DESARROLLO DE SOFTWARE EN ARQUITECTURAS PARALELAS

PRÁCTICA 2: PRIMOS

La práctica se ENTREGARÁ en el directorio:

/home/CUENTA/PRIMOS (CUENTA=cuenta personal).

Los ficheros o directorios que sea necesario crear pero no vayan a ser entregados al profesor deberán crearse directamente en el directorio /home/CUENTA y no dentro del subdirectorio de prácticas PRIMOS.

En esta práctica se desea que se realice la implementación paralela, sobre MPI, para obtener el número de enteros primos menores que uno dado n.

Se dice que un entero positivo p es primo si tiene exactamente dos divisores positivos distintos. Por ejemplo:

- Los divisores positivos de 13 son 1 y 13. Luego 13 es primo.
- El entero 15 no solo tiene como divisores positivos el 1 y 15. Tiene otros divisores, el 3 y el 5. Luego 15 no es primo.
- El entero 1 solo tiene un divisor positivo: el propio 1. Luego 1 no es primo.

Los algoritmos existentes para determinar si un entero es primo tienen el suficientemente coste computacional como para poder incluir paralelismo. Si a esto le añadimos que este cálculo se debe realizar muchas veces, el coste computacional se incrementa.

Un algoritmo básico para obtener todos los primos menores que `n` consiste en comprobar todos los enteros entre 2 y `n`, es decir,

Algoritmo número de primos

```
1        total_primos = 0;
2        for (int i = 2; i <= n; i++)
3           if (esPrimo(i) == 1)
4              total_primos++;
```

Donde `esPrimo(m)` representa un algoritmo básico para determinar si un entero `m`, mayor que 1, es primo. La función `esPrimo(m)` busca posibles divisores del entero `m` entre 2 y $\sqrt{m}$:

```
int esPrimo(int m) {
   for (int i = 2; i <= sqrt(m); i++)
      if (m%i == 0)
         return 0;
   return 1;
}
```

Si disponemos de un total de `nproc` procesos trabajando en paralelo, el **Algoritmo número de primos** anterior puede ser fácilmente paralelizado teniendo en cuenta que los cálculos representados por el bucle (2) son independientes. En consecuencia, lo único que debemos plantearnos es una estrategia para distribuir la carga de trabajo entre los procesos.

La distribución de los cálculos representados por el bucle (2) se realizará teniendo en cuenta una distribución cíclica. Por ejemplo, si `n=100` y `p=4`:

- El proceso 0 evalúa los enteros $2, 6, 10, \ldots$
- El proceso 1 evalúa los enteros $3, 7, 11, \ldots$
- El proceso 2 evalúa los enteros $4, 8, 12, \ldots$
- El proceso 3 evalúa los enteros $5, 9, 13, \ldots$

De esta forma, el código que desarrollaría un proceso, de un total de `nprocs`, determinado por su rango `myrank` ($0 \leq$ `myrank` $<$ `nprocs`), sería:

```
total_primos = 0;
for (int i = 2 + myrank; i <= n; i = i + nprocs)
   if (esPrimo(i) == 1)
      total_primos++;
```

La implementación en paralelo debe seguir un esquema similar a la secuencial. En concreto, dados tres enteros:

n_min Valor mínimo de `n` (=500),

n_max Valor máximo de `n` (=5000000) y

n_factor Factor de incremento para `n` (=10).

Se desea obtener el número de enteros primos para distintos valores de n. En concreto, los valores de `n` estarán comprendidos entre `n_min` y `n_max`, empezando por `n=n_min`, continuando por `n = n·n_factor`, sin sobrepasar `n_max`. Por ejemplo, para los valores indicados, se calculará el número de enteros primos para los valores de `n` = 500, 5000, 50000, 500000, 5000000 (ver versión secuencial).

Al igual que en el caso secuencial, el cálculo del número de enteros primos menores que `n` debe efectuarse, por cada proceso, dentro de una función que llamaremos `numero_primos()`. De esta forma, cada proceso calculará un número de primos local que deberá ser sumado en el proceso `root` mediante la función `MPI_Reduce()`. El proceso `root` calculará el tiempo total necesario incluyendo la recepción de los resultados parciales.

Cada proceso (incluyendo el `root`) calculará, además, el tiempo que ha necesitado para obtener el número de enteros primos locales. Al finalizar, estos tiempos se remitirán al proceso `root`. El envío y recepción de estos tiempos parciales no debe contabilizarse en el cálculo del tiempo total que realiza el proceso `root`.

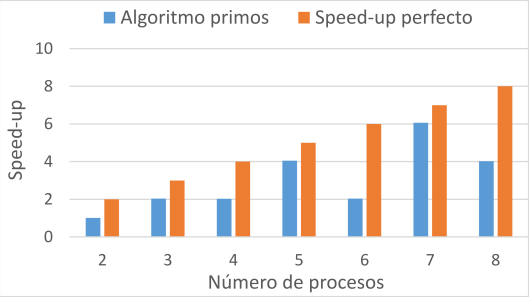
Adicionalmente, antes de iniciar el paralelismo, el proceso `root` calculará de forma secuencial el número de enteros primos menores que `n`, midiendo el tiempo secuencial, para poder calcular, con posterioridad, el speed-up y la eficiencia.

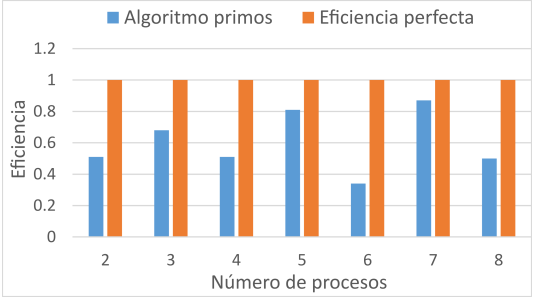
Al finalizar, el proceso `root` debe escribir por pantalla los siguientes datos (se muestra un ejemplo de ejecución para `n=5000` y `nprocs = 3` sin indicar tiempos reales):

Primos menores que 5000: 669. Tiempo secuencial: 6.00 s.
Primos menores que 5000: 669. Tiempo paralelo : 2.00 s. Tiempos parciales: 2.00 1.99 2.00
Speed-up: 3.00, Eficiencia: 1.00

Memoria a entregar: Se debe entregar una memoria que contenga:

1. Explicación de los pasos seguidos con comentarios del programa.
2. ¿Por qué los tiempos de algunos procesos son bastante inferiores a los del resto?
3. Gráficas comentadas de las mediciones obtenidas del speed-up y la eficiencia. Las gráficas se deben obtener para los distintos valores de `n` y para 2, 3, 4, 5, 6, 7 y 8 procesos. Por ejemplo, unas gráficas para $n = 5000000$ deben contener la siguiente información:





Para estas gráficas, se debe contestar a las siguientes preguntas: ¿Por qué el speed-up y la eficiencia no se acercan, en especial en algunos casos, al speed-up y eficiencia perfectos? ¿Explica qué solución podrías adoptar?

Ficheros a entregar:

primos_mpi.c Contendrá la unidad principal y la función `numero_primos()` que devolverá el número de primos calculado por cada proceso siguiendo una distribución cíclica de los datos. También contendrá la función ya suministrada `esPrimo()` que debe utilizar la función `numero_primos()`.

makefile Makefile utilizado para la compilación.