Treball Fi de Carrera

ENGINYERIA TÈCNICA EN INFORMÀTICA DE SISTEMES

Facultat de Matemàtiques

Universitat de Barcelona

Clustering en Linux

José Manuel Rodríguez Del Amo

Director: Jaume Timoneda Salat

Realitzat a: Departament de Matemàtica Aplicada i Anàlisi. UB

Barcelona, 6 de Juny de 2013

Indice:

- Overview.
- Introducción y definición de objetivos. (Página 5)
- HPC (H*igh Performance Computing*). (Página 6)
    - Cluster Beowulf. (Página 7)
        - Beowulf ¿Cómo configurar los nodos? (Página 8)
            - Configuración de la red.
            - Instalación y configuración del SSH (*Secure Shell*).
            - Instalación y configuración del NFS (*Network File System*).
        - Beneficios del cómputo distribuido. (Página 17)
    - ¿Cómo hay que programar para aprovechar la concurrencia? (Página 23)
        - Paralelización de programas. (Página 23)
        - PVM (*Parallel Virtual Machine*). (Página 24)
            - Funcionamiento de PVM.
            - Instalación y configuración de PVM.
            - PVM en funcionamiento.
            - Estructura de un programa con PVM.
            - Ejemplos de prueba.
            - Apagando nodos mientras ejecutan procesos PVM.
        - MPI (*Message Passing Interface*). (Página 50)
            - Funcionamiento de MPICH2.
            - Instalación y configuración de MPICH2.
            - MPICH2 en funcionamiento.
            - Estructura de un programa MPI.
            - Ejemplos de prueba.
            - Apagando nodos mientras ejecutan procesos MPI.
            - Comparativa PVM / MPI y conclusiones.
- MOSIX Multicomputer Operating System for UnIX. (Página 70)
    - Instalación.
    - Configuración.
    - MOSIX en funcionamiento.
    - Pruebas de compresión de música en MOSIX.
    - Pruebas de compresión de música MPICH2.
    - Tiempos dedicados a la compresión.
    - Apagando nodos mientras ejecutan procesos MOSIX.
    - *Checkpoints* en MOSIX.
    - Conclusiones y comparativa MPICH2 / MOSIX.
    - Ventajas de utilizar máquinas virtuales.

- Mare Nostrum III. (Página 95)
  - Introducción. (Página 95)
  - Extractos de la guía de usuario del MN3. (Página 98)
    - Conectarse mediante *Secure Shell*. (Página 98)
    - Sistemas de archivos. (Página 98)
      - Sistema de archivos GPFS.
      - Sistema de archivos raíz.
      - Disco duro local.
    - Ejecutando tareas. (Página 100)
      - Enviando tareas.
      - Comandos `LSF`.
      - Directivas de tareas.
      - Ejemplos.
    - Entorno de software. (Página 104)
      - Compiladores C.
      - Paralelismo de memoria distribuida.
      - Paralelismo de memoria compartida.
  - Documentación del MN3. (Página 107)
- *Grid computing*. (Página 117)
- Resumen y conclusiones. (Página 118)
- Bibliografía y referencias. (Página 119)
- Glosario. (Página 120)

Overview

High performance computing is essential and indispensable for present research and performed by supercomputers. The current supercomputers are based on a set of computers not so different to those who might have at home but connected by a high-performance network constituting a cluster.

This work is a study of several existing clustering solutions for HPC performance.

Starting with Beowulf Clusters, we introduce Parallel Virtual Machine as prior technology (though it still can be used by itself) but the study focuses on MPI. This kind of solutions (MPI and PVM) require rewriting of previous programs using parallelization or message passing libraries. As parallelization can be a complex task, also we show another solution available for HPC: MOSIX, that adds clustering capabilities to the operating system kernel in a transparent way to the user and does not require the parallelization of programs nor any special library. It is suitable for the implementation of intensive computing tasks with low/moderate use of input/output.

Finally we review the supercomputer Mare Nostrum III in the Barcelona Supercomputing Center. Reading the MN3 User's Guide shows that MPI and multithreading are essential in the development of programs running on current supercomputers.

Se autoriza la divulgación de esta obra mediante las condiciones de la licencia Reconocimiento-CompartirIgual (BY-SA) de Creative Commons. El texto completo de esta licencia se encuentra disponible en:

http://creativecommons.org/licenses/by-sa/3.0/es/

Introducción y definición de objetivos

Un *clúster* es un conjunto de ordenadores en una LAN que trabajan con un objetivo común.
Los *grids* son agrupaciones de ordenadores unidos por redes de área extensa (WAN).

Se denomina *clúster* (agrupación) de ordenadores a un grupo de ordenadores que trabajan con un fin común. Estos ordenadores agrupan hardware, redes de comunicación y software para trabajar conjuntamente como si fueran un único sistema. Existen muchas razones atrayentes para realizar estas agrupaciones, pero la principal es poder efectuar el procesamiento de la información de forma más eficiente y rápida como si fuera un único sistema. Generalmente, un *clúster* trabaja sobre una red de área local (LAN) y permite una comunicación eficiente, si bien las máquinas se encuentran dentro de un espacio físico próximo. Una concepción mayor del concepto es la llamada *grid*, donde el objetivo es el mismo, pero implica agrupaciones de ordenadores unidos por redes de área extensa (WAN). Algunos autores consideran el *grid* como un *clúster* de *clusters* en un sentido 'global'. Si bien cada vez más la tecnología y los costes permiten estas aproximaciones, los esfuerzos y la complejidad de utilización de decenas o centenares (en algunos casos, miles) es muy grande. Sin embargo, las ventajas en tiempo de cómputo hacen que, aun así, este tipo de soluciones para el cómputo de altas prestaciones (HPC, *high performance computing*) sean consideradas muy atractivas y en constante evolución.

La computación de alto rendimiento es fundamental e imprescindible para la investigación y se lleva a cabo mediante supercomputadores (o superordenadores). Los supercomputadores actuales se basan en: Un conjunto de computadores de potencia similar a los que podríamos tener en casa pero con la diferencia de que estos computadores están interconectados por una red de altas prestaciones constituyendo un clúster.

Este trabajo pretende ser un estudio de soluciones *Clustering* actuales para la Computación de Alto rendimiento:

Por un lado se estudia el Clúster Beowulf donde hay un sistema de archivos compartido mediante *Network File System* y se utiliza *Secure Shell* para ejecutar comandos remotamente en los nodos mediante *Scripts*. También se estudia las soluciones: *Parallel Virtual Machine* y *Message Passing Interface*.

Por otro lado se estudia MOSIX: Esta solución consiste en la modificación del núcleo del sistema operativo para añadirle, a este, funcionalidades de *clustering*.

Finalmente se examina el superordenador Mare Nostrum III del Centro de Súper Computación de Barcelona - Centro Nacional de Súper Computación. El MN3 es un clúster de computadores, el cual tiene instalado el software MPI que se estudia en este trabajo.

HPC (High Performance Computing)

Los avances en la tecnología han significado procesadores rápidos, de bajo coste y 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.
La historia de los sistemas informáticos es muy reciente (se puede decir que comienza en la década de los sesenta). En un principio eran sistemas grandes, pesados, caros, de pocos usuarios expertos, no accesibles, lentos. En la década de los setenta, 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 los ochenta se caracteriza por un aumento notable de las prestaciones (hasta hoy en día) y una reducción del tamaño en los llamados *microcomputers*. Su evolución ha sido a través de las estaciones de trabajo (*workstations*) y los avances en redes es un factor fundamental en las aplicaciones multimedia. Los sistemas distribuidos, por su parte, comenzaron su historia en la década de los setenta (sistemas de 4 u 8 ordenadores) y su salto a la popularidad lo hicieron en la década de los noventa.
Si bien su administración/instalación/mantenimiento es compleja, las razones básicas de su popularidad son el incremento de prestaciones que presentan en aplicaciones intrínsecamente distribuidas (por su naturaleza), la información compartida por un conjunto de usuarios, compartir recursos, la alta tolerancia a los fallos y la posibilidad de expansión incremental (capacidad de agregar más nodos para aumentar las prestaciones de modo incremental).

A la hora conseguir la mayor potencia de cómputo, sale más a cuenta económicamente interconectar un conjunto de computadores y utilizar el paso de mensajes entre procesos que tener una máquina con un procesador muy rápido (esta no es buena solución por culpa de las limitaciones tecnológicas) también sale más a cuenta que tener una Máquina de Memoria Compartida, es decir, muchos procesadores con muchos núcleos cada uno y con mucha memoria compartida entre todos los procesadores. En una MMC se pueden ejecutar procesos con muchísimos hilos de ejecución (threads) consiguiendo gran capacidad de cálculo gracias a la paralelización de memoria compartida pero hay que tener en cuenta que en una MMC solo se puede utilizar threads y tiene el problema de la escalabilidad, además es un hardware complejo por lo cual es caro, en cambio, interconectando un conjunto de ordenadores más sencillos y utilizando paso de mensajes entre procesos (Paralelismo de memoria distribuida) podemos conseguir una potencia de cálculo que económicamente no sería posible de otra manera, además no limita el uso de threads, aunque a nivel local, para poder aprovechar nodos multiprocesador y/o multinúcleo y/o tener concurrencia entre procesamiento y entrada/salida. Esta solución tiene el problema de la latencia en las comunicaciones pero gracias a los avances en este campo es la mejor solución, además tiene como punto a favor la escalabilidad y la relación rendimiento/coste.

Cluster Beowulf

Beowulf 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 a través de UNIX es posible (socket, rpc), puede significar 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.

La capa de software aportada por sistemas tales como *parallel virtual machine* (PVM) y *message passing interface* (MPI) facilita notablemente la abstracción del sistema y permite programar APD 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.

Hay que considerar que Beowulf no es un software que transforma el código del usuario en distribuido ni afecta al *kernel* del sistema operativo (como por ejemplo MOSIX). Simplemente, es una forma de agrupación (*cluster*) 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 *kernel* para obtener mejores prestaciones, pero es posible construir un *cluster* Beowulf a partir de un GNU/Linux estándar y de software convencional. La construcción de un *cluster B*eowulf de dos nodos, por ejemplo, se puede llevar a cabo simplemente con las dos máquinas conectadas por Ethernet mediante un *hub*, una distribución de GNU/ Linux estándar (Debian), el sistema de archivos compartido (NFS) y tener habilitados los servicios de red como SSH.

Beowulf ¿Cómo configurar los nodos?

Todas las pruebas que salen en estos textos han sido realizadas en máquinas virtuales al no disponer de las suficientes máquinas reales.

Se parte de una máquina virtual creada y con el sistema operativo `Linux Debian 6` instalado con el usuario `josemanuel` creado durante la instalación, se hacen 3 clonaciones de esta máquina y se conectan a través de un switch virtualmente.

Configuración de la red:

La interfaz de red eth1 conecta la máquina virtual con la maquina real a través de NAT y sirve tener conexión con internet. A esta interfaz no hace falta cambiarle la configuración pero a la interfaz de red eth2 , hay que cambiarle la configuración:

Este paso se hace idénticamente en las 3 máquinas pero a cada una, se le asigna una dirección IP diferente. A la primera se le pone la dirección 192.168.1.2, a la segunda 192.168.1.3 y a la tercera 192.168.1.4.

De este modo las máquinas pueden comunicarse a través de la red local.

Hay que modificar los archivos `/etc/hostname` y `/etc/hosts` como se ilustra a continuación:

```
josemanuel@ordenador1: ~
Archivo  Editar  Ver  Terminal  Ayuda
josemanuel@ordenador1:~$ ls
Descargas   Downloads    Imágenes   Música       Público
Documentos  Escritorio   jm         Plantillas   Vídeos
josemanuel@ordenador1:~$ cat /etc/hostname
ordenador1
josemanuel@ordenador1:~$ cat /etc/hosts
127.0.0.1       localhost.localdomain   localhost
192.168.1.2     ordenador1
192.168.1.3     ordenador2
192.168.1.4     ordenador3
192.168.1.5     ordenador4

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
josemanuel@ordenador1:~$
```

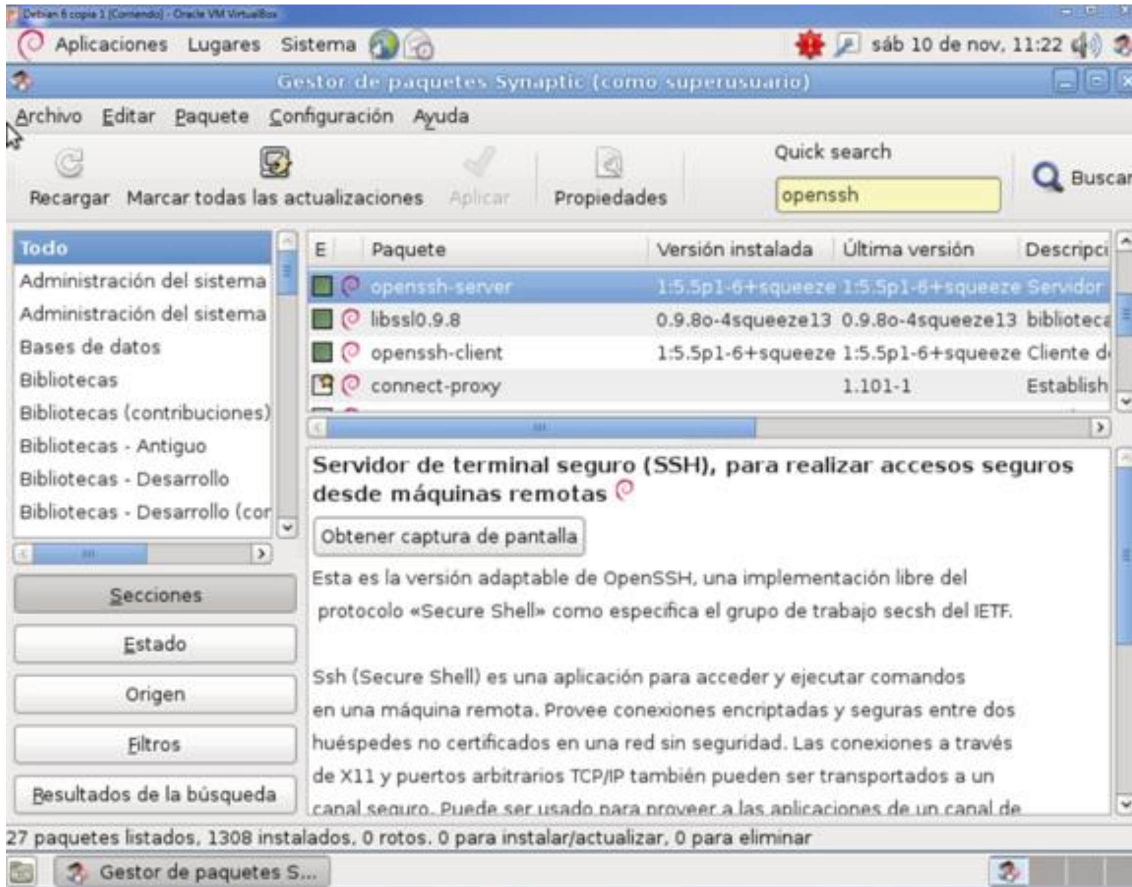
Este paso se hace idénticamente en las 3 máquinas pero la con la diferencia de que en el archivo `/etc/hostname` hay que poner ordenador1 en la primera máquina, ordenador2 en la segunda y ordenador3 en la tercera.

Para comprobar que la comunicación es correcta y que los nombres se resuelven bien, podemos hacer la prueba siguiente:

```
josemanuel@ordenador1: ~
Archivo  Editar  Ver  Terminal  Ayuda
josemanuel@ordenador1:~$ ping ordenador2
PING ordenador2 (192.168.1.3) 56(84) bytes of data.
64 bytes from ordenador2 (192.168.1.3): icmp_req=1 ttl=64 time=0.421 ms
64 bytes from ordenador2 (192.168.1.3): icmp_req=2 ttl=64 time=0.311 ms
64 bytes from ordenador2 (192.168.1.3): icmp_req=3 ttl=64 time=0.302 ms
^C
--- ordenador2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.302/0.344/0.421/0.058 ms
josemanuel@ordenador1:~$ ping ordenador3
PING ordenador3 (192.168.1.4) 56(84) bytes of data.
64 bytes from ordenador3 (192.168.1.4): icmp_req=1 ttl=64 time=0.460 ms
64 bytes from ordenador3 (192.168.1.4): icmp_req=2 ttl=64 time=0.401 ms
64 bytes from ordenador3 (192.168.1.4): icmp_req=3 ttl=64 time=0.310 ms
64 bytes from ordenador3 (192.168.1.4): icmp_req=4 ttl=64 time=0.309 ms
^C
--- ordenador3 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2998ms
rtt min/avg/max/mdev = 0.309/0.370/0.460/0.064 ms
josemanuel@ordenador1:~$
```

Instalación y configuración del SSH (Secure Shell).

Al hacer la instalación de Linux Debian hay que seleccionar instalar SSH (Secure Shell). En caso contrario habría que instalar los paquetes:



A continuación hay que configurar SSH para poder *logarse* y ejecutar comandos remotamente sin necesidad de introducir contraseña utilizando el método RSA. Esto es importante porque va a haber un ordenador (el ordenador1) que va a invocar comandos remotamente en otros ordenadores: En el ordenador2 y en el ordenador3. A continuación se ilustra el proceso. Primero se crea la clave pública:

```
josemanuel@debianJMR: ~
Archivo  Editar  Ver  Terminal  Ayuda
josemanuel@debianJMR:~$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/josemanuel/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/josemanuel/.ssh/id_rsa.
Your public key has been saved in /home/josemanuel/.ssh/id_rsa.pub.
The key fingerprint is:
5e:ff:c8:1e:48:88:92:f4:b8:a6:a9:3e:b1:f0:32:fd josemanuel@debianJMR
The key's randomart image is:
+--[ RSA 2048]----+
|                 |
|                 |
|     .           |
|    . + . .      |
|     + o S o     |
|..   o . o o     |
|.oo o   . . o    |
|ooo+        . +  |
|o=+.E       .+ . |
+-----------------+
josemanuel@debianJMR:~$
```

Nótese que: Se invoca el comando desde el directorio `home` del usuario `josemanuel` y que en `Enter passphrase` hay que dejarlo vacío, esto es importante para poder acceder sin introducir la *passphrase* con este usuario.

A continuación se ilustra cómo se añade la clave pública al archivo de claves autorizadas del ordenador en el cual queremos logarnos e invocar comandos remotamente sin introducir contraseña (debe existir en este ordenador el mismo usuario `josemanuel`). En este caso, como son máquinas virtuales y esta máquina es un clon de la otra, este usuario ya existe en las dos máquinas, en caso contrario habría que crearlo.

```
josemanuel@debianJMR: ~
Archivo  Editar  Ver  Terminal  Ayuda
|o=+.E       .+ .  |
+-----------------+
josemanuel@debianJMR:~$ cat .ssh/id_rsa.pub | ssh ordenador1 \ "cat - >>.ssh/authorized_keys"
The authenticity of host 'ordenador1 (192.168.1.2)' can't be established.
RSA key fingerprint is 07:68:68:c5:2e:6d:05:9e:51:0e:c0:6e:2d:b4:51:07.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'ordenador1,192.168.1.2' (RSA) to the list of known hosts.
josemanuel@ordenador1's password:
josemanuel@debianJMR:~$ ssh ordenador1
Linux ordenador1 2.6.32-5-amd64 #1 SMP Sun May 6 04:00:17 UTC 2012 x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sun Oct 28 13:04:31 2012 from ordenador1
josemanuel@ordenador1:~$ exit
logout
Connection to ordenador1 closed.
josemanuel@debianJMR:~$
```

De esta forma se puede acceder mediante SSH al ordenador1 (en este caso). Este proceso hay que hacerlo del ordenador1 al ordendor2 y del ordenador1 al ordenador3 para que el ordenador1 pueda invocar comandos remotamente en los ordenadores 2 y 3. También se puede hacer el proceso del ordenador1 al ordenador1 (aunque no es necesario).Aunque no hace falta habilitar el acceso por SSH sin contraseña hacia el mismo ordenador porque se puede invocar el comando directamente sin utilizar SSH.

