UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

CARRERA DE CIENCIAS DE LA COMPUTACIÓN

Paralelización y Análisis de Rendimiento del Algoritmo KNN con MPI

AUTORES

Chancan Chanca, Sanders [100 %]

Chamochumbi Gutierrez, Alexandro [100 %]

Osnayo Matos, Jose [100 %]

PROFESOR

Fiestas Iquira, Jose Antonio

Lima - Perú

2025

TABLA DE CONTENIDO

Pág.

I

Introducción

El algoritmo k-Nearest Neighbors (KNN) es un método clásico de clasificación supervisada ampliamente utilizado debido a su simplicidad y efectividad. Sin embargo, su costo computacional crece considerablemente con el tamaño del conjunto de datos, ya que para clasificar cada muestra de prueba es necesario calcular su distancia respecto a todas las muestras del conjunto de entrenamiento. Esto se traduce en una complejidad computacional del orden $O(n_{tr} \cdot n_{te})$, lo cual puede resultar impracticable en escenarios de grandes volúmenes de datos.

Con el objetivo de acelerar el procesamiento y mejorar el tiempo de respuesta, en este proyecto se plantea una solución paralela basada en el uso de la biblioteca `mpi4py`, que permite programación distribuida mediante el estándar MPI (Message Passing Interface). La idea principal consiste en dividir el conjunto de entrenamiento entre múltiples procesos, de manera que cada uno realice el cómputo de distancias de forma independiente y en paralelo.

El desarrollo del proyecto se gestionó mediante control de versiones en Git, en el repositorio:

https://github.com/Zanderz17/-CPD-Proyecto

El progreso se registró mediante iteraciones de código (versiones beta), donde se validaron los resultados paralelos frente a la versión secuencial, se incorporó el análisis teórico del rendimiento y, finalmente, se estudió la escalabilidad. Los principales hitos fueron:

- **23 de noviembre de 2025:** validación inicial del modelo paralelo con la versión secuencial.
- **24 de noviembre de 2025:** integración del modelo teórico y comparación con resultados experimentales.
- **27 de noviembre de 2025:** análisis de escalabilidad mediante speedup, eficiencia, FLOPs/s y determinación del número óptimo de procesos.

El diseño de paralelización se sustenta en el Diagrama de Dependencias (DAG) entregado en el laboratorio, que evidencia una alta proporción de tareas paralelizables. La comunicación se concentra en dos fases:

- distribución del conjunto de entrenamiento,
- recolección de vecinos locales para realizar la votación final.

La implementación utiliza las directivas de MPI **comm.bcast**, **comm.scatter** y **comm.gather**, lo que permite explotar el paralelismo de datos y mejorar el rendimiento general del algoritmo. Finalmente, se evalúa el desempeño obtenido en términos de tiempos de ejecución, comunicación, cómputo, precisión del modelo y comportamiento teórico, con el objetivo de determinar configuraciones paralelas eficientes.

II

Método

2.1 Descripción del algoritmo paralelo basado en el DAG

El diseño de la solución se basa en el Diagrama de Dependencias (DAG) mostrado en la Figura 2.1, proporcionado en el enunciado del laboratorio. En dicho DAG se observa que la fase dominante del algoritmo es el cálculo de distancias de KNN, el cual puede ejecutarse de manera completamente independiente por cada proceso. La comunicación solo ocurre en las etapas de distribución inicial de datos y de recolección final de resultados.

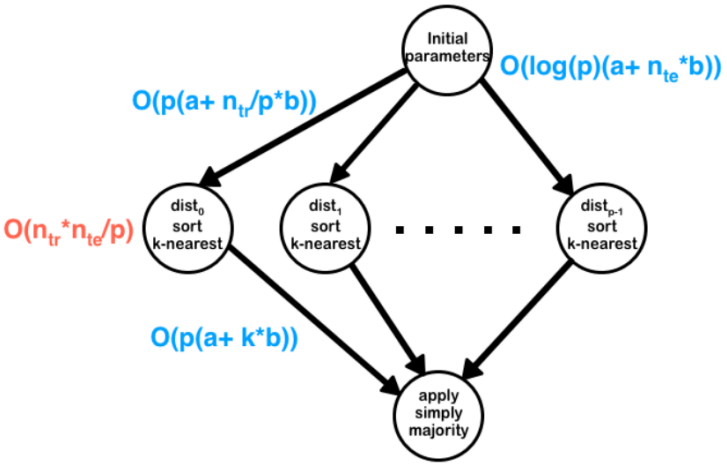


FIGURA 2.1: DAG de paralelización de KNN entregado en el enunciado.

A continuación, se presenta la implementación siguiendo esta estructura.

2.1.1 Inicialización de MPI y parámetros

Se inicializa el comunicador global y solo el proceso `rank = 0` procesa los argumentos, los cuales son difundidos a los demás procesos utilizando `MPI.Bcast`, tal como se muestra en las dependencias superiores del DAG.

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if rank == 0:
    parser = argparse.ArgumentParser()
    parser.add_argument("--n_samples", type=int, default=None)
    parser.add_argument("--k", type=int, default=3)
    parser.add_argument("--csv", type=str, default="knn_results.csv
        ")
    parser.add_argument("--m", type=int, default=1)
    args = parser.parse_args()
else:
    args = None

args = comm.bcast(args, root=0)
```

2.1.2 Distribución de datos (Scatter y Broadcast)

El proceso raíz carga y divide el conjunto de entrenamiento en p bloques que son distribuidos mediante `MPI.Scatter`. Por su parte, el conjunto de prueba se replica en todos los procesos usando `MPI.Bcast`, cumpliendo con la propagación de datos indicada en la parte superior del DAG.

```
if rank == 0:
    digits = load_digits()
    X = digits.data
```

```
    y = digits.target
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
    )
    X_train_chunks = np.array_split(X_train, size)
    y_train_chunks = np.array_split(y_train, size)
else:
    X_train_chunks = None
    y_train_chunks = None
    X_test = None
    y_test = None

X_train_local = comm.scatter(X_train_chunks, root=0)
y_train_local = comm.scatter(y_train_chunks, root=0)
X_test = comm.bcast(X_test, root=0)
y_test = comm.bcast(y_test, root=0)
```

2.1.3 Cálculo de vecinos locales en paralelo

Esta fase corresponde al paralelismo masivo del DAG. Cada proceso calcula las distancias entre todas las muestras del test y su partición local del entrenamiento, seleccionando los k vecinos más cercanos.

```
def local_knn_neighbors_batch(X_test, X_train_local, y_train_local,
    k):
    neighbors_all = []
    for x in X_test:
        dist = [euclidean_distance(x, t) for t in X_train_local]
        idx = np.argsort(dist)[:k]
        neighbors_all.append(
            [(dist[i], y_train_local[i]) for i in idx]
        )
    return neighbors_all
```

```
neighbors_local = local_knn_neighbors_batch(
    X_test, X_train_local, y_train_local, args.k
)
```

2.1.4 Recolección y clasificación final

Según el DAG, al final se produce la unificación de resultados: se reúnen todos los vecinos locales con `MPI.Gather` y se aplica la votación por mayoría.

