Algoritmo de la Criba de Eratóstenes usando MPI (Message Passing Interface)

Sistemas Concurrentes y Distribuidos

Ismael Sallami Moreno
Ingenería Informática + ADE
Universidad de Granada (UGR)

6 de diciembre de 2024

Índice

1 Enunciado de la Actividad

El objetivo de este ejercicio es programar una versión distribuida del famoso método de *la criba de Erastotenes* para obtener los N primeros números primos. Se utilizará una red de procesos concurrentes conectados por una tubería o "pipeline"de procesos, como en la figura siguiente, para generar los N primeros números primos.

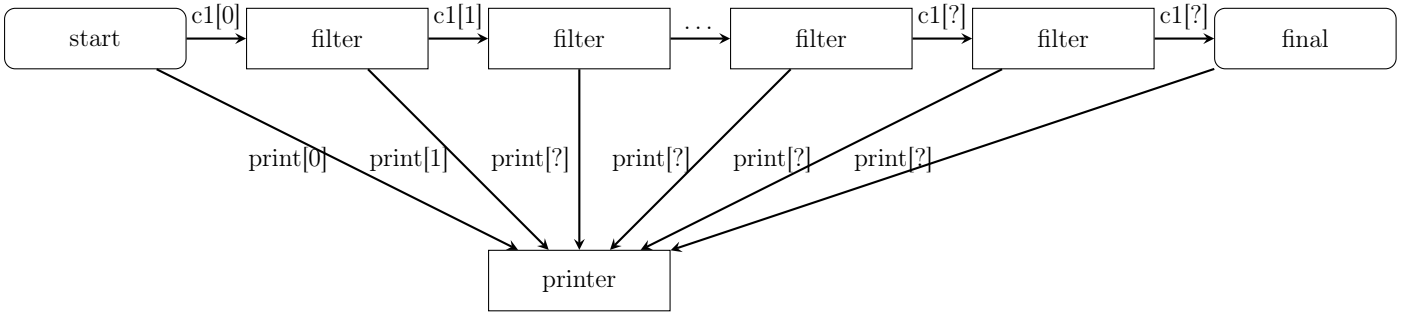

Figura 1: Pipeline de procesos para generar números primos

Para programarlo, suponer que se conoce una constante `nLimite` que es mayor o igual que el N-ésimo número primo. El proceso `start` generará el primer número primo (el 2) y lo enviará al proceso `printer`, que lo imprime; a continuación genera el siguiente primo (el 3), lo envía al primer proceso `filter`, al que seguirá enviando los siguientes números impares: 5, 7, 9, y así sucesivamente.

2 Solución con MPI

La solución propuesta consiste en ejecutar el programa de la criba de Eratóstenes utilizando MPI. A medida que se incrementa el número de procesos especificados con ‘mpirun‘, se añaden más filtros, lo que mejora la precisión y eficiencia del resultado.

```cpp
#include <mpi.h>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
    int rank, size, x, valor;       // Identificadores de rango, tamaño y valores
    int i = 1;                      // Índice para números primos
    bool fin = false;               // Indica si el procesamiento debe finalizar
    MPI_Status status;              // Estado para recibir información de mensajes
    MPI_Request request;            // Request para comunicación no bloqueante
    const int nLimite = 100;        // Límite de números primos a procesar

    MPI_Init(&argc, &argv);                     // Inicializa el entorno MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);       // Obtiene el rango del proceso actual
```

```
15      MPI_Comm_size(MPI_COMM_WORLD, &size);        // Obtiene el número total
            de procesos
16
17      if (size < 3) {
18          if (rank == 0) {
19              cerr << "Se requieren al menos 3 procesos para ejecutar este
                    programa." << endl;
20          }
21          MPI_Abort(MPI_COMM_WORLD, 1); // Aborta la ejecución si hay menos
                de 3 procesos
22      }
23
24      if (rank == 0) {
25          // Proceso principal (generador): envía números naturales al primer
                filtro
26          x = 2; // El primer número primo
27          MPI_Send(&x, 1, MPI_INT, size - 1, 0, MPI_COMM_WORLD); // Envía 2
                directamente al impresor
28          while (!fin) {
29              i += 1;
30              x = i;
31              if (x > nLimite) {
32                  fin = true;
33                  x = -1; // Señal de finalización
34              }
35              MPI_Send(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD); // Envía nú
                    meros al primer filtro
36              MPI_Irecv(&x, 1, MPI_INT, size - 1, MPI_ANY_TAG, MPI_COMM_WORLD
                    , &request);
37          }
38      } else if (rank == size - 1) {
39          // Último proceso (impresor): imprime números primos recibidos
40          while (!fin) {
41              MPI_Recv(&valor, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, &
                    status);
42              if (valor == -1) { // Señal de finalización
43                  fin = true;
44              } else {
45                  cout << "Proceso " << rank << ": " << valor << " es primo."
                        << endl;
46              }
47          }
48      } else {
49          // Procesos intermedios (filtros): filtran números no primos
50          MPI_Recv(&valor, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, &status);
                // Reciben el primer número
51          int primo = valor; // El número primo que filtra
52          MPI_Send(&primo, 1, MPI_INT, size - 1, 0, MPI_COMM_WORLD); // Enví
                an el primo al impresor
53          while (!fin) {
54              MPI_Recv(&x, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, &status);
                    // Reciben números del filtro anterior
55              if (x == -1) { // Señal de finalización
56                  fin = true;
57                  MPI_Send(&x, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD); //
                        Propagan la señal de finalización
```

```
58              } else if (x % primo != 0) { // Si el número no es divisible
                    por el primo actual
59                  if (rank < size - 2) {
60                      MPI_Send(&x, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
                            // Envía al siguiente filtro
61                  } else {
62                      MPI_Send(&x, 1, MPI_INT, size - 1, 0, MPI_COMM_WORLD);
                            // Envía al impresor
63                  }
64              }
65              MPI_Irecv(&x, 1, MPI_INT, size - 1, MPI_ANY_TAG, MPI_COMM_WORLD
                    , &request);
66          }
67      }
68
69      MPI_Finalize(); // Finaliza el entorno MPI
70      return 0;
71  }
```

