Desarrollo de Software en Arquitecturas Paralelas

Práctica 2: Cálculo de Números Primos con MPI

Jaime Hernández Delgado

14 de abril de 2025

Índice

1. Introducción

Esta práctica consiste en la implementación paralela del algoritmo para calcular el número de enteros primos menores que un valor dado n, utilizando la biblioteca MPI (Message Passing Interface). El objetivo principal es analizar la mejora en rendimiento que se obtiene al paralelizar el algoritmo respecto a su versión secuencial, midiendo métricas como el speed-up y la eficiencia.

Los algoritmos para determinar si un entero es primo tienen un coste computacional significativo que se presta bien a la paralelización, especialmente cuando se debe realizar este cálculo para un rango amplio de números.

2. Descripción del Algoritmo

2.1. Algoritmo Secuencial

El algoritmo secuencial para contar los números primos menores que un número n dado consiste en:

```
int numero_primos_sec(int n) {
    int total = 0;
    for (int i = 2; i <= n; i++) {
        if (esPrimo(i)) {
            total++;
        }
    }
    return total;
}
```

Para determinar si un número es primo, se utiliza la función `esPrimo()` que verifica si un número tiene divisores entre 2 y su raíz cuadrada:

```
int esPrimo(int p) {
    if (p <= 1) return 0;
    if (p == 2) return 1;
    if (p % 2 == 0) return 0;

    int limite = (int)sqrt(p) + 1;
    for (int i = 3; i <= limite; i += 2) {
        if (p % i == 0) {
            return 0;
        }
    }
    return 1;
}
```

2.2. Algoritmo Paralelo

La versión paralela distribuye el trabajo entre los procesos disponibles mediante una distribución cíclica de los números a evaluar:

```
int numero_primos(int n, int myrank, int nprocs) {
    int total_primos = 0;

    for (int i = 2 + myrank; i <= n; i += nprocs) {
        if (esPrimo(i)) {
            total_primos++;
        }
    }

    return total_primos;
}
```

Cada proceso, según su rango (`myrank`), evalúa un subconjunto de números:

- El proceso 0 evalúa los números 2, 2+p, 2+2p, ...
- El proceso 1 evalúa los números 3, 3+p, 3+2p, ...
- Y así sucesivamente para cada proceso.

Donde `p` es el número total de procesos.

3. Implementación

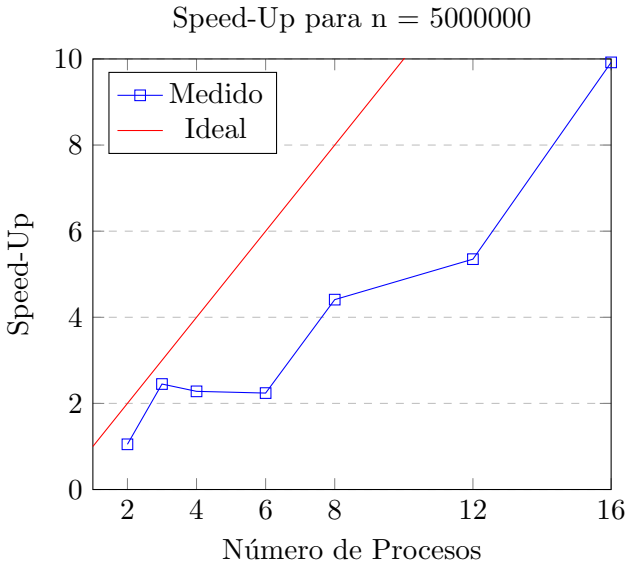
El programa implementa la versión paralela utilizando el siguiente flujo:

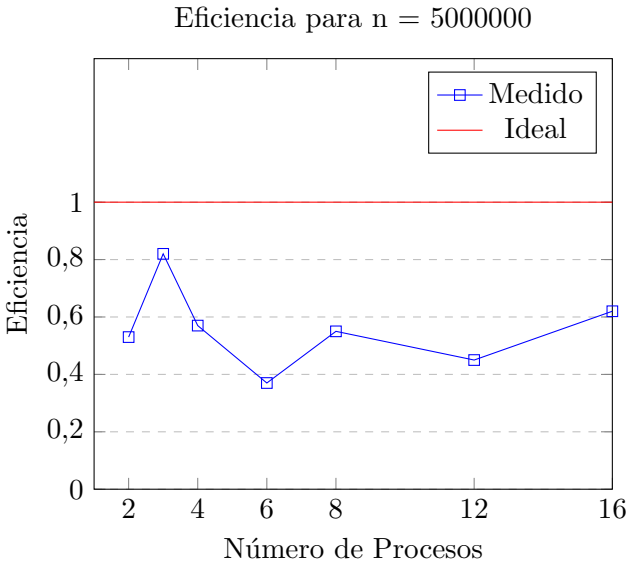
1. Inicialización del entorno MPI y obtención del rango y número total de procesos.
2. Para cada valor de n (desde n_min hasta n_max), incrementando por un factor n_factor:
   a) El proceso raíz ejecuta la versión secuencial y mide su tiempo de ejecución.
   b) Todos los procesos se sincronizan mediante `MPI_Barrier`.
   c) Cada proceso calcula su parte de números primos y mide su tiempo parcial.
   d) Se utiliza `MPI_Reduce` para sumar los resultados parciales.
   e) El proceso raíz calcula el tiempo total de ejecución paralela.
   f) Se recopilan los tiempos parciales mediante `MPI_Gather`.
   g) El proceso raíz calcula y muestra el speed-up y la eficiencia.
3. Finalización del entorno MPI.

4. Análisis de Resultados

A continuación, se presentan los resultados de las ejecuciones con diferentes números de procesos y tamaños del problema.

4.1. Speed-Up y Eficiencia

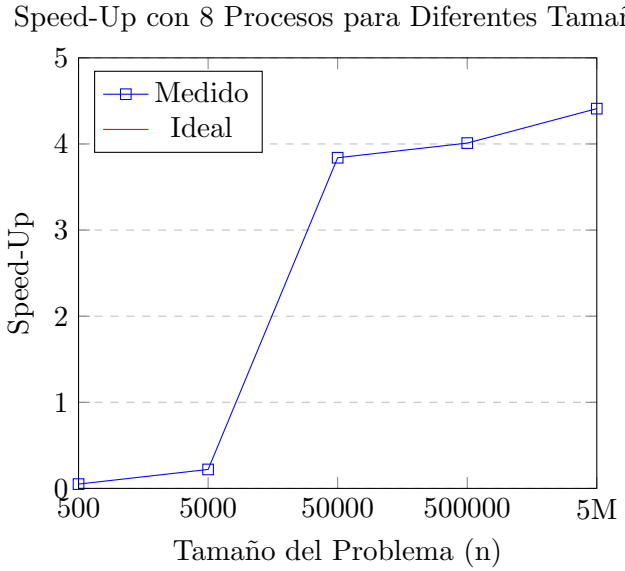




4.2. Análisis para Diferentes Tamaños de Problema

Para analizar cómo afecta el tamaño del problema al speed-up obtenido, se presenta la siguiente gráfica que muestra el speed-up para diferentes

valores de n con 8 procesos:



5. Discusión de Resultados

5.1. Tiempos Parciales Dispares

En los resultados de ejecución, se observa que algunos procesos tienen tiempos significativamente menores que otros. Esto ocurre debido a la distribución cíclica utilizada, donde cada proceso evalúa una secuencia diferente de números. En particular:

- Los procesos que evalúan números pares (excepto el 2) tienen un tiempo de ejecución mucho menor, ya que detectan inmediatamente que son divisibles por 2 y no son primos.
- Los procesos de rango par (0, 2, 4...) evalúan mayoritariamente números impares, realizando más cálculos completos con la función `esPrimo()`.
- Los procesos de rango impar (1, 3, 5...) evalúan más números pares, realizando menos cálculos complejos.

Esta distribución desigual de la carga de trabajo explica por qué los tiempos parciales varían significativamente entre procesos.

5.2. Speed-Up y Eficiencia No Óptimos

Los resultados muestran que el speed-up y la eficiencia no alcanzan los valores ideales, especialmente cuando aumenta el número de procesos. Esto se debe a varios factores:

1. **Distribución desigual de la carga:** Como se explicó anteriormente, la distribución cíclica simple genera un desequilibrio en la carga computacional.

2. **Overhead de comunicación:** A medida que aumenta el número de procesos, el tiempo dedicado a la comunicación entre ellos (barreras de sincronización, reducción de resultados, recopilación de tiempos) aumenta en proporción al tiempo de cálculo efectivo.

5.3. Mejora con el Tamaño del Problema

Se observa que la eficiencia mejora significativamente con el tamaño del problema. Para n = 5000000, se alcanzan eficiencias de hasta 0.82 con 3 procesos y 0.62 con 16 procesos, mientras que para valores pequeños de n, la eficiencia es cercana a cero.

Esto ilustra la ley de Amdahl: para que la paralelización sea efectiva, el problema debe ser lo suficientemente grande como para que el tiempo de cálculo domine sobre el overhead de comunicación.