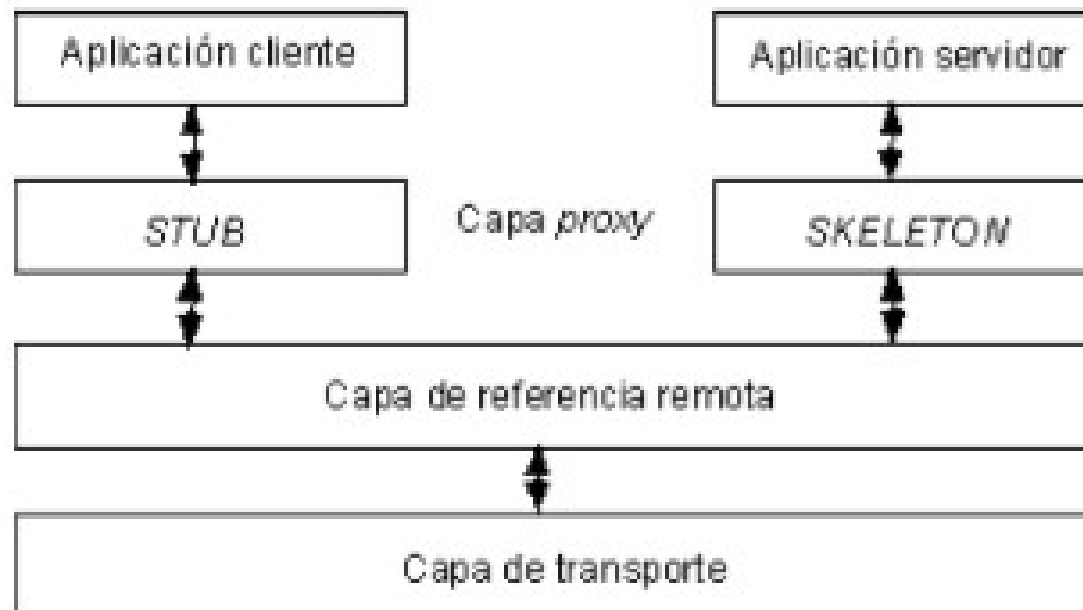


Programación con RMI


RMI (**Remote Method Invocation**) es un mecanismo que permite realizar llamadas a métodos de objetos remotos situados en distintas (o la misma) máquinas virtuales de Java, compartiendo así recursos y carga de procesamiento a través de varios sistemas.



Programación con RMI

Primera capa

La primera capa es la de aplicación y se corresponde con la **implementación real de las aplicaciones cliente y servidor**. Aquí tienen lugar las llamadas a alto nivel para acceder y exportar objetos remotos. Cualquier aplicación que quiera que sus **métodos estén disponibles para su acceso por clientes remotos debe declarar dichos métodos en una interfaz que extienda `java.rmi.Remote`**. Dicha interfaz se usa básicamente para “marcar” un objeto como remotamente accesible. Una vez que los métodos han sido implementados, el objeto debe ser exportado. Esto puede hacerse de forma implícita si el objeto extiende la clase `UnicastRemoteObject` (paquete `java.rmi.server`), o puede hacerse de forma explícita con una llamada al método `exportObject()` del mismo paquete.



Programación con RMI

Segunda capa

Es la capa proxy, o capa stub-skeleton. Esta capa es la que interactúa directamente con la capa de aplicación. Todas las llamadas a objetos remotos y acciones junto con sus parámetros y retorno de objetos tienen lugar en esta capa.

Tercera Capa

Es la de referencia remota, y es responsable del manejo de la parte semántica de las invocaciones remotas. También es responsable de la gestión de la replicación de objetos y realización de tareas específicas de la implementación con los objetos remotos, como el establecimiento de las persistencias semánticas y estrategias adecuadas para la recuperación de conexiones perdidas. En esta capa se espera una conexión de tipo stream (stream-oriented connection) desde la capa de transporte.