A continuación se ilustra cómo se puede acceder con el comando `ssh`: Del ordenador1 al ordenador2 y al ordenador3 sin introducir contraseña:

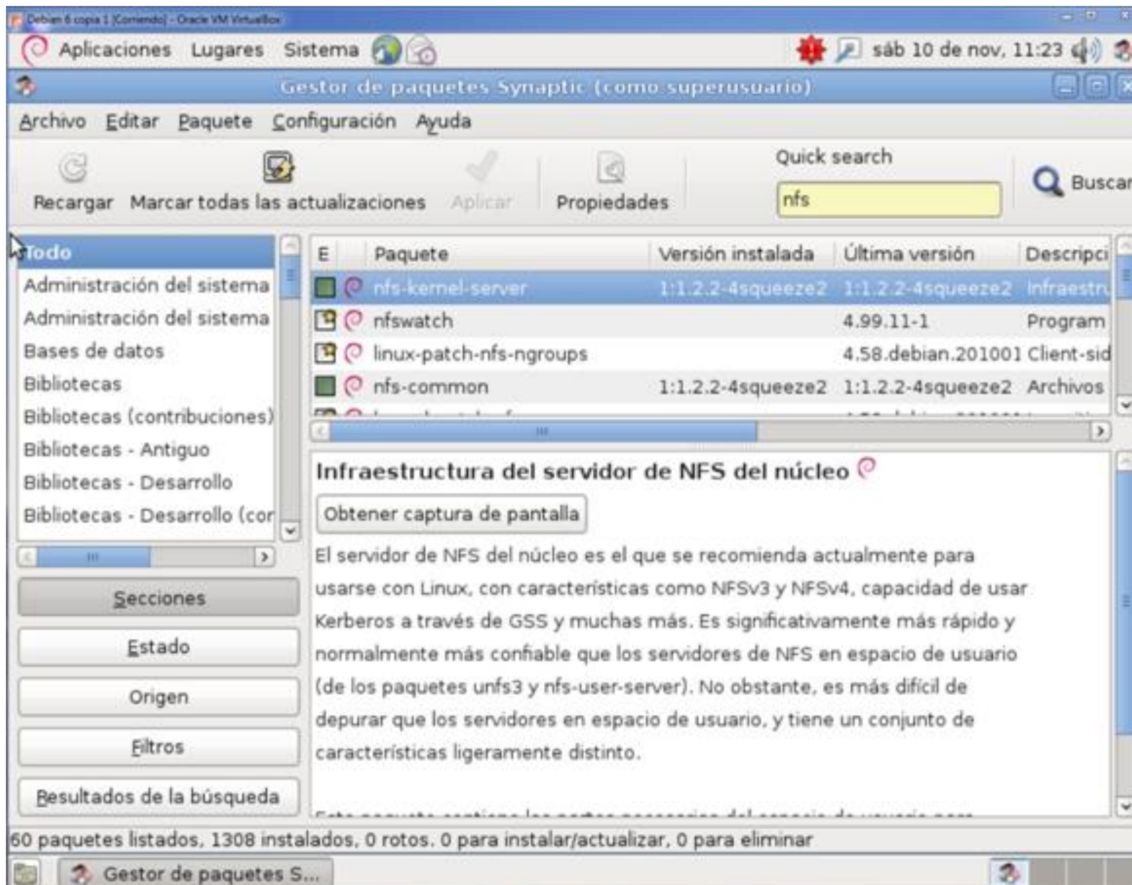
```
josemanuel@ordenador1: ~
Archivo Editar Ver Terminal Ayuda
josemanuel@ordenador1:~$ ssh ordenador2
Linux ordenador2 2.6.32-5-amd64 #1 SMP Sun May 6 04:00:17 UTC 2012 x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
You have new mail.
Last login: Sat Nov 10 14:32:17 2012 from ordenador1
josemanuel@ordenador2:~$ exit
logout
Connection to ordenador2 closed.
josemanuel@ordenador1:~$ ssh ordenador3
Linux ordenador3 2.6.32-5-amd64 #1 SMP Sun May 6 04:00:17 UTC 2012 x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Nov  9 17:26:53 2012 from ordenador1
josemanuel@ordenador3:~$ exit
logout
Connection to ordenador3 closed.
josemanuel@ordenador1:~$
```

Tenemos 3 ordenadores y todos tienen el usuario `josemanuel`.

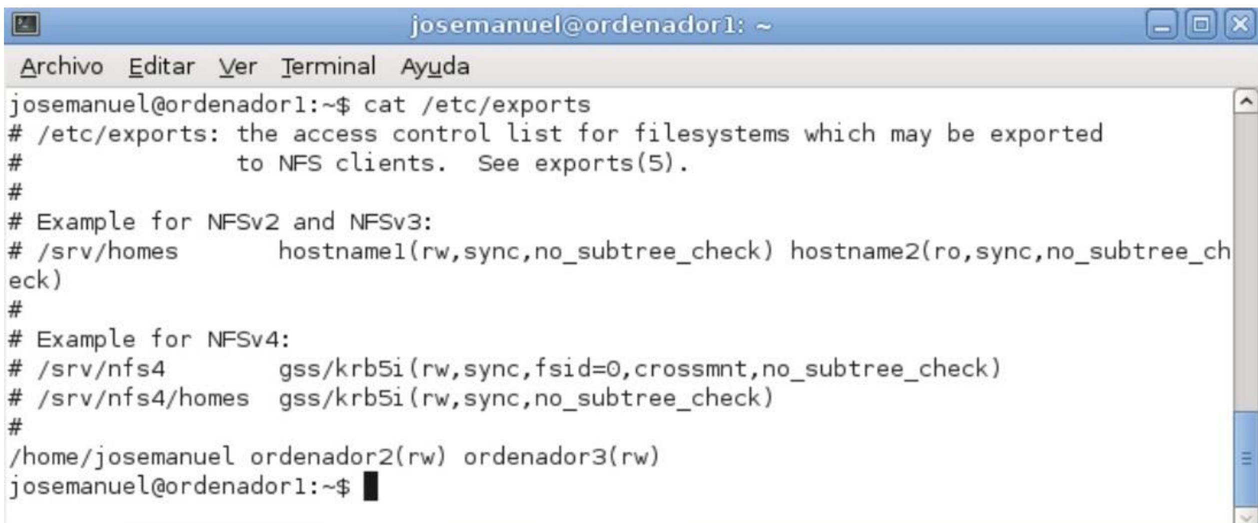
Instalación y configuración del NFS (Network File System)

Lo siguiente es: Hacer que los 3 usuarios `josemanuel` de las tres máquinas utilicen el mismo directorio `home`. Para esto: Hay que instalar el servidor NFS (Network File System) en el ordenador1 para que sirva el directorio `home` a los ordenadores 2 y 3.

Si no vienen instalados por defecto, hay que instalar los paquetes nfs-kernel-server y nfs-common.

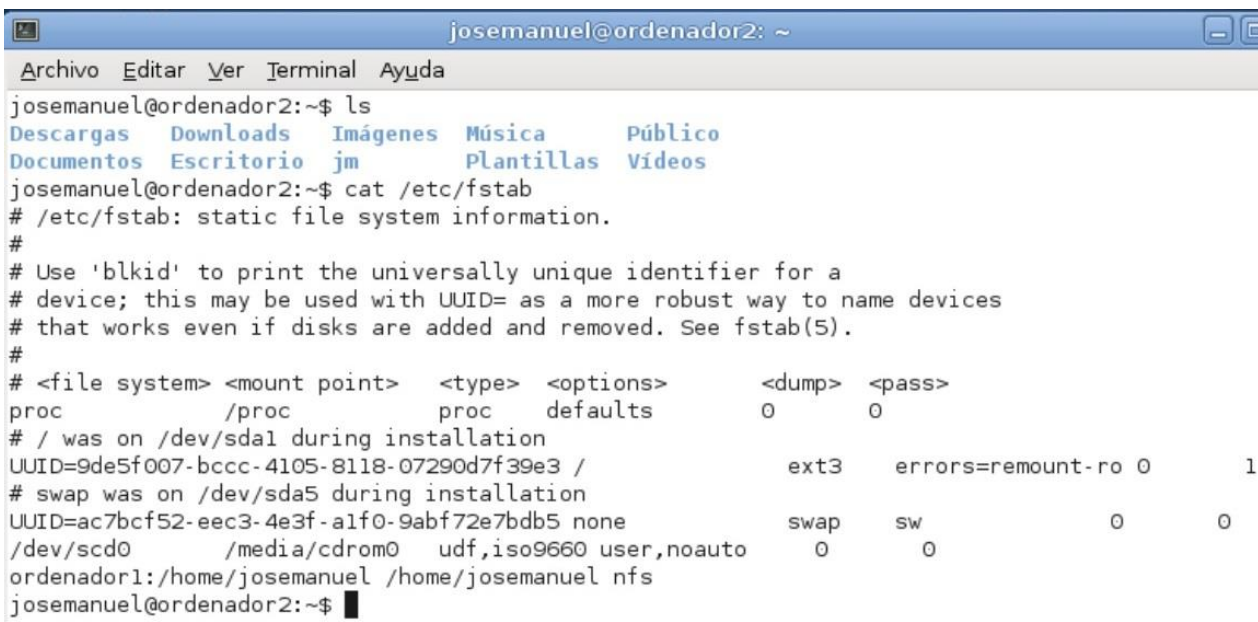


Hay que añadir la línea que sale abajo de todo de la ilustración, en el fichero `/etc/exports` del ordenador1 (servidor NFS):



De esta manera el directorio `/home/josemanuel` lo podrán montar los ordenadores2 y 3 mediante la red con permisos de lectura y escritura.

Después hay que añadir al archivo `/etc/fstab` del ordenador2 y del ordenador3, la línea de abajo del todo de la ilustración.



De esta manera los ordenadores 2 y 3 al iniciar el sistema, montan mediante de la red, el directorio `home` que tiene compartido el ordenador1.

Habría que reiniciar las tres máquinas, teniendo en cuenta que: El ordenador1 se debe iniciar primero porque hace de servidor NFS. Una vez iniciado el ordenador1, se pueden encender los ordenadores 2 y 3.

A continuación se muestra como el directorio `home` del usuario `josemanuel` en el ordenador1 es el mismo que el del mismo usuario en el ordenador2 y el mismo que en el del ordenador3. Se muestra ejecutando el comando `ls -l` en el ordenador1 y ejecutando el mismo comando remotamente en el ordenador 2 y ordenador3 mediante SSH con el usuario `josemanuel`:

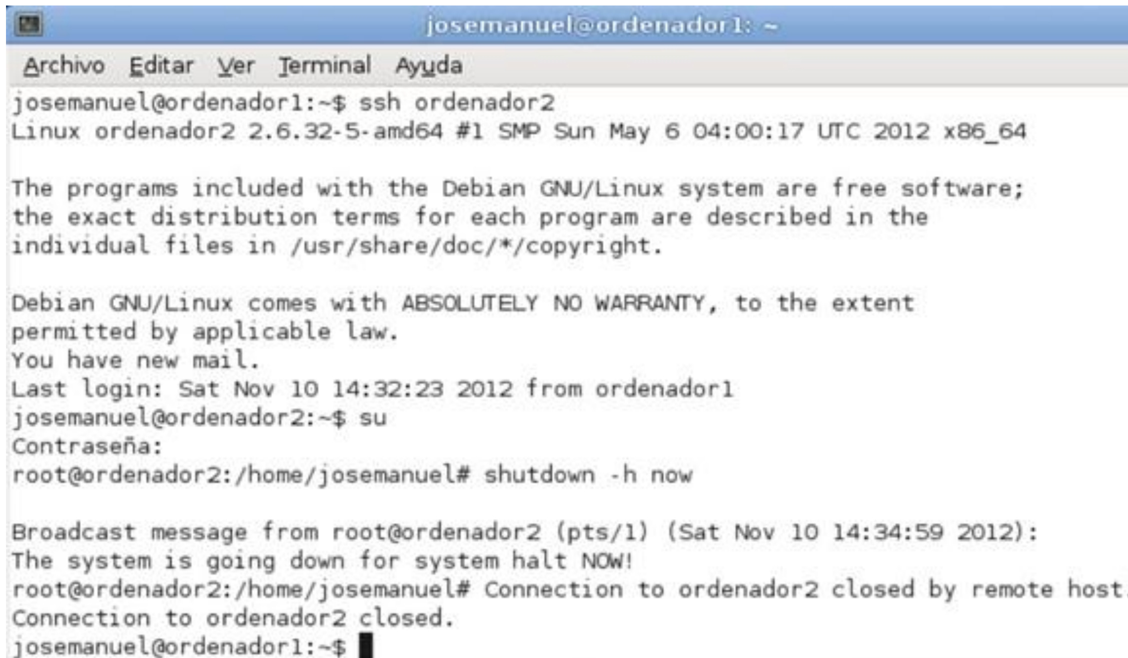
```
josemanuel@ordenador1: ~
Archivo  Editar  Ver  Terminal  Ayuda
josemanuel@ordenador1:~$ ls -l
total 40
drwxr-xr-x 2 josemanuel josemanuel 4096 nov  8 17:23 Descargas
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Documentos
drwx------ 2 josemanuel josemanuel 4096 oct 12 15:35 Downloads
drwxr-xr-x 2 josemanuel josemanuel 4096 nov 10 14:17 Escritorio
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Imágenes
drwxr-xr-x 3 josemanuel josemanuel 4096 nov  8 17:13 jm
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Música
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Plantillas
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Público
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Vídeos
josemanuel@ordenador1:~$ ssh ordenador2 ls -l
total 40
drwxr-xr-x 2 josemanuel josemanuel 4096 nov  8 17:23 Descargas
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Documentos
drwx------ 2 josemanuel josemanuel 4096 oct 12 15:35 Downloads
drwxr-xr-x 2 josemanuel josemanuel 4096 nov 10 14:17 Escritorio
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Imágenes
drwxr-xr-x 3 josemanuel josemanuel 4096 nov  8 17:13 jm
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Música
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Plantillas
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Público
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Vídeos
josemanuel@ordenador1:~$
```

```
josemanuel@ordenador1: ~
Archivo  Editar  Ver  Terminal  Ayuda
josemanuel@ordenador1:~$ ssh ordenador2 ls -l
total 40
drwxr-xr-x 2 josemanuel josemanuel 4096 nov  8 17:23 Descargas
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Documentos
drwx------ 2 josemanuel josemanuel 4096 oct 12 15:35 Downloads
drwxr-xr-x 2 josemanuel josemanuel 4096 nov 10 14:17 Escritorio
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Imágenes
drwxr-xr-x 3 josemanuel josemanuel 4096 nov  8 17:13 jm
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Música
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Plantillas
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Público
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Vídeos
josemanuel@ordenador1:~$ ssh ordenador3 ls -l
total 40
drwxr-xr-x 2 josemanuel josemanuel 4096 nov  8 17:23 Descargas
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Documentos
drwx------ 2 josemanuel josemanuel 4096 oct 12 15:35 Downloads
drwxr-xr-x 2 josemanuel josemanuel 4096 nov 10 14:17 Escritorio
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Imágenes
drwxr-xr-x 3 josemanuel josemanuel 4096 nov  8 17:13 jm
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Música
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Plantillas
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Público
drwxr-xr-x 2 josemanuel josemanuel 4096 oct  8 19:23 Vídeos
josemanuel@ordenador1:~$
```

De este modo tenemos un ordenador, el ordenador1 que hace de servidor de ficheros (NFS) y desde donde se pueden invocar comandos remotamente en los ordenadores 2 y 3 mediante SSH sin introducir contraseña (la autenticación del usuario se realiza mediante RSA).

El ordenador2 y el ordenador3 se podrían dejar encendidos sin teclado, ni ratón, ni pantalla para ahorrar energía y espacio.

Los ordenadores 2 y 3 se pueden gestionar remotamente:

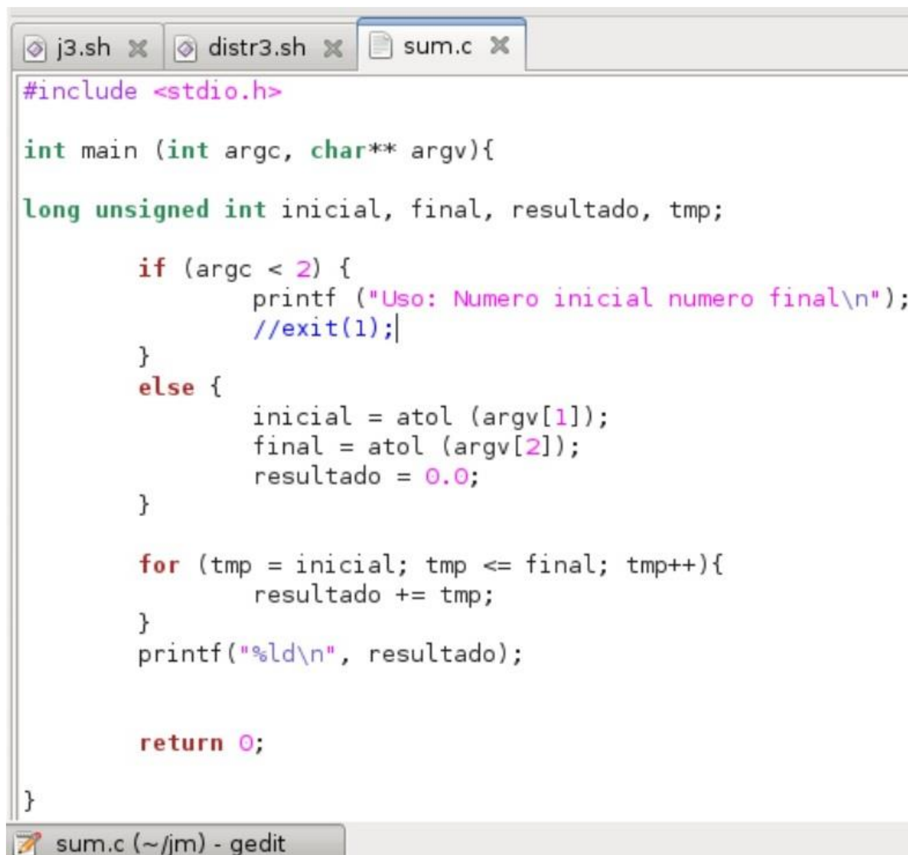


```
josemanuel@ordenador1: ~
Archivo Editar Ver Terminal Ayuda
josemanuel@ordenador1:~$ ssh ordenador2
Linux ordenador2 2.6.32-5-amd64 #1 SMP Sun May 6 04:00:17 UTC 2012 x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
You have new mail.
Last login: Sat Nov 10 14:32:23 2012 from ordenador1
josemanuel@ordenador2:~$ su
Contraseña:
root@ordenador2:/home/josemanuel# shutdown -h now

Broadcast message from root@ordenador2 (pts/1) (Sat Nov 10 14:34:59 2012):
The system is going down for system halt NOW!
root@ordenador2:/home/josemanuel# Connection to ordenador2 closed by remote host.
Connection to ordenador2 closed.
josemanuel@ordenador1:~$
```

Beneficios del cómputo distribuido:

Ejemplo demostrativo:

Al programa `sum.c` , de la ilustración, se le pasan dos parámetros: El número inicial y el número final, entonces el programa hace la suma de números enteros consecutivos.



Un ejemplo de uso sería:

```
./sum 1 10
```

El programa sumaría 1+2+3+4+5+6+7+8+9+10 e imprimiría el resultado por pantalla.

