Análisis por Regresión del Escalamiento Empírico de SampleMerge en Procesadores Multinúcleo

Benjamín Enrique Parra Barbet
Universidad Austral de Chile
Valdivia, Chile
benrrixworks@gmail.com

18 de Diciembre de 2025

Abstract El análisis de algoritmos paralelos suele apoyarse en modelos de complejidad asintótica que asumen un acceso uniforme a memoria. Sin embargo, en arquitecturas modernas, la jerarquía de caché y el ancho de banda introducen restricciones que alteran significativamente el comportamiento empírico. En este trabajo se estudia el algoritmo de ordenamiento paralelo *SampleMerge*, evaluando en qué medida un modelo de regresión lineal múltiple, construido a partir de sus términos asintóticos dominantes, es capaz de explicar el tiempo total de ejecución en un procesador x86-64 moderno.

A partir de un conjunto de experimentos controlados, se analizan métricas de escalabilidad, *throughput* y descomposición temporal por fases. Los resultados muestran que, si bien el modelo captura adecuadamente el escalamiento en un régimen *compute-bound*, su validez se degrada abruptamente cuando el tamaño de entrada supera la capacidad de la caché L3, dando lugar a un comportamiento *memory-bound* dominado por la fase de fusión paralela. Este cambio de régimen se manifiesta estadísticamente como heterocedasticidad y desviaciones sistemáticas de la normalidad en los residuos.

Restringiendo el dominio del análisis a tamaños de entrada que permanecen dentro de la caché L3 y empleando un ajuste mediante mínimos cuadrados ponderados, el modelo propuesto explica más del 98% de la varianza observada y permite cuantificar la contribución relativa de las fases de ordenamiento y fusión. Los resultados evidencian que el costo real del algoritmo está fuertemente condicionado por los patrones de acceso a memoria, destacando las limitaciones de los modelos de complejidad tradicionales para describir el rendimiento de algoritmos paralelos en arquitecturas contemporáneas.

Palabras clave: empirical complexity; regression analysis; cache hierarchy; memory-bound computation; performance modeling

I INTRODUCCIÓN

A Problema: Complejidad Teórica vs. Realidad Empírica

El análisis de algoritmos mediante cotas asintóticas suele ser la manera sobre la cual se realizan las estimaciones del tiempo de ejecución, midiendo la cantidad de operaciones elementales a realizar para cierta entrada y determinando su ejecución en una unidad de tiempo. Si bien el modelo RAM es solido para realizar estimaciones y comparaciones cuando los datos de entrada son muchos, el modelo de arquitectura actual presenta multiples capacidades las cuales hacen que estas aproximaciones se compliquen. Hoy surgen otros factores de arquitectura hacen mas dificil dar una estimación empirica correcta:

- La capacidad ***superescalar*** permite ejecutar instrucciones fuera de orden.
- La introducción de instrucciones SIMD facilitan ejecutar una instruccion sobre multiples datos, lo que en la práctica se traduce en la division de unidades de tiempo.
- La cantidad de núcleos en procesadores domésticos permite mantener multiples flujos paralelos, permitiendo diseñar algoritmos para dividir el trabajo (modelo PRAM).
- La programación distribuida permite llevar un problema a varios servicios de cómputo.
- La latencia física hace que algunas instrucciones se demoren mas que otras.
- El cuello de botella de memoria y los niveles de cache alteran la velocidad de búsqueda de memoria en ejecución.

Entre más de estas optimizaciones modernas se consideren, producir una estimación que se apegue a la realidad requiere cada vez mayor granularidad.

Un ejemplo son los tiny sorters, que suelen utilizar redes de ordenamiento para producir una salida pequeña ordenada en $O(n \log^2 n)$ comparaciones, que es peor que un ordenamiento estandar como *mergesort* $O(n \log n)$ o radixsort $O(nr)$ (con r la base numérica) pero se comporta mejor en la práctica debido a que es predecible y vectorizable para un procesador convencional.

B Algoritmo a Analizar: SampleMerge

Sample merge es una fusión sencilla entre el algoritmo de ordenamiento samplesort y mergesort. Su funcionamiento se resume en las siguientes fases:

1. **Ordenamiento por bloques:** Se determinan k bloques de tamaño $n/|Threads|$ para n el tamaño de entrada. Se ordenan de manera paralela con el algoritmo más rápido del estado del arte.

2. **Selección de pivotes:** Se toman k elementos candidatos equiespaciados por bloque. Cada elemento i seleccionado por bloque luego se ordena y se saca la mediana para determinarlo como pivote. Cada pivote se define como P_i con $i \in \{1, 2, .., k-1\}$.

3. **Categorización:** Se subparticionan las partes ordenadas en "Runs". La run i contiene todos los elementos mayores a P_{i-1} y menores a P_i. Se identifican por búsqueda binaria.

4. **Calculo de tamaño de buckets:** Se calcula la suma de los tamaños de cada run categorizada, y se realiza la suma acumulada (prefix sum).

5. **Fusión de runs:** Utilizando un arreglo adicional para el destino, se hace fusion de k-vias sobre las runs ordenadas.

El orden asintótico por fase, medido en la cantidad de comparaciones, donde p es la cantidad de threads, N la cantidad dee elementos, R la run mas grande y B el tamaño del bucket más grande, log representa al logaritmo en base 2:

1. $O(\frac{n}{k} log \frac{n}{k})$.
2. $O(k \log k)$.
3. $O(k \log R)$.
4. $O(k)$.
5. $O(B \log k)$ escalar con tournament tree, $O(B \log_W k)$ con arbol con W elementos comparables con hmin SIMD simultaneamente.

Siendo el tiempo asintotico total:
$T(n) = O(\frac{n}{k} log \frac{n}{k}) + O(k \log k) + O(k \log R) + O(k) + O(B \log_W k)$
O simplificado a las expresiones mas importantes:

$$T(n) \approx O(\frac{n}{k} log \frac{n}{k}) + O(B \log_W k)$$

El análisis anterior solo considera comparaciones, lo que esconde un problema importante. Como muchos otros algoritmos, al paralelizarse, la velocidad de cómputo deja de ser el cuello de botella, volviendose el costo de las busquedas en memoria los mas importantes, y en este caso, especialmente en la fase de fusión, tener muchos threads puede hacer que los accesos dispersos hagan que el procesador no pueda predecir bien que elementos traer, trayendo saturación de cache al intentar mantener gran parte del arreglo a cache.

