Asignación de Prácticas Número 3

Programación Concurrente y de Tiempo Real

Antonio J. Tomeu[1]

[1]Departamento de Ingeniería Informática
Universidad de Cádiz

PCTR, 2020

Contenido I

Objetivos de la Práctica

- Desarrollar paralelismo de datos por división del dominio de información entre múltiples tareas
- Efectuar la división del dominio de datos de forma manual
- Aprender a medir tiempos de ejecución y desarrollar curvas de tiempos
- Estudiar si la plataforma operativa influye o no en el rendimiento

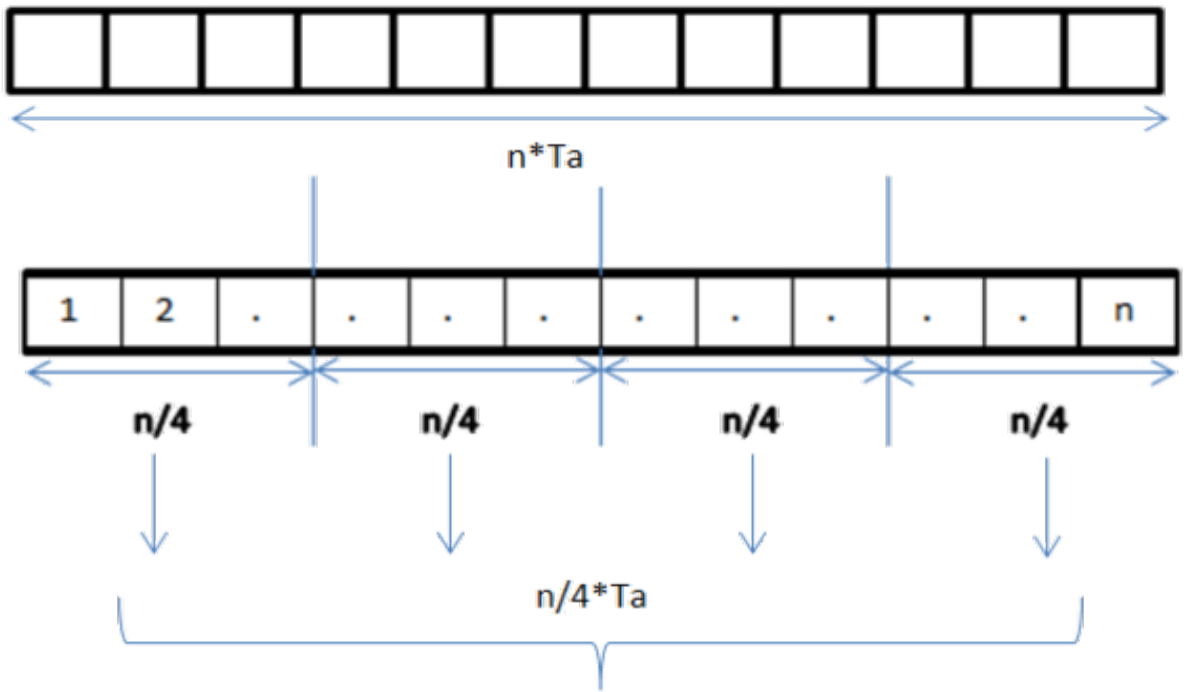
£Qué es el Paralelismo de Datos? I

- Es una (de muchas) técnicas de programación paralela
- Asume una nube de datos situada en memoria común, accesible a todas las hebras paralelas
- Cada hebra paralela procesa un subsegmento de dicha nube, y obtienen una solución parcial
- El programa principal -en su caso- unifica las soluciones parciales en una solución global
- Dependiendo de las relaciones entre los datos, las hebras pueden necesitar o no sincronización
- Cada hebra hace el mismo tipo de trabajo, pero sobre datos diferentes
- Idealmente, cada hebra dispone de un *core* dedicado para hacer su trabajo

