Introducción a la programación paralela

Jesús Sánchez Cuadrado

Resumen. En este documento de describen algunos conceptos básicos sobre la programación paralela. Los objetivos de este tema son.

- Entender qué es la programación paralela y su utilidad.
- Discutir la evolución de los sistemas paralelos.
- Presentar los elementos básicos de las arquitecturas de computación paralela.
- Presentar los diferentes modelos de programación paralela.

Índice

1. ¿Qué es la programación paralela? 1
1.1. Importancia de la computación paralela . . . 1
1.2. Aplicaciones de la programación paralela . . . 3
1.3. Tipos de sistemas paralelos . . . 3
1.4. Tipos de paradigmas de programación . . . 3
1.5. Paralelismo en sistemas secuenciales . . . 4

2. Clasificación de Flynn 4

3. Arquitecturas paralelas 5
3.1. Arquitectura de memoria compartida . . . 5

4. Modelos de programación paralela 6
4.1. Programación multi-hilo . . . 6
4.2. Breve introducción a la programación multi-hilo en Java . . . 7
4.3. Paso de mensajes . . . 9
4.4. Ejemplo . . . 10
4.4.1. Librerías y lenguajes de paso de mensajes . . . 10

1. ¿Qué es la programación paralela?

Tradicionalmente el software se ha desarrollado utilizando el modelo secuencial, tal y como fue propuesto por John Von Neumann (1903-1957). Según este modelo la computación se lleva a cabo ejecutando un secuencia de operaciones o instrucciones, de una en una y de forma sucesiva. Sin embargo, a nivel hardware los ordenadores actuales son intrínsicamente paralelos, como se ha estudiado en las asignaturas de arquitectura de computadores. Para simplificar la programación, desde un punto de vista lógico se asume que la ejecución de estas instrucciones la realiza un único procesador y que hay una única instrucción ejecutándose en cada momento. A este modelo lo denominamos **computación secuencial**.

La **computación paralela** hace referencia al uso de varios recursos de computación, al mismo tiempo. Un computador paralelo ofrece estos recursos, bien en la forma de varios procesadores independientes, varios nodos de computación conectados por una red, o una combinación de ambos. Así, un problema se descompone en partes que puedan ser resultas de manera concurrente, y cada parte tiene su propia secuencia de instrucciones que son ejecutadas en procesadores diferentes, y deberá existir algún mecanismo que permita coordinar el trabajo de todos estos procesadores para obtener un resultado común.

Un **computador paralelo** es un ordenador capaz de ejecutar varias sequencias de instrucciones de manera independiente.

La **programación paralela** hace referencia al conjunto técnicas y paradigmas de programación que hacen posible que un programador pueda aprovechar las características de los computadores paralelos para mejorar las prestaciones de sus aplicaciones. Las técnicas de programación paralela serán diferentes a las de la programación secuencial en tanto en cuanto introducen nuevos mecanismos para dividir el trabajo de un algoritmo en partes independientes que se puedan ejecutar de manera simultánea en un computador paralelo. Por tanto, el término *programación paralela* significa utilizar un conjunto de recursos para resolver algún problema en menos tiempo dividiendo el trabajo.

La mayoría de los programas están escritos pensando en procesadores de un solo núcleo, utilizando técnicas de programación secuencial. Dado un programa secuencial, no es actualmente realista pensar que un compilador es capaz de paralelizarlo para aprovechar varios procesadores. Por tanto, necesitamos conocer los lenguajes y técnicas de programación paralela para convertir estos programas secuenciales en programas paralelos.

Así, se denomina paralelismo a la posiblidad de dividir un problema computacional en partes que se pueden resolver de forma independiente. Nótese, que el paralalelismo es diferente a la concurrencia, ya que esta última no implica necesariamente la ejecución simultánea del código.

1.1. Importancia de la computación paralela

Los ordenadores secuenciales actuales tienen una potencia computacional muy elevada, que ha ido incrementándose a ritmo casi constante desde sus inicios (ver la Ley de Moore). Sin embargo, la mejora de las prestaciones ya se está reduciendo debido principalmente a los límites físicos del hardware. En la figura 1 se observa cómo en los últimos años el incremento en el rendimiento de los procesadores secuenciales ha sido cada vez menor.