En la práctica, se usará __gnu_parallel::multiway_merge para la etapa de fusion ($O(\frac{n}{k} \log k)$) y vqsort para la etapa inicial (quicksort vectorizado), teniendo en total un tiempo de ejecución aproximado (si los pivotes mantienen la entrada equilibrada):

$$T(n) \approx O(\frac{n}{k} log \frac{n}{k}) + O(\frac{n}{k} \log k)$$

II DEFINICIÓN DEL PROBLEMA

A Pregunta de Investigación

¿En qué medida un modelo de regresión lineal múltiple, basado en la complejidad asintótica teórica, es capaz de predecir el tiempo de ejecución del algoritmo *SampleMerge* en una arquitectura moderna, y de qué manera la jerarquía de memoria (caché L3) y el ancho de banda actúan como factores limitantes que invalidan los supuestos de normalidad y homocedasticidad del modelo?

B Hipótesis de Trabajo

Se proponen las siguientes conjeturas para el desarrollo del estudio:

1. H_1 **(Dominancia de fase):** Pese a que el orden asintótico de la fase de ordenamiento es superior, el coeficiente de la fase de fusión ($\frac{n}{k} \log k$) presentará una magnitud mayor en el modelo empírico debido a la ineficiencia en el acceso a memoria por dispersión de punteros.

2. H_2 **(Validez del Modelo):** La capacidad de predicción del modelo y el cumplimiento de los supuestos estadísticos de Gauss-Markov están condicionados a que el volumen de datos (n) no exceda la capacidad de la caché L3 (32 MB).

3. H_3 **(Límite de Escalabilidad):** El algoritmo dejará de mostrar ganancias de rendimiento antes de agotar los núcleos físicos disponibles, debido a la saturación del bus de datos durante la fase de fusión paralela.

C Justificación de la Hipótesis

La justificación de estas hipótesis se fundamenta en la discrepancia entre el **modelo de máquina RAM**

(donde el acceso a memoria es constante) y la **arquitectura real x86-64** del Ryzen 5 5600X.

La fase de fusión de k-vías de *__gnu_parallel_merge* requiere mantener múltiples iteradores activos. Al incrementar n, la probabilidad de *cache misses* aumenta exponencialmente una vez que el conjunto de trabajo supera los 32 MB de la L3. Esto introduce una latencia estocástica que no es capturada por los términos logarítmicos del modelo, manifestándose estadísticamente como una distribución de error con colas pesadas (alta curtosis) y varianza no constante. La eliminación de los datos que exceden la L3 se justifica, por tanto, como una delimitación del dominio del modelo hacia un entorno de ejecución *compute-bound* en lugar de *memory-bound*.

III DEFINICIÓN DEL MODELO ESTADÍSTICO

A Taxonomía de Variables y Fuentes de Datos

El análisis se sustenta en dos conjuntos de datos independientes, obtenidos en ejecuciones aisladas para garantizar la pureza de las mediciones. El primero permite una visión macroscópica y comparativa frente a librerías estándar, mientras que el segundo registra la instrumentación detallada de las fases internas.

A.1 Dataset de Aspecto General (`benchmark_results.csv`)

Este conjunto registra el rendimiento total del sistema. Se utiliza para el análisis de *speedup* y detección del punto de cruce. Los campos corresponden exactamente a la cabecera del archivo generado por el experimento macro. El archivo cuenta con 7560 observaciones.

A.2 Dataset de Análisis Granular (`benchmark_granular.csv`)

Este conjunto proviene de una ejecución independiente con instrumentación activa para desglosar el tiempo en etapas atómicas. Se utiliza para el ajuste de los modelos de regresión lineal. El archivo cuenta con 304 observaciones.

A.3 Justificación Metodológica

- **Tratamiento de n:** Aunque n es un parámetro definido por el diseño experimental, se trata como una variable de escala métrica. Esto permite capturar la relación funcional no lineal establecida por la complejidad asintótica teórica.
- **Modelado de Hilos (p):** Se opta por una codificación *dummy* (nominal). Esta decisión se fundamenta en la observación de una relación no monótona: el rendimiento no escala linealmente con p debido a la saturación del ancho de banda y al estancamiento observado empíricamente a partir de los 4 hilos físicos.
- **Variables de Respuesta:** En el análisis granular, los tiempos se registran en milisegundos (ms). Se utiliza la mediana de las réplicas para cada nivel de los factores para obtener un estimador robusto frente al ruido térmico y las fluctuaciones del planificador del sistema operativo.
- **Overhead de Medición:** Se reconoce que el dataset granular presenta un error experimental ligeramente superior ($\epsilon_g > \epsilon_a$) debido a la instrumentación frecuente del reloj de alta resolución entre fases, lo cual introduce una latencia mínima acumulativa no presente en el dataset general.

B Modelos de Regresión Propuestos

Se definen los siguientes modelos para evaluar la influencia de las fases de *SampleMerge* sobre el tiempo total T, basándose en los datos del archivo granular:

1. Modelo de Fase Dual (M1): $T_i = \beta_0 + \beta_1 \left(\frac{n_i}{k} \log \frac{n_i}{k}\right) + \beta_2 \left(\frac{n_i}{k} \log k\right) + \epsilon_i$
2. Modelo de Ordenamiento Dominante (M2): $T_i = \beta_0 + \beta_1 \left(\frac{n_i}{k} \log \frac{n_i}{k}\right) + \epsilon_i$
3. Modelo de Fusión Dominante (M3): $T_i = \beta_0 + \beta_2 \left(\frac{n_i}{k} \log k\right) + \epsilon_i$

Donde β_0 representa el *overhead* fijo de paralelización y ϵ_i el término de error. Se postula que ϵ_i presentará heterocedasticidad cuando n supere el tamaño de la caché L3 (32 MB).