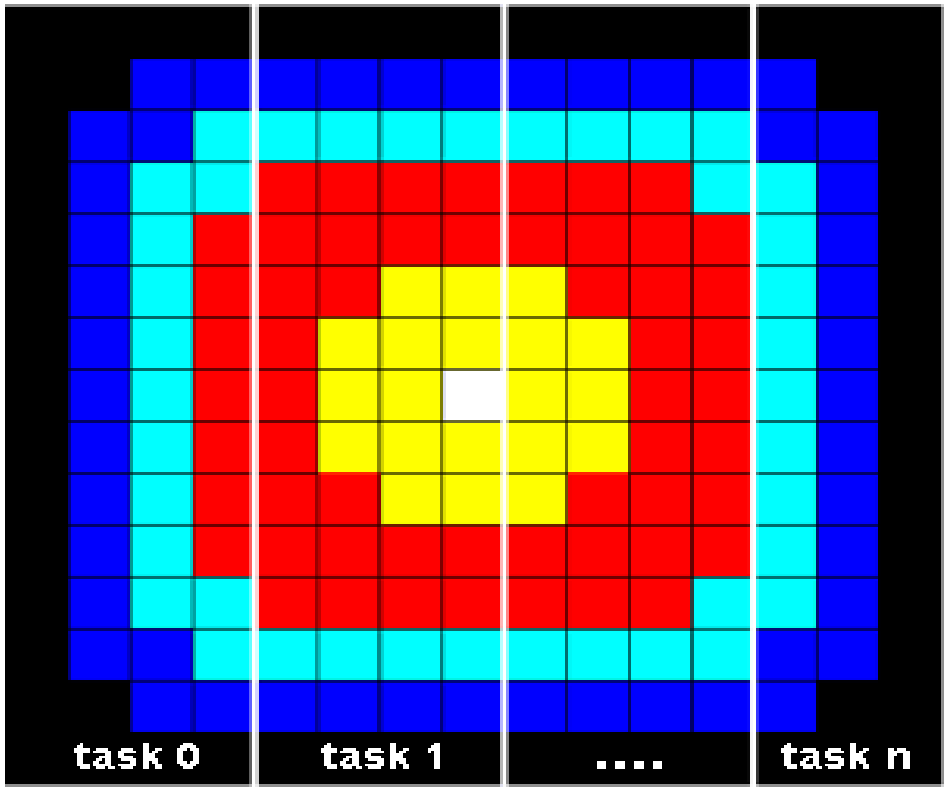
£Qué es el Paralelismo de Datos? II

- En nubes de datos reticulares (vectores, matrices, etc.) muy estructuradas y de gran tamaño, es la técnica ideal para aumentar el rendimiento

