Universidad de Costa Rica
Facultad de Ingeniería
Escuela de Ciencias de la Computación e Informática

CI-0117 Programación Paralela y Concurrente
Grupo 01
I Semestre

III Tarea programada: Números pares como suma de números primos

Profesor:
Francisco Arroyo

Estudiantes:
Rodrigo Vílchez Ulloa | B78292

10 de julio del 2020

Índice

1. Introducción

El objetivo de esta tarea es implementar la solución de manera serial, `PThread`, `OpenMP` y `MPI` para el problema de encontrar la representación de un número par mayor que cuatro como la suma de dos números primos. Independientemente de la solución, el resultado debe ser el más eficiente, especialmente en los casos en los que la implementación se realiza en paralelo, para poder comparar las cuatro soluciones con el mismo número de datos y poder escoger la que sería la implementación más óptima.

2. Objetivos

Números pares como suma de números primos:

- Construir un programa en C++ que resuelva el problema de representar números pares como suma de números primos de manera serial.
- Construir un programa en C++ que resuelva el problema de representar números pares como suma de números primos utilizando `PThreads`.
- Construir un programa en C++ que resuelva el problema de representar números pares como suma de números primos utilizando `OpenMP`.
- Construir un programa en C++ que resuelva el problema de representar números pares como suma de números primos utilizando `MPI`.

3. Descripción

Números pares como suma de números primos:

Se deben crear cuatro programas diferentes, uno para la implementación serial, otro para la implementación con `PThreads`, otro utilizando `OpenMP` y el último utilizando `MPI`. El programa debe leer de la entrada estándar un número n que será el límite superior del intervalo $]4, n]$. Para cada elemento de este conjunto, se debe encontrar una representación como suma de dos números primos. Además, debe medirse el tiempo que tarda cada programa en realizar esta tarea, pues este tiempo deberá compararse con el resto de implementaciones para elegir la solución más óptima.

