Números primos en un rango determinado

Bekier Lucas, Del Gobbo Julián

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

29 de octubre de 2017

Contenidos

Introducción al problema

- Un número primo es aquel que es divisible sólo por sí mismo y el 1.
- Existen muchas formas de calcular si un número en particular es primo.
  - Dividiendo el número por todos los anteriores hasta la raiz cuadrada y ver el resto en cada una.
  - Usar algoritmos probabilísticos como Miller Rabin que utilizan conceptos matemáticos más avanzados.
  - Calcular la criba de Eratóstenes hasta un número superior y consultar si lo es o no.

Introducción al problema

- Un número primo es aquel que es divisible sólo por sí mismo y el 1.
- Existen muchas formas de calcular si un número en particular es primo.
  - Dividiendo el número por todos los anteriores hasta la raiz cuadrada y ver el resto en cada una.
  - Usar algoritmos probabilísticos como Miller Rabin que utilizan conceptos matemáticos más avanzados.
  - Calcular la criba de Eratóstenes hasta un número superior y consultar si lo es o no.

Introducción al problema

- Un número primo es aquel que es divisible sólo por sí mismo y el 1.
- Existen muchas formas de calcular si un número en particular es primo.
  - Dividiendo el número por todos los anteriores hasta la raiz cuadrada y ver el resto en cada una.
  - Usar algoritmos probabilísticos como Miller Rabin que utilizan conceptos matemáticos más avanzados.
  - Calcular la criba de Eratóstenes hasta un número superior y consultar si lo es o no.

Introducción al problema

- Un número primo es aquel que es divisible sólo por sí mismo y el 1.
- Existen muchas formas de calcular si un número en particular es primo.
  - Dividiendo el número por todos los anteriores hasta la raiz cuadrada y ver el resto en cada una.
  - Usar algoritmos probabilísticos como Miller Rabin que utilizan conceptos matemáticos más avanzados.
  - Calcular la criba de Eratóstenes hasta un número superior y consultar si lo es o no.

Introducción al problema

- Un número primo es aquel que es divisible sólo por sí mismo y el 1.
- Existen muchas formas de calcular si un número en particular es primo.
  - Dividiendo el número por todos los anteriores hasta la raiz cuadrada y ver el resto en cada una.
  - Usar algoritmos probabilísticos como Miller Rabin que utilizan conceptos matemáticos más avanzados.
  - Calcular la criba de Eratóstenes hasta un número superior y consultar si lo es o no.

Introducción al problema

- La criba conviene usarla cuando se desea calcular todos los primos desde 1 hasta N.
- ¿ Pero qué pasa si queremos calcular los primos entre X e Z ?
- **La criba común no es la mejor elección.**

Introducción al problema

- La criba conviene usarla cuando se desea calcular todos los primos desde 1 hasta N.
- ¿ Pero qué pasa si queremos calcular los primos entre X e Z ?
- **La criba común no es la mejor elección.**

Introducción al problema

- La criba conviene usarla cuando se desea calcular todos los primos desde 1 hasta N.
- ¿ Pero qué pasa si queremos calcular los primos entre X e Z ?
- **La criba común no es la mejor elección.**

Introducción al problema

1. El algoritmo calcula todos los primos entre 1 e Z.
2. Desaprovechamos la propiedad matemática que dice que necesitamos solamente los primos hasta la raiz cuadrada de un número para determinar su primalidad.
3. Es un algoritmo completamente secuencial.

Introducción al problema

1. El algoritmo calcula todos los primos entre 1 e Z.
2. Desaprovechamos la propiedad matemática que dice que necesitamos solamente los primos hasta la raiz cuadrada de un número para determinar su primalidad.
3. Es un algoritmo completamente secuencial.

Introducción al problema

1. El algoritmo calcula todos los primos entre 1 e Z.
2. Desaprovechamos la propiedad matemática que dice que necesitamos solamente los primos hasta la raiz cuadrada de un número para determinar su primalidad.
3. Es un algoritmo completamente secuencial.

Como mejorar la performance?

Paralelismo Versión 1

- Proponemos el siguiente algoritmo naive que cuenta la cantidad de primos entre un número X e Z con un algoritmo naive.

-

```
dep: D(j) -> Y(j, i)
dep: Y(j,i) -> L(i)
dep: L(i) -> C
forall: X ≤ i ≤ Z
  par
    forall: 2 ≤ j ≤ √i
      D: d = j
    forall: d
      Y: y = divides(d, i)
    L: l = !any(y)
C: count(l)
```

- Como optimización se puede contemplar únicamente números impares entre los candidatos a divisores.