Paralelismo de Datos Vectoriales: Gráficamente I



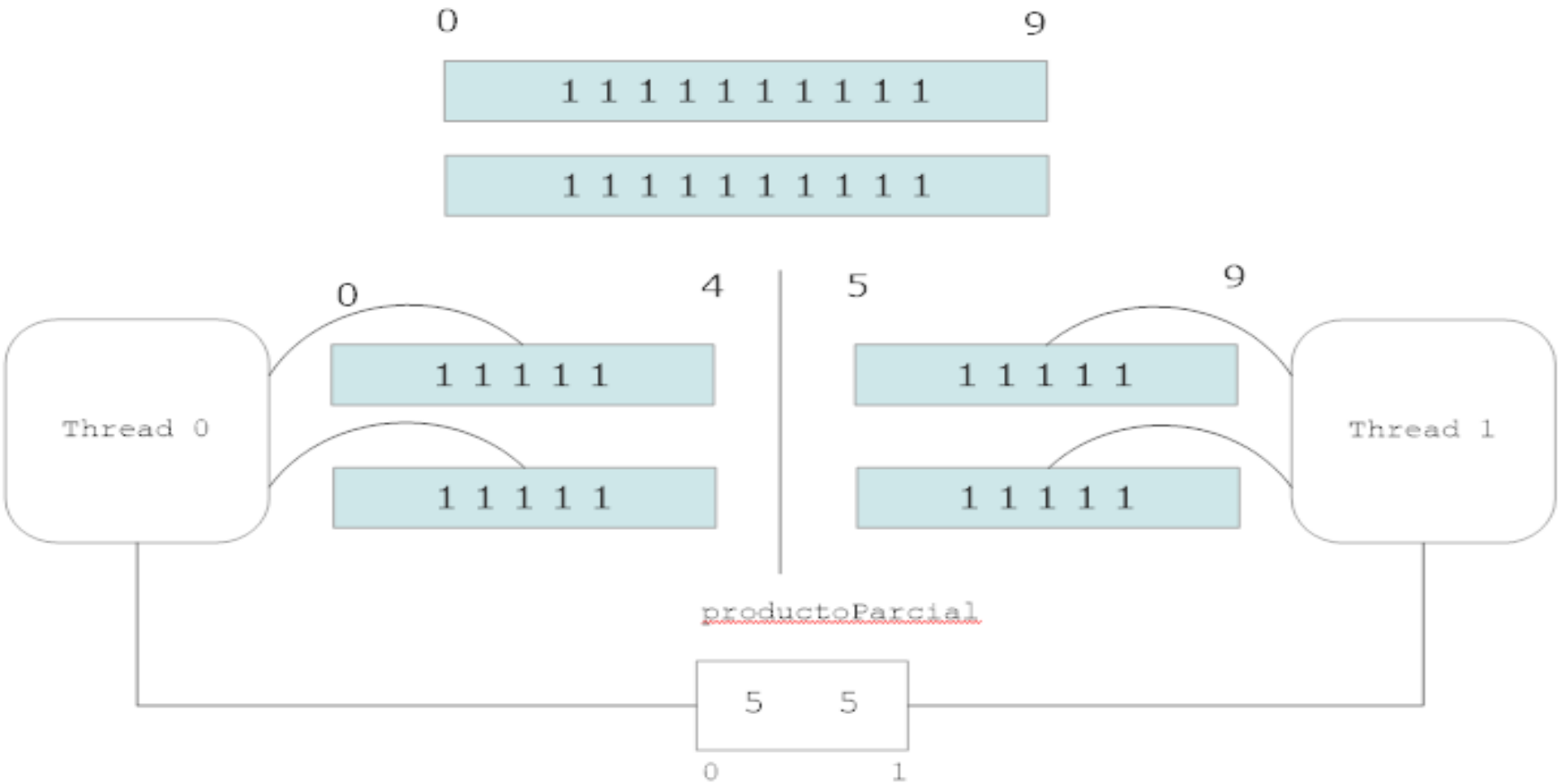
Paralelismo de Datos Matriciales: Gráficamente I



Trabajamos en el Ejercicio 1: Producto Escalar Paralelo I

- Queremos efectuar el producto escalar de dos vectores
- Divideremos el número de componentes de los vectores entre el número de hebras
- Cada hebra efectúa el producto escalar de dos subvectores de la forma habitual...
- ... y almacena su resultado parcial en una ranura diferente de un array `productoParcial`
- El programa principal recorre `productoParcial` y suma las componentes
- Es un paralelismo libro de bloqueos, dada que las hebras leen en zonas diferentes de las fuentes de datos y `escriben en zonas diferentes del vector de resultados` parciales