Se compila el programa y se muestra su funcionamiento:

```
josemanuel@ordenador1: ~/jm
Archivo  Editar  Ver  Terminal  Ayuda
josemanuel@ordenador1:~/jm$ gcc sum.c -o sum
josemanuel@ordenador1:~/jm$ ls -l
total 48
-rwxr-xr-x 1 josemanuel josemanuel  271 oct 28 17:02 distr2.sh
-rwxr-xr-x 1 josemanuel josemanuel  375 nov  1 11:23 distr3.sh
-rwxr-xr-x 1 josemanuel josemanuel  185 oct 28 17:01 distr_.sh
-rwxr-xr-x 1 josemanuel josemanuel  187 oct 28 17:00 distr.sh
-rwxr-xr-x 1 josemanuel josemanuel   88 oct 28 16:49 j2.sh
-rwxr-xr-x 1 josemanuel josemanuel   88 nov  1 11:00 j3.sh
-rwxr-xr-x 1 josemanuel josemanuel   88 oct 28 16:50 j_.sh
-rwxr-xr-x 1 josemanuel josemanuel   87 oct 28 16:58 j.sh
drwxr-xr-x 2 josemanuel josemanuel 4096 nov  9 17:16 pvm
prw-r--r-- 1 josemanuel josemanuel    0 nov  1 11:23 salida
-rwxr-xr-x 1 josemanuel josemanuel 6981 nov 15 16:47 sum
-rw-r--r-- 1 josemanuel josemanuel  385 nov  1 11:35 sum.c
josemanuel@ordenador1:~/jm$ ./sum 1 10
55
josemanuel@ordenador1:~/jm$
```

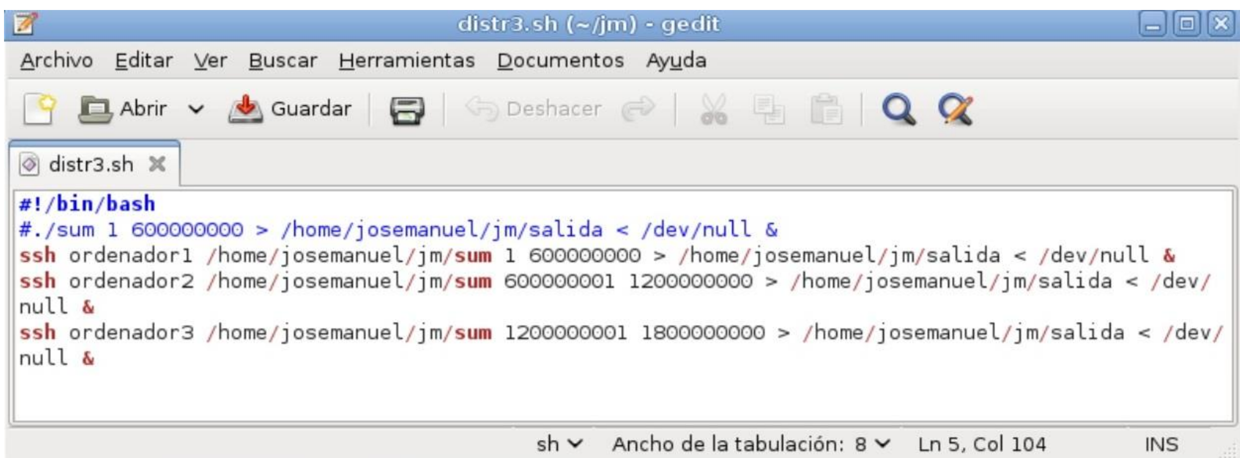
Aquí se ilustran los tiempos que tarda el programa `sum` en calcular la suma de los números del 1 al 10^6 y del 1 al $16*10^6$, en nuestra máquina virtual:

```
josemanuel@debianJMR: ~
Archivo  Editar  Ver  Terminal  Ayuda
josemanuel@debianJMR:~/jm$ time ./sum 1 1000000
500000500000

real    0m0.007s
user    0m0.008s
sys     0m0.000s
josemanuel@debianJMR:~/jm$ time ./sum 1 16000000
128000008000000

real    0m0.066s
user    0m0.064s
sys     0m0.000s
josemanuel@debianJMR:~/jm$
```

En calcular la suma del 1 al $16*10^6$ tarda 9 veces más que en hacer el mismo cálculo del 1 al 10^6

Hay que crear un archivo tubería FIFO (First In First Out) llamado `salida` :

Hay que crear los scripts `distr3.sh` y `j3.sh` que se ilustran en las figuras siguientes y darles permiso de ejecución:



El carácter "&" al final del comando hace que la tarea se ejecute en segundo plano, así para invocar el siguiente comando no hace falta esperar a que acabe el actual.

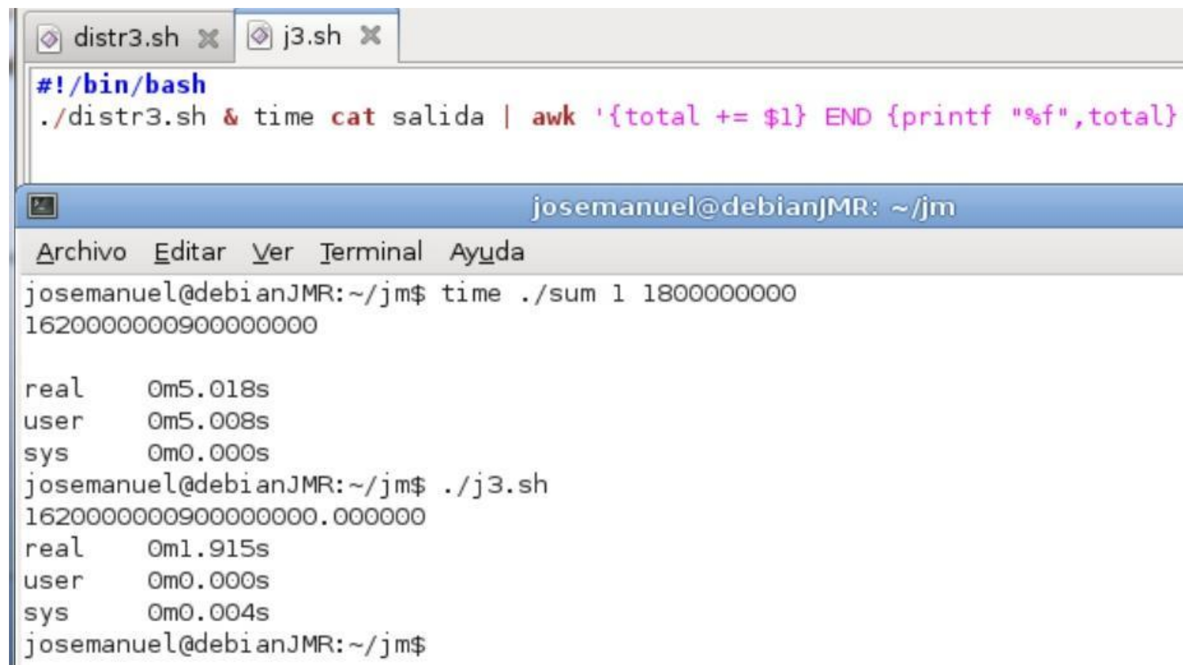
Si no estuviera configurado SSH para autenticarse sin contraseña, nos pediría introducirla cada vez que invocáramos un comando remoto.

El ordenador1 hace de servidor NFS y es desde donde se invocan los comandos remotos, este ordenador se puede utilizar (o no) para hacer cálculos, en este caso se utiliza para aprovechar más los sistemas disponibles.

El comando, en el ordenador1, se puede invocar directamente o mediante `ssh` , en las pruebas se hizo de las dos maneras para probar, al final se dejó comentada la opción sin `ssh` para mantener la homogeneidad en el código.

El script `distr3.sh` lanza el comando `sum` para sumar del 1 al 600000000 en el ordenador1 , sumar del 600000001 al 1200000000 en el ordenador2 y sumar del 1200000001 al 1800000000 en el ordenador3.

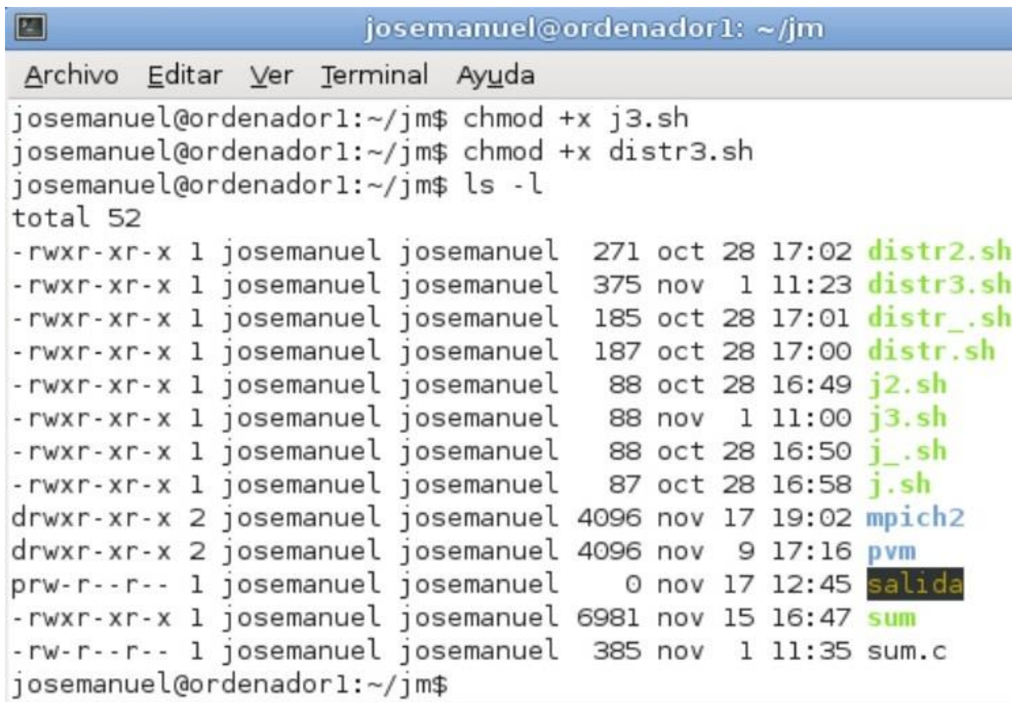
El comando `awk` invocado por el `script j3.sh` , suma las sumas parciales e imprime el resultado por pantalla.



En la imagen se ilustra la ejecución del programa `sum` para sumar del 1 al 1800000000 y también se ilustra la ejecución del mismo cálculo pero de manera distribuida invocando el script `j3.sh` , de esta manera se utilizan los 3 ordenadores para hacer la suma.

Se aprecia que el tiempo disminuye más de la mitad: El cómputo se realiza 2.6 veces más rápido. El tiempo tiende a dividirse entre el número de nodos que tenemos disponibles para hacer los cálculos (en este caso 3).

Para poder ejecutar los scripts anteriores, antes hay que darles permiso de ejecución como se ilustra a continuación:



En la imagen se ilustra el archivo FIFO `salida` que en el ejemplo se utiliza para ir recibiendo los resultados de las sumas parciales que hace cada ordenador. A medida que van llegando los resultados de cada ordenador al archivo FIFO, el comando `awk` va sumando los resultados parciales y al final imprime el resultado completo por pantalla.

Esto ha sido un ejemplo muy simple con fines demostrativos. En la práctica los programadores utilizan bibliotecas que les permiten realizar en tiempo de ejecución, la creación y comunicación de procesos en un sistema distribuido, por ejemplo PVM y MPI.

A continuación se muestra los scripts utilizados para el resto pruebas que se hicieron relativas al apartado anterior:

```
josemanuel@ordenador1: ~/jm
Archivo  Editar  Ver  Terminal  Ayuda
josemanuel@ordenador1:~/jm$ cat j.sh
#!/bin/bash
./distr.sh & time cat salida | awk '{total += $1} END {printf "%d",total}'
josemanuel@ordenador1:~/jm$ cat distr.sh
#!/bin/bash
ssh 192.168.1.3 /home/josemanuel/jm/sum 1 5 > /home/josemanuel/jm/salida < /dev/null &
ssh 192.168.1.4 /home/josemanuel/jm/sum 6 10 > /home/josemanuel/jm/salida < /dev/null &
josemanuel@ordenador1:~/jm$ cat j2.sh
#!/bin/bash
./distr2.sh & time cat salida | awk '{total += $1} END {printf "%d",total}'
josemanuel@ordenador1:~/jm$ cat distr2.sh
#!/bin/bash
ssh ordenador1 /home/josemanuel/jm/sum 1 3 > /home/josemanuel/jm/salida < /dev/null &
ssh ordenador2 /home/josemanuel/jm/sum 4 7 > /home/josemanuel/jm/salida < /dev/null &
ssh ordenador3 /home/josemanuel/jm/sum 8 10 > /home/josemanuel/jm/salida < /dev/null &
josemanuel@ordenador1:~/jm$ cat j3.sh
#!/bin/bash
./distr3.sh & time cat salida | awk '{total += $1} END {printf "%f",total}'
josemanuel@ordenador1:~/jm$ cat distr3.sh
#!/bin/bash
#./sum 1 600000000 > /home/josemanuel/jm/salida < /dev/null &
ssh ordenador1 /home/josemanuel/jm/sum 1 600000000 > /home/josemanuel/jm/salida < /dev/null
&
ssh ordenador2 /home/josemanuel/jm/sum 600000001 1200000000 > /home/josemanuel/jm/salida < /
dev/null &
ssh ordenador3 /home/josemanuel/jm/sum 1200000001 1800000000 > /home/josemanuel/jm/salida <
/dev/null &
josemanuel@ordenador1:~/jm$
```

```
josemanuel@ordenador1: ~/jm
Archivo  Editar  Ver  Terminal  Ayuda
josemanuel@ordenador1:~/jm$ ls
distr2.sh  distr3.sh  distr_.sh  distr.sh  j2.sh  j3.sh  j_.sh  j.sh  pvm  salida  sum  sum.c
josemanuel@ordenador1:~/jm$ cat j_.sh
#!/bin/bash
./distr_.sh & time cat salida | awk '{total += $1} END {printf "%d",total}'
josemanuel@ordenador1:~/jm$ cat distr_.sh
#!/bin/bash
ssh ordenador2 /home/josemanuel/jm/sum 1 5 > /home/josemanuel/jm/salida < /dev/null &
ssh ordenador3 /home/josemanuel/jm/sum 6 10 > /home/josemanuel/jm/salida < /dev/null &
josemanuel@ordenador1:~/jm$
```

Si se realizan los pasos descritos anteriormente: Se tiene configurado, con el usuario `josemanuel`: SSH y NFS en los tres ordenadores, para poder ejecutar comandos remotamente sin introducir contraseña y con el directorio `home` compartido.

De este modo se tiene: Un *cluster* beowulf para ejecutar aplicaciones que podrán ser PVM o MPI (se verá en los siguientes apartados).

¿Cómo hay que programar para aprovechar la concurrencia?

- Utilizando *threads* (o procesos).

- Utilizando procesos en diferentes procesadores que se comunican por mensajes (MPS, *message passing system*).

Las API más comunes hoy en día son PVM y MPI y además no limitan la posibilidad de utilizar *threads* (aunque a nivel local) y tener concurrencia entre procesamiento y entrada/salida. En cambio, en una máquina de memoria compartida (SHM, *shared memory*) sólo es posible utilizar *threads* y tiene el problema grave de la escalabilidad, ya que todos los procesadores utilizan la misma memoria y el número de procesadores en el sistema está limitado por el ancho de banda de la memoria.

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

- Utilizando un modelo cliente-servidor y programando a bajo nivel (*sockets*).

- El mismo modelo, pero con API de "alto" nivel (PVM, MPI).

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

Paralelización de programas:

La ley de Amdahl afirma que el incremento de velocidad (*speedup*) está limitado por la fracción de código (f) que puede ser paralelizado: $speedup = 1/(1-f)$

Esta ley implica que una aplicación secuencial $f = 0$ y el $speedup = 1$, con todo el código paralelo $f = 1$ y $speedup = \infty$, con valores posibles, 90% del código paralelo significa un $speedup = 10$ pero con $f = 0{,}99$ el $speedup = 100$. Esta limitación se puede evitar con algoritmos escalables y diferentes modelos de aplicación:

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

- *Single process multiple data* (SPMD): Mismo programa que se ejecuta con diferentes conjuntos de datos.

- Funcional: Varios programas que realizan una función diferente en la aplicación.

PVM (Parallel Virtual Machine)

PVM es una API que permite generar, desde el punto de vista de la aplicación, una colección dinámica de ordenadores, que constituyen una máquina virtual (VM).

Las tareas pueden ser creadas dinámicamente (*spawned*) y/o eliminadas (*killed*) y cualquier tarea PVM puede enviar un mensaje a otra.

El modelo soporta: Tolerancia a fallos, control de recursos, control de procesos, heterogeneidad en las redes y en los *hosts*.

El sistema (VM) dispone de herramientas para el control de recursos (agregar o quitar *hosts* de la máquina virtual), control de procesos (creación/eliminación dinámica de procesos), diferentes modelos de comunicación (*blocking send*, *blocking*/*nonblocking receive*, *multicast*), grupos de tareas dinámicos (una tarea puede anexarse a un grupo o no dinámicamente) y tolerancia a fallos (la VM detecta el fallo y se puede reconfigurar).

La estructura de PVM se basa por un lado en el *daemon* (pvm3d) que reside en cada máquina y se interconectan utilizando UDP, y por el otro, la biblioteca de PVM (libpvm3.a), que contiene todas las rutinas para enviar/recibir mensajes, crear/eliminar procesos, grupos, sincronización, etc. y que utilizará la aplicación distribuida.

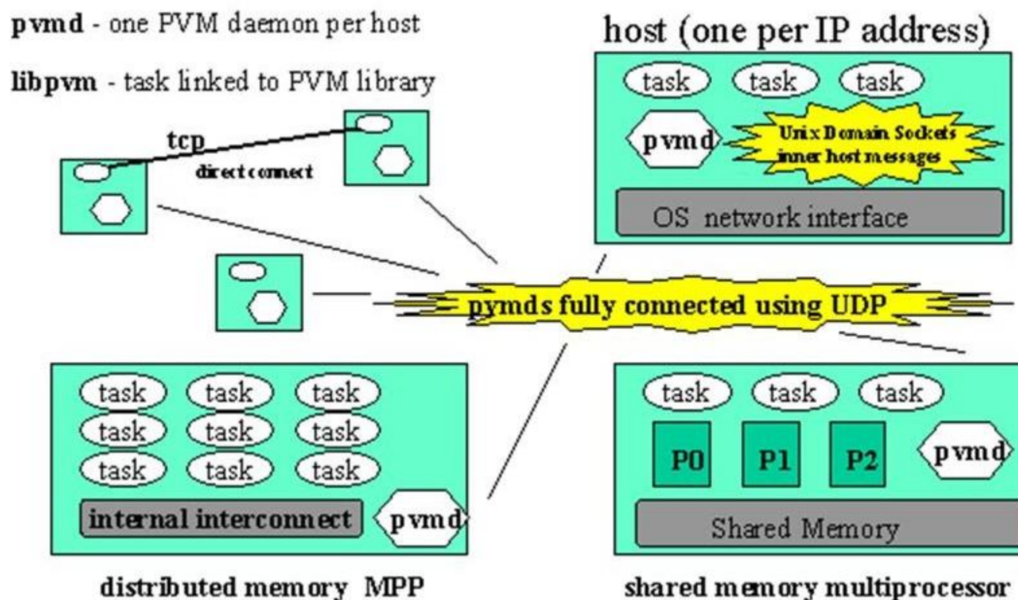
PVM dispone de una consola (pvm) que permite poner en marcha el *daemon*, crear la VM, ejecutar aplicaciones, etc.

Funcionamiento de PVM:

Al invocar la instrucción `pvm` se inicia la consola PVM y se inicia el servicio PVM `pvmd` .

Al añadir nodos al clúster, PVM utiliza SSH para arrancar remotamente el servicio `pvmd` en el nodo que se quiere añadir (Es deseable tener configurado el SSH sin contraseña para no tener que introducir la contraseña cada vez que se añade un nodo al clúster).

A continuación se muestra una imagen obtenida mediante la página oficial del PVM :



Hay un demonio `pvmd` ejecutándose en cada nodo.

El proceso demonio `pvmd` coordina los hosts de la máquina virtual, proporciona los mecanismos de comunicación y control de procesos que necesitan los programas de usuario PVM.

Los daemons `pvmd` se comunican utilizando UDP.