Paralelismo Versión 1

- Proponemos el siguiente algoritmo naive que cuenta la cantidad de primos entre un número X e Z con un algoritmo naive.
-

```
dep: D(j) -> Y(j, i)
dep: Y(j,i) -> L(i)
dep: L(i) -> C
forall: X ≤ i ≤ Z
  par
    forall: 2 ≤ j ≤ √i
      D: d = j
    forall: d
      Y: y = divides(d, i)
    L: l = !any(y)
C: count(l)
```

- Como optimización se puede contemplar únicamente números impares entre los candidatos a divisores.

Paralelismo Versión 1

- Proponemos el siguiente algoritmo naive que cuenta la cantidad de primos entre un número X e Z con un algoritmo naive.

-

```
        dep: D(j) -> Y(j, i)
        dep: Y(j,i) -> L(i)
        dep: L(i) -> C
        forall: X ≤ i ≤ Z
          par
            forall: 2 ≤ j ≤ √i
              D: d = j
            forall: d
              Y: y = divides(d, i)
            L: l = !any(y)
        C: count(l)
```

- Como optimización se puede contemplar únicamente números impares entre los candidatos a divisores.

Paralelismo Versión 2

- El siguiente algoritmo busca usar únicamente los primos hasta $\sqrt{Z}$ como posibles divisores.

-
```
seq
  P: p = primes(1, √Z)
  forall: X ≤ i ≤ Z
    forall: p
      Y: y = divides(p, i)
    L: l = !any(y)
  C: count(l)
```

- En este caso, *primes* es una función legacy que calcula los primos hasta la raiz, que si es lo suficientemente rápida genera mucha menor carga en la siguiente parte del algoritmo.

Paralelismo Versión 2

- El siguiente algoritmo busca usar únicamente los primos hasta $\sqrt{Z}$ como posibles divisores.

-

```
seq
  P: p = primes(1, √Z)
  forall: X ≤ i ≤ Z
    forall: p
      Y: y = divides(p, i)
    L: l = !any(y)
  C: count(l)
```

- En este caso, *primes* es una función legacy que calcula los primos hasta la raiz, que si es lo suficientemente rápida genera mucha menor carga en la siguiente parte del algoritmo.

Paralelismo Versión 2

- El siguiente algoritmo busca usar únicamente los primos hasta $\sqrt{Z}$ como posibles divisores.

-
```
seq
  P: p = primes(1, √Z)
  forall: X ≤ i ≤ Z
    forall: p
      Y: y = divides(p, i)
    L: l = !any(y)
  C: count(l)
```

- En este caso, *primes* es una función legacy que calcula los primos hasta la raiz, que si es lo suficientemente rápida genera mucha menor carga en la siguiente parte del algoritmo.

Paralelismo Versión 3

- En lugar de para cada número tratar de ver si es primo, hacer como la criba y marcar para cada número entre X e Y si uno lo divide.
- Uso de chunking para ganar velocidad.
- En pseudo FXML:

```
P =primes(1, √Z)
Parto el espacio en ((Z - X) / chunksize) chunks de tamano
    chunksize
forall: chunk
  forall: P
    Y: y = mark_multiples_in(chunk, P)
  T: t = sum(y)
C = sum(t)
```

Paralelismo Versión 3

- En lugar de para cada número tratar de ver si es primo, hacer como la criba y marcar para cada número entre X e Y si uno lo divide.
- Uso de chunking para ganar velocidad.
- En pseudo FXML:

```
P = primes(1, √Z)
Parto el espacio en ((Z - X) / chunksize) chunks de tamano
    chunksize
forall: chunk
  forall: P
    Y: y = mark_multiples_in(chunk, P)
  T: t = sum(y)
C = sum(t)
```

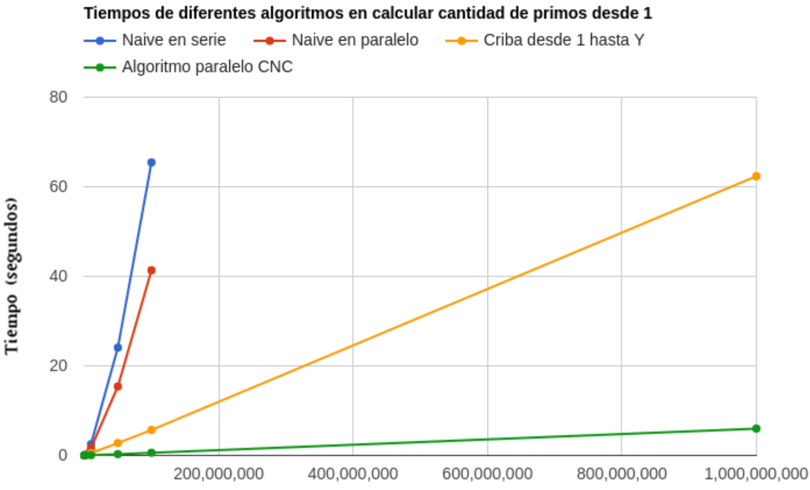
Paralelismo Versión 3

- En lugar de para cada número tratar de ver si es primo, hacer como la criba y marcar para cada número entre X e Y si uno lo divide.
- Uso de chunking para ganar velocidad.
- En pseudo FXML:

```
P = primes(1, √Z)
Parto el espacio en ((Z − X) / chunksize) chunks de tamano
    chunksize
forall: chunk
  forall: P
    Y: y = mark_multiples_in(chunk, P)
  T: t = sum(y)
C = sum(t)
```