Programación con RMI

Cuarta capa

Es la de transporte. Es la responsable de realizar las conexiones necesarias y manejo del transporte de los datos de una máquina a otra. El protocolo de transporte subyacente para RMI es JRMP (Java Remote Method Protocol), que solamente es “comprendido” por programas Java.



Programación con RMI

Toda aplicación RMI normalmente se descompone en 2 partes:

- Un **servidor**, que crea algunos objetos remotos, crea referencias para hacerlos accesibles, y espera a que el cliente los invoque.
- Un **cliente**, que obtiene una referencia a objetos remotos en el servidor, y los invoca.



Servidor RMI

Creación de un servidor RMI

Un servidor RMI consiste en definir un objeto remoto que va a ser utilizado por los clientes. Para crear un objeto remoto, se define una interfaz, y el objeto remoto será una clase que implemente dicha interfaz.

1. Definir la interfaz remota
2. Implementar la interfaz remota
3. Compilar y ejecutar el servidor



Servidor RMI

1. Definir la interfaz remota

- El interfaz debe ser público.
- Debe extender (heredar de) el interfaz `java.rmi.Remote`, para indicar que puede llamarse desde cualquier máquina virtual Java.
- Cada método remoto debe lanzar la excepción `java.rmi.RemoteException` en su clausula `throws`, además de las excepciones que pueda manejar.

```
public interface MiInterfazRemota extends java.rmi.Remote {  
    public void miMetodo1() throws java.rmi.RemoteException;  
    public int miMetodo2() throws java.rmi.RemoteException;  
}
```



Servidor RMI

2. Implementar la interfaz remota

```
public class MiClaseRemota extends
    java.rmi.server.UnicastRemoteObject implements
    MiInterfazRemota {

    public MiClaseRemota() throws java.rmi.RemoteException {
        // Código del constructor
    }

    public void miMetodo1() throws java.rmi.RemoteException {
        // Aquí ponemos el código que queramos
        System.out.println("Estoy en miMetodo1()");
    }
}
```



Servidor RMI

2. Implementar la interfaz remota (cont...)

```
public int miMetodo2() throws java.rmi.RemoteException {  
    // Aquí ponemos el código que queremos  
    Return 5;  
}  
  
public void otroMetodo() {  
    //  
    //  
}
```



Servidor RMI

2. Implementar la interfaz remota (cont...)

```
public static void main(String[] args) {  
    try {  
        MiInterfazRemota mir = new MiClaseRemota();  
  
        java.rmi.Naming.rebind("//" +  
            java.net.InetAddress.getLocalHost().getHostAddress() +  
                ":" + args[0] + "/PruebaRMI", mir);  
  
    } catch (Exception e) {  
    }  
}
```



Servidor RMI

3. Compilar y ejecutar el servidor

```
# javac MiInterfazRemota.java
```

```
# javac MiClaseRemota.java
```

```
# rmic -d . MiClaseRemota
```

```
# rmiregistry 1234
```

```
# java MiClaseRemota 1234
```



Cliente RMI

Creación del cliente RMI


1. Definir la clase para obtener los objetos remotos necesarios
2. Compilar y ejecutar el cliente



Cliente RMI

1. Definir la clase para obtener los objetos remotos necesarios

```
public class MiClienteRMI {  
    public static void main(String[] args) {  
        try {  
  
            MiInterfazRemota mir =  
                (MiInterfazRemota) java.rmi.Naming.lookup("//" +  
                    args[0] + ":" + args[1] + "/PruebaRMI");  
  
            // Imprimimos miMetodo1() tantas veces  
            // como devuelva miMetodo2()  
            for (int i = 1; i <= mir.miMetodo2(); i++)  
                mir.miMetodo1();  
  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```



Cliente RMI

2. Compilar y ejecutar el cliente

```
// Los archivos generados *_Stub.class deben estar accesibles  
// en el cliente...
```