Para paliar este problema surgieron los **procesadores multi-núcleo**, que son aquellos que combinan dos o más CPUs independientes dentro del mismo chip. Los procesadores multinúcleo permiten el paralelismo a nivel de hilo (*thread-level parallelism*; TLP), es decir, son capaces de mantener varios hilos de ejecución al mismo tiempo. Para aprovechar este tipo de procesadores es necesario que el

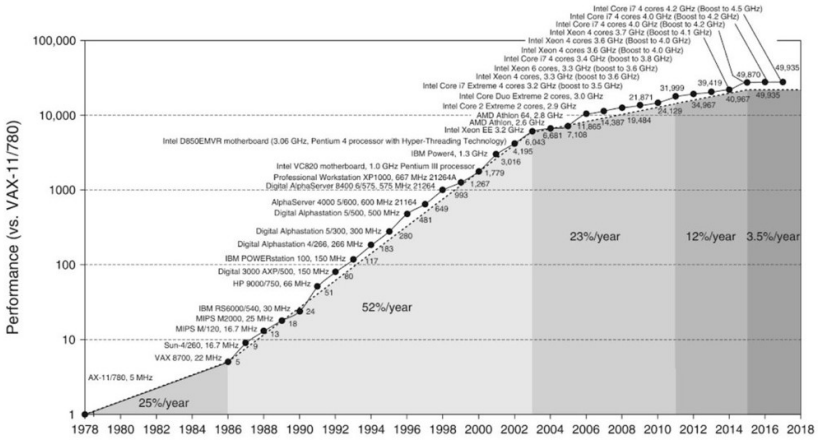

Figura 1: Incremento de las prestaciones de los procesadores secuenciales. Fuente: Hennessy y Patterson [2018]

sistema operativo y los programas estén preparados para ejecutar aplicaciones en paralelo.

En general, un elemento esencial de la computación paralela está en la organización de los recursos de computación. Se trata de organizar el computador (o un conjunto de computadores) como un conjunto de procesadores o nodos de computación que funcionen de manera independiente pero que puedan comunicarse para intercambiar resultados. Así, los computadores paralelos también aprovechan las mejoras tecnológicas de los computadores secuenciales, puesto que cada procesador usa la tecnología secuencial para realizar su trabajo.

Algunas razones por las que el uso de la computación paralela es importante son las siguientes:

- Ahorro de tiempo y dinero, ya que las tareas computacionales complejas terminarán antes y sus resultados estarán disponibles antes (p.ej., predicción meteorológica) lo que podría redundar en ahorro económico.
- Posibilidad de resolver problemas más complejos. Algunos problemas son tan complejos que requieren una capacidad de cálculo que sólo puede ofrecer un computador paralelo. En este sentido la programación paralela es la base de la *computación de altas prestaciones* (High Performance Computing; HPC). Los superordenadores actuales tienen como característica común que todos son paralelos. En `http://www.top500.org` se puede consultar la lista de los 500 ordenadores más rápidos del mundo. El número de núcleos de ejecución se encuentra en el rango de miles a millones. Esto permite abordar problemas computacionalmente muy complejos, pero requiere utilizar técnicas de programación específicas que puedan aprovechar el paralelismo ofrecido.
- La computación paralela ha sido esencial para el desarrollo de muchas aplicaciones comerciales que usamos día a día. Por ejemplo, los motores de búsqueda que deben de indexar millones de webs y estimar su relevancia realizan este trabajo en paralelo. Otros ejemplos son las tecnologías de *big data*, el *machine learning* y las redes neuronales donde el uso de GPUs es esencial, etc.
- Hacer buen uso del hardware paralelo. Los ordenadores actuales son inherentemente paralelos, puesto que ofrecen más de un procesador (*multi-core*). Los programas secuenciales no son capaces de aprovechar esta característica, por lo que se "malgasta" potencia computacional.

Buscadores

El cálculo de la relevancia de las páginas web y su indexación se realiza en paralelo utilizando el paradigma *map-reduce*. [4, 6]. La implementación de software libre más usada es Hadoop.

1.2. Aplicaciones de la programación paralela

La programación paralela se ha aplicado tradicionalmente a la computación científica para resolver problemas en dominios cómo la física, la geología, el modelado del clima, ingeniería, etc. Sin embargo, también se ha aplicado a otros campos:

- Los videojuegos actuales tienen que realidad cálculos relacionados con la física, como dinámica de fluidos, colisiones de cuerpos rígidos, óptica, etc. También hay que renderizar los gráficos, lo cual ser realiza en paralelo utilizando las GPUs de las tarjetas gráficas.
- Las bases de datos también utilizan paralelismo en ciertas partes de sus motores.
- Los navegadores web usan GPUs a través de WebGL y de otras APIs. También es posible utilizar varios hilos de ejecución en un navegador web a través de *web workers*.
- Los compiladores pueden realizar el procesamiento de los ficheros fuente en paralelo utilizando los cores de la CPU. Igualmente, los sistemas de construcción como Gradle también son capaces de construir un programa paralelizando tareas.
- Los sistemas operativos se encargan de gestionar tanto el paralelismo, tanto a nivel de una misma aplicación como entre aplicaciones.
- Los compresores/descompresores pueden funcionar en paralelo.
- Los algoritmos de recolección de basura actuales son capaces de liberar memoria en paralelo mejorando el rendimiento (*throughput*).

