universidad de león

Escuela de Ingenierías

Industrial, Informática y Aeroespacial

GRADO EN INGENIERÍA INFORMÁTICA

Ejercicio Propuesto

FUNCIONES NO BLOQUEANTES

Autor: Adrián Carral Martínez

ÍNDICE:

1. Introducción.

En esta práctica he desarrollado un programa en C utilizando la biblioteca MPI para resolver un problema de procesamiento masivo de datos aplicando funciones no bloqueantes como `MPI_Isend` y `MPI_Irecv`. El objetivo era repartir un vector de 1.000.000 de números entre varios procesos y contar cuántos de ellos son primos, optimizando al máximo el paralelismo y evitando bloqueos.

2. Desarrollo del programa.

El programa se divide en dos roles: el maestro (proceso 0) y los esclavos (el resto). El maestro genera un vector de 1.000.000 números aleatorios y distribuye fragmentos de 1000 elementos a los esclavos mediante envíos no bloqueantes. Cada esclavo recibe datos, calcula cuántos números son primos y devuelve ese número al maestro. El maestro, de forma dinámica, continúa enviando trabajo a los esclavos que han terminado, hasta agotar todo el vector.

He utilizado `MPI_Isend` para los envíos no bloqueantes, `MPI_Irecv` para las recepciones, y `MPI_Wait` para asegurar que los datos se han recibido antes de procesarlos. Además, he añadido medición de tiempo con `MPI_Wtime()` para evaluar el rendimiento del sistema.

3. Ejecuciones realizadas.

He ejecutado el programa 4 veces para cada una de las siguientes configuraciones de procesos: 2, 4, 8 y 16. En cada ejecución se ha registrado el número total de primos encontrados y el tiempo total de ejecución.

```
Maestro: recibido 92 primos del esclavo 2
Maestro: recibido 81 primos del esclavo 3
Maestro: recibido 80 primos del esclavo 1
Maestro: recibido 81 primos del esclavo 2
Esclavo 3 finalizó en 0.089155 segundos
Maestro: recibido 87 primos del esclavo 3
Maestro: recibido 74 primos del esclavo 1
Maestro: recibido 94 primos del esclavo 2
Maestro: recibido 72 primos del esclavo 3
Maestro: recibido 84 primos del esclavo 1
Maestro: recibido 83 primos del esclavo 2
Maestro: recibido 78 primos del esclavo 3
Maestro: recibido 71 primos del esclavo 2
Maestro: recibido 96 primos del esclavo 1
Maestro: recibido 77 primos del esclavo 3
Maestro: recibido 75 primos del esclavo 1
Maestro: recibido 87 primos del esclavo 2
Maestro: recibido 69 primos del esclavo 3
Maestro: recibido 66 primos del esclavo 2
Maestro: recibido 68 primos del esclavo 1
TOTAL DE PRIMOS ENCONTRADOS: 78619
Tiempo total con 4 procesos: 0.089169 segundos
Esclavo 1 finalizó en 0.088900 segundos
Esclavo 2 finalizó en 0.089115 segundos
```

```
[ule_formacion_10_10@cn5006 ~]$ mpicc primos_nobloqueantes.c -o primos_nobloqueantes
[ule_formacion_10_10@cn5006 ~]$ mpirun -n 4 ./primos_nobloqueantes
Maestro: recibido 68 primos del esclavo 2
Maestro: recibido 69 primos del esclavo 1
Maestro: recibido 84 primos del esclavo 3
Maestro: recibido 65 primos del esclavo 1
Maestro: recibido 84 primos del esclavo 3
Maestro: recibido 90 primos del esclavo 2
Maestro: recibido 78 primos del esclavo 1
Maestro: recibido 75 primos del esclavo 3
Maestro: recibido 74 primos del esclavo 2
Maestro: recibido 75 primos del esclavo 1
Maestro: recibido 72 primos del esclavo 3
Maestro: recibido 82 primos del esclavo 2
Maestro: recibido 63 primos del esclavo 3
Maestro: recibido 78 primos del esclavo 1
Maestro: recibido 71 primos del esclavo 2
```