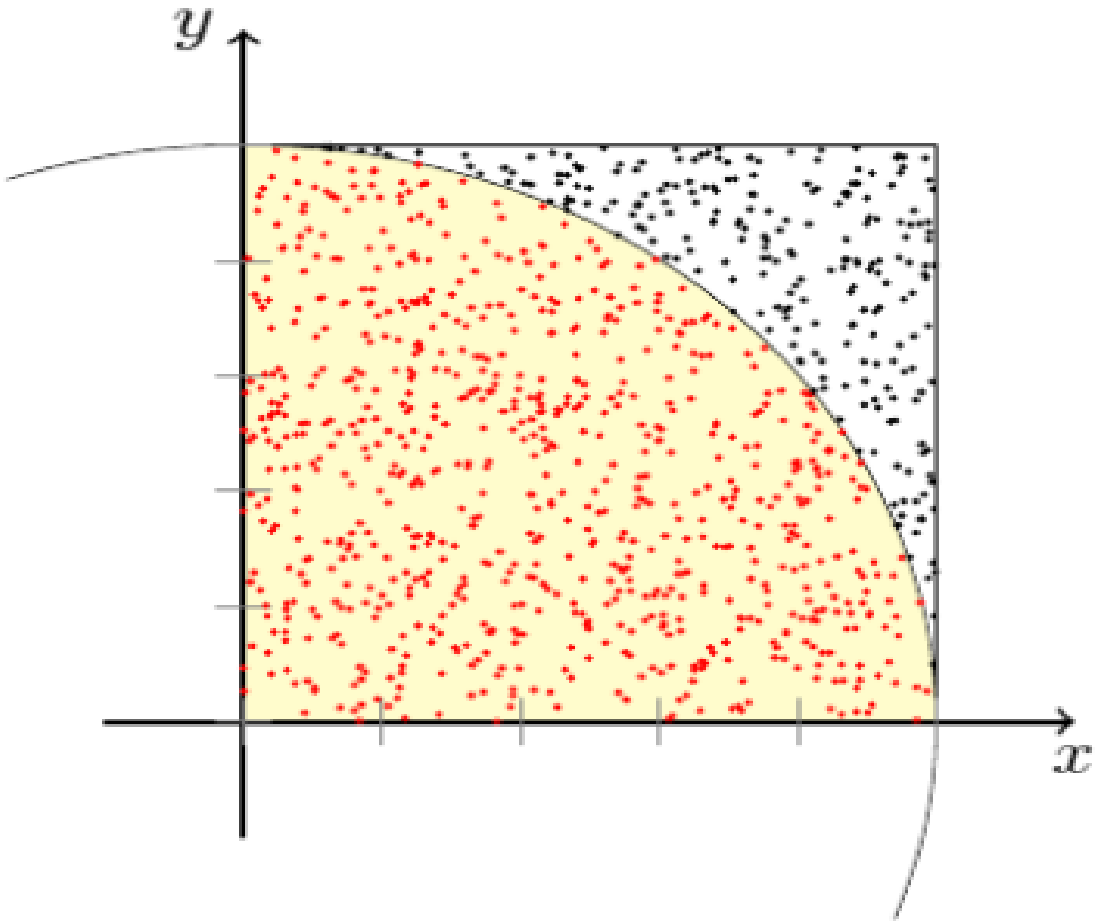
Producto Escalar Paralelo: Gráficamente I



£Cómo mido el tiempo de ejecución? I

- Veamos cómo tomar tiempos de ejecución en Java
- Lo hacemos calculando de forma paralela el valor del número π mediante el método de Monte-Carlo, que aproxima la superficie del segmento del círculo de radio $r = 1$, inscrito en el cuadrado de lado $l = 1$, utilizando la ecuación de la circunferencia con centro en $(0, 0)$ y radio $r = 1$.
- Se divide el número de puntos entre el número de hebras (son muchas en el ejemplo; idealmente con 4-8 bastaría, **pruébese**)
- Se utiliza el método `Math.random` aunque sabemos que está sincronizado y penalizará el rendimiento
- El acumulador de puntos está protegido en exclusión mutua mediante `synchronized` (por ahora, olvídese de esto)

Cálculo de π Paralelo I



Cáculo de π Paralelo I

```
1
2  public class piMonteCarloParalelo extends Thread{
3    private double cx, cy;
4    private static int intentos = 0;
5    private static Object lock = new Object();
6    private long vueltas;
7
8    public piMonteCarloParalelo(long n)
9    {vueltas = n;}
10
11   public void run(){
12     for(long i=0; i<vueltas; i++){
13       cx = Math.random();
14       cy = Math.random();
15       if(Math.pow(cx, 2)+Math.pow(cy, 2)<=1)
16         synchronized(lock){intentos++;}
17     }
18   }
19
20   public static void main(String[] args) throws Exception{
21     long nVueltas   = 100000000;
22     int nThreads    = 2000;
```

Cáculo de π Paralelo II

```
23        long inicTiempo = System.nanoTime();
24        piMonteCarloParalelo[] h = new
              piMonteCarloParalelo[nThreads];
25        for(int i=0; i<nThreads; i++)h[i]=new
              piMonteCarloParalelo((int)(nVueltas/nThreads));
26        for(int i=0; i<nThreads; i++)h[i].start();
27        for(int i=0; i<nThreads; i++)h[i].join();
28        long tiempoTotal =
              (System.nanoTime()-inicTiempo)/(long)1.0e9;
29        System.out.println("Aproximacion: "+4.0*intentos/nVueltas);
30        System.out.println("Valor Real: "+Math.PI);
31         System.out.println("en "+tiempoTotal+" segundos...");
32     }
33  }
```