C Entorno de Cómputo

Las pruebas se realizaron en una arquitectura controlada para aislar los efectos de la jerarquía de memoria en la latencia:

- **Procesador:** AMD Ryzen 5 5600X (Zen 3, 6 núcleos físicos / 12 hilos lógicos).
- **Memoria RAM:** 16 GB DDR4 G.Skill Trident Z Pro a 3600 MHz.
- **Jerarquía de Caché:** L1 (384 KB), L2 (3 MB) y **L3 (32 MB)**.
- **Software:** WSL Arch Linux (Kernel 2.5.7), GCC 15.2.1, banderas `-O3 -march=native`.

Tabela 1: Variables del Dataset de Aspecto General (`benchmark_results.csv`)

Variable (CSV)	Rol Estadístico	Unidad / Escala	Rango / Dominio
`Algorithm`	Factor de Contraste	Nominal	{`std::sort, SampleMerge`}
`Distribution`	Factor de Control	Nominal	{`Uniform, Normal`}
`Threads`	Factor de Diseño	Discreta	$\{0, 1, 2, 4, 8, 12, 16\}$
`InputSize`	Predictor Métrico	Razón (Elementos)	$[2^8, 2^{26}]$
`TimeSeconds`	Variable Respuesta	Continua (s)	R^+

Tabela 2: Variables del Dataset de Análisis Granular (`benchmark_granular.csv`)

Variable (CSV)	Rol Estadístico	Unidad / Escala	Rango / Dominio
`Algorithm`	Factor de Contraste	Nominal	{`SampleMerge`}
`Distribution`	Factor de Control	Nominal	{`Uniform, Normal`}
`Threads`	Factor de Diseño	Nominal (*Dummy*)	$\{0, 1, 2, 3, 4, 5, 6\}$
`InputSize`	Predictor Métrico	Razón (Elementos)	$[2^9, 2^{26}]$
`TotalTime`	Respuesta Principal	Continua (ms)	R^+
`SortPhase`	Respuesta Secund.	Continua (ms)	R^+
`PivotPhase`	Respuesta Secund.	Continua (ms)	R^+
`SubpartPhase`	Respuesta Secund.	Continua (ms)	R^+
`OffsetPhase`	Respuesta Secund.	Continua (ms)	R^+
`MergePhase`	Respuesta Secund.	Continua (ms)	R^+

D Diseño Experimental

El experimento sigue un diseño factorial completo explorando tres dimensiones:

1. **Tamaño (n):** De $n = 2^8$ (General) o 2^9 (Granular) hasta $n = 2^{26}$ elementos.

2. **Paralelismo (p):** Rango de 1 a 16 hilos. El análisis se enfoca en el intervalo $[2, 6]$ para detectar el punto de saturación del bus de memoria.

3. **Distribución:** Escenarios `Uniform` (base) y `Normal` (inducido para evaluar la sensibilidad al *skew*).

E Protocolo de Captura y Robustez

- **Reducción de Ruido:** Cada punto de medición es la **mediana de 3 experimentos consecutivos**.

- **Muestreo:** Se realizaron **30 réplicas** independientes por combinación para el dataset general, asegurando estabilidad estadística.

- **Precisión:** Captura mediante `std::chrono::high_resolution_clock`.

IV DESARROLLO Y EXPERIMENTACIÓN

A Proceso de medición

La meta implementación del algoritmo y su medición de tiempo por experimento se puede encontrar en el anexo A. Se utilizó python con Jupyter Notebook para hacer el análisis paso a paso.

B Análisis Exploratorio y Detección de Anomalías

Para esta sección se trabaja con el dataset del análisis de aspecto general (tabla 1) considerando datos de threads 2,4,6, quedando 114 observaciones que representan la mediana de 3 experimentos.

El gráfico en 1 muestra el speedup alcanzado respecto al ordenamiento estandar de la std con politica de ejecucion *std::execution::par_unseq* que cuenta con paralelismo (6 threads) y vectorización. Para esta ejecución se contempla una etapa de merge completamente paralela con *__gnu_parallel::multiway_merge* (Threads constante en etapa final), mientras se varía la cantidad de elementos y los threads en valores pares para simplificar la visualización. Tampoco se contempla ejecución con 1 thread ya que solo se ejecutaría la etapa 1.

Se puede observar que al alcanzar los 10.000 elementos, el radio de la velocidad de ordenamiento de SampleMerge y std::sort con 6 threads comienza a acercarse a 1, y

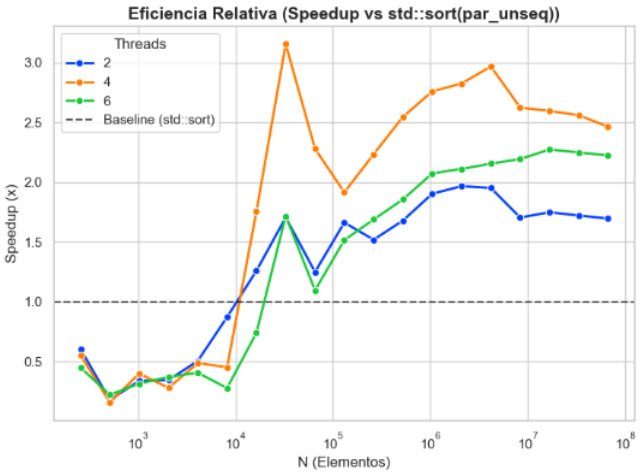

Figura 1: Escalabilidad

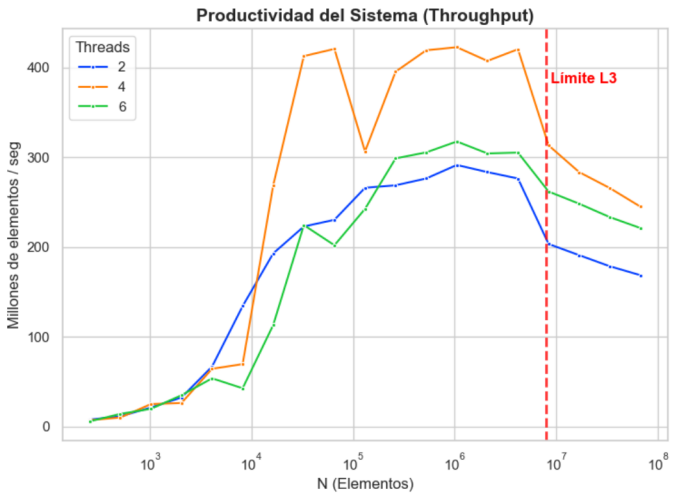

Figura 2: Througput vs cantidad de elementos

luego comienza a superarlo rápidamente. La eficiencia máxima alcanzada es con 4 threads y 32.765 elementos (recordando que va en potencias de 2).