2.1. Explicación detallada del código

El código presentado implementa la criba de Eratóstenes utilizando *MPI* (Message Passing Interface) para calcular números primos de forma concurrente y distribuida. A continuación, se describen las funciones principales de cada bloque del programa:

2.1.1. Inicialización y configuración

El programa comienza inicializando el entorno de ejecución con `MPI_Init`, obteniendo el rango (`rank`) y el tamaño (`size`) del comunicador global `MPI_COMM_WORLD`. Si el número de procesos es menor que 3, el programa imprime un mensaje de error y aborta la ejecución mediante `MPI_Abort`. Esto asegura que haya al menos un generador, un filtro y un impresor.

2.1.2. Generador de números (Proceso 0)

El proceso con `rank = 0` actúa como generador de números naturales. Inicialmente, envía el número 2 (el primer número primo) directamente al impresor (`rank = size - 1`). Luego, comienza un bucle que genera números naturales secuenciales. Cada número se envía al primer filtro (`rank = 1`) usando `MPI_Send`.

El generador finaliza cuando el número actual supera el límite predefinido (`nLimite`). En ese momento, envía una señal especial de terminación (`x = -1`) al primer filtro.

2.1.3. Filtros intermedios (Procesos con $1 \leq$ `rank` $<$ `size` $- 1$)

Cada filtro intermedio tiene la responsabilidad de descartar números que no son primos. El primer número recibido por un filtro se considera su número primo característico (`primo`).

Los filtros reciben números desde el proceso anterior mediante `MPI_Recv`. Si el número recibido no es divisible por `primo`, el filtro lo reenvía al siguiente filtro usando `MPI_Send`.

Si el filtro detecta la señal de terminación (`x = -1`), la propaga al siguiente proceso y termina su ejecución.

Este diseño asegura que cada filtro actúe como un módulo de exclusión para números no primos, delegando la validación de números más grandes a los filtros siguientes.

2.1.4. Impresor de números primos (Último proceso)

El proceso con `rank = size - 1` es responsable de imprimir los números primos. Recibe números desde el último filtro intermedio y los imprime en pantalla. Al recibir la señal de terminación (`x = -1`), el impresor detiene su ejecución.

2.1.5. Finalización del programa

Todos los procesos llaman a `MPI_Finalize` para limpiar el entorno de ejecución de MPI y asegurar una terminación ordenada.

2.2. Compilación

```
# Compilar el código : (2 maneras)
    - mpic++ -o criba criba.cpp
    - mpicxx -std=c++11 -o criba criba.cpp

# Ejecutar el programa con 4 procesos: (2 maneras)
    - mpirun -np 4 ./criba
    - mpirun --oversubscribe -np 4 criba
```

