Clusterización

Remo Suppi Boldrito

PID_00174431

Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

Los avances en la tecnología han llevado a procesadores rápidos, de bajo coste y a redes altamente eficientes, lo cual ha favorecido un cambio de la relación precio/prestaciones en favor de la utilización de sistemas de procesadores interconectados, en lugar de un único procesador de alta velocidad. Este tipo de arquitectura se puede clasificar en dos configuraciones básicas:

1) Sistemas fuertemente acoplados (*tightly coupled systems*): son sistemas donde la memoria es compartida por todos los procesadores (*shared memory systems*) y la memoria de todos ellos "se ve" (por parte del programador) como una única memoria.

2) Sistemas débilmente acoplados (*loosely coupled systems*): no comparten memoria (cada procesador posee la suya) y se comunican mediante mensajes pasados a través de una red (*message passing systems*).

En el primer caso, son conocidos como *sistemas paralelos de cómputo* (*parallel processing system*) y en el segundo, como *sistemas distribuidos de cómputo* (*distributed computing systems*).

> Un sistema distribuido es una colección de procesadores interconectados mediante una red, donde cada uno tiene sus propios recursos (memoria y periféricos) y se comunican intercambiando mensajes por la red.

En este módulo se verán diferentes formas de crear y programar un sistema de cómputo distribuido, así como las herramientas y librerías más importantes para cumplir este objetivo.

Objetivos

En los materiales didácticos de este módulo encontraréis los contenidos y las herramientas procedimentales para conseguir los objetivos siguientes:

1. Analizar las diferentes infraestructuras y herramientas para el cómputo de altas prestaciones (incluida la virtualización) (HPC).
2. Configurar e instalar un cluster de HPC y las herramientas de monitorización correspondientes.
3. Instalar y desarrollar programas de ejemplos en las principales API de programación: Posix Threads, OpenMPI, y OpenMP.
4. Instalar un clúster específico basado en una distribución ad hoc (Rocks).

1. Clusterización

La historia de los sistemas informáticos es muy reciente (se puede decir que comienza en la década de 1960). En un principio, eran sistemas grandes, pesados, caros, de pocos usuarios expertos, no accesibles y lentos. En la década de 1970, la evolución permitió mejoras sustanciales llevadas a cabo por tareas interactivas (*interactive jobs*), tiempo compartido (*time sharing*), terminales y con una considerable reducción del tamaño. La década de 1980 se caracteriza por un aumento notable de las prestaciones (hasta hoy en día) y una reducción del tamaño en los llamados microordenadores. Su evolución ha sido a través de las estaciones de trabajo (*workstations*) y los avances en redes (LAN de 10 Mb/s y WAN de 56 kB/s en 1973 a LAN de 1/10 Gb/s y WAN con ATM, *asynchronous transfer mode* de 1,2 Gb/s en la actualidad), que es un factor fundamental en las aplicaciones multimedia actuales y de un futuro próximo. Los sistemas distribuidos, por su parte, comenzaron su historia en la década de 1970 (sistemas de 4 u 8 ordenadores) y su salto a la popularidad lo hicieron en la década de 1990. Si bien su administración, instalación y mantenimiento es complejo, porque continúan creciendo en tamaño, las razones básicas de su popularidad son el incremento de prestaciones que presentan en aplicaciones intrínsecamente distribuidas (aplicaciones que por su naturaleza son distribuidas), la información compartida por un conjunto de usuarios, la compartición de recursos, la alta tolerancia a los fallos y la posibilidad de expansión incremental (capacidad de agregar más nodos para aumentar las prestaciones y de forma incremental). Otro aspecto muy importante en la actualidad es la posibilidad, en esta evolución, de la virtualización. Las arquitecturas cada vez más eficientes, con sistemas *multicore*, han permitido que la virtualización de sistemas se transforme en una realidad con todas las ventajas (y posibles desventajas) que ello comporta.

1.1. Virtualización

La virtualización es una técnica que está basada en la abstracción de los recursos de una computadora, llamada *Hypervisor* o VMM (*Virtual Machine Monitor*) que crea una capa de separación entre el hardware de la máquina física (*host*) y el sistema operativo de la máquina virtual (*virtual machine*, *guest*), y es un medio para crear una "versión virtual" de un dispositivo o recurso, como un servidor, un dispositivo de almacenamiento, una red o incluso un sistema operativo, donde se divide el recurso en uno o más entornos de ejecución. Esta capa de software (VMM) maneja, gestiona y administra los cuatro recursos principales de un ordenador (CPU, memoria, red y almacenamiento)

y los reparte de forma dinámica entre todas las máquinas virtuales definidas en el computador central. De este modo nos permite tener varios ordenadores virtuales ejecutándose sobre el mismo ordenador físico.

La máquina virtual, en general, es un sistema operativo completo que se ejecuta como si estuviera instalado en una plataforma de hardware autónoma.

Existen diferencias entre los diversos tipos de virtualización, de las cuales la más completa es aquella en que la máquina virtual simula un hardware suficiente para permitir la ejecución de forma aislada de un sistema operativo *guest* sin modificar y diseñado para la misma CPU (esta categoría también es llamada *hardware virtualization*).

Virtualización por hardware

Los ejemplos más comunes de virtualización por hardware son Xen, Qemu, VirtualBox, VMware Workstation/Server, Adeos, Virtual PC/Hyper-V. Algunos autores consideran una subcategoría dentro de la virtualización por hardware llamada *paravirtualization*, que es una técnica de virtualización con una interfaz de software para máquinas virtuales similar, pero no idéntica, al hardware subyacente. Dentro de esta categoría entrarían KVM, QEMU, VirtualBox y VMware, entre otros.

Enlace de interés

Para saber más sobre virtualización podéis visitar: http://en.wikipedia.org/wiki/Virtualization.
Se puede consultar una lista completa de programas de virtualización por hardware en: http://en.wikipedia.org/wiki/Comparison_of_platform_virtual_machines.

Un segundo tipo de virtualización (parcial) es lo que se denomina *Address Space Virtualization*. La máquina virtual simula múltiples instancias del entorno subyacente del hardware (pero no de todo), particularmente el *address space*. Este tipo de virtualización acepta compartir recursos y alojar procesos, pero no permite instancias separadas de sistemas operativos *guest* y se encuentra en desuso actualmente.

Una tercera categoría es la llamada *virtualización compartida* del sistema operativo, en la cual se virtualizan servidores en la capa del sistema operativo (núcleo, o *kernel*). Este método de virtualización crea particiones aisladas o entornos virtuales (VE) en un único servidor físico e instancia de SO para así maximizar los esfuerzos de administración del hardware, el software y el centro de datos. La virtualización del *Hypervisor* tiene una capa base (generalmente un núcleo de Linux) que se carga directamente en el servidor base. La siguiente capa superior muestra el hardware que debe virtualizarse para que así pueda ser asignado a las VM en que existe una copia completa de un sistema operativo y la aplicación. La diferencia entre instalar dos sistemas operativos y virtualizar dos sistemas operativos es que en el primer caso todos los sistemas operativos que tengamos instalados funcionarán de la misma manera que si estuvieran instalados en distintos ordenadores, y necesitaremos un gestor de arranque que al encender el ordenador nos permita elegir qué sistema operativo queremos utilizar. En cambio, la virtualización permite cambiar de sistema operativo como si se tratase de cualquier otro programa; sin embargo, esta agilidad tiene la desventaja de que un sistema operativo virtualizado no es tan potente como uno que ya estuviera instalado.

Enlaces de interés

Podéis encontrar referencias a las principales aportaciones de la virtualización compartida en: http://en.wikipedia.org/wiki/Operating_system-level_virtualization.
Podéis encontrar un guía rápida de instalación de Qemu [21] y su integración con KQemu y KVM para diferentes distribuciones de Debian en: http://wiki.debian.org/QEMU.
El equivalente para Xen [27] podéis encontrarlo en: http://wiki.debian.org/Xen.

1.2. Beowulf

Beowulf [23, 2, 25] es una arquitectura multiordenador que puede ser utilizada para aplicaciones paralelas/distribuidas (APD). El sistema consiste básicamente en un servidor y uno o más clientes conectados (generalmente) a través de Ethernet y sin la utilización de ningún hardware específico. Para explotar esta capacidad de cómputo, es necesario que los programadores tengan un modelo de programación distribuido que, si bien es posible mediante UNIX (socket, rpc), puede implicar un esfuerzo considerable, ya que son modelos de programación a nivel de *systems calls* y lenguaje C, por ejemplo; pero este modo de trabajo puede ser considerado de bajo nivel. Un modelo más avanzado en esta línea de trabajo son los **Posix Threads** que permiten explotar sistemas de memoria compartida y *multicores* de forma simple y fácil. La capa de software (interfaz de programación de aplicaciones, API) aportada por sistemas tales como ***Parallel Virtual Machine*** (PVM) y ***Message Passing Interface*** (MPI) facilita notablemente la abstracción del sistema y permite programar aplicaciones paralelas/distribuidas de modo sencillo y simple. La forma básica de trabajo es maestro–trabajadores (*master-workers*), en que existe un servidor que distribuye la tarea que realizarán los trabajadores. En grandes sistemas (por ejemplo, de 1.024 nodos) existe más de un maestro y nodos dedicados a tareas especiales como, por ejemplo, entrada/salida o monitorización. Otra opción no menos interesante es **OpenMP**, una API para la programación multiproceso de memoria compartida en múltiples plataformas. Esta capa de software permite añadir concurrencia a los programas escritos en C, C++ y Fortran sobre la base del modelo de ejecución *fork-join* y se compone de un conjunto de directivas de compilador, rutinas de biblioteca y variables de entorno que influencian el comportamiento en tiempo de ejecución y proporciona a los programadores una interfaz simple y flexible para el desarrollo de aplicaciones paralelas para todo tipo de plataformas.

