# multiplicamos

```
837649587637 * 984758392081 = ?
```

# multiplicamos otra vez

```
837649587637 * 984758392081 = ?
824882461048724816302597
```

# algoritmo secuencial

- Asumimos que tengamos solamente las operaciones aritméticas *sumar* y *subtraer* disponibles en un procesador ficticio y queremos multiplicar dos números positivos.
- Un posible algoritmo sequencial que multiplica el número *p* con el número *q* produciendo el resultado *r* es:

```
Initially:  set p and q to positive numbers

a: set r to 0
b: loop
c:   if q equal 0 exitloop
d:   set r to r+p
e:   set q to q-1
f: endloop
g: ...
```

# ¿Cómo se comprueba si el algoritmo es correcto?

- Primero tenemos que decir que significa correcto.
- El algoritmo (secuencial) es correcto si
  - una vez se llega a la instrucción `g:` el valor de la variable `r` contiene el producto de los valores de las variables `p` y `q` (se refiere a sus valores que han llegado a la instrucción `a:`)
  - se llega a la instrucción `g:` en algún momento
  - y la entrada habia sido la correcta.

# ¿Cómo se comprueba si el algoritmo es correcto?

- Tenemos que saber que las instrucciones atómicas son correctas,
- es decir, sabemos exactamente su significado, incluyendo todos los efectos segundarios posibles.
- Luego usamos el concepto de inducción para comprobar el bucle.

# Invariante e inducción

- Sean $p_i$, $q_i$, y $r_i$ los contenidos de los registros `p`, `q`, y `r`, después de la iteración $i$ del bucle.
- Una invariante cuya corrección hay que comprobar con el concepto de inducción es entonces:

$$r_i + p_i \cdot q_i = p \cdot q$$

- ¿Cómo encontrar una invariante adecuada?
- usar *ingenio*...
  (RAE: Facultad del ser humano para discurrir o inventar con prontitud y facilidad.)
- Observa, si comprobamos que $q_i$ al final (saliendo del bucle) es cero, entonces, obviamente, el registro `r` contendrá el producto.

# Comprobación de la Invariante mediante inducción

*¡Inserta la comprobación visto en clase!*

# algoritmo concurrente

Re-escribimos el algoritmo secuencial para que "*funcione*" con dos procesos:

```
Initially:  set p and q to positive numbers

a: set r to 0
 P0                              P1
b: loop                            loop
c:   if q equal 0 exit               if q equal 0 exit
d:   set r to r+p                    set r to r+p
e:   set q to q-1                    set q to q-1
f: endloop                         endloop
g: ...
```

# Implementamos la multiplicación con Java

- Realizamos una clase para valores enteros.
- Implementamos el bucle que realiza la multiplicación.
- Colocamos todo en un programa completo.

# Enteros como objectos

Realizamos una clase para tener los enteros como clase propia (`Integer` no nos vale):

```
class Int {
  int i;
  Int(int i) { this.i=i; }
  void Add(Int I) { i=i+I.i; }
  int Get() { return i; }
}
```

# Preparar trabajador

Preparamos los hilos trabajadores con acceso a las variables comunes:

```
class Mul implements Runnable {
  private int id; // thread identity
  private Int p;  // reference to shared first factor
  private Int q;  // reference to shared second factor
  private Int r;  // reference to shared result

  Mul(int id, Int p, Int q, Int r) {
    this.id=id;
    this.p=p;
    this.q=q;
    this.r=r;
  }
```

# El trabajo del trabajador

Implementamos el método `run()` para realizar el bucle de multiplicación:

```
  public void run() {
    try {
      System.out.println("starting worker... "+id);
      Int minusOne=new Int(-1);
      while(q.Get()!=0) {
        r.Add(p);
        q.Add(minusOne);
      }
    }
    catch(Exception E) { System.out.println("??? "+id);
    finally { System.out.println("exiting... "+id); }
  }
}
```

# El programa principal I

Tratar la entrada:

```
class Multi {
  static Int p,q,r; // our shared variables for r=p*q

  public static void main(String[] args) {
    try {
      if(args.length!=3) {
        System.out.println("please 3 arg's: p q n");
        System.exit(1);
      }

      p=new Int(Integer.parseInt(args[0]));
      q=new Int(Integer.parseInt(args[1]));
      r=new Int(0);

      final int n=Integer.parseInt(args[2]);
```