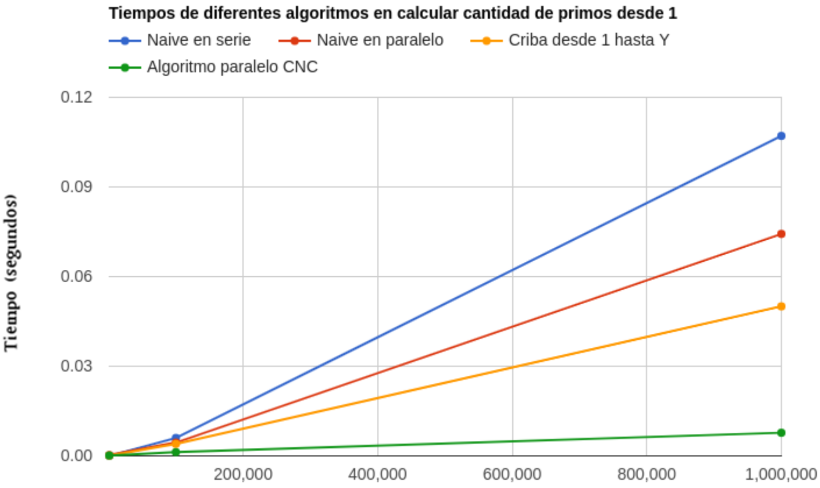
Grandes resultados

4 cores, 8 logical processors



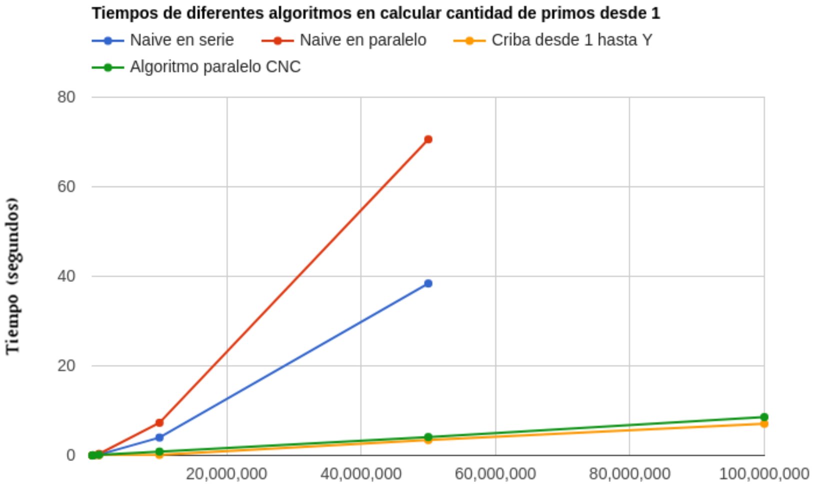
Grandes resultados

4 cores, 8 logical processors



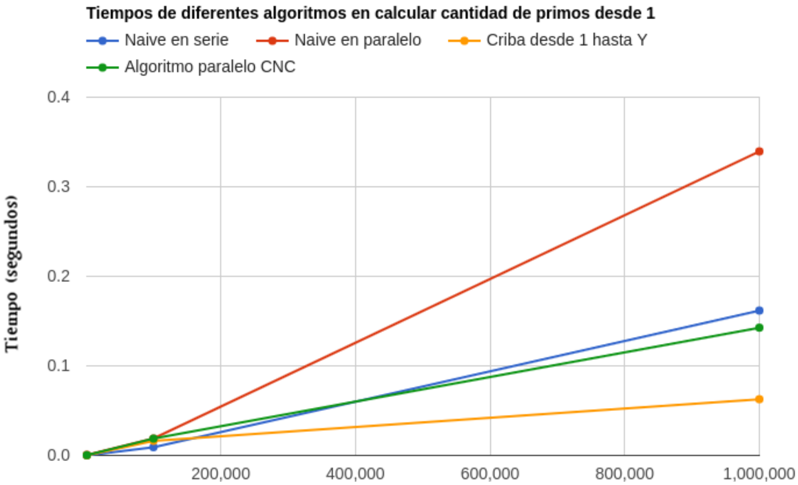
El overhead mata

2 cores, 2 logical processors



El overhead mata

2 cores, 2 logical processors



¿ Preguntas ?