Concurrencia y Distribución

2022/2023

Tercero, GREI, Quinto, PCEO, Puente

Dr. Arno Formella

Departamento de Informática
Escola Superior de Enxeñaría Informática
Universidade de Vigo

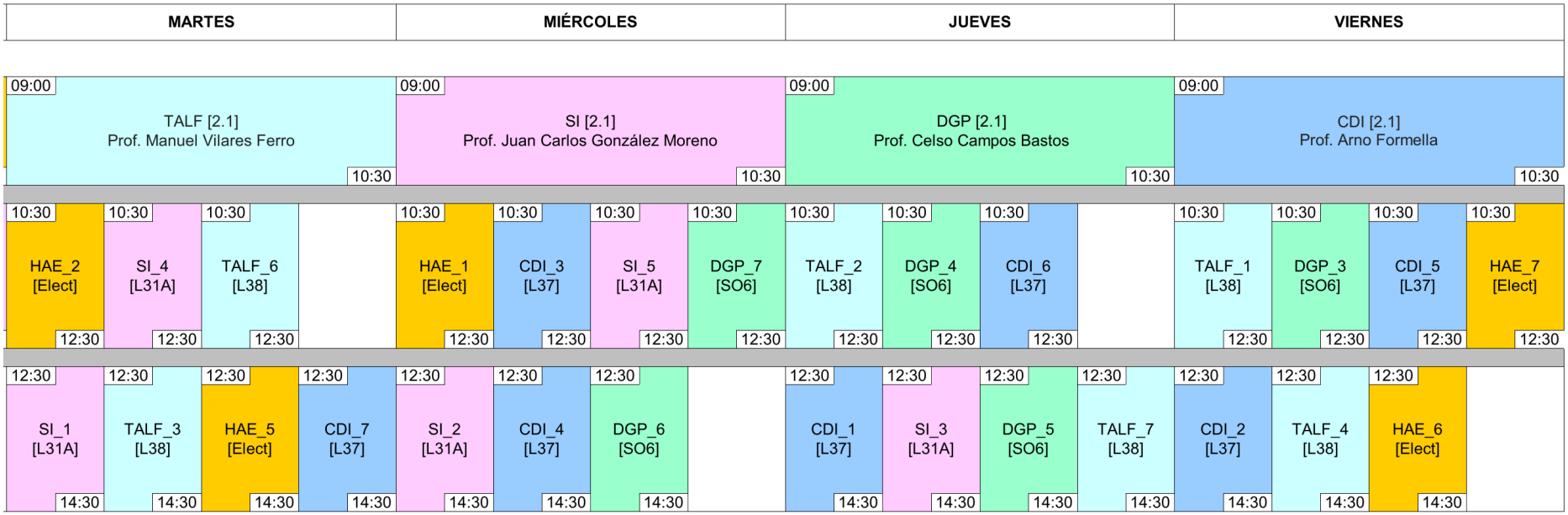
22/23

profesorado (teoría y prácticas)

- Arno FORMELLA
- Alba NOGUEIRA RODRÍGUEZ
- David RUANO ORDÁS
- cuarto/a todavía no se sabe
- http://formella.webs.uvigo.es
- MOODLE/MOOVI

horas presenciales

Teoría:	los viernes (13 sesiones), 9.00-10:30 horas presencial Aula 2.1
Prácticas:	7 grupos, 12 sesiones Lab37



horas presenciales

Luns	Martes	Mércores	Xoves	Venres	Sábado	Domingo
30/ene	31/ene	01/feb	02/feb	03/feb	04/feb	05/feb
06/feb	07/feb	08/feb	09/feb	10/feb	11/feb	12/feb
13/feb	14/feb	15/feb	16/feb	17/feb	18/feb	19/feb
20/feb	21/feb	22/feb	23/feb	24/feb	25/feb	26/feb
27/feb	28/feb	01/mar	02/mar	03/mar	04/mar	05/mar
06/mar	07/mar	08/mar	09/mar	10/mar	11/mar	12/mar
13/mar	14/mar	15/mar	16/mar	17/mar	18/mar	19/mar
20/mar	21/mar	22/mar	23/mar	24/mar	25/mar	26/mar
27/mar	28/mar	29/mar	30/mar	31/mar	01/abr	02/abr
03/abr	04/abr	05/abr	06/abr	07/abr	08/abr	09/abr
10/abr	11/abr	12/abr	13/abr	14/abr	15/abr	16/abr
17/abr	18/abr	19/abr	20/abr	21/abr	22/abr	23/abr
24/abr	25/abr	26/abr	27/abr	28/abr	29/abr	30/abr
01/may	02/may	03/may	04/may	05/may	06/may	07/may
08/may	09/may	10/may	11/may	12/may	13/may	14/may
15/may	16/may	17/may	18/may	19/may	20/may	21/may
22/may	23/may	24/may	25/may	26/may	27/may	28/may
29/may	30/may	31/may	01/jun	02/jun	03/jun	04/jun
05/jun	06/jun	07/jun	08/jun	09/jun	10/jun	11/jun

horas en aula presenciales

- 03.02. actividad introductoria, más algo...
- 12 sesiones magistrales
- 07.06. prueba final (10:00-14:00, 2++ horas)
- 14.07. prueba terminal (09:00-12:00, 2++ horas)
- 12 sesiones prácticas

horas del profesor (yo, aproximado, optimista)

220.0	horas anuales de docencia para CDI
	$(= 1600 \cdot 0.4 \cdot 134/160 \cdot 62.7/134)$
250.8	$-$ 19.5 horas presenciales teoría
231.3	$-$ 19.5 horas preparación clases teoría
211.8	$-$ 28 horas presenciales prácticas
183.8	$-$ 18 horas preparación de prácticas
165.8	$-\ 134/4 -$ 4 horas y corrección exámenes
128.3	$/134/15$ media por estudiante por semana, es decir
0.0638	horas por estudiante por semana, o
230	segundos por estudiante por semana

prerrequisitos

- matemáticas
- algoritmos y estructuras de datos
- todo sobre programación
- arquitectura de computadoras
- redes
- sistemas operativos
- algo de ofimática
- lenguajes de programación (Java, C++)

contexto en el plan de estudio

1/1S	1/2S	2/1S	2/2S	3/1S	3/2S	4/1S	4/2S
Fundamentos Matematicos	Tecnicas de Comunicacion	Enxenaria de Software I	Enxenaria de Software II	Interfaces de Usuarios	Direccion e Xestion de Proxectos	Aprendizaxe Baseada en Proxectos	Traballo Fin de Grao
Analise Matematica	Alxegra Lineal	Estadisticas	Bases de Datos I	Bases de Datos II	Sistemas Intelixentes	Optativa	
Programacion I	Algoritmos e Estruturas de Datos I	Algoritmos e Estruturas de Datos II	Redes de Computadoras I	Loxica para a Computacion	Teoria de Automatas e Lenguaxes	Optativa	Optativa
	Programacion II	Sistemas Operativos I	Sistemas Operativos II	Redes de Computadoras II	Concorrencia e Distribucion	Seguridade e Sistemas Informaticos	Optativa
Sistemas Digitales	Arquitectura de Computadoras I	Arquitectura de Computadoras II	Arquitecturas Paralelas	Centros de Datos	Hardware de Aplicacion Especifica	Optativa	Optativa

evaluación no-asistentes

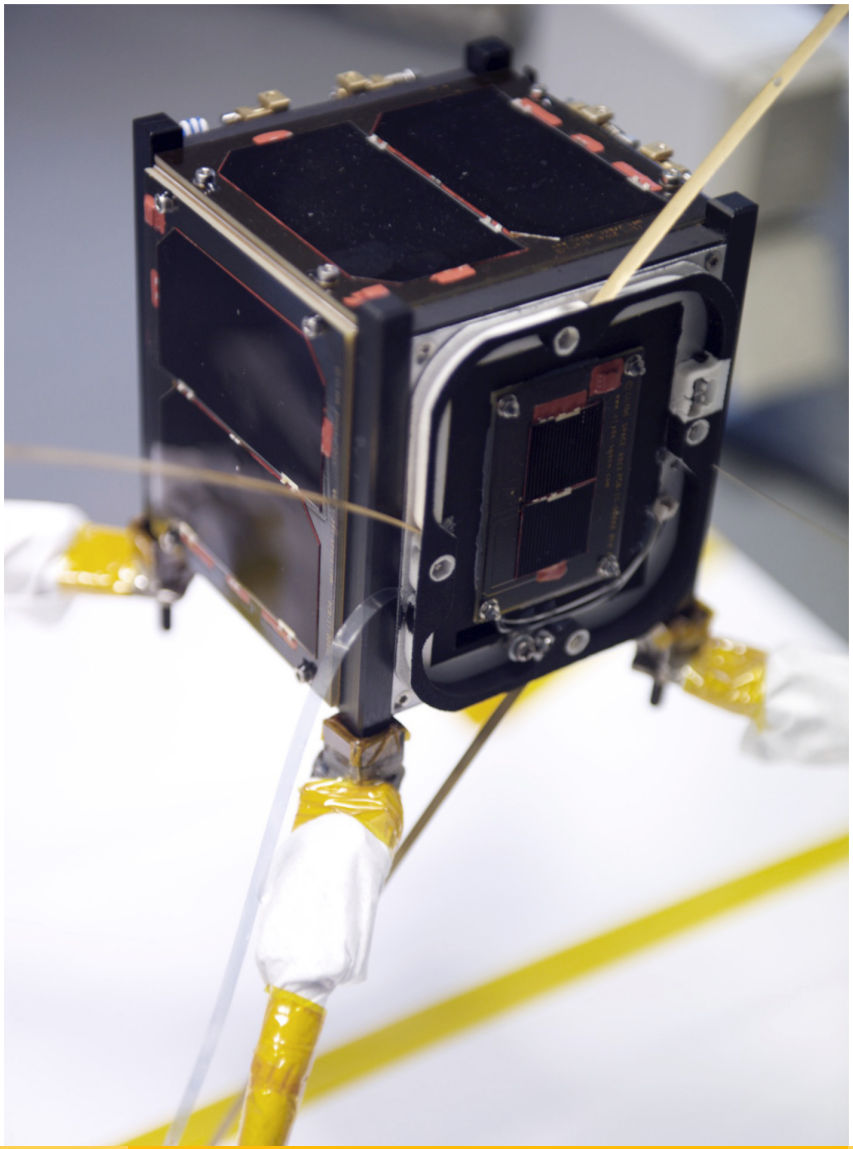
- examen (07.06.) de 3.5 horas (entre 10:00-14:00) que cubre todo el contenido de la asignatura (teoría y prácticas)
- y/o examen (14.07.) de 3.0 horas (entre 09:00-12:00) que cubre todo el contenido de la asignatura (teoría y prácticas)
- un alumno o bien se autodeclara no-asistente o lo muestra por no asistir a por lo menos 80 % de las clases presenciales (como mucho se puede faltar a 9 horas de clase)
- control en clase y/o participación en *quizzes* y asistencia a clase

algo sobre mi

- página web de docencia
  http://formella.webs.uvigo.es
- página web de investigación
  http://lia.esei.uvigo.es

al espacio...

participación en 4 satélites:



XaTcobeo, 14/02/2012

HumSAT-D, 21/11/2013

Serpens-B, 20/08–17/09/2015

Lume-I, 27/12/2018

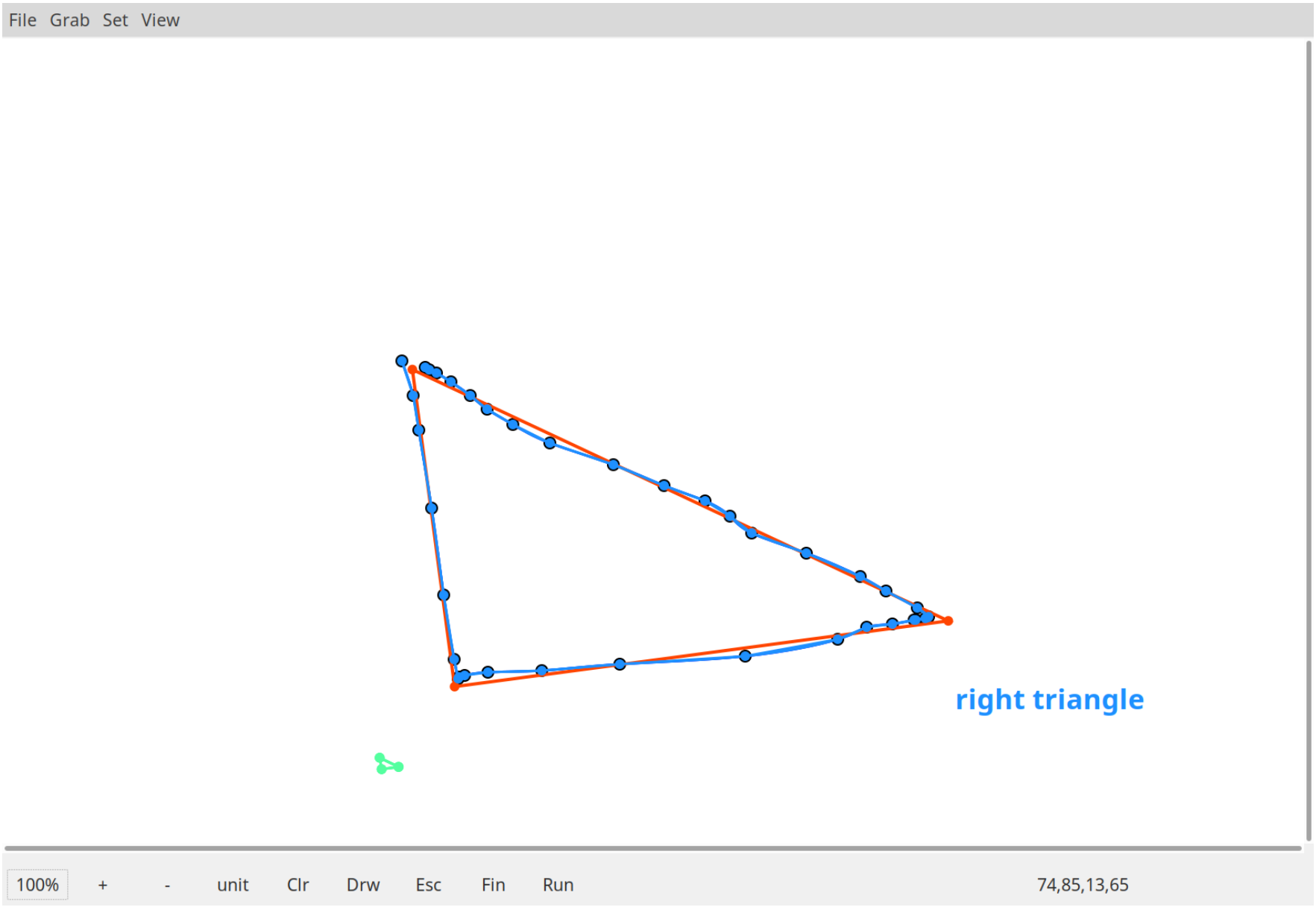
software en mi ordenador (*lines of code*)

- unos 1.100.000 líneas
- unos 36.500 líneas por año
- unos 160 líneas por día
- 22 metros impresos en papel

reconocimiento de formas

- investigación y desarrollo en el ámbito de reconocimiento de formas
  - reconocer formas
  - aprender formas
- uso en aplicaciones de
  - educación infantil,
  - educación matemática,
  - interfaces amigables
  - interfaces para motores de búsqueda
- miramos un poco el prototipo Shapewiz...