```
neighbors_all = comm.gather(neighbors_local, root=0)

if rank == 0:
    global_neighbors = [[] for _ in range(len(X_test))]
    for proc_chunk in neighbors_all:
        for i, neigh_list in enumerate(proc_chunk):
            global_neighbors[i].extend(neigh_list)

    y_pred = []
    for neighs in global_neighbors:
        k_best = sorted(neighs, key=lambda x: x[0])[:args.k]
        labels = [lab for (_, lab) in k_best]
        y_pred.append(Counter(labels).most_common(1)[0][0])
```

2.1.5 Medición de tiempos y métrica de accuracy

Se mide:

- tiempo total paralelo,
- tiempo de cómputo local,

- tiempo de comunicación,
- *accuracy* del modelo.

Todos estos valores se reportan desde el proceso raíz para posterior análisis.

```
accuracy = np.mean(np.array(y_pred) == y_test)
print(f"Accuracy: {accuracy:.4f}")
```

III

Resultados y análisis

3.1 Configuración experimental

Los experimentos fueron ejecutados en el clúster de alto rendimiento **Khipu** (UTEC), utilizando nodos CPU administrados por `SLURM`. Para la ejecución del código MPI se cargaron los siguientes módulos del entorno:

```
module load gnu12/12.4.0
module load openmpi4/4.1.6
module load py3-mpi4py/3.1.3
```

Hardware utilizado

- CPU: Intel Xeon Gold 6130 @2.10 GHz (32 núcleos por nodo)
- Memoria: ¿90 GB DDR4
- Red de interconexión: InfiniBand FDR (baja latencia)

Dataset y parámetros utilizados

Se utilizó el dataset `load_digits` de `scikit-learn` compuesto por **1797** imágenes de dígitos escritas a mano, representadas como vectores de dimensión 64. Se evaluaron los siguientes tamaños del conjunto total de datos:

$$N = \{200, 500, 1000, 1500, 1797\}$$

Asimismo, se incrementó el número de procesos MPI:

$$p = \{1, 2, 4, 8, 16, 32, 64, 80\}$$

Para cada combinación (p, N) se realizaron **7 repeticiones** (`--m 7`), promediando los resultados para reducir la variabilidad asociada a ruido de sistema.

Esta decisión metodológica se sustenta en las recomendaciones de Hoefler y Belli [1], quienes establecen lineamientos científicos para la experimentación rigurosa en cómputo paralelo, garantizando interpretabilidad y confiabilidad estadística de los tiempos medidos.

Modo de ejecución

Los experimentos fueron ejecutados usando:

```
mpiexec --oversubscribe -n p python3 knn_digits_mpi_timed.py \
  --n_samples N --k 3 --m 7 --csv knn_results.csv
```

En todos los casos se midieron:

- tiempo total paralelo,
- tiempo de cómputo local,
- tiempo de comunicación,
- *accuracy* del modelo KNN.

3.2 Resultados de tiempos

En esta sección se muestran los tiempos obtenidos al ejecutar el algoritmo KNN paralelizado usando MPI. Se midió el tiempo total, el tiempo de cómputo y el tiempo de comunicación, para distintos tamaños de datos (n) y cantidad de procesos (p).

3.2.1 Tiempo total, tiempo de cómputo y tiempo de comunicación

En la Figura 3.1 se puede ver que el tiempo total disminuye cuando se aumenta la cantidad de procesos desde $p = 1$ hasta $p = 32$. Esto ocurre porque el costo principal del algoritmo es el cálculo de distancias, y al dividir los datos entre procesos se reduce el trabajo de cada uno.

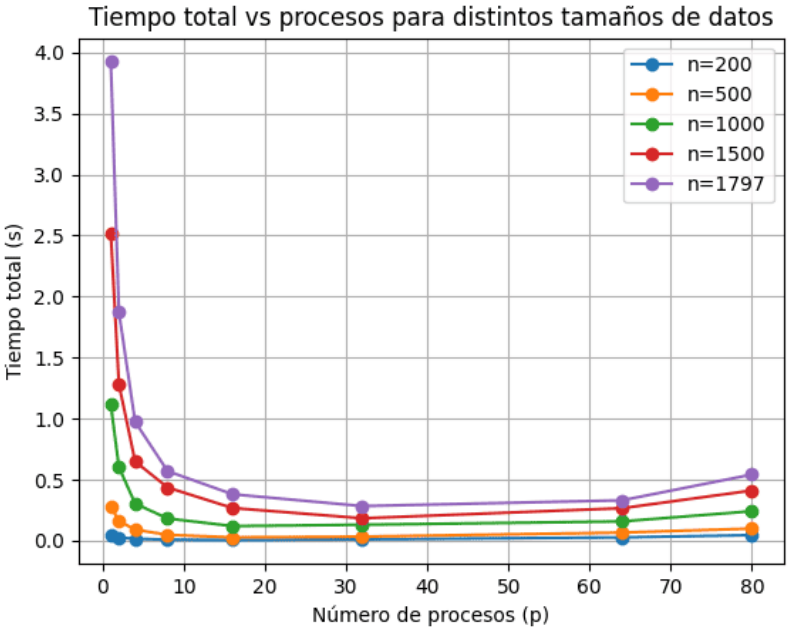


FIGURA 3.1: Tiempo total vs número de procesos para distintos tamaños de datos.

En la Figura 3.2 se observa que el tiempo de cómputo baja a medida que se agregan procesos. Sin embargo, después de $p = 32$, la reducción es mucho más pequeña, ya que cada proceso queda con una parte muy pequeña del entrenamiento.

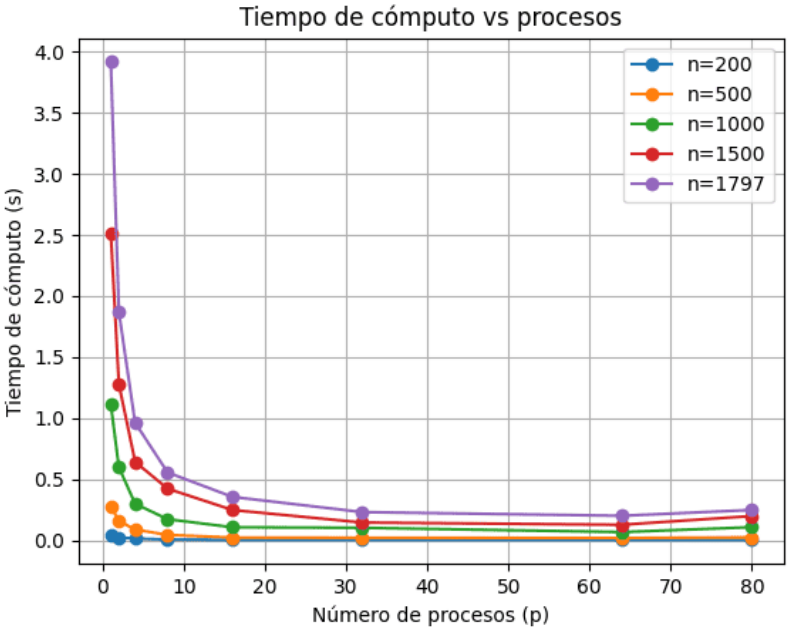


FIGURA 3.2: Tiempo de cómputo vs número de procesos.

A diferencia del cómputo, el tiempo de comunicación aumenta cuando se incrementa p, como se muestra en la Figura 3.3. Esto se debe a que se usan operaciones colectivas de MPI, como `scatter` y `gather`, que requieren enviar y recibir información entre todos los procesos.

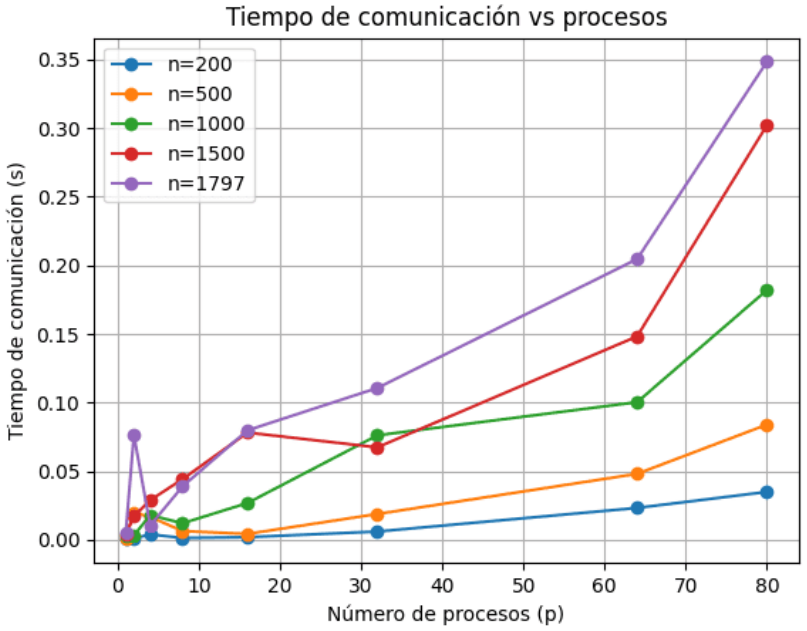


FIGURA 3.3: Tiempo de comunicación vs número de procesos.

3.2.2 Relación entre cómputo y comunicación

Los resultados muestran dos etapas muy claras:

- Cuando $p \leq 32$, el tiempo de cómputo baja bastante y la comunicación aún no afecta tanto el rendimiento. Por eso el tiempo total mejora mucho.
- Cuando $p \geq 64$, el costo de comunicación aumenta y empieza a afectar negativamente la ejecución, haciendo que el tiempo total vuelva a subir.

Esto concuerda con el comportamiento esperado según el DAG entregado, ya que la parte de cómputo es altamente paralelizable, pero la recolección final de vecinos requiere comunicación entre procesos.

3.2.3 Estimación del número óptimo de procesos

El mejor rendimiento se obtiene cuando el cómputo por proceso todavía es significativo, y la comunicación no es tan alta

De acuerdo con los resultados experimentales, el rango adecuado es:

$$32 \leq p_{\text{óptimo, exp}} \leq 64$$

El algoritmo presenta una buena escalabilidad en ese intervalo y empieza a perder eficiencia cuando se añaden más procesos, debido al mayor tiempo de comunicación necesario para sincronizar la información.

3.3 Precisión del modelo (Accuracy)

Además del análisis de rendimiento, se evaluó la precisión del modelo para verificar que la paralelización no afecte la calidad del clasificador KNN. En la Figura 3.4 se muestra la exactitud alcanzada para diferentes tamaños del conjunto de datos y distintos números de procesos p.

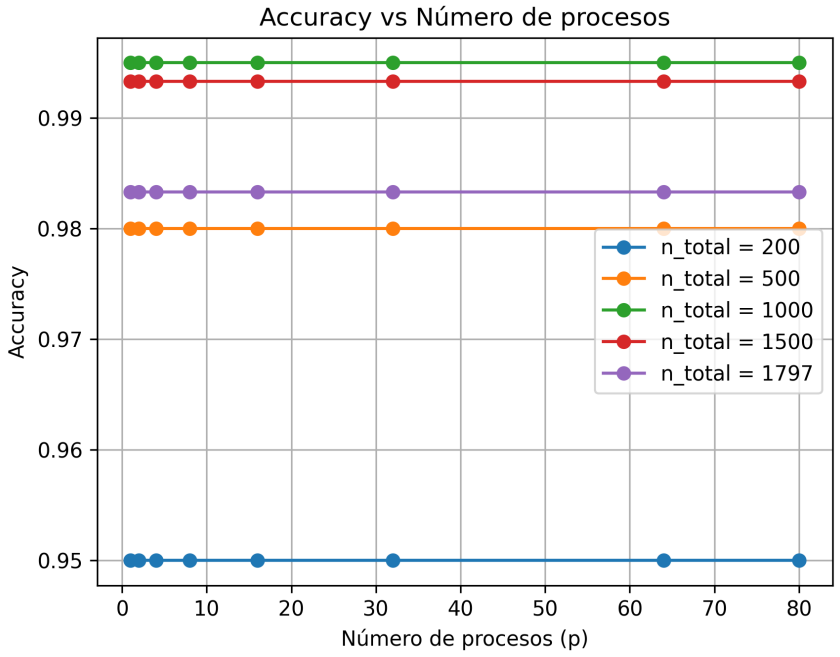


FIGURA 3.4: Exactitud del modelo KNN en función del número de procesos.

Los resultados muestran un comportamiento estable de la precisión:

- La exactitud depende principalmente del tamaño del dataset N y no del número de procesos.
- Para un mismo N, la precisión se mantiene constante para todos los valores de p evaluados.
- Los valores más altos se alcanzan para $N = 1000$ y $N = 1500$, superando el 99 %.
- El caso $N = 200$ presenta menor precisión ($\sim 95\,\%$), lo cual es esperable por la menor cantidad de datos disponibles para entrenamiento.

Este comportamiento es coherente con la teoría del algoritmo KNN:

- La paralelización solo divide los datos entre procesos para acelerar el cálculo, pero el resultado final sigue siendo el mismo conjunto global de vecinos más cercanos.
- Por tanto, el clasificador final es equivalente al secuencial.

Por lo tanto, la paralelización del algoritmo no afecta la calidad del modelo.

3.4 Validación con el modelo teórico

En esta sección se valida el comportamiento del algoritmo paralelo utilizando un modelo teórico que se obtiene sumando los costos indicados en el DAG del algoritmo KNN distribuido.

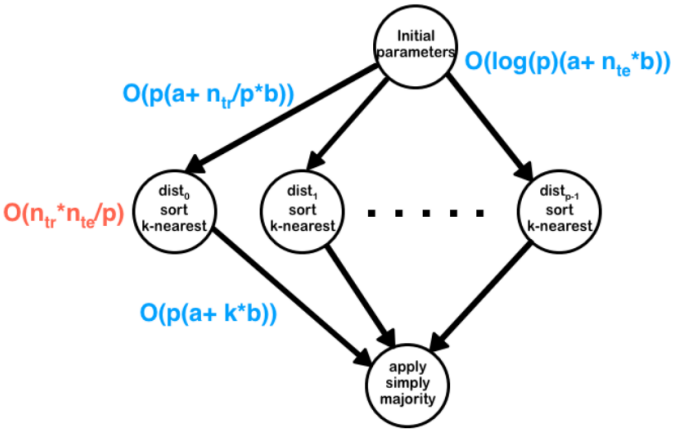


FIGURA 3.5: DAG del algoritmo KNN paralelo con los costos por etapa.