Luego de este punto, el speedup vs cantidad de elementos deja de aumentar significativamentey se mantiene un un rango estable. El hecho de que el speedup no se maximice con mas threads puede indicar dos cosas importantes:

- El computo no es el cuello de botella. (Esperado).
- El overhead de sincronización aumenta mucho por cantidad de threads.
- Las fase de merge empeora por tener que mantener mas accesos dispersos en memoria (Runs = $NUM\_THREADS^2$). (Cache thrashing posible).

El siguiente análisis contempla medir la cantidad de elementos (en millones) procesados por segundo. Esto se obtiene dividiendo el tamaño de la entrada por el tiempo de ordenamiento y se refleja como *throughput*. En la figura 2.

Se puede observar como el througput no es lineal, probablemente porque la fase de fusión comienza a pesar más por operación (Más runs significan mas comparaciones), por otro lado, se colocó una linea roja que marca el limite del nivel de cache mas alto del procesador donde se realizaron las pruebas. La cantidad de elementos que saturan L3 es alrededor de 32 MB, aproximadamente el máximo teórico situandose en $|L_3| \div |sizeof(int)| = 32.000.000 \div 4 = 8.000.000$ enteros.

Como se esperaba, al saturarse L3 el rendimiento cae de manera rápida, esto demuestra que la fase de merge efectivamente satura los accesos, teniendo que mantener una copia del arreglo en cache para que el paralelismo se aproveche al máximo.

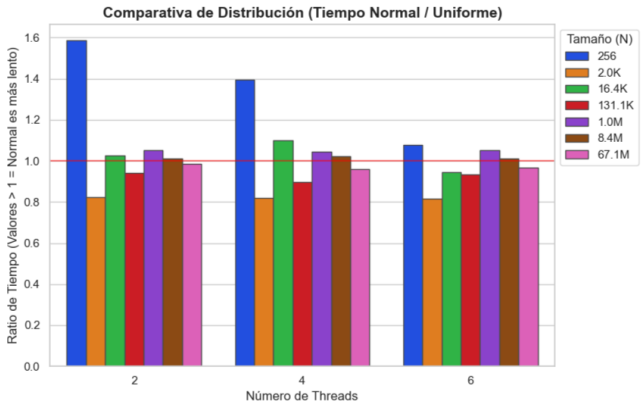

Figura 3: Resistencia a distribución - Radio de tiempo de ordenamiento normal vs uniforme

Entre otras visualizaciones, se midió el radio de tiempo de ordenamiento para ambas distribuciones, de manera de comparar si una distribución se demora mas que otra en ordenarse para distintos valores de entrada. El gráfico que representa esto se puede encontrar en la figura 3.

El eje X mide la **división entre el tiempo de ordenamiento que tardó en promedio el algoritmo con una distribución normal, entre lo que demoró en promedio con la distribución uniforme**. Los valores indican que el algoritmo es robusto a los cambios de distribuciones, manteniendo mas estabilidad con tamaños de entrada más grandes.

Para concluir el análisis exploratorio, se visibiliza la **latencia total** (tiempo de ordenamiento en segundos) contra std::sort paralelo, de manera de ver cuanto y cuando es la ganancia según la cantidad de elementos. También se agregó una linea roja para mostrar donde se encuentra el límite de la cache L3.

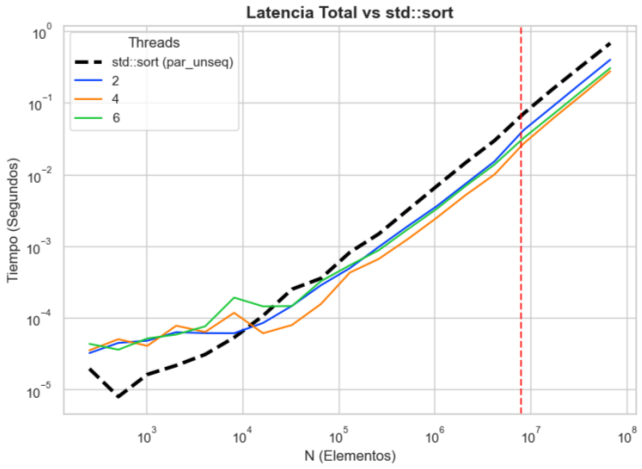

Figura 4: Latencia SampleMerge vs std::sort par_unseq

Se puede ver que mientras de acerca a la linea, la latencia sigue una curva que diverge pero de pronto se estabiliza. Otro punto notorio, es que la fase de ordenamiento con 2 threads es superior a std::sort, pero luego no hay tanto beneficio en una escala absoluta como lo es el tiempo. Esto se puede ver en el gráfico 4. El beneficio surge a partir de los 10^4 elementos como se vió anteriormente.

C Análisis por etapas

Antes de proseguir, para confirmar que es la fase de merge la que causa problemas, utilizando el análisis granular se realizó el grafico de la figura 5 que representa el tiempo promedio de las distintas etapas contra un speedup normalizado ($T_i = T_{i-1} \div 2$, $T_1 = t_1$, $i \in \{1, 2, 3, 4, 5, 6\}$ y t_1 representa el tiempo que tomo con un thread, equivalente a solo la fase de sort lineal). La figura muestra que el speedup del sort sigue aumentando levemente, pero la fase de merge empeora cada vez más. Esto era de esperarse considerando que mas threads implica mas fusiones.

D Metodología de Filtrado

Ya que los datos en L3 introducen un comportamiento poco predecible, es necesario que la regresion considere solo entradas que entren en L3 para ser efectiva, siendo este un supuesto añadido gracias al análisis exploratorio. El valor que recorrerá n entonces es desde 2^{10} (obtenido del análisis exploratorio, momento del que comienza a competir con *std::sort*) hasta los 2^{22}, lo que deja un margen de espacio para los procesos internos sin saturar la L_3. En total luego de estos filtros, se mantienen 540 observaciones por cada thread (de 2 hasta 6).

