```
Maria Jesús

 El volumen de la unidad D es Data
 El n£mero de serie del volumen es: 9013-7A66

 Directorio de D:\CLASE\PSP_2018\CAPITULO1

14/06/2018  00:14    <DIR>          .
14/06/2018  00:14    <DIR>          ..
14/06/2018  00:14               396 .classpath
14/06/2018  00:14               385 .project
14/06/2018  00:14    <DIR>          .settings
14/06/2018  00:15    <DIR>          bin
14/06/2018  00:15    <DIR>          src
               2 archivos            781 bytes
               5 dirs  134.146.215.936 bytes libres
```

REDIRECCIONANDO LA ENTRADA Y LA SALIDA

Los métodos **redirectOutput()** y **redirectError()** nos permiten redirigir la salida estándar y de error a un fichero. El siguiente ejemplo ejecuta el comando DIR y envía la salida al fichero *salida.txt*, si ocurre algún error se envía a *error.txt*:

```
import java.io.File;
import java.io.IOException;

public class Ejemplo7 {
  public static void main(String args[]) throws IOException {
    ProcessBuilder pb = new ProcessBuilder("CMD","/C" ,"DIR");

    File fOut = new File("salida.txt");
    File fErr = new File("error.txt");

    pb.redirectOutput(fOut);
    pb.redirectError(fErr);
    pb.start();
  }
}// Ejemplo7
```

También podemos ejecutar varios comandos del sistema operativo dentro de un fichero BAT. El siguiente ejemplo ejecuta los comandos MS-DOS que se encuentran en el fichero *fichero.bat*. Se utiliza el método **redirectInput()** para indicar que la entrada al proceso se encuentra en un fichero, es decir la entrada para el comando CMD será el *fichero.bat*. La salida del proceso se envía al fichero *salida.txt* y la salida de error al fichero *error.txt*:

```
import java.io.File;
import java.io.IOException;

public class Ejemplo8 {
  public static void main(String args[]) throws IOException {
    ProcessBuilder pb = new ProcessBuilder("CMD");

    File fBat = new File("fichero.bat");
    File fOut = new File("salida.txt");
    File fErr = new File("error.txt");
```

```
        pb.redirectInput(fBat);
        pb.redirectOutput(fOut);
        pb.redirectError(fErr);
        pb.start();

    }
}// Ejemplo8
```

Suponiendo que los comandos MS-DOS del *fichero.bat* son estos (este fichero se debe crear en el proyecto Eclipse):

```
MKDIR NUEVO
CD NUEVO
ECHO CREO FICHERO > Mifichero.txt
DIR
DIRR
ECHO FIN COMANDOS
```

Donde se crea una carpeta, nos dirigimos a dicha carpeta, se crea el fichero *Mifichero.txt*, se hace un DIR del directorio actual, el siguiente comando DIRR es erróneo y se visualiza FIN COMANDOS. Al ejecutarlo desde el entorno Eclipse el contenido del fichero de salida *salida.txt* es el siguiente:

```
Microsoft Windows [Versión 10.0.17134.48]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

D:\CLASE\PSP_2018\CAPITULO1>MKDIR NUEVO

D:\CLASE\PSP_2018\CAPITULO1>CD NUEVO

D:\CLASE\PSP_2018\CAPITULO1\NUEVO>ECHO CREO FICHERO > Mifichero.txt

D:\CLASE\PSP_2018\CAPITULO1\NUEVO>DIR
 El volumen de la unidad D es Data
 El número de serie del volumen es: 9013-7A66

 Directorio de D:\CLASE\PSP_2018\CAPITULO1\NUEVO

14/06/2018  00:34    <DIR>          .
14/06/2018  00:34    <DIR>          ..
14/06/2018  00:34                15 Mifichero.txt
               1 archivos             15 bytes
               2 dirs  134.146.215.936 bytes libres

D:\CLASE\PSP_2018\CAPITULO1\NUEVO>DIRR

D:\CLASE\PSP_2018\CAPITULO1\NUEVO>ECHO FIN COMANDOS
FIN COMANDOS

D:\CLASE\PSP_2018\CAPITULO1\NUEVO>
```

Y el del fichero de error *error.txt*:

```
"DIRR" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.
```

ACTIVIDAD 1.7

Modifica el *Ejemplo5.java* para que la salida del proceso y la salida de error se almacenen en un fichero de texto, y la entrada la tome desde otro fichero de texto.

Para llevar a cabo el redireccionamiento, tanto de entrada como de salida del proceso que se ejecuta, también podemos usar la clase **ProcessBuilder.Redirect**. El redireccionamiento puede ser uno de los siguientes:

- El valor especial **Redirect.INHERIT**, indica que la la fuente de entrada y salida del proceso será la misma que la del proceso actual.
- **Redirect.from (File)**, indica redirección para leer de un fichero, la entrada al proceso se encuentra en el objeto **File**.
- **Redirect.to(File)**, indica redirección para escribir en un fichero, el proceso escribirá en el objeto **File** especificado.
- **Redirect.appendTo (File)**, indica redirección para añadir a un fichero, la salida del proceso se añadirá al objeto **File** especificado.

El ejemplo anterior usando esta clase quedaría de esta manera:

```
pb.redirectInput(ProcessBuilder.Redirect.from(fBat));
pb.redirectOutput(ProcessBuilder.Redirect.to(fOut));
pb.redirectError(ProcessBuilder.Redirect.to(fErr));
```

El siguiente ejemplo muestra en la consola la salida del comando DIR,

```
import java.io.IOException;
public class Ejemplo9 {
  public static void main(String args[]) throws IOException {
      ProcessBuilder pb = new ProcessBuilder("CMD", "/C", "DIR");
      pb.redirectOutput(ProcessBuilder.Redirect.INHERIT);
      Process p = pb.start();
  }
}// Ejemplo9
```

ACTIVIDAD 1.8

Usando **ProcessBuilder.Redirect**, modifica el *Ejemplo5.java* para que la salida del proceso se muestre en la consola, la entrada la tome desde un fichero de texto, y la salida la lleve a un fichero de texto. Realiza los ejercicios 7, 8 y 9.

1.3. PROGRAMACIÓN CONCURRENTE

El diccionario *WordReference.com* (http://www.wordreference.com/definicion/) nos muestra varias acepciones de la palabra concurrencia. Nos quedamos con la tercera: "*Acaecimiento o concurso de varios sucesos en un mismo tiempo*". Si sustituimos sucesos por procesos ya tenemos una aproximación de lo que es la concurrencia en informática: la existencia simultánea de varios procesos en ejecución.

1.3.1. PROGRAMA Y PROCESO

Al principio del tema se definió un **proceso** como un programa en ejecución. Y ¿qué es un programa?, podemos definir **programa** como un conjunto de instrucciones que se aplican a un conjunto de datos de entrada para obtener una salida. Un proceso es algo activo que cuenta con una serie de recursos asociados, en cambio un programa es algo pasivo, para que pueda hacer algo hay que ejecutarlo.

Pero un programa al ponerse en ejecución puede dar lugar a más de un proceso, cada uno ejecutando una parte del programa. Por ejemplo, el navegador web, por un lado está controlando las acciones del usuario con la interfaz, por otro hace las peticiones al servidor web. Entonces cada vez que se ejecuta este programa crea 2 procesos.

En la Figura 1.12 existe un programa almacenado en disco y 3 instancias del mismo ejecutándose, por ejemplo, por 3 usuarios diferentes. Cada instancia del programa es un proceso, por tanto, existen 3 procesos independientes ejecutándose al mismo tiempo sobre el sistema operativo, tenemos entonces 3 procesos concurrentes.

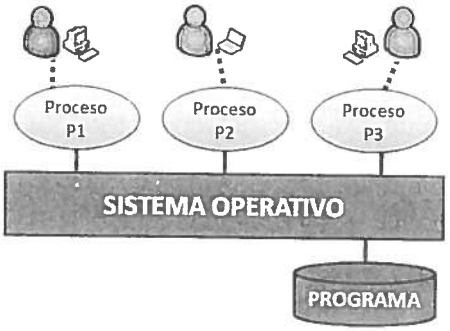

Figura 1.12. Un programa con 3 instancias ejecutándose.

Dos procesos serán concurrentes cuando la primera instrucción de uno de ellos se ejecuta después de la primera instrucción del otro y antes de la última. Es decir, existe un solapamiento o intercalado en la ejecución de sus instrucciones. No hay que confundir el solapamiento con la ejecución simultánea de las instrucciones, en este caso estaríamos en una situación de **programación paralela**, aunque a veces el hardware subyacente (más de un procesador) sí permitirá la ejecución simultánea.

Supongamos ahora que el programa anterior al ejecutarse da lugar a 2 procesos más, cada uno ejecutando una parte del programa, entonces la Figura 1.12 se convierte en la 1.13. Ya que un programa puede estar compuesto por diversos procesos, una definición más acertada de proceso es la de una actividad asíncrona susceptible de ser asignada a un procesador[1].

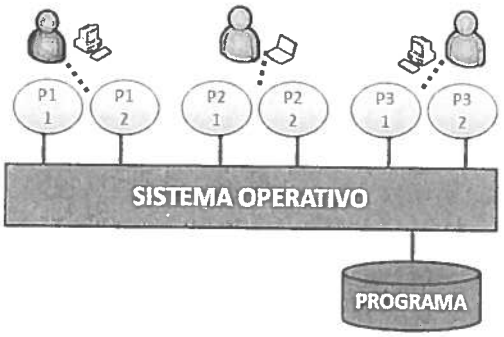

Figura 1.13. Un programa dando lugar a más de un proceso.