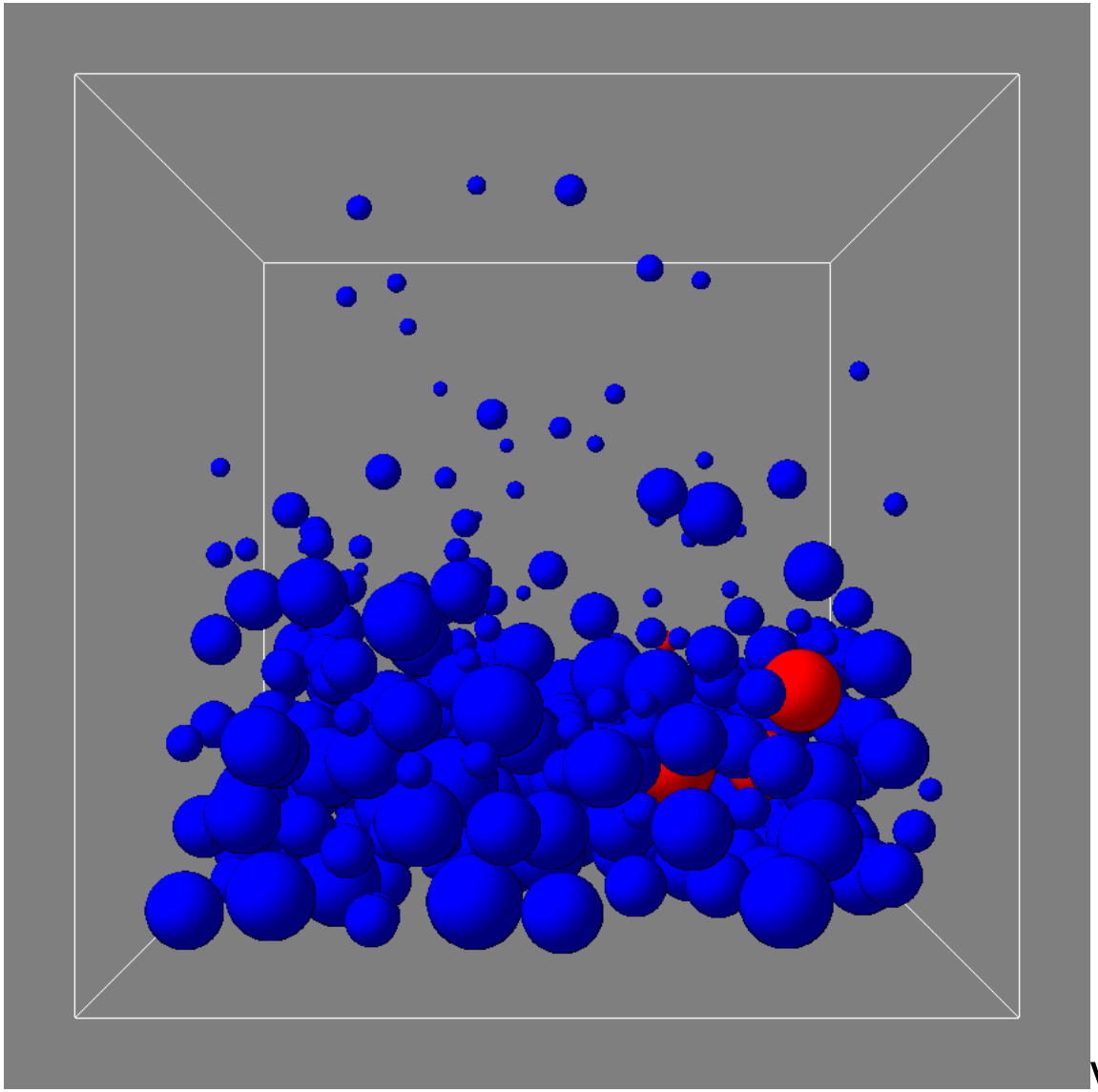
Shapewiz


File Grab Set View
right triangle
100% + - unit Clr Drw Esc Fin Run
74,85,13,65

simulación de partículas

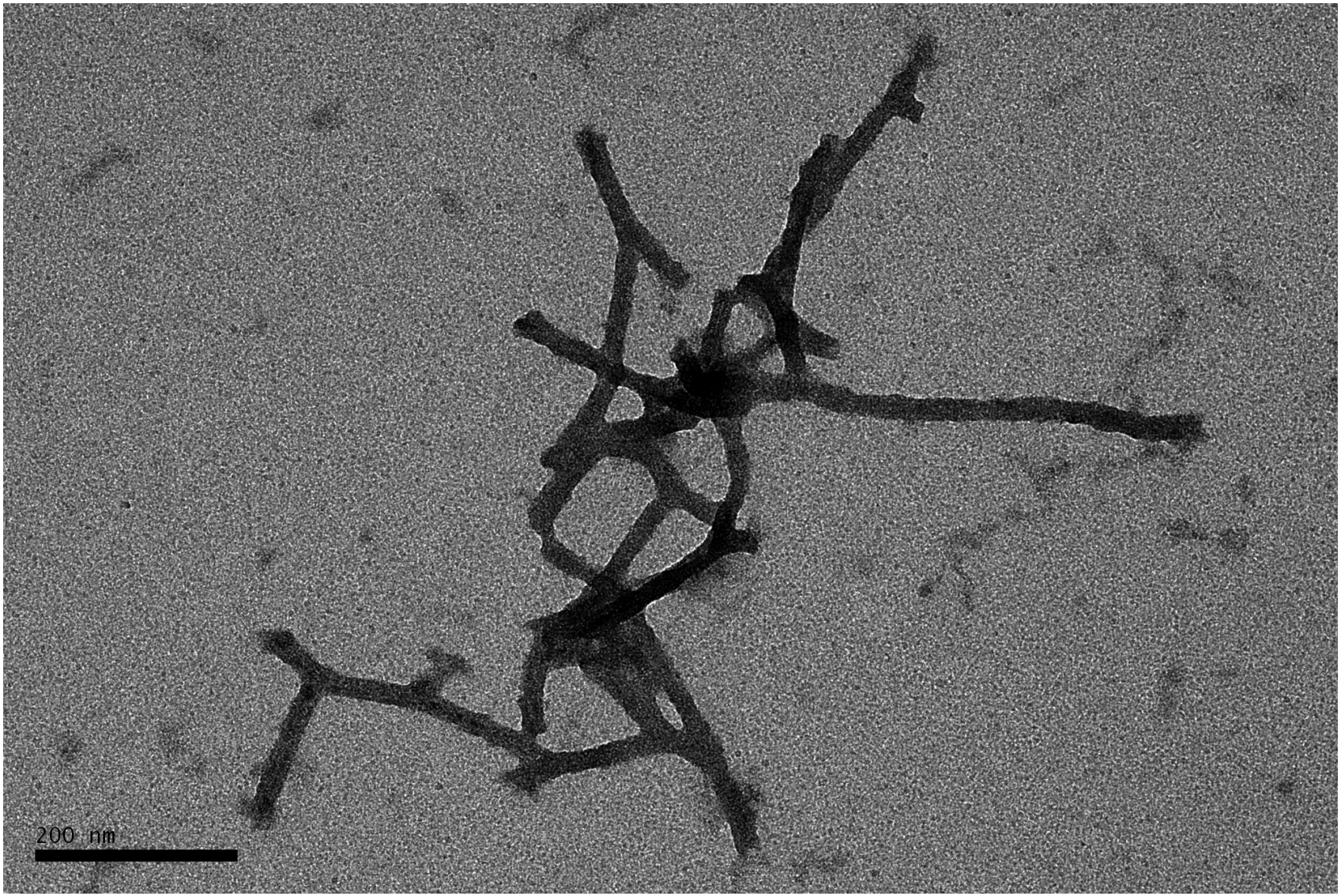
- descripción de un modelado de partículas
  (tamaño, propiedades pre/pos-colisión, otros objetos, etc.)
- simulación según modelado física
  (p.ej. con gravitación)
- simulación basado en eventos discretos
- o simulación basado en monte carlo
- estudio de efectos en materiales granulares y gases

pemd

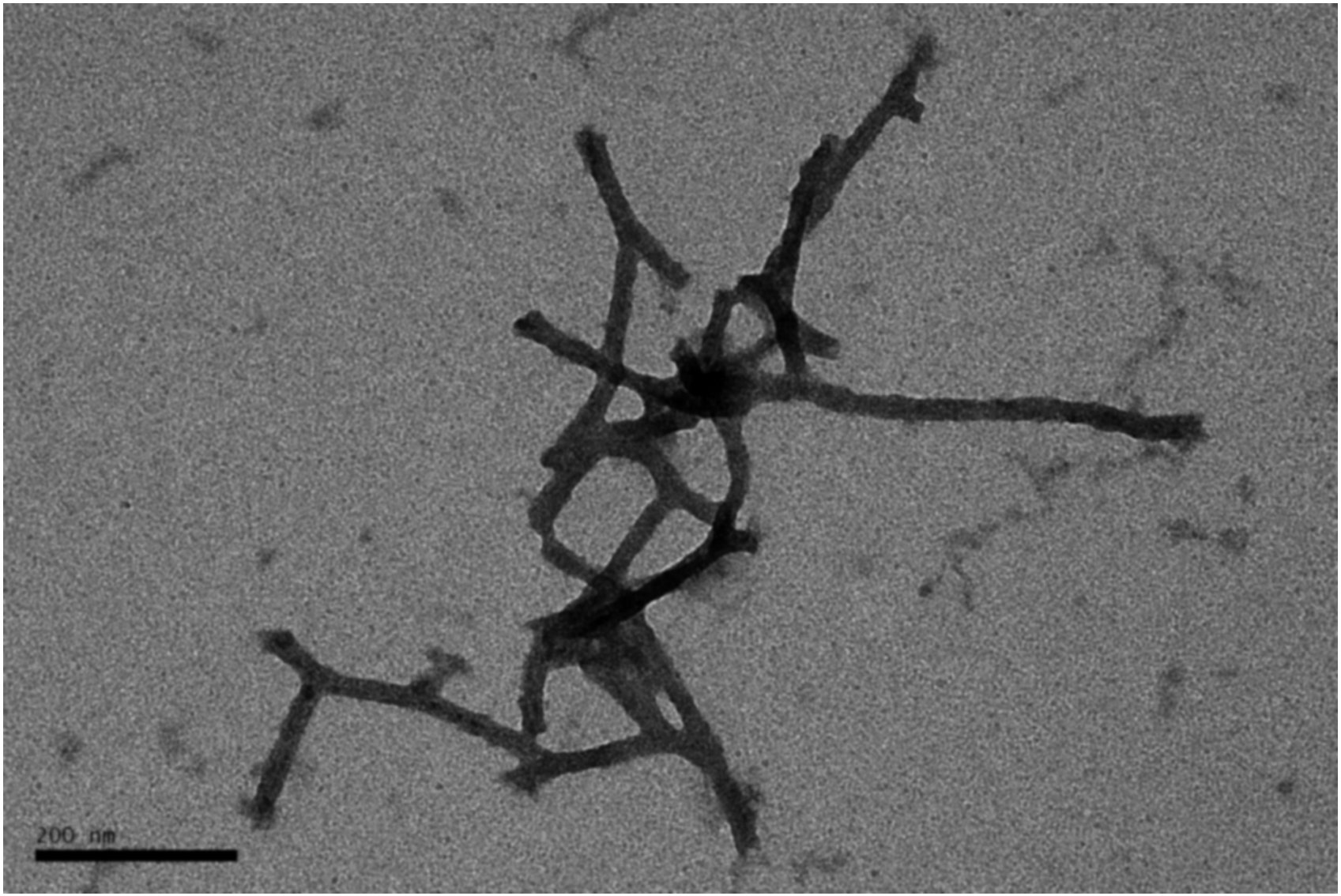


visualización

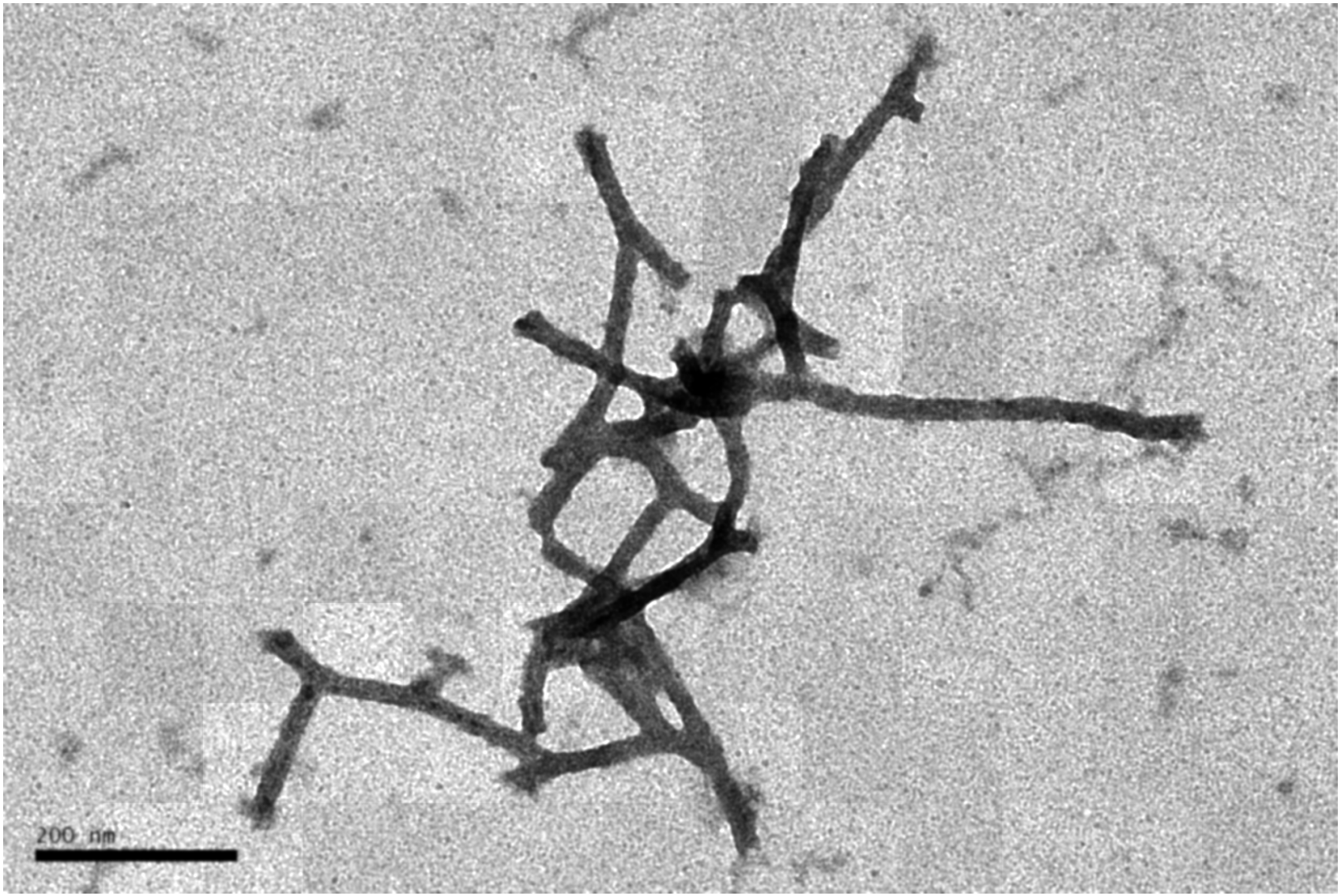
procesamiento de imágenes
bio-nanotubos (original)



procesamiento de imágenes
bio-nanotubos (menos ruido)



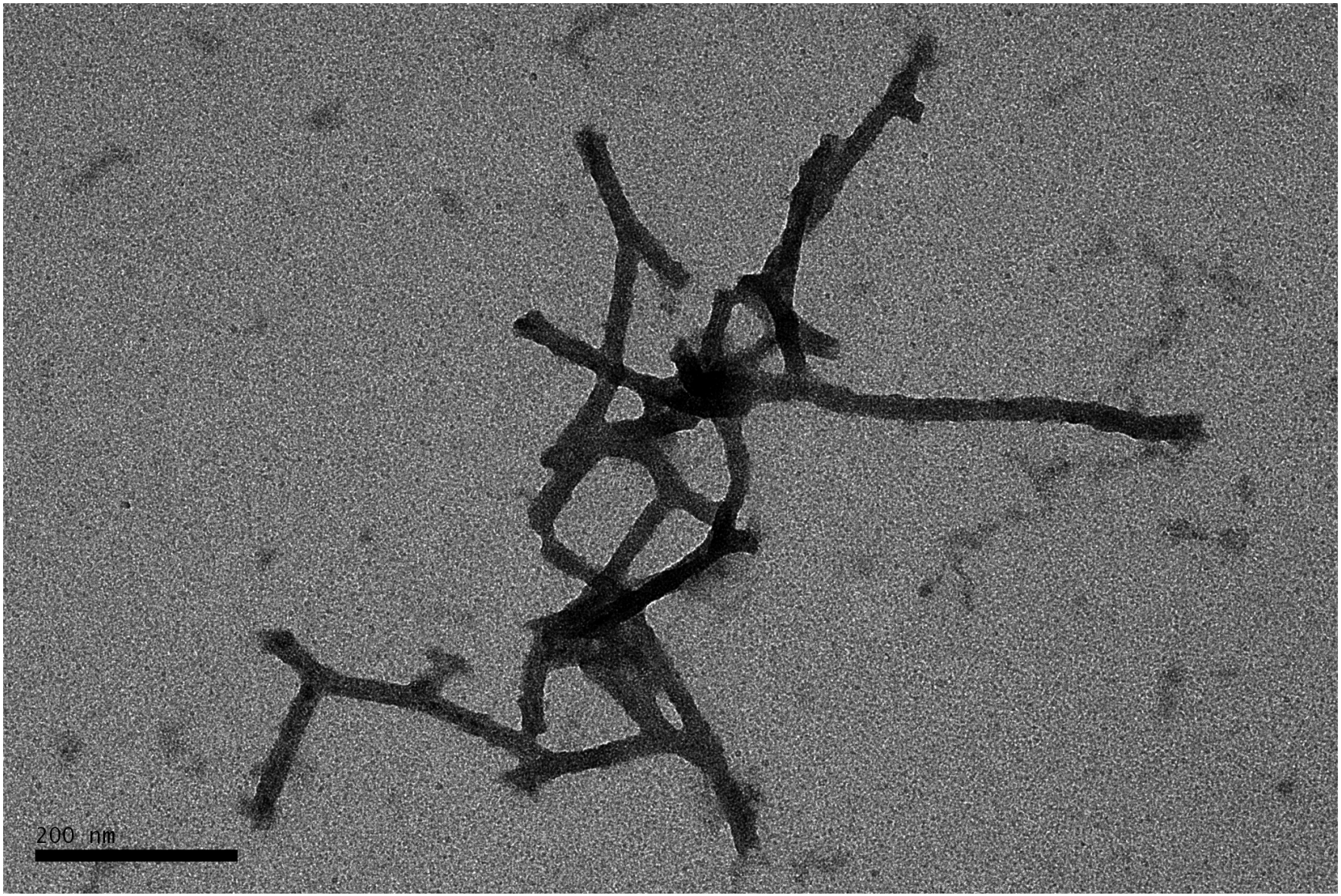
procesamiento de imágenes
bio-nanotubos (más contraste)