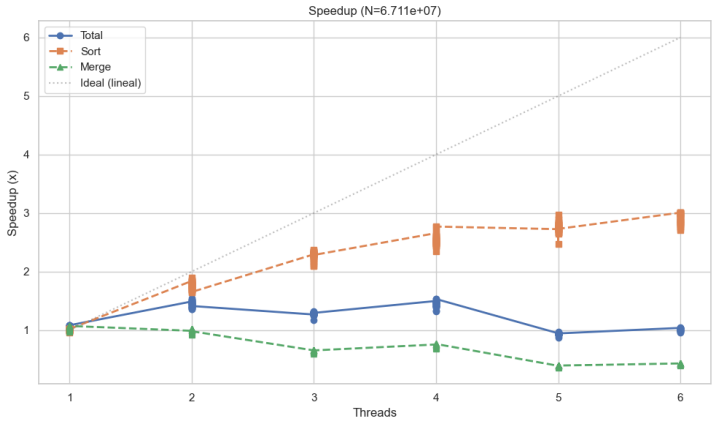

Figura 5: Speedup para $n = 2^{26}$ en fase de sort, merge y total

E Modelos de regresión

El modelo de regresión propuesto utiliza el análisis asintótico realizado anteriormente para determinar los factores reales de cada etapa. Los modelos de regresión propuestos se pueden encontrar en la subsección B. Para poder probar su validez, se deben cumplir los supuestos de normalidad de residuos y homocedasticidad (sabiendo que las pruebas son independientes).

E.1 Supuestos

- **Homocedasticidad**: Para comprobar si no hay heterocedasticidad, se realizó una regresión del modelo de fase dual y se graficó el error (ver gráfico 6). Con esto se puede ver que el error aumenta entre mayor es la cantidad de entrada, lo que rompe el supuesto de homocedasticidad fuertemente. De manera paralela, se realizó un test de Breusch-Pagan, que entregó un p-valor de $2.906e-28$, por lo que se puede decir con mucho mas de 95% de confianza de que no se cumple esta condición.

  Para reducir el ruido experimental, se utilizó la mediana de tres ejecuciones, sin embargo, con lo anterior se detectó que la varianza del error aumenta sistemáticamente con el tamaño de entrada n (heterocedasticidad). Ante esta violación de supuestos, el modelo se ajustó mediante Mínimos Cuadrados Ponderados (WLS) con pesos $w_i = 1/n_i$ para estabilizar la varianza. Finalmente, se emplearon errores estándar robustos (covarianza HC3) para garantizar que las inferencias y p-valores sean válidos frente a la no-normalidad y autocorrelación residual.

- **Normalidad**: El análisis de los residuos de la regresión anterior revela desviaciones respecto de una distribución normal. Sin embargo, la normalidad no

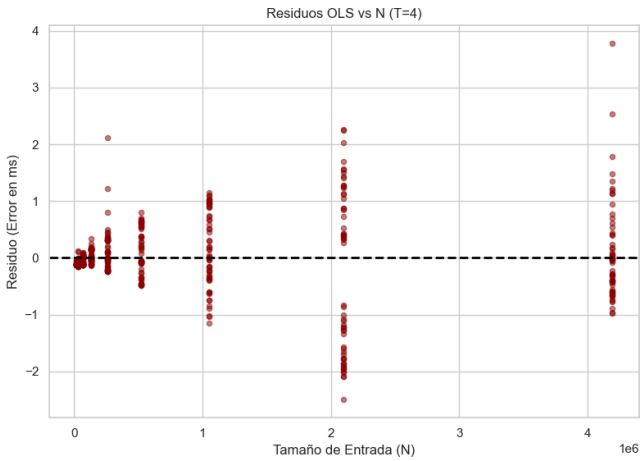

Figura 6: Error vs tamaño de entrada (modelo dual)

constituye un requisito estructural en este contexto, dado que el modelo se emplea para caracterizar el escalamiento empírico del algoritmo y no como una descripción probabilística completa del sistema. El tamaño de la muestra contribuye a la estabilidad del ajuste, permitiendo una interpretación consistente de los costos de ordenamiento y fusión.

V ANÁLISIS DE RESULTADOS

Para comenzar, se realizó un análisis de colinealidad sobre el modelo completo (ver snippet B). Los resultados se pueden encontrar en la tabla 3. Estos valores son esperados ya que al ser k igual a una variable dummy que es igual a la cantidad de threads, siempre será considerada una constante para los diferentes modelos, en base a esto, la fase de calculo de pivotes y suma de prefijos tiende a un valor fijo por lo que se pueden descartar. Por otro lado, Sort y Merge son fuertemente independientes y la fase de clasificación parece también encajar en el modelo sin causar problemas con las otras variables. Se agrego una variación mas como "modelo triple"que agrega la fase de clasificación al modelo dual.

Para las regresiones, debido al condition number inherente de la magnitud de los datos, se optó por agregar un modelo normalizado. Las métricas del modelo original se encuentran en la tabla 4.

Al analizar su indice de determinación, se muestra que los cuatro modelos pueden explicar un 98% de la varianza, con un AIC favoreciendo escoger al modelo triple sobre el dual. Si se profundiza viendo los coeficientes de las regresiones (tabla 5) se puede ver que el modelo triple tiene problemas para representar la realidad, dejando el coeficiente de la fase de fusión (β_{merge}) negativo para compensar el valor de la fase de ordenamiento y la

Tabla 3: Análisis de Multicolinealidad (VIF) - Modelo con Clasificación de Runs

Variable	**VIF**	**Estado**
Constante	58.56	-
Sort ($\frac{n}{k}\log\frac{n}{k}$)	1.44	Aceptable
Merge ($\frac{n}{k}\log k$)	1.93	Aceptable
Run_Clasification ($k\log\frac{n}{k}$)	**3.80**	**Aceptable**
Pivots ($k\log k$)	92.01	Multicolinealidad
Prefix (k)	93.99	Multicolinealidad

de clasificación. Debido a esto, la alternativa dual es el que mas se acerca sin sobrecomplejizar el modelo y manteniendo coeficientes posibles.