Cuando varios procesos se ejecutan concurrentemente puede haber procesos que colaboren para un determinado fin (por ejemplo, P1.1 y P1.2), y otros que compitan por los recursos del sistema (por ejemplo P2.1 y P3.1). Estas tareas de colaboración y competencia por los recursos exigen **mecanismos de comunicación y sincronización entre procesos.**

[1] Programación concurrente. José Tomás Palma Méndez y otros. Ed Paraninfo. ISBN: 9788497321846

1.3.2. CARACTERÍSTICAS

La **programación concurrente** es la disciplina que se encarga del estudio de las notaciones que permiten especificar la ejecución concurrente de las acciones de un programa, así como las técnicas para resolver los problemas inherentes a la ejecución concurrente (comunicación y sincronización).

BENEFICIOS

La programación concurrente aporta una serie de beneficios:

Mejor aprovechamiento de la CPU. Un proceso puede aprovechar ciclos de CPU mientras otro realiza una operación de entrada/salida.

Velocidad de ejecución. Al subdividir un programa en procesos, éstos se pueden "repartir" entre procesadores o gestionar en un único procesador según importancia.

Solución a problemas de naturaleza concurrente. Existen algunos problemas cuya solución es más fácil utilizando esta metodología:

- Sistemas de control: son sistemas en los que hay captura de datos, normalmente a través de sensores, análisis y actuación en función del análisis. Un ejemplo son los sistemas de tiempo real.
- Tecnologías web: los servidores web son capaces de atender múltiples peticiones de usuarios concurrentemente, también los servidores de chat, correo, los propios navegadores web, etc.
- Aplicaciones basadas en GUI: el usuario puede interactuar con la aplicación mientras la aplicación está realizando otra tarea. Por ejemplo, el navegador web puede estar descargando un archivo mientras el usuario navega por las páginas.
- Simulación: programas que modelan sistemas físicos con autonomía.
- Sistemas Gestores de Bases de Datos: Los usuarios interactúan con el sistema, cada usuario puede ser visto como un proceso.

CONCURRENCIA Y HARDWARE

En un sistema **monoprocesador** (de un solo procesador) se puede tener una ejecución concurrente gestionando el tiempo de procesador para cada proceso. El S.O. va alternando el tiempo entre los distintos procesos, cuando uno necesita realizar una operación de entrada salida, lo abandona y otro lo ocupa; de esta forma se aprovechan los ciclos del procesador. En la Figura 1.14 se muestra como el tiempo de procesador es repartido entre 3 procesos, en cada momento sólo hay un proceso. Esta forma de gestionar los procesos en un sistema monoprocesador recibe el nombre de **multiprogramación**.

Figura 1.14. Concurrencia.

En un sistema **monoprocesador** todos los procesos comparten la misma memoria. La forma de comunicar y sincronizar procesos se realiza mediante variables compartidas.

En un sistema **multiprocesador** (existe más de un procesador) podemos tener un proceso en cada procesador. Esto permite que exista paralelismo real entre los procesos, véase Figura 1.15.

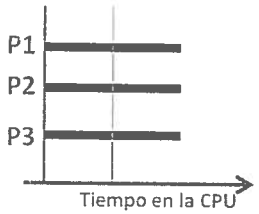

Figura 1.15. Paralelismo.

Estos sistemas se pueden clasificar en:

- Fuertemente acoplados: cuando poseen una memoria compartida por todos los procesadores, véase Figura 1.16.
- Débilmente acoplados: cuando los procesadores poseen memorias locales y no existe la compartición de memoria, véase Figura 1.17.

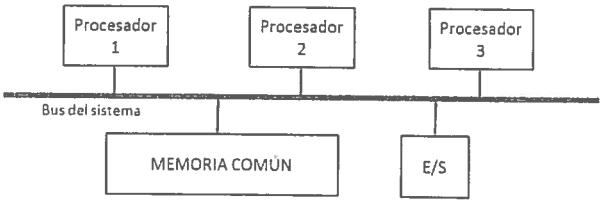

Figura 1.16. Fuertemente acoplados.

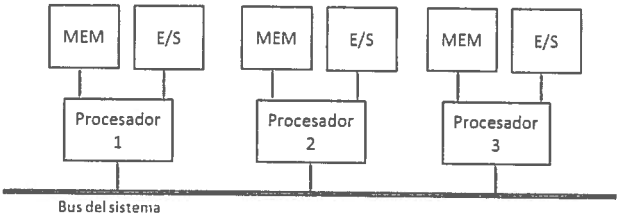

Figura 1.17. Débilmente acoplados.

Se denomina **multiproceso** a la gestión de varios procesos dentro de un sistema multiprocesador, donde cada procesador puede acceder a una memoria común.

1.3.3. PROGRAMAS CONCURRENTES

Un **programa concurrente** define un conjunto de acciones que pueden ser ejecutadas simultáneamente. Supongamos que tenemos estas dos instrucciones en un programa, está claro que el orden de la ejecución de las mismas influirá en el resultado final:

x=x+1; y=x+1;	La primera instrucción se debe ejecutar antes de la segunda.

En cambio, si tenemos estas otras, el orden de ejecución es indiferente:

x=1;
y=2;
z=3;
	El orden no interviene en el resultado final.

CONDICIONES DE BERNSTEIN

Bernstein definió unas condiciones para que dos conjuntos de instrucciones se puedan ejecutar concurrentemente. En primer lugar es necesario formar 2 conjuntos de instrucciones:

- **Conjunto de lectura:** formado por instrucciones que cuentan con variables a las que se accede en modo lectura durante su ejecución.
- **Conjunto de escritura:** formado por instrucciones que cuenta con variables a las que se accede en modo escritura durante su ejecución.

Por ejemplo, sean las siguientes instrucciones:

Instrucción 1:	x := y+1
Instrucción 2;	y := x+2
Instrucción 3:	z := a+b

Los conjuntos de lectura y escritura estarían formados por las variables siguientes:

	Conjunto lectura - L	Conjunto escritura - E
Instrucción 1- I1:	y	x
Instrucción 2- I2:	x	y
Instrucción 3- I3:	a,b	z

Se pueden expresar de la siguiente manera:

L(I1)={y}	E(I1)={x}
L(I2)={x}	E(I2)={y}
L(I3)={a,b}	E(I3)={z}

Para que dos conjuntos se puedan ejecutar concurrentemente se deben cumplir estas 3 condiciones:

- La intersección entre las variables leídas por un conjunto de instrucciones *Ii* y las variables escritas por otro conjunto *Ij* debe ser vacío, es decir, no debe haber variables comunes:

$$L(Ii) \cap E(Ij) = \emptyset$$

- La intersección entre las variables de escritura de un conjunto de instrucciones *Ii* y las variables leídas por otro conjunto *Ij* debe ser nulo, es decir, no debe haber variables comunes:

$$E(Ii) \cap L(Ij) = \emptyset$$

- Por último, la intersección entre las variables de escritura de un conjunto de instrucciones *Ii* y las variables de escritura de un conjunto *Ij* debe ser vacío, no debe haber variables comunes:

$$E(Ii) \cap E(Ij) = \emptyset$$

En el ejemplo anterior tenemos las siguientes condiciones, donde se observa que las instrucciones I1 e I2 no se pueden ejecutar concurrentemente porque no cumplen las 3 condiciones:

Conjunto I1 e I2	Conjunto I2 e I3	Conjunto I1 e I3
$L(I1) \cap E(I2) \neq \emptyset$	$L(I2) \cap E(I3) = \emptyset$	$L(I1) \cap E(I3) = \emptyset$
$E(I1) \cap L(I2) \neq \emptyset$	$E(I2) \cap L(I3) = \emptyset$	$E(I1) \cap L(I3) = \emptyset$
$E(I1) \cap E(I2) = \emptyset$	$E(I2) \cap E(I3) = \emptyset$	$E(I1) \cap E(I3) = \emptyset$

En los programas secuenciales hay un orden fijo de ejecución de las instrucciones, siempre se sabe por dónde va a ir el programa. En cambio, en los programas concurrentes hay un orden parcial. Al haber solapamiento de instrucciones no se sabe cuál va a ser el orden de ejecución, puede ocurrir que ante unos mismos datos de entrada el flujo de ejecución no sea el mismo. Esto da lugar a que los programas concurrentes tengan un comportamiento indeterminista donde repetidas ejecuciones sobre un mismo conjunto de datos puedan dar diferentes resultados.

1.3.4. PROBLEMAS INHERENTES A LA PROGRAMACIÓN CONCURRENTE

A la hora de crear un programa concurrente podemos encontrarnos con dos problemas:

- **Exclusión mutua.** En programación concurrente es muy típico que varios procesos accedan a la vez a una variable compartida para actualizarla. Esto se debe evitar, ya que puede producir inconsistencia de datos: uno puede estar actualizando la variable a la vez que otro la puede estar leyendo. Por ello es necesario conseguir la exclusión mutua de los procesos respecto a la variable compartida. Para ello se propuso la **región crítica**. Cuando dos o más procesos comparten una variable, el acceso a dicha variable debe efectuarse siempre dentro de la región crítica asociada a la variable. Sólo uno de los procesos podrá acceder para actualizarla y los demás deberán esperar, el tiempo de estancia es finito.
- **Condición de sincronización.** Hace referencia a la necesidad de coordinar los procesos con el fin de sincronizar sus actividades. Puede ocurrir que un proceso P1 llegue a un estado X que no pueda continuar su ejecución hasta que otro proceso P2 haya llegado a un estado Y de su ejecución. La programación concurrente proporciona mecanismos para bloquear procesos a la espera de que ocurra un evento y para desbloquearlos cuando este ocurra.