```
# javac MiClienteRMI.java
```

```
# java MiClienteRMI 127.0.0.1 1234
```



Activación automática de servicios RMI

En la forma anterior se usa RMI para trabajar con objetos remotos registrando el objeto servidor mediante el servicio de nombres RMI desde una máquina virtual de Java en ejecución, de forma que si la máquina virtual Java terminase, el objeto dejaría de existir y provocaría una excepción al invocarlo.

Pero también se puede utilizar el servicio de activación de objetos directamente desde el cliente de forma automática (se crea el objeto servidor desde una máquina virtual existente o nueva), registrando un método de activación a través del demonio de RMI del host. **Esto nos va a permitir optimizar recursos**, ya que el objeto servidor sólo se crea cuando se necesita.



Servidor automático RMI

1. Definir la interfaz remota

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
public interface MiInterfazRemota extends Remote {  
    public void miMetodo1() throws RemoteException;  
    public int miMetodo2() throws RemoteException;  
}
```



Servidor automático RMI

2. Implementar la interfaz remota

```
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;
import java.net.InetAddress;

public class MiClaseRemota2 extends Activatable implements
    MiInterfazRemota {

    public MiClaseRemota2(ActivationID a, MarshaledObject m)
        throws RemoteException {
        super(a, 0);
    }

    public void miMetodo1() throws RemoteException {
        ... // Código
    }

    public int miMetodo2() throws RemoteException {
        ... // Código
    }
}
```



Servidor automático RMI

2. Implementar la interfaz remota (cont...)

```
public static void main(String[] args) throws Exception {  
  
    try {  
        // Se establece el archivo de política de seguridad  
        Properties p = new Properties();  
        p.put("java.security.policy", "../policy");  
  
        // Necesitamos un grupo de activación para activar objetos  
        // remotos y los puedan acceder los clientes  
        ActivationGroupDesc.CommandEnvironment ace = null;  
        ActivationGroupDesc ejemplo = new  
            ActivationGroupDesc(p, ace);  
  
        // Se registra el grupo creado con el ActivationGroupDesc,  
        // y se obtiene el identificador del registro  
        ActivationGroupID agi =  
            ActivationGroup.getSystem().registerGroup(ejemplo);  
    }  
}
```



Servidor automático RMI

2. Implementar la interfaz remota (cont...)

```
// Objeto para indicar datos de inicialización,  
// si se requieren  
MarshaledObject m = null;  
  
// Descripción y registro del objeto en el demonio rmid  
ActivationDesc desc = new ActivationDesc(agi,  
    "MiClaseRemota2", "file://C:/rmi/servidor2/", m);  
MiInterfazRemoto mir =  
    (MiInterfazRemota)Activatable.register(desc);  
  
Naming.rebind("//" +  
    InetAddress.getLocalHost().getHostAddress() +  
    ":" + args[0] + "/PruebaRMI", mir);  
} catch (Exception e) { e.printStackTrace(); }  
System.exit(0);  
}  
}
```



Servidor automático RMI

3. Compilar y ejecutar el servidor

```
# javac MiInterfazRemota.java
```

```
# javac MiClaseRemota2.java
```

```
// A partir de la versión 1.5 este comando se puede omitir el uso  
// de la herramienta rmic para generar la capa Stub-Skeleton
```

```
# rmic -d . MiClaseRemota2
```

```
# rmiregistry 1234
```

```
# rmid -J-Djava.security.policy=policy
```

```
# java -Djava.security.policy=policy MiClaseRemota2 1234
```



Servidor automático RMI

3. Compilar y ejecutar el servidor (cont...)

Archivo “policy”

```
grant {  
    permission java.security.AllPermission;  
    permission java.net.SocketPermission "127.0.0.1:1234",  
        "connect,resolve";  
    permission java.lang.RuntimePermission "setFactory";  
};
```



Cliente RMI

1. Definir la clase para obtener los objetos remotos

```
public class MiClienteRMI2 {  
    public static void main(String[] args) {  
        try {  
            MiInterfazRemota mir =  
                (MiInterfazRemota) java.rmi.Naming.lookup("//" +  
                    args[0] + "/PruebaRMI");  
  
            mir.miMetodo1(args[1]);  
            int i = mir.miMetodo2();  
            System.out.println ("Valor de miMetodo2: " + i);  
  
        } catch (Exception e) {  
            System.out.println ("Error, no encuentro: " +  
                e.getMessage());  
        }  
    }  
}
```