3.4.1 Derivación de la complejidad esperada a partir del DAG

En el DAG de la Figura 3.5 se indican los siguientes costos asintóticos:

- Difusión de parámetros iniciales:

$$T_{\text{init}}(p) = O\big(\log(p)\,\big(a + n_{te}\,b\big)\big)$$

- Cálculo de distancias y selección de vecinos en cada proceso: cada proceso trabaja con aproximadamente n_{tr}/p elementos de entrenamiento y con los n_{te} elementos de prueba, por lo que el costo total es:

$$T_{\text{dist}}(p) = O\Big(a\,\frac{n_{tr}}{p}\,n_{te}\Big)$$

- Ordenamiento local de los k vecinos para cada elemento de prueba:

$$T_{\text{ord}}(p) = O\big(n_{te}\,k\,b\big)$$

- Recolección de vecinos y decisión por mayoría entre procesos:

$$T_{\text{maj}}(p) = O\big(p\,(a + k\,b)\big)$$

Sumando todos los términos se obtiene el costo total en función del número de procesos p:

$$\begin{aligned} T_p &= T_{\text{init}}(p) + T_{\text{dist}}(p) + T_{\text{ord}}(p) + T_{\text{maj}}(p) \\ &= O\big(\log(p)\,(a + n_{te}b)\big) + O\Big(a\,\frac{n_{tr}}{p}\,n_{te}\Big) + O\big(n_{te}k\,b\big) + O\big(p\,(a + k\,b)\big). \end{aligned}$$

Para el dataset `load_digits`, los valores a, b y k son constantes de implementación y la parte $n_{te}k\,b$ no depende de p. Agrupando constantes se puede reescribir como:

$$T_p(n_{tr}, n_{te}, p) \approx \underbrace{\alpha \frac{n_{tr}\, n_{te}}{p}}_{\text{cómputo}} + \underbrace{\beta\, p}_{\text{costos lineales de comunicación}} + \underbrace{\gamma\, \log_2(p)}_{\text{colectivas (broadcast/reduce)}} .$$

Renombrando α, β, γ como a, b, c, se obtiene el modelo que se usó en los ajustes:

$$\boxed{T(p) = a \frac{n_{tr} n_{te}}{p} + b\, p + c\, \log_2(p)}$$

donde:

- $a \frac{n_{tr} n_{te}}{p}$: representa el tiempo de cómputo paralelo (distancias y selección de vecinos) que decrece al aumentar p.
- $b\, p$: agrupa la parte de comunicación que crece de forma lineal con el número de procesos (por ejemplo, recolección y envío de vecinos).
- $c\, \log_2(p)$: modela las operaciones colectivas que se escalan de forma logarítmica con p (por ejemplo, difusiones de parámetros).

3.4.2 Ajuste del modelo a los datos experimentales

Para cada tamaño de dataset N se tienen medidas experimentales de tiempo de cómputo $T_{\text{exp}}(p)$ para $p \in \{1, 2, 4, 8, 16, 32, 64, 80\}$. Con estos datos se ajusta el modelo anterior mediante mínimos cuadrados.

Para un N fijo, se construye la matriz:

$$X = \begin{bmatrix} \dfrac{n_{tr}n_{te}}{p_1} & p_1 & \log_2(p_1) \\ \dfrac{n_{tr}n_{te}}{p_2} & p_2 & \log_2(p_2) \\ \vdots & \vdots & \vdots \\ \dfrac{n_{tr}n_{te}}{p_m} & p_m & \log_2(p_m) \end{bmatrix}, \quad \vec{T}_{\text{exp}} = \begin{bmatrix} T_{\text{exp}}(p_1) \\ T_{\text{exp}}(p_2) \\ \vdots \\ T_{\text{exp}}(p_m) \end{bmatrix}.$$

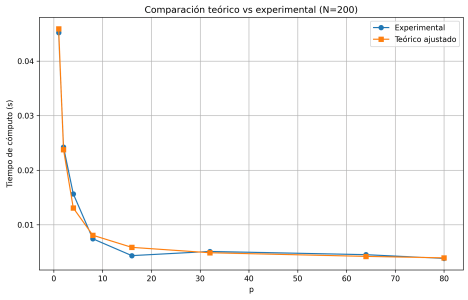
El vector de parámetros $\theta = (a, b, c)^\top$ se obtiene resolviendo

$$\theta^* = \arg\min_{\theta} \left\| X\theta - \vec{T}_{\text{exp}} \right\|_2^2,$$

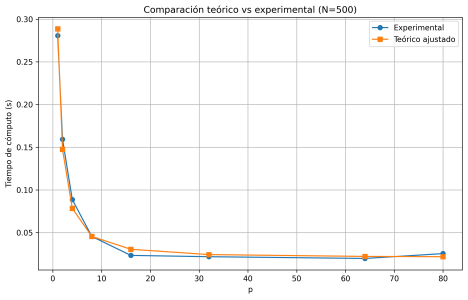
lo que corresponde a la solución de mínimos cuadrados (usada en el script de Python con `np.linalg.lstsq`).

3.4.3 Comparación entre tiempo teórico y experimental

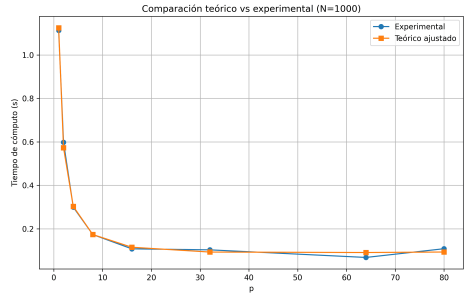
En la Figura 3.6 se muestra la comparación entre el tiempo de cómputo experimental y el tiempo entregado por el modelo ajustado para cada tamaño de dataset.



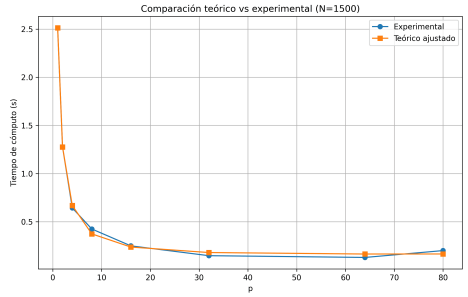
((A)) $N = 200$



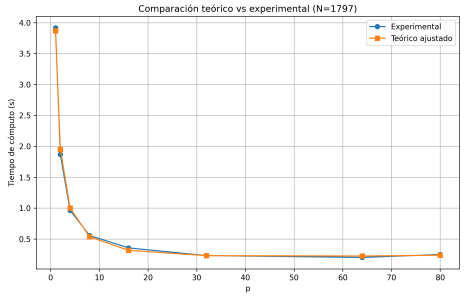
((B)) $N = 500$



((C)) $N = 1000$



((D)) $N = 1500$



((E)) $N = 1797$

FIGURA 3.6: Comparación entre tiempo de cómputo experimental y modelo teórico ajustado para distintos tamaños de datos.

En todos los casos la curva teórica sigue bien la forma de los datos medidos: el tiempo baja rápidamente al pasar de $p = 1$ a $p = 8$–16 y luego se estabiliza. Esto indica que el término de cómputo $a\,\dfrac{n_{tr} n_{te}}{p}$ domina al inicio, mientras que para valores más grandes de p la parte de comunicación $b\,p + c\log_2(p)$ empieza a ser más importante.

3.4.4 Estimación del número óptimo de procesos

El modelo teórico permite aproximar el número de procesos que minimiza el tiempo de ejecución. La derivada exacta de

$$T(p) = a\,\frac{n_{tr}n_{te}}{p} + b\,p + c\,\log_2(p)$$

es

$$\frac{dT}{dp} = -a\,\frac{n_{tr}n_{te}}{p^2} + b + \frac{c}{p\ln 2}.$$