Algunas herramientas para manejar la concurrencia son: la región crítica, los semáforos, región crítica condicional, buzones, sucesos, monitores y sincronización por rendez-vous.

ACTIVIDAD 1.9

Responde a las siguientes cuestiones:

Escribe alguna característica de un programa concurrente.

¿Cuál es la ventaja de la concurrencia en los sistemas monoprocesador?

¿Cuáles son las diferencias entre multiprogramación y multiproceso?

¿Cuáles son los dos problemas principales inherentes a la programación concurrente?

1.3.5. PROGRAMACIÓN CONCURRENTE CON JAVA

Al igual que el sistema operativo puede ejecutar varios procesos concurrentemente, dentro de un proceso podemos encontrarnos con varios hilos de ejecución. Un hilo es como una secuencia de control dentro de un proceso que ejecuta sus instrucciones de forma independiente, véase Figura1.18. Los hilos comparten el contexto del proceso, pero cada hilo mantiene una parte local.

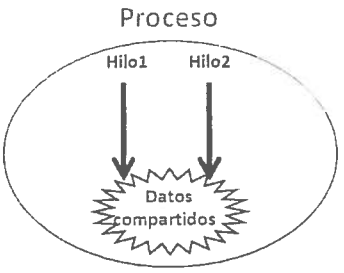

Figura 1.18. Hilos en un proceso.

Entre procesos e hilos hay algunas diferencias:

- Los hilos comparten el espacio de memoria del proceso, muchos comparten datos y espacios de direcciones; a diferencia de los procesos que generalmente poseen espacios de memoria de trabajo independientes e interactúan a través de mecanismos de comunicación dados por el sistema.
- Hilos y procesos pueden encontrarse en diferentes estados, pero los cambios de estado en los procesos son más costosos ya que los hilos pertenecen al mismo proceso. A los hilos también se les llama procesos ligeros.
- Se tarda menos tiempo en crear o en terminar un hilo que un proceso.
- En la comunicación entre procesos debe intervenir el núcleo del sistema, entre hilos no se necesita que intervenga el núcleo.

Para programar concurrentemente podemos dividir nuestro programa en hilos. Java proporciona la construcción de programas concurrentes mediante la clase **Thread** (hilo o hebra). Esta clase permite ejecutar código en un hilo de ejecución independiente.

En Java existen dos formas de utilizar o crear un hilo:

- Creando una clase que herede de la clase **Thread** y sobrecargando el método **run()**.
- Implementando la interface **Runnable**, y declarando el método **run()**. Se utiliza este modo cuando una clase ya deriva de otra. Por ejemplo, un applet deriva de la clase **Applet** por lo que no puede derivar también de **Thread**, en este caso tiene que implementar la interface **Runnable**.

El siguiente ejemplo crea un hilo de nombre *HiloSimple* heredando de la clase **Thread**. En el método **run()** se indican las líneas de código que se ejecutarán simultáneamente con las otras partes del programa. Cuando se termina la ejecución de ese método, el hilo de ejecución termina también:

```
public class HiloSimple extends Thread {
   public void run() {
     for (int i=0; i<5;i++)
       System.out.println("En el Hilo... ");
   }
}//
```

Para usar el hilo creo la clase *UsaHilo*:

```
public class UsaHilo {
    public static void main(String[] args) {
      HiloSimple hs = new HiloSimple();
      hs.start();
      for (int i=0; i<5;i++)
```

```
        System.out.println("Fuera del hilo..");
    }
}//
```

Desde esta clase se arranca el hilo: primero se invoca al operador *new* para crear el hilo y luego al método **start()** que invoca al método **run()**. La compilación y ejecución muestra la siguiente salida, en la que se puede observar que se intercala las instrucciones del hilo y de fuera del hilo. La salida puede variar cada vez que vez que ejecutemos el programa:

```
D:\CAPIT1>javac HiloSimple.java
D:\CAPIT1>javac UsaHilo.java
D:\CAPIT1>java UsaHilo
Fuera del hilo..
Fuera del hilo..
Fuera del hilo..
En el Hilo...
En el Hilo...
En el Hilo...
En el Hilo...
En el Hilo...
Fuera del hilo..
Fuera del hilo..
```

Las 2 clases anteriores implementando la interfaz **Runnable** quedarían así:

```
public class HiloSimple2 implements Runnable{
   public void run() {
      for (int i=0; i<5;i++)
         System.out.println("En el Hilo...");
   }
}//

public class UsaHilo2 {
    public static void main(String[] args) {
      HiloSimple2 hs = new HiloSimple2();
      Thread t = new Thread(hs);
      t.start();
      for (int i=0; i<5;i++)
        System.out.println("Fuera del hilo..");
   }
}//
```

En el siguiente capítulo se tratarán más ampliamente los hilos con Java.

1.4. PROGRAMACIÓN PARALELA Y DISTRIBUIDA

1.4.1. PROGRAMACIÓN PARALELA

Un **programa paralelo** es un tipo de programa concurrente diseñado para ejecutarse en un sistema multiprocesador. El procesamiento paralelo permite que muchos elementos de proceso independientes trabajen simultáneamente para resolver un problema. Estos elementos pueden ser un número arbitrario de equipos conectados por una red, un único equipo con varios procesadores o una combinación de ambos. El problema a resolver se divide en partes independientes de tal forma que cada elemento pueda ejecutar la parte de programa que le corresponda a la vez que los demás.

Recordemos que en un sistema **multiprocesador,** donde existe más de un procesador, podemos tener un proceso en cada procesador y todos juntos trabajan para resolver un problema. Cada procesador realiza una parte del problema y necesita intercambiar información con el resto. Según cómo se realice este intercambio podemos tener modelos distintos de programación paralela:

- Modelo de **memoria compartida**: los procesadores comparten físicamente la memoria, es decir, todos acceden al mismo espacio de direcciones. Un valor escrito en memoria por un procesador puede ser leído directamente por cualquier otro.
- Modelo de **paso de mensajes**: cada procesador dispone de su propia memoria independiente del resto y accesible sólo por él. Para realizar el intercambio de información es necesario que cada procesador realice la petición de datos al procesador que los tiene, y éste haga el envío. El entorno de programación PVM que veremos más adelante utiliza este modelo.

El intercambio de información entre procesadores depende del sistema de almacenamiento que se disponga. Según este criterio las arquitecturas paralelas se clasifican en: **Sistemas de memoria compartida o multiprocesadores**: los procesadores comparten físicamente la memoria; y **Sistemas de memoria distribuida o multicomputadores**: cada procesador dispone de su propia memoria.

Dentro de los sistemas de memoria distribuida o multicomputadores nos encontramos con los **Clusters**. Son sistemas de procesamiento paralelo y distribuido donde se utilizan múltiples ordenadores, cada uno con su propio procesador, enlazados por una red de interconexión más o menos rápida, de tal forma que el conjunto de ordenadores es visto como un único ordenador, más potente que los comunes de escritorio.

Tradicionalmente, el paralelismo se ha utilizado en centros de supercomputación para resolver problemas de elevado coste computacional en un tiempo razonable, pero en la última década su interés se ha extendido por la difusión de los procesadores con múltiples núcleos (combina dos o más procesadores independientes en un solo circuito integrado). Estos procesadores permiten que un dispositivo computacional exhiba una cierta forma del paralelismo a nivel de thread (thread-level parallelism) (TLP) sin incluir múltiples microprocesadores en paquetes físicos separados. Esta forma de TLP se conoce a menudo como multiprocesamiento a nivel de chip (chip-level multiprocessing) o CMP[2].

VENTAJAS E INCONVENIENTES

Ventajas del procesamiento paralelo:

- Proporciona ejecución simultánea de tareas.
- Disminuye el tiempo total de ejecución de una aplicación.
- Resolución de problemas complejos y de grandes dimensiones.
- Utilización de recursos no locales, por ejemplo, los recursos que están en una red distribuida, una WAN o la propia red internet.
- Disminución de costos, en vez de gastar en un supercomputador muy caro se pueden utilizar otros recursos más baratos disponibles remotamente.

Pero no todo son ventajas, algunos inconvenientes son:

---

[2] https://es.wikipedia.org/wiki/Procesador_multinúcleo

- Los compiladores y entornos de programación para sistemas paralelos son más difíciles de desarrollar.
- Los programas paralelos son más difíciles de escribir.
- El consumo de energía de los elementos que forman el sistema.
- Mayor complejidad en el acceso a los datos.
- La comunicación y la sincronización entre las diferentes subtareas.

La computación paralela resuelve problemas como: predicciones y estudios meteorológicos, estudio del genoma humano, modelado de la biosfera, predicciones sísmicas, simulación de moléculas... En algunos casos se dispone de tal cantidad de datos que serían muy lento o imposible tratar con máquinas convencionales.

ACTIVIDAD 1.10

Entra en la siguiente URL *https://computing.llnl.gov/tutorials/parallel_comp/* y responde a las siguientes cuestiones:

Cita algunas características de la computación serie.

Cita algunas características de la computación en paralelo.

Ámbitos en los que se usa la computación en paralelo.

¿Cómo hace uso de la computación paralela el proyecto SETI @ home?

1.4.2. PROGRAMACIÓN DISTRIBUIDA

Uno de los motivos principales para construir un sistema distribuido es compartir recursos. Probablemente, el sistema distribuido más conocido por todos es Internet que permite a los usuarios donde quiera que estén hacer uso de la World Wide Web, el correo electrónico y la transferencia de ficheros. Entre las aplicaciones más recientes de la computación distribuida se encuentra el *Cloud Computing* que es la computación en la nube o servicios en la nube, que ofrece servicios de computación a través de Internet.