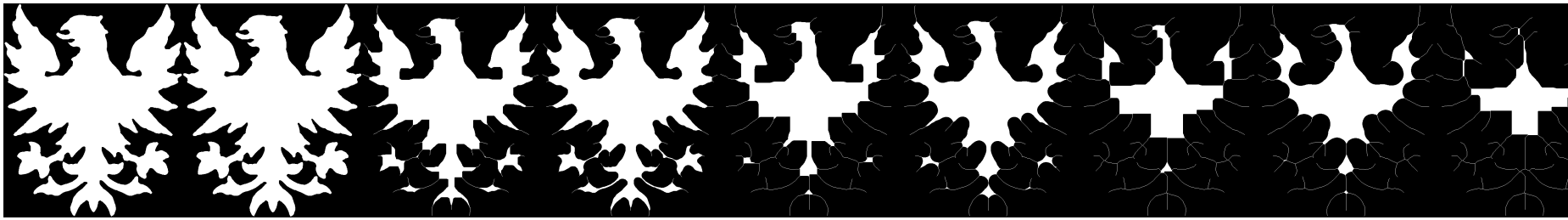
procesamiento de imágenes
bio-nanotubos (binarizado)

procesamiento de imágenes
bio-nanotubos (esqueleto)

procesamiento de imágenes
bio-nanotubos (original)



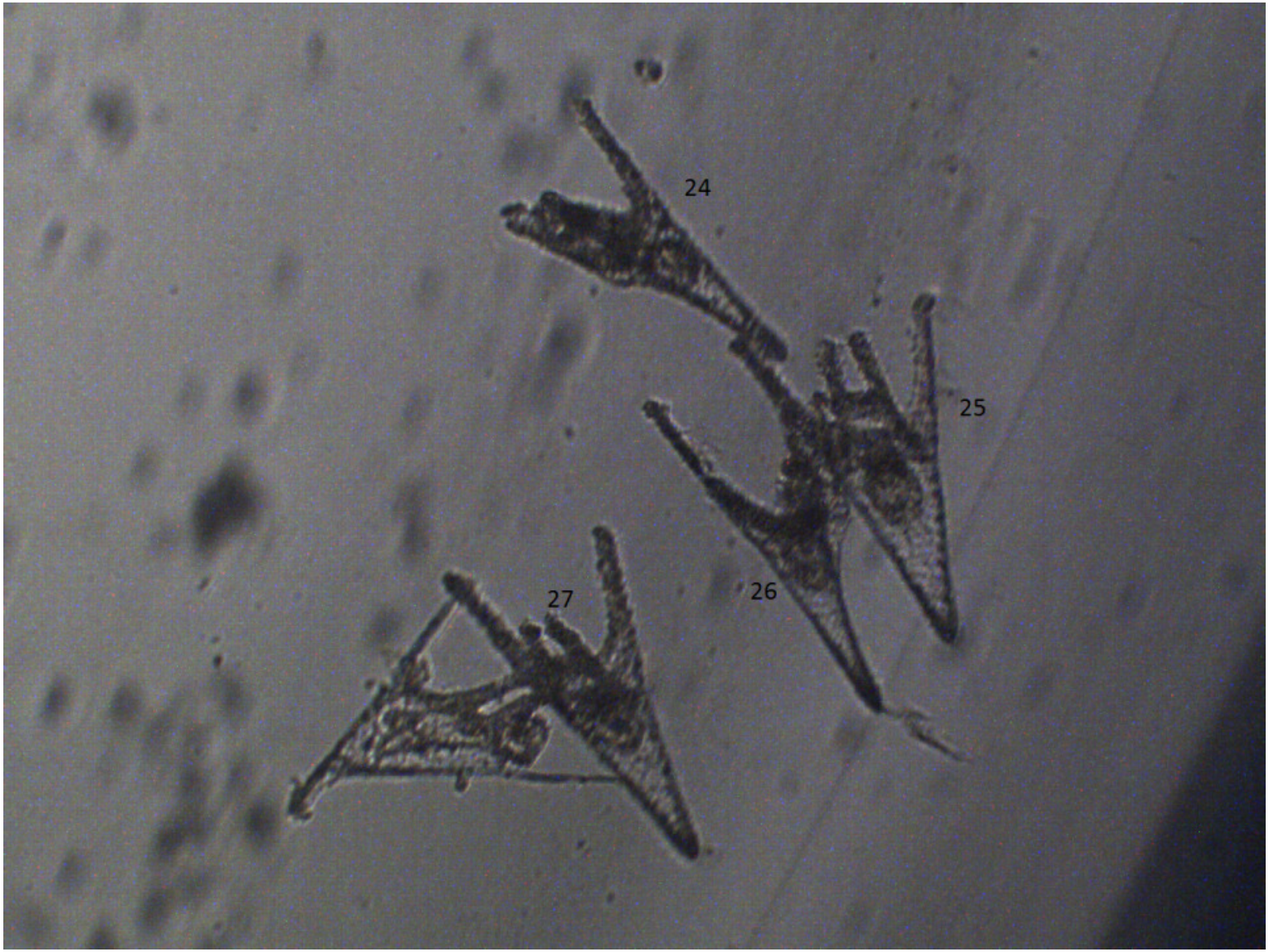
resultado algoritmo concurrente y eficiente!



animación del algoritmo de adelgazamiento en la página web...

Funciona ahora también en GPU.

procesamiento de imágenes (larvas de erizos de mar)



posible colaboración, ejemplos durante el curso

- Quien quiere colaborar, por ejemplo en su TFG, puede ponerse en contacto conmigo.

documento de transparencias

- Este documento crecerá durante el curso,
  ojo, no necesariamente solamente al final.
- Habrá más documentos (páginas web, manuales, etc.)
  con que trabajar durante el curso.
- Los ejemplos de programas y algoritmos serán en inglés.
- Las transparencias no están (posiblemente/probablemente)
  ni correctos ni completos.
- Las transparencias no son suficientes (incluso *chapadas*) para
  aprobar la asignatura.

introducción

Existen diversas definiciones de los términos en la literatura:

- programación concurrente
- programación paralela
- programación distribuida

definición

Una posible distinción (según mi opinión) es:

- la *programación concurrente* se dedica a desarrollar y aplicar conceptos para el uso de recursos en paralelo (desde el punto de vista de varios actores)
- la *programación en paralelo* se dedica a solucionar y analizar problemas bajo el concepto del uso de recursos en paralelo (desde el punto de vista de un sólo actor)

definición

Otra posibilidad de separar los términos es:

- un programa concurrente define las acciones que se pueden ejecutar simultáneamente, es decir, están en progreso
- un programa paralelo es un programa concurrente diseñado de ser ejecutado en hardware paralelo
- un programa distribuido es un programa paralelo diseñado de ser ejecutado en hardware distribuido, es decir, donde varios procesadores no tengan memoria compartida, tienen que intercambiar la información mediante transmisión de mensajes/datos.

intuición

Intuitivamente, todos tenemos una idea básica de lo que significa el concepto de concurrencia.

lo más simple

```
#include <cstdlib>
#include <cmath>
#include <iostream>

int main(
  int argc,
  char* argv[]
) {
  if(argc<2) {
    std::cout<<"# need argument\n";
    std::cout<<"# no data generated\n";
    return 1;
  }
  const double dmax(strtod(argv[1],0));
  for(double d(0.0); d<dmax; d+=0.1)
    std::cout<<d<<" "<<1.0-exp(-d)<<"\n";
  return 0;
}
```

herramientas de producción

- compilación para depuración:

```
g++ data.cpp -o data
```

- ejecución del programa con salida a fichero:

```
./data 5 > D.gdt
```

- visualización de resultados con gnuplot:

```
plot "D.gdt" w lp lw 3
```

(http://www.gnuplot.info, http://www.gnuplotting.org/ o cualquier otro programa que podéis manejar)

- todos estos comandos agrupamos en scripts correspondientes
  - `compile.sh` para compilar
  - `run.sh` para ejecutar
  - `D.gpl` para visualizar

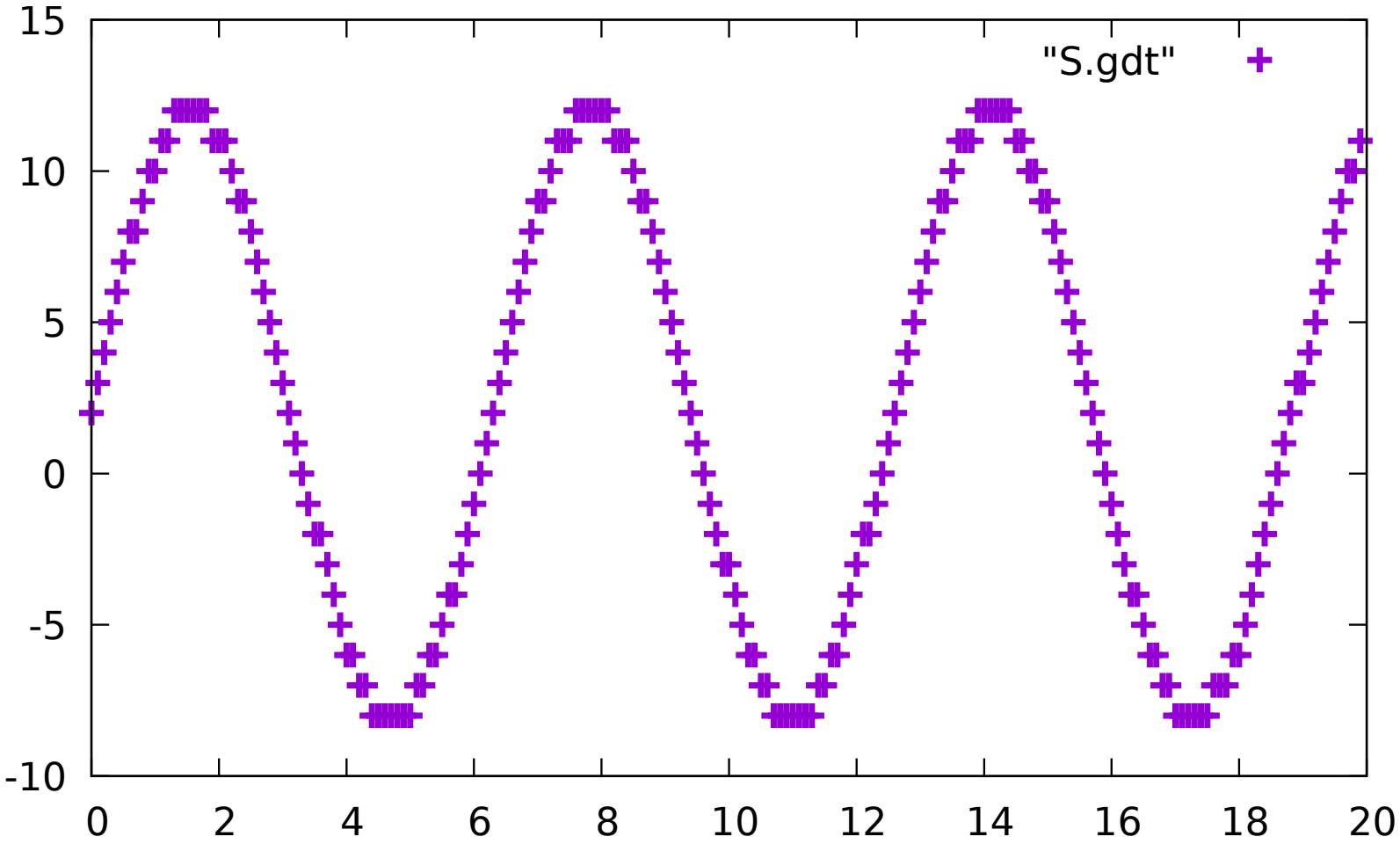
análisis de datos

- miramos otros datos

```
0 2
0.1 3
0.2 4
0.3 5
0.4 6
0.5 7
0.6 8
0.7 8
0.8 9
0.9 10
```

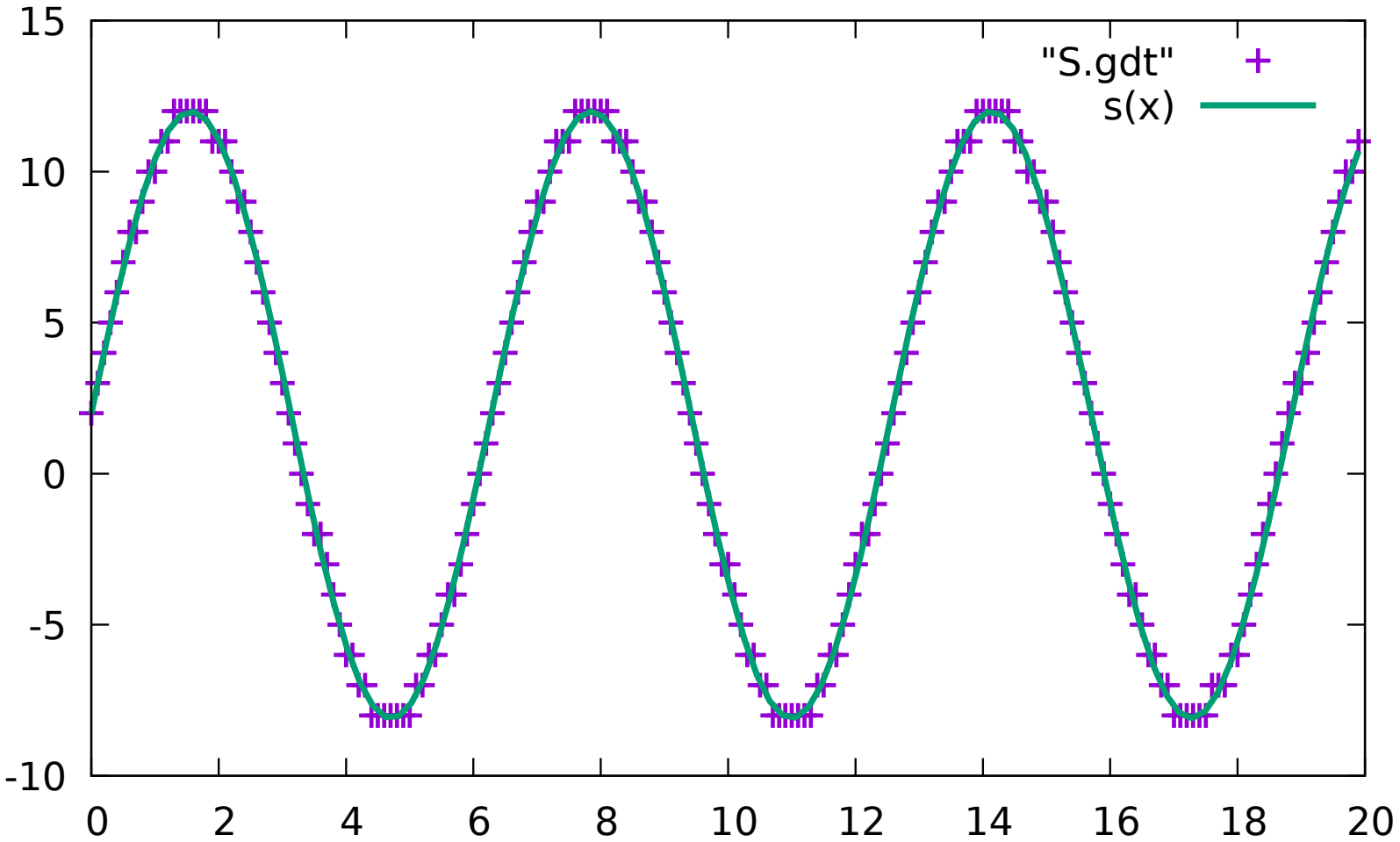
- y aproximamos los datos con una función adecuada

visualización de los datos



aproximación con función *adecuada*

$$s(x) = a_0 \cdot \sin(x - a_1) + a_2$$



aproximación con tal función en gnuplot

```
# start values for the fitting of the underlying function
a_0=10.0
a_1=1.0
a_2=1.0

# our parameterized sinoidal function
s(x) = a_0*sin(x-a_1)+a_2

# fit data to the curve
fit [0:20] s(x) "S.gdt" u 1:2 via a_0,a_1,a_2

# plotting the data with fit
plot "S.gdt" w lp lw 2, s(x) lw 3
```

info del algoritmo Marquardt-Levenberg de gnuplot

```
iter      chisq       delta/lim  lambda   a_0           a_1           a_2
   0 1.0112498375e+04   0.00e+00  5.80e+00    1.000000e+01   1.000000e+00   1.000000e+00
   1 2.3022645454e+03  -3.39e+05  5.80e-01    5.400103e+00   1.544881e-01   1.813287e+00
   2 1.9863785699e+02  -1.06e+06  5.80e-02    9.917101e+00  -1.381719e-01   1.949054e+00
   3 1.6103874361e+01  -1.13e+06  5.80e-03    9.951574e+00  -2.263053e-03   1.949272e+00
   4 1.5268905754e+01  -5.47e+03  5.80e-04    1.004252e+01  -3.565976e-03   1.949272e+00
   5 1.5268904319e+01  -9.40e-03  5.80e-05    1.004252e+01  -3.554176e-03   1.949272e+00
iter      chisq       delta/lim  lambda   a_0           a_1           a_2

After 5 iterations the fit converged.
final sum of squares of residuals : 15.2689
rel. change during last iteration : -9.39583e-08

degrees of freedom    (FIT_NDF)                        : 197
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.278401
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.0775071

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a_0             = 10.0425          +/- 0.0282       (0.2808%)
a_1             = -0.00355418      +/- 0.002749     (77.36%)
a_2             = 1.94927          +/- 0.01974      (1.013%)

correlation matrix of the fit parameters:
                a_0    a_1    a_2
a_0             1.000
a_1             0.037  1.000
a_2            -0.037  0.064  1.000
```

resumen:

- podemos generar ficheros con datos
- podemos visualizar los datos con un programa (p.ej. gnuplot)
- podemos aproximar los datos con funciones adecuadas
- podemos interpretar los datos con (cierto) sentido
- podemos trabajar con argumentos por línea de comando
- usamos *scripts* si es posible

ordenamos

queremos ordenar unos simples números dentro de un rango

- ¿Cómo procedemos?
- ¿Cuáles son los aspectos/problemas a tratar?
- ¿Cómo lo trasladamos a un entorno programable?

desafios

¿Cuáles son los desafios?

- selección del algoritmo
- ¡menos mal que en tercero ya sabemos que es un algoritmo!
- e implementarlo...

pero ahora con un algoritmo que actua de forma concurrente además tenemos que pensar en:

- división del trabajo
- distribución de los datos
- sincronización necesaria
- comunicación de los datos entre participantes
- comunicación de los resultados

desafios

Y también con:

- medición de características
- depuración del programa
- (fiabilidad de los componentes)
- (fiabilidad de la comunicación)
- (detección de la terminación)

Programar aplicaciones concurrentes puede ser divertido y frustante a la vez :-)

medición de tiempo: la carrera de *San Martiño*

explicado en pizarra en clase

medición de tiempo de ejecución (ejemplo en Java)

```
class Timer {
  private long[] startTime;
  private long[] stopTime;

  Timer(int n) {
    startTime=new long[n];
    stopTime=new long[n];
  }
  public void Start(int i) {
    startTime[i]=System.nanoTime();
  }
  public void Stop(int i) {
    stopTime[i]=System.nanoTime();
  }
  public double Elapsed() {
    long minTime=startTime[0];
    for(int i=1; i<startTime.length; ++i)
      if(startTime[i] < minTime) minTime=startTime[i];
    long maxTime=stopTime[0];
    for(int i=1; i<stopTime.length; ++i)
      if(stopTime[i] > maxTime) maxTime=stopTime[i];
    return (maxTime-minTime)/1000000.0;
  }
}
```

resumen del uso de OpenMP

- OpenMP es una API abierta para la programación paralela bastante cómoda para usar (`www.openmp.org`) en C++ y Fortran.
- La versión actual es la 5.2 (noviembre 2021), usamos la que viene con el compilador (no hacemos nada ni sofisticado ni específico).
- OpenMP usa en C++ la técnica de las `#pragma` para introducir instrucciones OpenMP (que se ignoran cuando compilamos sin OpenMP).
- Se incluye el fichero `omp.h` y se compila (en g++) con la opción `-fopenmp`.

Bucle de cálculo (compara con prácticas)

```
#pragma omp parallel for
    for(unsigned i=0; i<nIter; ++i) {
      double d(std::tan(std::atan(
        std::tan(std::atan(
          std::tan(std::atan(
            std::tan(std::atan(
              std::tan(std::atan(123456789.123456789))
            ))
          ))
        ))
      ))));
      d=std::cbrt(d);
    }
```

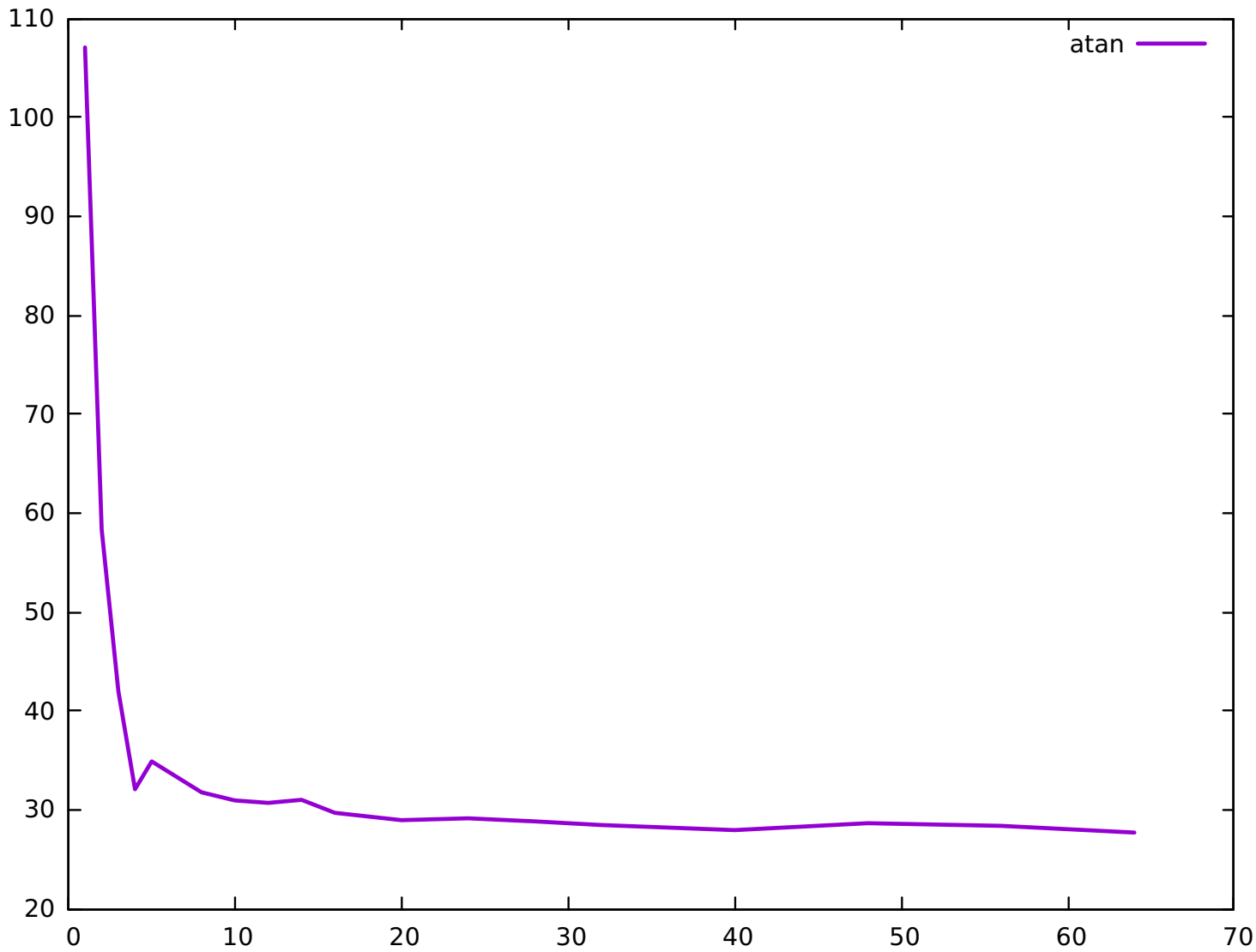
Optimizaciones del compilador

- si compilamos el código con optimización activado
- medimos un tiempo de ejecución casi 0 en caso secuencial
- dado que el compilador es capaz de detectar que no se realiza ninguna operación con la variable `d` fuera del bucle
- y elimina básicamente todo el bucle
  (*dead code elimination*, eliminación de código innecesario)

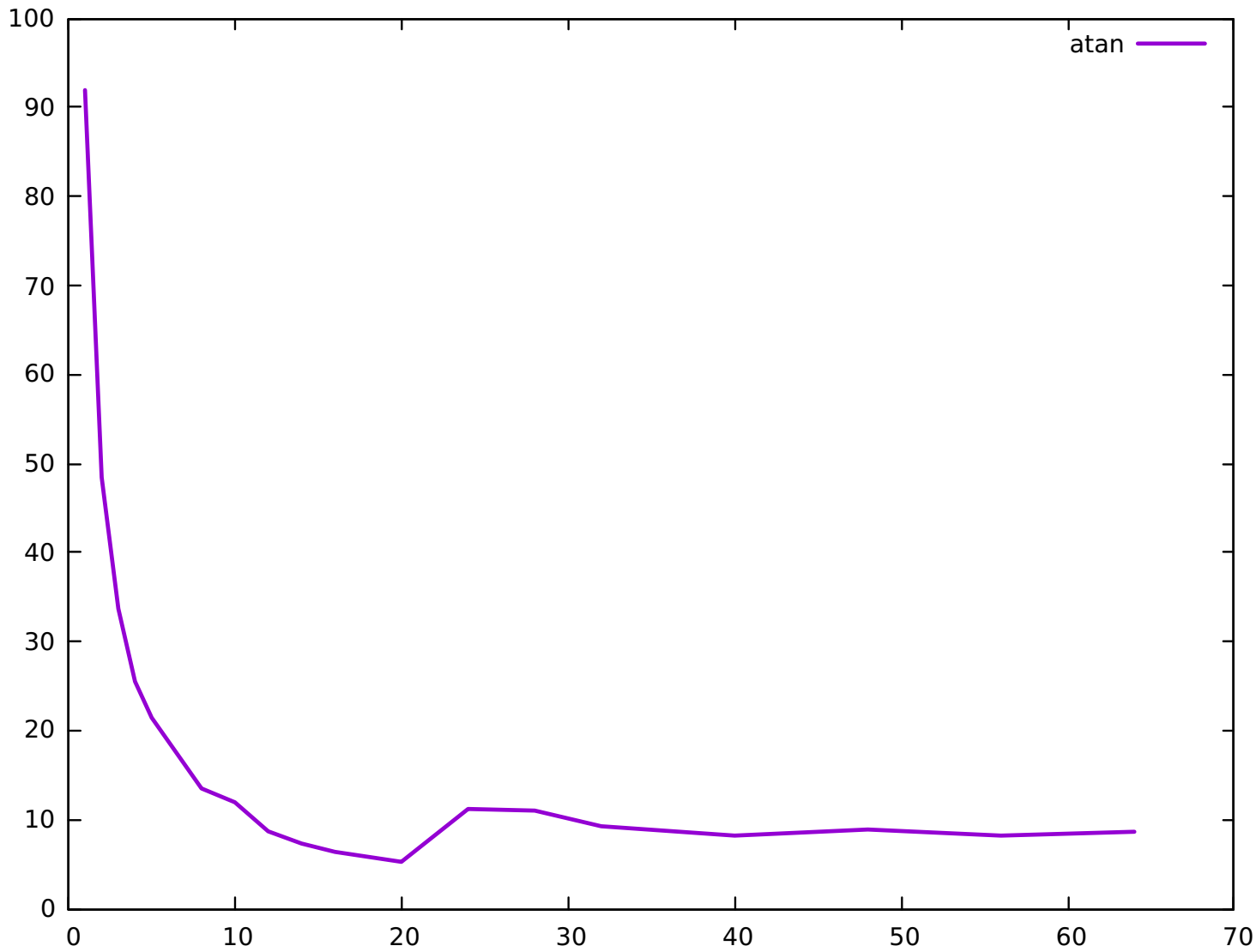
Bucle de cálculo con variable NO eliminable

```
#pragma omp parallel for \
    reduction(+:result)
    for(unsigned i=0; i<nIter; ++i) {
      double d(std::tan(std::atan(
        std::tan(std::atan(
          std::tan(std::atan(
            std::tan(std::atan(
              std::tan(std::atan(123456789.123456789))
            ))
          ))
        ))
      ))));
      result+=std::cbrt(d);
    }
```

tiempo de ejecución observado en i7



tiempo de ejecución observado en i9



ordenación

- La ordenación de datos es una tarea muy usada en muchas aplicaciones.
- ¿Cómo se ordena de forma concurrente (y eficiente)?
- En general no es nada trivial desarrollar algoritmos de ordenación paralelos, eficientes, escalables, etc.
- Miramos un caso especial (*counting sort*):
  ordenación de muchos datos (aquí enteros positivos) en cierto rango no demasiado grande, pero más grande que el número de procesos disponibles.

pizarra...

explicado el counting sort secuencial en pizarra en clase

pizarra...

explicado el counting sort paralelo en pizarra en clase

ordenación

- parallel counting sort (ordenación contando en paralelo)
  (mira: counting sort, radix sort, bucket sort)
- implementamos en C++ con OpenMP
- medimos el tiempo de ejecución en diferentes sistemas
- realizamos primero todo lo necesario alrededor,
  luego el propio algoritmo

comprobación de la ordenación

```
  // Checks whether data is sorted.
  // Note the standard library provides such a function,
  // but we do our own...
bool IsSorted(
  const std::vector<int>& data
) {
  const unsigned data_size(data.size());
  bool sorted(true);
  for(unsigned i=1; i<data_size; ++i) {
    if(data[i-1]>data[i]) sorted=false;
  }
  return sorted;
}
```

medición de tiempo de ejecución (ejemplo en C++)

```
  // Shortcut for complex timer type.
typedef std::chrono::high_resolution_clock::time_point
  TimePoint;

  // Starts our timer clock.
TimePoint StartClock(
) {
  return std::chrono::high_resolution_clock::now();
}

  // Stops our timer clock. Returns the elapsed time in [ms].
double StopClock(
  TimePoint start
) {
  const auto duration(
    std::chrono::duration_cast<std::chrono::milliseconds>(
      std::chrono::high_resolution_clock::now()-start
  ));
  return static_cast<double>(duration.count());
}
```

bucle principal de medición

```
  // Measure the runtime of our counting sort algorithm
  // in a measuring loop to smooth measuring fluctuations.
double time_countingsort(0.0);
for(unsigned i(0); i<measuringLoops; ++i) {
  W=V;
  start=StartClock();
  CountingSort(W);
  time_countingsort+=StopClock(start);
  if(!IsSorted(W)) std::cerr<<"Error: not sorted?\n";
}
```

lo que falta...