Una de las principales diferencias entre Beowulf y un clúster de estaciones de trabajo (*cluster of workstations*, COW) es que Beowulf "se ve" como una única máquina donde se accede a los nodos remotamente, ya que no disponen de terminal (ni de teclado), mientras que un COW es una agrupación de ordenadores que pueden ser utilizados tanto por los usuarios de la COW como por otros usuarios en forma interactiva a través de su pantalla y teclado. Hay que considerar que Beowulf no es un software que transforma el código del usuario en distribuido ni afecta al núcleo del sistema operativo (como, por ejemplo, Mosix). Simplemente, es una forma de agrupación (un clúster) de máquinas que ejecutan GNU/Linux y actúan como un superordenador. Obviamente, existe gran cantidad de herramientas que permiten obtener una configuración más fácil, bibliotecas o modificaciones al núcleo para obtener mejores prestaciones, pero es posible construir un clúster Beowulf a partir de un GNU/Linux estándar y de software convencional. La construcción de un clúster Beowulf de dos nodos, por ejemplo, se puede llevar a cabo simplemente con las dos máquinas conectadas por Ethernet mediante un concentrador (*hub*), una distribución de GNU/ Linux estándar (Debian), el sistema de ar-

chivos compartido (NFS) y tener habilitados los servicios de red como rsh o ssh. En estas condiciones, se puede argumentar que se dispone de un clúster simple de dos nodos.

1.2.1. ¿Cómo configurar los nodos?

Primero se debe modificar, de cada nodo, el `/etc/hosts` para que la línea de `localhost` solo tenga el 127.0.0.1 y no incluya ningún nombre de la máquina:

Por ejemplo:
127.0.0.1 localhost
Y añadir las IP de los nodos (y para todos los nodos), por ejemplo:
192.168.0.1 pirulo1
192.168.0.2 pirulo2
...
Se debe crear un usuario (nteum) en todos los nodos, crear un grupo y añadir este usuario al grupo:
groupadd beowulf
adduser nteum beowulf
echo umask 007 >> /home/nteum/.bash_profile

Así, cualquier archivo creado por el usuario nteum o cualquiera dentro del grupo será modificable por el grupo beowulf. Se debe crear un servidor de NFS (y los demás nodos serán clientes de este NFS). Sobre el servidor hacemos:

```
mkdir /mnt/nteum
chmod 770 /mnt/nteum
chown nteum:beowulf /mnt/nteum -R
```

Ahora exportamos este directorio desde el servidor:

```
cd /etc
cat >> exports
/mnt/wolf 192.168.0.100/192.168.0.255 (rw)
<control d>
```

Se debe tener en cuenta que la red propia será 192.168.0.xxx y que es una red privada, es decir, que el clúster no se verá desde Internet y se deberán ajustar las configuraciones para que todos los nodos se vean desde todos los otros (desde los cortafuegos).

A continuación verificamos que los servicios están funcionando (tened en cuenta que el comando `chkconfig` puede no estar instalado en todas las distribuciones):

```
chkconfig -add sshd
chkconfig -add nfs
chkconfig -add rexec
chkconfig -add rlogin
chkconfig -level 3 rsh on
chkconfig -level 3 nfs on
```

```
chkconfig -level 3 rexec on
chkconfig -level 3 rlogin on
```

En caso contrario, también es posible consultar en los directorios correspondientes (`etc/rc3.d/`). Si bien estaremos en un red privada, para trabajar de forma segura es importante trabajar con ssh en lugar de rsh, por lo cual deberíamos generar las claves para interconectar en modo seguro las máquinas-usuario nteum sin `passwd`. Para eso modificamos (quitamos el comentario #), de `/etc/ssh/sshd_config` en las siguientes líneas:

```
RSAAuthentication yes
AuthorizedKeysFile .ssh/authorized_keys
```

Reiniciamos la máquina y nos conectamos como usuario nteum, ya que el clúster lo operará este usuario. Para generar las llaves:

```
ssh-keygen -b 1024 -f ∼/.ssh/id_rsa -t rsa -N ""
```

En el directorio `/home/nteum/.ssh` se habrán creado los siguientes archivos: `id_rsa y id_rsa.pub`. Se debe copiar `id_rsa.pub` en un archivo llamado `authorized_keys` en el mismo directorio. Luego modificamos los permisos con `chmod 644 ∼/.ssh/aut*` y `chmod 755 ∼/.ssh`. Puesto que solo el nodo principal se conectará a todos los restantes (y no a la inversa), necesitamos solo copiar la clave pública (`d_rsa.pub`) a cada nodo en el directorio/archivo `/home/nteum/.ssh/authorized_keys` de cada nodo. Sobre cada nodo, además, se deberá montar el NFS agregando `/etc/fstab` a la línea `pirulo1:/mnt/nteum /mnt/nteum nfs rw,hard,intr 0 0`.

A partir de aquí ya tenemos un clúster Beowulf para ejecutar aplicaciones que podrán ser PVM o MPI (como se verá en los siguientes subapartados). Sobre FC existe una aplicación (system-config-cluster) que permite configurar un clúster en base a una herramienta gráfica.

Enlace de interés

Para obtener más información consultad: http://www.redhat.com /docs/manuals/enterprise /RHEL-5-manual /clúster_Administration /index.html.

1.3. Beneficios del cómputo distribuido

¿Cuáles son los beneficios del cómputo en paralelo? Veremos esto con un ejemplo [23]. Consideremos un programa para sumar números (por ejemplo, $4 + 5 + 6 + \ldots$) llamado `sumdis.c`:

```
#include <stdio.h>

int main (int argc, char** argv){
long inicial, final, resultado, tmp;
    if (argc < 2) {
   printf (" Uso: %s N.º inicial N.º final\n",argv[0]);
     return (4); }
    else {
   inicial = atol (argv[1]);
   final = atol (argv[2]);
```

```
   resultado = 0;}
    for (tmp = inicial; tmp <= final; tmp++){resultado += tmp; };
    printf("%f\n", resultado);
    return 0;
}
```

Lo compilamos con `gcc -o sumdis sumdis.c` y si miramos la ejecución de este programa con, por ejemplo,

```
time ./sumdis 1 1000000
499941376000.000000

    real    0m0.006s
    user    0m0.004s
    sys     0m0.000s
```

se podrá observar que el tiempo en una máquina Debian 2.6.26 sobre una máquina virtual (Virtualbox) con procesador i7 es (aproximadamente) 6 milésimas de segundo (real = 0,006 y user = 0,004). Si, en cambio, hacemos desde 1 a 16 millones, el tiempo real sube hasta 0,082 s, es decir, 14 veces más, lo cual, si se considera 160 millones, el tiempo será del orden de decenas de minutos.

La idea básica del cómputo distribuido es repartir el trabajo. Así, si disponemos de un clúster de 4 máquinas (nodo1–nodo4) con un servidor y donde el fichero se comparte por NFS, sería interesante dividir la ejecución mediante rsh (no recomendable, pero como ejemplo es aceptable), de modo que el primero sume de 1 a 40.000.000, el segundo, de 40.000.001 a 80.000.000, el tercero, de 80.000.001 a 120.000.000 y el cuarto, de 120.000.001 a 160.000.000. Los siguientes comandos muestran una posibilidad. Consideramos que el sistema tiene el directorio `/home` compartido por NFS y el usuario nteum, que ejecutará el *script*, tiene adecuadamente configurado el `.rhosts` para acceder sin contraseña a su cuenta. Además, si se ha activado el `tcpd` en `/etc/inetd.conf` en la línea de `rsh`, debe existir la correspondiente en el `/etc/hosts.allow`, que permita acceder a las cuatro máquinas del clúster:

mkfifo out1 *Crea una cola fifo en /home/nteum*
./distr.sh & time cat out1 | awk '{total + = $1 } END {printf " %lf", total}'

Se ejecuta el comando `distr.sh`; se recolectan los resultados y se suman mientras se mide el tiempo de ejecución. El *shell script* `distr.sh` puede ser algo como:

```
rsh nodo1 /home/nteum/sumdis 1 40000000 > /home/nteum/out1 < /dev/null &
rsh nodo2 /home/nteum/sumdis 40000001 80000000 > /home/nteum/out1 < /dev/null &
rsh nodo3 /home/nteum/sumdis 80000001 120000000 > /home/nteum/out1 < /dev/null &
rsh nodo4 /home/nteum/sumdis 120000001 160000000 > /home/nteum/out1 < /dev/null &
```

Podremos observar que el tiempo se reduce notablemente (aproximadamente en un valor cercano a 4) y no exactamente de forma lineal, pero muy próxima.

Obviamente, este ejemplo es muy simple y solo válido para fines demostrativos. Los programadores utilizan bibliotecas que les permiten realizar el tiempo de ejecución, la creación y comunicación de procesos en un sistema distribuido (por ejemplo, PVM, MPI u OpenMP).

1.3.1. ¿Cómo hay que programar para aprovechar la concurrencia?

Existen diversas maneras de expresar la concurrencia en un programa. Las tres más comunes son:

1) Utilizando hilos (o procesos) en el mismo procesador (multiprogramación con solapamiento del cómputo y la E/S).

2) Utilizando hilos (o procesos) en sistemas *multicore*.

3) Utilizando procesos en diferentes procesadores que se comunican por medio de mensajes (MPS, *Message Passing System*).

Estos métodos pueden ser implementados sobre diferentes configuraciones de hardware (memoria compartida o mensajes) y, si bien ambos métodos tienen sus ventajas y desventajas, los principales problemas de la memoria compartida son las limitaciones en la escalabilidad (ya que todos los *cores*/procesadores utilizan la misma memoria y el número de estos en el sistema está limitado por el ancho de banda de la memoria) y, en los los sistemas de paso de mensajes, la latencia y velocidad de los mensajes en la red. El programador deberá evaluar qué tipo de prestaciones necesita, las características de la aplicación subyacente y el problema que se desea solucionar. No obstante, con los avances de las tecnologías de *multicores* y de red, estos sistemas han crecido en popularidad (y en cantidad). Las API más comunes hoy en día son Posix Threads y OpenMP para memoria compartida y MPI (en sus versiones OpenMPI o Mpich) para paso de mensajes. Como hemos mencionado anteriormente, existe otra biblioteca muy difundida para pasos de mensajes, llamada PVM, pero que la versatilidad y prestaciones que se obtienen con MPI ha dejado relegada a aplicaciones pequeñas o para aprender a programar en sistemas distribuidos. Estas bibliotecas, además, no limitan la posibilidad de utilizar hilos (aunque a nivel local) y tener concurrencia entre procesamiento y entrada/salida.

Para realizar una aplicación paralela/distribuida, se puede partir de la versión serie o mirando la estructura física del problema y determinar qué partes pueden ser concurrentes (independientes). Las partes concurrentes serán candidatas a reescribirse como código paralelo. Además, se debe considerar si es posible reemplazar las funciones algebraicas por sus versiones paralelizadas (por ejemplo, ScaLapack *Scalable Linear Algebra Package*, disponible en Debian –scalapack-pvm, mpich-test, dev, scalapack1-pvm, mpich según sean

para PVM o MPI–). También es conveniente averiguar si hay alguna aplicación similar paralela (por ejemplo, para PVM*) que pueda orientarnos sobre el modo de construcción de la aplicación paralela.