Se define un sistema distribuido como aquel en el que los componentes hardware o software, localizados en computadores unidos mediante una red, comunican y coordinan sus acciones mediante el paso de mensajes. Esta definición tiene las siguientes consecuencias[3]:

- **Concurrencia:** lo normal en una red de ordenadores es la ejecución de programas concurrentes.
- **Inexistencia de reloj global:** cuando los programas necesitan cooperar coordinan sus acciones mediante el paso de mensajes. No hay una temporalización, los relojes de los host no están sincronizados.
- **Fallos independientes:** cada componente del sistema puede fallar independientemente, permitiendo que los demás continúen su ejecución.

La programación distribuida es un paradigma de programación enfocado en desarrollar sistemas distribuidos, abiertos, escalables, transparentes y tolerantes a fallos. Este paradigma es el resultado natural del uso de las computadoras y las redes. Casi cualquier lenguaje de

[3] Sistemas Distribuidos: Conceptos y Diseño. George Coulouris y otros. Ed: Addison-Wesley.

programación que tenga acceso al máximo al hardware del sistema puede manejar la programación distribuida, considerando una buena cantidad de tiempo y código[4].

Una arquitectura típica para el desarrollo de sistemas distribuidos es la arquitectura **cliente-servidor**. Los clientes son elementos activos que demandan servicios a los servidores realizando peticiones y esperando la respuesta, los servidores son elementos pasivos que realizan las tareas bajo requerimientos de los clientes.

Por ejemplo, un cliente web solicita una página, el servidor web envía al cliente la página solicitada. Véase Figura 1.19. La comunicación entre servidores y clientes se realiza a través de la red.

Existen varios modelos de programación para la comunicación entre los procesos de un sistema distribuido:

- **Sockets.** Proporcionan los puntos extremos para la comunicación entre procesos. Es actualmente la base de la comunicación. Pero al ser de muy bajo nivel de abstracción, no son adecuados a nivel de aplicación. En el capítulo 3 se tratarán los sockets en Java.
- **Llamada de procedimientos remotos o RPC** (*Remote Procedure Call*). Permite a un programa cliente llamar a un procedimiento de otro programa en ejecución en un proceso servidor. El proceso servidor define en su interfaz de servicio los procedimientos disponibles para ser llamados remotamente.
- **Invocación remota de objetos**. El modelo de programación basado en objetos ha sido extendido para permitir que los objetos de diferentes procesos se comuniquen uno con otro por medio de una *invocación a un método remoto* o **RMI** (*Remote Method Invocation*). Un objeto que vive en un proceso puede invocar métodos de un objeto que reside en otro proceso. **Java RMI** extiende el modelo de objetos de Java para proporcionar soporte de objetos distribuidos en lenguaje Java.

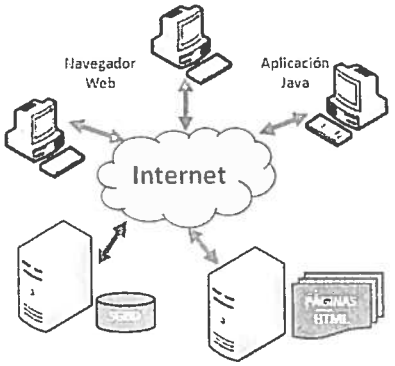

Figura 1.19. Cliente-servidor sobre web.

VENTAJAS E INCONVENIENTES

Ventajas que aportan los sistemas distribuidos:

- Se pueden compartir recursos y datos.
- Capacidad de crecimiento incremental.

[4] http://es.wikipedia.org/wiki/Programación_distribuida

- Mayor flexibilidad al poderse distribuir la carga de trabajo entre diferentes ordenadores.
- Alta disponibilidad.
- Soporte de aplicaciones inherentemente distribuidas.
- Carácter abierto y heterogéneo.

Pero no todo son ventajas, algunos inconvenientes son:

- Aumento de la complejidad, se necesita nuevo tipo de software.
- Problemas con las redes de comunicación: pérdida de mensajes, saturación del tráfico.
- Problemas de seguridad como por ejemplo ataques de denegación de servicio en la que se "bombardea" un servicio con peticiones inútiles de forma que un usuario interesado en usar el servicio no pueda usarlo.

ACTIVIDAD 1.11

Busca en Internet aplicaciones de los sistemas distribuidos.

PROGRAMACION CONCURRENTE, PARALELA Y DISTRIBUIDA

Programación Concurrente: Tenemos varios elementos de proceso (hilos, procesos) que trabajan de forma conjunta en la resolución de un problema. Se suele llevar a cabo en un único procesador o núcleo.

Programación Paralela: Es programación concurrente cuando se utiliza para acelerar la resolución de los problemas, normalmente usando varios procesadores o núcleos.

Programación Distribuida: Es programación paralela cuando los sistemas están distribuidos a través de una red (una red de procesadores); se usa paso de mensajes.

1.4.3. PVM. INSTALACIÓN Y CONFIGURACIÓN

PVM (*Paralel Virtual Machine* - Máquina virtual en paralelo) es un conjunto de herramientas software que permiten emular un marco de computación concurrente, distribuido y de propósito general, utilizando para ello grupos de ordenadores conectados, de manera que ni los ordenadores ni las redes que los conectan tienen las mismas características arquitectónicas.

Permite conectar entre sí ordenadores Unix y Windows (WIN95, NT 3.5, NT 4.0) para ser usados como un único gran ordenador paralelo de alto rendimiento. Así, grandes problemas de cómputo se pueden resolver de manera más rentable aprovechando la potencia y memoria de muchos equipos conectados.

Cientos de sitios en todo el mundo están usando PVM para resolver importantes problemas científicos, industriales y médicos, además de su uso como una herramienta educativa para enseñar programación paralela.

El modelo de computación de PVM se basa en considerar que una aplicación es una colección de tareas que se comunican y sincronizan mediante el paradigma de paso de mensajes. El sistema de PVM se compone de 3 partes[5]:

[5] Procesamiento paralelo teoría y programación. Sebastián Dormido Canto y otros, Ed: Sanz y Torres.

- **El demonio**, llamado **pvmd3** y a veces abreviado **PVMD**, que reside en todos los equipos que componen la máquina virtual. Uno de los equipos de la máquina virtual actúa como equipo anfitrión o maestro y los demás como esclavos, el maestro es el que inicia las tareas en paralelo. Figura 1.20.

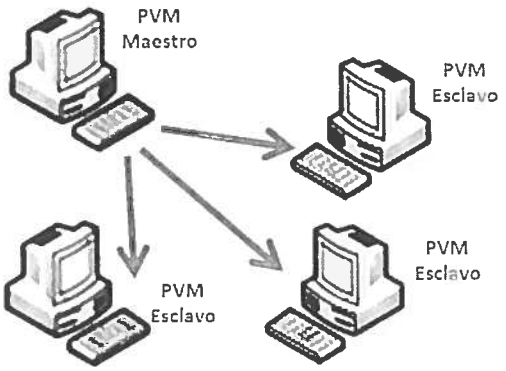

Figura 1.20. Máquina virtual PVM con varios equipos.

- **La biblioteca de desarrollo (API),** que contiene un repertorio de funciones que son necesarias para operar con las tareas de una aplicación, transmitir mensajes entre ellas y alterar la configuración de la máquina virtual.
- **La consola de PVM**, que actúa a modo de intérprete de comandos proporcionando una interfaz entre el usuario y el demonio.

VENTAJAS E INCONVENIENTES

Entre las ventajas podemos destacar las siguientes:

- Es una de las librerías de paso de mensajes más fáciles de usar.
- Fácil de instalar.
- Fácil de configurar. La aplicación decide dónde y cuando ejecutar o terminar las tareas, qué máquinas se añaden o se eliminan desde la máquina virtual paralela, qué tareas se pueden comunicar y/o sincronizar con otras.
- Se puede incorporar cualquier ordenador al esquema de la máquina virtual.

Inconvenientes:

- Al ser un esquema heterogéneo de ordenadores, el rendimiento puede verse mermado o incrementado, dependiendo de la capacidad de procesamiento de los ordenadores que formen parte del esquema de la máquina virtual. Lo que era una ventaja tiene su desventaja.
- Es algo deficiente en cuanto al paso de mensajes se refiere.
- PVM no es un estándar.

INSTALACIÓN

En este apartado veremos cómo instalar PVM en una máquina que tiene instalado Ubuntu. La instalación se debe realizar en todos los equipos que formen parte de PVM. Se recomienda tener privilegios de administrador para llevar a cabo la instalación. Podemos instalarlo de dos formas:

- Desde la línea de comandos de Ubuntu escribiendo las siguientes órdenes (incluye el paquete de ejemplos):

```
$ sudo apt-get install pvm pvm-dev pvm-examples
```

- Usando el entorno gráfico desde la opción de menú: ***Sistema → Administración → Centro de software de Ubuntu***. Buscamos el paquete **pvm, pvm-dev** y **pvm-examples**, al seleccionarlos nos pedirá confirmar la instalación de otros paquetes. Pulsamos el botón *Instalar*. Véase Figura 1.21.

