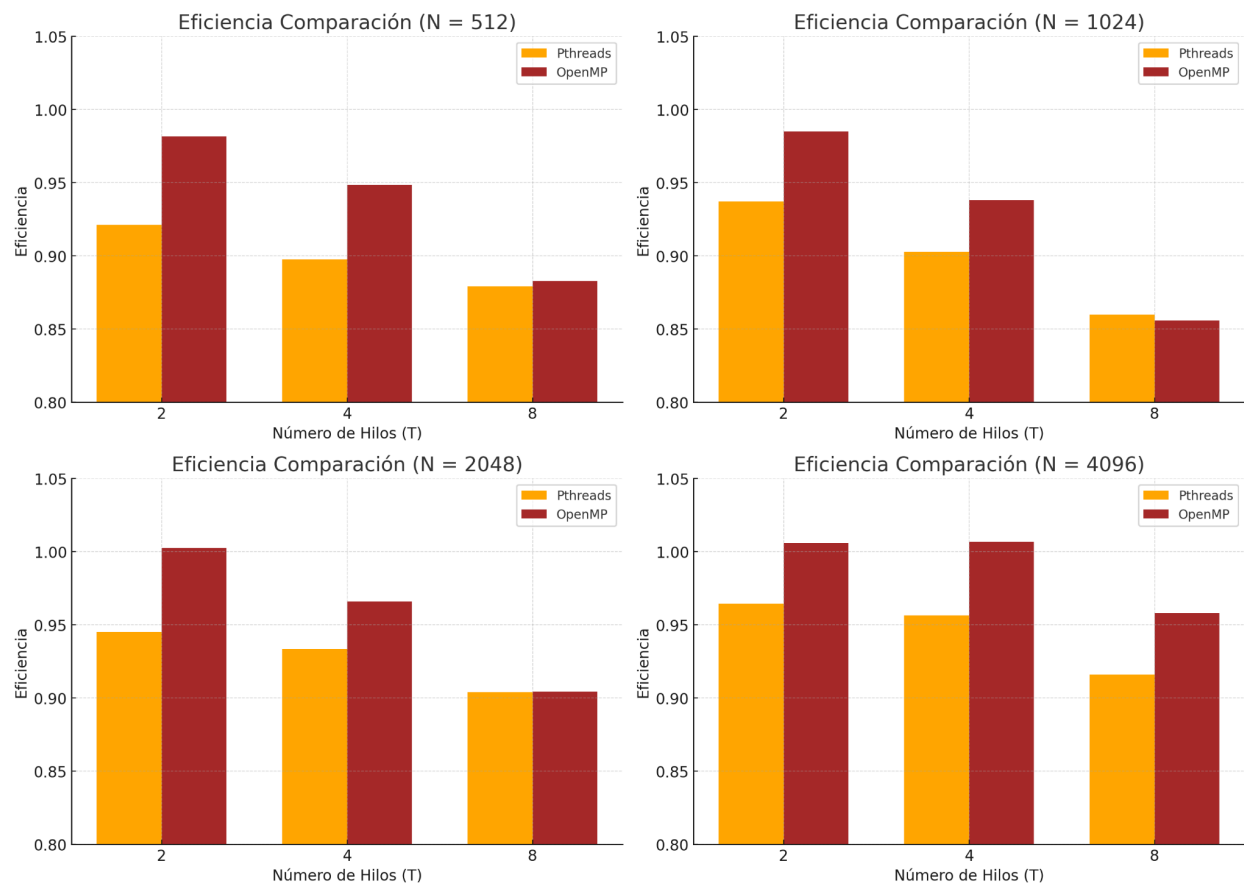
### Comparación de speedup entre Pthread y OpenMP



Ambos enfoques paralelos (PThreads y OpenMP) muestran un aumento en el speedup al incrementar los hilos (T), con OpenMP logrando un rendimiento ligeramente superior. Esto sugiere que OpenMP tiene una mejor eficiencia en la utilización de recursos, especialmente para matrices más grandes (N).