# El programa principal II

Crear, lanzar, y sincronizar los trabajadores:

```
final Thread[] threads=new Thread[n];

System.out.println(
  p.Get()+"*"+q.Get()+" with "+n+" threads"
);

for(int i=0; i<threads.length; ++i) {
  threads[i]=new Thread(new Mul(i,p,q,r));
}

for(int i=0; i<threads.length; ++i) {
  threads[i].start();
}

for(int i=0; i<threads.length; ++i) {
  threads[i].join();
}
```

# El programa principal III

Visualización del resultado y terminar:

```
        System.out.println(
          args[0]+"*"+args[1]+"="+r.Get()+" ??"
        );
      }
      catch(Exception E) {
        System.out.println("catched an exception...");
        System.exit(1);
      }
      finally {
        System.out.println("exiting...");
      }
   }
}
```

# ¿Qué es lo que observamos?

- El algoritmo es no-determinista,
- en el sentido que no se sabe de antemano en qué orden (en un procesador o en un conjunto de procesadores) se van a ejecutar las instrucciones,
- o más preciso, cómo se van a intercalar las instrucciones atómicas de ambos procesos.
- El no-determinismo puede provocar situaciones con errores, es decir, el fallo ocurre solamente si las instrucciones se ejecutan en un orden específico.
- El resultado del programa no es predecible, y lo peor es, a veces es correcto.
- Desde el punto de vista de concurrencia tiene varios *problemas*.

# funcionamiento correcto I

Generalmente se dice que un programa es correcto, si dada una entrada, el programa produce los resultados deseados.
Más formal:

- Sea $P(x)$ una propiedad de una variable $x$ de entrada (aquí el símbolo $x$ refleja cualquier conjunto de variables de entradas).
- Sea $Q(x, y)$ una propiedad de una variable $x$ de entrada y de una variable $y$ de salida.

# funcionamiento correcto II

Se define dos tipos de funcionamiento correcto de un programa:

funcionamiento correcto parcial:

dada una entrada $a$, si $P(a)$ es verdadero, y si se lanza el programa con la entrada $a$, entonces si el programa termina habrá calculado $b$ y $Q(a, b)$ también es verdadero.

funcionamiento correcto total:

dado una entrada $a$, si $P(a)$ es verdadero, y si se lanza el programa con la entrada $a$, entonces el programa termina y habrá calculado $b$ con $Q(a, b)$ siendo también verdadero.

# funcionamiento correcto III

Para un programa secuencial existe solamente un orden total de las instrucciones atómicas (en el sentido que un procesador secuencial siempre sigue el mismo orden de las instrucciones... bueno, es mentira..., hay *out-of-order and speculative execution*), mientras que para un programa concurrente puedan existir varios órdenes.
Por eso se tiene que exigir:

funcionamiento correcto concurrente:

un programa concurrente funciona correctamente, si el resultado $Q(x, y)$ no depende del orden de las instrucciones atómicas entre todos los órdenes posibles.

# Funcionamiento correcto IV

Entonces:

- Se debe asumir que los hilos pueden intercalarse en cualquier punto en cualquier momento.
- Los programas no deben estar basados en la suposición de que habrá un intercalado específico entre los hilos por parte del planificador (que conmuta los procesos).

# funcionamiento correcto V

- Para comprobar si un programa concurrente es *incorrecto* basta con encontrar una sola intercalación de instrucciones que nos lleva en un fallo.
- Para comprobar si un programa concurrente es *correcto* hay que comprobar que no se produce ningún fallo en ninguna de las intercalaciones posibles.

# comprobación exhaustiva no es práctico

- El número de posibles intercalaciones de los procesos en un programa concurrente crece exponencialmente con el número de unidades que maneja el planificador y líneas por intercalar.
- ¿Cuántos son? (Ayuda: calcular las combinaciones posibles de una lista dentro de otra)
- Por eso es prácticamente imposible comprobar con la mera enumeración si un programa concurrente es correcto bajo todas las ejecuciones posibles.
- En la argumentación hasta ahora era muy importante que las instrucciones se ejecutaran de forma atómica, es decir, sin interrupción ninguna.
- Por ejemplo, se observa una gran diferencia si el procesador trabaja directamente en memoria o si trabaja con registros.