Web Workers

Los web workers son una API estandar de Javascript para ejecutar scripts en un navegador en un hilo separado del principal. La comunicación se realiza a través de paso de mensajes.

```
var myWorker =
  new Worker("my_task.js");

myWorker.onmessage =
  function (event) {
    console.log("Mensaje: " + event.
        data);
  };

myWorker.postMessage("hi");

```

1.3. Tipos de sistemas paralelos

Los sistemas computacionales actuales están constituidos por varios componentes, por lo que son sistemas paralelos, pero estos componentes están organizados de formas distintas:

- Sistemas con varios cores y una memoria común a todos. Por ejemplo, un ordenador de escritorio o un portatil moderno.
- Sistemas multiprocesador, donde hay varios procesadores conectados entre sí y compartiendo una memoria.
- Redes de multicores, cada uno con su memoria y posiblemente su tarjeta gráfica. Por ejemplo, un cluster de ordenadores.
- Varias redes conectadas entre sí de forma remota. Por ejemplo, servicios conectados a través de Internet.

Dependiendo del tipo de sistema paralelo es posible que sea mejor (o incluso requerido) utilizar unas técnicas concretas u otras.

1.4. Tipos de paradigmas de programación

- Una única secuencia de ejecución de las instrucciones (programación secuencial).
- Una secuencia de ejecución, con algunas instrucciones que mandan trabajo a realizar a otro componente computacional.

- Distintas secuencias que se ejecutarán en elementos computacionales distintos colaborando a través de la memoria.
- Distintas secuencias que se ejecutarán en elementos computacionales distintos colaborando intercambiando información por medio de mensajes.

1.5. Paralelismo en sistemas secuenciales

Las mejoras tecnológicas propiciadas por las mejores en los materiales semiconductores ha sido una de los principales causas del aumento de las prestaciones de los procesadores. Sin embargo, la inclusión de conceptos del paralelismo para mantener ocupados los elementos funcionales de un procesador han sido una de las principales técnicas que han mejorado las capacidades de los procesadores secuenciales. Una buena introducción al tema está disponible en *Modern Microprocessors A 90-Minute Guide!*[1].

Se recomienda leer el documento citado.

2. Clasificación de Flynn

Existen diferentes tipos de clasificaciones de los computadores paralelos[2], y cada una de ellas hace énfasis en unos elementos del procesamiento. Una de las clasificaciones más utilizada ela *taxonomía de Flynn* [5]. Es una clasificación independiente de la arquitectura del computador, que se basa en el concepto de flujo de instrucciones y flujo de datos. La idea es que una CPU se puede ver como un sistema que ejecuta un flujo de instrucciones y procesa un flujo de datos y escribe de vuelta otro flujo de datos. La clasificación de Flynn se basa distinguir si estos dos elementos funcionan de manera secuencial o en paralelo. La Fig. 2 muestra la cuatro maneras en que se pueden combinar el flujo de instrucciones y el flujo de datos. El flujo de instrucciones se representa con `I`, y el flujo de datos con `D`. Si un flujo es secuencial se representa con `S` (de *single*) y si funciona en paralelo se representa con `M` (de *multiple*).