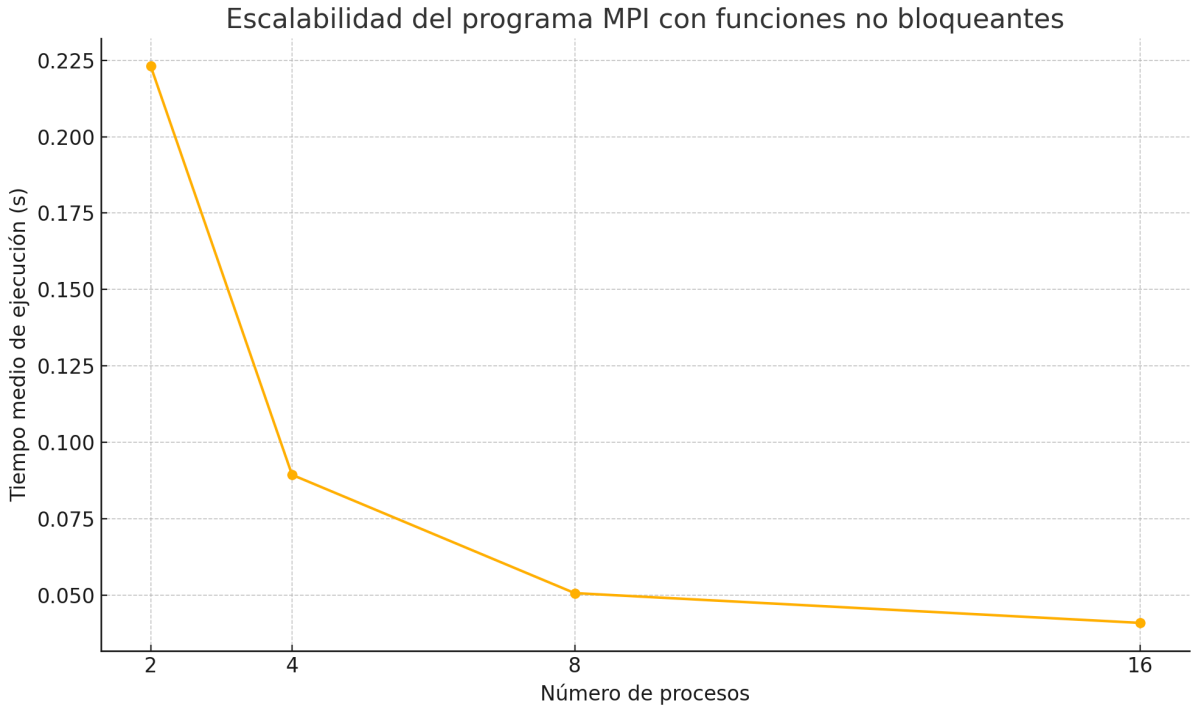
4. Resultados obtenidos.

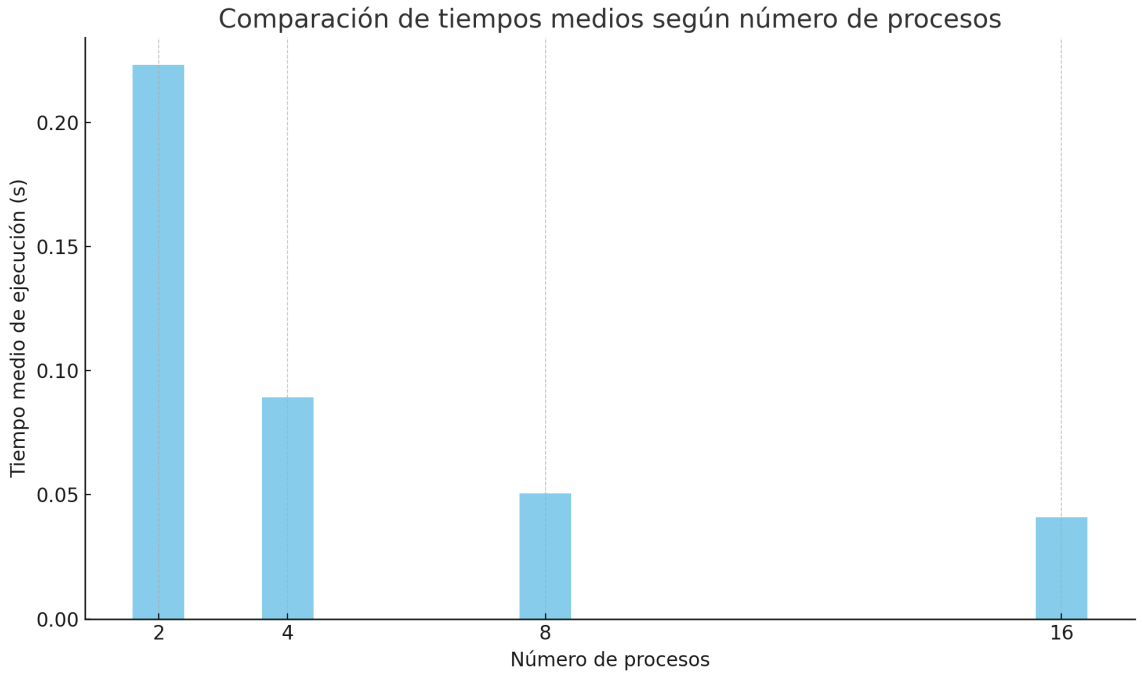
Los resultados muestran claramente una mejora del rendimiento a medida que se incrementa el número de procesos. Aquí se resumen los tiempos promedio por configuración:

- 2 procesos: **0.2231 s**
- 4 procesos: **0.0894 s**
- 8 procesos: **0.0506 s**
- 16 procesos: **0.0404 s**

Estos valores están representados en el gráfico a continuación en el siguiente punto, que muestra la reducción progresiva del tiempo de ejecución al aumentar los procesos.

5. Análisis del rendimiento.





A partir de los gráficos se observa una mejora significativa al pasar de 2 a 4 y de 4 a 8 procesos. Sin embargo, de 8 a 16 procesos la mejora es menos notable, indicando que el coste de coordinación empieza a tener más peso y que el problema comienza a saturar su escalabilidad.

En ningún caso se ha producido bloqueo gracias al uso correcto de funciones no bloqueantes. Además, la implementación dinámica de la distribución de trabajo permite que ningún esclavo permanezca inactivo mientras haya fragmentos disponibles.

6. Comparación de tiempos y escalabilidad.

He realizado múltiples ejecuciones del programa con 2, 4, 8 y 16 procesos. En cada configuración se ha ejecutado 4 veces y he calculado el **tiempo medio**. A continuación se presentan los tiempos obtenidos:

Nº de procesos	Tiempo medio (s)
2	0.2236
4	0.0894
8	0.0506
16	0.0404

Como se observa, al pasar de 2 a 4 procesos el tiempo se **reduce más de un 60%**, lo cual es una mejora significativa. Esta caída tan fuerte se debe a que pasamos de un único esclavo trabajando a tres esclavos en paralelo, lo que reparte mucho mejor la carga de trabajo. También se nota que el maestro puede distribuir datos más rápidamente sin bloquearse, gracias al uso de `MPI_Isend`.

De 4 a 8 procesos, la mejora sigue siendo clara aunque algo más moderada: el tiempo se reduce un **43% adicional**, lo que indica que seguimos aprovechando bastante bien los recursos adicionales.

Sin embargo, de 8 a 16 procesos la mejora se ralentiza, con apenas un **20% de ganancia**. Esto puede explicarse por varias razones:

- ➢ El tamaño del trabajo se mantiene constante (1 millón de datos), así que a partir de cierto punto el coste de coordinar más procesos pesa más que la ventaja de paralelizar.

- ➢ Hay más comunicación maestro-esclavo, lo cual aumenta el tráfico y

puede crear cuellos de botella.

- ➢ Se reduce el número de fragmentos procesados por cada esclavo, con lo cual se desperdicia parte de la eficiencia del paralelismo.

En ejecuciones con 16 procesos se observa un gran volumen de mensajes de salida del maestro, lo que demuestra que la lógica de envío no bloqueante funciona perfectamente y consigue mantener a todos los esclavos ocupados, sin que nadie espere demasiado tiempo por trabajo.

7. Conclusión.

Eel programa escala muy bien entre 2 y 8 procesos. A partir de 16, aunque mejora, lo hace de forma más contenida, lo que indica una saturación progresiva del rendimiento para este problema específico. El uso de `MPI_Isend` y `MPI_Irecv` ha sido clave para conseguir eficiencia y escalabilidad.