# dependencia de atomicidad I

Si `increment` es atómico:

```
P1:   inc N
P2:   inc N

P2:   inc N
P1:   inc N
```

Se observa: las dos intercalaciones posibles producen el resultado correcto.

# dependencia de atomicidad II

Si `increment` no es atómico:

```
P1:  load  R1,N
P2:  load  R2,N
P1:  inc   R1
P2:  inc   R2
P1:  store R1,N
P2:  store R2,N
```

Es decir, existe una intercalación que produce un resultado falso.
Ejemplos de Java:

- accesos a variables con más de 4 bytes no son atómicos.
- el operador ++ no es atómico.
- o en otras palabras: lectura y escritura de flotantes no es linearizable en Java

# Una mejora

Nuestro programa con dos hilos ejecutaba muy lento (e incorrecto). Hacemos una primera modificación: usamos como condición para `q` que sea mayor que cero.

```
  public void run() {
    try {
      System.out.println("starting worker... "+id);
      Int minusOne=new Int(-1);
      while(q.Get()>0) {
        r.Add(p);
        q.Add(minusOne);
      }
    }
    catch(Exception E) { System.out.println("??? "+id);
    finally { System.out.println("exiting... "+id); }
  }
}
```

Nota que sigue correcto en su version secuencial.

# ¿Y?

- ¿El algoritmo concurrente de multiplicación con dos hilos de arriba es correcto? (correcto parcial? correcto total?)
- ¿Cómo lo compruebas?
- ¿Te ocurre un algoritmo concurrente correcto que funcione con varios hilos?
- ¿Te ocurre un algoritmo concurrente correcto y eficiente, es decir, donde varios hilos juntos son más rápido que uno sólo?

# Multiplicación concurrente correcto

Tenemos cuidado que los hilos no hagan nada "demás"... es decir, comparamos con $q > n$:

```
public void run() {
  try {
    final Int minusOne=new Int(-1);
      // Here, we check "greater than n" to avoid negative q !!
    while(q.Get()>n) {
      r.Add(p);
      q.Add(minusOne);
    }
  }
  catch(Exception E) {
    System.out.println("some error..."+id);
  }
```

# Multiplicación concurrente correcto

Nos preocupamos que se realiza todo lo que hay que hacer... realizamos los posibles remanentes en el hilo principal después de la sincronización con las trabajadoras.

```
for(int i=0; i<threads.length; ++i) {
  threads[i].join();
}
  // Here we take care of the left-overs.
final Int minusOne=new Int(-1);
while(q.Get()>0) {
  r.Add(p);
  q.Add(minusOne);
}
```

# Reformulado...

- para conseguir un algoritmo correcto (aunque todavía no eficiente) teniamos que resolver dos problemas:
- la operación `Add` no tenía carácter de atomicidad, es decir, si se intercalaba con otra operación de otro hilo, el resultado fue no-determinista, y
- la condición del bucle y la operación en el bucle estaban acoplados, es decir, la semántica dependía de las acciones de los demás hilos (aunque cada operación individual sí era atómica)

estos dos variantes de problemas están conocidas como conceptos de *linerarizabilidad* y *serializabilidad*.

# Linearizabilidad

- Hemos usado hasta ahora la noción de atomicidad,
- es decir, si una operación (o transacción) se ejecuta en un instante, y el resultado de la propia operación no depende de los demás actores
- entonces la operación es linerarizable.
- Si tenemos tal propiedad para todas las operaciones entonces podemos decir que las operaciones (transacciones) son linearizables.
- Es una propiedad de sistemas concurrentes (y distribuidos): la tienen o no la tienen.
- Para conseguirlo se necesita necesariamente algún tipo de coordinación (en este momento lo asumimos como dado, Java funciona :-), pero en adelante vemos algo más profundo...).

# Linearizamos los accesos a q en nuestra multiplicación

# Linearizabilidad (en bases de datos)

- La linearizabilidad es un concepto importante también en el mundo de las bases de datos (de hecho viene de ahí),
- por ejemplo: pensad en un servicio de ficheros con acceso concurrente,
  - ¿qué transacciones con el servidor serían interesantes y cuales serían sus semánticas por implementar?
  - ¿Podemos implementar linearizabilidad de las transacciones?
  - ¿incluso en caso de ciertos tipos de fallos?
  - seguramente depende del tipo de aplicación que usa tal servicio...