Para obtener una expresión simple de p_{opt}, se puede ignorar el término logarítmico en la derivada (el aporte de $\frac{c}{p\ln 2}$ es pequeño frente a los otros dos términos en el rango de p considerado) y resolver

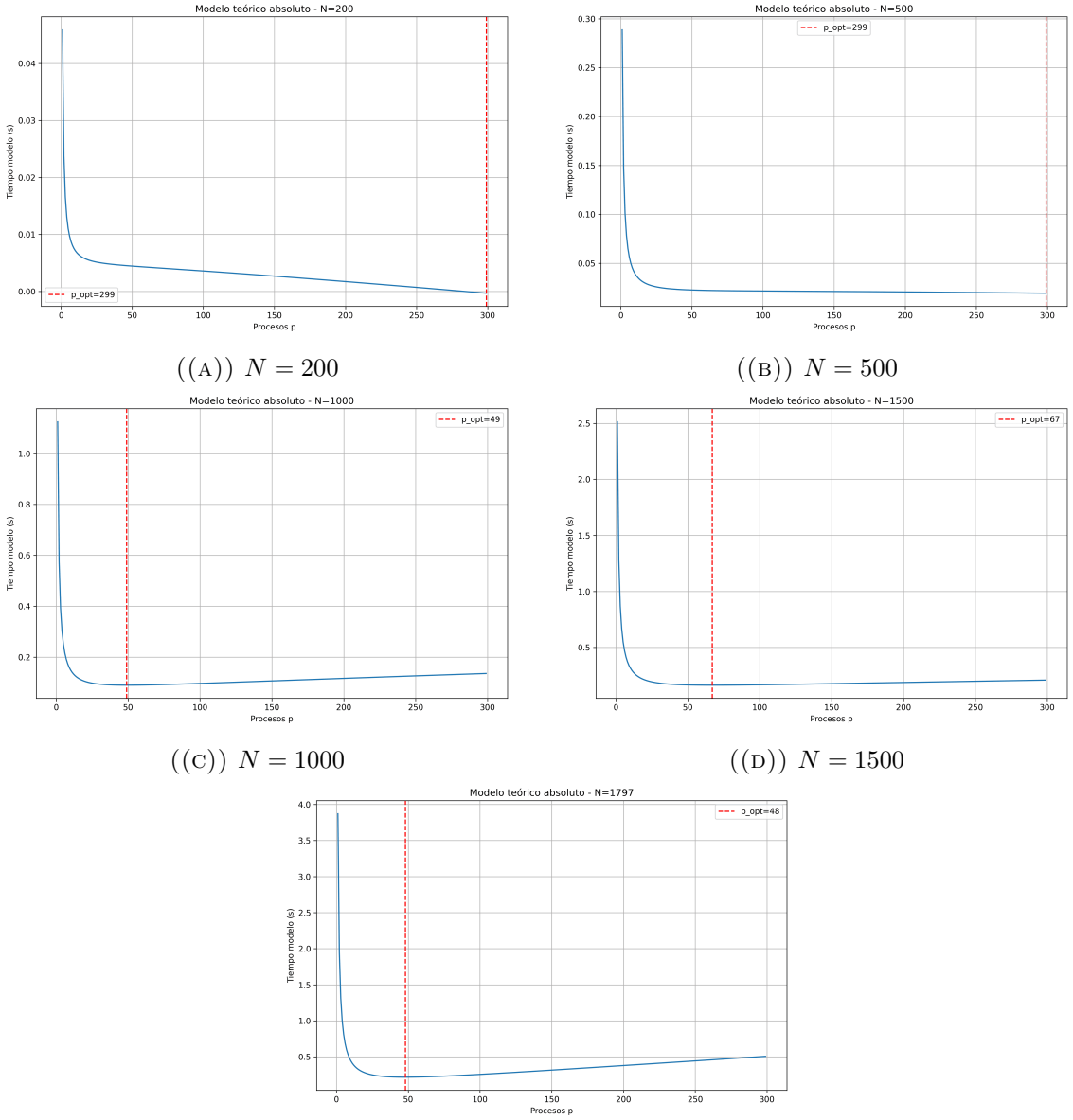
$$-a\,\frac{n_{tr}n_{te}}{p^2} + b = 0 \quad \Rightarrow \quad \boxed{p_{\text{opt}} \approx \sqrt{\frac{a\,n_{tr}n_{te}}{b}}}$$

A partir de los parámetros ajustados para cada N, el script calculó $T(p)$ para $p = 1, \ldots, 299$ y eligió el mínimo exacto sobre ese rango. Los valores obtenidos fueron coherentes con esta aproximación. En el caso del conjunto completo ($N = 1797$) se obtuvo

$$p_{\text{opt}} \approx 48,$$

lo que coincide con el mejor rendimiento observado entre $p = 32$ y $p = 64$ en los experimentos.

La Figura 3.7 muestra las curvas del modelo y la posición del mínimo teórico para cada tamaño de datos.



((A)) $N = 200$

((B)) $N = 500$

((C)) $N = 1000$

((D)) $N = 1500$

((E)) $N = 1797$

FIGURA 3.7: Curvas del modelo teórico y posición del mínimo p_{opt} para distintos tamaños de datos.

En resumen, para el dataset completo la región de mejor rendimiento se encuentra en el intervalo

$$32 \leq p \leq 64,$$

lo que es consistente con el valor teórico $p_{\text{opt}} \approx 48$ obtenido a partir del modelo ajustado.

Comportamiento del modelo en tamaños pequeños de datos

En los casos donde el tamaño del dataset es reducido ($N = 200$ o $N = 500$), se observó que el modelo ajustado predice valores de p_{opt} muy grandes (por ejemplo, $p_{\text{opt}} \approx 299$ dentro del rango evaluado). Esto se debe a que, al derivar el modelo:

$$T(p) = a\,\frac{n_{tr}n_{te}}{p} + b\,p + c\,\log_2(p), \qquad \frac{dT}{dp} = -a\,\frac{n_{tr}n_{te}}{p^2} + b + \frac{c}{p\ln 2},$$

el término de cómputo $-a\,\frac{n_{tr}n_{te}}{p^2}$ es muy pequeño cuando n_{tr} y n_{te} son bajos. Es decir:

$$n_{tr} \cdot n_{te} \text{ es muy pequeño} \quad \Rightarrow \quad \left|-a\,\frac{n_{tr}n_{te}}{p^2}\right| \ll b$$

Entonces, para todo p en el rango medido:

$$\frac{dT}{dp} > 0,$$

lo que implica que el tiempo no disminuye al aumentar el número de procesos. Es decir, el modelo indica que no es conveniente paralelizar en estos tamaños.

Este comportamiento coincide con lo reportado en la literatura: cuando el trabajo por proceso es muy pequeño, el paralelismo pierde eficiencia porque el tiempo de comunicación y sincronización empieza a ser dominante respecto al cálculo útil [2]. Además, el tiempo de arranque de operaciones colectivas y el ruido del sistema operativo se vuelve comparable al tiempo de cómputo real [3].

Por lo tanto, para tamaños pequeños de dataset, el modelo ajustado no debe usarse como criterio de optimización, ya que ejecutar con muchos procesos solo incrementa la sobrecarga. En cambio, cuando el dataset es suficientemente grande ($n_{tr} \cdot n_{te} \gg p$), el modelo describe bien el comportamiento experimental y predice un p_{opt} adecuado para la ejecución en el clúster.

3.5 Speedup y eficiencia

3.5.1 Cálculo del speedup

Para evaluar la ganancia obtenida con el paralelismo se calculó el *speedup* como

$$S(p, n) = \frac{T_{\text{seq}}(n)}{T_{\text{par}}(p, n)},$$

donde $T_{\text{seq}}(n)$ es el tiempo de ejecución secuencial para un tamaño de problema n y $T_{\text{par}}(p, n)$ es el tiempo total del algoritmo paralelo usando p procesos. La Figura 3.8 muestra el speedup medido para los distintos tamaños de datos.

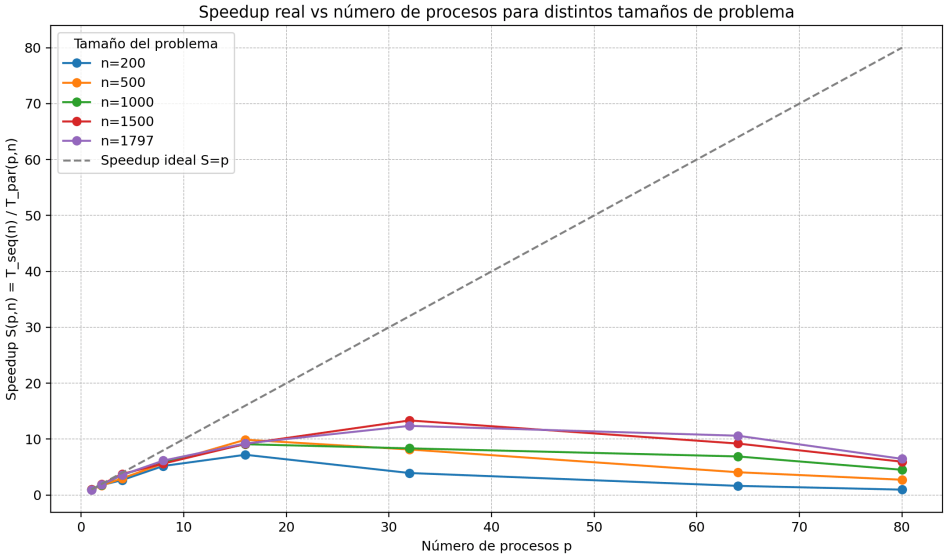


FIGURA 3.8: Speedup real $S(p, n)$ vs número de procesos para distintos tamaños de problema.

A partir de la figura se observa que:

- Para los tamaños grandes ($n = 1500$ y $n = 1797$), el speedup crece casi de forma lineal hasta aproximadamente $p = 16$. En este rango el tiempo paralelo está claramente dominado por el cómputo, por lo que dividir el trabajo entre más procesos reduce el tiempo de manera efectiva.

- El máximo speedup para el dataset completo se alcanza alrededor de $p = 32$, donde se obtiene una aceleración cercana a $S \approx 12$. A partir de este punto, añadir más procesos ya no mejora el tiempo y el speedup incluso empieza a bajar.

- Para tamaños pequeños ($n = 200$ y $n = 500$), el speedup se satura mucho antes. Después de $p = 8$–16, el tiempo de comunicación y la falta de trabajo por proceso hacen que el rendimiento deje de mejorar.