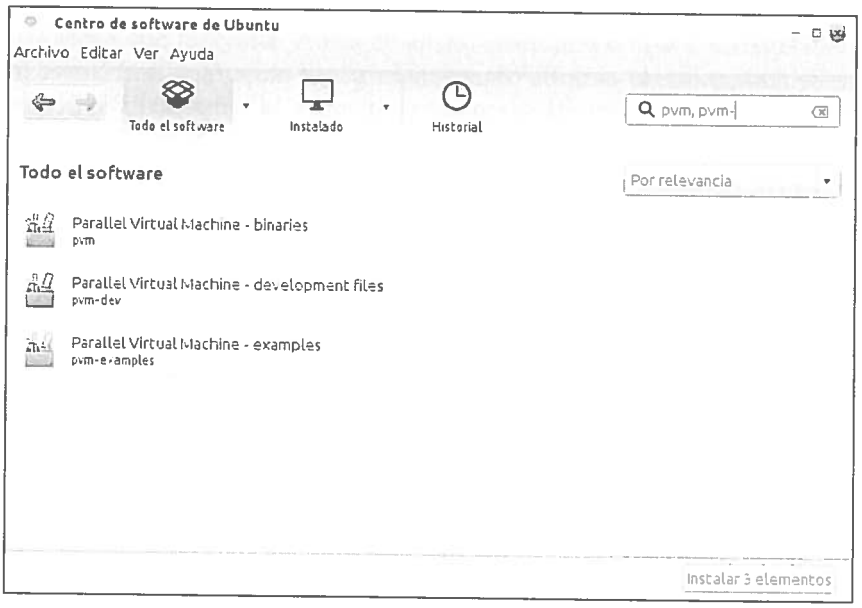

Figura 1.21. Instalación de PVM.

Una vez instalados los paquetes se habrá creado la carpeta **pvm3** en **/usr/lib/**. También se habrán creado 3 subcarpetas: **bin** que contiene un único fichero, **conf** que contienen dos ficheros llamados *LINUX.def* y *LINUX.m4* (que hacen referencia a la arquitectura del equipo) y **lib** donde se encuentran los ejecutables, Figura 1.22.

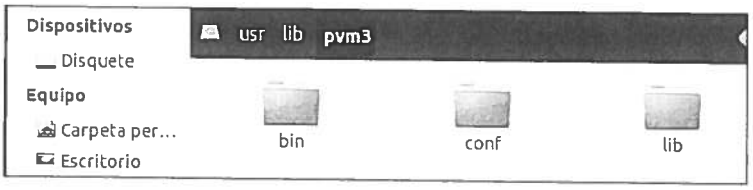

Figura 1.22. Carpetas /usr/lib/pvm3.

A continuación, configuramos PVM para trabajar con diferentes equipos. Supongamos que tengo 2 equipos con Ubuntu12.04, uno se llama *ubuntu12-maestro* y el otro *ubuntu12-esclavo*. El primer equipo actuará como maestro. Estos equipos tienen que tener instalado PVM. En ambos existe una cuenta de usuario con nombre *administrador* y clave *admin1234*.

Creo el fichero **.rhosts** (o lo edito si ya existe) en el directorio personal del equipo maestro y del esclavo, en este caso en */home/administrador*. Escribo una entrada por cada equipo al que me voy a conectar y el nombre de usuario para inicio de sesión en ese equipo, por ejemplo, desde el equipo maestro para conectarme al equipo de nombre *ubuntu12-esclavo* con usuario de nombre *administrador* escribo: *ubuntu12-esclavo administrador*. Desde el equipo esclavo añado la entrada: *ubuntu12-maestro administrador* para conectarme al equipo maestro con usuario *administrador*.

En ambos equipos se añaden las siguientes variables de entorno en el fichero **.bashrc** que está en la carpeta */home/administrador*:

```
export PVM_ROOT=/usr/lib/pvm3
export PVM_ARCH=$PVM_ROOT/lib/pvmgetarch
export PVM_RSH=/usr/bin/ssh
```

Con esto se indica donde está PVM instalado y qué arquitectura usamos (se define en el fichero *pvmgetarch*). En los equipos esclavos creamos la carpeta LINUX para guardar los ejecutables en */usr/lib/pvm3/bin*. Nos debe quedar: **/usr/lib/pvm3/bin/LINUX**.

Realizamos la primera prueba donde se usa la función **pvm_mytid()** para saber el identificador de la tarea o TID. Creamos el programa C de nombre *hola.c* en nuestra carpeta personal:

```
#include <stdio.h>
#include <pvm3.h>
int main()
{
  int mytid;

  mytid = pvm_mytid();
  printf("Mi TID es %x\n", mytid);
  pvm_exit();
  return 0;
}
```

La compilamos con la opción **–lpvm3**:

```
administrador@ubuntu12-maestro:~$ gcc hola.c -o hola -lpvm3
```

Después ejecutamos PVM y lanzamos la tarea con **spawn** de la siguiente manera:

```
administrador@ubuntu12-maestro:~$ pvm
pvm> spawn -> hola
spawn -> hola
[1]
0 successful
No such file
```

Se visualiza un error, *No such file*, indicando que no existe el fichero. Es porque falta indicar en una variable de entorno donde están los ejecutables. Salimos de PVM con **halt** y definimos la variable **PVM_PATH**, y volvemos a entrar para lanzar la tarea:

```
pvm> halt
halt
Terminado
administrador@ubuntu12-maestro:~$ PVM_PATH=/home/administrador
administrador@ubuntu12-maestro:~$ export PVM_PATH
administrador@ubuntu12-maestro:~$ pvm
pvm> spawn -> hola
spawn -> hola
[2]
1 successful
```

```
t40003
pvm> [2:t40003] Mi TID es 40003
[2:t40003] EOF
[2] finished

pvm>
```

De momento vemos que funciona. Antes de añadir máquinas o host a nuestro PVM hemos de instalar **SSH** (*Secure SHell*, intérprete de órdenes segura) si no lo tenemos instalado; ya que nos permite acceder a máquinas remotas a través de una red. Lo instalamos en el maestro y en el esclavo.

Podemos instalarlo desde el **Centro de software de Ubuntu**, buscamos *ssh* y pulsamos el botón *Instalar*. Véase Figura 1.23. Se deben instalar también los paquetes afectados.

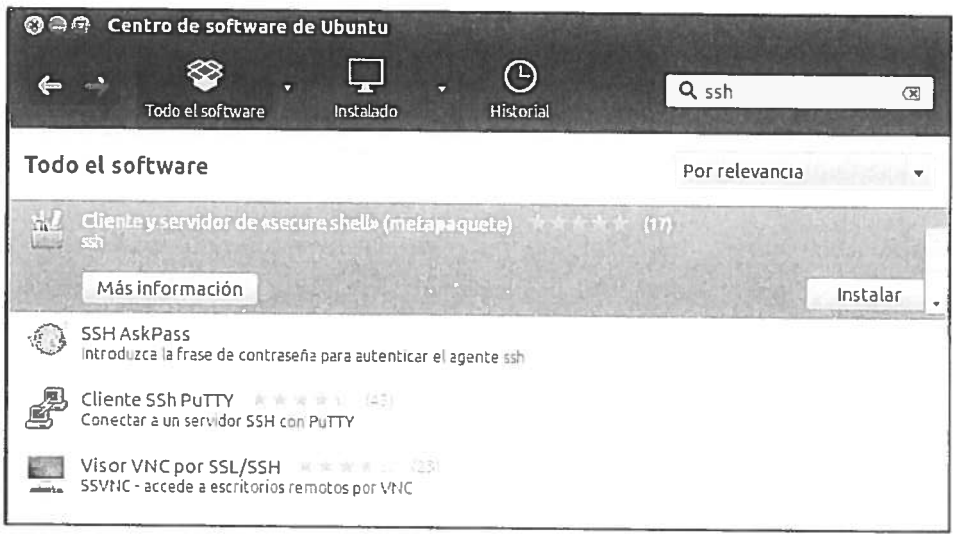

Figura 1.23. Instalación de SSH.

Para añadir un host desde PVM escribimos: *add nombrehost*. Por ejemplo, para añadir la máquina *ubuntu12-esclavo* escribo:

```
pvm> add ubuntu12-esclavo
add ubuntu12-esclavo
0 successful
                    HOST     DTID
          ubuntu12-esclavo No such host
pvm>
```

Pero puede ocurrir que no encuentre el host (*No such host*). Entonces añado la IP del host al que me voy a conectar en el fichero **/etc/hosts**, también añado la IP del maestro (las añadimos al principio del fichero), por ejemplo:

```
192.168.176.130   ubuntu12-maestro
192.168.176.134   ubuntu12-esclavo
```

Esto se hace en todas las máquinas que formen parte de PVM. Una vez realizado el cambio entro en PVM para añadir la máquina *ubuntu12-esclavo*:

```
pvm> add ubuntu12-esclavo
add ubuntu12-esclavo
The authenticity of host 'ubuntu12-esclavo (192.168.176.134)' can't be
established.
ECDSA key fingerprint is
cd:4e:ac:ad:21:77:96:27:ac:c7:6d:18:ed:1b:86:c7.
Are you sure you want to continue connecting (yes/no)? yes
administrador@ubuntu12-esclavo's password:
1 successful
                    HOST      DTID
        ubuntu12-esclavo     80000
pvm>
```

Se visualiza un mensaje indicando que la autenticidad del host *ubuntu12-esclavo* no puede ser establecida, y nos muestra la huella digital (fingerprint) de la clave ECDSA. Nos pregunta si queremos continuar, al decir *yes* agrega el host *ubuntu12-esclavo* a la lista de host conocidos o confiables y entonces pide la contraseña del usuario con nombre *administrador* de la máquina *ubuntu12-esclavo*. A continuación, se muestra el host y el DTID que se le ha asignado. Cada vez que nos conectemos a *ubuntu12-esclavo* nos pedirá la clave. Luego veremos cómo quitarlo.