# Serializabilidad

- Es un concepto más allá de la simple linearizabiliad, es decir, se quiere que la ejecución de las operaciones (o transacciones) linearizables de un sistema concurrente son equivalentes a un orden en un sistema secuencial.
- Si tal semántica de la ejecución en serie está bien conocida y se sabe que concurrentemente pasa lo mismo, entonces es más fácil de argumentar sobre la corrección.

Esta propiedad hemos conseguido con el cambio del algoritmo. Da lo mismo con qué intercalación los hilos ejecutan su bucle, al final (después del `join`) sabemos exactamente que ha pasado: `r` contiene el producto menos como mucho $n \cdot p$, y sabemos como corregir el producto con un bucle secuencial.

# ¿Se necesita siempre?

- Las condiciones de linearizabilidad y serializabilidad son en ciertas aplicaciones quizá demasiado restrictivas...
- por ejemplo: ¿por qué multiplicar correcto si al final se rondea? (sabíamos que el fallo como mucho era $n * p$.)
- Serializabilidad suele estar presente en bases de datos (clásicos) sobre todo si tratan con dinero.
- En adelante intentamos seguir haciendo programas correctos.

# un caso (ya no tan reciente) ...

Update (March 4 2014): During the investigation into stolen funds we have determined that the extent of the theft was enabled by a flaw within the front-end (???). The attacker logged into the flexcoin front end from IP address XXX under a newly created username and deposited to address XXX. The coins were then left to sit until they had reached 6 confirmations.
The attacker then successfully exploited a flaw in the code which allows transfers between flexcoin users. By sending thousands of simultaneous requests, the attacker was able to *move* coins from one user account to another until the sending account was overdrawn, before balances were updated. This was then repeated through multiple accounts, snowballing the amount, until the attacker withdrew the coins.

# ... un caso reciente ...

Flexcoin has made every attempt (???) to keep our servers as secure as possible, including regular testing. In our approx. 3 years of existence we have successfully repelled thousands of attacks. But in the end, this was simply not enough.
Having this be the demise of our small company, after the endless hours of work we've put in, was never our intent. We've failed our customers, our business, and ultimatley the Bitcoin community.

# ... un caso.

Look and smile:

```
mybalance = database.read("account-number")
newbalance = mybalance - amount
database.write("account-number", newbalance)
dispense_cash(amount)   // or send bitcoins to customer
```

(taken from http://hackingdistributed.com/2014/04/06/another-one-bites-the-dust-flexcoin/)

# ¡entendido!

- Una vez haber entendido esta problemática que se tiene que realizar ciertas acciones dentro de una aplicación concurrente de tal manera que un proceso puede actuar con exclusión mutua durante cierto tiempo sobre ciertos datos con el fin de realizar programas correctos,...
- ...vamos a ver algunos conceptos y herramientas a alto nivel para conseguirlo que extendemos luego a bajo nivel para ver, por lo menos unas pinceladas, como se realiza en sistemas reales.
- O en otras palabras: alguien tiene que construir ordenadores y programar máquinas virtuales de Java (o sistemas operativos).

# Uso de clases de `java.util.concurrent`

Java proporciona el paquete `java.util.concurrent`
Se puede implementar el algoritmo de multiplicación de antes con, por ejemplo:

- métodos sincronizados
- `AtomicInteger`
- `LongAdder` (a partir de Java8)
- `LongAccumulator` (a partir de Java8)

y un poco más adelante retomaremos este asunto con:

- semáforos (*semaphores*)
- cerrojos (*locks*)

y veremos que hay diferencias en eficiencia.

# Enteros con `Integer` y método sincronizado

```
// Our integer with an Integer
// and synchronized Add.
class Int {
  private Integer i;
  Int(int i) { this.i=i; }
  synchronized void Add(Int I) { i=i+I.i; }
  int Get() { return i; }
}
```

# Enteros con `AtomicInteger`

```
import java.util.concurrent.atomic.*;

// Our integer with an AtomicInteger.
class Int {
  private AtomicInteger i;
  Int(int i) { this.i=new AtomicInteger(i); }
  void Add(Int I) { i.getAndAdd(I.Get()); }
  int Get() { return i.get(); }
}
```