---

## Comparación de eficiencia entre Pthread y OpenMP

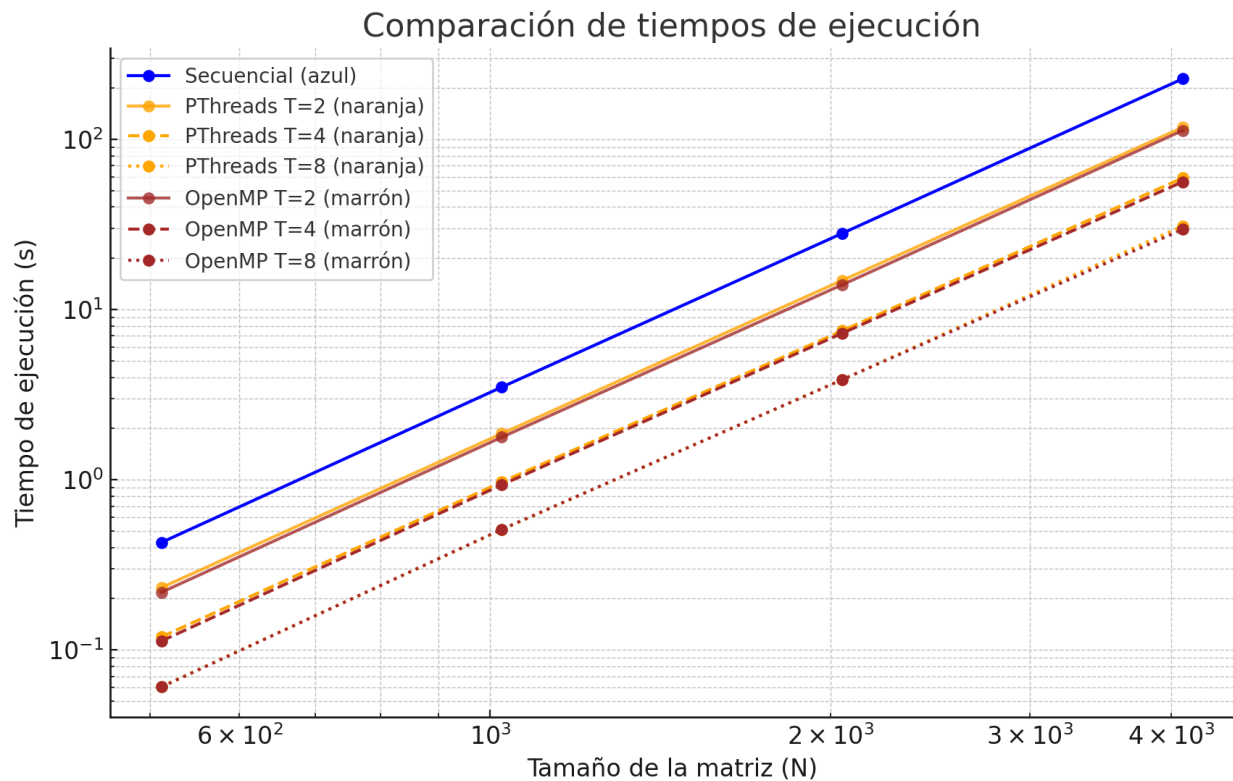


OpenMP muestra una eficiencia consistentemente superior a PThreads, especialmente en matrices grandes (N), alcanzando el 100% en algunos casos debido al mejor aprovechamiento de los recursos.



---

## Conclusiones respecto a los tiempos



Los resultados de tiempo de ejecución en el clúster remoto mostraron que tanto Pthreads como OpenMP lograron mejoras significativas en el tiempo de ejecución en comparación con la versión secuencial. Siendo OpenMP ligeramente más rápido en casi todos los casos.