Ahora escribimos desde PVM la orden **conf** para que nos muestre los hosts que están en PVM; vemos dos, el maestro y el esclavo:

```
pvm> conf
conf
2 hosts, 1 data format
                    HOST      DTID      ARCH   SPEED       DSIG
        ubuntu12-maestro     40000     LINUX    1000 0x00408841
        ubuntu12-esclavo     80000     LINUX    1000 0x00408841

pvm>
```

Con esta orden se puede comprobar el nombre y número de los hosts que componen la máquina virtual, así como la arquitectura (columna ARCH), número base para identificadores de tareas (DTID), velocidad relativa (SPEED) y DSIG.

Hasta aquí ya tenemos preparado el entorno para empezar a ejecutar los programas. Antes veamos como hacer para que no nos vuelva a pedir la clave del usuario *administrador* cada vez que se añada el host *ubuntu12-esclavo*.

Desde la línea de comandos (fuera de PVM) y en la máquina que hace de maestro hacemos lo siguiente: nos vamos a la carpeta **.ssh**, y ejecutamos **ssh-keygen -t dsa** para generar un par de claves pública/privada; cada vez que nos pida la clave pulsamos la tecla [Intro], es decir dejamos vacío el *passphrase*:

```
administrador@ubuntu12-maestro:~$ cd ~/.ssh
administrador@ubuntu12-maestro:~/.ssh$ ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (/home/administrador/.ssh/id_dsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/administrador/.ssh/id_dsa.
Your public key has been saved in /home/administrador/.ssh/id_dsa.pub.
The key fingerprint is:
8d:1e:4b:98:dd:df:e1:a9:15:e1:31:a8:24:3b:27:eb administrador@ubuntu12-
maestro
The key's randomart image is:
```

```
+--[ DSA 1024]----+
|                 |
|               .   |
|         . . . +   |
|        + B . . + |
|       o S =   +   |
|        o B . o + |
|         +   . =   |
|        .     o    |
|         E   .     |
+-----------------+
administrador@ubuntu12-maestro:~/.ssh$
```

Esto genera 2 ficheros: *id_dsa.pub* (public key) e *id_dsa* (private key). Se lo pasamos al usuario *administrador* en *ubuntu12-esclavo* mediante el comando **ssh-copy-id**, nos pedirá la clave del usuario:

```
administrador@ubuntu12-maestro:~/.ssh$ ssh-copy-id
                                        administrador@ubuntu12-esclavo
administrador@ubuntu12-esclavo's password:
Now try logging into the machine, with "ssh 'administrador@ubuntu12-
esclavo'", and check in:

  ~/.ssh/authorized_keys

to make sure we haven't added extra keys that you weren't expecting.

administrador@ubuntu12-maestro:~/.ssh$
```

Probamos que no nos pide la clave al conectarnos a *ubuntu12-esclavo* con el comando *ssh*:

```
administrador@ubuntu12-maestro:~/.ssh$ ssh administrador@ubuntu12-
                                                          esclavo
Welcome to Ubuntu 12.04.3 LTS (GNU/Linux 3.5.0-40-generic i686)

 * Documentation:  https://help.ubuntu.com/

New release '14.04.5 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Wed Jul  5 11:57:49 2017 from localhost
administrador@ubuntu12-esclavo:~$
```

Para salir usamos el comando **exit**:

```
administrador@ubuntu12-esclavo:~$ exit
logout
Connection to ubuntu12-esclavo closed.
administrador@ubuntu12-maestro:~/.ssh$
```

NOTA: La dirección IP de los host debe ser fija. Hemos de tenerlo en cuenta cada vez que empecemos a trabajar con PVM. Si es dinámica puede que un día funcione y otro no; entonces hemos de revisar las IPs y el fichero **/etc/hosts**.

Si no tenemos máquinas físicas podemos usar entornos como VMware con varios sistemas Linux instalados.

En la siguiente tabla se muestran algunos de los comandos más importantes del intérprete de comandos PVM:

Comandos	
add máquina	Incorpora la máquina indicada a PVM
delete máquina	Elimina la máquina del entorno PVM, no se puede eliminar la máquina desde la que estamos ejecutando los comandos
conf	Muestra la configuración actual de PVM.
ps	Listado de procesos de PVM.
halt	Apaga y sale de PVM.
help	Lista los comandos de PVM.
id	Visualiza el TID de la consola.
jobs	Genera un listado de los trabajos en ejecución
kill	Mata un proceso de la PVM.
quit	Sale de la máquina paralela virtual sin apagarla.
spawn	Arranca una aplicación bajo PVM.
version	Visualiza la versión de PVM
reset	Inicializa PVM.

Con esto ya podemos empezar a crear programas paralelos.

1.4.4. EJECUCIÓN DE TAREAS EN PARALELO CON PVM

PVM está basado en el paso de mensajes. Inicialmente para cada tarea PVM se crea un buffer activo de envío y otro de recepción, no siendo necesario, en la mayoría de los casos, la creación de nuevos buffers. Todas las operaciones de empaquetamiento y desempaquetamiento se realizarán en el buffer activo. El envío de un mensaje requiere 3 pasos:

- Inicialización de un buffer de envío. Usaremos la función **pvm_initsend()**.
- Empaquetamiento del mensaje en el buffer, para ello usaremos las funciones **pvm_pkXXX()**.
- Envío del mensaje a una o varias tareas, función **pvm_send()**.

La recepción requiere dos pasos:

- Recepción del mensaje, función **pvm_recv()**;
- Desempaquetamiento de los datos enviados en el mensaje, funciones **pvm_upkXXX()**.

Veamos a continuación estas funciones y otras que nos serán útiles para lanzar tareas en paralelo:

int tid = pvm_mytid(void): devuelve el TID del proceso o tarea que invoca a la función. Un valor menor que 0 indica error.

int info = pvm_exit(void): sale del ambiente PVM.

int tid = pvm_parent(void): devuelve el TID del proceso que creó la tarea que invoca a la función. Si el proceso no fue creado con **pvm_spawn()**, entonces *tid = PvmNoParent*.

int numt = pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids): se utiliza para la creación de procesos. Genera *ntask* copias del proceso cuyo nombre es el string *task*. El parámetro *argv* es un puntero a un array de argumentos para *task*. El argumento

flag puede tener varios valores, es 0 cuando PVM elige en qué máquina ejecutar los procesos (también se puede poner *PvmTaskDefault* en lugar de 0), *flag* = 1 cuando el proceso se ejecuta en la máquina indicada en el parámetro *where* (se puede usar *PvmTaskHost* en lugar de 1), con *flag* = 2 (*PvmTaskArch*) el parámetro *where* especifica el tipo de arquitectura en que se va a crear el proceso, etc. *where* es una cadena de caracteres que indica donde crear el proceso. La función devuelve en *numt* el número de tareas que se crearon satisfactoriamente y en el vector *tids* los TIDs de las tareas creadas con éxito. Ejemplo:

```
cc = pvm_spawn("esclavo1", (char**)0, 0, "", 1, &tid);
```

Se lanza el proceso *esclavo1*, el parámetro *argv* es nulo es decir no se envían argumentos al proceso, *flag* es 0 y *where* es nulo es decir "", en este caso PVM elige la máquina donde crear el proceso; el parámetro *ntask* es 1 indica que se crea 1 copia del proceso y por último *&tid* contiene el TID de la tarea creada.

int bufid = pvm_initsend(int encoding): esta función se utiliza para limpiar el buffer de envío que se encuentre activo. Siempre debe usarse antes de proceder al empaquetado de un mensaje. Inicializa el buffer de envío dándole un esquema de codificación identificado por *encoding*. Los valores posibles son: *PvmDataDefault* es el modo por defecto igual a XDR encoding, *PvmDataRaw* no se realiza codificación, etc. Ejemplo:

```
pvm_initsend(PvmDataDefault);
```

int info = pvm_send(int tid, int msgtag): envía a la tarea identificada por *tid* el mensaje previamente empaquetado en el buffer activo. *msgtg* es la etiqueta que se da al mensaje que está en el buffer activo.

int info = pvm_pkstr(char *cp): empaqueta una cadena de caracteres, recibe un puntero a la cadena a empaquetar. Devuelve un código de estado de la operación, un valor menor que 0 indica error. El siguiente ejemplo limpia el buffer de envío (con **pvm_initsend()**), empaqueta una cadena de caracteres (**pvm_pkstr()**) y se la envía (**pvm_send()**) al proceso padre identificado por *ptid*:

```
char buf[100];
int ptid = pvm_parent();
strcpy(buf, "Hola Mundo");
pvm_initsend(PvmDataDefault);
pvm_pkstr(buf);
pvm_send(ptid, 1);
```

int pvm_recv(int tid, int msgtag): espera a recibir un mensaje etiquetado como *msgtag* enviado por la tarea *tid*. Cuando se coloca el valor -1 en *tid* y/o *msgtag* se aceptarán mensajes con cualquier etiqueta y/o de cualquier tarea. Una vez que se recibe el mensaje se crea un nuevo buffer de recepción. Devuelve el identificador del buffer de recepción que ha sido creado del mensaje activo.

int info = pvm_bufinfo(int bufid, int *bytes, int *msgtag, int *tid): devuelve información sobre el mensaje almacenado en el buffer *buffid*. *bytes* es la longitud del mensaje en bytes, *msgtag* es la etiqueta del mensaje, *tid* el identificador de la tarea. El siguiente ejemplo recibe un mensaje de cualquier tarea lanzada y obtiene información sobre él:

```
bufid = pvm_recv(-1, -1);
pvm_bufinfo(bufid, &longitud, &tipo, &tarea_origen);
```

int info = pvm_upkstr(char *cp): desempaqueta los datos recibidos (cadena de caracteres). El siguiente ejemplo espera a recibir un mensaje, lo desempaqueta del buffer activo y lo visualiza:

```
bufid = pvm_recv(-1, -1);
pvm_upkstr(buf);
printf("Mensaje: %s\n", buf);
```

int info = pvm_pkint(int *ip, int nitem, int stride): empaqueta enteros. *ip* es un puntero al primer elemento del array a empaquetar. *nitem* es el número total de enteros a empaquetar. *stride* establece la distancia entre 2 elementos consecutivos, un valor de 1 indica que se empaqueta el array de forma continua, un valor de 2 indica que cada 2 elementos se empaquetan y así sucesivamente.

int info = pvm_upkint(int *ip, int nitem, int stride): desempaqueta enteros. Los parámetros significan lo mismo que en la función de empaquetado.