En resumen, el algoritmo escala bien mientras cada proceso tiene suficiente trabajo de cómputo (región hasta $p \approx 16$–32 para los tamaños grandes). Más allá de ese punto, el costo de comunicación empieza a dominar y el speedup se aleja del ideal $S(p) = p$ (línea punteada de la figura).

3.5.2 Eficiencia y escalabilidad

La eficiencia paralela se definió como

$$E(p, n) = \frac{S(p, n)}{p} = \frac{T_{\text{seq}}(n)}{p\, T_{\text{par}}(p, n)},$$

que mide qué fracción del rendimiento ideal se está aprovechando cuando se usan p procesos. La Figura 3.9 resume los valores de eficiencia para todos los tamaños de problema.

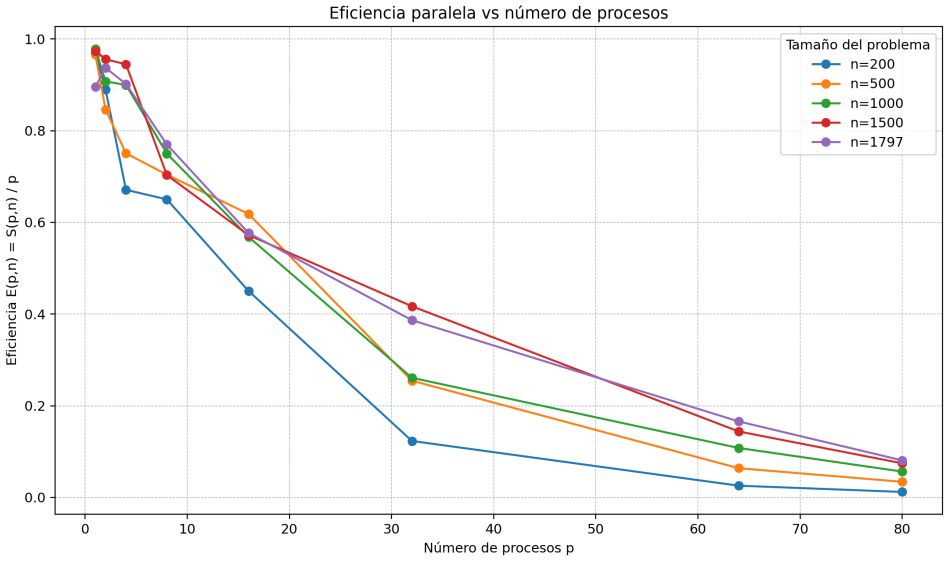


FIGURA 3.9: Eficiencia paralela $E(p, n)$ vs número de procesos para distintos tamaños de problema.

De estos resultados se puede destacar lo siguiente:

- Para $p = 2$ y $p = 4$, la eficiencia es alta en todos los tamaños de datos (cerca de $E \approx 0.9$ para los casos grandes), lo que indica que casi se aprovecha el rendimiento ideal.
- A medida que se aumenta p, la eficiencia disminuye de forma progresiva. Para el dataset completo, todavía se mantiene en un nivel aceptable alrededor de $p = 16$–32, mientras que para $p \geq 64$ la eficiencia cae por debajo de 0.2, lo que refleja un uso poco efectivo de los recursos.
- En los tamaños pequeños ($n = 200$), la eficiencia cae mucho más rápido: al llegar a $p = 32$ ya es muy baja. Esto confirma que no vale la pena usar muchos procesos cuando el problema tiene pocos datos.

En términos de escalabilidad fuerte, los resultados muestran que el KNN paralelo escala bien hasta un tamaño moderado de procesos (aprox. $p = 16$–32 para el conjunto completo). A partir de ahí, el aumento del número de procesos solo incrementa la comunicación y la sobrecarga de sincronización, mientras que el trabajo de cómputo por proceso se hace demasiado pequeño.

Por eso, combinando speedup y eficiencia, el intervalo

$$32 \leq p \leq 64$$

aparece como el rango razonable para ejecutar el algoritmo en el clúster con el dataset completo: se obtiene una aceleración importante sin que la eficiencia caiga a valores demasiado bajos.

3.6 Cálculo y análisis de FLOPs

El cálculo de la distancia euclidiana entre dos vectores de dimensión d implica tres operaciones por componente:

$$\text{FLOPs por componente} = \underbrace{(x_i - y_i)}_{\text{resta}} + \underbrace{(x_i - y_i)^2}_{\text{multiplicación}} + \underbrace{\text{acumulación}}_{\text{suma}} = 3$$

Por tanto, el costo de una distancia completa es:

$$\text{FLOPs}_{\text{dist}} = 3d$$

En el dataset `load_digits`, $d = 64$, por lo que:

$$\text{FLOPs}_{\text{dist}} = 3 \cdot 64 = 192$$

Cada proceso calcula distancias entre su parte del entrenamiento (n_{tr}/p) y todos los elementos de prueba (n_{te}), entonces:

$$\text{FLOPs totales} = n_{te} \cdot \frac{n_{tr}}{p} \cdot 192$$

El rendimiento en operaciones por segundo es:

$$\text{FLOPs/s} = \frac{\text{FLOPs totales}}{T_{\text{compute}}(p, n)}$$

donde $T_{\text{compute}}(p, n)$ es el tiempo de cómputo medido experimentalmente (sin considerar comunicación).

3.6.1 FLOPs por segundo vs número de procesos

La Figura 3.10 presenta los resultados obtenidos para distintos tamaños del dataset.

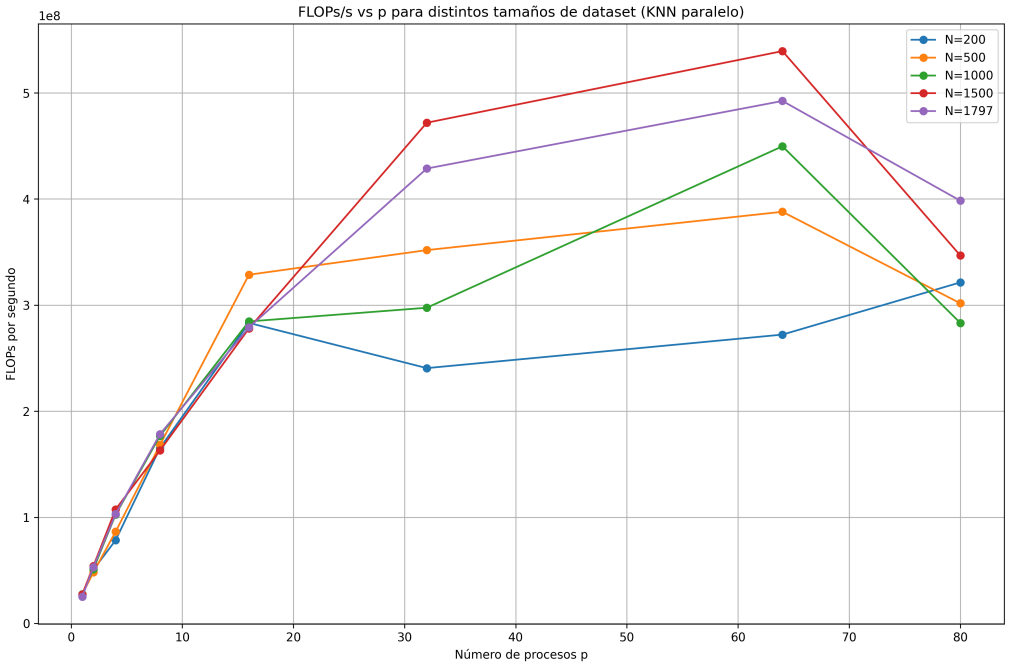


FIGURA 3.10: FLOPs/s vs número de procesos para distintos tamaños de dataset.

3.6.2 Análisis del rendimiento basado en FLOPs

De la figura se observa que:

- El rendimiento mejora al aumentar p cuando el dataset es suficientemente grande ($N \geq 1000$).

- Para $N = 1500$ y $N = 1797$, el rendimiento máximo se alcanza en el intervalo $32 \leq p \leq 64$, logrando valores cercanos a 5.3×10^8 FLOPs/s.
- Cuando $p > 64$, el rendimiento cae debido a que el tiempo de comunicación aumenta y el tiempo de cómputo es cada vez menor.
- En tamaños pequeños ($N = 200$), hay poco trabajo por proceso, por lo que el paralelismo no se aprovecha y el rendimiento disminuye con valores altos de p.

En conclusión, el análisis de FLOPs/s confirma lo observado previamente en el speedup y la eficiencia:

El paralelismo beneficia mientras el trabajo por proceso sea suficiente.

Para el dataset completo ($N = 1797$):

$$32 \leq p_{\text{óptimo}} \leq 64,$$

lo que coincide con la estimación del modelo teórico y con la mejoría experimental en tiempos y rendimiento.

3.7 Escalabilidad con tamaño variable del problema

Se evaluó también la escalabilidad con un tamaño de problema variable, es decir, cómo se comporta el algoritmo cuando se incrementa el número de procesos y el volumen de datos.