Los procesos PVM:

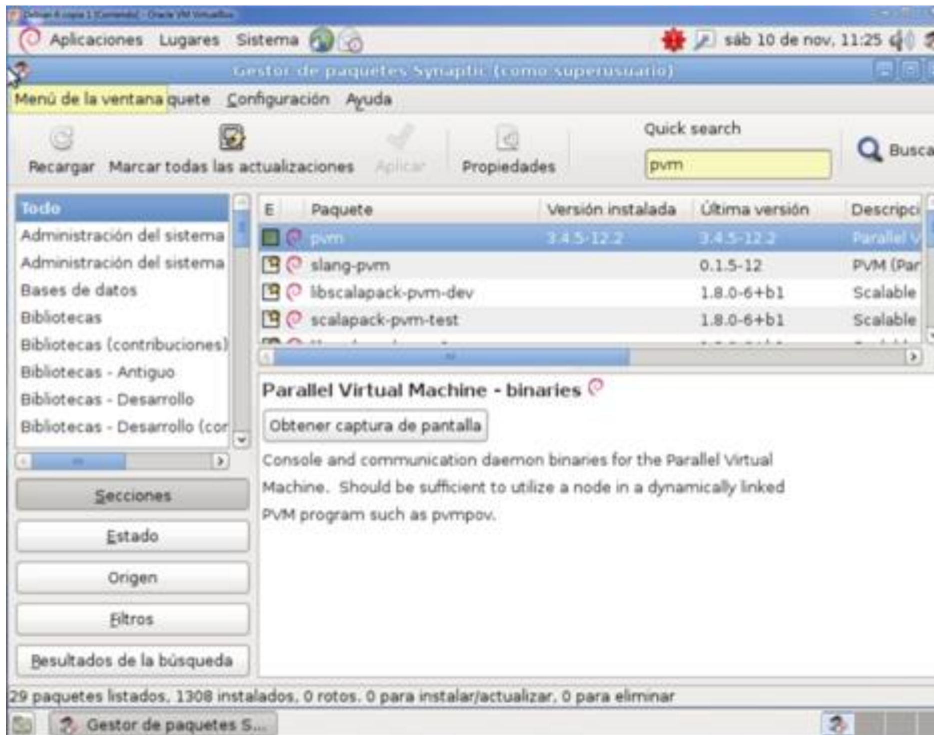
-Si son de nodos diferentes: Se comunican directamente utilizando TCP .

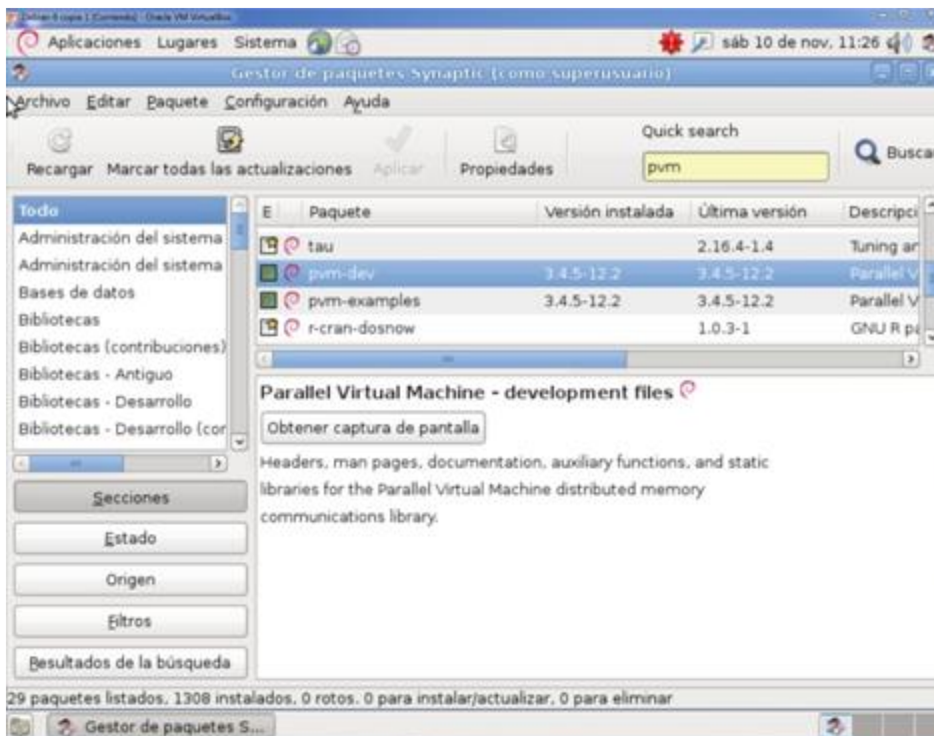
- Si son del mismo nodo: Se comunican utilizando la comunicación interna a través del demonio `pvmd` , la memoria compartida o mediante Sokets.

Instalación y configuración de PVM:

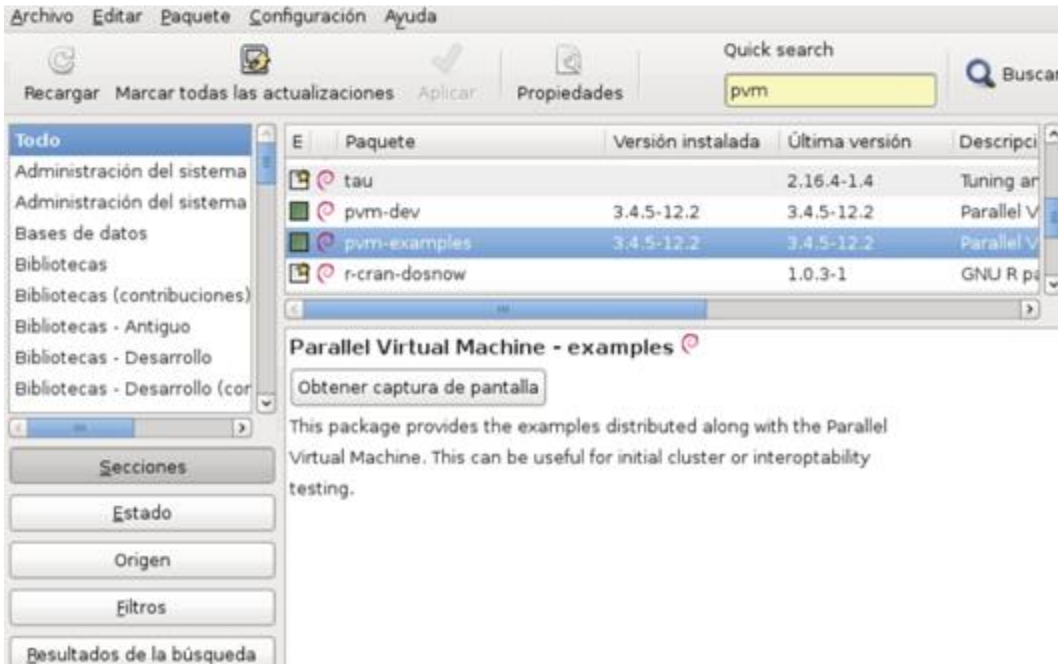
Dentro del directorio `/home/josemanuel/jm` hay que crear un nuevo directorio llamado `pvm` para las pruebas de este apartado.

Hay que instalar los paquetes pvm , pvm-dev:





También hay que instalar el paquete pvm-examples, para que se instalen también los archivos que contienen programas de ejemplo que utilizan PVM y así poder probar que todo funciona correctamente.



Es necesario modificar el archivo `.bashrc` que se encuentra en el directorio `/home/josemanuel` para establecer el valor de algunas variables de entorno:

```
.bashrc

# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
# for examples

#variables PVM
PVM_PATH=/home/josemanuel/jm/pvm
PVM_ROOT=/usr/lib/pvm3
PVM_ARCH="LINUX64"
PATH=$PATH:$PVM_ROOT:$PVM_ROOT/bin:$PVM_PATH:/usr/bin:/usr/lib:/usr/lib/pvm3
PVM_RSH=/usr/bin/ssh
export PVM_PATH PVM_ROOT PVM_ARCH PATH PVM_RSH PATH



# If not running interactively, don't do anything
[ -z "$PS1" ] && return

# don't put duplicate lines in the history. See bash(1) for more options
# don't overwrite GNU Midnight Commander's setting of `ignorespace'.
HISTCONTROL=$HISTCONTROL${HISTCONTROL+:}ignoredups
# ... or force ignoredups and ignorespace
```

- La variable de entorno `PVM_PATH` indica el directorio donde están los programas que ejecuta el usuario, en nuestro caso: `/home/josemanuel/pvm`
- La variable de entorno `PVM_ROOT` indica el directorio donde están las librerías PVM.
- La variable de entorno `PVM_ARCH` indica la arquitectura del procesador, en este caso es de 64 bits.
- La variable de entorno `PVM_RSH` estableciéndole el valor `/usr/bin/ssh` indica que se utilice SSH para invocar los comandos remotos.

A la variable de entorno `PATH` se le añaden las rutas de los directorios:

- Directorio donde se encuentran los binarios de PVM.

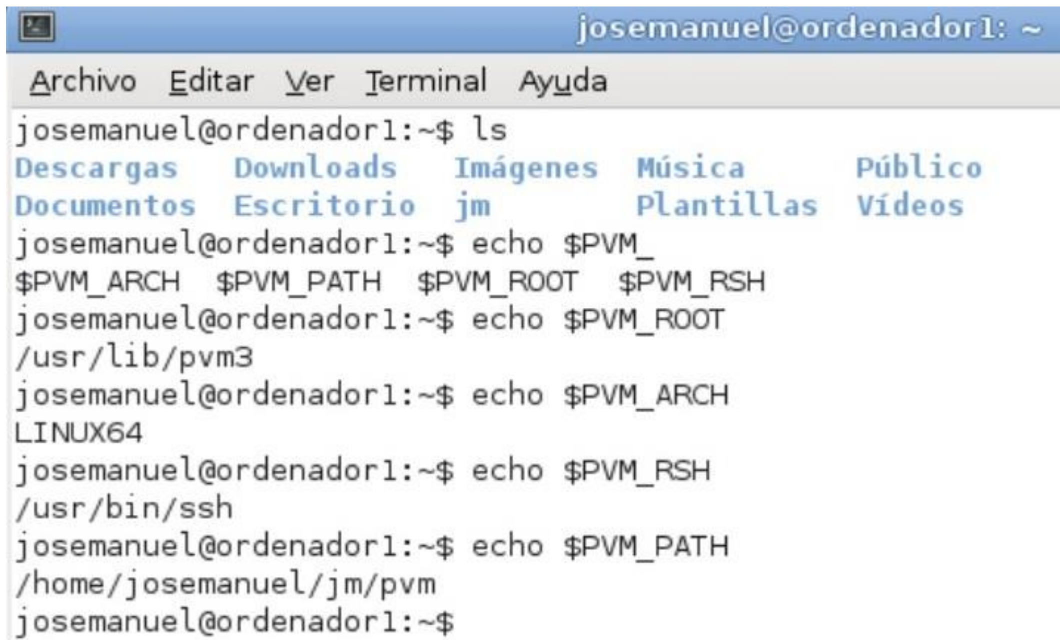
- Directorio donde se encuentran las librerías de PVM.

- Directorio de trabajo donde están los programas

- `PVM_ROOT`

Una vez modificado el archivo `.bashrc` las próximas veces que se inicie la consola estarán las variables de entorno establecidas con los valores adecuados.

En la siguiente ilustración se muestra como comprobar los valores de las variables de entorno PVM:



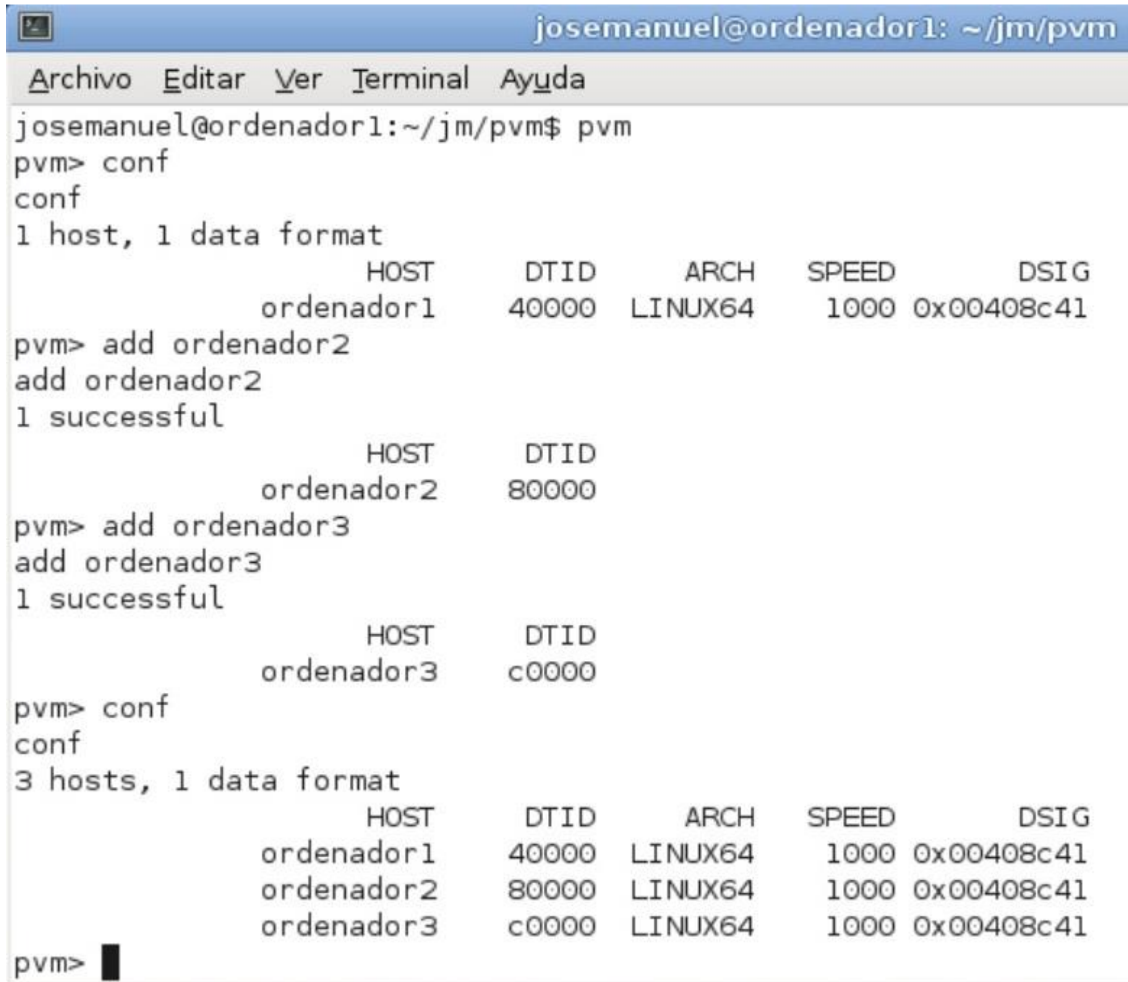
Las variables de entorno son las mismas en el ordenador1, en el ordenador2 y en el ordenador3 , ya que el fichero `.bashrc` (modificado anteriormente) es el mismo en los tres ordenadores porque todos, al iniciar el sistema, montan el mismo directorio `/home/josemanuel` mediante NFS.

PVM en funcionamiento:

Invocando el comando `pvm` se inicia la consola PVM y el demonio `pvmd`.

Con el comando `add` se puede añadir ordenadores al clúster (esto se muestra en la siguiente ilustración).

Los ordenadores que añadimos deben que tener instalados los paquetes pvm y pvm-dev .



Invocando el comando `conf` dentro de la consola PVM, se muestran los ordenadores que forman el clúster.

El ordenador1 (desde el cual se opera) no hace falta añadirlo (como se puede apreciar en la ilustración) a este ordenador se le puede llamar “master”.

Con el comando `quit` se puede salir de la consola PVM pero el demonio `pvmd` seguirá ejecutándose (en el ordenador local y en los ordenadores que hayamos añadido).

Se puede apagar el demonio `pvmd` invocado la instrucción `halt` desde la consola PVM.

Los ordenadores se pueden añadir utilizando un fichero que contenga los nombres de los ordenadores a añadir.

Se puede iniciar el demonio `pvmd` sin iniciar la consola invocando el comando `pvmd`. A este comando se le puede pasar como argumento la ruta de un archivo que contenga el nombre de los ordenadores a añadir (un ordenador por línea).

Estructura de un programa con PVM:

Para poder utilizar las rutinas de PVM es necesario incluir el fichero `pvm3.h`

Inicialización de PVM:

Este paso es necesario en cualquier programa que emplee PVM. La inicialización comprende los siguientes pasos:

-Enrolar los procesos en PVM obteniendo el identificador de proceso mediante la función `pvm_mytid()`

-Arrancar el resto de procesos que van a trabajar en paralelo. Esto se consigue mediante la función `pvm_spawn()` Esta función crea copias del proceso que se le indique en el mismo o diferentes computadores de la máquina virtual. Una de las informaciones retornadas es un vector con los identificadores de los procesos creados. Suele ser habitual enviar ese vector a todos los procesos, de esa forma, todos los procesos conocerán los identificadores de todos los demás; esto será útil a la hora de enviar o recibir mensajes.

Bloque de cálculo:

Este bloque dependerá del tipo de problema. Sin embargo, los cálculos deberán incluir el envío y recepción de mensajes. Por cada mensaje enviado, se deben efectuar las siguientes acciones:

Limpiar el buffer de mensaje enviado. Ello se lleva a cabo mediante la función `pvm_initsend()`. Esto es necesario hacerlo para cada mensaje porque si no, los mensajes se acumularán.

-Empaquetar todos los objetos que se vayan a enviar en el mensaje. Para efectuar esta tarea existe un conjunto de funciones que van incorporando objetos al mensaje. Existe una función diferente para cada tipo de objeto. Estas funciones de empaquetamiento se llaman genéricamente `pvm_pk_<tipo>` donde <tipo> es uno de los tipos simples del lenguaje C.

-Proceder al envío del mensaje indicando el proceso al que va dirigido. Esto puede hacerse con la función `pvm_send()` si el mensaje va dirigido a un solo proceso o mediante `pvm_mcast()` si los destinatarios son varios procesos.

De forma análoga, aunque inversa, las acciones que deben efectuarse por cada mensaje recibido son las siguientes:

-Recibir el mensaje. Esto se hace mediante la función `pvm_recv()`.

-Desempaquetar el contenido del mensaje extrayendo los objetos en el mismo orden en que se empaquetaron. Para ello existen unas rutinas de desempaquetamiento cuyo nombre genérico es `pvm_upk<tipo>`.

Terminar la ejecución del programa:

Antes de devolver el control al sistema operativo, el programa debe informar al `pvmd` que el proceso va a terminar. Esto se hace mediante la funcion `pvm_exit()` con lo que se liberarán los recursos que el proceso PVM haya utilizado.