Cliente RMI

2. Compilar y ejecutar el cliente

```
# javac MiClienteRMI2.java
```

```
# java MiClienteRMI2 127.0.0.1:1234
```



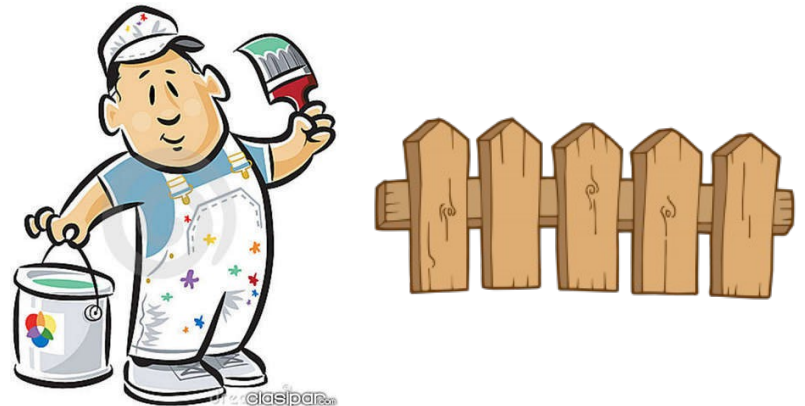
Aceleración y eficiencia

Aceleración (simple)

Objetivo, medir que tanto se acelera la ejecución de cómputo contra el mejor código serial (tiempo serial dividido por el tiempo paralelo).

Ejemplo: pintar una cerca de tablitas

- 30 minutos preparación (serial)
- 1 minuto para pintar una tabla
- 30 minutos para limpiar (serial)



Por lo tanto, 300 tablas toman 360 minutos (tiempo serial)



Aceleración y eficiencia

Calculando la aceleración

Numero de pintores	Tiempo	Speedup
1	$30 + 300 + 30 = 360$	1.0X
2	$30 + 150 + 30 = 210$	1.7X
10	$30 + 30 + 30 = 90$	4.0X
100	$30 + 3 + 30 = 63$	5.7X
Infinito	$30 + 0 + 30 = 60$	6.0X

Ley de Amdahl: La aceleración potencial está restringida por la porción serial.

Que pasaría si el dueño de la barda usa un spray para pintar 300 tablas en una hora?, mejor algoritmo serial?, cual es la máxima paralelización?