Inicialmente se realizaron pruebas sobre un dataset mayor (`MNIST`) con:

$$N = \{5000, 10000, 20000\}$$

En estos experimentos se observó que el tiempo total disminuye al aumentar el número de procesos, pero la mejora se vuelve cada vez menor a partir de $p = 32$, especialmente cuando se llega a $p = 64$, donde el costo de comunicación empieza a dominar sobre el cómputo.

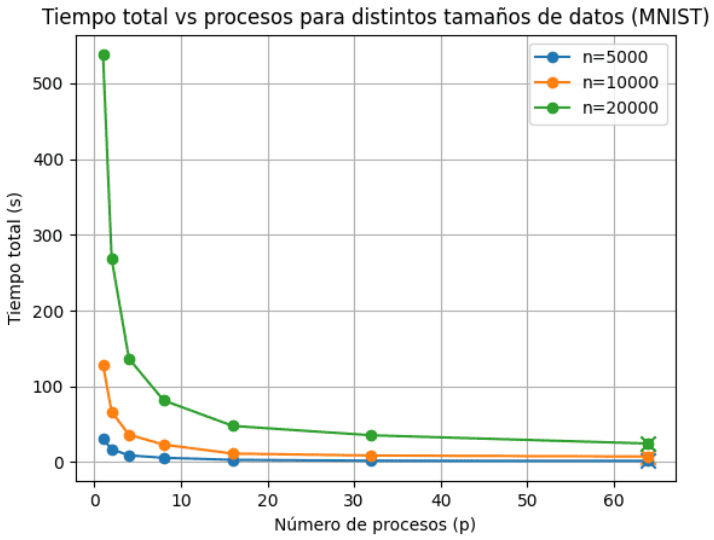


FIGURA 3.11: Tiempo total vs número de procesos para distintos tamaños de datos en `MNIST`.

Debido a que estas ejecuciones en `MNIST` no mostraban una mejora tan grande al seguir aumentando p, se decidió enfocar el análisis detallado de escalabilidad en el dataset `digits` que sugería el enunciado del laboratorio, que permitió estudiar mejor la relación entre cómputo y comunicación.

Análisis teórico de escalabilidad

A partir del modelo obtenido del DAG:

$$T_p(n_{tr}, n_{te}, p) \approx \alpha \, \frac{n_{tr} n_{te}}{p} + \beta \, p + \gamma \log_2(p),$$

definimos el tamaño del problema como:

$$W = n_{tr} \cdot n_{te}.$$

El tiempo secuencial se puede aproximar como:

$$T_1(W) \approx \alpha W.$$

Entonces, la eficiencia paralela queda:

$$E(W, p) = \frac{T_1(W)}{p \, T_p(W)} = \frac{\alpha W}{\alpha W + \beta p^2 + \gamma p \log_2(p)}.$$

Si queremos mantener una eficiencia aproximadamente constante $E(W, p) = E_0$, el término dominante del denominador debe crecer en la misma escala que el numerador. Para valores moderados de p, la parte que más crece con p es βp^2, de modo que:

$$\alpha W \sim \beta p^2 \quad \Rightarrow \quad \boxed{W = \Theta(p^2)}.$$

En nuestro caso práctico:

$$n_{tr} + n_{te} = N \quad \text{y normalmente} \quad n_{tr} \approx n_{te} \approx \frac{N}{2},$$

por lo que:

$$W = n_{tr} n_{te} \approx \frac{N^2}{4} \quad \Rightarrow \quad N^2 = \Theta(p^2) \quad \Rightarrow \quad \boxed{N = \Theta(p)}.$$

Es decir, para mantener una eficiencia similar al escalar, el tamaño del dataset total N debería crecer de forma aproximadamente lineal con el número de procesos p.

Análisis experimental de la escalabilidad

Los resultados con el dataset `digits` son coherentes con este análisis teórico:

- Para tamaños pequeños ($N = 200$ y $N = 500$):
  - El speedup crece muy poco.
  - La eficiencia cae rápido al aumentar p.
  - El costo de comunicación y el overhead de MPI se vuelven comparables o mayores que el cómputo local.

  En términos del modelo, esto se refleja en que el mínimo teórico de $T(p)$ cae en $p < 1$, es decir, lo mejor sería no paralelizar.
- Para tamaños medianos y grandes en `digits` ($N = 1000$, $N = 1500$, $N = 1797$):
  - El **speedup** mejora de forma clara hasta $p = 32$.
  - La **eficiencia** no decae mucho.

    - Los **FLOPs/s** aumentan hasta cerca de $p = 64$, lo que indica buen aprovechamiento del hardware.

- A partir de $p \geq 64$:
    - El beneficio adicional es pequeño o nulo.
    - El tiempo total empieza a subir, porque la comunicación global y la sincronización entre procesos dominan el costo total.

El modelo teórico ajustado confirmó que, para el dataset completo ($N = 1797$), el rango de procesos más adecuado se encuentra entre:

$$32 \leq p_{\text{opt}} \leq 64,$$

con un valor teórico aproximado de:

$$p_{\text{opt}} \approx 48,$$

lo cual coincide con el mejor comportamiento observado en las gráficas de tiempo, speedup, eficiencia y FLOPs.

El algoritmo KNN muestra buena escalabilidad siempre que el tamaño del problema crezca de forma proporcional al número de procesos. En ese régimen, el cómputo domina sobre la comunicación y se obtienen mejoras claras en el tiempo de ejecución. Sin embargo, cuando el problema es pequeño o se usan demasiados procesos, la comunicación y el overhead asociado superan al cómputo útil y la escalabilidad se pierde.

IV

Conclusiones

En este proyecto se implementó y analizó una versión paralela del algoritmo k-Nearest Neighbors (KNN) utilizando `mpi4py` sobre el clúster Khipu. A partir de los resultados obtenidos y del modelo teórico desarrollado, se pueden extraer las siguientes conclusiones principales.

Conclusiones técnicas

- El KNN paralelo presenta una buena mejora de rendimiento para tamaños de problema medianos y grandes. Para el dataset completo de `digits` ($N = 1797$), se alcanzó un speedup cercano a 12x con $p = 32$, manteniendo una eficiencia razonable y un alto número de FLOPs/s.