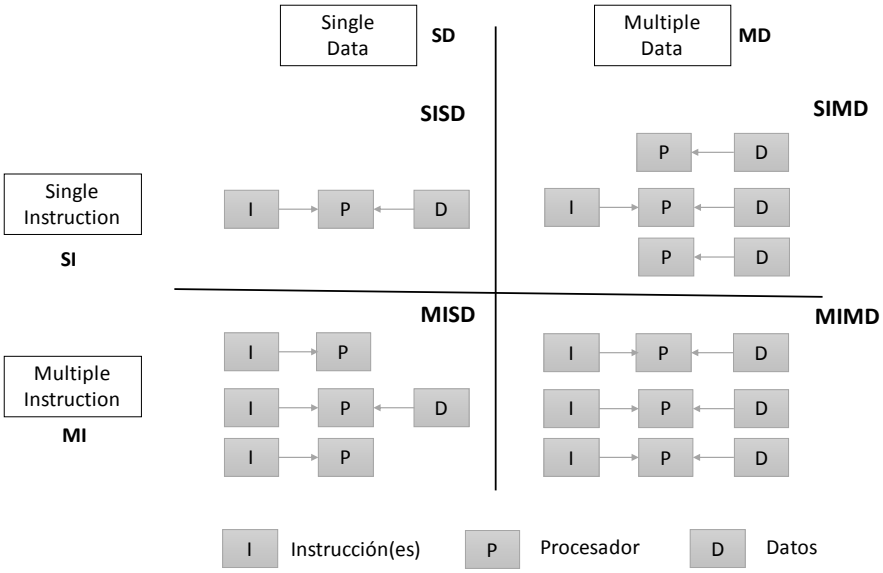

Figura 2: Taxonomía de Flynn.

[1] http://www.lighterra.com/papers/modernmicroprocessors/

[2] Ver `https://computing.llnl.gov/tutorials/parallel_comp/parallelClassifications.pdf`

SISD: Single Instruction, Single Data En esta clase de computadores hay un único flujo de instrucciones, que se ejecutan sobre un único conjunto de datos. Los computadores secuenciales que siguen la arquitectura Von Neumann son de este tipo.

SIMD: Single Instruction, Multiple Data. Un flujo único de instrucciones se ejecuta sobre diferentes conjuntos de datos. Es *single instruction* porque todas las unidades ejecutan la misma instrucción por ciclo de reloj. Es *multiple data* porque cada unidad de procesamiento opera sobre un dato diferente, al mismo tiempo. Se puede considerar que un sistema SIMD tiene un unidad de control pero varias unidades de proceso. Eso permite la paralelización de determinados tipos de algoritmos. Por ejemplo, supongamos que queremos sumar dos vectores númericos:

```
1 for(i = 0; i < n; i++)
2   x[i] = x[i] + y[i]
```

Si disponemos de, por ejemplo, 4 unidades de procesamiento podemos dividir el trabajo del bucle en 4 partes y que cada unidad se encargue de sumar esa parte de los vectores. De esta forma se estaría aplicando una estrategia de *paralelismo de datos* puesto que una única instruccion se ejecuta simultaneamente sobre multiples de datos.

Por tanto, el modelo SIMD es adecuado para problemas con una gran regularidad, como el procesamiento de imágenes. Los procesadores vectoriales disponen de unidades vectoriales que pueden trabajar con registros vectoriales. Estos registros en lugar de almacenar un único valor (un escalar) almacenan un vector de tamaño fijo [7, p. 30–32]. Las CPUs actuales habitualmente ofrecen instrucciones específicas de tipo vectorial, denominadas SIMD. Las GPUs también siguen este diseño a nivel de cada multiprocesador.

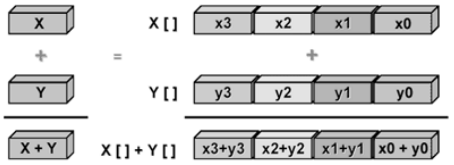

Figura 3: Suma de vectores con el modelo SIMD

MISD: Multiple Instruction, Single Data. En esta configuración se aplican varias instrucciones a los mismos datos. En principio, este tipo de máquinas no tienen un gran interés comercial. Quizás, su principal aplicación podría ser la construcción de computadores tolerantes a fallos: varias CPUs ejecutan varias versiones del mismo programa sobre los mismos datos y el resultado final se puede obtener por votación entre las CPUs.

MIMD: Multiple Instruction, Multiple Data. Es el tipo de organización más versatil. Corresponde a un computador con varias unidades de procesamiento y cada una de ellas puede ejecutar su propia secuencia de instrucciones. Los supercomputadores (HPC), los networked parallel computers (*grids*) y los procesadores multi-core son ejemplos de esta clase. Las GPUs también siguen este modelo.

Durante el curso nos centraremos principalmente en este tipo de organización.

3. Arquitecturas paralelas

Distinguiremos dos tipos básicos de arquitecturas: memoria compartida y memoria distribuida. En estas arquitecturas el computador paralelo estará formado por varios procesadores, por los que se denominarán *multiprocesadores con memoria compartida* y *multiprocesadores con memoria distribuida*.