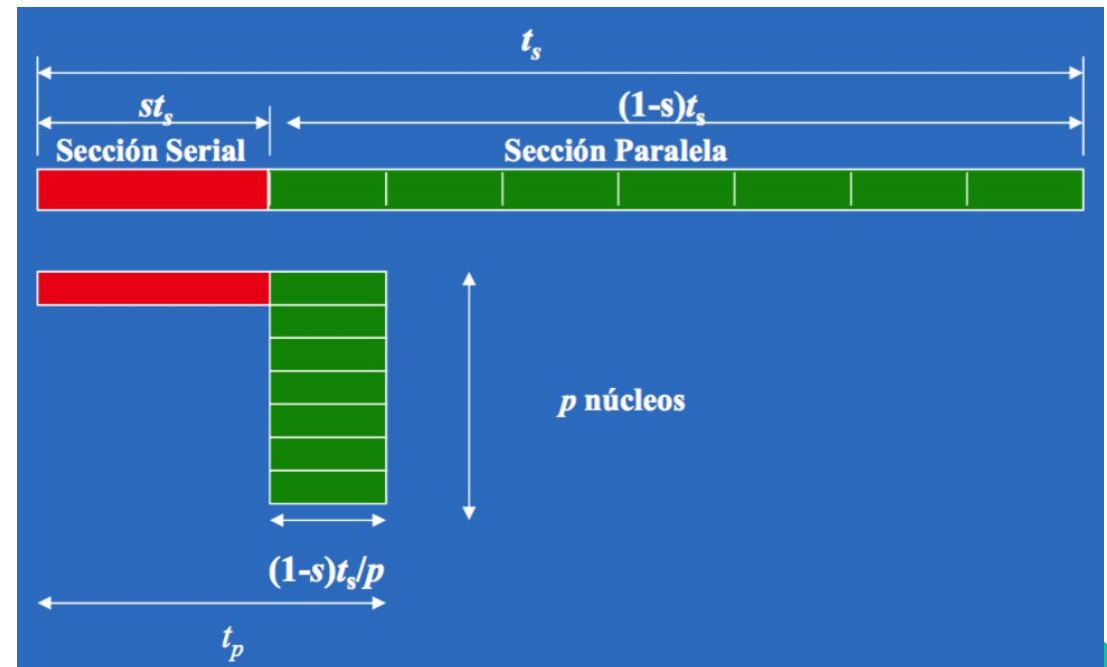
Aceleración y eficiencia

Aceleración

El acuerdo más importante es que la concurrencia está limitada por la naturaleza de la aplicación.

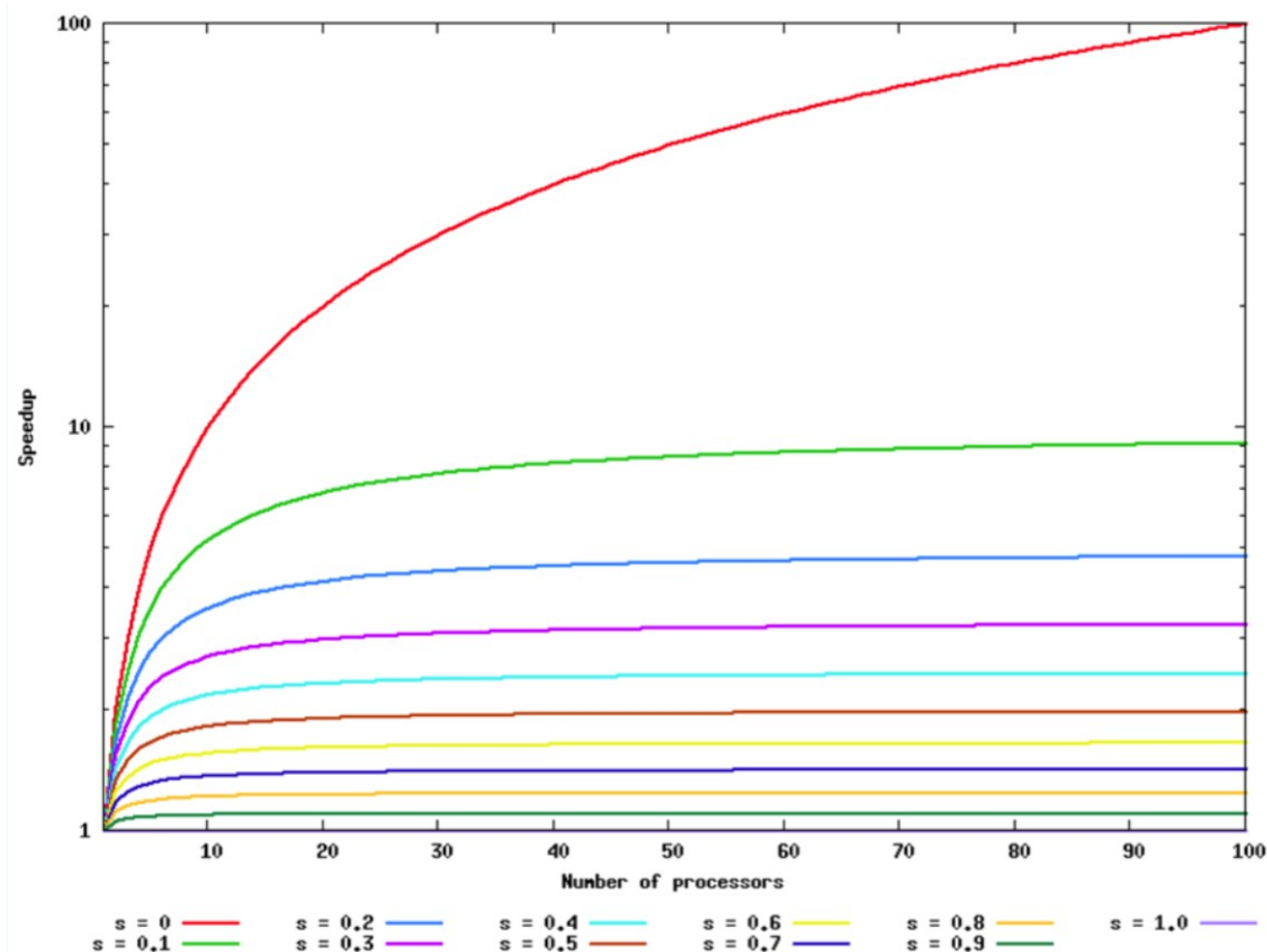
- Si s es la fracción de trabajo serial intrínseco
- El tiempo de computación usando p núcleos