Ejemplos de prueba:

En el paquete pvm-exemples se pueden encontrar ejemplos de programas escritos en C que utilizan PVM. A continuación se mostrará los 3 tres primeros ejemplos de programación PVM contenidos en este paquete, que son: Un programa tipo “hola mundo” , un programa tipo “maestro - esclavos” y un programa del tipo “Single Program Multiple Data” (Un programa múltiples datos)

El primer ejemplo es un programa del tipo hello:

El programa `hello.c` es un programa escrito en lenguaje C y utiliza la librería pvm3.h :

hello.c

```
#include <stdio.h>
#include "pvm3.h"

main()
{
     int cc, tid;
     char buf[100];




/*imprime por pantalla el identificador de proceso pvm*/

     printf("i'm t%x\n", pvm_mytid());

/*pvm_spawn se utiliza para lanzar nuevos procesos, en este caso el
procesos es "hello_other", en el ultimo parámetro se le pasa una variable
de tipo int que almacena el identificador del nuevo proceso lanzado*/

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




     if (cc == 1) {

/*pvm_rev se utiliza para recibir datos de otro proceso, los valores -1,-
1 indican que se recibirá cualquier mensaje de cualquier proceso*/

          cc = pvm_recv(-1, -1);
          pvm_bufinfo(cc, (int*)0, (int*)0, &tid);

/*las funciones del tipo pvm_upkTIPO se utilizan para desempaquetar
datos, en este caso se desempaqueta el dato recibido anteriormente en el
vector de caracteres buf */
          pvm_upkstr(buf);
          printf("from t%x: %s\n", tid, buf);

     } else
          printf("can't start hello_other\n");

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

hello_other.c

```
#include "pvm3.h"

main()
{
      int ptid;
      char buf[100];

      /*ptid almacena el identificador de proceso del padre (del que ha
invocado la función pvm_spawn) */

      ptid = pvm_parent();

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

/*antes de enviar datos hay que iniciar el buffer, esto se hace con la
función siquiente/
      pvm_initsend(PvmDataDefault);
/*antes de enviar un dato hay que empaquetarlo, en este caso es un string
y se empaqueta con la función siguiente*/
      pvm_pkstr(buf);

/*se envia el datos al proceso padre, el numero 1 es un identificador de
mensaje pera poder identificarlo al recibirlo*/
      pvm_send(ptid, 1);

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

En la siguiente ilustración se muestra como compilar el programa `hello.c` y de la misma forma hay que compilar el programa `hello_other.c` para que estos ejemplos funcionen.

```
josemanuel@ordenador1: ~/jm/pvm
Archivo  Editar  Ver  Terminal  Ayuda
josemanuel@ordenador1:~/jm/pvm$ ls
hello     hello_other     j         master1.c   Readme   slave1.c   spmd.c
hello.c   hello_other.c   master1   pvm.hosts   slave1   spmd
josemanuel@ordenador1:~/jm/pvm$ gcc hello.c -o hello -lpvm3
```

En la ilustración siguiente se muestra la ejecución del programa de ejemplo `hello.c` el cual utiliza `hello_other.c`

PVM distribuye la ejecución entre las diversas máquinas disponibles.

```
josemanuel@ordenador1: ~/jm/pvm
Archivo  Editar  Ver  Terminal  Ayuda
josemanuel@ordenador1:~/jm/pvm$ cat pvm.hosts
ordenador2
ordenador3
josemanuel@ordenador1:~/jm/pvm$ pvm pvm.hosts
pvm> conf
conf
3 hosts, 1 data format
                    HOST     DTID     ARCH   SPEED       DSIG
              ordenador1    40000  LINUX64    1000 0x00408c41
              ordenador2    80000  LINUX64    1000 0x00408c41
              ordenador3    c0000  LINUX64    1000 0x00408c41
pvm> quit
quit

Console: exit handler called
pvmd still running.
josemanuel@ordenador1:~/jm/pvm$ ./hello
i'm t40002
from t80001: hello, world from ordenador2
josemanuel@ordenador1:~/jm/pvm$ ./hello
i'm t40003
from tc0001: hello, world from ordenador3
josemanuel@ordenador1:~/jm/pvm$ ./hello
i'm t40004
from t40005: hello, world from ordenador1
josemanuel@ordenador1:~/jm/pvm$
```

La ejecución del mismo programa `hello` pero con un único nodo, el master:

```
josemanuel@ordenador1:~/jm/pvm$ ls
hello    hello_other    j        master1.c  Readme  slave1.c  spmd.c
hello.c  hello_other.c  master1  pvm.hosts  slave1  spmd
josemanuel@ordenador1:~/jm/pvm$ pvm
pvmd already running.
pvm> conf
conf
1 host, 1 data format
                    HOST     DTID     ARCH   SPEED       DSIG
              ordenador1    40000  LINUX64    1000 0x00408c41
pvm> quit
quit

Console: exit handler called
pvmd still running.
josemanuel@ordenador1:~/jm/pvm$ ./hello
i'm t4001b
from t4001c: hello, world from ordenador1
josemanuel@ordenador1:~/jm/pvm$ ./hello
i'm t4001d
from t4001e: hello, world from ordenador1
josemanuel@ordenador1:~/jm/pvm$ ./hello
i'm t4001f
from t40020: hello, world from ordenador1
josemanuel@ordenador1:~/jm/pvm$ ./hello
i'm t40021
from t40022: hello, world from ordenador1
josemanuel@ordenador1:~/jm/pvm$
```

El segundo ejemplo que proporciona el paquete pvm-exemples , es un programa PVM del tipo maestro trabajador (o servidor esclavo) donde el servidor crea los hijos, les envía datos, éstos cooperan entre ellos para resolver la tarea, circulando los datos entre los hijos (el de la izquierda recibe un dato, lo procesa y se lo envía al de la derecha) mientras el padre espera que cada hijo termine.

master1.c

```
#include <stdio.h>
#include "pvm3.h"
#define SLAVENAME "slave1"

main()
{
    /* mi identificador de proceso */
    int mytid;

    /* identificadores de los procesos esclavos */
    int tids[32];

    int n, nproc, numt, i, who, msgtype, nhost, narch;
    float data[100], result[32];
    struct pvmhostinfo *hostp;
/*al invocar cualquier funcion pvm el proceso queda inscrito en pvm*/
```

```
/* se inscribe en pvm*/
    mytid = pvm_mytid();

    /* establece el numero de esclavos para iniciarlos*/

    pvm_config( &nhost, &narch, &hostp );
    nproc = nhost * 3;
    if( nproc > 32 ) nproc = 32 ;
     printf("Spawning %d worker tasks ... " , nproc);

    /* se inician los procesos esclavos con la función pvm_spawn*/
    numt=pvm_spawn(SLAVENAME, (char**)0, 0, "", nproc, tids);
    if( numt < nproc ){
       printf("\n Trouble spawning slaves. Aborting. Error codes
are:\n");
       for( i=numt ; i<nproc ; i++ ) {
          printf("TID %d %d\n",i,tids[i]);
       }
       for( i=0 ; i<numt ; i++ ){
          pvm_kill( tids[i] );
       }
       pvm_exit();
       exit(1);
    }
     printf("SUCCESSFUL\n");



    n = 100;
    /* se inicializan los datos*/
    for( i=0 ; i<n ; i++ ){
       data[i] = 1.0;
    }

/* Se envia a todos los procesos esclavos los datos iniciales */
/* Para ello primero se empaquetan los datos y luego se envían */

     pvm_initsend(PvmDataDefault);
     pvm_pkint(&nproc, 1, 1);
     pvm_pkint(tids, nproc, 1);
     pvm_pkint(&n, 1, 1);
     pvm_pkfloat(data, n, 1);
    pvm_mcast(tids, nproc, 0);

    /* se espera los resultados de los eclavos */
    msgtype = 5;
    for( i=0 ; i<nproc ; i++ ){
       pvm_recv( -1, msgtype );
       pvm_upkint( &who, 1, 1 );
       pvm_upkfloat( &result[who], 1, 1 );
       printf("I got %f from %d; ",result[who],who);
       if (who == 0)
                printf( "(expecting %f)\n", (nproc - 1) * 100.0);
       else
                printf( "(expecting %f)\n", (2 * who - 1) * 100.0);

    }
    /* el programa finaliza y detiene pvm antes de finalizar */
```

```
    pvm_exit();
}
```

slave1.c

```
#include <stdio.h>
#include "pvm3.h"

main()
{
    int mytid;       /* mi identificador de proceso pvm */
    int tids[32];    /* identificadores de procesos   */
     int n, me, i, nproc, master, msgtype;
     float data[100], result;
    float work();

    /* se inscribe en pvm */
     mytid = pvm_mytid();


    /* se reciben los datos del master */
/* para ello se hace la recepción y luego se desempaquetan los datos*/
    msgtype = 0;
    pvm_recv( -1, msgtype );
     pvm_upkint(&nproc, 1, 1);
     pvm_upkint(tids, nproc, 1);
     pvm_upkint(&n, 1, 1);
     pvm_upkfloat(data, n, 1);


    /*se determina que proceso esclavo soy*/
    for( i=0; i<nproc ; i++ )
       if( mytid == tids[i] ){ me = i; break; }

    /* se hacen los calculos con los datos */
    result = work( me, n, data, tids, nproc );




/*se empaquetan los datos y se envían al master*/
    pvm_initsend( PvmDataDefault );
    pvm_pkint( &me, 1, 1 );
    pvm_pkfloat( &result, 1, 1 );
    msgtype = 5;
    master = pvm_parent();
    pvm_send( master, msgtype );

    /* el programa finaliza y detiene pvm antes de finalizar */
    pvm_exit();
}
```

```
Float work(me, n, data, tids, nproc )

/*es un ejemplo sencillo donde los procesos esclavos
 intercambian datos con el proceso anterior*/
    int me, n, *tids, nproc;
    float *data;
{
    int i, dest;
    float psum = 0.0;
    float sum = 0.0;
    for( i=0 ; i<n ; i++ ){
       sum += me * data[i];
    }
    /* ilustra la comunicación nodo a nodo */
    pvm_initsend( PvmDataDefault );
    pvm_pkfloat( &sum, 1, 1 );
    dest = me+1;
    if( dest == nproc ) dest = 0;
    pvm_send( tids[dest], 22 );
    pvm_recv( -1, 22 );
    pvm_upkfloat( &psum, 1, 1 );

    return( sum+psum );}
```

Antes de ejecutar el programa se añade el ordenador2 y 3 al clúster:

```
josemanuel@ordenador1:~/jm/pvm$ cat pvm.hosts
ordenador2
ordenador3
josemanuel@ordenador1:~/jm/pvm$ pvm pvm.hosts
pvm> conf
conf
3 hosts, 1 data format
                    HOST     DTID     ARCH   SPEED       DSIG
              ordenador1    40000  LINUX64    1000 0x00408c41
              ordenador2    80000  LINUX64    1000 0x00408c41
              ordenador3    c0000  LINUX64    1000 0x00408c41
pvm> quit
quit
```

Se muestra la ejecución del programa `master1`, el cual necesita el programa `slave1`:

```
josemanuel@ordenador1:~/jm/pvm$ ./master1
Spawning 9 worker tasks ... SUCCESSFUL
I got 1500.000000 from 8; (expecting 1500.000000)
I got 1300.000000 from 7; (expecting 1300.000000)
I got 700.000000 from 4; (expecting 700.000000)
I got 100.000000 from 1; (expecting 100.000000)
I got 1100.000000 from 6; (expecting 1100.000000)
I got 300.000000 from 2; (expecting 300.000000)
I got 900.000000 from 5; (expecting 900.000000)
I got 500.000000 from 3; (expecting 500.000000)
I got 800.000000 from 0; (expecting 800.000000)
josemanuel@ordenador1:~/jm/pvm$ pvm
pvmd already running.
pvm> delete ordenador3
delete ordenador3
1 successful
                    HOST   STATUS
              ordenador3   deleted
pvm> conf
conf
2 hosts, 1 data format
                    HOST     DTID     ARCH    SPEED       DSIG
              ordenador1    40000  LINUX64     1000 0x00408c41
              ordenador2    80000  LINUX64     1000 0x00408c41
pvm>
```

En la imagen anterior se muestra como se elimina el ordenador3 del clúster invocando la instrucción `delete` desde la consola PVM.

A continuación se muestra cual es el resultado de ejecutar el programa master / slave con PVM configurado con dos ordenadores:

```
josemanuel@ordenador1:~/jm/pvm$ ./master1
Spawning 6 worker tasks ... SUCCESSFUL
I got 500.000000 from 3; (expecting 500.000000)
I got 300.000000 from 2; (expecting 300.000000)
I got 900.000000 from 5; (expecting 900.000000)
I got 700.000000 from 4; (expecting 700.000000)
I got 100.000000 from 1; (expecting 100.000000)
I got 500.000000 from 0; (expecting 500.000000)
josemanuel@ordenador1:~/jm/pvm$
```

A continuación se elimina el ordenador2 y se muestra la ejecución:

```
josemanuel@ordenador1:~/jm/pvm$ pvm
pvmd already running.
pvm> conf
conf
2 hosts, 1 data format
                    HOST     DTID     ARCH   SPEED       DSIG
              ordenador1    40000  LINUX64    1000 0x00408c41
              ordenador2    80000  LINUX64    1000 0x00408c41
pvm> delete ordenador2
delete ordenador2
1 successful
                    HOST  STATUS
              ordenador2  deleted
pvm> conf
conf
1 host, 1 data format
                    HOST     DTID     ARCH   SPEED       DSIG
              ordenador1    40000  LINUX64    1000 0x00408c41
pvm> quit
quit

Console: exit handler called
pvmd still running.
josemanuel@ordenador1:~/jm/pvm$ ./master1
Spawning 3 worker tasks ... SUCCESSFUL
I got 300.000000 from 2; (expecting 300.000000)
I got 100.000000 from 1; (expecting 100.000000)
I got 200.000000 from 0; (expecting 200.000000)
josemanuel@ordenador1:~/jm/pvm$
```

El siguiente ejemplo es un programa del tipo SPMD (Single Program Multiple Data, Un programa múltiples datos):

spmd.c

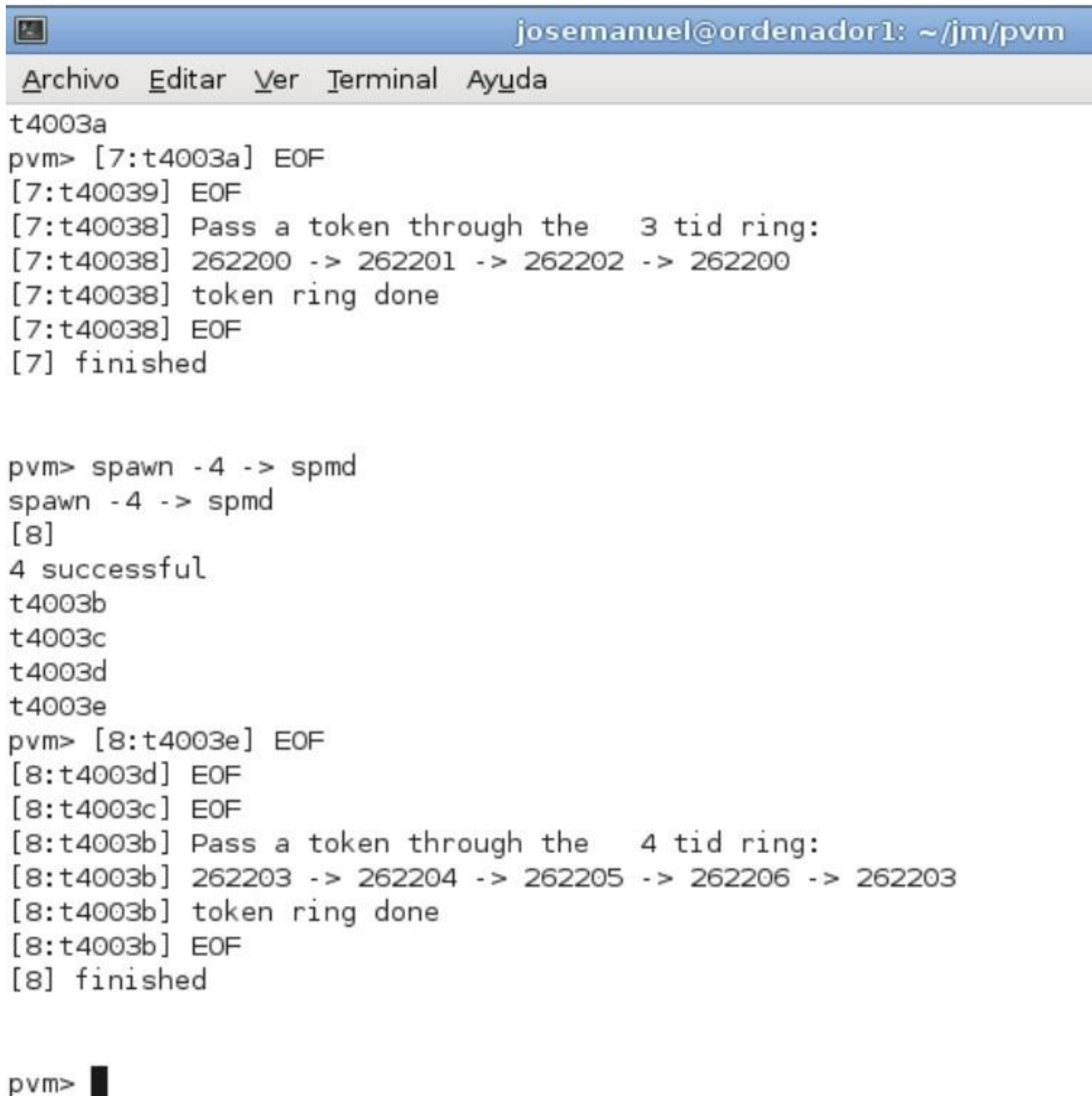
```
#include <stdio.h>
#include <sys/types.h>
#include "pvm3.h"
#define MAXNPROC 32

void dowork();

main()
{
    int mytid;                /* mi identificador de proceso pvm */
    int *tids;   /*array de identificadores de proceso */
    int me;                       /*mi numero de proceso */
    int i;
    int ntids;

    /* se inscribe en pvm */
    mytid = pvm_mytid();

/* utiliza la función pvm_siblings() para determinar el numero de
procesos (y sus identificadores) que se lanzaron con la función pvm_spawn
*/

      ntids = pvm_siblings(&tids);

      for (i = 0; i < ntids; i ++)
            if ( tids[i] == mytid)
            {
                  me = i;
                  break;
            }

      if (me == 0)
      {
            printf("Pass a token through the %3d tid ring:\n", ntids);
            for (i = 0; i < ntids; i ++)
            {
                  printf( "%6d -> ", tids[i]);
                  if (i % 6 == 0 && i > 0)
                        printf("\n");
            }
            printf("%6d \n", tids[0]);
      }
/*-------------------------------------------------------------------------
---*/
     dowork( me, ntids, tids );

     /* el  programa finaliza y detiene pvm */
     pvm_exit();
     exit(1);
}
```

```
/*ejemplo sencillo donde se van intercambiando datos recorriendo un
anillo simbólico formado por los procesos*/

void dowork( me, nproc, tids )
     int me;
     int nproc;
      int tids[];
{
     int token;
     int src, dest;
     int count  = 1;
     int stride = 1;
     int msgtag = 4;

     /* determina los extremos del anillo */
     if ( me == 0 )
          src = tids[nproc -1];
      else
          src = tids[me -1];

     if (me == nproc - 1)
        dest = tids[0];
     else
        dest = tids[me + 1];

     if( me == 0 )
     {
        token = dest;
        pvm_initsend( PvmDataDefault );
        pvm_pkint( &token, count, stride );
        pvm_send( dest, msgtag );
        pvm_recv( src, msgtag );
        printf("token ring done\n");
     }
     else
     {
        pvm_recv( src, msgtag );
        pvm_upkint( &token, count, stride );
        pvm_initsend( PvmDataDefault );
        pvm_pkint( &token, count, stride );
        pvm_send( dest, msgtag );
     }
}
```

A continuación se muestra la ejecución del programa `spmd` invocando el comando: `spawn -numero_de_procesos -> spmd` desde la consola PVM, donde número de procesos es el número de copias del proceso que se ejecutan simultáneamente.

Este programa suele ser arrancado desde la consola mediante el comando `spawn`.

Este programa utiliza la función `pvm_siblings()` de la librería PVM para obtener el número de procesos (y sus identificadores) arrancados con el comando `spawn`.



Los resultados de los programas de ejemplo son los esperados, PVM funciona correctamente.

