Universidad de Santiago de Compostela

Escuela Técnica Superior de Ingeniería

# Práctica 2

Fundamentos de sistemas paralelos
30 de noviembre de 2024

*David Corral Pazos*

# 1. Introducción

En este documento se estudiará cómo evoluciona un código hecho en C con funciona de comunicación colectivas de MPI a medida que aumentamos el tamaño del problema y el número de procesadores.

# 2. Entorno de ejecución

Para llevar a cabo este estudio, los programas empleados serán ejecutados en el supercomputador de Galicia, el FinisTerrae III.

El equipo actualmente en servicio, el FinisTerrae III, fue instalado en el año 2021 y puesto en producción en el año 2022. Este supercomputador es un modelo Bull ATOS bullx, distribuido en 13 racks y compuesto por 354 nodos de computación. Dispone de un total de 22,656 núcleos Intel Xeon Ice Lake 8352Y, junto con 128 GPUs Nvidia A100 y 16 Nvidia T4 para aceleración de cómputo. Su capacidad de memoria es de 118 TB, complementada por un sistema de almacenamiento de altas prestaciones Lustre con 5,000 TB. La interconexión de todos los nodos se realiza mediante una red Mellanox Infiniband HDR, diseñada para ofrecer baja latencia y alta velocidad en las comunicaciones, lo cual es fundamental para la eficiencia en la ejecución de aplicaciones paralelas. La capacidad máxima de cómputo teórica de este sistema es de 4 PetaFlops, lo que le permite abordar aplicaciones de alta demanda computacional.

# 3. Marco teórico

## 3.1. Programación paralela y MPI

La programación paralela consiste en distribuir una tarea compleja en múltiples procesadores para reducir el tiempo de ejecución y aumentar la eficiencia. En un entorno de memoria distribuida, cada procesador tiene su propia memoria, y se requiere una comunicación explícita entre ellos para coordinar el trabajo. MPI (*Message Passing Interface*) es una biblioteca que permite esta comunicación entre procesos independientes, facilitando el desarrollo de aplicaciones de alto rendimiento en sistemas de memoria distribuida.

MPI ofrece funciones para el envío y la recepción de mensajes, que son esenciales en la programación de aplicaciones paralelas en las que los datos deben sincronizarse y compartirse entre procesos. En esta práctica, utilizamos las funciones *MPI_Init*, *MPI_Comm_rank*, *MPI_Comm_size*, *MPI_Send*, y *MPI_Recv* para implementar la comunicación entre procesos y coordinar la ordenación de filas de una matriz cuadrada.

## 3.2. Problema a resolver

El código intentará resolver el problema de ordenar las filas de una matriz cuadrada A en función del valor del sumatorio de los elementos de dicha fila, de

forma que la primera pase a ser la que tiene el menor sumatorio, y la última el que tiene el mayor.

Se ha programado de modo que el proceso de *rank* 0 obtiene inicialmente los valores de A y realiza una distribución de A cíclica por bloques. Concretamente, en este caso se ha diseñado de forma que el proceso de *rank* 0 envíe la información a los demás procesos, pero no realiza ningún cálculo. Los demás procesos le devuelve en resultando de las sumas de las filas que se les asignaron al proceso con *rank* 0 y este ordena finalmente la matriz.

### 3.3. Modelo SPMD

La estrategia de programación SPMD (*Single Program Multiple Data*) implica que cada proceso ejecute el mismo código, pero trabaje sobre diferentes subconjuntos de datos. En esta práctica, cada proceso calcula la suma de un subconjunto de filas de la matriz, utilizando su identificador único (*rank*) en el grupo de procesos MPI (*MPI_COMM_WORLD*) para determinar su parte del trabajo.

Este enfoque permite maximizar la independencia de cada proceso y minimizar la necesidad de comunicación, lo cual es fundamental en aplicaciones paralelas de alto rendimiento.

### 3.4. Métricas de evaluación

Para evaluar el rendimiento y la precisión del cálculo de $\pi$, se consideran las siguientes métricas:

- **Tiempo de Ejecución (T)**: Representa el tiempo total empleado en el cálculo, excluyendo las operaciones de entrada/salida.
- **Aceleración o *Speedup* (S)**: Es una medida que compara el tiempo de ejecución del proceso más lento entre varios procesos con el tiempo de ejecución del programa en modo secuencial.
- **Eficiencia (F)**: Se define como $F = \frac{SpeedUp}{numProcesos}$. Representa cuán eficientemente se utilizan los recursos disponibles y refleja la proporción de la mejora de rendimiento lograda en relación con el número de procesos empleados.