# Enteros con LongAdder

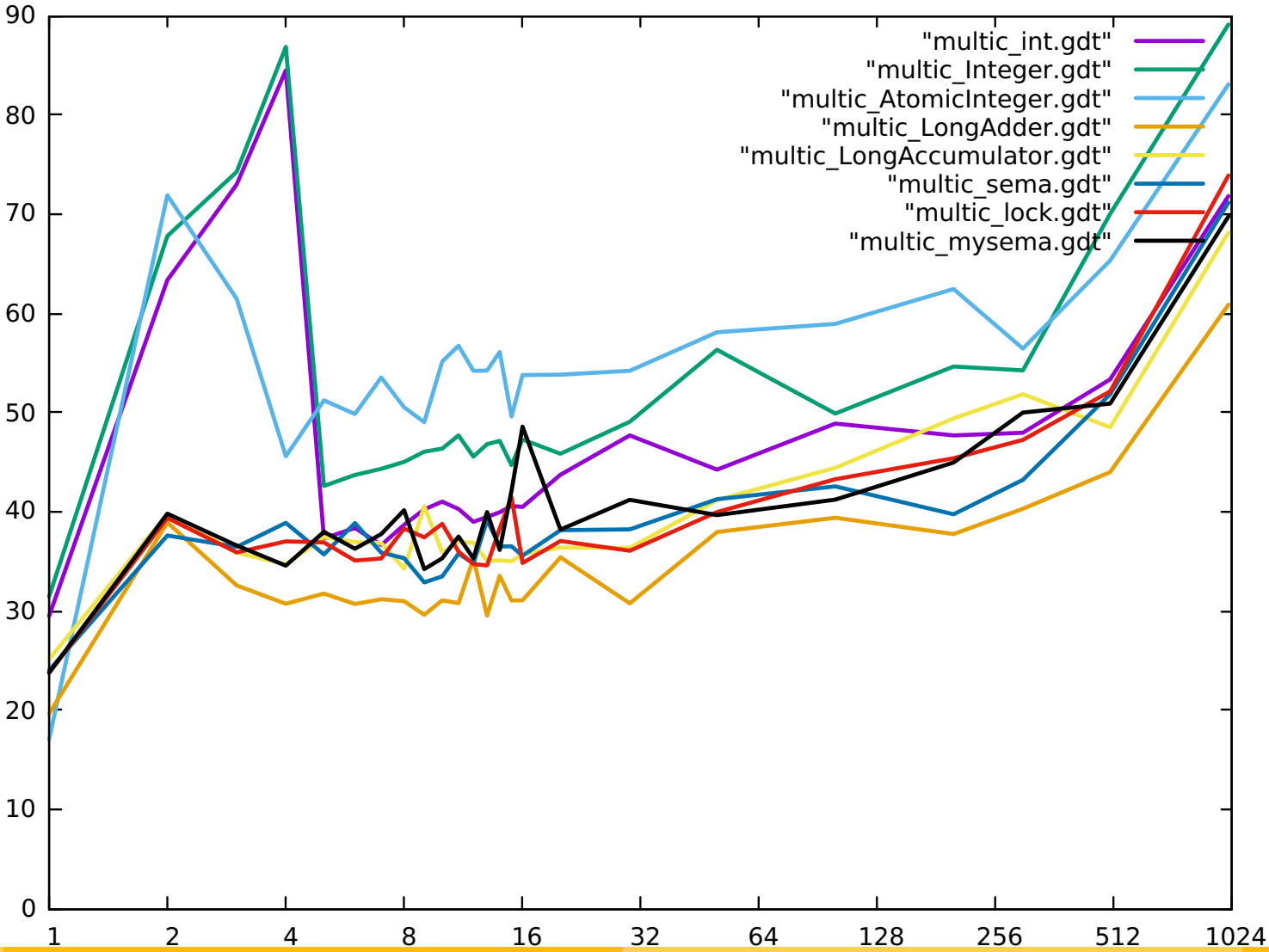
```
import java.util.concurrent.atomic.*;

// Our integer with a LongAdder.
class Int {
  private LongAdder i;
  Int(int i) {
    this.i=new LongAdder();
    this.i.add(i);
  }
  void Add(Int I) { i.add(I.Get()); }
  int Get() { return i.intValue(); }
}
```

# Enteros con `LongAccumulator`

```
import java.util.concurrent.atomic.*;

// Our integer with a LongAccumulator.
class Int {
  private LongAccumulator i;
  Int(int i) {
    this.i=new LongAccumulator(Long::sum, i);
  }
  void Add(Int I) { i.accumulate(I.Get()); }
  int Get() { return i.intValue(); }
}
```