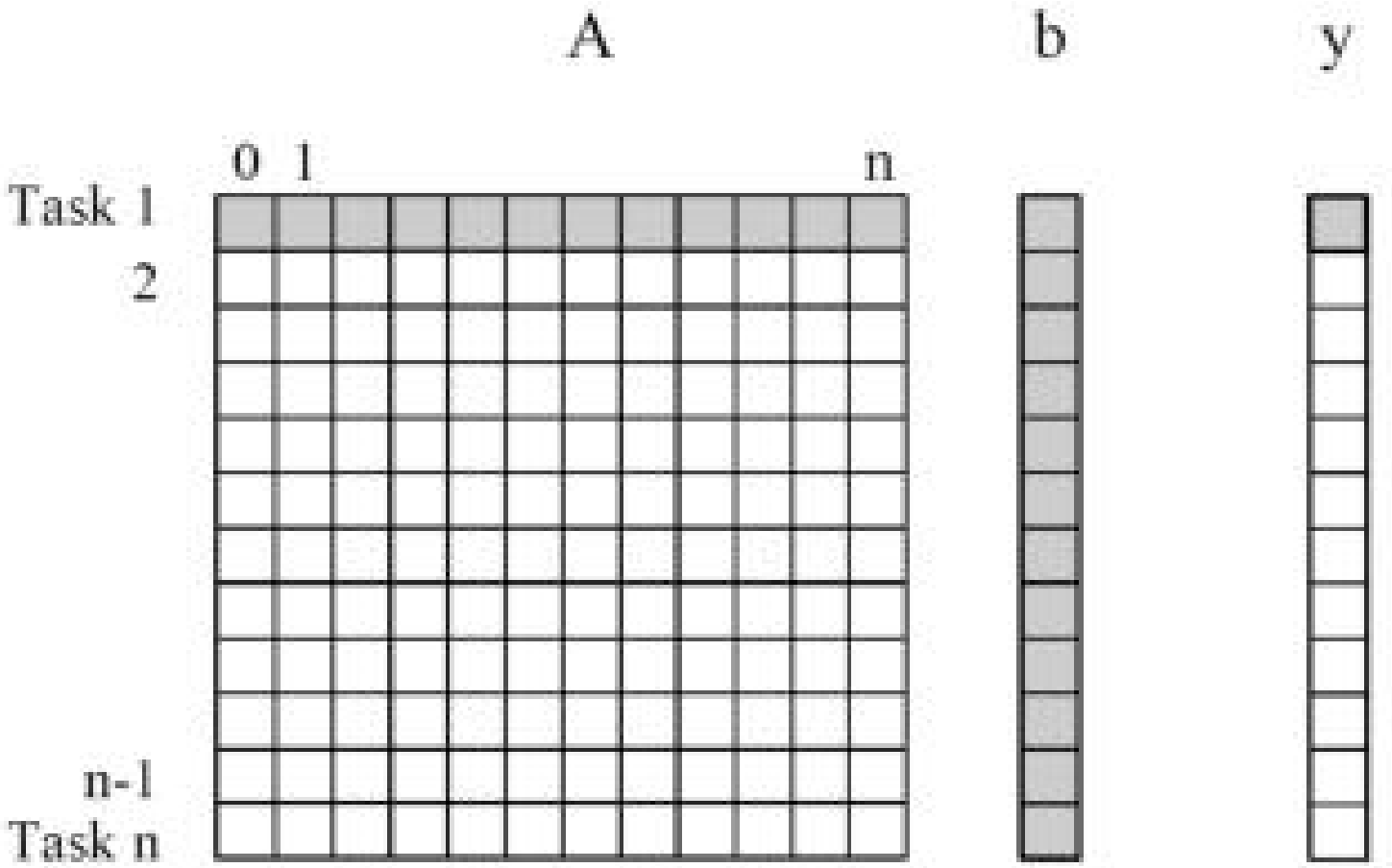
Trabajamos en el Ejercicio 1: £Qué hago ahora? I