## 4. Metodología

### 4.1. Algoritmo implementado

Una vez el proceso con *rank* 0 tiene todos los valores de la matriz A, este reorganiza de manera cíclica la matriz para distribuir sus filas equitativamente entre los procesos. También se calculan los desplazamientos (*displs*) y cantidades enviadas (*sendcounts*) para determinar qué datos envía el proceso raíz a cada proceso con *MPI_Scatterv*.

Una vez cada proceso recibe el subconjunto de filas correspondiente, calcula la suma de los elementos de cada una.

Luego, las sumas calculadas localmente se recolectan en el proceso raíz usando *MPI_Gatherv*.

Finalmente, el proceso raíz ordena las filas de la matriz según las sumas calculadas utilizando *QuickSort*.

## 4.2. Medición del rendimiento

Se utiliza la función *gettimeofday()* para medir el tiempo total de ejecución ($T$) desde que el proceso raíz tiene los datos de A hasta que se obtiene la matriz ordenada, excluyendo las operaciones de entrada y salida.

## 4.3. Configuración del experimento

El experimento se ha diseñado para evaluar el impacto del número de procesadores y del de la matriz en el tiempo de ejecución, aceleración y eficiencia. Para ello, se ha configurado una serie de ejecuciones en las que el número de procesadores varía entre 2 y 64. Como referencia de rendimiento, se toma el tiempo de ejecución con dos procesadores para un tamaño de matriz dado, y este se compara con los tiempos obtenidos al incrementar el número de procesadores manteniendo constante el tamaño de la matriz, permitiendo así observar cómo afecta la paralelización al rendimiento.

El motivo por el que se se parten las mediciones de 2 y no de 1 es porque, tal y como se explicó anteriormente, se ha diseñado el código de forma que el proceso raíz no realiza ningún cálculo, sino que solo se encarga de enviar las filas y ordenar la matriz con las sumas recibidas. Esto se ha hecho para no comparar un código secuencial con un único procesador con uno con llamadas a MPI, y así evitar los *overheads* de las llamadas a funciones MPI. De todas formas, y tal y como se acaba de explicar, la ejecución con dos procesadores se realiza de manera secuencial, ya que uno no puede continuar hasta que el otro haya terminado su trabajo correspondiente.

Para cada cantidad de procesadores, se prueba una configuración variable del tamaño de la matriz, comenzando en 100 y aumentando progresivamente hasta alcanzar valores de 20100. Este amplio rango permite evaluar el efecto que tiene el tamaño de la matriz en la paralelización del problema, así como en el tiempo de ejecución.

La fase de análisis se lleva a cabo con scripts en Python, que permiten realizar cálculos adicionales y generar gráficos que ilustran la relación entre el número de procesadores, el tamaño del problema y el tiempo de ejecución. Mediante el análisis gráfico, es posible identificar tendencias en el rendimiento, y evaluar el comportamiento del algoritmo en términos de escalabilidad y eficiencia.

### 4.4. Consideraciones

En una primera instancia se había desarrollado el código de forma que leía de un documento la matriz a ordenar. Sin embargo, debido a las limitaciones de almacenamiento del CESGA, el tamaño de la matriz era muy limitado, llegando a poco más de 6000. Por ello, se terminó haciendo que sea el propio código el que genere la matriz con valores aleatorios de entre 1 y 9. De todas formas, teniendo el código adjunto, es trivial modificarlo para que la matriz sea leída desde un archivo dado.

## 5. Resultados obtenidos

### 5.1. Introducción a los resultados

En esta sección, se presentan los resultados obtenidos mediante la ejecución del algoritmo paralelo para la ordenación de las filas de una matriz mediante el sumatorio de sus valores, evaluando su desempeño en términos de tiempo de ejecución. A lo largo de las pruebas, se exploraron diversas configuraciones que involucran el tamaño de la matriz y el número de procesadores. Los resultados obtenidos nos permitirán analizar la eficiencia del algoritmo en función de estos parámetros y estudiar cómo se comportan las métricas clave, como el tiempo de ejecución.

### 5.2. Relación entre tiempo de ejecución y número de procesadores

En este apartado, se analiza cómo el tiempo de ejecución del algoritmo paralelo varía con el número de procesadores empleados. En principio, al incrementar el número de procesadores, el tiempo de ejecución disminuye debido a la distribución de la carga de trabajo entre múltiples unidades de procesamiento. Este comportamiento es coherente con el objetivo del paralelismo, que busca reducir el tiempo de cómputo mediante el uso de recursos adicionales.