3.1. Arquitectura de memoria compartida

En el modelo de memoria compartida todos los procesadores tienen acceso a una memoria común, o siendo más precisos a un espacio de direcciones común. Así, los procesos o hilos de un programa paralelo "ven" los mismos datos de memoria. La Figura 4 muestra un esquema básico de este tipo de organización.

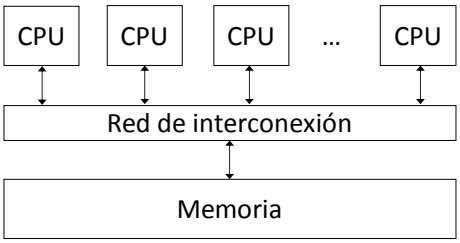

Figura 4: Esquema básico de un multiprocesador con memoria compartida.

En este modelo la comunicación entre los procesos se realiza utilizando este espacio de direcciones común, a través de lecturas y escrituras. Un proceso escribirá en una cierta dirección de memoria, y el resto de procesos podrán leer el contenido de esa dirección. Como veremos, es fundamental coordinar adecuadamente los accesos a memoria para que los hilos puedan realizar su trabjo correctamente.

Hay dos tipos fundamentales de máquinas de memoria compartida, dependiendo cómo es el acceso a memoria.

Uniform memory access (UMA). Todos los accesos a memoria se realizan a la misma velocidad. Es el caso los procesadores multi-núcleo, como el Intel i7. Son multiprocesadores construidos en un solo chip, pero el sistema operativo percibe cada núcleo como un procesador diferente.

Non-Uniform memory access (NUMA). Las máquinas con más procesadores y más memoria no acceden a la misma velocidad a todas las localizaciones de memoria. En los multiprocesadores NUMA cada procesador tiene su propia memoria local y estos accesos son rápidos, pero también puede acceder a la memoria del resto de procesadores pero estos accesos son más lentos.

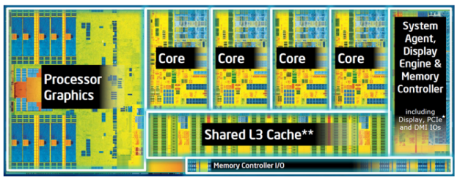

Figura 5: Esquema del Intel i7.

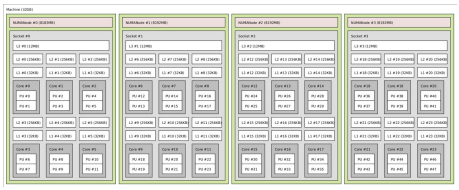

Figura 6: Esquema del multiprocesador Saturno del cluster Heterosolar obtenido con `hwloc`

4. Modelos de programación paralela

Existen diferentes modelos de programación paralela para escribir programas orientados a arquitecturas paralelas. Dependiendo de la arquitectura puede ser más conveniente utilizar un modelo de programación u otro.

Una forma básica de programación paralela es el **IPC (Inter-Process communication)**. Consiste en utilizar procesos del sistema operativo para dividir el trabajo de una aplicación, y utilizar los mecanismos de comunicación y sincronización ofrecidos por éste: memoria compartida, ficheros compartidos, mutexes, pipes, etc. Más información en [1, 2].

Un inconveniente del IPC es que los procesos son costosos de crear y su programación es generalmente de muy bajo nivel. La programación basada en **multi-hilo** aborda este problema utilizando el concepto de hilo. Un mismo proceso puede tener más de una secuencia de instrucciones ejecutándose al mismo tiempo. Cada una de estas líneas de ejecución independientes se denomina hilo (*thread*). Un hilo comparte los recursos del proceso padre (ej., toda la memoria es compartida) y es menos costoso de crear.

Una alternativa es utilizar el paradigma de **paso de mensajes**, en el que los procesos (aquí el término proceso es diferente al de proceso del sistema operativo) no pueden acceder a la memoria de otro proceso y la comunicación se produce enviando y recibiendo mensajes entre ellos.

Es importante tener en cuenta que un modelo de programación es una abstracción sobre las arquitecturas hardware y de memoria. Por ejemplo, es posible realizar programación con hilos sobre un computador basado en paso de mensajes o al revés, realizar programación basada en paso de mensajes en una arquitectura de memoria compartida.

Existen muchos más modelos de programación paralela (ej., Linda [3]) y variaciones de los anteriores. A continuación se presentarán brevemente las ideas básicas de la programación multi-hilo y del paso de mensajes.