En el gráfico 7 en los anexos, se puede ver un QQ-Plot para evaluar la normalidad de estos modelos. De esto se puede decir que se acerca pero aun tiene outliers al final en los cuatro modelos.

A Evaluación del Ajuste (R^2 y Estadístico F)

Ahora que se seleccionó el modelo dual, en esta sección se presentan los resultados de la regresión. La primera consideración es que debido al crecimiento exponencial del tamaño de elementos, la escala de visualización se ajusta de manera logaritmica, al igual que el tiempo de ejecución. De esta manera, utilizando el modelo de regresión dual, con Weighted Least Squares ($w_i = 1/n_i$) y covarianza HC3 para corregir la homocedasticidad, filtrado de datos a 540 entradas ($2^{12} < n \leq 2^{22}$) que entran en L3 y el ajuste de escala logaritmico se puede visualizar los resultados junto con un QQ-Plot para evaluar normalidad por cada numero de threads, la estimación final se puede ver en la figura 8 del anexo.

Los valores de los coeficientes por cantidad de threads se encuentran en la tabla 7.

De estos resultados se puede observar que la fase de fusión tiene coeficiente negativo en 2 y 3 threads. En estos casos, la velocidad de fusión se vuelve dificil de separar de la fase de ordenamiento al aportar valores parecidos de tiempo, pero como ya se discutió que el speedup de la fase de fusión es negativo, con mas threads comienza a demorarse mas y se vuelve un factor mas importante para determinar el tiempo total. Esta tendencia se puede visualizar tambien mientras la cantidad de threads aumenta, los coeficientes muestran que el costo de β_{merge} continua aumentando.

VI CONCLUSIÓN

En este trabajo se evaluó empíricamente la capacidad de un modelo de regresión lineal múltiple, basado en la

Tabela 4: Comparativa de métricas de ajuste para modelos de complejidad ($T = 4$)

Modelo	R^2	**AIC**	**Cond. No.**	**JB p-value**
Solo Sort	0.9828	224.54	1.29×10^6	$< 2.2 \times 10^{-16}$
Solo Merge	0.9824	235.70	1.37×10^5	$< 2.2 \times 10^{-16}$
Dual	**0.9834**	**207.75**	1.73×10^6	$< 2.2 \times 10^{-16}$
Triple	0.9840	190.90	2.87×10^7	$< 2.2 \times 10^{-16}$

Tabela 5: Coeficientes originales (milisegundos por operación teórica, T=4)

Modelo	**Constante**	β_{sort}	β_{merge}	β_{run_class}
Solo Sort	0.0402	9.81×10^{-7}	—	—
Solo Merge	-0.0196	—	9.00×10^{-6}	—
Dual	0.0136	5.49×10^{-7}	4.00×10^{-6}	—
Triple	-0.6407	1.27×10^{-6}	-3.00×10^{-6}	0.0137

complejidad asintótica teórica, para explicar y predecir el tiempo de ejecución del algoritmo *SampleMerge* en una arquitectura moderna x86-64 (Ryzen 5 5600X). Los resultados muestran que, si bien los términos derivados del análisis asintótico capturan adecuadamente la tendencia global del escalamiento, su validez está fuertemente condicionada por las restricciones impuestas por la jerarquía de memoria y costo real de operaciones.

El análisis exploratorio evidenció que, al superar la capacidad de la caché L3 (32 MB), el rendimiento del algoritmo deja de escalar de forma predecible. La fase de fusión paralela introduce una latencia dominante posiblemente asociada a accesos dispersos a memoria, lo que se manifiesta empíricamente como una pérdida de *speedup*, una caída abrupta del *throughput* y una variabilidad creciente en los tiempos. Este comportamiento confirma que el algoritmo transita desde un escenario *compute-bound* a uno *memory-bound*, donde eventos particulares no siempre tomados en cuenta fuera de *HPC*, como el *cache trashing* y las latencias jerarquicas de memoria comienzan a afectar al rendimiento significativamente.

Desde el punto de vista estadístico, esta transición se manifestó en la ruptura de los supuestos clásicos del modelo lineal. En particular, la presencia de heterocedasticidad motivó el uso de mínimos cuadrados ponderados (WLS) junto con errores estándar robustos (HC3), permitiendo estabilizar el ajuste dentro de un dominio restringido ($2^{12} < n \leq 2^{22}$). Bajo estas condiciones, el Modelo Dual logró explicar más del 98% de la varianza observada, manteniendo coeficientes coherentes y una interpretación física consistente de las fases del algoritm

El análisis de coeficientes normalizados mostró que la fase de fusión adquiere un peso creciente a medida que aumenta el paralelismo (k), convirtiéndose en el principal factor limitante del rendimiento. Este resultado valida la hipótesis de que el costo real del algoritmo no está determinado únicamente por la cantidad de comparaciones, sino por el patrón de acceso a memoria inducido por la paralelización.

En conjunto, estos resultados sugieren que los modelos de complejidad tradicionales son útiles como guías estructurales, pero insuficientes para caracterizar el comportamiento real de algoritmos paralelos en arquitecturas contemporáneas. Para que un modelo empírico sea válido, es necesario restringir su dominio de aplicación y considerar la jerarquía de memoria como una de las variables crítica que condiciona la escalabilidad real.