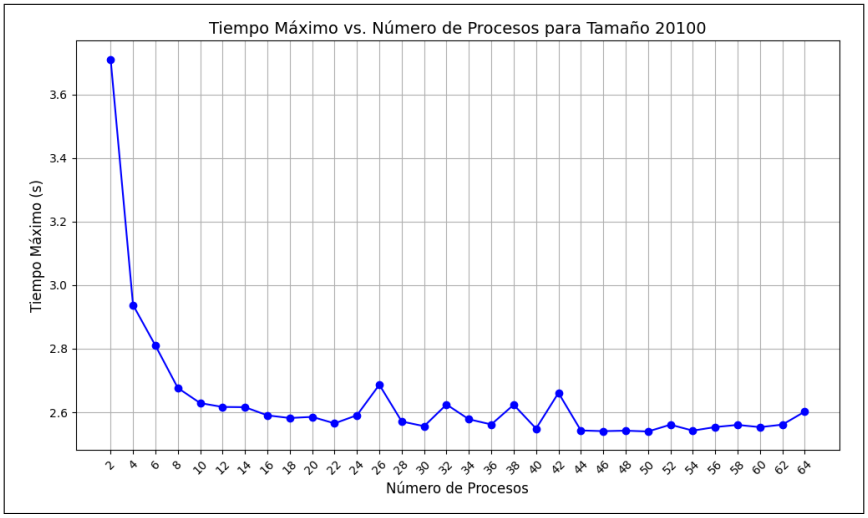


Figura 1: Tiempo de ejecución en función del número de procesadores para tamaño de matriz 20100.

En la Figura 1, se observa que los primeros incrementos en el número de procesadores generan una reducción significativa en el tiempo de ejecución. Sin embargo, a medida que el número de procesadores supera este valor, la reducción en el tiempo de ejecución se vuelve marginal. Este fenómeno es consistente con la Ley de Amdahl, la cual establece que la aceleración máxima alcanzable en un sistema paralelo está limitada por la fracción del código que debe ejecutarse de forma secuencial.

Otro aspecto importante por considerar en esta gráfica es el aumento de los costos de comunicación y sincronización entre procesos a medida que se añaden más núcleos. Estos costos, que incluyen el tiempo necesario para coordinar el procesamiento distribuido y transmitir datos entre los distintos nodos, tienden a contrarrestar los beneficios del paralelismo en niveles elevados. En este experimento, esto se refleja en el comportamiento asintótico del tiempo de ejecución a partir de los 12 procesadores, punto en el cual la mejora en el rendimiento es mínima.

También vemos una serie de picos para unas cantidades de procesadores concretos pero al final acaba estabilizándose. Es normal que haya algunas ligeras subidas y bajadas debido a que todos los procesadores reciben la misma carga de trabajo, pero cuando la división de las filas no es exacta, algunos de ellos realizan más sumas que otros. En estos casos, algunos procesadores realizan más operaciones que otros, lo que provoca que el tiempo total del programa dependa del procesador que termine último. Cuando "sobran" pocas filas, el programa en general espera a que pocos procesadores terminen. Por ejemplo, si tuviésemos 100 filas y 9 procesadores, la división no sería exacta ($100 \,\%9 = 1$). En este caso, cada procesador procesaría inicialmente 11 filas, pero uno de ellos tendría que procesar una fila adicional (12 en total). Esto genera un pequeño desequilibrio:

mientras ese procesador realiza su operación extra, los demás deben esperar para sincronizarse antes de continuar o finalizar. Este tipo de comportamiento puede provocar pequeñas fluctuaciones en los tiempos medidos para distintas cantidades de procesadores.

A medida que aumentamos el número de procesadores, este efecto tiende a suavizarse porque las cargas de trabajo individuales se vuelven más pequeñas y las diferencias en el reparto tienen menor impacto relativo. Sin embargo, el costo de sincronización sigue siendo un factor a considerar, ya que aumenta con el número total de procesadores.

## 5.3. Aceleración

La aceleración o *Speed Up* es una medida que compara el tiempo de ejecución del proceso más lento entre varios procesos con el tiempo de ejecución del programa en modo secuencial.