4.1. Programación multi-hilo

En el modelo de programación multi-hilo un proceso puede tener varios hilos de ejecución simultáneos. Un hilo tiene sus propios datos locales (p.ej., su propia pila de llamadas), pero comparte todos los recursos con el resto de hilos del programa. Esto hace que haya relativamente poca sobrecarga a la hora de crear un hilo hijo, porque no hay que replicar casi ningún recurso. Además, todos los

hilos comparte el mismo espacio de memoria, de manera que si dos punteros de dos hilos diferentes apuntan a la misma dirección, acceden al mismo dato. La Fig. 7 muestra cómo se organizan los datos en un hilo.

Los hilos se comunican unos con otros a través de la memoria global (la memoria montón normalmente), actualizando y leyendo la misma dirección de memoria. Esto requiere asegurar que dos hilos no actualizan la misma dirección al mismo tiempo.

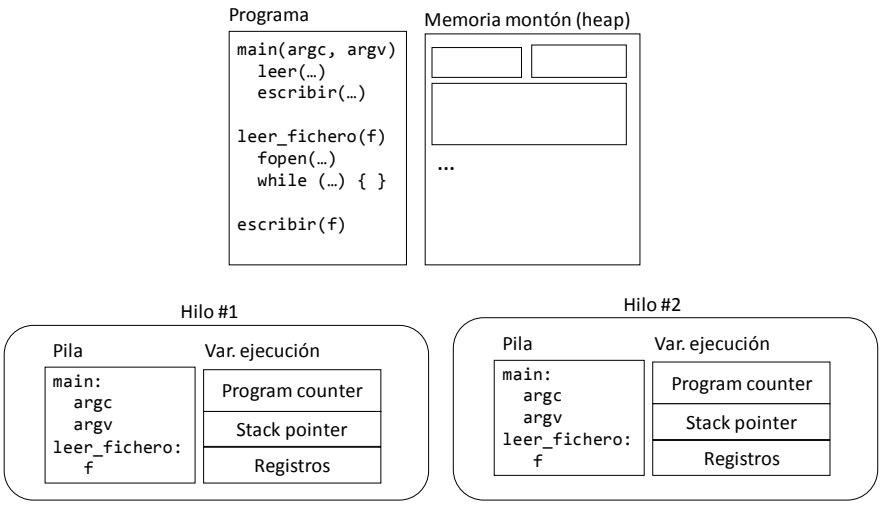

Figura 7: Esquema de los datos privados y compartidos de un hilo.

Pila y memoria *heap*

Las variables locales de una función se guardan en la pila de llamadas. Cada vez que se invoca a una función se crea un nuevo *stack frame* que contiene los parámetros y variables de esa función. Al crear memoria dinámica (ej., `malloc` en C, `new` en Java) se reserva un trozo de la memoria montón (*heap*) que es compartida por todo el programa.

4.2. Breve introducción a la programación multi-hilo en Java

Supongamos que tenemos una colección de ficheros de texto (documentos) y queremos contar cuántas palabras hay en total. La Fig. 8 muestra cómo se organiza el trabajo para realizar el procesamiento secuencial simplemente poniendo todos los documentos en una lista, invocando uno tras otro una función que cuente las palabras del documento (`words` en este caso) y acumulando el resultado en una variable (`t` en este caso). Podemos escribir fácilmente un programa secuencial en Java como el siguiente.

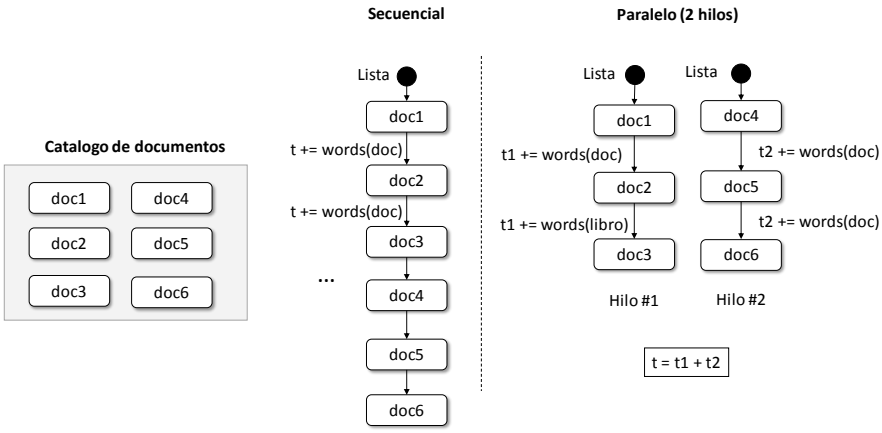

Figura 8: Organización del trabajo para contar palabras de documentos en versión secuencial o paralelo con dos hilos.