*http://www.epm.ornl.gov/pvm

Paralelizar un programa no es una tarea fácil, ya que se debe tener en cuenta la **ley de Amdahl**, que afirma que el incremento de velocidad (*speedup*) está limitado por la fracción de código (f) que puede ser paralelizado, de la siguiente manera:

$$\text{speedup} = \frac{1}{1-f}$$

Esta ley implica que con una aplicación secuencial $f = 0$ y el speedup = 1, mientras que con todo el código paralelo $f = 1$ y el speedup se hace infinito (!). Si consideramos valores posibles, un 90 % ($f = 0,9$) del código paralelo significa un speedup gual a 10, pero con $f = 0,99$ el speedup es igual a 100. Esta limitación se puede evitar con algoritmos escalables y diferentes modelos de programación de aplicación (paradigmas):

1) Maestro-trabajador: el maestro inicia a todos los trabajadores y coordina su trabajo y entrada/salida.

2) *Single Process Multiple Data* (SPMD): mismo programa que se ejecuta con diferentes conjuntos de datos.

3) Funcional: varios programas que realizan una función diferente en la aplicación.

En resumen, podemos concluir:

1) Proliferación de máquinas multitarea (multiusuario) conectadas por red con servicios distribuidos (NFS y NIS YP).

2) Son sistemas heterogéneos con sistemas operativos de tipo NOS (*Networked Operating System*), que ofrecen una serie de servicios distribuidos y remotos.

3) La programación de aplicaciones distribuidas se puede efectuar a diferentes niveles:

a) Utilizando un modelo cliente-servidor y programando a bajo nivel (*sockets*) o utilizando memoria compartida a bajo nivel (Posix Threads).

b) El mismo modelo, pero con API de "alto" nivel (OpenMP, MPI).

c) Utilizando otros modelos de programación como, por ejemplo, programación orientada a objetos distribuidos (RMI, CORBA, Agents, etc.).

1.4. Memoria compartida. Modelos de hilos (*threading*)

Normalmente, en una arquitectura cliente-servidor, los clientes solicitan a los servidores determinados servicios y esperan que estos les contesten con la mayor eficacia posible. Para sistema distribuidos con servidores con una carga muy alta (por ejemplo, sistemas de archivos de red, bases de datos centralizadas o distribuidas), el diseño del servidor se convierte en una cuestión crítica para determinar el rendimiento general del sistema distribuido. Un aspecto crucial en este sentido es encontrar la manera óptima de manejar la E/S, teniendo en cuenta el tipo de servicio que ofrece, el tiempo de respuesta esperado y la carga de clientes. No existe un diseño predeterminado para cada servicio y escoger el correcto dependerá de los objetivos y restricciones del servicio y de las necesidades de los clientes.

Las preguntas que debemos contestar antes de elegir un determinado diseño son: ¿Cuánto tiempo se tarda en un proceso de solicitud del cliente? ¿Cuántas de esas solicitudes es probable que lleguen durante ese tiempo? ¿Cuanto tiempo puede espera el cliente? ¿Cuánto afecta esta carga del servidor a las prestaciones del sistema distribuido? Además, con el avance de la tecnología de procesadores nos encontramos con que disponemos de sistemas *multicore* (múltiples núcleos de ejecución) que pueden ejecutar secciones de código independientes. Si se diseñan los programas en forma de múltiples secuencias de ejecución y el sistema operativo lo soporta (y GNU/Linux es uno de ellos), la ejecución de los programas se reducirá notablemente y se incrementarán en forma (casi) lineal las prestaciones en función de los *cores* de la arquitectura.

1.4.1. Multihilos (*multithreading*)

Las últimas tecnologías en programación para este tipo aplicaciones (y así lo demuestra la experiencia) es que los diseños más adecuados son aquellos que utilizan modelos de multihilos (*multithreading models*), en los cuales el servidor tiene una organización interna de procesos paralelos o hilos cooperarantes y concurrentes.

> Un hilo (*thread*) es una secuencia de ejecución (hilo de ejecución) de un programa, es decir, diferentes partes o rutinas de un programa que se ejecutan concurrentemente en un único procesador y accederán a los datos compartidos al mismo tiempo.

¿Qué ventajas aporta esto respecto a un programa secuencial? Consideremos que un programa tiene tres rutinas A, B y C. En un programa secuencial, la rutina C no se ejecutará hasta que se hayan ejecutado A y B. Si, en cambio, A, B y C son hilos, las tres rutinas se ejecutarán concurrentemente y, si en ellas hay

E/S, tendremos concurrencia de ejecución con E/S del mismo programa (proceso), cosa que mejorará notablemente las prestaciones de dicho programa. Generalmente, los hilos están contenidos dentro de un proceso y diferentes hilos de un mismo proceso pueden compartir algunos recursos, mientras que diferentes procesos no. La ejecución de múltiples hilos en paralelo necesita el soporte del sistema operativo y en los procesadores modernos existen optimizaciones del procesador para soportar modelos multihilo (*multithreading*).

Generalmente, existen cuatro modelos de diseño por hilos (en orden de complejidad creciente):

1) Un hilo y un cliente: en este caso el servidor entra en un bucle sin fin escuchando por un puerto y ante la petición de un cliente se ejecutan los servicios en el mismo hilo. Otros clientes deberán espera a que termine el primero. Es fácil de implementar pero solo atiende a un cliente a la vez.

2) Un hilo y varios clientes con selección: en este caso el servidor utiliza un solo hilo, pero puede aceptar múltiples clientes y multiplexar el tiempo de CPU entre ellos. Se necesita una gestión más compleja de los puntos de comunicación (*sockets*), pero permite crear servicios más eficientes, aunque presenta problemas cuando los servicios necesitan una alta carga de CPU.

3) Un hilo por cliente: es, probablemente, el modelo más popular. El servidor espera por peticiones y crea un hilo de servicio para atender a cada nueva petición de los clientes. Esto genera simplicidad en el servicio y una alta disponibilidad, pero el sistema no escala con el número de clientes y puede saturar el sistema muy rápidamente, ya que el tiempo de CPU dedicado ante una gran carga de clientes se reduce notablemente y la gestión del sistema operativo puede ser muy compleja.

4) Servidor con hilos en granja (*worker threads*): este método es más complejo pero mejora la escalabilidad de los anteriores. Existe un número fijo de hilos trabajadores (*workers*) a los cuales el hilo principal distribuye el trabajo de los clientes. El problema de este método es la elección del número de trabajadores: con un número elevado, caerán las prestaciones del sistema por saturación; con un número demasiado bajo, el servicio será deficiente (los clientes deberán esperar). Normalmente será necesario sintonizar la aplicación para trabajar con un determinado entorno distribuido.

Existe diferentes formas de expresar a nivel de programación con hilos: paralelismo a nivel de tareas o paralelismo a través de los datos. Elegir el modelo adecuado minimiza el tiempo necesario para modificar, depurar y sintonizar el código. La solución a esta disyuntiva es describir la aplicación en términos de dos modelos basados en un trabajo en concreto:

- Tareas paralelas con hilos independientes que pueden atender tareas independientes de la aplicación. Estas tareas independientes serán encapsuladas en hilos que se ejecutarán asincrónicamente y se deberán utilizar

bibliotecas como Posix Threads (Linux/Unix) o Win32 Thread API (Windows), que han sido diseñadas para soportar concurrencia nivel de tarea.

- Modelo de datos paralelos para calcular lazos intensivos; es decir, la misma operación debe repetirse un número elevado de veces (por ejemplo comparar una palabra frente a las palabras de un diccionario). Para este caso es posible encargar la tarea al compilador de la aplicación o, si no es posible, que el programador describa el paralelismo utilizando el entorno OpenMP, que es una API que permite escribir aplicaciones eficientes bajo este tipo de modelos.

Una aplicación de información personal (*Personal Information Manager*) es un buen ejemplo de una aplicación que contiene concurrencia a nivel de tareas (por ejemplo, acceso a la base de datos, libreta de direcciones, calendario, etc.). Esto podría ser en pseudocódigo:

```
Function addressBook;
Function inBox;
Function calendar;
Program PIM {
   CreateThread (addressBook);
   CreateThread (inBox);
   CreateThread (calendar); }
```

Podemos observar que existen tres ejecuciones concurrentes sin relación entre ellas. Otro ejemplo de operaciones con paralelismo de datos podría ser un corrector de ortografía, que en pseudocódigo sería: `Function SpellCheck {loop (word = 1, words_in_file) compareToDictionary (word);}`

Se debe tener en cuenta que ambos modelos (hilos paralelos y datos paralelos) pueden existir en una misma aplicación. A continuación se mostrará el código de un productor de datos y un consumidor de datos basado en Posix Threads. Para compilar sobre Linux, por ejemplo, se debe utilizar `gcc -o pc pc.c -lptread`.