- Escribir un programa secuencial para efectuar el producto escalar de dos vectores
- Reutilizar el programa para producto por escalar de la asignación anterior puede ser una buena idea
- Paralelizar diviendo el dominio de datos manualmente con 2, 4, 6,... hebras
- Tome los tiempos de ejecución para todas las versiones
- Construya una tabla de dos columnas: número de hebras y tiempo de ejecución, y saque sus propias conclusiones.
- No tiene por qué haber enormes mejoras (el producto escalar secuencial se puede calcular muy eficientemente)
- Trabajamos en esto 20-30 minutos, y seguimos...

Trabajamos en el Ejercicio 2: Producto Matriz-Vector Paralelo I

- Es una extensión natural de las ideas ya expuestas para el ejercicio anterior
- Escriba una programa secuencial para multiplicar una matriz por un vector
- Lo primero que a uno se le ocurre es tener una hebra por fila pensando que cuantas más hebras, más eficiente será el programa.
- No es así; crear hebras es caro, y cargar con ellas al sistema también (cientos o miles de hebras con matrices grandes, y el planificador secuenciándolas por los *cores* a toda pastilla).
- Es necesario saber determinar el número de hebras adecuado para cada problema (en próximas prácticas).

Trabajamos en el Ejercicio 2: Producto Matriz-Vector Paralelo II



Trabajamos en el Ejercicio 2: Producto Matriz-Vector Paralelo III

- Otra aproximación (mejor): ahora, cada hebra será responsable de un conjunto de filas que irá multiplicando por el vector para obtener el vector resultado. El algoritmo paralelo estándar tampoco lo hace así.
- £Qué hago ahora?
- Paralelizar diviendo el dominio de datos manualmente (número de filas) entre el número de hebras
- Tome los tiempos de ejecución para todas las versiones
- Construya una curva `tiempo=f(número de hebras)`...
- ... y saque sus propias conclusiones (recomendamos *gnuplot* para graficar curvas).
- En esta ocasión, quizás aprecie alguna mejora en los tiempos frente a la versión secuencial

Trabajamos en el Ejercicio 2: Ayuda (Suma Paralela de Matrices Grandes) I

NOTA: La ejecución de este código puede requerir ampliar la memoria de la JVM parametrizando al binario java con el flag -Xmx

```
1  import java.util.*;
2  public class matrizPorFilas
3    extends Thread{
4
5    int [][] matrizInput, matrizOutput;
6    int inicio, fin;
7
8    public matrizPorFilas(int inicio, int fin, int[][] m, int[][]
         n){
9      this.inicio=inicio; this.fin=fin; this.matrizInput=m;
           this.matrizOutput=n;}
10
11   public void run(){
12     for(int i=inicio; i<=fin; i++)
13       for(int j=0; j<matrizInput.length; j++)
```

Trabajamos en el Ejercicio 2: Ayuda (Suma Paralela de Matrices Grandes) II

```
14          matrizOutput[i][j]=matrizInput[i][j]+matrizOutput[i][j];
15      }
16      public static void printMatriz (int[][] m){
17        for(int i=0; i<m.length; i++)
18          for(int j=0; j<m.length; j++)
19            if(j==m.length-1)System.out.println(m[i][j]);
20            else System.out.print(m[i][j]+" ");
21      }
22      public static void main(String[] args)
23        throws Exception{
24       Random generador = new Random();
25       int[][] matrix = new int[24000][24000];
26       int[][] matrixx = new int[24000][24000];
27        System.out.println("Llenando matrices. Este tiempo no
              cuenta...");
28        for(int i=0; i<matrix.length; i++)
29          for(int j=0; j<matrix.length; j++){
30            matrix[i][j]  = generador.nextInt(32000);
31            matrixx[i][j] = generador.nextInt(32000);}
32        System.out.println("Matrices llenas...");
```