A continuación se muestra un extracto del archivo "leeme" que documenta los ejemplos de programación PVM que contiene el paquete pvm-exemples:

```
PVM Version 3.4
 EXAMPLES
______________________________________________________________________

 This directory contains C and FORTRAN example programs using PVM 3.4

 All examples assume that pvm is installed and currently running.

 Each target can be made separately by typing in

       % aimk <target>

 Each section tells how to compile the example, run the example,
 and what output is expected. Some of the examples run slightly
 differently on MPPs (PGON and SP2MPI), the difference are noted,
 where applicable in the section "MPP notes".

 ===============================================================
 1.) hello + hello_other:
       Two programs that cooperate - shows how to create a new task
       and pass messages between tasks.

       Source files:
              hello.c hello_other.c

       To compile:
              % aimk hello hello_other

       Run from shell:
              % hello

     Run from pvm console:
              pvm> spawn -> hello

       Sample output:
              i'm t40002
              from t40003: hello, world from gollum.epm.ornl.gov

       MPP Notes:
              If you desire to run the hello example from the shell, then
              you need to compile the "helloh"
              % aimk helloh
              % helloh


 ===============================================================
```

```
2.) master1 + slave1, fmaster1 + fslave1
      A master/slave example where the master process creates and
      directs some number of slave processes that cooperate to do the
      work.  C and FORTRAN versions. [f]master1 will spawn 3*<#hosts>
      slave tasks on the virtual machine.

      Source files:
            master1.c slave1.c master1.f slave1.f

      To compile:
            % aimk master1 slave1 fmaster1 fslave1

      To run from shell (C version):
            % master1

      To run from shell (Fortran version)
            % fmaster1

      To Run from PVM console:
            pvm> spawn -> master1

            OR

            pvm> spawn -> fmaster1


      Sample output:
            Spawning 3 worker tasks ... SUCCESSFUL
        I got 100.000000 from 1; (expecting 100.000000)
        I got 200.000000 from 0; (expecting 200.000000)
        I got 300.000000 from 2; (expecting 300.000000)

      MPP Notes:
            If you desire to run the master examples from the shell,
then
            you need to compile the "master1h", fmaster1h"
            % aimk master1h fmaster1h
            % master1h


===============================================================
```

3.) spmd, fspmd

An SPMD (Single Program Multiple Data) example that uses pvm_siblings (pvmfsiblings) to determine the number of tasks (and their task ids) that were spawned together. The parallel computation then performs a simple token ring and passes a message. This program should be run from the pvm console.

Source Files:

```
spmd.c spmd.f
```

To compile:

```
% aimk spmd fspmd
```

To run from pvm console:

```
pvm> spawn -4 -> spmd

OR

pvm> spawn -4 -> fspmd
```

Sample output:

```
[4:t4000f]  Pass a token through the 4 tid ring:
[4:t4000f]  262159 ->  262160 ->  262161 ->  262162 ->  262159
[4:t4000f]  token ring done
```

Apagando nodos mientras ejecutan procesos PVM

El programa `prueba.c` puede utilizarse para comprobar que sucede al apagar máquinas cuando están ejecutando algún proceso PVM:

```
prueba.c

#include <stdio.h>
#include "pvm3.h"

main()
{

        printf("soy el proceso t%x\n", pvm_mytid());
        sleep(50);
        printf("el proceso:%x finaliza ejecucion\n", pvm_mytid());
        pvm_exit();
        //exit(0);
}
```

La siguiente imagen muestra la ejecución normal del programa (sin apagar ningún nodo) ejecutando tres copias del proceso `prueba` en paralelo (se ejecuta un proceso en cada ordenador):

```
josemanuel@ordenador1: ~/jm/pvm
Archivo  Editar  Ver  Terminal  Ayuda
                    HOST     DTID     ARCH   SPEED       DSIG
              ordenador1    40000  LINUX64    1000 0x00408c41
              ordenador2    80000  LINUX64    1000 0x00408c41
              ordenador3    c0000  LINUX64    1000 0x00408c41
pvm> spawn -3 -> prueba
spawn -3 -> prueba
[1]
3 successful
t80001
tc0001
t40002
pvm> [1:t40002] soy el proceso t40002
[1:t40002] el proceso:40002 finaliza ejecucion
[1:t40002] EOF
[1:tc0001] soy el proceso tc0001
[1:tc0001] el proceso:c0001 finaliza ejecucion
[1:tc0001] EOF
[1:t80001] soy el proceso t80001
[1:t80001] el proceso:80001 finaliza ejecucion
[1:t80001] EOF
[1] finished


pvm>
```

Se aprecia cómo los 3 procesos `prueba` acaban su ejecución normalmente.

A continuación se muestra la ejecución de 3 procesos `prueba` en paralelo (ejecutando un proceso en cada ordenador) pero en este caso apagando el ordenador3 (botón "apagar sistema") antes de que ninguno de los 3 procesos finalice la ejecución:

```
josemanuel@ordenador1: ~/jm/pvm
Archivo  Editar  Ver  Terminal  Ayuda
  unalias     Undefine command alias
  unexport    Remove environment variables from spawn export list
  version     Show libpvm version
pvm> conf
conf
3 hosts, 1 data format
                    HOST     DTID     ARCH    SPEED       DSIG
              ordenador1    40000  LINUX64     1000 0x00408c41
              ordenador2    80000  LINUX64     1000 0x00408c41
              ordenador3    c0000  LINUX64     1000 0x00408c41
pvm> spawn -3 -> prueba
spawn -3 -> prueba
[1]
3 successful
t40007
t80003
tc0003
pvm> [1:t40007] soy el proceso t40007
[1:t40007] el proceso:40007 finaliza ejecucion
[1:t40007] EOF
[1:t80003] soy el proceso t80003
[1:t80003] el proceso:80003 finaliza ejecucion
[1:t80003] EOF
```

Se puede apreciar que el proceso que se estaba ejecutando en el ordenador3 no acaba la ejecución y la consola se queda bloqueada.

MPI (Message Passing Interface)

La definición de la API de MPI ha sido el trabajo resultante del MPI Fourum (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 son: 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, de modo que permitiera la comunicación eficiente (*memory-to-memory copy*), cómputo y comunicación concurrente. Además, 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...), 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 la característica más relevante: la portabilidad. El MPP se puede ejecutar sobre máquinas multiprocesadores, redes de WS e incluso sobre máquinas de memoria compartida.

MPI es un standard definido por un consorcio de organizaciones, en estos textos se ha utilizado MPICH2 (que es la implementación más actual de MPI para Debian) .

MPICH2 es una implementación del estándar MPI 2.

Durante la elaboración de este estudio ha salido el estándar MPI 3.0 (en Diciembre de 2012).

Funcionamiento de MPICH2:

En estos textos hemos utilizado MPICH2 con el gestor de procesos por defecto MPD y con la configuración de los mecanismos de comunicación interna por defecto:

Device: CH3, Channel: Nemesis configurado con el módulo TCP .

Para crear el anillo de nodos hay que invocar la instrucción:

```
mpdboot –n num_nodos –f fichero_de_nodos
```

`mpdboot` arranca el servicio `mpd` en el nodo local y utiliza SSH para arrancar remotamente el servicio `mpd` en cada nodo que se quiere añadir al anillo .

(Es deseable tener configurado SSH sin contraseña para no tener que introducir la contraseña cada vez que se añade un nodo al anillo).

Los demonios `mpd` arrancados mediante SSH se conectan al demonio local.

Cada demonio funciona como un nano-kernel que proporciona paso de mensajes entre los procesos MPI.

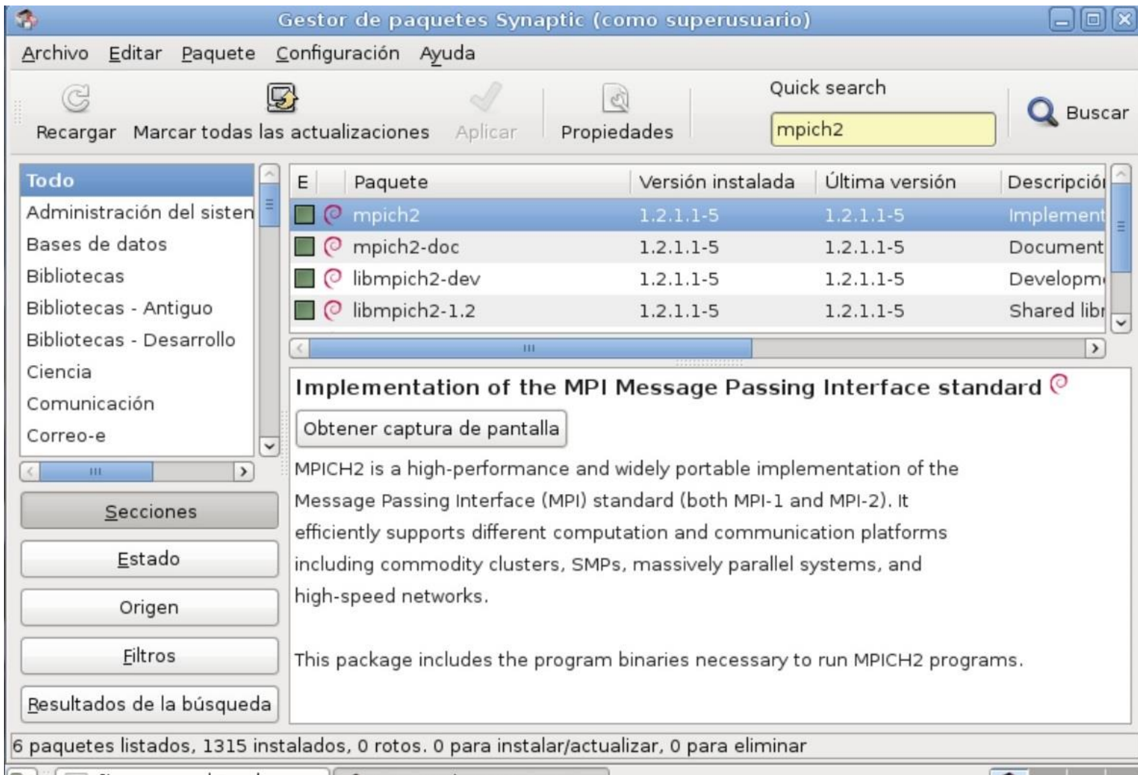
Las comunicaciones internas se hacen por TCP y aprovechando la memoria compartida.

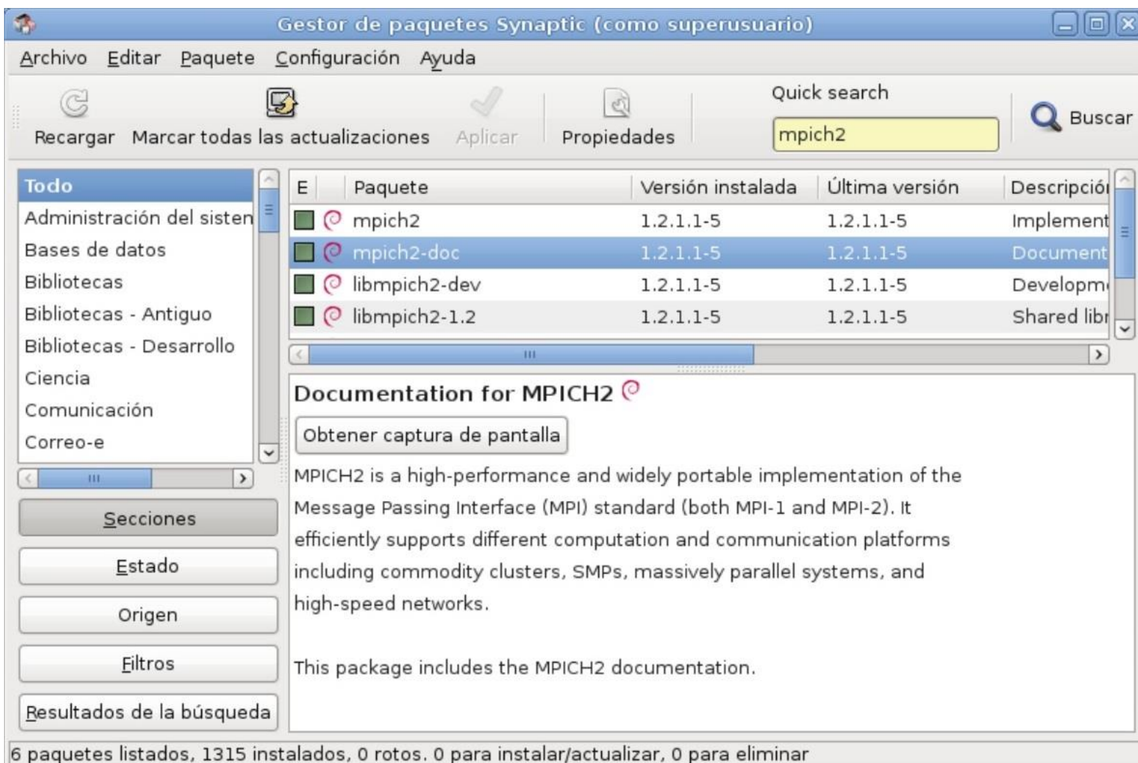
Las optimizaciones de memoria compartida están habilitadas por defecto para mejorar el rendimiento en plataformas multiprocesador / multinúcleo. Estas optimizaciones pueden deshabilitarse (a costa de rendimiento).

Instalación y configuración de MPICH2

Dentro del directorio `/home/josemanuel/jm` hay que crear un nuevo directorio llamado `mpich2` donde hacer las pruebas MPI y guardar los archivos relativos a este apartado.

Hay que instalar los paquetes mpich2, mpich2-doc, libmpich2-dev, libmpich2-1.2 en los tres ordenadores:





Gestor de paquetes Synaptic (como superusuario)

Archivo Editar Paquete Configuración Ayuda

Recargar | Marcar todas las actualizaciones | Aplicar | Propiedades | Quick search: mpich2 | Buscar

Todo
Administración del sisten
Bases de datos
Bibliotecas
Bibliotecas - Antiguo
Bibliotecas - Desarrollo
Ciencia
Comunicación
Correo-e

Secciones | Estado | Origen | Filtros | Resultados de la búsqueda

E	Paquete	Versión instalada	Última versión	Descripció
	mpich2	1.2.1.1-5	1.2.1.1-5	Implement
	mpich2-doc	1.2.1.1-5	1.2.1.1-5	Document
	libmpich2-dev	1.2.1.1-5	1.2.1.1-5	Developm
	libmpich2-1.2	1.2.1.1-5	1.2.1.1-5	Shared libr

Development files for MPICH2

Obtener captura de pantalla

MPICH2 is a high-performance and widely portable implementation of the Message Passing Interface (MPI) standard (both MPI-1 and MPI-2). It efficiently supports different computation and communication platforms including commodity clusters, SMPs, massively parallel systems, and high-speed networks.

This package includes the MPICH2 headers and static libraries, as well as the compiler wrappers needed to build MPICH2 programs.

6 paquetes listados, 1315 instalados, 0 rotos. 0 para instalar/actualizar, 0 para eliminar

Gestor de paquetes Synaptic (como superusuario)

Archivo Editar Paquete Configuración Ayuda

Recargar | Marcar todas las actualizaciones | Aplicar | Propiedades | Quick search: mpich2 | Buscar

Todo
Administración del sisten
Bases de datos
Bibliotecas
Bibliotecas - Antiguo
Bibliotecas - Desarrollo
Ciencia
Comunicación
Correo-e

Secciones | Estado | Origen | Filtros | Resultados de la búsqueda

E	Paquete	Versión instalada	Última versión	Descripció
	mpich2	1.2.1.1-5	1.2.1.1-5	Implement
	mpich2-doc	1.2.1.1-5	1.2.1.1-5	Document
	libmpich2-dev	1.2.1.1-5	1.2.1.1-5	Developm
	libmpich2-1.2	1.2.1.1-5	1.2.1.1-5	Shared libr

Shared libraries for MPICH2

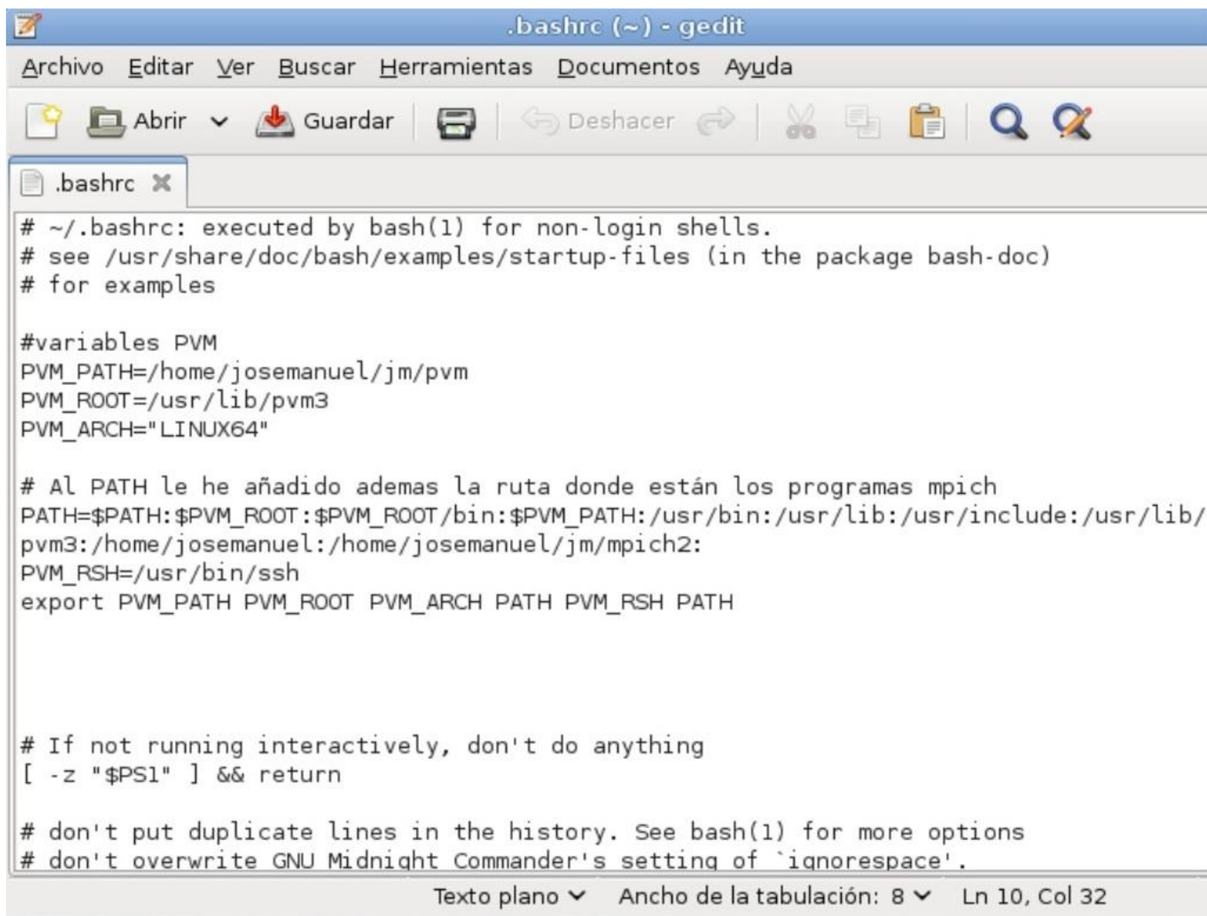
Obtener captura de pantalla

MPICH2 is a high-performance and widely portable implementation of the Message Passing Interface (MPI) standard (both MPI-1 and MPI-2). It efficiently supports different computation and communication platforms including commodity clusters, SMPs, massively parallel systems, and high-speed networks.

This package includes the MPICH2 shared libraries.

6 paquetes listados, 1315 instalados, 0 rotos. 0 para instalar/actualizar, 0 para eliminar

Hay que modificar el archivo `/home/josemanuel/.bashrc` para añadirle a la variable de entorno `$PATH`, la ruta donde se encuentran los programas MPI de prueba:



Hay que crear un archivo oculto llamado `.mpd.conf` en el directorio `home` del usuario: `/home/josemanuel` (recordemos que está compartido por NFS para los tres ordenadores del clúster), este archivo debe contener la cadena `MPD_SECRETWORD=palabra_secreta` donde `palabra_secreta` es una cadena que elige el usuario, en este caso `JM` como se aprecia en la ilustración.