java/mpp.countwords

Ver Sequential.java

```
1 public static void main(String[] args) throws IOException {
```

```
  List<File> files = Utils.getFiles(args[0]);
  int count = 0;

  for (File file : files) {
    count += Utils.countWords(file);
  }

  System.out.println("Total palabras: " + count + " en " + files.size() + " ficheros");
}

public static int countWords(File f) throws IOException {
  int words = 0;

  try(BufferedReader reader = new BufferedReader(new FileReader(f))) {
  String line;
    while ((line = reader.readLine()) != null) {
      words += line.split("\\s").length;
    }
  }
  return words;
}
```

En un procesador i7-8550U CPU ejecutar este programa para 55 ficheros, con un tamaño total de 169 MB tarda entre 5 y 6 segundos. ¿Cómo podemos mejorar el rendimiento de este programa? Dado que el el conteo de las palabras de cada fichero es totalmente independiente uno de otros, sería posible realizarlo en paralelo utilizando una estrategia como la Fig. 8: la lista original se divide en dos y cada lista es procesada por un hilo diferente. Es importante observar cómo cada hilo acumula los resultados parciales en una variable propia (`t1` y `t2`) y al final de la ejecución se suman para obtener el resultado final.

En Java la programación multi-hilo es explícita, en el sentido de que el programador tiene que crear y gestionar los hilos explícitamente. El siguiente código muestra una implementación multi-hilo del ejemplo. Utiliza un candado para proteger una variable compartida que todos los hilos actualizan cuando han calculado su resultado parcial (esta no es la mejor estrategia, pero ayuda a mostrar la necesidad de los mecanismos de sincronización).

```
public class Parallel_Fixed {

  private List<File> files;

  // Variable compartida por los hilos
  private int count;

  private Lock lock = new ReentrantLock();

  public Parallel_Fixed(List<File> files) {
    this.files = files;
  }

  public void doCount() {
    int numThreads = Runtime.getRuntime().availableProcessors();
    List<List<File>> split = Utils.split(files, numThreads);
    List<Counter> threads = new ArrayList<>();

    for (List<File> files : split) {
      Counter counter = new Counter(files);
      threads.add(counter);
      counter.start();
    }

    for (Counter counter : threads) {
      try {
        counter.join();
      } catch (InterruptedException e) { }
    }

```

Programa ineficiente

Suele ser buena idea escribir los programas de la manera más simple posible para que sean correctos desde el principio. Sin embargo, puede suceder que eso implique que el programa sea ineficiente.

¿Puedes mejorar el método `countWords` para que sea más eficiente? ¿Cómo se consigue más mejora paralelizando o con una implementación eficiente?

```
  }


  public int getCount() {
    return count;
  }

  public static void main(String[] args) {
    List<File> files = Utils.getFiles(args[0]);

    long init = System.currentTimeMillis();

    Parallel_Fixed counter = new Parallel_Fixed(files);
    counter.doCount();
    int count = counter.getCount();

    long end = System.currentTimeMillis();

    double time = (end - init) / (1_000.0);

    System.out.println("Total palabras: " + count + " en " + files.size() + " ficheros");
    System.out.println(String.format("%.2f", time));
  }

  public class Counter extends Thread {

    private List<File> files;

    public Counter(List<File> files) {
      this.files = files;
    }

    @Override
    public void run() {
      int localCount = 0;
      for (File file : files) {
        try {
          localCount += Utils.countWords(file);
        } catch (IOException e) {
          e.printStackTrace();
        }
      }

      lock.lock();
      count += localCount;
      lock.unlock();
    }

  }
}
```

4.3. Paso de mensajes

El paradigma de paso mensaje se basa en la idea de dividir la ejecución de un programa en dos o más procesos. Cada proceso tiene su propia memoria local y el resto de los procesos no pueden acceder a ella directamente. Los procesos se coordinan e intercambian datos enviando y recibiendo mensajes (de ahí la denominación paso de mensajes)

Los mensajes que se envían los procesos pueden ser:

- Punto a punto. Este tipo de mensajes tienen un único emisor y un único receptor.
- Globales. En este caso el emisor envía un mensaje que es recibido por todos los procesos del sistema (o un subconjunto de ellos).