Trabajamos en el Ejercicio 2: Ayuda (Suma Paralela de Matrices Grandes) III

```
33        //printMatriz(matrix);
34        //System.out.println("\n");
35        //printMatriz(matrixx);
36        //System.out.println("\n");
37        //procesamiento con una tarea...
38        System.out.println("procesando con una hebra...");
39        long inicTiempo = System.nanoTime();
40        matrizPorFilas h1 = new matrizPorFilas(0, 23999, matrix,
              matrixx);
41        h1.start(); h1.join();
42        long tiempoTotal = System.nanoTime()-inicTiempo;
43        System.out.println("en "+tiempoTotal+" nanosegundos...");
44        //printMatriz(matrixx);
45        System.out.println("Llenando matrices con nuevos valores.
              Este tiempo no cuenta...");
46        for(int i=0; i<matrix.length; i++)
47          for(int j=0; j<matrix.length; j++){
48            matrix[i][j]  = generador.nextInt(32000);
49            matrixx[i][j] = generador.nextInt(32000);}
50        System.out.println("Matrices llenas...");
```

Trabajamos en el Ejercicio 2: Ayuda (Suma Paralela de Matrices Grandes) IV

```
51      //printMatriz(matrix);
52      //System.out.println("\n");
53      //printMatriz(matrixx);
54      System.out.println("Estabilizamos los cores durante unos
            segundos... e insinuamos a la JVM que limpie...");
55      System.gc();
56      Thread mainThread=Thread.currentThread();
57      mainThread.sleep(6000);
58      //procesamiento con cuatro tareas...
59      System.out.println("procesando con ocho hebras...");
60      inicTiempo = System.nanoTime();
61      h1 = new matrizPorFilas(0, 2999, matrix, matrixx);
62      matrizPorFilas h2 = new matrizPorFilas(3000, 5999, matrix,
            matrixx);
63      matrizPorFilas h3 = new matrizPorFilas(6000, 8999, matrix,
            matrixx);
64      matrizPorFilas h4 = new matrizPorFilas(9000, 11999, matrix,
            matrixx);
65      matrizPorFilas h5 = new matrizPorFilas(12000, 14999,
            matrix, matrixx);
```

Trabajamos en el Ejercicio 2: Ayuda (Suma Paralela de Matrices Grandes) V

```
66        matrizPorFilas h6 = new matrizPorFilas(15000, 17999,
              matrix, matrixx);
67        matrizPorFilas h7 = new matrizPorFilas(18000, 20999,
              matrix, matrixx);
68        matrizPorFilas h8 = new matrizPorFilas(21000, 23999,
              matrix, matrixx);
69        h1.start(); h2.start(); h3.start(); h4.start(); h5.start();
              h6.start(); h7.start(); h8.start();
70        h1.join();h2.join(); h3.join(); h4.join(); h5.join();
              h6.join(); h7.join(); h8.join();
71        tiempoTotal = System.nanoTime()-inicTiempo;
72        //xžprintMatriz(matrixx);
73        System.out.println("en "+tiempoTotal+" nanosegundos...");
74    }
75  }
```

Trabajamos en el Ejercicio 3: £Influye el Sistema Operativo En Todo Esto? I

- Se trata ahora de repetir el ejercicio anterior cambiando de sistema operativo
- La idea es ver si la plataforma de base influye algo en el rendimiento
- Evidentemente, será difícil establecer conclusión alguna con carácter definitivo, ya que:
  - los algoritmos de planificación serán distintos
  - los modelos de hebras a las que se mapean las hebras de la JVM quizás sean distintos
  - cada sistema operativo tendrá una carga de trabajo u otra según como esté configurado...
  - ... y media docena de variables más
- Aun así, como mínimo le servirá para experimentar con el paralelismo en Java tanto en Windows como en Linux

Curvas: Tipología y Construcción I

- A partir de los datos en bruto, se pueden trazar curvas sencillas que ilustran cómo estamos mejorando el rendimiento y aprovechando la capacidad de cálculo
- Típicamente interesan curvas de tiempo, uso de CPU y *speedup*
- Ahora, ponemos un ejemplo generado con Excel, para salir del paso...
- ... pero les aconsejamos utilizar GnuPlot, que permite una apariencia mucho más profesional y una configurabilidad enormemente superior... además de integrarse muy bien con LaTeX

Curvas: Tipología y Construcción II