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define QUEUESIZE 10
#define LOOP 20

void *producer (void *args);
void *consumer (void *args);
typedef struct { /* Estructura del buffer compartido y descriptores de threads */
int buf[QUEUESIZE]; long head, tail; int full, empty;
pthread_mutex_t *mut; pthread_cond_t *notFull, *notEmpty;
    } queue;
queue *queueInit (void);   /* Prototipo de función: inicialización del buffer */
void queueDelete (queue *q); /* Prototipo de función: Borrado del buffer*/
void queueAdd (queue *q, int in); /* Prototipo de función: insertar elemento en el buffer */
void queueDel (queue *q, int *out); /* Prototipo de función: quitar elemento del buffer */

int main () {
queue *fifo; pthread_t pro, con; fifo = queueInit ();
```

```
if (fifo ==  NULL) { fprintf (stderr, " Error al crear buffer.\n");  exit (1);  }
pthread_create (&pro, NULL, producer, fifo);   /* Creación del thread productor */
pthread_create (&con, NULL, consumer, fifo); /* Creación del thread consumidor*/
pthread_join (pro, NULL);   /* main () espera hasta que terminen ambos threads */
pthread_join (con, NULL);
queueDelete (fifo);       /* Eliminación del buffer compartido */
return 0; }               /* Fin */

void *producer (void *q) { /*Función del productor */
queue *fifo; int i;
fifo = (queue *)q;
for (i = 0; i < LOOP; i++) {    /* Inserto en el buffer elementos=LOOP*/
pthread_mutex_lock (fifo->mut);   /* Semáforo para entrar a insertar */
while (fifo->full) {
printf ("Productor: queue FULL.\n");
pthread_cond_wait (fifo->notFull, fifo->mut);  }
/* Bloqueo del productor si el buffer está lleno, liberando el semáforo mut
    para que pueda entrar el consumidor. Continuará cuando el consumidor ejecute
    pthread_cond_signal (fifo->notFull);*/
         queueAdd (fifo, i); /* Inserto elemento en el buffer */
                   pthread_mutex_unlock (fifo->mut); /* Libero el semáforo */
         pthread_cond_signal (fifo->notEmpty);/*Desbloqueo consumidor si está bloqueado*/
         usleep (100000);  /* Duermo 100 mseg para permitir que el consumidor se active */
}
return (NULL); }

void *consumer (void *q) { /*Función del consumidor */
queue *fifo; int i, d;
fifo = (queue *)q;
for (i = 0; i < LOOP; i++) { /* Quito del buffer elementos=LOOP*/
pthread_mutex_lock (fifo->mut); /* Semáforo para entrar a quitar */
while (fifo->empty) {
printf (" Consumidor: queue EMPTY.\n");
pthread_cond_wait (fifo->notEmpty, fifo->mut); }
    /* Bloqueo del consumidor si el buffer está vacío, liberando el semáforo mut
    para que pueda entrar el productor. Continuará cuando el consumidor ejecute
    pthread_cond_signal (fifo->notEmpty);*/
queueDel (fifo, &d); /* Quito elemento del buffer */
pthread_mutex_unlock (fifo->mut); /* Libero el semáforo */
pthread_cond_signal (fifo->notFull); /*Desbloqueo productor si está bloqueado*/
printf (" Consumidor: Recibido %d.\n", d);
usleep(200000);/* Duermo 200 mseg para permitir que el productor se active */
}
return (NULL); }

queue *queueInit (void) {
queue *q;
q = (queue *)malloc (sizeof (queue)); /* Creación del buffer */
if (q == NULL) return (NULL);
q->empty = 1;  q->full = 0;  q->head = 0;  q->tail = 0;
q->mut = (pthread_mutex_t *) malloc (sizeof (pthread_mutex_t));
pthread_mutex_init (q->mut, NULL); /* Creación del semáforo */
q->notFull = (pthread_cond_t *) malloc (sizeof (pthread_cond_t));
pthread_cond_init (q->notFull, NULL); /* Creación de la variable condicional notFull*/
q->notEmpty = (pthread_cond_t *) malloc (sizeof (pthread_cond_t));
pthread_cond_init (q->notEmpty, NULL);   /* Creación de la variable condicional notEmpty*/
return (q); }

void queueDelete (queue *q) {
pthread_mutex_destroy (q->mut);  free (q->mut);
pthread_cond_destroy (q->notFull); free (q->notFull);
pthread_cond_destroy (q->notEmpty); free (q->notEmpty);
free (q); }

void queueAdd (queue *q, int in) {
q->buf[q->tail] = in; q->tail++;
if (q->tail == QUEUESIZE) q->tail = 0;
if (q->tail == q->head) q->full = 1;
q->empty = 0;
return; }

void queueDel (queue *q, int *out){
```

```
*out = q->buf[q->head]; q->head++;
if (q->head == QUEUESIZE) q->head = 0;
if (q->head == q->tail) q->empty = 1;
q->full = 0;
return; }
```

1.5. OpenMP

El OpenMP (Open-Multi Processing) es un interfaz de programación de aplicaciones (API) con soporte multiplataforma para la programación en C/C++ y Fortran de procesos con uso de memoria compartida sobre plataformas Linux/Unix (y también Windows). Esta infraestructura se compone de un conjunto de directivas del compilador, rutinas de la biblioteca y variables de entorno que permiten aprovechar recursos compartidos en memoria y en tiempo de ejecución. Definido conjuntamente por un grupo de los principales fabricantes de hardware y software, OpenMP permite utilizar un modelo escalable y portátil de programación, que proporciona a los usuarios un interfaz simple y flexible para el desarrollo, sobre plataformas paralelas, de aplicaciones de escritorio hasta aplicaciones de altas prestaciones sobre superordenadores. Una aplicación construida con el modelo híbrido de la programación paralela puede ejecutarse en un ordenador utilizando tanto OpenMP como Message Passing Interface (MPI) [1].

OpenMP es una implementación multihilo, mediante la cual un hilo maestro divide la tareas sobre un conjunto de hilos trabajadores. Estos hilos se ejecutan simultáneamente y el entorno de ejecución realiza la asignación de estos a los diferentes procesadores de la arquitectura. La sección del código que está diseñada para funcionar en paralelo está marcada con una directiva de preprocesamiento que creará los hilos antes que la sección se ejecute. Cada hilo tendrá un identificador (`id`) que se obtendrá a partir de una función (`omp_get_thread_num()` en C/C++) y, después de la ejecución paralela, los hilos se unirán de nuevo en su ejecución sobre el hilo maestro, que continuará con la ejecución del programa. Por defecto, cada hilo ejecutará una sección paralela de código independiente pero se pueden declarar secciones de "trabajo compartido" para dividir una tarea entre los hilos, de manera que cada hilo ejecute parte del código asignado. De esta forma, es posible tener en un programa OpenMP paralelismo de datos y paralelismo de tareas conviviendo conjuntamente.

Los principales elementos de OpenMP son las sentencias para la creación de hilos, la distribución de carga de trabajo, la gestión de datos de entorno, la sincronización de hilos y las rutinas a nivel de usuario. OpenMP utiliza en C/C++ las directivas de preprocesamiento conocidas como *pragma* (`#pragma omp <resto del pragma>`) para diferentes construcciones. Así por ejemplo, `omp parallel` se utiliza para crear hilos adicionales para ejecutar el trabajo indicado en la sentencia paralela donde el proceso original es el hilo maestro (`id=0`). El conocido programa que imprime "Hello, world" utilizando OpenMP y multihilos es*:

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[]){
   #pragma omp parallel
   printf("Hello, world.\n");
   return 0;}
```

***Compilad con** `gcc -fopenmp -o hell hello.c`**; se debe tener instalada la biblioteca GCC OpenMP (GOMP)**
`apt-get install libgomp1`

Para especificar *work-sharing constructs* se utiliza:

- **omp for o omp do**: reparte las iteraciones de un lazo en múltiples hilos.
- **sections**: asigna bloques de código independientes consecutivos a diferentes hilos.
- **single**: especifica que el bloque de código será ejecutado por un solo hilo con una sincronización (*barrier*) al final del mismo.
- **master**: similar a `single`, pero el código del bloque será ejecutado por el hilo maestro y no hay *barrier* implicado al final.

Por ejemplo, para inicializar los valores de un *array* en paralelo utilizando hilos para hacer una porción del trabajo (compilad, por ejemplo, con `gcc -fopenmp -o init2 init2.c`):

```
#include <stdio.h>
#include <omp.h>
#define N 1000000
int main(int argc, char *argv[]) {
   float a[N]; long i;
    #pragma omp parallel for
   for (i=0;i<N;i++)  a[i]= 2*i;
return 0;
}
```

Si ejecutamos con el *pragma* y después lo comentamos y calculamos el tiempo de ejecución (`time ./init2`), vemos que el tiempo de ejecución pasa de 0.006 s a 0.008 s, lo que muestra la utilización del *dualcore* del procesador.

Ya que OpenMP es un modelo de memoria compartida, muchas variables en el código son visibles para todos los hilos por defecto. Pero a veces es necesario tener variables privadas y pasar valores entre bloques secuenciales del código y bloques paralelos, por lo cual es necesario definir atributos a los datos (*data clauses*) que permitan diferentes situaciones:

- **shared**: los datos en la región paralela son compartidos y accesibles por todos los hilos simultáneamente.
- **private**: los datos en la región paralela son privados para cada hilo, y cada uno tiene una copia de ellos sobre una variable temporal.

- **default**: permite al programador definir cómo serán los datos dentro de la región paralela (`shared`, `private` o `none`).

Otro aspecto interesante de OpenMP son las directivas de sincronización:

- **critical section**: el código enmarcado será ejecutado por hilos, pero solo uno por vez (no habrá ejecución simultanea) y s emantiene la exclusión mutua.
- **atomic**: similar a `critical section`, pero avisa al compilador para que use instrucciones de hardware especiales de sincronización y así obtener mejores prestaciones.
- **ordered**: el bloque es ejecutado en el orden como si de un programa secuencial se tratara.
- **barrier**: cada hilo espera que los restantes hayan acabado su ejecución (implica sincronización de todos los hilos al final del código).
- **nowait**: especifica que los hilos que terminen el trabajo asignado pueden continuar.

Además, OpenMP provee de sentencias para la planificación (*scheduling*) del tipo `schedule(type, chunk)` (donde el tipo puede ser `static`, `dynamic` o `guided`) o proporciona control sobre la sentencias `if`, lo que permitirá definir si se paraleliza o no en función de si la expresión es verdadera o no. OpenMP también proporciona un conjunto de funciones de biblioteca, como por ejemplo:

- **omp_set_num_threads**: define el número de hilos a usar en la siguiente región paralela.
- **omp_get_num_threads**: obtiene el número de hilos que se están usando en una región paralela.
- **omp_get_max_threads**: obtiene la máxima cantidad posible de hilos.
- **omp_get_thread_num**: devuelve el número del hilo.
- **omp_get_num_procs**: devuelve el máximo número de procesadores que se pueden asignar al programa.
- **omp_in_parallel**: devuelve un valor distinto de cero si se ejecuta dentro de una región paralela.