Y también, según el estilo de sincronización:

HTTP

El protocolo HTTP es un ejemplo del paradigma de paso de mensajes. Los tipos de mensajes está predefinidos (`GET`, `POST`, etc.), y la comunicación es punto a punto y asíncrona.

- Síncronos. Los procesos que intervienen en la comunicación se bloquean el enviar y recibir. El proceso emisor no puede continuar su trabajo hasta que el receptor haya recibido los datos (y posiblemente devuelva una respuesta). El proceso receptor está bloqueado esperando recibir mensajes.
- Asíncronos. Los procesos no se bloquean. El proceso emisor puede seguir trabajando tras el envío y el que recibe sólo se bloquea para procesarlos una vez los recibe (es decir, no está parado esperando que le lleguen los datos).

El paradigma es flexible, y un proceso puede ejecutarse en un único procesador, o bien un mismo procesador puede ejecutar varios procesos en el mismo procesador (por ejemplo, utilizando internamente hilos del sistema operativo). Por tanto, aunque el paradigma de paso de mensajes está particularmente adaptado a los sistemas de memoria distribuida, puede utilizarse en otras arquitecturas.

4.4. Ejemplo

La Fig. 9 muestra cómo se resolvería el ejemplo anterior acerca del conteo de palabras en un catálogo de documentos. Un proceso maestro se encargaría de leer los datos y distribuirlos entre los procesos esclavos (`Proceso #1` y `Proceso #2`). Las primitivas de comunicación usadas en el ejemplo son `enviar` y `recibir`, ambas punto a punto y síncronas.

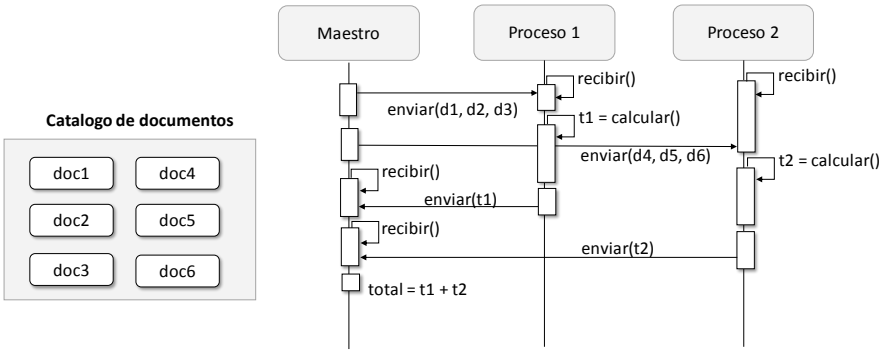

Figura 9: Cálculo del número de palabras en un catálogo de documentos con paso de mensajes. Ilustrado usando notación de diagramas de secuencia.

Un aspecto importante es que todos los cálculos se hacen utilizando los datos de la memoria local de cada proceso. Por tanto, no es necesario ninguna primitiva para asegurar la coherencia de las lecturas/escrituras a memoria.

4.4.1. Librerías y lenguajes de paso de mensajes

Existen diversas implementaciones de este paradigma. Algunas implementaciones son en forma de un lenguaje que tiene construcciones específicas para el envío y la recepción de mensajes (por ejemplo, Erlang) y en otros casos se hace a través de librerías para un lenguaje concreto.

En este curso se utilizará MPI (Message Passing Interface) para construir programas con paso de mensajes en el lenguaje C. MPI es un estándar que especifica una API de programación. Esta API está disponible en varios lenguajes como C/C++ y Fortran, y existen diversas implementaciones, algunas gratuitas: MPICH, LAMMPI, OpenMPI.

Ejercicios

- Encuentra y describe brevemente aplicaciones concretas de la programación paralela. ¿Cuál es la aplicación? ¿Qué estrategia se sigue? ¿Qué aporta la programación paralela? Puedes partir de la lista dada en la sección 1.2 u otras aplicaciones.
- Investiga cómo se pueden ejecutar en paralelo programas con el shell de Linux. Por ejemplo, `&`, `sleep`, `parallel`.

Referencias

[1] Inter-process communication in linux: Shared storage. `https://opensource.com/article/19/4/interprocess-communication-linux-storage`.

[2] Inter-process communication in linux: Using pipes and message queues. `https://opensource.com/article/19/4/interprocess-communication-linux-channels`.

[3] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *Computer*, (8):26–34, 1986.

[4] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[5] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[6] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[7] P. Pacheco. *An introduction to parallel programming*. Elsevier, 2011.