# Multiplicación concurrente correcto

El algoritmo no es eficiente, ya que hay mucha congestión accediendo a las variable compartidas `q` y `r` con exclusión mutua:



# Multiplicación concurrente correcto y eficiente

```
class Mul implements Runnable {
  private final int id;
  private final int n;
  private int p;
  private int q;
  private Int r;

  Mul(int id, int n, Int p, Int q, Int r) {
    this.id=id;
    this.n=n;
    this.p=p.Get();
    this.q=q.Get()/n;
    this.r=r;
  }
```

# Multiplicación concurrente correcto y eficiente

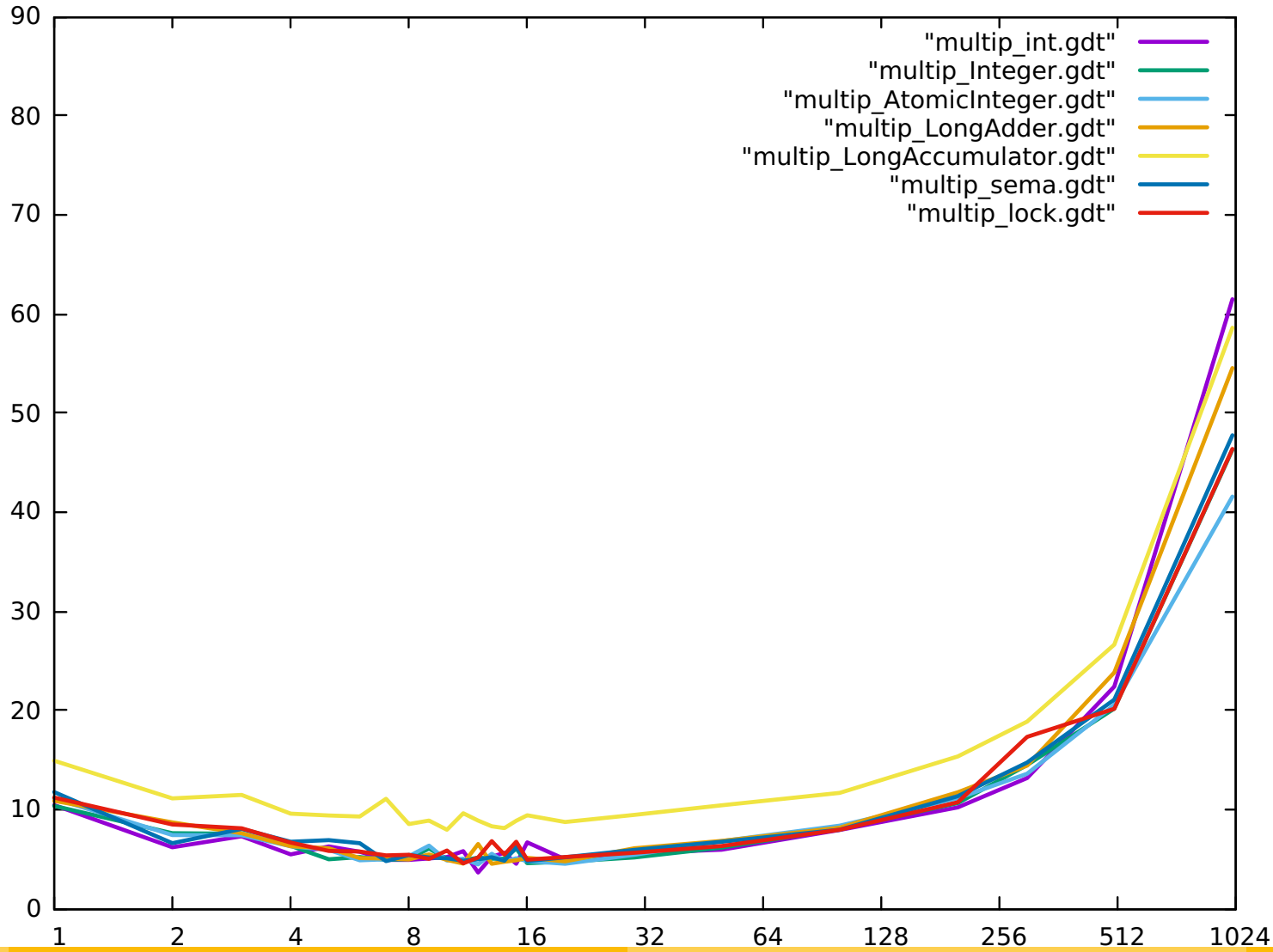
```
public void run() {
  try {
      // Here, we run on local variables.
    int local_r=0;
    while(q>0) {
      local_r+=p;
      --q;
    }
    final Int My_r=new Int(local_r);
    r.Add(My_r);
  }
  catch(Exception E) {
    System.out.println("some error..."+id);
  }
}
```