- paso de parámetros via línea de comando
- ayuda e información de copyright
- inicialización de los datos con valores aleatorios
- visualización de los datos (uso durante depuración)
- miramos estas funciones y,
  un poco más tarde, la implementación del propio algoritmo
  directamente en el fichero de código

las pragmas que usamos

- `#pragma omp parallel`
  comienza una región paralela, es decir, los hilos (el principal incluido) empiezan actuar en paralelo
- `#pragma omp for schedule(static)`
  realiza un bucle `for` que automaticamente se divide en trozos iguales para cada hilo participante
- `#pragma omp single`
  comienza una región secuencial dentro de una paralela, es decir, esta parte ejecuta solamente un hilo
- observa: tenemos sincronización implícita entre bucles, es decir, todos empiezan dentro de la región paralelo los bucles `for` al mismo tiempo (y en la región secuencial los que no actuan tampoco avanzan más allá)

comprobación de la ordenación en paralelo

```
  // Checks whether data is sorted.
  // Note the standard library provides such a function,
  // but we do our own...
bool IsSorted(
  const std::vector<int>& data
) {
  const unsigned data_size(data.size());
  bool sorted(true);
#pragma omp parallel \
  default(none) \
  shared(data) \
  shared(sorted)
  {
#pragma omp for schedule(static)
    for(unsigned i=1; i<data_size; ++i) {
      if(data[i-1]>data[i]) sorted=false;
    }
  }
  return sorted;
}
```

miramos el código en C++

mira el fichero `Sort.cpp` disponible

un breve análisis del tiempo de ejecución

- acordamos:
  n número de datos, v rango de valores, p número de procesos
- el algoritmo usa 8 bucles:
  1. copia y min/max-búsqueda (en paralelo): $O(n/p)$
  2. inicialización de tamaños (en paralelo): $O(v/p)$
  3. contar ocurrencias (en paralelo): $O(n/p)$
  4. cálculo de tamaños (en paralelo): $O(v/p*p) = O(v)$
  5. prefix-sum para tamaños (secuencial): $O(v)$
  6. inicialización de índices (en paralelo): $O(v/p)$
  7. prefix-sum para índices (en paralelo): $O(v/p*p) = O(v)$
  8. copia de datos (en paralelo): $O(n/p)$
- entonces tiempo de ejecución: $O(n/p+v)$
- observa: hay trabajo por hacer en la versión paralela (pasos 4,5,6, y 7) que no es necesario en la versión secuencial

un breve análisis del uso de memoria

- estructuras de datos principales:
  1. array de copia de datos: $O(n)$
  2. array para tamaños: $O(v)$
  3. arrays para contadores/índices: $O(v * p)$
- entonces uso de memoria: $O(n + v * p)$
- observa: la versión paralela necesita más memoria que la versión secuencial (aunque poco más en este caso)

retornamos a los desafios

- selección del algoritmo
- división del trabajo
- distribución de los datos
- sincronización necesaria
- comunicación de los datos entre participantes
- comunicación de los resultados
- medición de características

¿cómo los hemos afrontado?

selección del algoritmo

- hemos usado un algoritmo que en principio nos permite un *speedup* lineal en el número de procesos
  (aunque en realidad mediendo no lo hemos visto...)
- el algoritmo es más rápido para nuestro problema que el algoritmo de la biblioteca estándar
- hemos visto que el algoritmo de la biblioteca tiene un buen *speedup* (si nuestro problema fuese la ordenación general)

Hay que seleccionar el algoritmo adecuado para la situación. Si se usa una biblioteca (trabajos de otros), hay que mirar si usan las características concurrentes del sistema, sino se pierde mucha eficiencia.

división del trabajo

- hemos dividido, con la ayuda del compilador con OpenMP, todos los bucles largos

  ```
  #pragma omp for schedule(static)
  ```

  en trozos de igual tamaño para todos los procesos participantes
- algún código hemos ejecutado con un único proceso

  ```
  #pragma omp single
  ```

distribución de los datos

- estamos en arquitecturas con memoria común
- hemos declarado por defecto ninguna variable compartida y los que nos interesan como compartidas
- constantes están implicitamente compartidas (en OpenMP)
- variables declaradas en el bloque paralelo son automáticamente local al proceso (en OpenMP)
- hemos diseñado el algoritmo de tal manera que los procesos escriben en memoria exclusiva, es decir, nunca escriben en el mismo sitio a la vez

sincronización necesaria

- OpenMP facilita bastante la sincronización simple
- al comienzo del bloque paralelo se crean los procesos cuyo número se ha fijado previamente en algún momento
- al comienzo de un bucle `for` se lanza todos los procesos con la división del trabajo establecida
- al final de un bucle `for` se sincroniza automáticamente con una espera a que todos los participantes hayan terminado
  (nota: eso es cambiable!)
- algunas partes del algoritmo hemos ejecutado a propósito con un único proceso
- al final del bloque paralelo se terminan automáticamente todos los procesos

comunicación de los datos

- no sabemos todavía como el compilador (con OpenMP) consigue que todos los procesos saben que trozos tienen que calcular, por ejemplo, hallar los valores locales de `k` para cada proceso,
- esta comunicación implícita es una de las grandes ventajas de usar OpenMP
- no sabemos todavía como funciona la operación de *reducción* donde todos los procesos participantes consiguen el resultado total de sus valores locales en una variable ompartida
- esta técnica implícita es una de las grandes ventajas de usar OpenMP
- hemos usado una variable booleana compartida `sorted` con modificación posiblemente concurrente
- en este caso no se manifiestan problemas de escrituras concurrentes porque si los procesos escriben algo, entonces escriben lo mismo.

comunicación de los resultados

- el resultado de la ordenación se almacena en una variable compartida
- los paramétros de entrada gestiona un solo proceso y se comunica mediante variables compartidas con los demás
- el resultado del programa (podemos considerar, por ejemplo, la medición del tiempo), escribe un único proceso

hola mundo

El famoso *hola mundo* se programa en Java así:

```
class Hello {
  public static void main(String[] args) {
    System.out.println(
      "Hello world, this is "+args[0]
    );
  }
}
```

El programa principal se llama `main()` y tiene que ser declarado público y estático. No devuelve ningún valor (por eso se declara como `void`).
Los parámetros de la línea de comando se pasan como un vector de cadenas de letras (`String`).

```
javac Hello.java
java Hello primero segundo tercero
```

¿Qué se comenta?

- Se usa doxygen (o javadoc, u otro bueno) para generar automáticamente la documentación. Todos tienen unos comandos para aumentar la documentación.
- Existen varias posibilidades de escribir comentarios:

`//`	comentario de línea
`/// ...`	comentario de documentación
`/* ... */`	comentario de bloque
`/** ... */`	comentario de documentación

- Se documenta sobre todo lo que no es obvio, las interfaces (en el sentido amplio de la palabra), y los casos límite.
- Es decir: Los comentarios son las respuestas a preguntas del *¿Cómo?* y del *¿Por qué?*.

Hilos

- Se usan los hilos para ejecutar varias secuencias de instrucciones de modo cuasi–paralelo.
- Es decir, la ejecución parece ser en paralelo, pero si es realmente paralelo (y no secuencial) depende del hardware (más preciso, del número de CPUs involucrados) que se está usando en cada momento.

creación de un hilo (para empezar)

- Se crea un hilo con
  `Thread worker = new Thread()`
- Se inicializa el hilo y se define su comportamiento.
- Normalmente se usa una clase derivada o la interfaz `Runnable`.
- Se lanza el hilo con
  `worker.start()`
- Pero en esta versión simple no hace nada. Hace falta sobreescribir el método `run()` especificando algún código útil.
- ¿Cuándo arranca de verdad este nuevo hilo?

la interfaz `Runnable`

- A veces no es conveniente extender la clase `Thread` porque se pierde la posibilidad de extender otro objeto.
- Es una de las razones por que existe la interfaz `Runnable` que declara nada más que el método `public void run()` y que se puede usar fácilmente para crear hilos trabajadores.

resumen Java respecto a concurrencia

algunos ya vimos/usamos otros vendrán...

- `Thread, Runnable`
- `join, isAlive, activeCount`
- `try-catch-finally`
- `volatile`
- `synchronized`
- `wait, notify, notifyAll`
- `finalize`
- `transient`
- `java.util.concurrent`
- `java.util.concurrent.atomic`
- etc.

try'n'catch

- Para facilitar la programación de casos excepcionales Java usa el concepto de lanzar excepciones.
- Una excepción es una clase predefinida y se accede con la sentencia

```
try { ... }
catch(SomeExceptionObject e) { ... }
catch(AnotherExceptionObject e) { ... }
finally { ... }
```

orden de ejecución I

- El bloque `try` contiene el código normal por ejecutar.
- Un bloque `catch(ExceptionObject)` contiene el código excepcional por ejecutar en caso de que durante la ejecución del código normal (que contiene el bloque `try`) se produzca la excepción del tipo adecuado.
- Pueden existir más de un (o ningún) bloque `catch` para reaccionar directamente a más de un (ningún) tipo de excepción.
- Hay que tener cuidado en ordenar las excepciones correctamente, es decir, las más específicas antes de las más generales.

orden de ejecución II

- El bloque `finally` se ejecuta siempre una vez terminado o bien el bloque `try` o bien un bloque `catch` o bien una excepción no tratada o bien antes de seguir un `break`, un `continue` o un `return` hacia fuera de la sentencia `try-catch-finally`.
- Por eso es un buen punto donde ejecutar código en aplicaciones concurrentes que deben realizar cierto *trabajo* final, incluso en casos excepcionales.

construcción de clases de excepción

Normalmente se extiende la clase `Exception` para implementar clases propias de excepciones, aún también se puede derivar directamente de la clase `Throwable` que es la superclase (interfaz) de `Exception` o de la clase `RuntimeException`.

```
class MyException extends Exception {
  public MyException() { super(); }
  public MyException(String s) { super(s); }
}
```

declaración de excepciones lanzables

- Entonces, una excepción no es nada más que un objeto que se crea en el caso de aparición del caso excepcional.
- La clase principal de una excepción es la interfaz `Throwable` que incluye un `String` para mostrar una línea de error legible.
- Para que un método pueda lanzar excepciones con las sentencias `try-catch-finally`, es imprecindible declarar las excepciones posibles antes del bloque de código del método con `throws ...`.

```
public void myfunc(...) throws MyException {...}
```

- En C++ es al revés, se declara lo que se puede lanzar como mucho.

propagación de excepciones

- Durante la ejecución de un programa se propagan las excepciones desde su punto de aparición subiendo las invocaciones de los métodos hasta que se haya encontrado un bloque `catch` que se ocupa de tratar la excepción.
- En el caso de que no haya ningún bloque responsable, la excepción será tratada por la máquina virtual con el posible resultado de abortar el programa.

lanzar excepciones

- Se pueden lanzar excepciones directamente con la palabra `throw` y la creación de un nuevo objeto de excepción, por ejemplo:

```
throw new MyException("this is an exception");
```

- También los constructores pueden lanzar excepciones que tienen que ser tratados en los métodos que usan dichos objetos construidos.

excepciones de ejecución

- Además de las excepciones así declaradas existen siempre excepciones que pueden ocurrir en qualquier momento de la ejecución del programa, por ejemplo, `RuntimeException` o `Error` o `IndexOutOfBoundException`.
- La ocurrencia de dichas excepciones refleja normalmente un flujo de control erróneo del programa que se debe corregir antes de distribuir el programa a posibles usuarios.

excepciones de ejecución

- Recomendación: Se usan excepciones solamente para casos excepcionales, es decir, si pasa algo no esperado.
- Excepciones en programas concurrentes se convierten rápidamente en pesadillas.
  Ya que aflorece el problema que hacer si muchos procesos empiezan a lanzar excepciones individualmente...
- Mirad Safe (Disciplined) Exception Handling principle con sus dos posibilidades de acción
  - fallo: re-establecer una invariante necesaria (p.ej.: cerrar conexiones, cerrar ficheros, actualizar vitácora etc.) y seguramente abandonar el programa
  - re-intentar: re-hacer la operación con (quizá) cierta modificación (p.ej.: volver a un menu/bucle principal abandonando un módulo) y seguramente seguir con el programa

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?

Sincronización

- A veces se quiere que un hilo actua sin que otros interfieran en su tarea.
- Es decir, se quiere una ejecución con exclusión mutua del código.
- Dicho concepto se llama también atomicidad de las operaciones,
- o también ejecución segura con hilos (*threadsafe*).
- En Java existen diferentes posibilidades para conseguir exclusión mutua.

Atomicidad en Java

- Solo las asignaciones a variables de tipos simples de 32 bits son atómicas.
- `long` y `double` no son simples en este contexto porque son de 64 bits.
- Hay que declarar esas variables como `volatile` para obtener acceso atómico.

synchronized

- En Java es posible forzar la ejecución del código en un bloque de modo sincronizado, es decir, como mucho un hilo, que tenga acceso compartido a `obj`, puede ejecutar el código dentro de dicho bloque.
  ```
  synchronized(obj) { ... }
  ```
- La expresión entre paréntesis `obj` tiene que evaluar a una referencia a un objeto o a un vector.
- Declarando un método con el modificador `synchronized` garantiza que dicho método se ejecuta por un sólo hilo (y ningún otro método sincronizado del mismo objeto tampoco).
- La máquina virtual instala un cerrojo (mejor dicho, un monitor, ya veremos dicho concepto más adelante) que se cierra de forma atómica antes de entrar en la región crítica y que se abre antes de salir.

Métodos syncronizados

- Declarar un método como
  `synchronized void f(...) { ... }`
  es equivalente a usar un bloque sincronizado en su interior:
  `void f(...) { synchronized(this) { ... } }`
- Los monitores (implementados en la MVJ) permiten que el mismo hilo puede acceder a otros métodos o bloques sincronizados del mismo objeto sin problema.
- Se libera el cerrojo sea el modo que sea que termine el método.
- Los constructores no se pueden declarar `synchronized`.

Enteros con `int` y método sincronizado

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

¡Implementa con esta clase el algoritmo!

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...).