$$T_p = st_s + \frac{(1-s)t_s}{p}$$



Aceleración y eficiencia

Ley de Amdahl



Aceleración y eficiencia

Eficiencia

Medir que tan efectivamente los recursos de cómputo están ocupados

- Aceleración dividida entre el número de hilos
- Expresada como porcentaje promedio de tiempo no ocioso

Numero de pintores	Tiempo	Aceleración	Eficiencia
1	360	1.0X	100%
2	$30 + 150 + 30 = 210$	1.7X	85%
10	$30 + 30 + 30 = 90$	4.0X	40%
100	$30 + 3 + 30 = 63$	5.7X	5.7%
Infinito	$30 + 0 + 30 = 60$	6.0X	Muy baja



Granularidad

Puede definirse como la cantidad de procesamiento que una tarea realiza antes de necesitar comunicarse con otra tarea. Por supuesto, un grano puede crecer o decrecer agrupando o desagrupando tareas. Esto establece una relación entre el tiempo de cálculo con respecto al número de eventos de comunicación (granularidad de la aplicación). Así, la granularidad, es la relación entre la computación y la comunicación en lo interno de una aplicación.

La granularidad en el paralelismo de un sistema se encuentra en determinar que tanto se puede paralelizar código de procesos, rutinas, módulos o bucles a nivel de instrucción. **El término granularidad se usa como el mínimo componente del sistema que puede ser preparado para ejecutarse de manera paralela.**



Granularidad

Granularidad fina (pequeña)

Es la granularidad más pequeña y basa prácticamente todo su funcionamiento en propiedades del hardware. El hardware puede ser suficientemente inteligente para que el programador no tenga que hacer mucho por soportar esta granularidad; Implica más comunicación y cambios de contextos entre las tareas (es una aplicación con comunicación intensiva).

Granularidad media

El paralelismo de grano medio en general es explotado por el programador o el compilador. El hardware normalmente también se prepara para poder aprovechar este tipo de paralelismo, por ejemplo, los procesadores pueden disponer de instrucciones especiales para ayudar en el cambio de una tarea a otra que realiza el sistema operativo.



Granularidad

Granularidad gruesa (grande)

El paralelismo de grano grueso es el que explota el programador mediante programas que no tienen por qué requerir la utilización de ninguna librería externa o hardware especializado, sino solamente el uso de conocimientos de programación para paralelizar un algoritmo.

Se basa principalmente en cualquier tipo de medio que utilice el programador para crear un programa, que solucione un problema de manera paralela, sin tener por que hacer uso más que de su habilidad de programador y de un buen algoritmo. Son los más limitados al carecer de métodos específicos para comunicación entre nodos o procesadores.

Implica menos comunicación, pero puede que potenciales tareas concurrentes queden agrupadas, y por consiguiente, ejecutadas secuencialmente



Granularidad

Grano ideal...

La determinación del grano ideal, para una aplicación dada, es importante, para definir la mejor relación entre paralelismo (grano pequeño) y comunicación (grano grande).

El problema de determinación del tamaño óptimo del grano para una aplicación dada es un problema de optimización del tipo MaxMin (Maximizar paralelismo y Minimizar comunicación).



