Ingeniería de computadores 2023

Alejandro Pareja Penalva
15419139P

Contenido

Objetivos. 3
Presentación del problema propuesto. 3
Estudio del problema previo a su implementación. 4
Pseudocódigo. 5
Grafo de control de flujo. 6
Variables calientes, acceso a lectura/escritura. 7
Estudio de variación de la carga. 8
Ejercicios a resolver. 11
Bibliografía. 13

Objetivos.

- Revisar problemas y aplicaciones derivadas de cualquier rama del conocimiento, actuales o no, cuya carga computacional sea tan elevada como para justificar acometer un proceso de paralelización. Elegir una aplicación secuencial candidata a ser paralelizada.
- Defender mediante métricas ya conocidas en otras asignaturas por qué la aplicación en cuestión es idónea para su paralelización.
- Proponer arquitecturas idóneas para la paralelización de su aplicación.
- Estudiar cómo pueden afectar los parámetros de compilación al rendimiento de la aplicación en una máquina paralela.
- Aplicar métodos y técnicas propios de esta asignatura para estimar las ganancias máximas y la eficiencia del proceso de paralelización.

Presentación del problema propuesto.

Conteo de números primos.

El algoritmo pretende contar los números primos que hay hasta cierto número, por lo que, si por ejemplo, ponemos como número máximo el 100, va a contar los números primos que hay desde 0 hasta 100.

Igualmente si ponemos el 1000, el resultado del algoritmo va a ser la cantidad de números primos que hay entre el 0 y el 1000.

Estudio del problema previo a su implementación.

¿Por qué este código es un buen candidato a la paralelización?

Debido a que se verifica si cada número es primo o no, este es un buen ejemplo de algoritmo que implica una gran cantidad de datos y un cálculo intensivo para verificar si cada número en un rango es primo. Además, cuándo el límite superior es muy grande, la carga de trabajo se vuelve significativa y se beneficia del paralelismo.

También tenemos que tener en cuenta la independencia de datos que existe en este código, ya que la verificación de si un número es primo o no, es independiente de los otros números en el rango, por lo que cada iteración del bucle **for** que se realiza se puede realizar de forma independiente.

Teniendo en cuenta ese bucle **for**, la estructura para iterar a través de este bucle, es posible paralelizarlo, los hilos paralelos pueden trabajar en diferentes partes del rango sin conflictos.

Además, también usamos la claúsula “reduction” gracias a OpenMP, para realizar la operación de reducción simple en “count”.

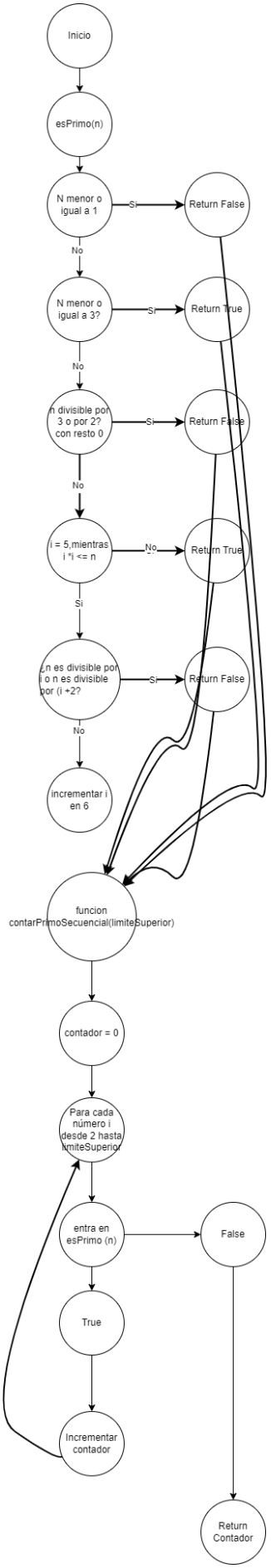
Pseudocódigo.

```
Función esPrimo(n)
    Si n es menor o igual a 1
        Devolver falso
    Fin Si
    Si n es menor o igual a 3
        Devolver verdadero
    Fin Si
    Si n es divisible por 2 o n es divisible por 3
        Devolver falso
    Fin Si
    i = 5
    Mientras i * i sea menor o igual a n
        Si n es divisible por i o n es divisible por (i + 2)
            Devolver falso
        Fin Si
        Incrementar i en 6
    Fin Mientras
    Devolver verdadero

Función contarPrimosSecuencial(límiteSuperior)
    contador = 0
    Para cada número i desde 2 hasta límiteSuperior
        Si esPrimo(i)
            Incrementar contador en 1
        Fin Si
    Devolver contador

límiteSuperior = 100000  # Parámetro que escala el problema
resultado = contarPrimosSecuencial(límiteSuperior)

Imprimir "Número de Números Primos hasta", límiteSuperior, ":", resultado

```

En esta captura anterior, podemos observar el pseudocódigo antes de la implementación del algoritmo de contar cuántos números primos hay entre 0 y el número puesto.