4. Diseño

Números pares como suma de números primos

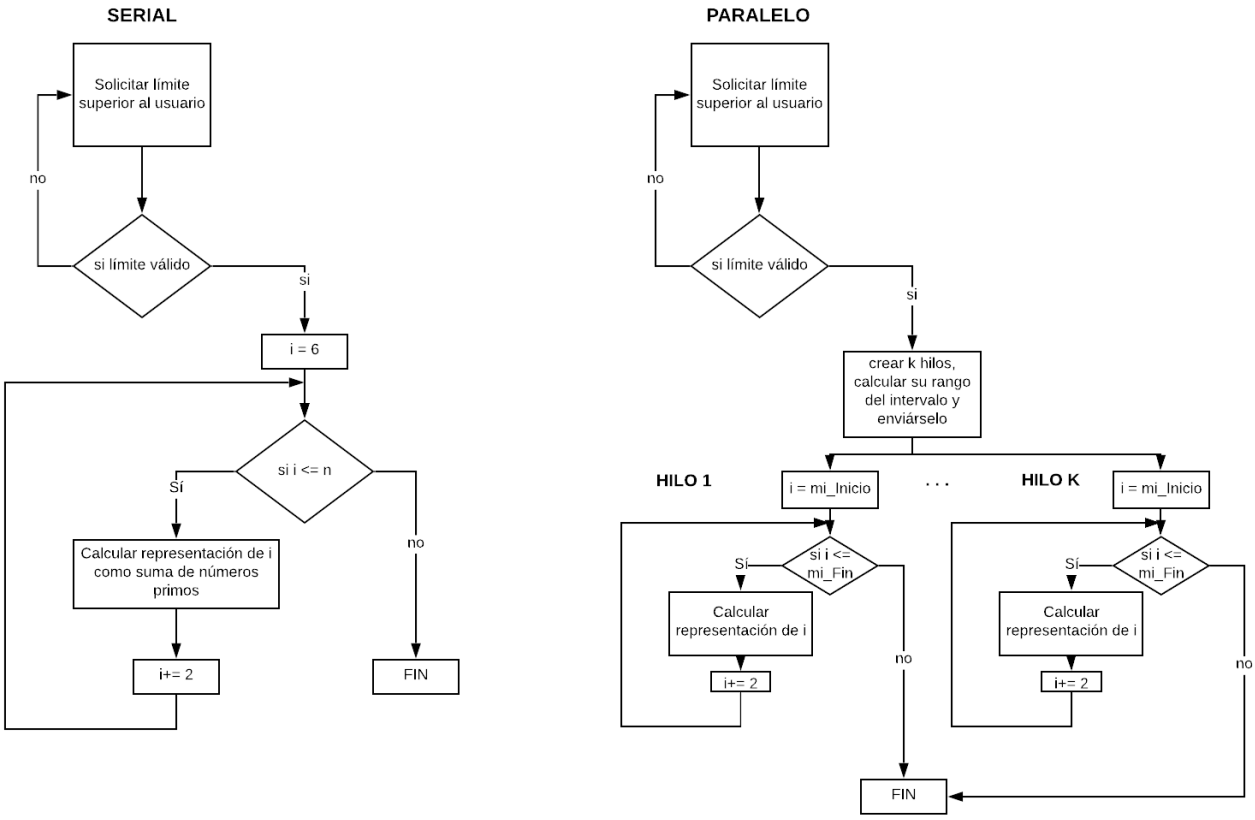


Figura 1: Diseño de la solución para Números pares como suma de números primos.

5. Desarrollo

Números pares como suma de números primos

- **Serial**:
  Para esta solución, el programa comienza solicitando el límite superior al usuario por medio de la entrada estándar y verifica que este límite sea válido, es decir: $n\%2 == 0$ y $n > 4$. Si el límite es correcto, el programa entra en un ciclo `for`, el cual comienza en $i = 6$, aumenta de dos en dos y termina hasta que $i <= n$. Cada i es enviado como parámetro a la función **primosSerial**, el cual se encarga de ir iterando entre dos valores mayores que 3 y evalúa si estos dos valores son primos y si su suma es igual al valor de i recibido como parámetro, en cuyo caso se imprime en pantalla esa representación.

- **PThread**:
  Para la solución con `PThread`, el programa comienza creando un array de tamaño NUM_HILOS, el cual es una macro definida al inicio del archivo. Este array es de tipo `std::thread`. El programa continúa solicitando el límite superior al usuario por medio de la entrada estándar y verifica que este límite sea válido, es decir: $n\%2 == 0$ y $n > 4$. Si el límite es correcto, el programa entra en un ciclo `for`, el cual itera desde $i = 0$ hasta que $i < NUM\_HILOS$, incrementando de uno en uno. Cada iteración representa un proceso que es creado para calcular una parte del intervalo previamente definido. Se calculan dos valores, los cuales son los extremos del intervalo que ese proceso debe procesar. Posteriormente se llama a `std::thread()`, el cual crea el proceso, enviando como parámetros a los dos extremos de su intervalo y corriendo la función `primosParalelo`, la cual es análoga a `primosSerial`, solo que en este caso, la función recibe dos números, los cuales representan los extremos del intervalo a procesar.

- **OpenMP**:
  Esta solución es similar a la implementada en la solución serial, la única diferencia es que el `for` es ejecutado como una sentencia pragma, esto es, que el programa divide las iteraciones del ciclo `for` entre una cantidad de trabajadores para que se procesen de forma paralela. La cantidad de trabajadores está definida en una macro NUM_HILOS y la asignación de las iteraciones a los trabajadores es estática.

- **MPI**:
  La solución utilizando `MPI` es la que más difiere del resto. El programa comienza ejecutando las funciones `MPI_Init`, `MPI_Comm_size` y `MPI_Comm_rank`, las cuales permiten almacenar la cantidad de trabajadores indicada al momento de correr el programa y de darle un identificador a cada trabajador. Si el id del trabajador es 0, este se tomará como trabajador principal, el cual le solicita al usuario un n como extremo del intervalo, posteriormente lo valida. Luego, este mismo trabajador se encarga de calcular los extremos de los intervalos que debe procesar el resto de los trabajadores, una vez calculados, el trabajador principal le envía estos dos valores al trabajador correspondiente mediante la función `MPI_Send` y `MPI_Recv`. Una vez hecho esto, cada trabajador, incluyendo el principal, procesa su intervalo para encontrar la representación de todos los números pares de su intervalo como suma de dos números primos. Una vez que todos los trabajadores hayan terminado, le enviarán un mensaje al trabajador principal indicando que han finalizado su tarea, de esta forma el trabajador principal, una vez que haya calculado su parte, espera a que el resto de trabajadores terminen y finalizará el programa.