Granularidad

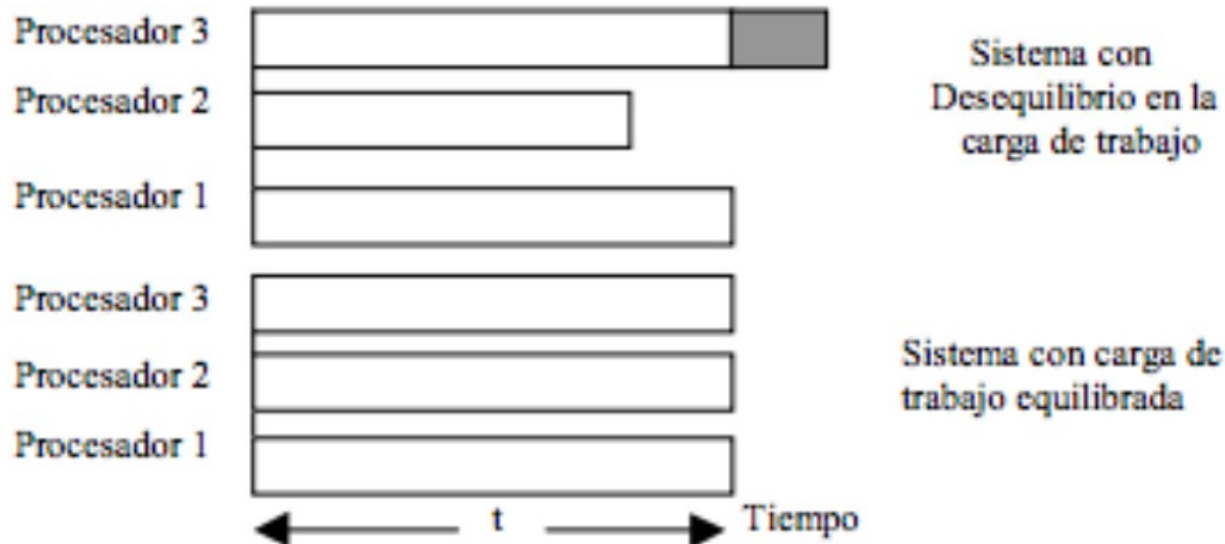
Grano ideal...

Tipo de código	Granularidad	Modo Paralelo
Programas	Gruesa	Concurrente
Rutinas	Media - Fuerte	Concurrente
Instrucciones	Media - Fina	Concurrente - Paralelo
Operadores y Bits	Fina	Paralelo



Balanceo de carga

En el balanceo de carga busca mantener los procesos/procesadores ocupados todo el tiempo, para que ellos finalicen sus ejecuciones, aproximadamente al mismo tiempo. Ciclos de procesadores pueden ser perdidos si algunos nodos deben esperar a que terminen otros. Esta técnica mitiga el efecto de diferencias a nivel de las velocidades de los procesadores.



Técnicas de balanceo de carga

Modelos de presión

En este caso los trabajos son tratados como un fluido que fluye a través de la arquitectura. Un computador tiene una presión interna (trabajos a realizar localmente) y externa (trabajos que hacen otros computadores). El equilibrio de la carga de trabajo consiste en dirigir los trabajos a lo largo de la red a áreas que corresponden a bajas presiones (grupos de computadores con poca carga). En este caso no se requiere información global, un computador sólo debe tener información de sus vecinos y alguna indicación de su propia carga. Cuando un computador alcanza cierto valor crítico, entonces busca cuáles de sus vecinos tienen carga baja para transferirles trabajo. En el tiempo, los trabajo tienden a propagarse a áreas con baja carga. Es parecido a la técnica de Difusión, pero aplicada repetidamente.



Técnicas de balanceo de carga

Algoritmo Round Robin

Consiste en colocar secuencialmente una tarea sobre cada procesador y regresar al primero una vez recorrido todos.

Bisección recursiva

Recursivamente se divide el problema (carga) en subproblemas con más o menos iguales esfuerzos computacionales.