- La precisión del modelo (accuracy) se mantuvo prácticamente constante para todos los valores de p y solo dependió del tamaño del dataset. Esto indica que la paralelización fue correcta y no modificó el resultado del clasificador respecto a la versión secuencial.
- El rango de procesos adecuado para el problema estudiado se sitúa aproximadamente entre

  $$32 \leq p_{\text{óptimo}} \leq 64,$$

  con un valor teórico de referencia $p_{\text{opt}} \approx 48$. Fuera de ese rango, especialmente para $p \geq 64$, el costo de comunicación crece y el tiempo total vuelve a aumentar.

- Para tamaños pequeños de dataset ($N = 200$ y $N = 500$) la paralelización no es recomendable: el speedup se satura muy rápido y la eficiencia cae de forma notable. En estos casos, el trabajo por proceso es tan bajo que la comunicación y el overhead de MPI superan al cómputo útil.

- El modelo teórico
$$T(p) = a\,\frac{n_{tr}n_{te}}{p} + b\,p + c\,\log_2(p)$$
ajustado a partir del DAG describe bien el comportamiento observado para tamaños de problema grandes. Las curvas teóricas siguen la forma de las mediciones y permiten estimar el número de procesos que minimiza el tiempo de ejecución.

- El análisis de FLOPs/s confirmó las conclusiones de speedup y eficiencia: el rendimiento crece con p mientras cada proceso mantiene una carga de trabajo razonable; cuando el problema se reparte demasiado, el rendimiento cae porque el tiempo de cómputo deja de ser el factor dominante.

Conclusiones sobre escalabilidad

- A partir del modelo teórico y del análisis de eficiencia, se obtuvo que, para mantener una eficiencia aproximadamente constante, el tamaño del problema debe crecer como
$$W = n_{tr}n_{te} = \Theta(p^2) \quad \Rightarrow \quad N = \Theta(p),$$
asumiendo $n_{tr} \approx n_{te} \approx N/2$. Es decir, para sostener la escalabilidad es necesario incrementar el número de datos de forma proporcional al número de procesos.

- Las pruebas adicionales con `MNIST` mostraron que, aunque el tiempo total disminuye al aumentar p, la mejora se vuelve limitada para p grandes. Esto refuerza la idea de que el paralelismo debe balancear cuidadosamente cómputo y comunicación, incluso cuando se trabaja con más datos.

Aprendizajes del curso y del proyecto

- El proyecto permitió reforzar los conceptos del curso de Computación Paralela y Distribuida de manera práctica: se trabajó con modelos de tiempo, speedup, eficiencia, escalabilidad fuerte y débil, así como con la interpretación de curvas teóricas frente a resultados reales.
- Se ganó experiencia directa usando MPI en un clúster real, configurando módulos de compilación y lidiando con tiempos de ejecución, sobrecargas y ruido del sistema que no aparecen en ejemplos simples.
- Aunque en clase la mayoría de ejemplos se desarrollaron en C++, este proyecto permitió aplicar los mismos conceptos usando Python y la biblioteca `mpi4py`. Esto ayudó a ver que las ideas de paralelismo (DAG, costos de comunicación, partición de datos) son independientes del lenguaje y pueden trasladarse a diferentes entornos.
- El trabajo con scripts de medición, generación de CSV, gráficos en `matplotlib` y ajuste de modelos con `NumPy/pandas` fortaleció el manejo de la metodología experimental en HPC: repetición de pruebas, promedio de tiempos, control de parámetros y análisis cuidadoso de resultados, complementado con la **revisión de bibliografía** especializada (como Hoefler y Belli, Grama et al.), lo que permitió fundamentar las decisiones experimentales según prácticas reconocidas en computación paralela y distribuida.

- Finalmente, el desarrollo incremental (versiones beta, validación contra el código secuencial, comparación con el modelo teórico y análisis de escalabilidad) aportó una visión completa del ciclo de vida de un experimento en computación paralela: desde el diseño del algoritmo hasta la interpretación final de los resultados.

En conjunto, el proyecto nos permitió consolidar de forma aplicada los conceptos teóricos del curso y ganar práctica real en programación paralela con Python sobre un clúster de cómputo de alto rendimiento.

REFERENCIAS BIBLIOGRÁFICAS

[1] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. ACM, 2015.

[2] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Addison-Wesley, 2003, capítulo 3: Granularidad y eficiencia paralela.

[3] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2010, capítulo 2: Costos de comunicación vs cómputo.