6. Manual de usuario

Requerimientos de Software

- **Sistema Operativo:** Linux
- **Arquitectura:** 32/64 bits
- **Ambiente:** Terminal

Compilación

- **Contador de Etiquetas HTML**
  Para compilar los programas, se utiliza el comando `make`:

```
$ make $
```

Especificación de las funciones del programa

Para comenzar a correr uno de los cuatro programas, solamente se ejecuta el comando correspondiente:

```
$ ./primosSerial $
```

```
$ ./primosPThread $
```

```
$ ./primosOMP $
```

```
$ mpiexec -n NUM_HILOS ./primosMPI $
```

Es necesario recalcar que los programas no imprimen en pantalla la representación de cada número par, pues las líneas que imprimen el mensaje están comentadas, pues un intervalo lo suficientemente grande podría entorpecer el cálculo para encontrar el tiempo de ejecución del programa. Para mostrar en pantalla cada representación, lo único necesario es descomentar estas líneas, las cuales se encuentran casi al comienzo del programa, dentro de la función `primosSerial()` o `primosParalelo()`, dependiendo de cada caso.

7. Casos de Prueba

Prueba 1:

Se corren las cuatro distintas implementaciones, con 2 trabajadores en las soluciones paralelas y un $n = 8$, esto con el objetivo de mostrar la funcionalidad del programa.

```
rigovil@rodrigo:~/Escritorio/UCR/
 TP$ ./primosSerial
Ingrese el limite superior:
8
6 = 3 + 3
8 = 3 + 5
Tiempo: 0 seg, 433878382 nseg
rigovil@rodrigo:~/Escritorio/UCR/
 TP$ ./primosPThread
Ingrese el limite superior:
8
6 = 3 + 3
8 = 3 + 5
Tiempo: 0s, 582789988ns
rigovil@rodrigo:~/Escritorio/UCR/
 TP$ ./primosOMP
Ingrese el limite superior:
8
8 = 3 + 5
6 = 3 + 3
Tiempo: 0s, 710660691ns
rigovil@rodrigo:~/Escritorio/UCR/
 TP$ mpiexec -n 2 ./primosMPI
Ingrese el limite superior:
8
6 = 3 + 3
8 = 3 + 5
Tiempo: 0s, 308048231ns
```

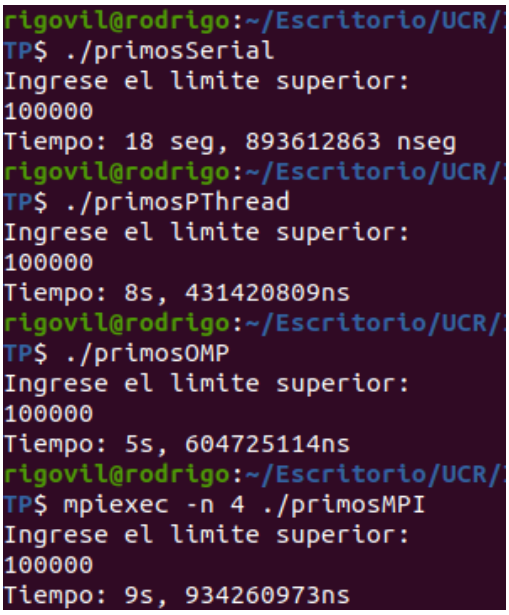
Prueba 2:

Se corren las cuatro distintas implementaciones, con 8 trabajadores en las soluciones paralelas y un $n = 50000$.

```
rigovil@rodrigo:~/Escritorio/UCR/
 TP$ ./primosSerial
Ingrese el limite superior:
50000
Tiempo: 4 seg, 726956348 nseg
rigovil@rodrigo:~/Escritorio/UCR/
 TP$ ./primosPThread
Ingrese el limite superior:
50000
Tiempo: 1s, 526603111ns
rigovil@rodrigo:~/Escritorio/UCR/
 TP$ ./primosOMP
Ingrese el limite superior:
50000
Tiempo: 1s, 217322420ns
rigovil@rodrigo:~/Escritorio/UCR/
 TP$ mpiexec -n 2 ./primosMPI
Ingrese el limite superior:
50000
Tiempo: 3s, 504500162ns
```

Prueba 3:

Se corren las cuatro distintas implementaciones, con 4 trabajadores en las soluciones paralelas y un $n = 100000$.