Veremos a continuación algunos ejemplos simples (compilad con la instrucción `gcc -fopenmp -o out_file input_file.c`):

```
/* Programa simple multihreading con OpenMP */
#include <omp.h>
int main()  {
     int iam =0, np = 1;

     #pragma omp parallel private(iam, np)
     #if defined (_OPENMP)
         np = omp_get_num_threads();
         iam = omp_get_thread_num();
     #endif
```

```
        printf("Hello from thread %d out of %d \n", iam, np);
        }
    }

    /* Programa simple con Threads anidados con OpenMP */
    #include <omp.h>
    #include <stdio.h>
        main(){
            int x=0,nt,tid,ris;

            omp_set_nested(2);
            ris=omp_get_nested();
            if (ris) printf("Paralelismo anidado activo %d\n", ris);
            omp_set_num_threads(25);
            #pragma omp parallel private (nt,tid) {
                tid = omp_get_thread_num();
                printf("Thread %d\n",tid);
                nt = omp_get_num_threads();
                if (omp_get_thread_num()==1)
                printf("Número de Threads: %d\n",nt);
                }
    }

    /* Programa simple de integración con OpenMP */
    #include <omp.h>
    #include <stdio.h>
    #define N 100
        main() {
       double local, pi=0.0, w; long i;
       w = 1.0 / N;
       #pragma omp parallel private(i, local)
           {
           #pragma omp single
           pi = 0.0;
           #pragma omp for reduction(+: pi)
           for (i = 0; i < N; i++) {
             local = (i + 0.5)*w;
             pi = pi + 4.0/(1.0 + local*local);
             printf ("Pi: %f\n",pi);
               }
           }
        }

    /* Programa simple de reduccióncon OpenMP */
    #include <omp.h>
    #include <stdio.h>
    #define NUM_THREADS 2
        main () {
       int i; double ZZ, res=0.0;
            omp_set_num_threads(NUM_THREADS);
        #pragma omp parallel for reduction(+:res) private(ZZ)
           for (i=0; i< 1000; i++) {
         ZZ = i*i;
         res = res + ZZ;
         printf("ZZ: %f, res: %f\n", ZZ, res);}
                }
           }
```

Nota

Se recomienda ver también los ejemplos que se pueden encontrar en la referencia [17].

1.6. MPI, *Message Passing Interface*

La definición de la API de MPI [15, 16] ha sido el trabajo resultante del MPI Forum (MPIF), que es un consorcio de más de 40 organizaciones. MPI tiene influencias de diferentes arquitecturas, lenguajes y trabajos en el mundo del paralelismo, como por ejemplo: WRC (Ibm), Intel NX/2, Express, nCUBE, Vertex, p4, Parmac y contribuciones de ZipCode, Chimp, PVM, Chamaleon, PICL.

El principal objetivo de MPIF fue diseñar una API, sin relación particular con ningún compilador ni biblioteca, y que permitiera la comunicación eficiente (*memory-to-memory copy*), cómputo y comunicación concurrente y descarga de comunicación, siempre y cuando exista un coprocesador de comunicaciones. Además, se pedía que soportara el desarrollo en ambientes heterogéneos, con interfaz C y F77 (incluyendo C++, F90), donde la comunicación fuera fiable y los fallos, resueltos por el sistema. La API también debía tener interfaz para diferentes entornos (PVM, NX, Express, p4, etc.), disponer una implementación adaptable a diferentes plataformas con cambios insignificantes y que no interfiera con el sistema operativo (*thread-safety*). Esta API fue diseñada especialmente para programadores que utilizaran el *Message Passing Paradigm* (MPP) en C y F77, para aprovechar su característica más relevante: la portabilidad. El MPP se puede ejecutar sobre máquinas multiprocesador, redes de estaciones de trabajo e incluso sobre máquinas de memoria compartida. La versión MPI1 (la versión más extendida) no soporta creación (*spawn*) dinámica de tareas, pero MPI2 (versión que implementa OpenMPI) sí que lo hace.

Muchos aspectos han sido diseñados para aprovechar las ventajas del hardware de comunicaciones sobre SPC (*scalable parallel computers*) y el estándar ha sido aceptado en forma mayoritaria por los fabricantes de hardware en paralelo y distribuido (SGI, SUN, Cray, HPConvex, IBM, etc.). Existen versiones libres (por ejemplo, Mpich y openMPI) que son totalmente compatibles con las implementaciones comerciales realizadas por los fabricantes de hardware e incluyen comunicaciones punto a punto, operaciones colectivas y grupos de procesos, contexto de comunicaciones y topología, soporte para F77 y C y un entorno de control, administración y *profiling*.

Pero existen también algunos puntos que pueden presentar algunos problemas en determinadas arquitecturas como son la memoria compartida, la ejecución remota, las herramientas de construcción de programas, la depuración, el control de hilos, la administración de tareas y las funciones de entrada/salida concurrentes (la mayor parte de estos problemas de falta de herramientas están resueltos en la versión 2 de la API MPI2). El funcionamiento en MPI1, al no tener creación dinámica de procesos, es muy simple, ya que los procesos/tareas se ejecutan de manera autónoma con su propio código estilo MIMD (*Multiple Instruction Multiple Data*) y comunicándose vía llamadas MPI. El código puede ser secuencial o multihilo (concurrentes) y MPI funciona en modo *threadsafe*, es decir, se pueden utilizar llamadas a MPI en hilos concurrentes, ya que las llamadas son reentrantes.

Para la instalación de MPI se recomienda utilizar la distribución, ya que su compilación es sumamente compleja (por las dependencias que necesita de otros paquetes). Debian incluye la versión Mpich 1.2.x y también OpenMPI 1.2.x (también se incluye la versión MPD –versión de *multipurpose daemon* que incluye soporte para la gestión y control de procesos en forma escalable–). La mejor elección será OpenMPI, ya que combina las tecnologías y los recursos de varios otros proyectos (FT-MPI, LA-MPI, LAM/MPI y PACX-MPI) y

soporta totalmente el estándar MPI-2. Entre otras características de OpenMPI tenemos: es conforme a MPI-2, *thread safety* & *concurrency*, creación dinámica de procesos, alto rendimiento y gestión de trabajos tolerantes a fallos, instrumentación en tiempo de ejecución, *job schedulers*, etc. Para ello se debe instalar los paquetes `libopenmpi-dev`, `libopenmpi1`, `openmpi-bin`, `openmpi-common` y `openmpi-doc` (recordad que hay que tener activado el *non-free* como fuente de software en Debian). Además, esta distribución incluye otra implementación de MPI llamada LAM (paquetes `lam*` y documentación en `/usr/doc/lam-runtime/release.html`). Las implementaciones son equivalentes desde el punto de vista de MPI pero se gestionan de manera diferente.

1.6.1. Configuración de un conjunto de máquinas para hacer un clúster adaptado a OpenMPI

Para la configuración de un conjunto de máquinas para hacer un clúster adaptado a OpenMPI [29] se han de seguir los siguientes pasos:

1) Hay que tener las máquinas "visibles" (por ejemplo a través de un `ping`) a través de TCP/IP (IP pública/privada).

2) Es recomendable que todas las máquinas tengan la misma arquitectura de procesador, así es más fácil distribuir el código, con versiones similares de Linux (a ser posible con el mismo tipo de distribución).

3) Se debe generar un mismo usuario (por ejemplo mpiuser) en todas las máquinas y el mismo directorio `$HOME` montado por NFS.

4) Nosotros llamaremos a las máquinas como `slave1`, `slave2`, etc. (ya que luego resultará más fácil hacer las configuraciones) pero se puede llamar a las máquinas como cada uno prefiera.

5) Una de las máquinas será el maestro y las restantes, *slaveX*.

6) Se debe instalar en todos los nodos (supongamos que tenemos Debian): openmpi-bin, openmpi-common, libopenmpi1, libopenmpi-dev (esta última no es necesaria para los nodos esclavos). Hay que verificar que en todas las distribuciones se trabaja con la misma versión de OpenMPI.

7) En Debian los ejecutables están en `/usr/bin` pero si están en un *path* diferente deberá agregarse a mpiuser y también verificar que `LD_LIBRARY_PATH` apunta a `/usr/lib`.

8) En cada nodo esclavo debe instalarse el SSH server (instalad el paquete `openssh-server`) y sobre el maestro, el cliente (paquete `openssh-client`).

9) Se deben crear las claves públicas y privadas haciendo `ssh-keygen -t dsa` y copiar

```
cp /home/mpiuser/.ssh/id_dsa.pub /home/mpiuser/.ssh/authorized_keys
```

Si no se comparte el directorio hay que asegurar que cada esclavo conoce que el usuario mpiuser se puede conectar por ejemplo haciendo desde el *slaveX*

```
scp /home/mpiuser/.ssh/id_dsa.pub mpiuser@slave1:.ssh/authorized_keys
```

y cambiar los permisos

```
chmod 700 /home/mpiuser/.ssh; chmod 600 /home/mpiuser/.ssh/authorized_keys
```

Se puede verificar que esto funciona haciendo, por ejemplo, `ssh slave1`.

10) Se debe configurar la lista de las máquinas sobre las cuales se ejecutará el programa, por ejemplo `/home/mpiuser/.mpi_hostfile` y con el siguiente contenido:

```
# The Hostfile for Open MPI
# The master node, slots=2 is used because it is a dual-processor machine.
localhost slots=2
# The following slave nodes are single processor machines:
slave1
slave2
slave3
```

11) OpenMPI permite utilizar diferentes lenguajes, pero aquí utilizaremos C. Para ello hay que ejecutar sobre el maestro `mpicc testprogram.c`. Si se desea ver que incorpora mpicc, se puede hacer `mpicc -showme`.

12) Se debe verificar que los *slaveX* no piden ninguna contraseña (podemos indicar la contraseña para la llave ssh con `ssh-add ~/.ssh/id_dsa`).

13) Para ejecutar en local podríamos hacer `mpirun -np 2 ./myprogram` y para ejecutar sobre los nodos remotos (por ejemplo 5 procesos) `mpirun -np 2 -hostfile ./mpi_hostfile ./myprogram`.