# Multiplicación concurrente correcto y eficiente

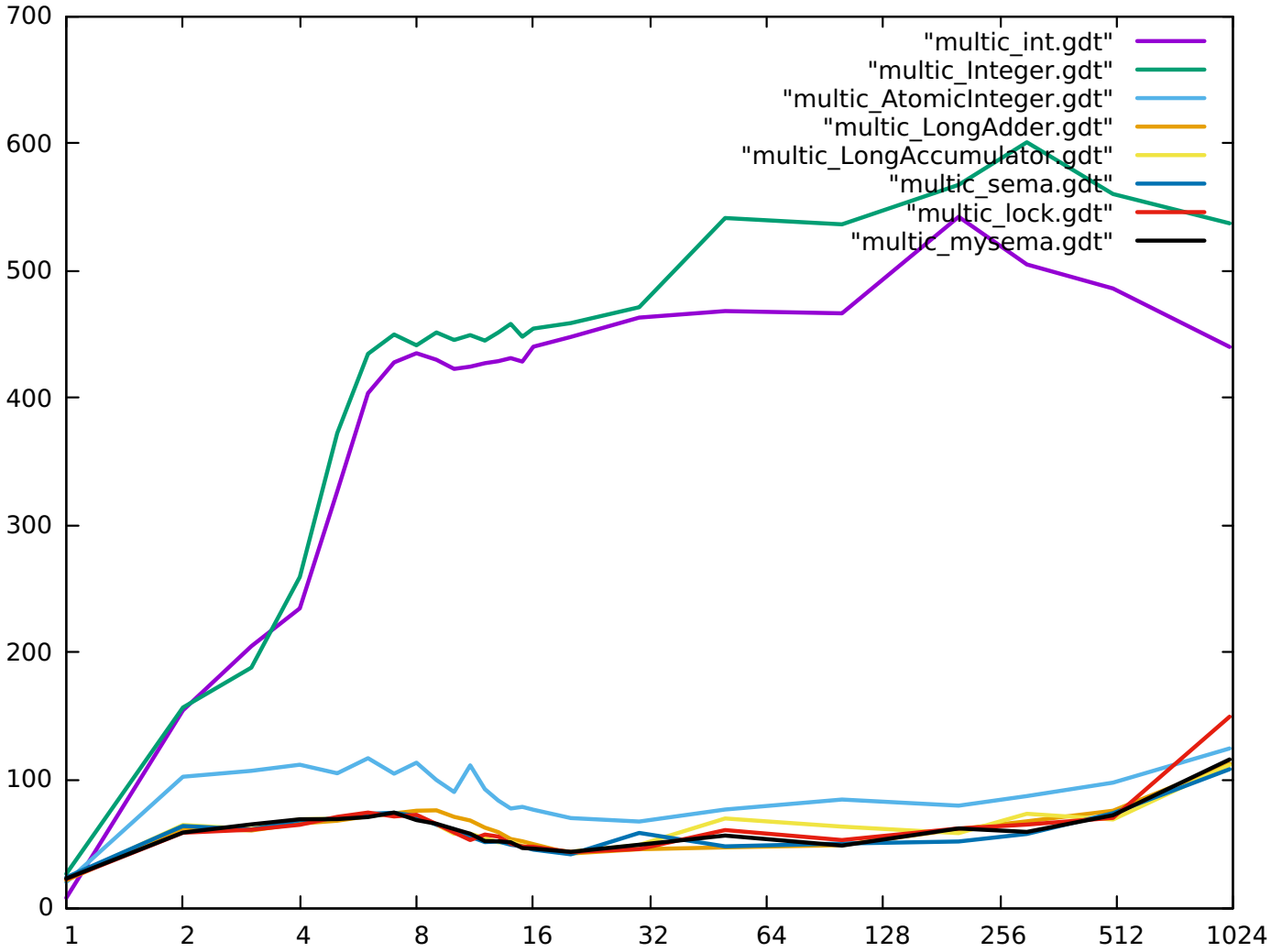
```
}
for(int i=0; i<threads.length; ++i) {
  threads[i].start();
}
  // Here we help with the left-overs.
int local_p=p.Get();
int local_q=q.Get()%n;
int local_r=0;
while(local_q>0) {
  local_r+=local_p;
  --local_q;
}

for(int i=0; i<threads.length; ++i) {
  threads[i].join();
}
final Int My_r=new Int(local_r);
```