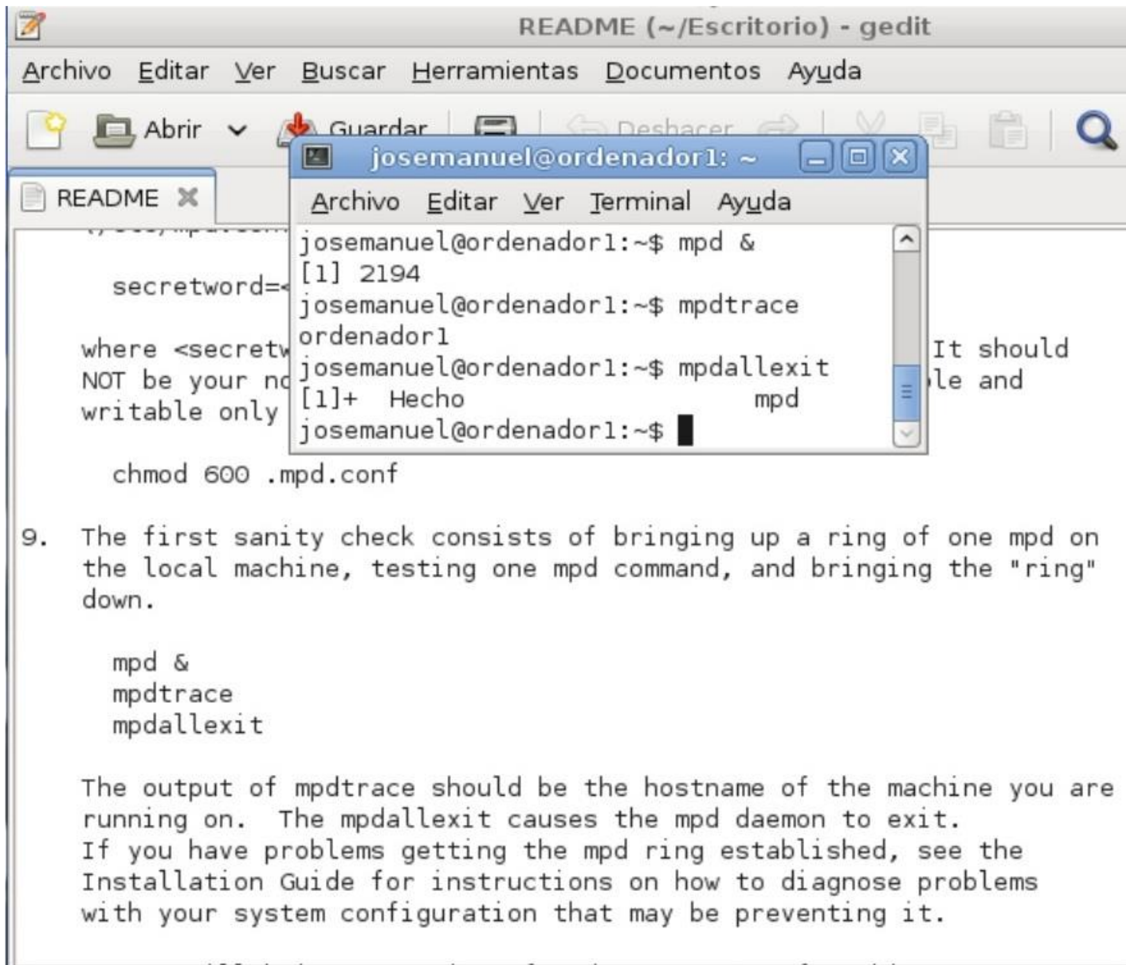
También hay que crear otro archivo oculto llamado `.mpd.hosts` con los nombres de máquina que forman el clúster, este archivo debe contener un nombre de maquina por línea. Los ordenadores se añaden al anillo, se le llama anillo porque MPI distribuye las tareas entre los ordenadores añadidos al anillo en orden de adicción, el primer proceso se ejecuta en el primer ordenador añadido, el segundo proceso en el segundo ordenador añadido, … y cuando llega al último vuelve a empezar por el primero.

```
josemanuel@ordenador1: ~
Archivo  Editar  Ver  Terminal  Ayuda
josemanuel@ordenador1:~$ ls -l .mpd*
-rw------- 1 josemanuel josemanuel 18 nov 17 17:10 .mpd.conf
josemanuel@ordenador1:~$ cat .mpd.conf
MPD_SECRETWORD=JM
josemanuel@ordenador1:~$ ls -l ./jm/mpich2/.mpd*
-rw-r--r-- 1 josemanuel josemanuel 33 nov 18 11:03 ./jm/mpich2/.mpd.hosts
josemanuel@ordenador1:~$ cat ./jm/mpich2/.mpd.hosts
ordenador1
ordenador2
ordenador3
josemanuel@ordenador1:~$
```

Para que el archivo `.mpd.conf` tenga los permisos adecuados hay que invocar la instrucción siguiente:

```
josemanuel@ordenador1: ~
Archivo  Editar  Ver  Terminal  Ayuda

josemanuel@ordenador1:~$ chmod 600 .mpd.conf
josemanuel@ordenador1:~$ ls .mpd.conf
.mpd.conf
josemanuel@ordenador1:~$ ls .mpd.conf -l
-rw------- 1 josemanuel josemanuel 18 nov 17 17:10 .mpd.conf
josemanuel@ordenador1:~$ cat .mpd.conf
MPD_SECRETWORD=JM
josemanuel@ordenador1:~$
```

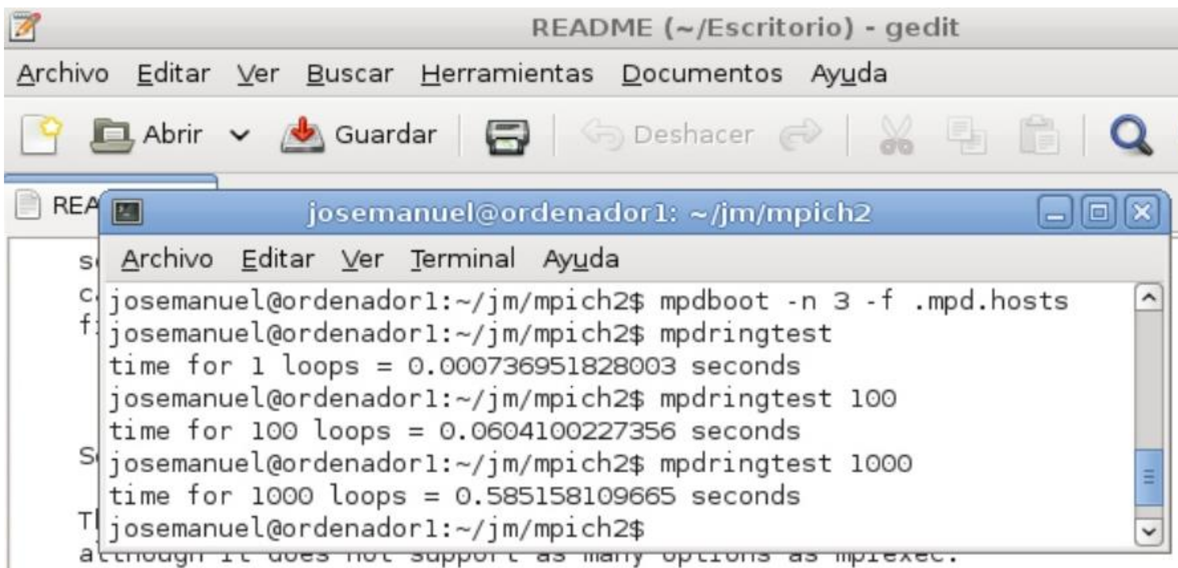
A continuación se muestran unos test para ver que MPI funciona correctamente:



Invocando el comando `mpd` se inicia el servicio MPI.

Invocando el comando `mpdallexit` se detienen todos servicios MPI de todos los ordenadores añadidos al anillo.

Invocando la instrucción `mpdringtest` se puede comprobar lo que tardan en comunicarse los ordenadores que forman el anillo, podemos elegir el número de *loops*:

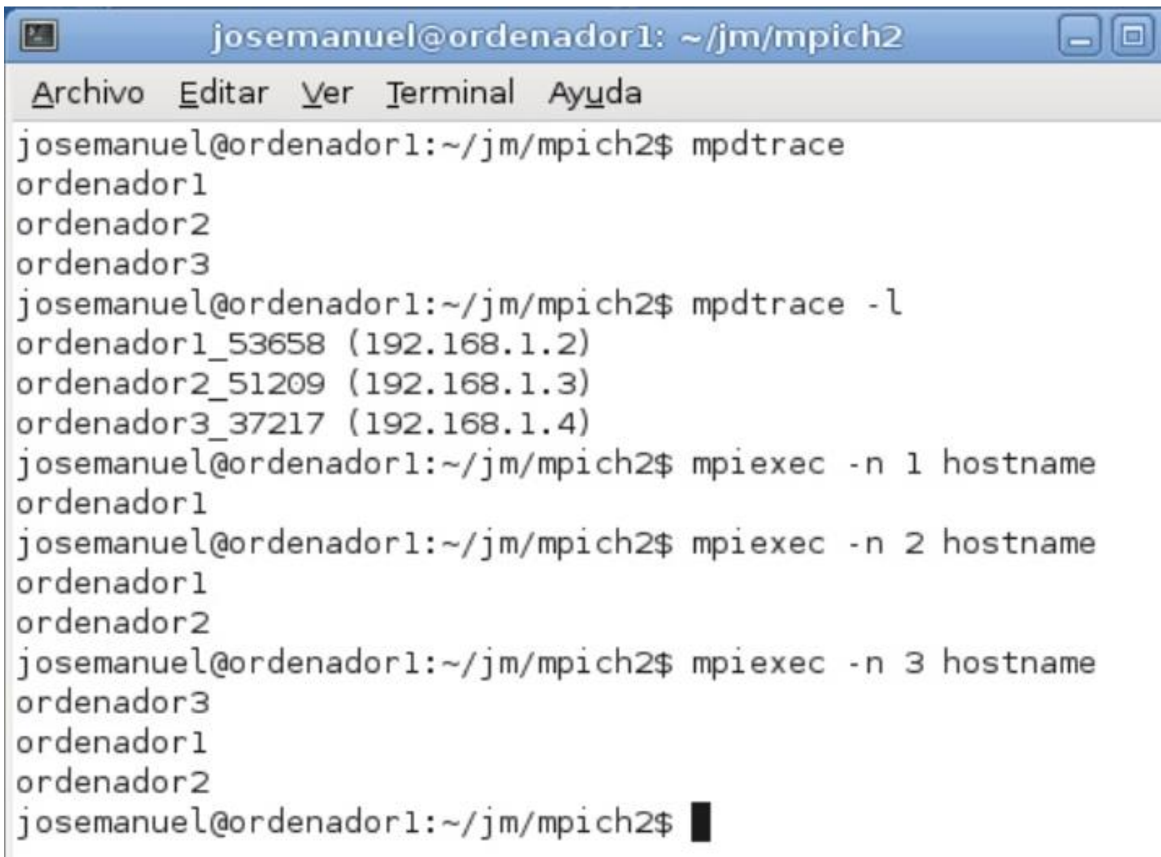


MPICH2 en funcionamiento:

Invocando la instrucción `mpdboot` como se muestra en la siguiente ilustración, se inicia el clúster MPI con los tres ordenadores añadidos al anillo en orden de adicción:

Invocando la instrucción `mpdtrace -l` se muestran los ordenadores que están añadidos al anillo clúster.

Para comprobar que MPI funciona correctamente se puede ejecutar la instrucción `hostname` con 1, 2 y 3 réplicas. MPI distribuye la ejecución de las instrucciones entre el anillo de ordenadores que forman el clúster, distribuye la ejecución de los 3 procesos `hostname` entre los ordenadores en orden de adicción al anillo:



Se puede ver en que ordenador se está ejecutando la instrucción, ya que `hostname` imprime por pantalla el nombre de máquina.

Estructura de un programa MPI:

La estructura normal de un programa MPI es la siguiente:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[]){
    /* Declaracion de variables */
    MPI_Init(&argc, &argv);
    /* Reparto de contenido */
    /* Bucle principal del programa */
    MPI_Finalize();


}
```

Ejemplos de prueba:

Este es un programa tipo hola mundo escrito en C que utiliza MPI:

```
#include <stdio.h>
#include "mpi.h"
int main( int argc, char *argv[] ){
    int numprocs,myid;
/* Inicializa la estructura de comunicación de MPI entre los procesos */
    MPI_Init(&argc,&argv);
/*se guarda en la variable numprocs el numero de procesos total que se
han creado en el ámbito MPI_COMM_WORLD este ámbito abarca todos los
procesos MPI en ejecución*/
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
/* se guarda en la variable  myid el identificador de proceso que está
invocando la función*/
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
/*se imprimen por pantalla  el numero de procesos total y myid*/
    printf("Hola mundo desde el proceso: %d de d\n",myid,numprocs);
/* Finaliza la comunicación paralela entre los procesos */
    MPI_Finalize();
    return 0;
}
```

A continuación se muestra como compilar el programa MPI `holamundo.c` y como ejecutarlo (en este caso se muestran 2 ejecuciones: Una creando 3 procesos `holamundo` y otra creando 6)

En los parámetros `-n numero` número indica el número de procesos que se tienen que ejecutar simultáneamente.

El programa siguiente muestra como enviar y recibir mensajes entre procesos utilizando MPI.

Este programa ha sido diseñado para ejecutar 2 réplicas del proceso simultáneamente: El primero con identificador 0 y el segundo con identificador 1:

```
#include <stdio.h>
#include "mpi.h"

int main( int argc, char *argv[] ){
    int numprocs,myid,value;
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    if (myid==0){
     value=100;
     printf("soy el proceso:%d\n",myid);
     printf("y voy a enviar el valor: %d al proceso 1 con el tag 0
\n\n",value);
/*realiza el envio de los datos contenidos en la variable "value"(un
entero) se lo envía al proceso con id 1, el mensaje lo envía con la
etiqueta 0 dentro del ámbito MPI_COMM_WORLD (el ámbito de todos los
procesos)*/
     MPI_Send(&value,1,MPI_INT,1,0,MPI_COMM_WORLD);
    }else{
/*recibe los datos del mensaje con etiqueta 0 en la variable "value" (1
entero) desde el proceso 0, en el mismo ámbito que en la función de envio
anterior, estatus es una variable tipo MPI_Status la cual almacena
información sobre la recepción */
     MPI_Recv(&value,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
```

```
/*se imprime por pantalla el identificador del proceso que invoca la
función "myid"*/
      printf("soy el proceso:%d\n",myid);
      printf("he recibido del proceso 0 con el tag 0 el valor:
%d\n",value);

    }

    MPI_Finalize();
    return 0;
}
```

A continuación se muestra como compilar y ejecutar el programa MPI `enviar_recibir.c` este programa ha sido diseñado para ejecutar 2 réplicas del proceso simultáneamente como se muestra a continuación:

```
josemanuel@ordenador1: ~/jm/mpich2
Archivo  Editar  Ver  Terminal  Ayuda
josemanuel@ordenador1:~/jm/mpich2$ ls
cpi     enviar_recibir.c   holamundo    mpd.hosts      srtest.c
cpi.c   hola               holamundo.c  README_mpich2
josemanuel@ordenador1:~/jm/mpich2$ mpicc enviar_recibir.c -o enviar_recibir
josemanuel@ordenador1:~/jm/mpich2$ mpirun.mpich2 -np 2 enviar_recibir
soy el proceso:0
y voy a enviar el valor: 100 al proceso 1 con el tag 0

soy el proceso:1
he recibido del proceso 0 con el tag 0 el valor: 100
josemanuel@ordenador1:~/jm/mpich2$
```

El programa `cpi.c` escrito en lenguaje C, realiza el cálculo del número Pi en paralelo por integración numérica, este programa se puede obtener instalando el paquete mpich2-doc, el cual contiene la documentación de Mpich2.

Este programa utiliza funciones explicadas anteriormente pero además utiliza MPI_Bcast y MPI_Reduce.

El programa utiliza la función `MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD)` para repartir el factor de precisión entre todos los procesos, cada proceso calcula su parte y después se tienen que sumar los resultados parciales de cada proceso obteniendo el resultado final el proceso 0 para mostrar el resultado por pantalla. La recolección y suma se hace con la función: `MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD)` con la cual el proceso 0 obtiene la suma de los resultados parciales de todos los procesos.

```
/*
   (C) 2001 by Argonne National Laboratory.
       See COPYRIGHT in top-level directory.
*/
#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 = 3.141592653589793238462643;
    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);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);
    fprintf(stderr,"Process %d on %s\n",
         myid, processor_name);

    n = 0;
    while (!done)
    {
        if (myid == 0)
        {
/*
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);
*/
         if (n==0) n=100; else n=0;

         startwtime = MPI_Wtime();
        }

/* se reparte el factor de precisión entre todos los procesos*/

        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        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;

/* después de que todos los procesos hayan hecho los cálculos se obtiene
la suma de los resultados de cada procesos*/

            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);

            if (myid == 0)
         {
```

```
                printf("pi is approximately %.16f, Error is %.16f\n",
                        pi, fabs(pi - PI25DT));
            endwtime = MPI_Wtime();
            printf("wall clock time = %f\n",
                    endwtime-startwtime);
           }
        }
    }
    MPI_Finalize();
    return 0;
}
```

A continuación se muestra como compilar el programa `cpi.c` y se muestra la ejecución de 8 réplicas del programa simultáneamente:

```
josemanuel@ordenador1: ~/jm/mpich2
Archivo  Editar  Ver  Terminal  Ayuda
josemanuel@ordenador1:~/jm/mpich2$ mpicc cpi.c -o cpi
josemanuel@ordenador1:~/jm/mpich2$ mpirun.mpich2 -np 8 cpi
Process 2 on ordenador3
Process 0 on ordenador1
Process 3 on ordenador1
Process 5 on ordenador3
Process 6 on ordenador1
Process 1 on ordenador2
Process 4 on ordenador2
Process 7 on ordenador2
pi is approximately 3.1416009869231245, Error is 0.0000083333333314
wall clock time = 0.013984
josemanuel@ordenador1:~/jm/mpich2$
```

A continuación se muestran pruebas ejecutando el programa con diferentes números de réplicas (1,2 y 3):

```
josemanuel@ordenador1: ~/jm/mpich2
Archivo  Editar  Ver  Terminal  Ayuda
josemanuel@ordenador1:~/jm/mpich2$ mpirun -n 1 cpi
Process 0 on ordenador1
pi is approximately 3.1416009869231254, Error is 0.0000083333333323
wall clock time = 0.000217
josemanuel@ordenador1:~/jm/mpich2$ mpirun -n 2 cpi
Process 0 on ordenador1
Process 1 on ordenador2
pi is approximately 3.1416009869231241, Error is 0.0000083333333309
wall clock time = 0.004126
josemanuel@ordenador1:~/jm/mpich2$ mpirun -n 3 cpi
Process 0 on ordenador1
Process 1 on ordenador2
Process 2 on ordenador3
pi is approximately 3.1416009869231254, Error is 0.0000083333333323
wall clock time = 0.010171
josemanuel@ordenador1:~/jm/mpich2$
```

Para ejecutar el programa `cpi` se puede invocar la instrucción `mpirun` o la instrucción `mpiexec` indistintamente y con los mismos argumentos, de hecho `mpirun` es un link a `mpiexec`.

A continuación se muestra la ejecución del programa `cpi` ejecutando 3, 6 y 9 procesos `cpi`.

```
josemanuel@ordenador1: ~/jm/mpich2
Archivo  Editar  Ver  Terminal  Ayuda
josemanuel@ordenador1:~/jm/mpich2$ mpicc cpi.c -o cpi
josemanuel@ordenador1:~/jm/mpich2$ mpirun.mpich2 -n 3 cpi
Process 0 on ordenador1
Process 1 on ordenador3
Process 2 on ordenador2
pi is approximately 3.1416009869231254, Error is 0.0000083333333323
wall clock time = 0.021826
josemanuel@ordenador1:~/jm/mpich2$ mpirun.mpich2 -n 6 cpi
Process 0 on ordenador1
Process 1 on ordenador3
Process 3 on ordenador1
Process 4 on ordenador3
Process 2 on ordenador2
Process 5 on ordenador2
pi is approximately 3.1416009869231249, Error is 0.0000083333333318
wall clock time = 0.010504
josemanuel@ordenador1:~/jm/mpich2$ mpirun.mpich2 -n 9 cpi
Process 0 on ordenador1
Process 1 on ordenador3
Process 2 on ordenador2
Process 4 on ordenador3
Process 6 on ordenador1
Process 5 on ordenador2
Process 3 on ordenador1
Process 7 on ordenador3
Process 8 on ordenador2
pi is approximately 3.1416009869231249, Error is 0.0000083333333318
wall clock time = 0.016850
josemanuel@ordenador1:~/jm/mpich2$ █
```

La correcta ejecución de este programa y de las pruebas realizadas anteriormente demuestran que el clúster MPI funciona correctamente, según la documentación (archivo léeme de MPICH2).

A continuación muestro un extracto del archivo "léeme" de MPICH2:

```
MPICH2 Release 1.2.1p1

MPICH2 is a high-performance and widely portable implementation of the
MPI-2.2 standard from the Argonne National Laboratory. This release has
all MPI 2.2 functions and features required by the standard with the
exception of support for the "external32" portable I/O format and user-
defined data representations for I/O.