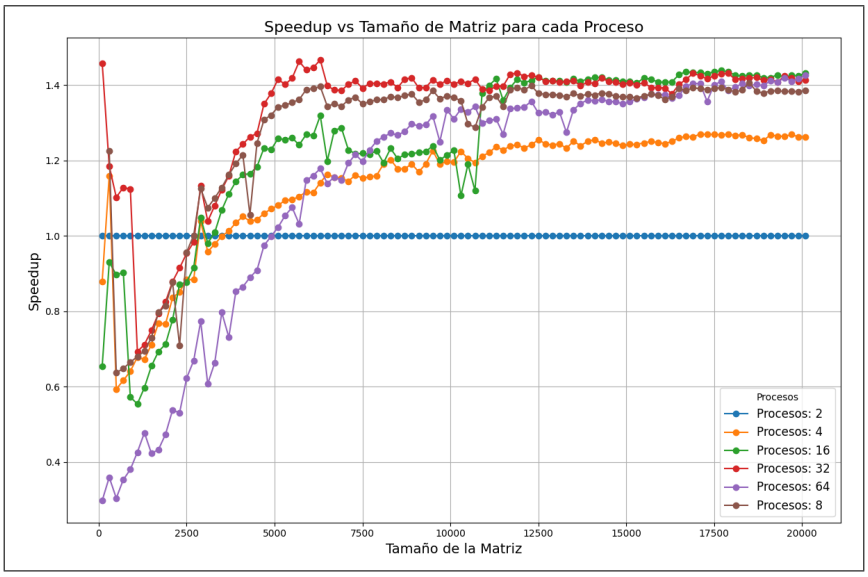


Figura 2: *Speed Up* vs Tamaño de la matriz para cada proceso

En la figura superior podemos ver, en primer lugar, como el código que se ejecuta de manera secuencial tiene una aceleración de 1, que hace de referencia.

Podemos comprobar que, para tamaños pequeños del problema el código que se ejecuta secuencial suele aportar mejores resultados que el que emplea múltiples procesadores para repartir la carga de trabajo. Esto es debido a que la sobrecarga asociada con la distribución del trabajo entre múltiples procesadores, no es compensada con el tiempo que se ahorra paralelizando las sumas.

Sin embargo, a medida que se va aumentando el tamaño del problema, la eficiencia también aumenta para cualquier número de procesos, a excepción de

cuando el programa se ejecuta de manera secuencial, hasta que llegan a un punto en el que esta tendencia se estabiliza, indicando que la ganancia en eficiencia se mantiene constante para problemas de mayor escala.

## 5.4. Matriz de isoeficiencia

Una matriz de isoeficiencia es una herramienta que describe la relación entre el tamaño del problema y el número de procesadores necesarios para mantener una eficiencia constante en el sistema, en nuestro caso el tamaño del problema es el tamaño de la matriz. Su objetivo es mostrar cómo cambia la eficiencia del algoritmo a medida que se aumenta el número de procesadores y el tamaño del problema. Es decir, en una matriz de isoeficiencia, en cada celda se representa la eficiencia de un programa para un tamaño de problema y número de procesadores determinado.

La eficiencia (F) se definía como $F = \frac{SpeedUp}{numProcesos}$, y representa cuán eficientemente se utilizan los recursos disponibles y refleja la proporción de la mejora de rendimiento lograda en relación con el número de procesos empleados.