Sobre Debian se puede instalar el paquete `update-cluster` para ayudar en su administración (hay que ir con cuidado, ya que este paquete sobrescribe archivos importantes). Es importante notar que `np` es el número de procesos o procesadores en que se ejecutará el programa y se puede poner el número que se desee, ya que OpenMPI intentará distribuir los procesos de forma equilibrada entre todas las máquinas. Si hay más procesos que procesadores, OpenMPIMpich utilizará las características de intercambio de tareas de GNU/Linux para simular la ejecución paralela. A continuación se verán dos ejemplos: Srtest es un programa simple para establecer comunicaciones entre procesos punto a punto, y cpi calcula el valor del número π de forma distribuida (por integración).

```
/* Srtest Progam */
#include "mpi.h"
#include <stdio.h>
#include <string.h>
#define BUFLEN 512
```

```
int main(int argc, char *argv[]){
    int myid, numprocs, next, namelen;
    char buffer[BUFLEN], processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Status status;
    MPI_Init(&argc,&argv);  /* Debe ponerse antes de otras llamadas MPI, siempre */
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);    /*Integra el proceso en un grupo de comunicaciones*/
    MPI_Get_processor_name(processor_name,&namelen); /*Obtiene el nombre del procesador*/

    fprintf(stderr,"Proceso %d sobre %s\n", myid, processor_name);
    fprintf(stderr,"Proceso %d de %d\n", myid, numprocs);
    strcpy(buffer,"hello there");
    if (myid == numprocs-1) next = 0;
    else next = myid+1;

    if (myid == 0) { /*Si es el inicial, envía string de buffer*/
        printf("%d sending '%s' \n",myid,buffer);fflush(stdout);
    MPI_Send(buffer, strlen(buffer)+1, MPI_CHAR, next, 99, MPI_COMM_WORLD);
        /*Blocking Send, 1:buffer, 2:size, 3:tipo, 4:destino, 5:tag, 6:contexto*/
    printf("%d receiving \n",myid);fflush(stdout);
    MPI_Recv(buffer, BUFLEN, MPI_CHAR, MPI_ANY_SOURCE, 99, MPI_COMM_WORLD,&status);
    printf("%d received '%s' \n",myid,buffer);fflush(stdout);
       /* mpdprintf(001,"%d receiving \n",myid); */
        }
    else {
       printf("%d receiving  \n",myid);fflush(stdout);
   MPI_Recv(buffer, BUFLEN, MPI_CHAR, MPI_ANY_SOURCE, 99, MPI_COMM_WORLD,&status);
       /* Blocking Recv, 1:buffer, 2:size, 3:tipo, 4:fuente, 5:tag, 6:contexto, 7:status*/
   printf("%d received '%s' \n",myid,buffer);fflush(stdout);
   /* mpdprintf(001,"%d receiving \n",myid); */
   MPI_Send(buffer, strlen(buffer)+1, MPI_CHAR, next, 99, MPI_COMM_WORLD);
   printf("%d sent '%s' \n",myid,buffer);fflush(stdout);
       }
    MPI_Barrier(MPI_COMM_WORLD); /*Sincroniza todos los procesos*/
    MPI_Finalize(); /*Libera los recursos y termina*/
    return (0);
    }
```

```
/* CPI Program */
#include "mpi.h"
#include <stdio.h>
#include <math.h>
double f( double );
double f( double a) { return (4.0 / (1.0 + a*a)); }
int main( int argc, char *argv[] ) {
    int done = 0, n, myid, numprocs, i;
    double PI25DT = 3.14159265358979323846264643;
    double mypi, pi, h, sum, x; double startwtime = 0.0, endwtime;
    int namelen; char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs); /*Indica el número de procesos en el grupo*/
    MPI_Comm_rank(MPI_COMM_WORLD,&myid); /*Id del proceso*/
    MPI_Get_processor_name(processor_name,&namelen); /*Nombre del proceso*/
    fprintf(stderr, "Proceso %d sobre %s\n", myid, processor_name);
    n = 0;
    while (!done) {
        if (myid ==0) { /*Si es el primero...*/
            if (n ==0) n = 100; else n = 0;
            startwtime = MPI_Wtime();} /* Time Clock */
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD); /*Broadcast al resto*/
            /*Envía desde el 4 arg. a todos los procesos del grupo Los restantes que no son 0
            copiarán el buffer desde 4 o arg -proceso 0-*/
            /*1:buffer, 2:size, 3:tipo, 5:grupo */
        if (n == 0) done = 1;
        else {h = 1.0 / (double) n;
            sum = 0.0;
            for (i = myid + 1; i <= n; i += numprocs) {
                x = h * ((double)i - 0.5); sum += f(x); }
            mypi = h * sum;
            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);
```

```
                /* Combina los elementos del Send Buffer de cada proceso del grupo usando la
                operación MPI_SUM y retorna el resultado en el Recv Buffer. Debe ser llamada
                por todos los procesos del grupo usando los mismos argumentos*/
                /*1:sendbuffer, 2:recvbuffer, 3:size, 4:tipo, 5:oper, 6:root, 7:contexto*/
            if (myid == 0){ /*Sólo el P0 imprime el resultado*/
                printf("Pi es aproximadamente %.16f, el error es %.16f\n", pi,fabs(pi - PI25DT));
                endwtime = MPI_Wtime();
                printf("Tiempo de ejecución = %f\n", endwtime-startwtime); }
            }
        }
        MPI_Finalize(); /*Libera recursos y termina*/
        return 0;
    }
```

Para visualizar la ejecución de un código paralelo/distribuido en MPI existe una aplicación llamada XMPI (en Debian xmpi) que permite "ver", durante la ejecución, el estado de la ejecución de las aplicaciones sobre MPI. También es posible instalar una biblioteca, `libxmpi4`, `libxmpi4-dev`, que implementa el protocolo XMPI para analizar gráficamente programas MPI con más detalles que los ofrecidos por xmpi.

1.7. Rocks Cluster

Rocks Cluster es una distribución de Linux para clústers de computadores de alto rendimiento. Las versiones actuales de Rocks Cluster están basadas en CentOS y, como instalador, Anaconda con ciertas modificaciones, que simplifica la instalación "en masa" en muchos ordenadores. Rocks Cluster incluye muchas herramientas (tales como MPI) que no forman parte de CentOS pero son los componentes integrales que hacen un grupo de ordenadores en un clúster. Las instalaciones pueden personalizarse con paquetes de software adicionales, utilizando CD especiales (llamados *roll CD*). Los *rolls* extienden el sistema integrando automáticamente los mecanismos de gestión y empaquetamiento usados por el software básico, y simplifican ampliamente la instalación y configuración de un gran número de computadores. Se han creado una gran cantidad de *rolls*, como el SGE roll, el Condor roll, el Lustre roll, el Java roll y el Ganglia roll. Rocks Cluster es una de las distribuciones más empleadas en el ámbito de clústers, por su facilidad de instalación e incorporación de nuevos nodos e incorpora gran cantidad de programas para el mantenimiento y monitorización del clúster.

Enlaces de interés

Para una lista detallada de las herramientas incluidas en Rocks Cluster, consultad: http://www.rocksclusters.org/roll-documentation/base/5.4/x8109.html.

Las principales características de Rocks Cluster son:

1) Facilidad de instalación, ya que solo es necesario completar la instalación de un nodo llamado *nodo maestro* (*frontend*), el resto se instala con Avalache que es un programa P2P que lo hace de forma automática y evita tener que instalar los nodos uno a uno.

2) Disponibilidad de programas (conjunto muy amplio de programas que no es necesario compilar y transportar) y facilidad de mantenimiento (solo se debe mantener el nodo maestro).

3) Diseño modular y eficiente, pensado para minimizar el tráfico de red y utilizar el disco duro propio de cada nodo para solo compartir la información mínima e imprescindible.

La instalación se puede seguir paso a paso desde el sitio web de la distribución [24] y los autores garantizan que no se tarda más de una hora para una instalación básica. Rocks Cluster permite, en la etapa de instalación, diferentes módulos de software, los *rolls*, que contienen todo lo necesario para realizar la instalación y la configuración de sistema con estas nuevas "adiciones" de forma automática y, además, decidir en el *frontend* cómo será la instalación en los nodos esclavos, qué *rolls* estarán activos y qué arquitectura se usará. Para el mantenimiento, incluye un sistema de copia de respaldo del estado, llamado Roll Restore. Este *roll* guarda los archivos de configuración y *scripts* (e incluso se pueden añadir los archivos que se desee).

1.7.1. Guía rápida de instalación

Este es un resumen breve de la instalación propuesta en [24] para la versión 5.4 y se parte de la base que el *frontend* tiene (mínimo) 20 GB de disco duro, 1 GB de RAM, arquitectura x86-64 y 2 interfaces de red; para los nodos 20 GB de disco duro, 512 MB de RAM y 1 interfaz de red. Después de obtener el disco de *kernel boot*, seguimos los siguientes pasos:

Enlaces de interés

Se puede descargar el disco de *kernel boot* de: http://www.rocksclusters.org/wordpress/?page_id=80

1) Arrancamos el *frontend* y veremos una pantalla en la cual introducimos `build` y nos preguntará la configuración de la red (IPV4 o IPV6).

2) El siguiente paso es seleccionar los `rolls` (por ejemplo seleccionando los "CD/DVD-based Roll" y vamos introduciendo los siguientes *rolls*) y marcar en las sucesivas pantallas cuáles son los que deseamos.

3) La siguiente pantalla nos pedirá información sobre el clúster (es importante definir bien el nombre de la máquina, *Fully-Qualified Host Name*, ya que en caso contrario fallará la conexión con diferentes servicios) y también información para la red privada que conectará el *frontend* con los nodos y la red pública (por ejemplo, la que conectará el *frontend* con Internet) así como DNS y pasarelas.

4) A continuación se solicitará la contraseña para el root, la configuración del servicio de tiempo y el particionado del disco (se recomienda escoger "auto").

5) Después de formatear los discos, solicitará los CD de *rolls* indicados e instalará los paquetes y hará un *reboot* del *frontend*.

6) Para instalar los nodos se debe entrar como root en el *frontend* y ejecutar `insert-ethers`, que capturará las peticiones de DHCP de los nodos y los agregará a la base de datos del *frontend*, y seleccionar la opción "Compute" (por defecto, consultad la documentación para las otras opciones).

7) Encendemos el primer nodo y en el *boot order* de la BIOS generalmente se tendrá CD, PXE (Network Boot), Hard Disk, (si el ordenador no soporta PXE, entonces haced el *boot* del nodo con el *Kernel Roll CD*). En el *frontend* se verá la petición y el sistema lo agregará como `compute-0-0` y comenzará la descarga e instalación de los archivos. Si la instalación falla, se deberán reiniciar los servicios httpd, mysqld y autofs en el *frontend*.

8) A partir de este punto se puede monitorizar la instalación ejecutando la instruacción `rocks-console`, por ejemplo con `rocks-console compute-0-0`. Después de que se haya instalado todo, se puede salir de `insert-ethers` apretando la tecla "F8". Si se dispone de dos *racks*, después de instalado el primero se puede comenzar el segundo haciendo `insert-ethers -cabinet=1` los cuales recibirán los nombres `compute-1-0`, `compute-1-1`, etc.