A continuación, vamos a probar algunos ejemplos que vienen con la instalación de PVM en el paquete **pvm-examples** y que se instalan en la carpeta **/usr/share/doc/pvm-examples**. Localizamos un fichero de nombre **examples.tar.gz**. Lo descomprimimos y extraemos los ejemplos **hello.c** y **hello_other.c.** El primero es un proceso (maestro) que lanza una copia del proceso *hello_other* (esclavo) a las máquinas que forman parte del entorno PVM, recibe un mensaje del proceso llamado. En la Figura 1.24 se muestra el proceso maestro **hello.c**.

```
#include <stdio.h>
#include "pvm3.h"                          <- Librería de PVM
#include <stdlib.h>
main()
{
    int cc, tid;
    char buf[   ];
                                           TID de este proceso
    printf("                 ", pvm_mytid());
                                                  Lanzar tarea hello_other
                                                  en equipo remoto
    cc = pvm_spawn(                 , (char**) ,  ,  ,  , &tid);

    if (cc ==  ) {                         Recepción del mensaje
        cc = pvm_recv(-  , -  );           <-
                                                  Información del mensaje
        pvm_bufinfo(cc, (int*)  , (int*)  , &tid);   <-
        pvm_upkstr(buf);                   <-  Desempaquetado del mensaje
        printf("                 ", tid, buf);

    } else
        printf("                 ");

    pvm_exit();                            <-  Salir de PVM
    exit( );
}
```

Figura 1.24. Programa *hello.c*.

Nos fijamos en los métodos **pvm_mytid()** que obtiene el TID de este proceso, **pvm_spawn()** para lanzar la tarea, **pvm_recv()** que espera a recibir un mensaje de cualquier tarea lanzada, **pvm_bufinfo()** que obtiene información del mensaje y **pvm_upkstr()** que desempaqueta el mensaje recibido. El segundo proceso **hello_other.c** se muestra en la Figura 1.25, lo que hace es enviar un mensaje al proceso padre en el que se incluye el nombre del host desde el que se envía.

Nos fijamos en los métodos **pvm_parent()** que obtiene el TID del proceso padre, **pvm_initsend()** que limpia el buffer de envío, **pvm_pkstr()** que empaqueta el mensaje a enviar, en este caso una cadena de caracteres almacenada en *buf*; y **pvm_send()** que envía el mensaje al proceso padre. La función **gethostname()** (no es una función PVM) devuelve el nombre de la máquina.

```
#include "pvm3.h"                    <-- Librería de PVM
#include <stdlib.h>
#include <string.h>
main()
{
    char buf[100];
    int ptid = pvm_parent();         <-- TID del proceso padre
                                     <-- Prepara el mensaje a enviar

    strcpy(buf, "hello, world from ");
    gethostname(buf + strlen(buf), 64);

    pvm_initsend(PvmDataDefault);    <-- Limpia el buffer
    pvm_pkstr(buf);                  <-- Empaquetado del mensaje
    pvm_send(ptid, 1);               <-- Envío del mensaje

    pvm_exit();                      <-- Salir de PVM
    exit(0);
}
```

Figura 1.25. Programa *hello_other.c*.

Para probar estos programas copiamos los dos ficheros en nuestra carpeta personal, en el ordenador que hace de maestro (*ubuntu12-maestro*), y los compilamos (hemos de incluir en los ficheros los includes que se muestran en la imagen, en caso de que no aparezcan):

```
administrador@ubuntu12-maestro:~$ gcc hello.c -o hello -lpvm3
```

Copiamos en la carpeta **/usr/lib/pvm3/bin/LINUX** de la máquina que hace de esclavo (*ubuntu12-esclavo*) el programa ejecutable *hello_other*. También podemos compilar el programa desde el ordenador esclavo:

```
administrador@ubuntu12-esclavo:~$ gcc hello_other.c -o
                                      hello_other -lpvm3
administrador@ubuntu12-esclavo:~$ sudo cp hello_other
                                      /usr/lib/pvm3/bin/LINUX/
```

A continuación en el maestro definimos la variable **PVM_PATH** y entramos en PVM. Primero añadimos el host esclavo y después lanzamos el proceso *hello* con la orden **spawn -> hello**:

```
administrador@ubuntu12-maestro:~$ PVM_PATH=/home/administrador
administrador@ubuntu12-maestro:~$ export PVM_PATH
administrador@ubuntu12-maestro:~$ pvm

pvm> add ubuntu12-esclavo
add ubuntu12-esclavo
1 successful
```

```
                    HOST      DTID
        ubuntu12-esclavo     80000
pvm> conf
conf
2 hosts, 1 data format
                    HOST      DTID     ARCH   SPEED       DSIG
        ubuntu12-maestro     40000    LINUX    1000 0x00408841
        ubuntu12-esclavo     80000    LINUX    1000 0x00408841
spawn -> hello
[2]
1 successful
t40002
pvm> [2:t80002] EOF
[2:t40002] i'm t40002
[2:t40002] from t80002: hello, world from ubuntu12-esclavo
[2:t40002] EOF
[2] finished

pvm>
```

Los números que aparecen entre corchetes identifican las tareas de los hosts y tienen relación con el DTID del host.

NOTA: Podemos probar el programa maestro y el esclavo en la misma máquina, es decir sin añadir ningún host. Veremos que los identificadores de tareas hacen referencia al host donde se ejecutan.

En el siguiente ejemplo el maestro envía una cadena al esclavo, y el esclavo se la devuelve al maestro en mayúsculas. Habrá dos tipos de mensajes, uno cuando el maestro envía al esclavo y el otro cuando el esclavo envía al maestro. Al primer mensaje se le etiqueta con la variable *etqenvio*, con valor 1. Al segundo se le etiqueta con la variable *etqrecibe* con valor 2. El código del proceso maestro **cadmaestro.c** es:

```
#include <stdio.h>
#include "pvm3.h"
#include <stdlib.h>
#include <string.h>
main()
{
  int tareas, cc, tid;
  char buf[100];
  int etqenvio=1;
  int etqrecibe=2;
  strcpy(buf, "mensaje en minúscula");

  tareas = pvm_spawn("cadesclavo", (char**)0, 0, "", 1, &tid);

  //ENVIO CADENA
  pvm_initsend(PvmDataDefault);
  pvm_pkstr(buf);
  pvm_send(tid, etqenvio); //envio con etiq 1
  printf("\tENVÍO A t%x: %s\n", tid, buf);

  //RECIBO CADENA
```

```
    cc = pvm_recv(tid, etqrecibe); //recibo con etiq 2
    pvm_upkstr(buf); //desempaqueta un entero
    printf("\tRECIBO DE t%x: %s\n", tid, buf);

    pvm_exit();
    exit(0);
}
```

El proceso esclavo recibe del maestro el mensaje etiquetado con 1 y envía el mensaje con etiqueta 2. El código del proceso esclavo **cadesclavo.c** es el siguiente:

```
#include <stdio.h>
#include "pvm3.h"
#include <stdlib.h>
#include <string.h>
main()
{
    char buf2[100];
    char buf[100];
    int i, parent_tid = pvm_parent();

    //SE RECIBEN LOS DATOS DEL MAESTRO
    pvm_recv(parent_tid, 1); //recibo con etiqueta 1
    pvm_upkstr(buf);

    for(i = 0; i< strlen(buf) ; i++)
       buf2[i] = toupper(buf[i]);

    buf2[strlen(buf)]='\0';

    //SE ENVIA EL RESULTADO AL MAESTRO
    pvm_initsend(PvmDataDefault);
    pvm_pkstr(buf2);
    pvm_send(parent_tid, 2); //envio con etiqueta 2

    pvm_exit();
    exit(0);
}
```

Compilamos y ejecutamos, no olvidemos copiar el ejecutable del proceso esclavo *cadesclavo* en la máquina que hace de esclavo; o bien compilarlo en la máquina que hace de esclavo y copiarlo en la carpeta **/usr/lib/pvm3/bin/LINUX/:**

```
administrador@ubuntu12-maestro:~$gcc cadmaestro.c -o cadmaestro -lpvm3

administrador@ubuntu12-esclavo:~$ gcc cadesclavo.c -o cadesclavo -lpvm3
administrador@ubuntu12-esclavo:~$ sudo cp cadesclavo
              /usr/lib/pvm3/bin/LINUX/
administrador@ubuntu12-esclavo:~$
```

Desde la máquina que hace de maestro ejecutamos:

```
pvm> spawn -> cadmaestro
spawn -> cadmaestro
[3]
1 successful
```

```
t40003
pvm> [3:t80003] EOF
[3:t40003]  ENVÍO A t80003: mensaje en minúscula
[3:t40003]  RECIBO DE t80003: MENSAJE EN MINúSCULA
[3:t40003] EOF
[3] finished
```

ACTIVIDAD 1.12

Para enviar enteros se usa la función **pvm_pkint().** Por ejemplo para enviar un único entero almacenado en la variable *num* a un proceso escribo **pvm_pkint(&num, 1, 1)**. Escribe un proceso maestro que envíe a un proceso esclavo un número y el esclavo devuelva el cubo del número que recibe.