8. Comparación de las pruebas

Tablas de tiempos de duración

- Tamaño de datos: $n = 50000$

`Serial`	`Pthreads`	`OpenMP`	`MPI`	Trabajadores
4s, 518167168ns	3s, 529231651ns	2s, 891438492ns	3s, 537532076ns	2
4s, 518167168ns	2s, 974522966ns	1s, 807591948ns	2s, 515967800ns	4
4s, 518167168ns	1s, 632117952ns	1s, 296928070ns	1s, 964215376ns	8
4s, 518167168ns	1s, 823008658ns	1s, 653585795ns	2s, 516233153ns	16
4s, 518167168ns	1s, 718508748ns	1s, 295292878ns	1s, 987092080ns	32
4s, 518167168ns	1s, 671421775ns	1s, 678474483ns	1s, 995373967ns	64

- Tamaño de datos: $n = 100000$

`Serial`	`Pthreads`	`OpenMP`	`MPI`	Trabajadores
18s, 514935138ns	14s, 542521643ns	9s, 758637697ns	15s, 427254730ns	2
18s, 514935138ns	8s, 526936582ns	5s, 956124787ns	10s, 504125890ns	4
18s, 514935138ns	5s, 898312308ns	5s, 727516976ns	8s, 518789994ns	8
18s, 514935138ns	5s, 778442800ns	5s, 680402289ns	8s, 678757186ns	16
18s, 514935138ns	5s, 764609126ns	5s, 808271955ns	8s, 276221335ns	32
18s, 514935138ns	5s, 552529213ns	5s, 746974931ns	9s, 616242149ns	64

- Tamaño de datos: $n = 200000$

`Serial`	texttPthreads	texttOpenMP	texttMPI	Trabajadores
78s, 922789950ns	61s, 600311048ns	42s, 685726735ns	65s, 766564839ns	2
78s, 922789950ns	36s, 446489510ns	23s, 706572644ns	43s, 572752092ns	4
78s, 922789950ns	22s, 821788324ns	23s, 784141942ns	39s, 960247337ns	8
78s, 922789950ns	25s, 946726780ns	23s, 973639569ns	35s, 825095701ns	16
78s, 922789950ns	20s, 906164122ns	23s, 765950286ns	39s, 657364611ns	32
78s, 922789950ns	23s, 796191248ns	23s, 764606483ns	36s, 904244522ns	64

Análisis de los datos

Se utilizaron estos tamaños de n, pues una cantidad distinta podría tardar o muy poco, volviendo irrelevante la comparación, o mucho tiempo, que no es lo ideal para hacer pruebas cortas. Como se observa en las tablas, la solución más eficiente en la mayoría de los casos fue la de `OpenMP`, donde las mejores implementaciones se dieron cuando se utilizaron 8 trabajadores. Cada prueba tiene un tamaño de datos que es el doble que la prueba anterior con la intención de encontrar alguna relación entre los tiempos de duración, se muestra que, para cada implementación, si se dobla el tamaño de los datos, el tiempo de duración crece con un factor promedio de $4{,}5$ veces con respecto al tiempo anterior. Hay casos excepcionales, que se muestran en la última tabla: la implementación en `PThreads` y `OpenMP` con 2 trabajadores y para `PThreads` con 4 trabajadores, pues al parecer esta combinación de datos hace que el tiempo de duración sea considerablemente mayor en cuanto al tiempo de duración utilizado para la misma cantidad de datos con distinto número de trabajadores. Por último, se observa que la solución con `MPI` es muy mala en comparación con el resto de implementaciones paralelas si se utiliza un n bastante grande, aún así, la solución serial demuestra ser la peor, por lo que se justifica la implementación en paralelo para este problema.

FIN