2.3. Ejecución y/o salida

2.3.1. Con 3 procesos

```
Proceso 2: 2 es primo.
Proceso 2: 3 es primo.
Proceso 2: 5 es primo.
Proceso 2: 7 es primo.
Proceso 2: 9 es primo.
Proceso 2: 11 es primo.
Proceso 2: 13 es primo.
Proceso 2: 15 es primo.
Proceso 2: 17 es primo.
Proceso 2: 19 es primo.
Proceso 2: 21 es primo.
Proceso 2: 23 es primo.
Proceso 2: 25 es primo.
Proceso 2: 27 es primo.
Proceso 2: 29 es primo.
Proceso 2: 31 es primo.
Proceso 2: 33 es primo.
Proceso 2: 35 es primo.
Proceso 2: 37 es primo.
Proceso 2: 39 es primo.
```

```
Proceso 2: 41 es primo.
Proceso 2: 43 es primo.
Proceso 2: 45 es primo.
Proceso 2: 47 es primo.
Proceso 2: 49 es primo.
Proceso 2: 51 es primo.
Proceso 2: 53 es primo.
Proceso 2: 55 es primo.
Proceso 2: 57 es primo.
Proceso 2: 59 es primo.
Proceso 2: 61 es primo.
Proceso 2: 63 es primo.
Proceso 2: 65 es primo.
Proceso 2: 67 es primo.
Proceso 2: 69 es primo.
Proceso 2: 71 es primo.
Proceso 2: 73 es primo.
Proceso 2: 75 es primo.
Proceso 2: 77 es primo.
Proceso 2: 79 es primo.
Proceso 2: 81 es primo.
Proceso 2: 83 es primo.
Proceso 2: 85 es primo.
Proceso 2: 87 es primo.
Proceso 2: 89 es primo.
Proceso 2: 91 es primo.
Proceso 2: 93 es primo.
Proceso 2: 95 es primo.
Proceso 2: 97 es primo.
Proceso 2: 99 es primo.
```

2.3.2. Con 4 procesos

```
Proceso 3: 3 es primo.
Proceso 3: 5 es primo.
Proceso 3: 7 es primo.
Proceso 3: 11 es primo.
Proceso 3: 13 es primo.
Proceso 3: 17 es primo.
Proceso 3: 19 es primo.
Proceso 3: 23 es primo.
Proceso 3: 25 es primo.
Proceso 3: 29 es primo.
Proceso 3: 31 es primo.
Proceso 3: 35 es primo.
Proceso 3: 37 es primo.
Proceso 3: 41 es primo.
Proceso 3: 43 es primo.
Proceso 3: 47 es primo.
Proceso 3: 49 es primo.
Proceso 3: 53 es primo.
Proceso 3: 55 es primo.
Proceso 3: 59 es primo.
Proceso 3: 61 es primo.
```

```
Proceso 3: 65 es primo.
Proceso 3: 67 es primo.
Proceso 3: 71 es primo.
Proceso 3: 73 es primo.
Proceso 3: 77 es primo.
Proceso 3: 79 es primo.
Proceso 3: 83 es primo.
Proceso 3: 85 es primo.
Proceso 3: 89 es primo.
Proceso 3: 91 es primo.
Proceso 3: 95 es primo.
Proceso 3: 97 es primo.
```

2.3.3. Con 5 procesos

```
Proceso 4: 5 es primo.
Proceso 4: 7 es primo.
Proceso 4: 11 es primo.
Proceso 4: 13 es primo.
Proceso 4: 17 es primo.
Proceso 4: 19 es primo.
Proceso 4: 23 es primo.
Proceso 4: 29 es primo.
Proceso 4: 31 es primo.
Proceso 4: 37 es primo.
Proceso 4: 41 es primo.
Proceso 4: 43 es primo.
Proceso 4: 47 es primo.
Proceso 4: 49 es primo.
Proceso 4: 53 es primo.
Proceso 4: 59 es primo.
Proceso 4: 61 es primo.
Proceso 4: 67 es primo.
Proceso 4: 71 es primo.
Proceso 4: 73 es primo.
Proceso 4: 77 es primo.
Proceso 4: 79 es primo.
Proceso 4: 83 es primo.
Proceso 4: 89 es primo.
Proceso 4: 91 es primo.
Proceso 4: 97 es primo.
```

2.3.4. Con 10 procesos

```
Proceso 9: 19 es primo.
Proceso 9: 23 es primo.
Proceso 9: 29 es primo.
Proceso 9: 31 es primo.
Proceso 9: 37 es primo.
Proceso 9: 41 es primo.
Proceso 9: 43 es primo.
Proceso 9: 47 es primo.
```

```
Proceso 9: 53 es primo.
Proceso 9: 59 es primo.
Proceso 9: 61 es primo.
Proceso 9: 67 es primo.
Proceso 9: 71 es primo.
Proceso 9: 73 es primo.
Proceso 9: 79 es primo.
Proceso 9: 83 es primo.
Proceso 9: 89 es primo.
Proceso 9: 97 es primo.
```

2.3.5. Con 20 procesos

```
Proceso 19: 61 es primo.
Proceso 19: 67 es primo.
Proceso 19: 71 es primo.
Proceso 19: 73 es primo.
Proceso 19: 79 es primo.
Proceso 19: 83 es primo.
Proceso 19: 89 es primo.
Proceso 19: 97 es primo.
```

2.3.6. Justificación del comportamiento según el número de procesos

El comportamiento del programa varía significativamente dependiendo del número de procesos utilizados en la ejecución. Este fenómeno puede explicarse en términos de cómo los procesos filtran números no primos y cómo se organiza la comunicación entre ellos.