9) A partir de la información generada en consola por `rocks list host`, generaremos la información para el archivo `machines.conf`, que tendrá un aspecto como:

```
nteum slot=2
compute-0-0 slots=2
compute-0-1 slots=2
compute-0-2 slots=2
compute-0-3 slots=2
```

y que luego deberemos ejecutar con

```
mpirun -np 10 -hostfile ./machines.conf ./mpi_program_to_execute.
```

1.8. Monitorización del clúster

1.8.1. Ganglia

Ganglia [8] es una herramienta que permite monitorizar de forma escalable y distribuida el estado de un clúster. Muestra al usuario las estadísticas de forma remota (por ejemplo, los promedios de carga de la CPU o la utilización de la red) de todas las máquinas que conforman el clúster basándose en un diseño jerárquico y utilizando *multicast* para escuchar/anunciar el protocolo con el fin de controlar el estado dentro de los grupos y conexiones punto a punto entre los nodos del clúster para intercambiar el estado. Ganglia utiliza XML para la representación de datos, XDR para el transporte compacto y portátil de datos y RRDtool para almacenamiento de datos y visualización. El sistema se compone de dos *daemons* (gmond y gmetad), una página de visualización basada en PHP y algunas herramientas de utilidad.

Gmond es un *daemon* multihilo que se ejecuta en cada nodo del clúster que se desea supervisar (no es necesario tener un sistema de archivos NFS o base de datos ni mantener cuentas especiales de los archivos de configuración). Las tareas de Gmond son: monitorizar los cambios de estado en el *host*, enviar

los cambios pertinentes, escuchar el estado de otros nodos (a través de un canal *unicast* o *multicast*) y responder a las peticiones de un XML del estado del clúster. La federación de los nodos se realiza con un árbol de conexiones punto a punto entre los nodos determinados (representativos) del clúster para agregar el estado de los restantes nodos. En cada nodo del árbol se ejecuta el daemon **Gmetad** que periódicamente solicita los datos de los restantes nodos, analiza el XML, guarda todos los parámetros numéricos y exporta el XML agregado por un socket TCP. Las fuentes de datos pueden ser *daemons* gmond, en representación de determinados grupos, u otros *daemons* gmetad, en representación de conjuntos de grupos. Finalmente la web de Ganglia proporciona una vista de la información recogida de los nodos del clúster en tiempo real. Por ejemplo, se puede ver la utilización de la CPU durante la última hora, día, semana, mes o año y muestra gráficos similares para el uso de memoria, uso de disco, estadísticas de la red, número de procesos en ejecución y todos los demás indicadores de Ganglia.

Instalación de Ganglia (sobre Debian)

En este caso haremos la instalación sobre un Debian Lenny, pero es similar para otras distribuciones:

1) Ganglia usa RRDTool para generar y mostrar los gráficos, con lo que debemos ejecutar `apt-get install rrdtool librrds-perl librrd2-dev` y para visualizar diagramas de secciones, `apt-get install php5-gd`.

2) Excepto por el *frontend*, Ganglia está incluido en el repositorio Debian y sobre el servidor web debemos instalar `apt-get install ganglia-monitor gmetad`. Sobre todos los otros nodos solo se necesita tener instalado el paquete `ganglia-monitor`.

3) Para el *frontend* es necesario compilar el paquete entero de Ganglia de la página web*, por ejemplo haciendo dentro del directorio de root

***http://ganglia.info/?page_id=66**

```
wget http://downloads.sourceforge.net/ganglia/ganglia-3.1.7.tar.gz,
```

cambiando la version según corresponda, y luego hacer

```
tar xvzf ganglia*.tar.gz.
```

4) Antes de configurar el paquete fuente (y dependiendo de la instalación que se tenga) se deberán incluir a través del apt-get o Synaptic las siguients bibliotecas: `libapr1-dev`, `libconfuse-dev`, `libpcre3-dev`.

5) Dentro del directorio creado (`$HOME/ganglia-3.1.7/` por ejemplo) ejecutad

```
./configure -prefix=/opt/ganglia -enable-gexec -with-gmetad -sysconfdir=/etc/ganglia
```

o similar (véase `./configure -help` para las diferentes opciones) y en este caso utilizaremos el directorio `/opt` para instalarlo. Solucionad los diferentes problemas que pueda dar el `configure` por falta de bibliotecas necesarias (el procedimiento es fácil mirando la biblioteca faltante e instalarla a traves del apt-get o Synaptic). Cuando el `configure` termine, mostrará algo como:

```
Welcome to..
     ______                    ___
    / ____/___ _____  ____ _/ (_)___ _
   / / __/ __ '/ __ \/ __ '/ / / __ '/
  / /_/ / /_/ / / / / /_/ / / / /_/ /
  \____/\__,_/_/ /_/\__, /_/_/\__,_/
                   /____/
Copyright (c) 2005 University of California, Berkeley
Version: 3.1.7 Library: Release 3.1.7 0:0:0
Type "make" to compile.
```

6) Se debe hacer un `make` y luego un `make install` (se podrá comprobar que se ha instalado en `/opt/ganglia`).

7) Finalmente se debe configurar el servidor web haciendo primero `mkdir /var/www/ganglia`, copiando el contenido del directorio `./web` al directorio del servidor `cp web/* /var/www/ganglia` y haciendo la configuración del directorio adecuado en Apache. Para ello, debéis buscar en el archivo `/etc/apache2/sites-enabled/000-default` la sección de código del directorio `/var/www` y agregad una redirección. Esta sección deberá quedar como:

```
<Directory /var/www/>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride None
    Order allow,deny
    allow from all
    RedirectMatch ^/$ /ganglia/
</Directory>
```

A continuación solo hace falta reiniciar Apache con `/etc/init.d/apache2 restart` y a través de un navegador ir a `http://localhost/ganglia` y tendremos la visualización del nodo local!

8) Teniendo en cuenta que ya hemos instalado el *Ganglia meta daemon gmetad* sobre el nodo de cabecera en el segundo paso (`apt-get install gmetad`) deberemos configurar las opciones básicas en `/etc/gmetad.conf`. Los cambios son pocos y las opciones están comentadas por defecto: `authority`, para indicar la URL (por ejemplo `host.dominio.org/ganglia`), sobre todo si se está detrás de un cortafuegos con NAT a través de `iptables`. `trusted_hosts`, si el servidor web tiene múltiples dominios, podrían ser listados en esta opción (en caso contrario, dejadlo vacío). `gridname` es el nombre con el cual se quiere visualizar la instalación. `rrd_rootdir` si desea cambiar el destino donde se almacenan los datos ya que Ganglia necesita almacenar gran cantidad de ellos (el destino por defecto es `/var/lib/ganglia/rrds`). Finalmente hay que reiniciar gmetad con `/etc/init.d/gmetad restart`.

9) En cada *host* que se desee monitorizar se deberá ejecutar el monitor gmon incluyendo el *host* que ejecuta gmetad si también se desea monitorizar. Para ello es encesario instalar en cada nodo `apt-get install ganglia-monitor` y modificar el archivo `/etc/gmond.conf` para que se pueda conectar al servidor que recogerá los datos. Editad los siguientes valores: `name` es el nombre del clúster al cual está asociado este nodo. `mcast_if` sirve para seleccionar qué interfaz de red usará el nodo para conectarse, si es que tiene múltiples. `num_nodes` es el número de nodos en el clúster. Finalmente, hay que reiniciar gmon con `/etc/init.d/ganglia-monitor restart`.

10) Después de haber configurado todos los nodos, el gmetad sobre el servidor del web debe ser reiniciado con `/etc/init.d/gmetad restart` y esperar unos segundo para ver los cambios en los nodos.

1.8.2. Cacti

Cacti [3] es una solución para la visualización de estadísticas de red y fue diseñada para aprovechar el poder de almacenamiento y la funcionalidad de generar gráficas que posee RRDtool. Esta herramienta, desarrollada en PHP, provee diferentes formas de visualización, gráficos avanzados y dispone de una una interfaz de usuario fácil de usar, que la hacen interesante tanto para redes LAN como para redes complejas con cientos de dispositivos.

Su instalación es simple y requiere previamente tener instalado MySQL. Luego, haciendo `apt-get install cacti` se instalará el paquete y al final nos pedirá si queremos instalar la interfaz con la base de datos, solicitándonos el nombre de usuario y contraseña de la misma y la contraseña del usuario admin de la interfaz de Cacti. A continuación deberemos copiar el sitio de Cacti en el directorio `/var/www` haciendo `cp -r /usr/share/cacti/site /var/www/cacti` y después conectarnos a `http://localhost/cacti`. Al inicio nos preguntará por ajustes en la instalación, nos pedirá que cambiemos la contraseña y nos mostrará a continuación la consola de administración para crear los gráficos que deseemos y ajustar mediante la pestaña “Settings” un conjunto de opciones, entre ellas la frecuencia de actualización.

1.9. Introducción a la metacomputación o la computación distribuida o en rejilla

Los requerimientos de cómputo necesarios para ciertas aplicaciones son tan grandes que requieren miles de horas para poder ejecutarse en entornos de clústers. Tales aplicaciones han dado lugar a la generación de ordenadores virtuales en red, metacomputación (*metacomputing*) o computación distribuida o en rejilla (*grid computing*). Esta tecnología ha permitido conectar entornos de ejecución, redes de alta velocidad, bases de datos, instrumentos, etc., distribuidos geográficamente. Esto permite obtener una potencia de procesamiento

que no sería económicamente posible de otra forma y con excelentes resultados. Ejemplos de su aplicación son experimentos como el I-WAY Networking (que conecta superordenadores de 17 sitios diferentes) en América del Norte, DataGrid o CrossGrid en Europa y IrisGrid en España. Estos metaordenadores, u ordenadores en rejilla, tienen mucho en común con los sistemas paralelos y distribuidos (SPD), pero también difieren en aspectos importantes. Si bien están conectados por redes, estas pueden ser de diferentes características, no se puede asegurar el servicio y están localizadas en dominios diferentes. El modelo de programación y las interfaces deben ser radicalmente diferentes (con respecto a la de los sistemas distribuidos) y adecuadas para el cómputo de altas prestaciones. Al igual que en SPD, las aplicaciones de metacomputación requieren una planificación de las comunicaciones para lograr las prestaciones deseadas; pero dada su naturaleza dinámica, son necesarias nuevas herramientas y técnicas. Es decir, mientras que la metacomputación puede formarse sobre la base de los SPD, para estos es necesario la creación de nuevas herramientas, mecanismos y técnicas [7].