Grafo de control de flujo.

El grafo de flujo del pseudocódigo demuestra que el proceso de contar números primos hasta un límite superior se puede dividir en tareas independientes que pueden realizarse de forma simultánea. Esto sugiere la viabilidad de la paralelización de la búsqueda de números primos, lo que puede acelerar significativamente el proceso, especialmente cuando se trabaja con límites superiores muy grandes. La paralelización puede distribuir la carga de trabajo en múltiples hilos o núcleos de CPU, mejorando así el rendimiento y la eficiencia del cálculo de números primos.



Variables calientes, acceso a lectura/escritura.

- Count: Esta variable se utiliza para contar el número de números primos encontrados en el rango. Es una variable caliente, ya que se actualiza con frecuencia a medida que se encuentran números primos. La cláusula reduction asegura que las actualizaciones de count se realicen de manera segura en paralelo.

- i: Esta variable se utiliza como contador en el bucle que itera a través de los números en el rango. Es una variable caliente, ya que se incrementa en cada iteración.

- n: La variable n se utiliza como entrada para determinar si un número es primo o no. En la función isPrime, n se utiliza para realizar cálculos y comprobaciones para determinar la primalidad de un número. Es una **variable caliente**, ya que se utiliza para verificar si es primo o no, de cada número en el rango.

El uso de la directiva **#pragma omp parallel for** en el bucle principal permite que múltiples hilos trabajen en paralelo para contar números primos.

La cláusula **reduction** se utiliza para garantizar que los hilos actualicen la variable count de manera segura en paralelo.

La paralelización de este programa es una solución efectiva para acelerar el proceso de contar números primos, especialmente cuando se trabaja con un límite superior grande.

Cada hilo se encarga de un subconjunto del rango de números y verifica la primalidad de manera independiente. Luego, se combinan los resultados de manera segura mediante la reducción.

Respecto a los problemas de la caché, es posible, ya que la paralelización de bucles como en este programa puede ser un problema.

Cuándo se acceden y actualizan a varias variables e hilos, es posible que se produzcan conflictos de caché y problemas que puedan afectar al rendimiento.

Concretamente en este problema, puede ser que haya probelmas con “count” y con la variable “i” ya que se actualizan en bucles paralelos y e posible que pueda haber conflictos de escritura enter hilos que intentan actualizarla al mismo tiempo.

Aún así, contamos con la claúsula “reduction” que ayuda a mitigar ese problema, ya que garantiza que cada hilo mantenga su propia copia de “count” y luego se combine de manera segura al final del bucle.

Esto hace que los conflictos en la escritura variable compartida sean mínimos, pero aún así, como he dicho anteriormente, es posible que hayan.

En la variable “i” también se incrementa en cada iteración y no se actualiza directamente en el bucle en paralelo, pueden haber problemas de lectura y escritura en caché debido a los múltiples hilos que acceden a ella (aunque esto será en muy pocos casos).

Estudio de variación de la carga.

Muestra	Tiempo en segundos
100	**0,671**
10000	**0,785**
1000000	**0,974**
10000000	**7,835**
10000000000	**143,825**

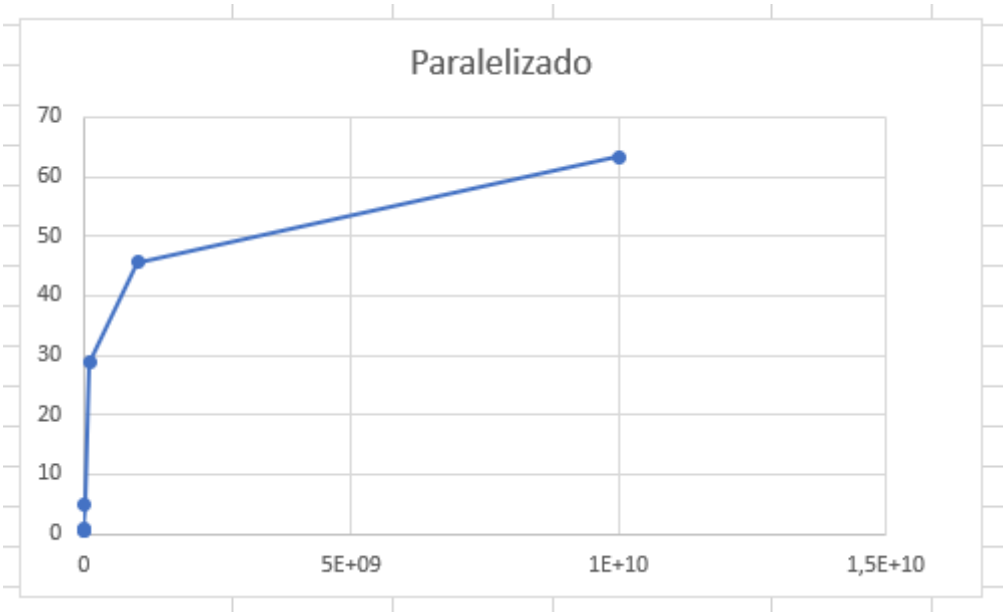
Como podemos ver en el código, en nuestro caso, el parámetro que varia la carga, es el parámetro upperlimit, el cuál es el que cuánto vemos que más alto, más carga para nuestro programa.