Tabela 6: Coeficientes normalizados (Importancia relativa de las fases, T=4)

Modelo	**Constante**	β'_{sort}	β'_{merge}	β'_{run_class}
Solo Sort	-8.50×10^{-4}	1.0013	—	—
Solo Merge	≈ 0	—	0.9666	—
Dual	≈ 0	**0.5604**	**0.4264**	—
Triple	≈ 0	1.2958	-0.3220	0.0219

Tabela 7: Coeficientes del modelo dual ajustados por número de hilos (T)

Hilos (T)	R^2	**Constante**	β_{sort} **(ms)**	β_{merge} **(ms)**
2	0.9869	0.0667	6.94×10^{-7}	-5.00×10^{-6}
3	0.9841	0.0517	1.13×10^{-6}	-4.00×10^{-6}
4	0.9834	0.0136	5.49×10^{-7}	4.00×10^{-6}
5	0.9926	0.0451	9.34×10^{-7}	9.00×10^{-6}
6	0.9913	0.0278	7.49×10^{-7}	1.20×10^{-5}

A Implementación de SampleMerge

A continuación se presenta la función principal que orquesta las fases del algoritmo y realiza la captura de métricas temporales mediante la biblioteca `chrono`.

```
    template <size_t NUM_THREADS = 16, bool SWAP_PTR = false, typename T>
    SMTimeMetrics sampleMerge(T*& begin, T*& end) {
        size_t total_size = end - begin;
        if (total_size == 0) return {};

        const size_t chunk_size = (total_size + NUM_THREADS - 1) / NUM_THREADS;
        T* auxBuffer = new T[total_size];

        auto oldNumThreads = omp_get_max_threads();
        omp_set_num_threads(NUM_THREADS);

        // 1. Ordenamiento de bloques
        auto sortTimeStart = std::chrono::high_resolution_clock::now();
        sortChunksPhase<NUM_THREADS>(begin, end, chunk_size);
        auto sortTimeEnd = std::chrono::high_resolution_clock::now();

        // 2. Seleccion de pivotes
        auto pivotTimeStart = std::chrono::high_resolution_clock::now();
        auto pivots = pivotSelectionPhase<NUM_THREADS>(begin, end, chunk_size);
        auto pivotTimeEnd = std::chrono::high_resolution_clock::now();

        // 3. Clasificacion de Runs
        auto subpartTimeStart = std::chrono::high_resolution_clock::now();
        auto partitions = subpartitionPhase<NUM_THREADS>(begin, end, pivots, chunk_size);
        auto subpartTimeEnd = std::chrono::high_resolution_clock::now();

        // 4. Calcular Offsets (prefix sum)
        auto offsetTimeStart = std::chrono::high_resolution_clock::now();
        auto bucket_offsets = calculateBucketOffsets<NUM_THREADS>(partitions);
        auto offsetTimeEnd = std::chrono::high_resolution_clock::now();

        // 5. Fase de fusion
        auto mergeTimeStart = std::chrono::high_resolution_clock::now();
        mergePartitionsPhase<NUM_THREADS>(partitions, bucket_offsets, auxBuffer);
        auto mergeTimeEnd = std::chrono::high_resolution_clock::now();

```

```
        // Gestion de memoria y punteros de salida
        T* beginTmp = begin;
        begin = auxBuffer;
        end = auxBuffer + total_size;

        // Cleanup asincrono
        std::thread([beginTmp, delPar = std::move(partitions), delPiv = std::move(pivots)](){
            delete[] beginTmp;
        }).detach();

        omp_set_num_threads(oldNumThreads);

        return {
            std::chrono::duration<double, std::milli>(sortTimeEnd - sortTimeStart).count(),
            std::chrono::duration<double, std::milli>(pivotTimeEnd - pivotTimeStart).count(),
            std::chrono::duration<double, std::milli>(subpartTimeEnd - subpartTimeStart).count(),
            std::chrono::duration<double, std::milli>(offsetTimeEnd - offsetTimeStart).count(),
            std::chrono::duration<double, std::milli>(mergeTimeEnd - mergeTimeStart).count()
        };
    }

```

Listing 1: Implementación de la función principal SampleMerge con captura de métricas por fase.

B Calculo de VIF para modelo completo

```
    from statsmodels.stats.outliers_influence import variance_inflation_factor

    def calcular_vif_modelo_completo(df_granular):
        df = df_granular[df_granular['Threads'] > 0].copy()
        n = df['InputSize']
        k = df['Threads']

        df_vif = pd.DataFrame({
            'const': 1,
            'Sort': (n / k) * np.log2(n / k),
            'Pivots': k * np.log2(k),
            'K_Linear': k,
            'Merge': (n / k) * np.log2(k)
        })

        # Calculamos VIF por cada columna
        vif_data = pd.DataFrame()
        vif_data["Variable"] = df_vif.columns
        vif_data["VIF"] = [variance_inflation_factor(df_vif.values, i) for i in range(len(df_vif.
    columns))]

        return vif_data

    print(calcular_vif_modelo_completo(df_granular))