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

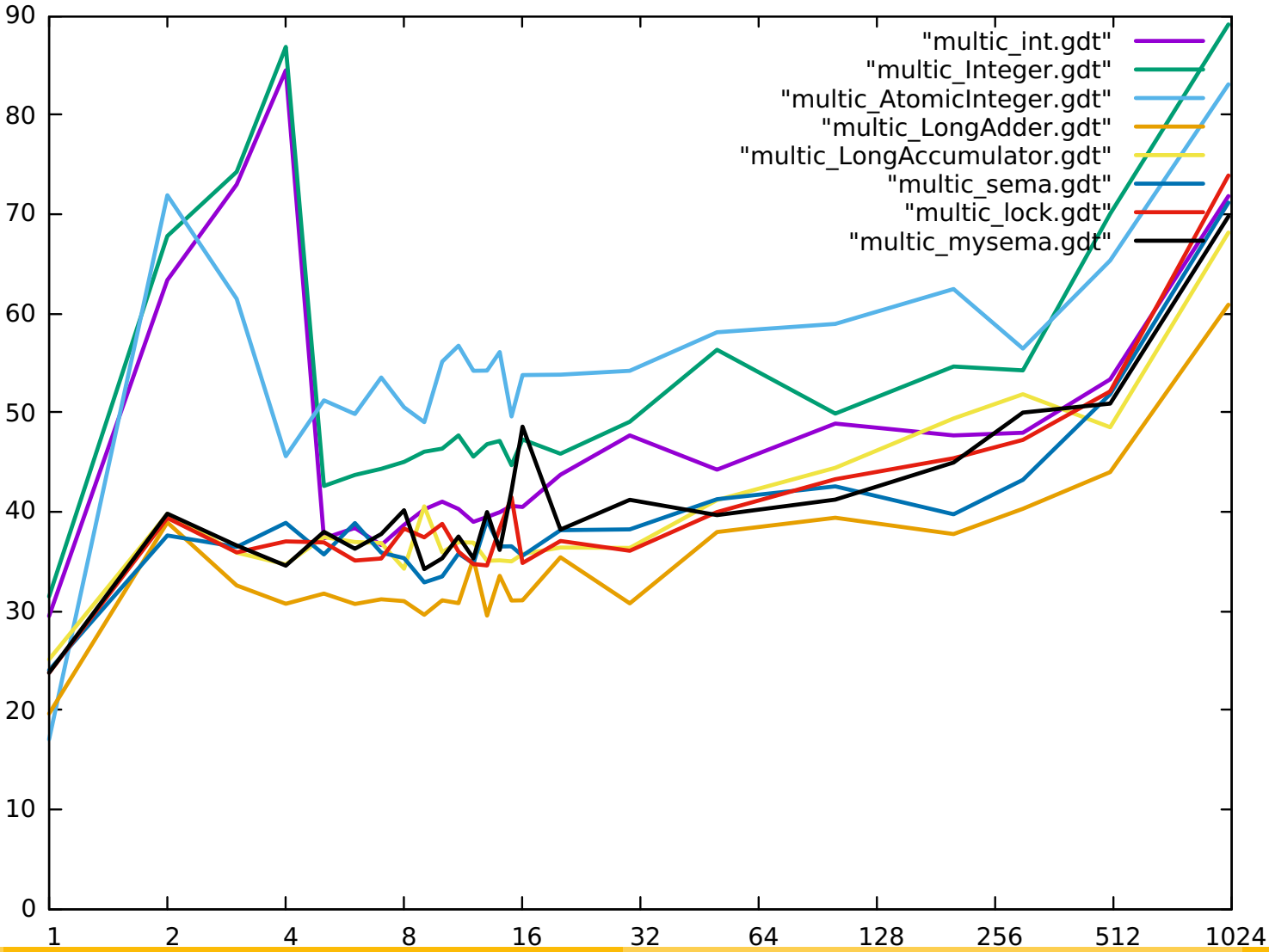
```
import java.util.concurrent.atomic.*;

// 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

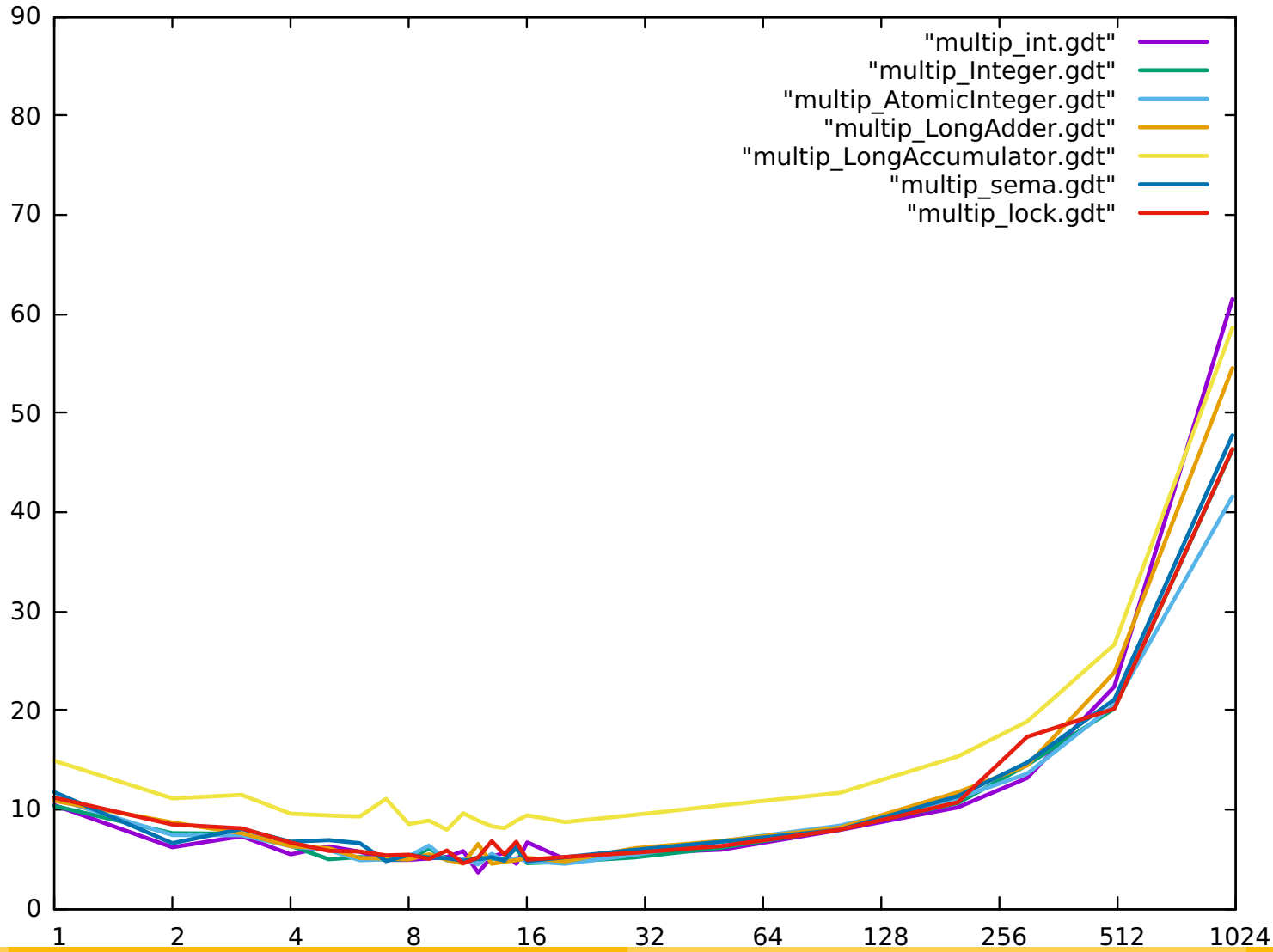
```
public void run() {
  try {
      // Here, we run on local variables.
    int local_r=0;
    while(q>0) {
      local_r+=p;
      --q;
    }
    final Int My_r=new Int(local_r);
    r.Add(My_r);
  }
  catch(Exception E) {
    System.out.println("some error..."+id);
  }
}
```

Multiplicación concurrente correcto y eficiente

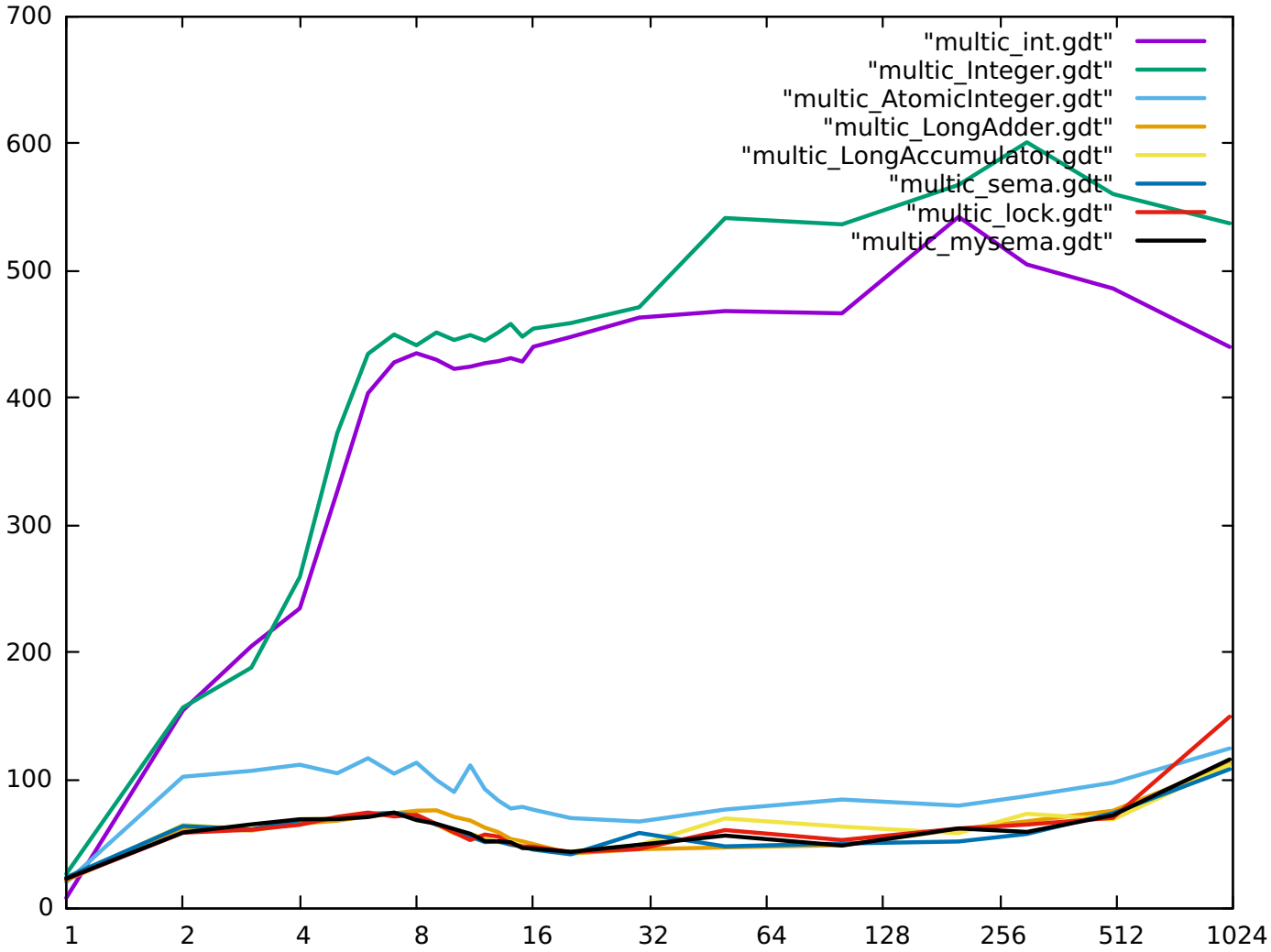
```
}
for(int i=0; i<threads.length; ++i) {
  threads[i].start();
}
  // Here we help with the left-overs.
int local_p=p.Get();
int local_q=q.Get()%n;
int local_r=0;
while(local_q>0) {
  local_r+=local_p;
  --local_q;
}

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)

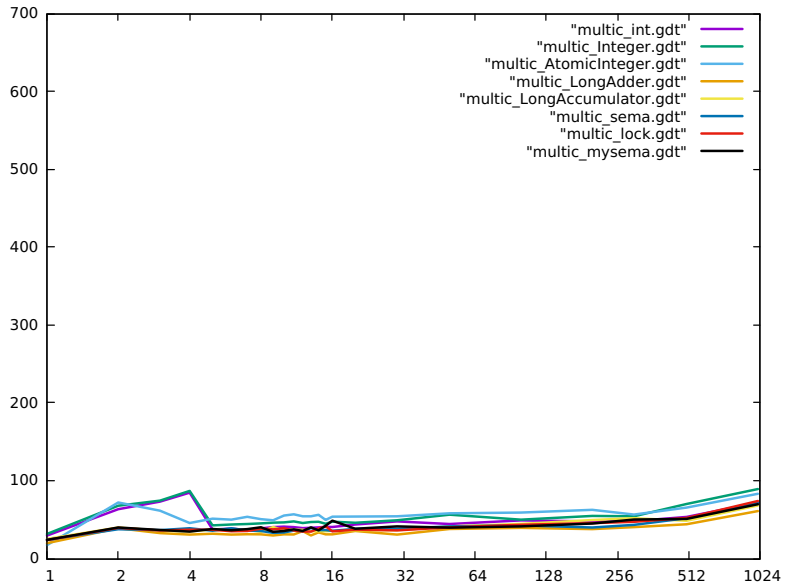


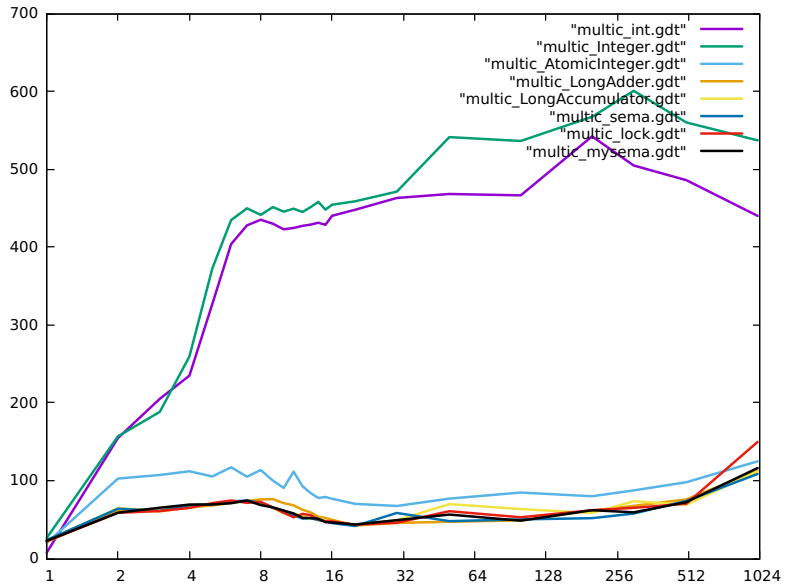
Ojo, si no hacemos las cosas *bien*...



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.)

Sincronización y herencia

- No hace falta mantener el modo sincronizado sobre-escribiendo métodos síncronos mientras se extiende una clase.
- No se puede *forzar* un método sincronizada en una interfaz.
- Sin embargo, una llamada al método de la clase súperior (con `super.`) sigue funcionando de modo síncrono.
- Los métodos estáticos también se pueden declarar `synchronized` garantizando su ejecución de manera exclusiva entre varios hilos.

Protección de miembros estáticos

En ciertos casos se tiene que proteger el acceso a miembros estáticos con un cerrojo. Para conseguir eso es posible sincronizar con un cerrojo de la clase, por ejemplo:

```
class MyClass {
  static private int nextID;
  ...
  MyClass() {
    synchronized(MyClass.class) {
      idNum=nextID++;
    }
  }
  ...
}
```

¡Ojo con el concepto!

- Declarar un bloque o un método como síncrono solo prevee que ningún otro hilo pueda ejecutar al mismo tiempo dicha región crítica (u otra sincronizada con el mismo objeto),
- sin embargo, cualquier otro código asíncrono puede ser ejecutado mientras tanto y su acceso a variables críticas puede dar como resultado fallos o efectos inesperados en el programa.

Objetos síncronos

Se obtienen objetos totalmente sincronizados siguiendo las reglas:

- todos los métodos son `synchronized`,
- no hay miembros/atributos públicos,
- todos los métodos son `final`,
- se inicializa siempre todo bien,
- el estado del objeto se mantiene siempre consistente incluyiendo los casos de excepciones.

Limitaciones para la programación concurrente

- no se puede interrumpir la espera a un cerrojo
  (una vez llegado a un `synchronized` no hay vuelta atrás)
- no se puede influir mucho en la política del cerrojo
  (distinguir entre lectores y escritores, diferentes justicias, etc.)
- no se puede confinar el uso de los cerrojos
  (en cualquier línea se puede escribir un bloque sincronizado de cualquier objeto)
- no se puede adquirir/liberar un cerrojo en diferentes flujos de control, se está obligado a una estructura de bloques

Paquete especial para la programación concurrente

- Por eso, y otros motivos, se ha introducido desde Java 5 un paquete especial para la programación concurrente.

  `java.util.concurrent`

- Hay que leer/estudiar todo su manual.
- Partes mencionaremos en clase de teoría y de prácticas en su momento.
- Imprecindible también está el tutorial sobre concurrencia de Java.

un programa concurrente

- Asumimos que tengamos un programa concurrente que quiere realizar acciones con recursos.
  (por ejemplo, los factores de la multiplicacion, o mirad las prácticas)
- Si los recursos de los diferentes procesos son diferentes no hay problema (mira por ejemplo la práctica de filtrado de matriz),
- Si dos (o más procesos) quieren manipular el mismo recurso ¿Qué hacemos?
- Vimos ya ayudas Java como `AtomicInteger`, o el `synchronized`, o los otros...
- Levantamos el nivel a algo más abstracto, revisamos los ya mencionados y luego implementamos a nivel bajo.

un programa concurrente

Tenemos básicamente tres opciones para tratar posibles escrituras concurrentes:

- se implementa exclusión mutua, es decir, solamente un proceso tiene acceso, los demás esperan;
- se implementa comportamiento idéntico, es decir, desde el algoritmo se garantiza que todos los procesos actuan igual (sobre todo: escriben lo mismo en caso que escriban concurrentemente);
- se implementa comportamiento transaccional, es decir, solo un proceso gana, lo que hacen los demás no influe en su resultado (con la opción que los demás se notifiquen en caso de fracaso) (con la opción que cualquiera o uno específico gana).

¿Qué es exclusión mutua?