Opción	Descripción
-ffast-math	**Habilita optimizaciones que asumen que las operaciones matemáticas cumplen con las propiedades de la aritmética estándar. Puede acelerar cálculos matemáticos.**
-ftree-vectorize	**Habilita la vectorización automática de bucles, lo que permite el uso de instrucciones SIMD para mejorar el rendimiento.**
fopenmp	**Habilita el soporte para OpenMP, permitiendo la paralelización de bucles y secciones de código.**

Se pueden usar diferentes optimizaciones para la ejecución de nuestro código como:

-fopenmp -O2 → podemos usar este tipo de opciones siempre y cuándo estén habilitadas.

A continuación mostramos las dos gráficas de los resultados, tanto sin paralizar, como paralelizados:



A pesar de que a primera vista puedan parecer iguales, tenemos bastante diferencia en el eje Y, ya que como podemos ver, los valores que toma la gráfica sin paralelizar, son mucho más altos en cuánto a segundos de ejecución que la gráfica en la cuál el código se ha paralelizado.

Ejercicios a resolver.

- Un grifo tarda 4 horas en llenar un cierto depósito de agua y otro grifo 20 horas en llenar el mismo depósito. Si usamos los dos grifos para llenar el depósito, que está inicialmente vacío, ¿cuánto tiempo tardaremos? ¿Cuál será la ganancia en velocidad? ¿Y la eficiencia?

  Primer grifo -> depósito lleno = 4h = ¼ de depósito por hora.
  Segundo grifo -> depósito lleno = 20h = 1/20 de depósito por hora.
  Los dos grifos = ¼ + 1/20 = 6/20 = 3/10.
  Usando los dos grifos, 3/10 de depósito por hora.
  Para llenar el depósito tardaremos 3h y 6min.
  Para la ganacia en velocidad comparamos el más lento con el actual:
  20h – 3h y 6min = 16h y 54min. → 20/3.1 = 6.45 estos significa que se llena 6.45 veces más rápido.

- Suponga que tiene ahora 2 grifos de los que tardan 4 horas en llenar el depósito.Mismas cuestiones que el punto anterior.

  Primer grifo -> depósito lleno = 4h = ¼ de depósito por hora.
  Segundo grifo -> depósito lleno = 4h = ¼ de depósito por hora.
  Los dos grifos = ¼ +¼ =2/4.
  Usando los dos grifos, tardaremos 2h en llenarlo.
  Para la ganancia en velocidad comparamos el más lento con el actual:
  4h – 2h = 2h → 4/2 = 2, esto significa que se llena 2 veces más rápido.

- Y ahora suponga que tiene 2 grifos de los que tardan 20 horas. Proceda también a realizar los cálculos.

  Los dos grifos = 2/20 por hora = 1/10 por hora.
  Por lo que 10h para llenar un depósito.
  Ganancia en velocidad = Tiempo con un grifo / Tiempo con ambos grifos
  Ganancia en velocidad = 20 horas / 10 horas = 2

- Ahora tiene 3 grifos: 2 de los que tardan 20 horas y 1 de los que tardan 4. ¿Qué pasaría ahora?

  Grifo 1 (20 horas): Tasa de trabajo = 1/20 depósitos por hora.
  Grifo 2 (20 horas): Tasa de trabajo = 1/20 depósitos por hora.
  Grifo 3 (4 horas): Tasa de trabajo = 1/4 depósitos por hora.
  Tasa de trabajo total = (1/20) + (1/20) + (1/4) = 1/20 + 1/20 + 5/20 = 7/20 depósitos por hora
  Tiempo = 1 / Tasa de trabajo total
  Tiempo = 1 / (7/20) horas = 20/7 horas ≈ 2.857 horas
  Ganancia en velocidad = Tiempo con un grifo / Tiempo con tres grifos
  Ganancia en velocidad = 20 horas / 2.857 horas ≈ 6.992

Bibliografía.

https://vikman90.blogspot.com/2014/01/busqueda-paralela-de-numeros-primos.html

https://www.etsisi.upm.es/sites/default/files/asigs/arquitecturas_avanzadas/practicas/MPI/ejemplosparalelos.pdf

https://lsi2.ugr.es/jmantas/ppr/teoria/descargas/PPR_Tema2_ParalelizarTareas.pdf

https://docplayer.es/95440085-Practica-3-generacion-de-numeros-primos.html

https://es.wikipedia.org/wiki/Anexo:N%C3%BAmeros_primos