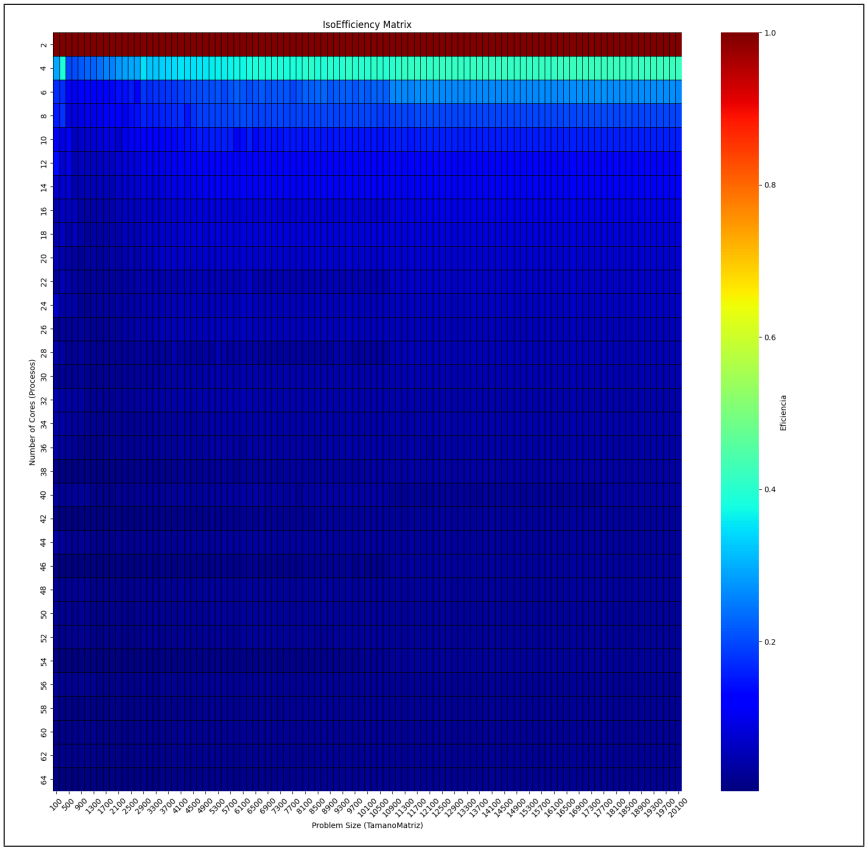


Figura 3: Matriz de isoeficiencia

Como podemos observar, la fila de arriba tiene una eficiencia de 1, ya que todo se ejecuta de manera secuencial. Al principio, para tamaños de problema muy pequeños, aumentar el número de procesadores no resulta rentable en términos de eficiencia, ya que el tiempo que se dedica a dividir y distribuir el trabajo entre los procesadores es mayor que el beneficio obtenido al paralelizar la ejecución. Y lo mismo ocurre para este problema en particular a medida que vamos aumentando el tamaño del problema.

A fin de cuentas, cuando todos los procesos terminan de realizar sus sumas y envían los resultados al proceso raíz, es este el que tiene que ordenar la matriz, presentando un cuello de botella.

Se ha probado a medir los tiempos para los mismos procesadores y para los mismos tamaños de matriz sin tener en cuenta esta última ordenación final y el resultado de la eficiencia sigue siendo muy similar.

A fin de cuentas, y aunque sí encontremos una mejora en el tiempo de ejecución, estamos paralelizando sumas, que, aunque son operaciones que se pueden dividir entre múltiples procesadores, no representan un problema altamente de-

mandante en términos de paralelismo. Esto se debe a que el coste computacional de las sumas es relativamente bajo en comparación con el tiempo invertido en la comunicación y coordinación entre procesadores.

En problemas donde la comunicación entre procesos ocupa una proporción significativa del tiempo total, el rendimiento paralelo suele verse limitado por este factor. Según la ley de Amdahl, la ganancia en eficiencia se reduce a medida que el porcentaje de la tarea que no puede ser paralelizado crece, lo cual explica por qué incluso al optimizar ciertas etapas del proceso, la eficiencia no escala proporcionalmente con el número de procesadores.

En resumen, aunque el paralelismo permite reducir el tiempo de ejecución, su efectividad está intrínsecamente ligada a la naturaleza del problema y a la relación entre las operaciones computacionales y los costes de comunicación. Por lo tanto, optimizar este tipo de problemas requiere un balance cuidadoso entre la cantidad de trabajo paralelo y la carga que supone coordinar los procesos involucrados.

# 6. Conclusiones

A lo largo de esta práctica, se han evaluado los aspectos de rendimiento, aceleración y eficiencia en la paralelización de la ordenación de filas de una matriz mediante MPI. Los resultados obtenidos muestran que, aunque la paralelización reduce significativamente el tiempo de ejecución en comparación con la ejecución secuencial, la eficiencia del proceso presenta importantes limitaciones debido a los costos de comunicación y sincronización entre procesos.

La eficiencia mostró una tendencia decreciente a medida que se incrementaba el número de procesadores y el tamaño del problema. Esto se debe a que, en problemas con cargas computacionales moderadas como el abordado en esta práctica, los beneficios de distribuir las sumas entre múltiples procesadores se ven contrarrestados por la sobrecarga de dividir, comunicar y coordinar el trabajo.

Por otro lado, se observó que el tamaño del problema tiene un impacto directo en la aceleración. Para tamaños pequeños de la matriz, el paralelismo no resulta rentable, ya que los costos asociados al manejo de múltiples procesadores superan los beneficios obtenidos. Sin embargo, a medida que aumenta el tamaño del problema, la aceleración mejora y se estabiliza, destacando la importancia de la relación entre el tamaño de la tarea y el número de recursos paralelos empleados.

En conclusión, si bien esta práctica demuestra la capacidad de la paralelización para reducir tiempos de ejecución, también subraya las limitaciones inherentes en términos de eficiencia, especialmente en problemas de bajo coste computacional. Esto pone de manifiesto la necesidad de encontrar un balance óptimo entre el tamaño del problema y el número de procesadores para maximizar el rendimiento y la eficiencia en aplicaciones paralelas.

# Referencias

[1] Centro de supercomputación de Galicia, Enlace.

[2] Ley de Amdahl, Enlace.

[3] Introducción a MPI, Universidad de Valencia, Enlace.