# Multiplicación concurrente correcto y eficiente

Ahora hay mucho menos congestión... (unos 1000 veces más rápido, aumentamos $q$ por un factor de 100)

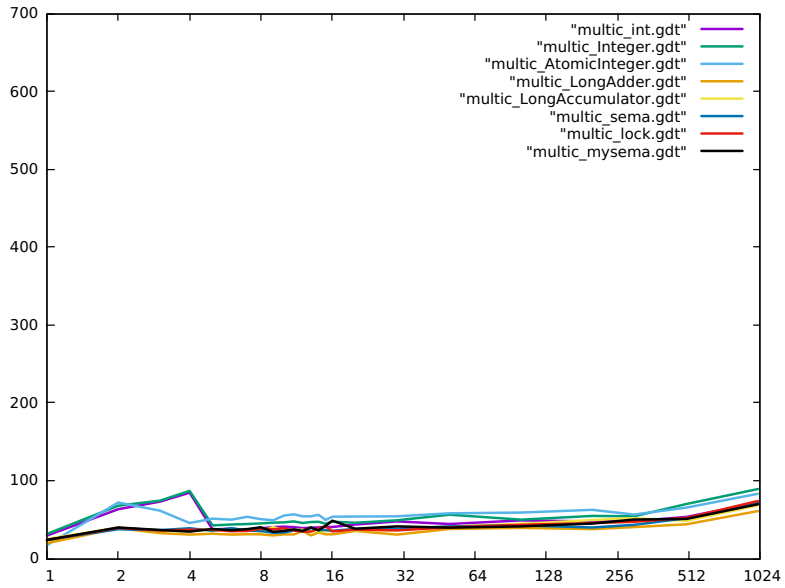


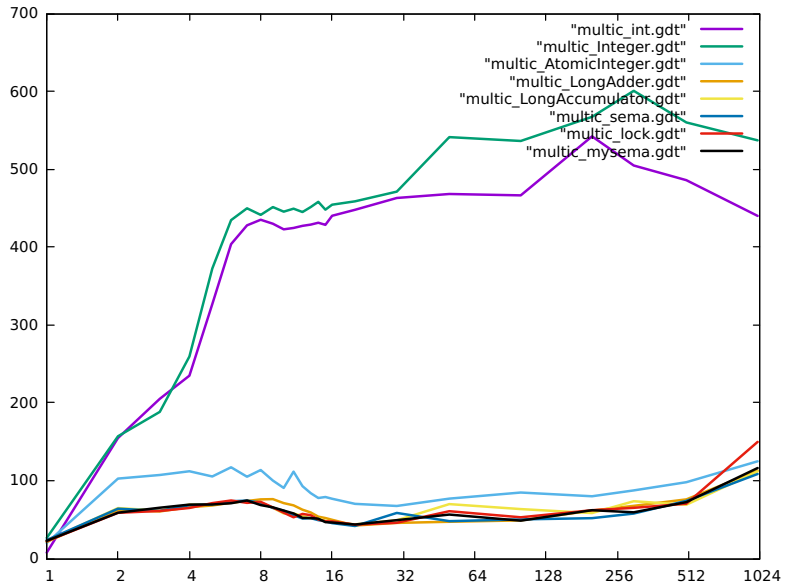
# Ojo, si no hacemos las cosas *bien*...




Es decir, nuestro primer algoritmo correcto pero ineficiente es muy lento en un sistema moderno si usamos `synchronized`.

# el sistema más potente se convierte en el más lento!




Es decir, si el programa concurrente no está bien hecho, es posible que un sistema moderno incluso es más lento que el viejo!
(Este efecto observé la primera vez en el año 2018.)