- Para evitar el acceso concurrente a recursos compartidos hace falta instalar un mecanismo de control
  - que permite la entrada de un proceso si el recurso está disponible y
  - que prohibe la entrada de un proceso si el recurso está ocupado.
- Es importante entender cómo se implementan los protocolos de entrada y salida para realizar la exclusión mutua.
- Para implementar exclusión mutua se necesita algo básico a nivel hardware.
- Un método es usar un tipo de protocolo de comunicación basado en las instrucciones básicas disponibles en el hardware.
  (Eso veremos más adelante.)

estructura general basada en protocolos

Entonces el protocolo para cada uno de los participantes refleja una estructura como sigue (si protegemos código):

```
P0                          ... Pi
...                             ...
entrance protocol               entrance protocol
critical section                critical section
exit protocol                   exit protocol
...                             ...
```

Enteros con `lock`

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

// Implementation of out integer with a reentrant lock.
class Int {
  private int i;
  private ReentrantLock lock;
  Int(int i) {
    this.i=i;
    lock=new ReentrantLock();
  }
  void Add(Int I) {
    lock.lock(); // Entrance protocol.
    try {
      i+=I.i;
    } finally {
      lock.unlock(); // Exit protocol.
    }
  }
  int Get() { return i; }
}
```

conceptos

- El concepto de usar estructuras de datos a nivel alto libera al/a programador/a de los detalles de su implementación.
- Se puede asumir que las operaciones están implementadas correctamente y se puede basar el desarrollo del programa concurrente en un funcionamiento correcto de las operaciones de los tipos de datos abstractos.
- Para que se puedan utilizar con provecho hay que entender en detalle las propiedades de tales estructuras de datos.

región crítica

- Un lenguaje de programación puede realizar directamente una implementación de una región crítica.
- Así parte de la responsabilidad se traslada desde el programador al compilador.
- De alguna manera (depende del lenguaje de programación en concreto) se identifica que algún bloque de código se debe tratar como región crítica
  (así funciona Java con sus bloques sincronizados):

```
V is shared variable
region V do
  code of critical region
```

observaciones

- El compilador asegura que la variable $\mathbb{V}$ tenga un protocolo de entrada y salida adjunto que se usa para controlar el acceso exclusivo de un solo proceso a la región crítica.
- De este modo no hace falta que el programador use directamente las operaciones de los protocolos para controlar el acceso con el posible error de olvidarse de alguna parte esencial.
- Adicionalmente es posible que dentro de la región crítica se llame a otra parte del programa que a su vez contenga una región crítica. Si ésta está controlada por la misma variable $\mathbb{V}$ el proceso obtiene automáticamente también acceso a dicha región.

regiones críticas condicionales

- En muchas situaciones es conveniente controlar el acceso de varios procesos a una región crítica por una condición.
- Con las regiones críticas simples, vistas hasta ahora, no se puede realizar tal control. Hace falta otra construcción, por ejemplo:

```
V is shared variable
C is boolean expression
region V when C do
  code of critical region
```

- Mira el uso de `Condition` junto con `Lock` en Java (lo veremos en las prácticas del productor/consumidor).

detalles de implementación

Las regiones críticas condicionales pueden funcionar internamente de la siguiente manera:

- Un proceso que quiere entrar en la región crítica espera hasta que tenga permiso.
- Una vez obtenido permiso comprueba el estado de la condición, si la condición lo permite, entra en la región, en caso contrario, libera el cerrojo y se pone de nuevo esperando en la cola de acceso.

semáforo

Un semáforo es un tipo de datos abstracto que permite el uso de un recurso de manera exclusiva cuando varios procesos están competiendo y que cumple la siguiente semántica:

- El estado interno del semáforo cuenta cuantos procesos todavía pueden utilizar el recurso. Se puede realizar, por ejemplo, con un número entero que nunca debe llegar a ser negativo.
- Existen tres operaciones con un semáforo: `init()`, `wait()`, y `signal()` que realizan lo siguiente:

operaciones del semáforo I

`init()`:

- Inicializa el semáforo antes de que cualquier proceso haya ejecutado ni una operación `wait()` ni una operación `signal()` al límite de número de procesos que tengan derecho a acceder el recurso.
- Si se inicializa con 1, se ha construido un semáforo binario.
- En lenguajes orientados a objectos, la operación `init()` se suele realizar en la construcción del objeto correspondiente.

operaciones del semáforo II

`wait()`:

- Si el estado indica cero, el proceso se queda atrapado en el semáforo hasta que sea despertado por otro proceso.
- Si el estado indica que un proceso más puede acceder el recurso se decrementa el contador y la operación termina con exito.
- La operación `wait()` tiene que estár implementada como una instrucción atómica. Sin embargo, en muchas implementaciones la operación `wait()` puede ser interrumpida.
- Normalmente existe una forma de comprobar si la salida del `wait()` es debido a una interrupción o porque se ha dado acceso al semáforo.
- Importante: esta comprobación se debe hacer!

operaciones del semáforo III

`signal()`:

- Una vez se ha terminado el uso del recurso, el proceso lo señaliza al semáforo. Si queda algún proceso bloqueado en el semáforo, uno de ellos sea despertado, sino se incrementa el contador.
- La operación `signal()` también tiene que estár implementada como instrucción atómica. En algunás implementaciones es posible comprobar si se ha despertado un proceso con exito en caso que había alguno bloqueado.
- Para despertar los procesos se pueden implementar varias formas que se destinguen en su política de justicia (p.ej. FIFO).

uso del semáforo

El acceso mutuo a secciones críticas se arregla con un semáforo que permita el acceso a un sólo proceso

```
S.init(1)
```

```
P1                              P2
a: loop                             loop
b:    S.wait()                        S.wait()
c:    critical section                critical section
d:    S.signal()                      S.signal()
e:    non-critical section            non-critical section
f: endloop                          endloop
```

Enteros con Semaphore

```
import java.util.concurrent.Semaphore;

// Implementation of our integer with a semaphore.
class Int {
  private int i;
  private Semaphore semaphore;
  Int(int i) {
    this.i=i;
    semaphore=new Semaphore(1);
  }
  void Add(Int I) {
    try {
      semaphore.acquire(); // Entrance protocol.
      i+=I.i;
    }
    catch(InterruptedException E) {
      System.out.println("got interrupted...??");
    }
    finally {
      semaphore.release(); // Exit protocol.
    }
  }
  int Get() { return i; }
}
```

semáforos binarios/generales

Si existen en un entorno solamente semáforos binarios (es decir, cerrojos simples), se puede simular un semáforo general usando dos semáforos binarios y un contador, por ejemplo, con las variables `delay`, `mutex` y `count`.

- La operación `init()` inicializa el contador al número máximo permitido.
- El semáforo `mutex` asegura acceso mutuamente exclusivo al contador.
- El semáforo `delay` atrapa a los procesos que superan el número máximo permitido.

detalles de la implementación I

La operación `wait()` se implementa de la siguiente manera:

```
delay.wait()
mutex.wait()
decrement count
if count greater 0 then delay.signal()
mutex.signal()
```

detalles de la implementación II

La operación `signal()` se implementa de la siguiente manera:

```
mutex.wait()
increment count
if count equal 1 then delay.signal()
mutex.signal()
```

Implementación de este semáforo en Java

```
class MySemaphore {
  private int cnt;
  private ReentrantLock mutex;
  private ReentrantLock delay;
  MySemaphore(int n) {
    cnt=n;
    mutex=new ReentrantLock();
    delay=new ReentrantLock();
  }
  public void acquire() {
    delay.lock();
    mutex.lock();
    --cnt;
    if(cnt>0) delay.unlock();
    mutex.unlock();
  }
  public void release() {
    mutex.lock();
    ++cnt;
    if(cnt==1) delay.unlock();
    mutex.unlock();
  }
}
```

principales desventajas de semáforos

- No se puede imponer el uso correcto de las llamadas a los `wait()`s y `signal()`s.
- No existe una asociación entre el semáforo y el recurso.
- Entre `wait()` y `signal()` el usuario puede realizar cualquier operación con el recurso.

observaciones II

- Las regiones críticas no son lo mismo que los semáforos, porque no se tiene acceso directo a las operaciones `init()`, `wait()` y `signal()`.
- Con semáforos se puede emular regiones críticas pero no al revés.

Multiplicación concurrente correcto

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

- En cada iteración del bucle de cálculo todos los hilos compiten por el acceso exclusivo a `q` y `r`.
- Es más eficiente dividir el trabajo por hacer en trozos que puede realizar cada hilo independiente de los demás.
- Y solamente al final se une los resultados individuales.

Multiplicación concurrente correcto y eficiente

Usamos variables privados dentro de los trabajadores:

```
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

Ejecutamos el bucle central con las variables locales, solamente al final se suma a la variable compartida (con exclusión mutua).

```
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

El hilo principal ayuda con el trabajo no distribuido:

```
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);
r.Add(My_r);
```

monitor

Un monitor es un tipo de datos abstracto que contiene

- un conjunto de procedimientos de control de los cuales solamente un subconjunto es visible desde fuera,
- un conjunto de datos privados, es decir, no visibles desde fuera.

y permite realizar operaciones/transacciones con exclusión mutua.

detalles de implementación de un monitor

- El acceso al monitor está permitido solamente a través de los métodos públicos y el compilador garantiza exclusión mutua para todos los accesos.
- La implementación del monitor controla la exclusión mutua con mecanismos de entrada que contengan todos los procesos bloqueados mientras haya uno accediendo.
- Pueden existir varias colas (o estructuras de datos) y el controlador/planificador del monitor elige de cual cola se va a escoger el siguiente proceso para actuar sobre los datos.
- Un monitor no tiene acceso a variables exteriores con el resultado de que su comportamiento no puede depender de ellos.
- Una desventaja de los monitores es la exclusividad de su uso, es decir, la concurrencia está limitada si muchos procesos hacen uso del mismo monitor.

sincronización condicional

- Un aspecto que se encuentra en muchas implementaciones de monitores es la sincronización condicional, es decir, mientras un proceso está ejecutando un procedimiento del monitor (con exclusión mutua) puede dar paso a otro proceso liberando el cerrojo. (Estas operaciones se suele llamar `wait` o `delay`).
- El proceso que ha liberado el cerrojo se queda bloqueado hasta que otro proceso le despierta de nuevo. Este bloqueo temporal está realizado dentro del monitor.
- Dicha técnica se refleja en Java con `wait()` y `notify()` o `notifyAll()`.
- La técnica permite la sincronización entre procesos porque, actuando sobre el mismo recurso, los procesos pueden cambiar el estado del recurso y pasar así información de un proceso a otro.

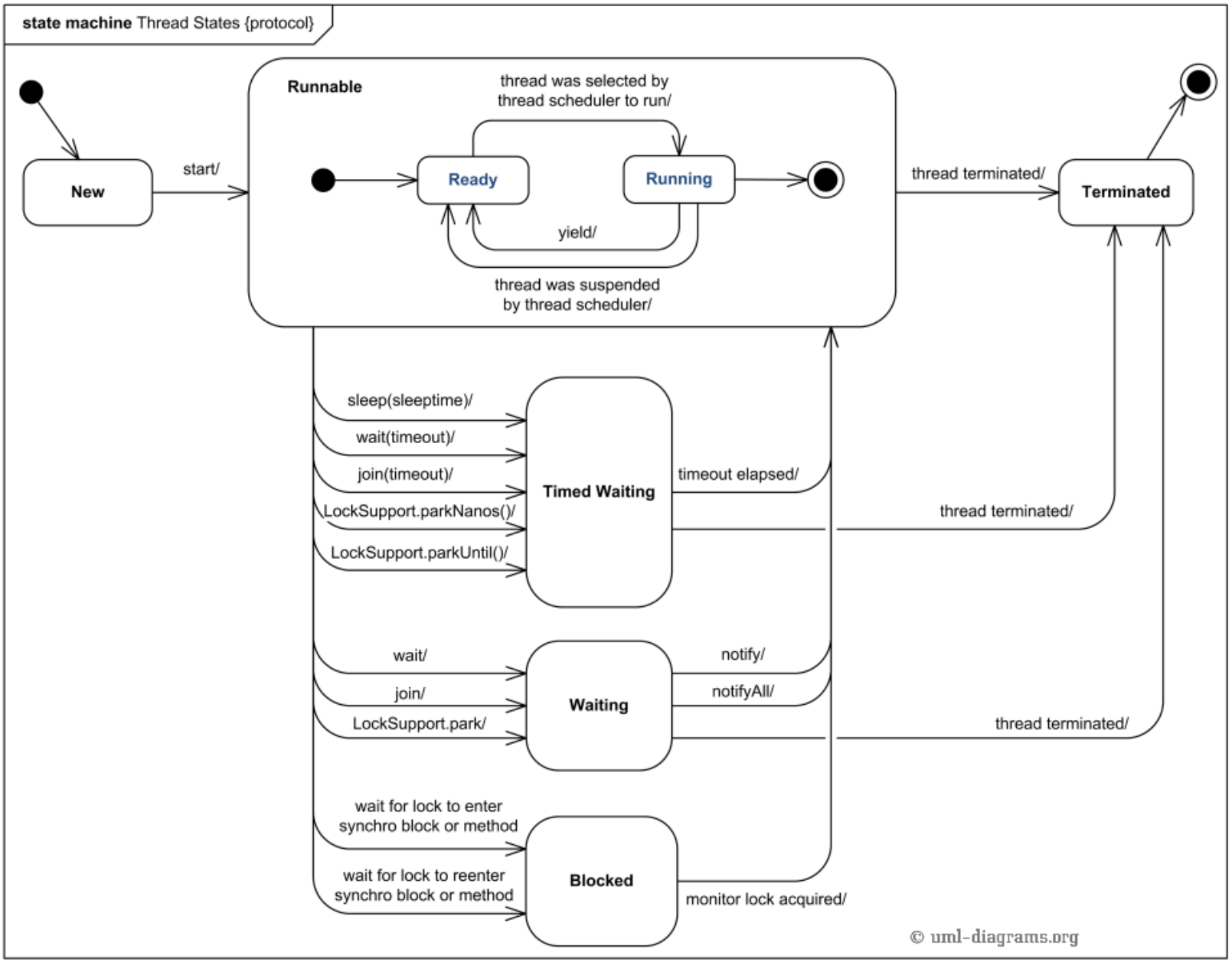
disponibilidad de monitores

- Lenguajes de alto nivel que facilitan la programación concurrente suelen tener monitores implementados dentro del lenguaje (por ejemplo en Java: todos los objetos derivan de `Object` que contiene los métodos `wait()`, `notify`, y `notifyAll()`.
- El uso de monitores es bastante costoso, porque se puede perder eficiencia por bloquear los procesos innecesariamente y el trabajo adicional por el uso del monitor.
- Por eso se intenta aprovechar de primitivas más potentes para aliviar este problema (alternativas libres de cerrojos (*lock free*) y/o libres de espera (*wait free*)).

desventajas de uso de sincronización a alto nivel

- No se distingue entre accesos de solo lectura y de escritura que limita la posibilidad de accesos en paralelo.
- Cualquier interrupción (p.ej. por falta de página de memoria) relantiza el avance de la aplicación,
- por eso las MVJ usan los procesos del sistema operativo para implementar los hilos, así el S.O. puede conmutar a otro hilo.
- Sigue presente el problema de llamar antes a `notify()`, o `notifyAll()` que a `wait()`
  (condición de carrera o *race condition*).

Java máquina de estados de hilos



instrucciones básicas

- Obviamente tenemos que asumir que ciertas acciones de un proceso se pueden realizar correctamente independientemente de las acciones de los demás procesos.
- Dichas acciones se llaman (también) atómicas (porque son indivisibles) y se garantizan por hardware.
- Normalmente, asumimos que podemos acceder a variables de cierto tipo (p.ej. enteros) de forma atómica con lectura y escritura (`load` y `store`)

Implementamos varios protocolos de exclusión mutua (aquí solamente para dos procesos) que solamente están basadas en instrucciones simples.

Un posible protocolo (asimétrico)

```
  P0                                  P1
a: loop                               loop
b:  non-critical section               non-critical section
c:  set v0 to true                     set v1 to true
d:  wait until v1 equals false         while v0 equals true
e:                                      set v1 to false
f:                                      wait until v0 equals false
g:                                      set v1 to true
h:  critical section                   critical section
i:  set v0 to false                    set v1 to false
j: endloop                            endloop
```

principio (teorema) de la bandera

El principio de la bandera es un teorema que podemos usar para comprobar si está garantizado la exclusión mutua para dos procesos en un código concreto.

- Si dos procesos primero levantan sus banderas
- y después miran al otro lado
- por lo menos uno de los procesos ve la bandera del otro levantado.

comprobación con contradicción

- asumimos P0 era el último en mirar
- entonces la bandera de P0 está levantada
- asumimos que P0 no ha visto la bandera de P1
- entonces P1 ha levantado la bandera después de la mirada de P0
- pero P1 mira después de haber levantado la bandera
- entonces P0 no era el último en mirar

propiedades de interés del protocolo

Normalmente, un protocolo de entrada y salida debe cumplir con las siguientes condiciones:

- sólo un proceso debe obtener acceso a la sección crítica (garantía del acceso con exclusión mutua)
- por lo menos un proceso debe obtener acceso a la sección crítica después de un tiempo de espera *finito*.
- Obviamente se asume que ningún proceso ocupa la sección crítica durante un tiempo infinito.

propiedades de interés del protocolo

La propiedad de espera finita se puede analizar según los siguientes criterios:

justicia:
: hasta que medida influyen las peticiones de los demás procesos en el tiempo de espera de un proceso

espera:
: hasta que medida influyen los protocolos de los demás procesos en el tiempo de espera de un proceso

tolerancia a fallos:
: hasta que medida influyen posibles errores de los demás procesos en el tiempo de espera de un proceso.

análisis del protocolo asimétrico

Analizamos el protocolo de antes respecto a dichos criterios:

- ¿Está garantizado la exclusión mutua?
- ¿Influye el estado de uno (sin acceso) en el acceso del otro?
- ¿Quién gana en caso de peticiones simultáneas?
- ¿Qué pasa en caso de error?

soporte hardware

- Dependiendo de las capacidades del hardware la implementación de los protocolos de entrada y salida es más fácil o más difícil, además las soluciones pueden ser más o menos eficientes.
- Vimos que se pueden realizar protocolos seguros solamente con las instrucciones `load` y `store` de un procesador.
- Las soluciones no suelen ser muy eficientes, especialmente si muchos procesos compiten por la sección crítica. *Pero: su desarrollo y la presentación de la solución ayuda en entender el problema principal.*
- A veces no hay otra opción disponible.
- Todos los microprocesadores modernos proporcionan instrucciones básicas que permiten realizar los protocolos de forma más directo y en muchas ocaciones más eficiente.

protocolo simple: primer intento

Usamos una variable `v` que nos indicará cual de los dos procesos tiene su turno.