En los ejemplos anteriores se generaba una copia del proceso esclavo (*hello_other*), el ejecutable se localizaba en *ubuntu12-esclavo*. A continuación copiamos el ejecutable en **/usr/lib/pvm3/bin/LINUX** de la máquina que hace de maestro (*ubuntu12-maestro*). Si no hemos creado la carpeta LINUX tendremos que crearla y después copiar *hello_other*:

```
administrador@ubuntu12-maestro:~$ gcc hello_other.c -o hello_other
                                      -lpvm3
administrador@ubuntu12-maestro:~$ sudo mkdir /usr/lib/pvm3/bin/LINUX/
administrador@ubuntu12-maestro:~$ sudo cp hello_other
                                       /usr/lib/pvm3/bin/LINUX/
administrador@ubuntu12-maestro:~$
```

Cambiamos el programa **hello.c**, le llamamos **hello2.c**. En este caso se lanzarán 2 tareas (o procesos) con **pvm_spawn()**, una en cada host, por tanto se generarán dos copias del proceso esclavo: *tareas = pvm_spawn("hello_other", (char**)0, 0, "", 2, tid);*

Después se hace un bucle que recogerá los mensajes enviados por cada uno de los procesos lanzados:

```
#include <stdio.h>
#include "pvm3.h"
#include <stdlib.h>
main()
{
      int tareas, cc, tid[2], i;
      char buf[100];

      printf("PROCESO MAESTRO: t%x\n", pvm_mytid());
      tareas = pvm_spawn("hello_other", (char**)0, 0, "", 2, tid);
      printf("NUMERO DE TAREAS lanzadas :%d \n", tareas);
      //RECIBO MENSAJES
      for(i=0; i< tareas; i++) {
            cc = pvm_recv(tid[i], -1);//se recibe mensaje
            pvm_upkstr(buf); //se desempaqueta
            printf("MENSAJE DE t%x: %s\n", tid[i], buf);
      }
      pvm_exit();
      exit(0);
}
```

A la hora de visualizar el mensaje recibido se puede ver de qué máquina viene, uno de *ubuntu12-esclavo* y el otro de *ubuntu12-maestro*. Lo compilamos y luego lo probamos:

```
administrador@ubuntu12-maestro:~$ gcc hello2.c -o hello2 -lpvm3

pvm> spawn -> hello2
spawn -> hello2
[5]
1 successful
t40005
pvm> [5:t80005] EOF
[5:t40005] PROCESO MAESTRO: t40005
[5:t40005] NUMERO DE TAREAS lanzadas :2
[5:t40005] MENSAJE DE t80005: hello, world from ubuntu12-esclavo
[5:t40005] MENSAJE DE t40006: hello, world from ubuntu12-maestro
[5:t40005] EOF
[5:t40006] EOF
[5] finished

pvm>
```

En el siguiente ejemplo los procesos esclavos realizarán la suma de un array. El proceso maestro envía a los esclavos los elementos del array a sumar. Estos se repartirán entre las dos máquinas que forman PVM de tal forma que una máquina suma los 5 primeros elementos y la otra los 5 siguientes. Habrá dos tipos de mensajes, uno cuando el maestro envía al esclavo y el otro cuando el esclavo envía al maestro (como se vio en un ejemplo anterior), cada uno tendrá una etiqueta diferente. El código del proceso maestro **summaestro.c** es el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
#include <pvm3.h>
int main()
  {
      int tabla[10];    /* Tabla con los números a sumar */
      int tresult[2];   /* resultados de la suma por los esclavos*/
      int tareas, cc, tid[2], i, sum =0 ;
      int etqenvio=1, etqrecibe=2; /*etiquetas de envio y recepcion*/

      printf("PROCESO MAESTRO: t%x\n", pvm_mytid());

      //SE LLENA LA TABLA CON NUMEROS
      for(i = 0; i < 10; i++) tabla[i] = i;

      //SE CREAN LOS ESCLAVOS
      tareas = pvm_spawn("sumesclavo", (char**)0, 0, "", 2, tid);
      printf("TAREAS: %d\n", tareas);

      //SE ENVIAN LOS DATOS
      for(i = 0; i < 2; i++) {
          pvm_initsend(PvmDataDefault);
          pvm_pkint(tabla + i*5, 5, 1); //5 num para cada esclavo
          pvm_send(tid[i], etqenvio);   //envía con etiq 1
      }
      //SE RECIBEN LOS DATOS
      for(i = 0; i < 2; i++) {
          cc = pvm_recv(tid[i], etqrecibe); //recibe con etiq 2
          pvm_upkint(tresult + i, 1, 1); //desempaqueta un entero
          printf("- RECIBO DE t%x: %d\n", tid[i], tresult[i]);
      }
```

```
        //SE OBTIENE LA SUMA
        for(i = 0; i < 2; i++) sum = sum + tresult[i];
        printf("LA SUMA ES = %d\n", sum);

        pvm_exit();
        exit(0);
    }
```

El código del proceso esclavo **sumesclavo.c** es el siguiente:

```
#include <stdlib.h>
#include <stdio.h>
#include <pvm3.h>
int main()
{
        int mytid, parent_tid;
        int tabla[5];    //tabla para enviar al maestro
        int sum =0, i;

        mytid = pvm_mytid();
        parent_tid = pvm_parent();

        //SE RECIBEN LOS DATOS DEL MAESTRO
        pvm_recv(parent_tid, 1); //recibo con etiqueta 1 los 5 numeros
        pvm_upkint(tabla, 5, 1);

        //SE CALCULA LA SUMA
        for(i = 0; i < 5; i++) sum = sum + tabla[i];

        //SE ENVIA EL RESULTADO AL MAESTRO
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&sum, 1, 1);
        printf("\tESCLAVO t%x Suma = %d ", mytid,  sum);
        pvm_send(parent_tid, 2); //envio con la etiqueta 2

        pvm_exit();
        exit(0);
 }
```

Lo compilamos y ejecutamos, no debemos olvidar dejar el proceso esclavo en la carpeta **/usr/lib/pvm3/bin/LINUX** de ambas máqinas:

```
administrador@ubuntu12-maestro:~$ gcc summaestro.c -o summaestro -lpvm3
administrador@ubuntu12-maestro:~$ gcc sumesclavo.c -o sumesclavo -lpvm3
administrador@ubuntu12-maestro:~$ sudo cp sumesclavo
                                  /usr/lib/pvm3/bin/LINUX/
[sudo] password for administrador:
administrador@ubuntu12-maestro:~$

administrador@ubuntu12-esclavo:~$ gcc sumesclavo.c -o sumesclavo -lpvm3
administrador@ubuntu12-esclavo:~$ sudo cp sumesclavo
                                  /usr/lib/pvm3/bin/LINUX/
[sudo] password for administrador:
administrador@ubuntu12-esclavo:~$
```

Desde la máquina que hace de maestro ejecutamos:

```
pvm> spawn -> summaestro
spawn -> summaestro
[6]
1 successful
t40009
pvm> [6:t40009] PROCESO MAESTRO: t40009
[6:t40009] TAREAS: 2
[6:t40009] - RECIBO DE t80009: 10
[6:t40009] - RECIBO DE t4000a: 35
[6:t40009] LA SUMA ES = 45
[6:t40009] EOF
[6:t80009]  ESCLAVO t80009 Suma = 10
[6:t80009] EOF
[6:t4000a]  ESCLAVO t4000a Suma = 35
[6:t4000a] EOF
[6] finished

pvm>
```

En la siguiente tabla se resumen las funciones de PVM:

Funciones	
Para el control de procesos	pvm_spawn(), pvm_exit(), pvm_kill(), pvm_start_pvmd()
De información	pvm_mytid(), pvm_parent(), pvm_tidtohost(), pvm_config(), pvm_tasks(), pvm_perror(), pvm_pstat(), pvm_mstat(), pvm_setopt(), pvm_getopt(), pvm_bufinfo()
De configuración dinámica de la máquina virtual	pvm_addhosts(), pvm_delhosts()
Para señalización	pvm_sendsig(), pvm_notify()
Para control de buffers	pvm_initsend(), pvm_mkbuf(),pvm_freebuf(), pvm_getsbuf(), pvm_getrbuf(), pvm_setsbuf(), pvm_setrbuf()
Para operaciones colectivas	pvm_joingroup (), pvm_lvgroup(), pvm_gettid(), pvm_getinst(), pvm_getsize(), pvm_barrier(),pvm_bcast(), pvm_reduce(), pvm_scatter(), pvm_gather()
Para envío y recepción de mensajes	pvm_send(), pvm_psend(), pvm_mcast(),pvm_recv(), pvm_nrecv(), pvm_trecv(), pvm_probe(), pvm_bufinfo()
Para empaquetado y desempaquetado de datos (XXX tipo de dato)	En general pvm_pkXXX(), pvm_upkXXX() Para cadenas: pvm_pkstr(), pvm_upkstr()

Más información de las funciones PVM se puede obtener desde esta URL: http://www.csm.ornl.gov/pvm/man/manpages.html.