```

Listing 2: Calculo de VIF para modelo completo

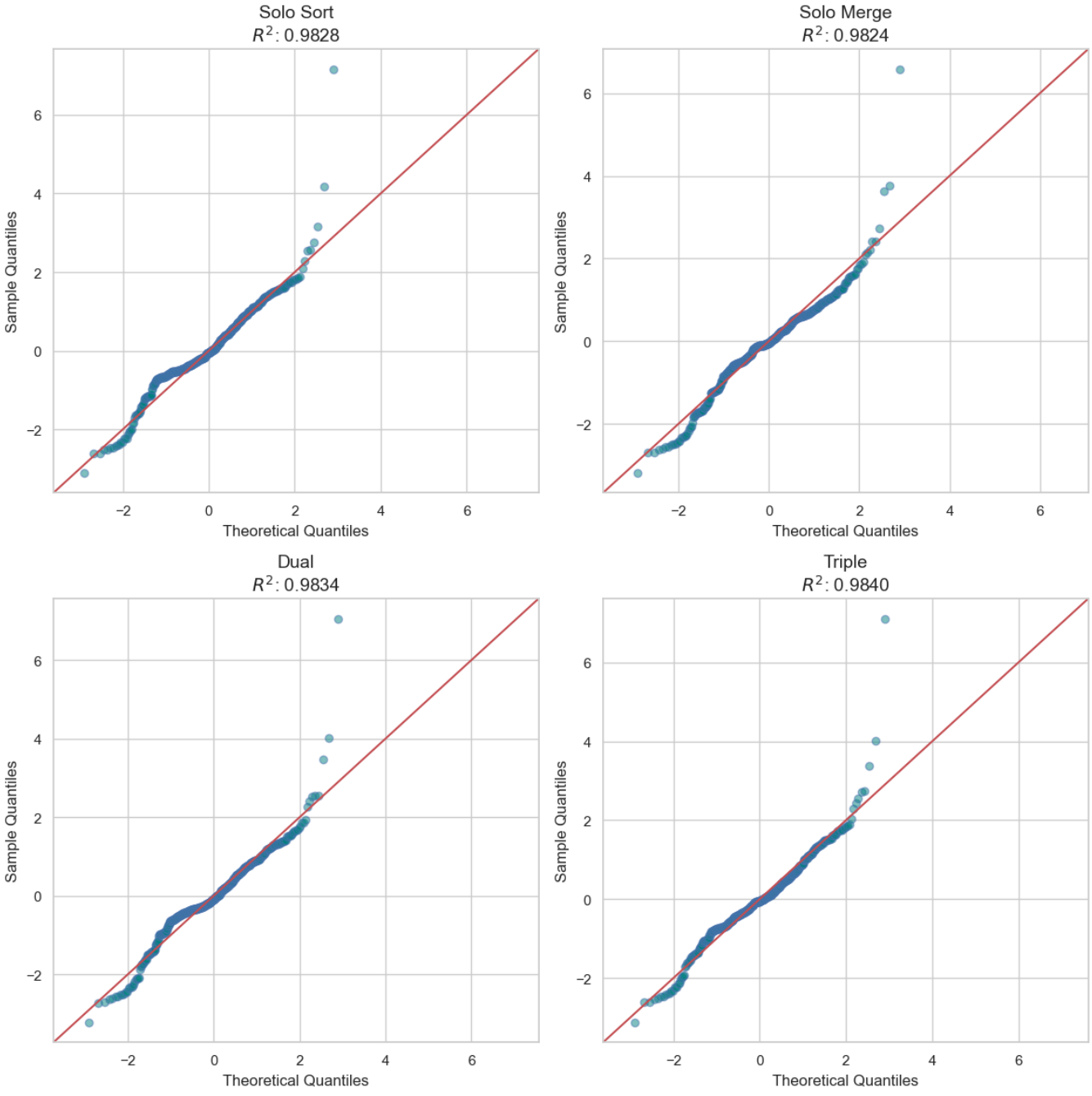

Figura 7: QQ-Plot de modelos de regresión propuestos (T=4)

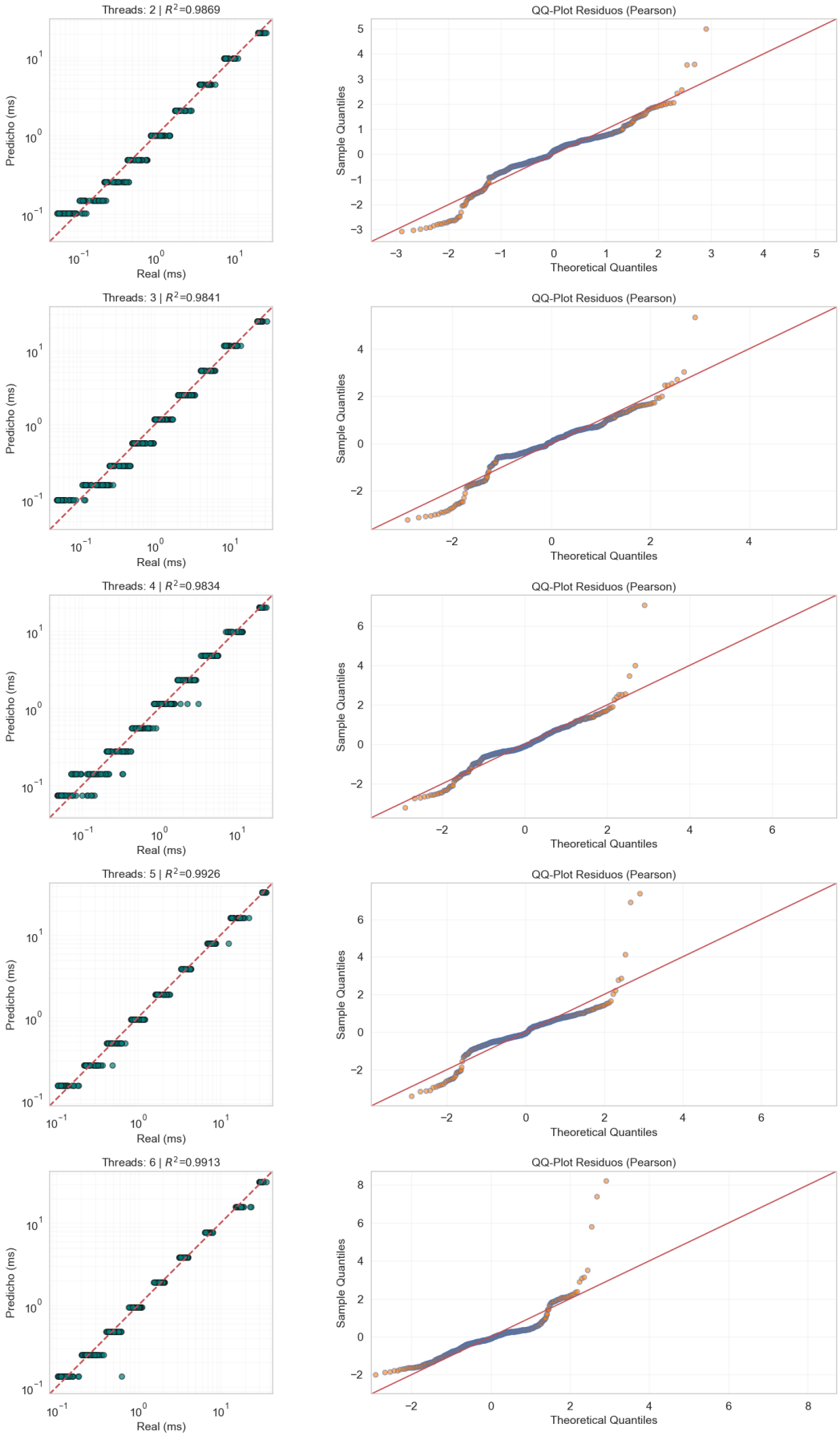

Figura 8: Resultados de la regresión dual