Si se tiene en cuenta solo el aspecto de potencia de cálculo, podemos ver que existen diversas soluciones en función del tamaño y las características del problema. En primer lugar, se podría pensar en un superordenador (servidor), pero presentan problemas, como falta de escalabilidad, equipos y mantenimiento costoso, cómputo de picos (pasan mucho tiempo desaprovechados) y problemas de fiabilidad. La alternativa económica es un conjunto de ordenadores interconectados mediante una red de altas prestaciones (*Fast Ethernet –LAN– o Myrinet –SAN–*) lo cual formaría un clúster de estaciones dedicado a computación paralela/distribuida (SPD) con un rendimiento muy alto (relación coste/rendimiento de 3 a 15 veces). Pero estos sistemas presentan inconvenientes tales como coste elevado de las comunicaciones, mantenimiento, modelo de programación, etc. Sin embargo, es una solución excelente para aplicaciones de grano medio o HTC, *High Time Computing* (computación de alta productividad). Otro concepto interesante es el de *Intranet Computing*, que significa la utilización de los equipos de una red local (por ejemplo, una red de clase C) para ejecutar trabajos secuenciales o paralelos con la ayuda de una herramienta de administración y carga; es decir, es el siguiente paso a un clúster y permite la explotación de potencia computacional distribuida en una gran red local con las consiguientes ventajas, al aumentar el aprovechamiento de los recursos (ciclos de CPU a bajo coste), la mejora de la escalabilidad y una administración no demasiado compleja. Para este tipo de soluciones existe software como Oracle (Sun) Grid Engine de Sun Microsystems [19], Condor de la Universidad de Wisconsin (ambos gratuitos) [28] o LSF de Platform Computing (comercial) [20]. La opción del *Intranet Computing* presenta algunos inconvenientes tales como la imposibilidad de gestionar recursos fuera del dominio de administración.

Algunas de las herramientas mencionadas (Condor o SGE) permiten la colaboración entre diferentes subnodos del sistema, pero todos ellos deben tener la misma estructura administrativa, las mismas políticas de seguridad y la mis-

ma filosofía en la gestión de recursos. Si bien representa un paso adelante en la obtención de cómputo a bajo precio, solo gestionan la CPU y no los datos compartidos entre los subnodos. Además, los protocolos e interfaces son propietarios y no están basados en ningún estándar abierto, no se pueden amortizar los recursos cuando están desaprovechados ni se puede compartir recurso con otras organizaciones [2, 5, 4]. El crecimiento de los ordenadores y de la tecnología de redes en la última década ha motivado el desarrollo de una de las arquitecturas para HPC más relevantes de todos los tiempos: la computación distribuida en Internet o *grid computing* (GC). La computación distribuida ha transformado las grandes infraestructuras con el objetivo de compartir recursos en Internet de modo uniforme, transparente, seguro, eficiente y fiable. Esta tecnología es complementaria a las anteriores, ya que permite interconectar recursos en diferentes dominios de administración respetando sus políticas internas de seguridad y su software de gestión de recursos en la intranet. Según uno de sus precursores, Ian Foster, en su artículo "What is the Grid? A Three Point Checklist" (2002), una rejilla (*grid*) es un sistema que:

1) coordina recursos que no están sujetos a un control centralizado,

2) utiliza protocolos e interfaces estándares, abiertas y de propósitos generales, y

3) genera calidades de servicio no triviales.

Entre los beneficios que presenta esta nueva tecnología, se pueden destacar el alquiler de recursos, la amortización de recursos propios, gran potencia sin necesidad de invertir en recursos e instalaciones, colaboración y compartición entre instituciones y organizaciones virtuales, etc. [13].

El proyecto Globus [9] es uno de los más representativos en este sentido, ya que es el precursor en el desarrollo de un *toolkit* para la metacomputación o computación distribuida y que proporciona avances considerables en el área de la comunicación, la información, la localización y la planificación de recursos, la autentificación y el acceso a los datos. Es decir, Globus permite compartir recursos localizados en diferentes dominios de administración, con diferentes políticas de seguridad y gestión de recursos y está formado por un paquete de software (*middleware*) que incluye un conjunto de bibliotecas, servicios y API. La herramienta Globus (*Globus toolkit*) está formada por un conjunto de módulos con interfaces bien definidas para interactuar con otros módulos o servicios y su instalación es compleja y meticulosa, ya que requiere un conjunto de pasos previos, nombres y certificados que solo se justifican en caso de una gran instalación. La funcionalidad de estos módulos es la siguiente:

- **Localización y asignación de recursos**: permite comunicar a las aplicaciones cuáles son los requerimientos y ubicar los recursos que los satisfagan, ya que una aplicación no puede saber dónde se encuentran los recursos sobre los cuales se ejecutará.

- **Comunicaciones**: provee de los mecanismos básicos de comunicación, que representan un aspecto importante del sistema, ya que deben permitir diversos métodos para que se pueda utilizar eficientemente por las aplicaciones. Entre ellos se incluyen el paso de mensajes (*message passing*), llamadas a procedimientos remotos (RPC), memoria compartida distribuida, flujo de datos (*stream-based*) y *multicast*.

- **Servicio de información (Unified Resource Information Service)**: provee un mecanismo uniforme para obtener información en tiempo real sobre el estado y la estructura del metasistema donde se están ejecutando las aplicaciones.

- **Interfaz de autentificación**: son los mecanismos básicos de autentificación para validar la identidad de los usuarios y los recursos. El módulo genera la capa superior que luego utilizarán los servicios locales para acceder a los datos y los recursos del sistema.

- **Creación y ejecución de procesos**: utilizado para iniciar la ejecución de las tareas que han sido asignadas a los recursos, pasándoles los parámetros de ejecución y controlando la misma hasta su finalización.

- **Acceso a datos**: es el responsable de proveer un acceso de alta velocidad a los datos almacenados en archivos. Para DB, utiliza tecnología de acceso distribuido o a través de CORBA y es capaz de obtener prestaciones óptimas cuando accede a sistemas de archivos paralelos o dispositivos de entrada/salida por red, tales como los HPSS (*High Performance Storage System*).

Enlaces de interés

En http://www.globus.org/toolkit/about.html se puede observar la estructura interna de Globus. El sitio web de The Globus Alliance es http://www.globus.org [9]. Aquí se pueden encontrar tanto el código fuente, como toda la documentación necesaria para transformar nuestra intranet como parte de una rejilla (*grid*). Formar parte de una rejilla significa ponerse de acuerdo y adoptar las políticas de todas las instituciones y empresas que forman parte de ella. En España existen diferentes iniciativas basadas en Globus. Una de ellas es IrisGrid [13], a la cual es posible unirse para obtener las ventajas de esta tecnología. Para mayor información, consultad http://www.rediris.es/irisgrid.

Actividades

1. Instalad y configurad OpenMPI sobre un nodo; compilad y ejecutad el programa `cpi.c` y observad su comportamiento mediante xmpi.

2. Instalad y configurad OpenMP; compilad y ejecutad el programa de multiplicación de matrices (https://computing.llnl.gov/tutorials/openMP/exercise.html) en 2 *cores* y obtened pruebas de la mejora en la ejecución.

3. Utilizando dos nodos (puede ser con VirtualBox), instalad Ganglia y Cacti y monitorizad su uso.

4. Utilizando Rocks y VirtualBox, instalad dos máquinas para simular un clúster.

5. Instalad una versión de Debian virtualizado utilizando Qemu.

Bibliografía

[1] **Barney, B.** *OpenMP*. Lawrence Livermore National Laboratory.
<https://computing.llnl.gov/tutorials/openMP/>

[2] *Beowulf Web Site*.
<http://www.beowulf.org>

[3] *Cacti*.
<http://www.cacti.net/>

[4] **Dietz, H.** (2002). *Linux Parallel Processing*.
<http://cobweb.ecn.purdue.edu/ pplinux/PPHOWTO/pphowto.html>

[5] *Extreme Linux Web Site*.
<http://www.extremelinux.info/>

[6] *FAQ OpenMPI*.
<http://www.open-mpi.org/faq/>

[7] **Foster, I.; Kesselmany, K.** (2003). *Globus: A Metacomputing Infrastructure Toolkit*.
<http://www.globus.org>

[8] *Ganglia Monitoring System*.
<http://ganglia.sourceforge.net/>

[9] *Globus5*.
<http://www.globus.org/toolkit/docs/5.0/5.0.2/>

[10] *GT5 Quick Start Guide*.
<http://www.globus.org/toolkit/docs/5.0/5.0.2/admin/quickstart/>

[11] *Guía de instalación de Ganglia y Cacti*.
<http://debianclusters.org/index.php/Main_Page>

[12] *Guía rápida de instalación de Xen*.
<http://wiki.debian.org/Xen>

[13] **Martín Llorente, I.** *Estado de la Tecnología Grid y la Iniciativa IrisGrid*.
<http://www.irisgrid.es/>

[14] *MPI Examples*.
<http://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples/main.htm>(Download Section)

[15] *Mpich Project*.
<http://www.mcs.anl.gov:80/mpi/>

[16] *MPICH2 high-performance implementation of the MPI standard Mpich Project*.
<http://www.mcs.anl.gov/research/projects/mpich2/>

[17] *OpenMP Exercise*.
<https://computing.llnl.gov/tutorials/openMP/exercise.html>

[18] *OpenMPI*.
<http://www.open-mpi.org/>

[19] *Oracle (Sun) Grid Engine*.
<http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html>

[20] *Plataform LSF*.
<http://www.platform.com>

[21] *Qemu*.
<http://wiki.qemu.org/Main_Page>

[22] *QEMU. Guía rápida de instalación e integración con KVM y KQemu*.
<http://wiki.debian.org/QEMU>

[23] **Radajewski, J.; Eadline, D.** *Beowulf: Installation and Administration*.

[24] *Rocks Cluster. Installation Guide*.
<http://www.rocksclusters.org/roll-documentation/base/5.4/>

[25] **Swendson, K**. *Beowulf HOWTO* (tlpd).

[26] *System-config-cluster (FC)*.
<http://www.redhat.com/docs/manuals/enterprise/RHEL-5-manual/clúster_Administration/index.html>

[27] *The Xen hypervisor*.
<http://www.xen.org/>

[28] **Wisconsin University**. *Condor Web Site*.
<http://www.cs.wisc.edu/condor>

[29] **Woodman, L**. *Setting up a Beowulf cluster Using Open MPI on Linux*.
<http://techtinkering.com/articles/?id=32>