```
  P0                                P1
a: loop                              loop
b:  wait until v equals P0            wait until v equals P1
c:  critical section                  critical section
d:  set v to P1                       set v to P0
e:  non-critical section              non-critical section
f: endloop                           endloop
```

primer intento: propiedades

- Está garantizada la exlusión mutua porque un proceso llega a su línea `c:` solamente si el valor de `v` corresponde a su identificación (que asumimos siendo única).
- Obviamente, los procesos pueden acceder al recurso solamente alternativamente, que puede ser inconveniente porque acopla los procesos fuertemente.
- Un proceso no puede entrar más de una vez seguido en la sección crítica.
- Si un proceso termina el programa o no llega más por alguna razón a su línea `d:`, el otro proceso puede resultar bloqueado.
- La solución se puede ampliar fácilmente a más de dos procesos.

primer intento en Java (comenzar es fácil)

El protocolo del primer intento para implementar un ping pong.

```
public class PingPong {
  volatile static int turn;  // It's v.

  public static void main(String[] args) {
    Thread ping1 = new Player(1);
    Thread ping2 = new Player(2);

    ping1.start();
    ping2.start();

    turn=1;
  }
```

primer intento en Java (comenzar es fácil)

```
class Player extends Thread {
  private int id;
  public Player(int id) { this.id=id; }
  public void run() {
    while(true) {                          // a:
      while(
        id!=PingPong.turn                  // b:
      );
      System.out.print("ping"+id+" ");// c:
      PingPong.turn=id%2+1;                // d:
    }                                      // f:
  }
}
```

Excurso: primer intento en Java (terminar no es tan fácil)

un método para abreviar:

```
  static void Wait(int us) {
    try {
      Thread.sleep(us);
    } catch(InterruptedException e) {
      System.out.println("sleeping interrupted");
    }
  }
}
```

Excurso: primer intento en Java (terminar no es tan fácil)

```
public class PingPong {
  // volatile static int turn;  // It's v.
  static int turn;  // It's v.
  public static void main(String[] args) {
    Thread ping1 = new Player(1);
    Thread ping2 = new Player(2);
    ping1.start();
    ping2.start();
    System.out.println("playing some seconds");
    turn=1;
    Wait(2000);
    System.out.println("waiting for players");
    turn=3;  // Try to stop :-)
    try {
      ping1.join();
      ping2.join();
    }
    catch(InterruptedException e) {
      System.out.println("got interrupted");
    }
    System.out.println("finished");
```

primer intento en Java (comenzar es fácil)

```
class Player extends Thread {
  private int id;
  public Player(int id) { this.id=id; }
  public void run() {
    while(true) {                          // a:
      while(
        id!=PingPong.turn                  // b:
      );
      System.out.print("ping"+id+" ");// c:
      PingPong.turn=id%2+1;                // d:
    }                                      // f:
  }
}
```

Excurso: primer intento en Java (terminar no tanto)

```
class Player extends Thread {
  private int id;
  public Player(int id) { this.id=id; }
  public void run() {
    while(PingPong.turn!=3) {
      while(
        id!=PingPong.turn &&
        PingPong.turn!=3
      );
      System.out.print("ping"+id+" ");
      if(PingPong.turn==id) {
        PingPong.turn=id%2+1;
      }
    }
  }
}
```

Excurso: primer intento en Java (terminar no tanto)

```
class Player extends Thread {
  private int id;
  public Player(int id) { this.id=id; }
  public void run() {
    while(PingPong.turn!=3) {
      while(
        id!=PingPong.turn &&
        PingPong.turn!=3
      );
      System.out.print("ping"+id+" ");
      if(PingPong.turn==id) {
        PingPong.Wait(100);        // And blocking!!!
        PingPong.turn=id%2+1;
      }
    }
  }
}
```

segundo intento

Intentamos evitar la alternancia. Usamos para cada proceso una variable, `v0` para `P0` y `v1` para `P1` respectivamente, que indican si el correspondiente proceso está usando el recurso.

```
  P0                                  P1
a: loop                               loop
b:  wait until v1 equals false         wait until v0 equals false
c:  set v0 to true                     set v1 to true
d:  critical section                   critical section
e:  set v0 to false                    set v1 to false
f:  non-critical section               non-critical section
g: endloop                            endloop
```

segundo intento: propiedades

- Ya no existe la situación de la alternancia.
- Sin embargo: el algoritmo no es correcto, porque los dos procesos pueden alcanzar sus secciones críticas simultáneamente.

- El problema está escondido en el uso de las variables de control. `v0` se debe cambiar a verdadero solamente si `v1` sigue siendo falso.
- ¿Cuál es la intercalación maligna?

tercer intento

Cambiamos el lugar donde se modifica la variable de control:

```
  P0                                P1
a: loop                              loop
b:  set v0 to true                    set v1 to true
c:  wait until v1 equals false        wait until v0 equals false
d:  critical section                  critical section
e:  set v0 to false                   set v1 to false
f:  non-critical section              non-critical section
g: endloop                           endloop
```

tercer intento: propiedades

- Está garantizado que ambos procesos no entren al mismo tiempo en sus secciones críticas.
- Pero se bloquean mutuamente en caso que lo intenten simultáneamente que resultaría en una espera infinita.

- ¿Cuál es la intercalación maligna?

cuarto intento

Modificamos la instrucción `c:` para dar la oportunidad que el otro proceso encuentre su variable a favor.

```
  P0                              P1
a: loop                           loop
b:  set v0 to true                 set v1 to true
c:  repeat                         repeat
d:   set v0 to false                set v1 to false
e:   set v0 to true                 set v1 to true
f:  until v1 equals false          until v0 equals false
g:  critical section               critical section
h:  set v0 to false                set v1 to false
i:  non-critical section           non-critical section
j: endloop                        endloop
```

cuarto intento: propiedades

- Está garantizado la exclusión mutua.
- Se puede producir una variante de bloqueo: los procesos hacen algo pero no llegan a hacer algo útil (*livelock*)

- ¿Cuál es la intercalación maligna?

algoritmo de Dekker: quinto intento

```
Initially:  v0,v1 are equal to false, v is equal to P0 o P1

  P0                              P1
a: loop                           loop
b:  set v0 to true                 set v1 to true
c:  loop                           loop
d:   if v1 equals false exit        if v0 equals false exit
e:   if v equals P1                 if v equals P0
f:    set v0 to false                set v1 to false
g:    wait until v equals P0         wait until v equals P1
h:    set v0 to true                 set v1 to true
i:   fi                             fi
j:  endloop                        endloop
k:  critical section               critical section
l:  set v0 to false                set v1 to false
m:  set v to P1                    set v to P0
n:  non-critical section           non-critical section
o: endloop                        endloop
```

quinto intento: propiedades

El algoritmo de Dekker resuelve el problema de exclusión mutua en el caso de dos procesos, donde se asume que la lectura y la escritura de un valor íntegro de un registro se puede realizar de forma atómica.

otros algoritmos para obtener exclusión mutua

Existen otros algoritmos que solamente con operaciones atómicas de `load` y `store` consiguen un acceso con exclusión mutua a secciones críticas. Algunos históricos que resuelven para n-procesos:

- algoritmo de Peterson
- algoritmo de Lamport
- algoritmo de Eisenberg–McGuire

Operaciones en la memoria

- Existen instrucciones más potentes (que los simples `load` y `store`) en los microprocesadores actuales para la realización la exclusión mutua más fácil.
- (Casi) todos los procesadores implementan varios tipos de instrucciones atómicas que realizan algún cambio en la memoria al mismo tiempo que devuelve el contenido anterior de la memoria.
- La idea principal es implementar ciclos de *read-modify-write* de forma atómica directamente en memoria compartida.
- El hecho que tienen que actuar finalmente en memoria compartida suelen tener una eficiencia reducida ya que no pueden aprovechar tanto de la jerarquia de memoria (cachés).

TAS

La instrucción `test-and-set` (TAS) implementa

- una comprobación a cero del contenido de una variable en la memoria
- al mismo tiempo que varía su contenido (normal a 1 en hardware)
- en caso que la comprobación se realizó con el resultado verdadero.
- https://en.wikipedia.org/wiki/Test-and-set

CAS

- La instrucción `compare-and-swap` (CAS) es una generalización de la instrucción `test-and-set`.
- La instrucción trabaja con dos variables, les llamamos `C` (de *compare*) y `S` (de *swap*).
- Se intercambia el valor en la memoria por `S` si el valor en la memoria es igual que `C`.
- Se devuelve un booleano con un valor dependiendo si se ha realizado el intercambio o no.
- https://en.wikipedia.org/wiki/Compare-and-swap
- Es la operación que se usa por ejemplo en los procesadores de Intel y es la base para facilitar la concurrencia en la máquina virtual de Java 1.5 para dicha familia de procesadores.

otras posibilidades

Existen otras operaciones hardware para conseguir sincronización entre procesos en la memoria, ejemplos son:

- EXCH (atomic exchange)
- F&A (fetch-and-add)
- DCAS (double compare-and-swap)
- LL/SC (link load / store conditional)
- y muchos más...

problema

Un bloqueo se produce cuando un proceso está esperando algo que nunca se cumple.

Ejemplo:

Cuando dos procesos P_0 y P_1 quieren tener acceso simultaneamente a dos recursos r_0 y r_1, es posible que se produzca un bloqueo de ambos procesos. Si P_0 accede con éxito a r_1 y P_1 accede con éxito a r_0, ambos se quedan atrapados intentando tener acceso al otro recurso.

condiciones necesarias

Se tienen que cumplir cuatro condiciones para que sea posible que se produzca un bloqueo entre procesos:

1. los procesos tienen que compartir recursos con exclusión mutua
2. los procesos quieren acceder a un recurso más mientras ya tienen acceso exclusivo a otro
3. los recursos solo permiten ser usados por menos procesos que lo intentan al mismo tiempo
4. existe una cadena circular entre peticiones de procesos y asignaciónes de recursos

funcionamiento parcial

Un problema adicional con los bloqueos es que es posible que el programa siga funcionando correctamente según la definición, es decir, el resultado obtenido es el resultado deseado, pero algunos de sus procesos están bloqueados durante la ejecución (es decir, se produjo solamente un bloqueo parcial).

técnicas de actuar

Existen algunas técnicas que se pueden usar para que no se produzcan bloqueos:

- Detectar y actuar
- Evitar
- Prevenir

detectar y actuar

Se implementa un proceso adicional que vigila si los demás forman una cadena circular.
Más preciso, se define el grafo de asignación de recursos:

- Los procesos y los recursos representan los nodos de un grafo.
- Se añade cada vez una arista entre un nodo tipo recurso y un nodo tipo proceso cuando el proceso ha obtenido acceso exclusivo al recurso.
- Se añade cada vez una arista entre un nodo tipo recurso y un nodo tipo proceso cuando el proceso está pidiendo acceso exclusivo al recurso.
- Se eliminan las aristas entre proceso y recurso y al revés cuando el proceso ya no usa el recurso.

actuación

Cuando se detecta en el grafo resultante un ciclo, es decir, cuando ya no forma un grafo acíclico, se ha producido una posible situación de un bloqueo.
Se puede reaccionar de dos maneras si se ha encontrado un ciclo:

- No se da permiso al último proceso de obtener el recurso.
- Sí, se da permiso, pero una vez detectado el ciclo se aborta todos o algunos de los procesos involucrados.

desventaja

Sin embargo, las técnicas pueden dar como resultado que el programa no avance, incluso, el programa se puede quedar atrapado haciendo trabajo inútil: crear situaciones de bloqueo y abortar procesos continuamente.

evitar

El sistema no da permiso de acceso a recursos si es posible que el proceso se bloquee en el futuro.
Un método es el algoritmo del banquero (Dijkstra) que es un algoritmo centralizado y por eso posiblemente no muy practicable en muchas situaciones.
Se garantiza que todos los procesos actuan de la siguiente manera en dos fases:

1. primero se obtiene todos los cerrojos necesarios para realizar una tarea, eso se realiza solamente si se puede obtener todos a la vez,
2. después se realiza la tarea durante la cual posiblemente se liberan recursos que no son necesarias.

prevenir

Se puede prevenir el bloqueo siempre y cuando se consiga que alguna de las condiciones necesarias para la existencia de un bloqueo no se produzca.

1. los procesos tienen que compartir recursos con exclusión mutua
2. los procesos quieren acceder a un recurso más mientras ya tienen acceso exclusivo a otro
3. los recursos no permiten ser usados por más de un proceso al mismo tiempo
4. existe una cadena circular entre peticiones de procesos y alocación de recursos

prevenir exclusión mutua

los procesos tienen que compartir recursos con exclusión mutua:

- No se de a un proceso directamente acceso exclusivo al recurso, si no se usa otro proceso que realiza el trabajo de todos los demás manejando una cola de tareas (por ejemplo, un demonio para imprimir con su cola de documentos por imprimir).

prevenir accesos consecutivos

los procesos quieren acceder a un recurso más mientras ya tienen acceso exclusivo a otro:

- Se exige que un proceso pida todos los recursos que va a utilizar al comienzo de su trabajo

prevenir uso único

los recursos no permiten ser usados por más de un proceso al mismo tiempo:

- Se permite que un proceso aborte a otro proceso con el fin de obtener acceso exclusivo al recurso. Hay que tener cuidado de no caer en *livelock*
- (Separar lectores y escritores alivia este problema también.)

prevenir ciclos

existe una cadena circular entre peticiones de procesos y alocación de recursos:

- Se ordenan los recursos línealmente y se fuerza a los procesos que accedan a los recursos en el orden impuesto. Así es imposible que se produzca un ciclo.

Bloqueo en Java

Un ejemplo de un bloqueo en Java muestra el siguiente trozo de código, incluso si se asume que un hilo ya está durmiendo. ¿Por qué?

```
hilo0:

synchronized(A) {
  ...
  synchronized(B) {
    ...
    A.notify();
    B.wait();
  }
}
```

```
hilo1:

synchronized(B) {
  ...
  synchronized(A) {
    ...
    B.notify();
    A.wait();
  }
}
```

problema clásico

El problema del productor y consumidor es un ejemplo clásico de programa concurrente y consiste en la situacion siguiente:
de una parte se produce algún producto (datos en nuestro caso) que se coloca en algún lugar (una cola en nuestro caso) para que sea consumido por otra parte. Como algoritmo obtenemos:

```
producer:
  forever
    produce(item)
    place(item)
```

```
consumer:
  forever
    take(item)
    consume(item)
```

requerimientos

Queremos garantizar que el consumidor no coja los datos más rápido de lo que los está produciendo el productor. Más concreta:

1. el productor puede generar sus datos en cualquier momento, pero no debe producir nada si no lo puede colocar
2. el consumidor puede coger un dato solamente cuando hay alguno
3. para el intercambio de datos se usa una estructura de datos compartida a la cual ambos tienen acceso,
4. si se usa una cola se garantiza un orden temporal
5. ningún dato no está consumido una vez haber sido producido (por lo menos se descarta...)

cola infinita

Si la cola puede crecer a una longitud infinita (siendo el caso cuando el consumidor consume más lento de lo que el productor produce), basta con la siguiente solucion que garantiza exclusion mutua a la cola:

```
producer:                    consumer:
  forever                      forever
    produce(item)                itemsReady.wait()
    place(item)                  take(item)
    itemsReady.signal()          consume(item)
```

donde `itemsReady` es un semáforo general que se ha inicializado al principio con el valor `0`.

más participantes

Queremos ampliar el problema introduciendo más productores y más consumidores que trabajen todos con la misma cola. Para asegurar que todos los datos estén consumidos lo más rápido posible por algún consumidor disponible tenemos que proteger el acceso a la cola con un semáforo binario (llamado `mutex` abajo):

```
producer:
  forever
    produce(item)
    mutex.wait()
    place(item)
    mutex.signal()
    itemsReady.signal()
```

```
consumer:
  forever
    itemsReady.wait()
    mutex.wait()
    take(item)
    mutex.signal()
    consume(item)
```

cola finita

- Normalmente no se puede permitir que la cola crezca infinitamente, es decir, hay que evitar produccion en exceso también.
- Como posible solucion introducimos otro semáforo general (llamado `spacesLeft`) que cuenta cuantos espacios quedan libres en la cola.
- Se inicializa el semáforo con la longitud máxima permitida de la cola.
- Un productor queda bloqueado si ya no hay espacio en la cola y un consumidor señala su consumision.

cola finita

```
producer:
  forever
    spacesLeft.wait()
    produce(item)
    mutex.wait()
    place(item)
    mutex.signal()
    itemsReady.signal()
```

```
consumer:
  forever
    itemsReady.wait()
    mutex.wait()
    take(item)
    mutex.signal()
    consume(item)
    spacesLeft.signal()
```

otras estructuras compartidas

- En un sistema con multiples productores y/o consumidores, puede ser difícil establecer un orden temporal con una semántica adecuada.
- Se puede aflojar la condición de usar una cola, y usar estructuras de datos que permitan más concurrencia.
- Un ejemplo simple serían vectores de contenedores inspeccionados en orden cíclico por los productores y consumidores.
- O se usa estructuras de datos concurrentes ya disponibles en las librerías de programación.