Con un menor número de procesos (por ejemplo, 3): Cuando el número de procesos es bajo, la cantidad de filtros disponibles es insuficiente para eliminar todos los números compuestos (no primos). Por ejemplo, con 3 procesos:

- El primer proceso (`rank = 1`) filtra números divisibles por 2.
- El segundo proceso (`rank = 2`) filtra números divisibles por 3.

Sin embargo, no hay suficientes procesos para filtrar números compuestos mayores, como los divisibles por 5, 7, etc. Como resultado, algunos números no primos pasan a través del sistema y son enviados al impresor. Esto genera una salida con más números, pero no todos son primos.

Con un mayor número de procesos: A medida que se incrementa el número de procesos, se añaden más filtros, cada uno dedicado a eliminar los números divisibles por un número primo específico. Por ejemplo:

- Un proceso adicional (`rank = 3`) filtra los números divisibles por 5.
- Otro proceso (`rank = 4`) filtra los números divisibles por 7.

Con más procesos, los números compuestos tienen más probabilidades de ser eliminados antes de llegar al impresor, lo que garantiza que la salida sea exclusivamente números primos. Sin embargo, este aumento de precisión también implica que se procesen menos números, ya que cada filtro elimina una mayor cantidad de candidatos.

2.4. Demostración Matemática de la Correctitud del Algoritmo

El algoritmo implementado se basa en la Criba de Eratóstenes, un método clásico para encontrar números primos en un rango dado. A continuación, demostramos matemáticamente que este enfoque es correcto.

2.4.1. Definición de Número Primo

Un número $n > 1$ es primo si y solo si no es divisible por ningún número entero k tal que $2 \leq k \leq \sqrt{n}$. Es decir, no existen divisores propios de n en el rango mencionado.

2.4.2. Esquema del Algoritmo

El algoritmo utiliza un conjunto de procesos para filtrar números no primos de manera iterativa:

1. El primer proceso (`rank == 0`) genera números consecutivos x comenzando desde 2 y los envía al primer filtro.
2. Cada proceso filtro P_i conserva un número primo p_i, recibido del proceso anterior. Filtra los números x recibidos eliminando aquellos que satisfacen $x \mod p_i = 0$ (es decir, los múltiplos de p_i).
3. Los números que no son múltiplos de p_i son enviados al siguiente filtro.
4. Finalmente, el último proceso (`rank == size - 1`) imprime todos los números que llegan hasta él, ya que no han sido descartados por ningún filtro previo.

2.4.3. Correctitud del Algoritmo

Para demostrar que este algoritmo es correcto, debemos probar que:

1. **Todo número primo es identificado correctamente.**
2. **Todo número compuesto es eliminado correctamente.**

Identificación de números primos Por definición, un número primo p no tiene divisores propios en el rango $[2, \sqrt{p}]$. En el algoritmo:

- El primer filtro retiene 2, el siguiente retiene 3, y así sucesivamente.
- Un número p que llega al filtro P_i no es divisible por $p_1, p_2, \ldots, p_{i-1}$, ya que estos filtros han descartado previamente los múltiplos de sus primos asociados.
- Si p llega al último filtro, entonces p no tiene divisores en $[2, \sqrt{p}]$, lo que implica que es primo.

Por lo tanto, el algoritmo identifica todos los números primos correctamente.

Eliminación de números compuestos Sea n un número compuesto. Por definición, n tiene al menos un divisor d tal que $2 \leq d \leq \sqrt{n}$. En el algoritmo:

- El filtro correspondiente al primo d descarta n, ya que $n \mod d = 0$.
- Como $d \leq \sqrt{n}$, el filtro asociado a d se encuentra antes de los procesos que manejan n, asegurando su eliminación.

Por lo tanto, ningún número compuesto llega al último proceso impresor.

2.4.4. Conclusión del Análisis

El algoritmo implementado en MPI garantiza que:

- Todos los números primos dentro del rango son impresos.
- Ningún número compuesto es identificado como primo.

Esto demuestra matemáticamente que el algoritmo es correcto y cumple con los requisitos de la Criba de Eratóstenes para la identificación de números primos, por lo tanto queda demostrado el algoritmo de la Criba de Eratóstenes.