The distribution has been tested by us on a variety of machines in our
environments as well as our partner institutes. If you have problems with
the installation or usage of MPICH2, please send an email to mpich-
discuss@mcs.anl.gov (you need to subscribe to this list
```

(https://lists.mcs.anl.gov/mailman/listinfo/mpich-discuss) before sending an email). If you have found a bug in MPICH2, we request that you report it at our bug tracking system:
(https://trac.mcs.anl.gov/projects/mpich2/newticket).

This README file should contain enough information to get you started with MPICH2. More extensive installation and user guides can be found in the doc/installguide/install.pdf and doc/userguide/user.pdf files respectively. Additional information regarding the contents of the release can be found in the CHANGES file in the top-level directory, and in the RELEASE_NOTES file, where certain restrictions are detailed. Finally, the MPICH2 web site,http://www.mcs.anl.gov/research/projects/mpich2, contains information on bug fixes and new releases.

…

---

7. Add the bin subdirectory of the installation directory to your path:

   for csh and tcsh:

```
setenv PATH /home/you/mpich2-install/bin:$PATH
```

   for bash and sh:

```
PATH=/home/you/mpich2-install/bin:$PATH ; export PATH
```

   Check that everything is in order at this point by doing

```
which mpd
which mpiexec
which mpirun
```

All should refer to the commands in the bin subdirectory of your install directory. It is at this point that you will need to duplicate this directory on your other machines if it is not in a shared file system such as NFS.

8. MPICH2 uses an external process manager for scalable startup of large MPI jobs. The default process manager is called MPD, which is a ring of daemons on the machines where you will run your MPI programs. In the next few steps, you will get his ring up and ested. More details on interacting with MPD can be found in the README file in mpich2-1.2.1p1/src/pm/mpd, such as how to list running jobs, kill, suspend, or otherwise signal them, and how to debug programs with "mpiexec -gdb".

If you have problems getting the MPD ring established, see the Installation Guide for instructions on how to diagnose problems with your system configuration that may be preventing it. Also see that guide if you plan to run MPD as root on behalf of users. Please be aware that we do not recommend running MPD as root until you have done testing to make sure that all is well.

Begin by placing in your home directory a file named .mpd.conf (/etc/mpd.conf if root), containing the line

```
secretword=<secretword>
```

where <secretword> is a string known only to yourself. It should NOT be your normal Unix password. Make this file readable and writable only by you:

```
chmod 600 .mpd.conf
```

9. The first sanity check consists of bringing up a ring of one mpd on the local machine, testing one mpd command, and bringing the "ring" down.

```
mpd &
mpdtrace
mpdallexit
```

The output of mpdtrace should be the hostname of the machine you are running on. The mpdallexit causes the mpd daemon to exit. If you have problems getting the mpd ring established, see the Installation Guide for instructions on how to diagnose problems with your system configuration that may be preventing it.

10. Now we will bring up a ring of mpd's on a set of machines. Create a file consisting of a list of machine names, one per line. Name this file mpd.hosts. These hostnames will be used as targets for ssh or rsh, so include full domain names if necessary. Check that you can reach these machines with ssh or rsh without entering a password.

You can test by doing

```
ssh othermachine date
```

or

```
rsh othermachine date
```

If you cannot get this to work without entering a password, you will need to configure ssh or rsh so that this can be done, or else use the workaround for mpdboot in the next step.

11. Start the daemons on (some of) the hosts in the file mpd.hosts

```
mpdboot -n <number to start>
```

The number to start can be less than 1 + number of hosts in the file, but cannot be greater than 1 + the number of hosts in the file. One mpd is always started on the machine where mpdboot is run, and is counted in the number to start, whether or not it occurs in the file.

There is a workaround if you cannot get mpdboot to work because of difficulties with ssh or rsh setup. You can start the daemons "by hand" as follows:

```
mpd &                # starts the local daemon
mpdtrace -l          # makes the local daemon print its host
                     # and port in the form <host>_<port>
```

Then log into each of the other machines, put the install/bin
directory in your path, and do:

```
mpd -h <hostname> -p <port> &
```

where the hostname and port belong to the original mpd that you
started. From each machine, after starting the mpd, you can do

```
mpdtrace
```

to see which machines are in the ring so far. More details on
mpdboot and other options for starting the mpd's are in
mpich2-1.2.1p1/src/pm/mpd/README.

!! ***************************
If you are still having problems getting the mpd ring established,
you can use the mpdcheck utility as described in the Installation Guide
to diagnose problems with your system configuration.
!! ***************************

12. Test the ring you have just created:

```
mpdtrace
```

The output should consist of the hosts where MPD daemons are now running.
You can see how long it takes a message to circle this ring with

```
mpdringtest
```

That was quick. You can see how long it takes a message to go
around many times by giving mpdringtest an argument:

```
mpdringtest 100
mpdringtest 1000
```

13. Test that the ring can run a multiprocess job:

```
mpiexec -n <number> hostname
```

The number of processes need not match the number of hosts in the ring;
if there are more, they will wrap around. You can see the effect of this
by getting rank labels on the stdout:

```
mpiexec -l -n 30 hostname
```

You probably didn't have to give the full pathname of the hostname
command because it is in your path. If not, use the full pathname:

```
mpiexec -l -n 30 /bin/hostname
```

14. Now we will run an MPI job, using the mpiexec command as specified in
the MPI-2 standard. There are some examples in the install directory,
which you have already put in your path, as well as in the directory

mpich2-1.2.1p1/examples. One of them is the classic cpi example, which computes the value of pi by numerical
integration in parallel.

```
mpiexec -n 5 cpi
```

The number of processes need not match the number of hosts.
The cpi example will tell you which hosts it is running on.
By default, the processes are launched one after the other on the hosts in the mpd ring, so it is not necessary to specify hosts when running a job with mpiexec.

There are many options for mpiexec, by which multiple executables can be run, hosts can be specified (as long as they are in the mpd ring), separate command-line arguments and environment variables can be passed to different processes, and working directories and search paths for executables can be specified. Do

```
mpiexec --help
```

for details. A typical example is:

```
mpiexec -n 1 master : -n 19 slave
```

or

```
mpiexec -n 1 -host mymachine : -n 19 slave
```

to ensure that the process with rank 0 runs on your workstation.

The arguments between ':'s in this syntax are called "argument sets", since they apply to a set of processes. Some arguments, called "global", apply across all argument sets and must appear first. For example, to get rank labels on standard output, use

```
mpiexec -l -n 3 cpi
```

See the User's Guide for much more detail on arguments to mpiexec.

The mpirun command from the original MPICH is still available, although it does not support as many options as mpiexec.

If you have completed all of the above steps, you have successfully installed MPICH2 and run an MPI example.

More details on arguments to mpiexec are given in the User's Guide in the doc subdirectory. Also in the User's Guide you will find help on debugging. MPICH2 has some some support for the TotalView debugger, as well as some other approaches described there.

Apagando nodos mientras ejecutan procesos MPI

El programa `prueba.c` puede utilizarse para comprobar que sucede al apagar máquinas cuando están ejecutando algún proceso MPI:

```
prueba.c

#include <stdio.h>
#include "mpi.h"

int main( int argc, char *argv[] ){
    int numprocs,myid;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    printf("\nHola mundo desde el proceso: %d de %d\n",myid,numprocs);
    sleep(50);
    printf("el proceso: %d finaliza la ejecucion\n",myid);
    MPI_Finalize();
    return 0;
}
```

La siguiente imagen muestra la ejecución normal del programa (sin apagar ningún nodo) ejecutando tres copias del proceso `prueba` en paralelo (ejecutando un proceso en cada ordenador) y también se muestra como se vuelve a hacer el mismo paso pero apagando el ordenador3 (botón "apagar sistema") antes de que ninguno de los 3 procesos finalicen la ejecución:

```
josemanuel@ordenador1:~/jm/mpich2$ mpdtrace
ordenador1
ordenador2
ordenador3
josemanuel@ordenador1:~/jm/mpich2$ mpiexec -np 3  prueba

Hola mundo desde el proceso: 0 de 3

Hola mundo desde el proceso: 1 de 3

Hola mundo desde el proceso: 2 de 3
el proceso: 0 finaliza la ejecucion
el proceso: 2 finaliza la ejecucion
el proceso: 1 finaliza la ejecucion
josemanuel@ordenador1:~/jm/mpich2$ mpiexec -np 3  prueba

Hola mundo desde el proceso: 0 de 3

Hola mundo desde el proceso: 1 de 3

Hola mundo desde el proceso: 2 de 3
el proceso: 0 finaliza la ejecucion
```

Se puede apreciar que en la primera ejecución de los 3 procesos en paralelo, estos finalizan de forma normal pero en el segundo caso (apagando el ordenador3) solo finaliza la ejecución uno de los procesos y la consola se queda bloqueada.

Comparativa PVM / MPI y conclusiones:

MPI es un standard definido por un consorcio de organizaciones. En estos textos se ha utilizado MPICH2 (que es la implementación más actual de MPI para Linux Debian) .

MPICH2 es una implementación del estándar MPI 2.

Durante la elaboración de este estudio ha salido el estándar MPI 3.0 (en Diciembre de 2012).

El superordenador `Mare Nostrum` del Centro de Súper computación de Barcelona y todos los superordenadores de la Red Española de Súper Computación tienen instalado `MPICH2` para que los usuarios programadores puedan hacer uso de MPI en sus programas y así poder aprovechar la potencia de cálculo de los supercomputadores.

PVM existe, funciona y se puede utilizar para crear software paralelo / distribuido pero MPI es una tecnología más actualizada y que continua en evolución. MPI recoge aportaciones de PVM.

-En MPI para iniciar la infraestructura de paso de mensajes entre procesos: Hay que invocar la función `MPI_INIT`, en cambio en PVM es implícito: La infraestructura se inicializa al invocar por primera vez cualquier función PVM.

-MPI tiene funciones de procesado de datos en grupo (de procesos) como es la función `MPI_Reduce` utilzada en el programa `cpi.c` explicado en páginas anteriores. En `cpi.c` se utiliza para calcular la suma de valores que cada proceso tiene y el resultado lo acaba obteniendo un proceso raíz. En este caso era la suma pero también podía haber sido obtener el máximo de los valores o cualquiera de las otras funciones definidas o también podría haber sido una función definida por el programador.

-Se puede apreciar que MPI es ligeramente más compacto, requiriendo menos código que PVM:

En general PVM es un poco más laborioso de utilizar que MPI, requiriendo en ocasiones más pasos para lograr el mismo resultado. Por ejemplo: La realización de un *broadcast* en MPI requiere únicamente una llamada a función, mientras que en PVM se requieren dos (una para empaquetar la información y otra para realizar el envío). En MPI se requieren un menor número de llamadas a funciones.

-El proyecto PVM es antiguo y está parado desde hace años.

-MPI recoge aportaciones de PVM y es tecnología actual.

-MPI se utiliza para aprovechar la potencia de cálculo de los superordenadores actuales como por ejemplo: El Mare Nostrum III.

Teniendo en cuenta lo explicado anteriormente:

MPI debería ser la elección para la programación de proyectos nuevos.

MOSIX
Multicomputer Operating System for UnIX

MOSIX es un paquete software que transforma un conjunto de máquinas conectadas por red bajo GNU/Linux en un *clúster*. Éste equilibra la carga automáticamente entre los diferentes nodos del *clúster* y los nodos pueden unirse o dejar el *clúster* sin interrumpir el servicio. La carga se distribuye entre los nodos teniendo en cuenta la velocidad de la conexión y la CPU. MOSIX forma parte del *kernel* (a través de un Linux Kernel Patch) y mantiene total compatibilidad con GNU/Linux, los programas de usuario, archivos y recursos.

MOSIX está dirigido a la computación de alto rendimiento.La principal utilidad de MOSIX es la migración de procesos. Un proceso que se arranca en un nodo puede migrar y ejecutarse en otros nodos y volver a migrar si es necesario incluso volver al nodo que lo inició, todo dependiendo de las condiciones.

Las migraciones de los procesos ocurren de forma automática y transparente en respuesta a la disponibilidad de los recursos.

La migración de procesos se utiliza para optimizar el rendimiento global. Por ejemplo, si un proceso de cálculo intensivo es lanzado en una maquina 386 del clúster, este proceso podrá migrar a la mejor maquina disponible en el clúster, pudiendo ejecutarse, por ejemplo, en una maquina Core i5. Además cada nodo del clúster puede utilizarse de forma "normal" sin recurrir a las funcionalidades de MOSIX.

Las características de MOSIX están destinadas a proporcionar a los usuarios y a las aplicaciones la impresión de que se ejecutan las tareas en un único equipo con varios procesadores, sin cambiar la interfaz ni el entorno de ejecución de sus respectivos nodos de inicio de sesión.

MOSIX reconoce dos tipos de procesos: Procesos Linux y procesos MOSIX (arrancados mediante `mosrun`). Los procesos Linux no se ven afectados por MOSIX, estos se ejecutan en modo Linux nativo y no pueden migrar. Los procesos MOSIX pueden migrar.

Los procesos hijos de los procesos MOSIX permanecen bajo la disciplina MOSIX (con la excepción de la utilidad "native utility" que permite a los programas y Shells invocados mediante `mosrun` arrancar procesos hijos en modo Linux nativo).

Si una Shell es arrancada mediante el comando `mosrun`, todos los procesos hijos de la Shell también serán migrables.

El proceso de migración puede ser activado de forma automática o manualmente. La migración de procesos se consigue haciendo una copia de la imagen de memoria del proceso y estableciendo su entorno de ejecución. Para reducir el uso de la red, la imagen de memoria se comprime normalmente utilizando lzop [9].

Las migraciones automáticas están supervisadas por un algoritmo *on-line* que continuamente intenta mejorar el rendimiento global, ej: Por balaceo de carga, por migración de procesos que necesitan más memoria de la disponible (suponiendo que hay otro nodo con suficiente memoria libre) o por migración de procesos de nodos lentos a rápidos.

Estos algoritmos son especialmente útiles para aplicaciones con requerimiento de recursos imprevisibles o cambiantes y cuando hay varios usuarios ejecutando tareas simultáneamente.

Las decisiones de migración automática están basadas en perfiles de proceso (en tiempo de ejecución) y en la última información de disponibilidad de recursos.

El perfilado de procesos se realiza recolectando información continuamente sobre las características de los procesos, ej: Tamaño, cantidad de llamadas al sistema, volumen de entrada / salida. Esta información es utilizada por algoritmos *on-line* competitivos para determinar la mejor ubicación para cada proceso.

Estos algoritmos tienen en cuenta las velocidades y las cargas actuales de los nodos, el tamaño del proceso migrado, la memoria libre disponible en los diferentes nodos y las características de los procesos. De esta manera, cuando el perfil de un proceso cambia o cuando nuevos recursos pasan a estar disponibles, el algoritmo responde automáticamente considerando la reubicación de los procesos para conseguir un mayor rendimiento global

El *launching-node*: Es el nodo desde donde se arranca el proceso invocando `mosrun programa`.
El *home-node*: Es el nodo al cual pertenece el proceso.

Normalmente el *launching node* y el *home node* es el mismo nodo, al no ser que se invoque `mosrun` con el modificador `-M` en este caso pueden ser diferentes.

Las entradas / salidas hechas por los processos migrados en nodos remotos se hacen mediante sus respectivos *home-nodes*. Si el uso de las entradas/salidas es significativo esto provoca que el proceso sea migrado de vuelta a su *home-node*.

El entorno en tiempo de ejecución:

MOSIX está implementado como una capa de software que permite que las aplicaciones se ejecuten en nodos remotos, fuera de sus respectivos *home-nodes* (nodo al cual pertenece el proceso). Esto se logra mediante la intercepción de todas llamadas al sistema, entonces si el proceso ha migrado, la mayoría de sus llamadas al sistema se envían a su *home-node* en el cual, se llevan a cabo en nombre del proceso, las llamadas al sistema como si se estuviera ejecutando en el *home-node* y los resultados de las llamadas al sistema se envían de nuevo al proceso.

En MOSIX, las aplicaciones se ejecutan en un entorno donde los procesos migrados parecen estar ejecutándose en su *home-node*.
Como resultado, los usuarios no necesitan saber dónde están sus programas ejecutándose, no es necesario modificar las aplicaciones, ni enlazarlas a ninguna biblioteca, ni logarse o copiar archivos en los nodos remotos.

Además, la consistencia de los archivos y de los datos, así como los mecanismos tales como señales, semáforos y los identificadores de los procesos, quedan intactos.

Cuando un nodo deja de funcionar (por ejemplo, porque se interrumpe la alimentación) todos los procesos que se estaban ejecutando en este nodo o fueron lanzados desde este nodo se eliminan.

Los puertos utilizados por MOSIX:

TCP ports 249 - 253.

UDP ports 249 - 250.

En un clúster MOSIX: Se puede arrancar los programas desde cualquier nodo del clúster pero normalmente hay un nodo (el nodo de acceso) en el cual los usuarios se logan y arrancan los programas.

Instalación

En estos textos se ha utilizado la última versión, MOSIX 3.3.0.0 que ha sido diseñado para utilizarse con el Kernel de Linux 3.7.0.

Hay que descargar MOSIX desde la página oficial www.mosix.org y descomprimir los archivos.

El script `mosix.install` guia paso a paso y automatiza el proceso de instalación y configuración de MOSIX .

El proceso de instalación y configuración es el siguiente:

Crear los directorios `/etc/mosix` y `/etc/mosix/var`

Para la instalación de MOSIX es necesario copiar los binarios en sus directorios correspondientes, dar los permisos adecuados y crear los enlaces simbólicos necesarios:

```
man/*             /usr/local/man
mos_checkconf    /sbin
mosbestnode      /bin            chmod u+s /bin/mosbestnode
mosconf*         /sbin
mosctl           /bin
mosd             /sbin
mosenv           /bin            ln -s mosenv /bin/mosbatchenv
mosixd           /sbin
moskillall       /bin
mosmigrate       /bin
mosmon           /bin
mosnative        /bin
mospipe          /bin
mospostald       /sbin
mosps            /bin            chmod u+s /bin/mosps
mosq             /bin            chmod u+s /bin/mosq
mosqmd           /sbin
mosqueue         /bin
mosrc            /bin            chmod u+s /bin/mosrc
mosrcd           /sbin
mosremoted       /sbin
mosrun           /bin            chmod u+s /bin/mosrun
                                 ln -s mosrun /bin/mosbatch
mossetcl         /sbin
mossetpe         /sbin
mostestload      /bin
mostimeof        /bin            chmod u+s /bin/mostimeof
other/patch-3.7 parche del kernel Linux-3.7  ("patch -p1")
```

```
Hay que parchear el kernel 3.7 con el parche de Mosix 3.3.0.0:


1.Obtener una copia del codigo fuente del Kernel de Linux 3.7 de
   http://www.kernel.org/pub/linux/kernel/v3.0/linux-3.7.tar.bz2
2. Desempaquetar el codigo fuente del nucleo:
     tar xjf linux-3.7.tar.bz2
3. Entrar en el directorio de las funtes del nucleo:
     cd linux-3.7
```

4. Aplicar el parche de Mosix al kernel:
```
patch -p1 < ../other/patch-3.7
```
5. Configurar el kernel:
```
make menuconfig
```

Configurar las siguientes opciones:

```
 CONFIG_FUSE_FS
File systems  --->  FUSE (Filesystem in Userspace) support
 CONFIG_SYSVIPC
General setup  --->  System V IPC
```

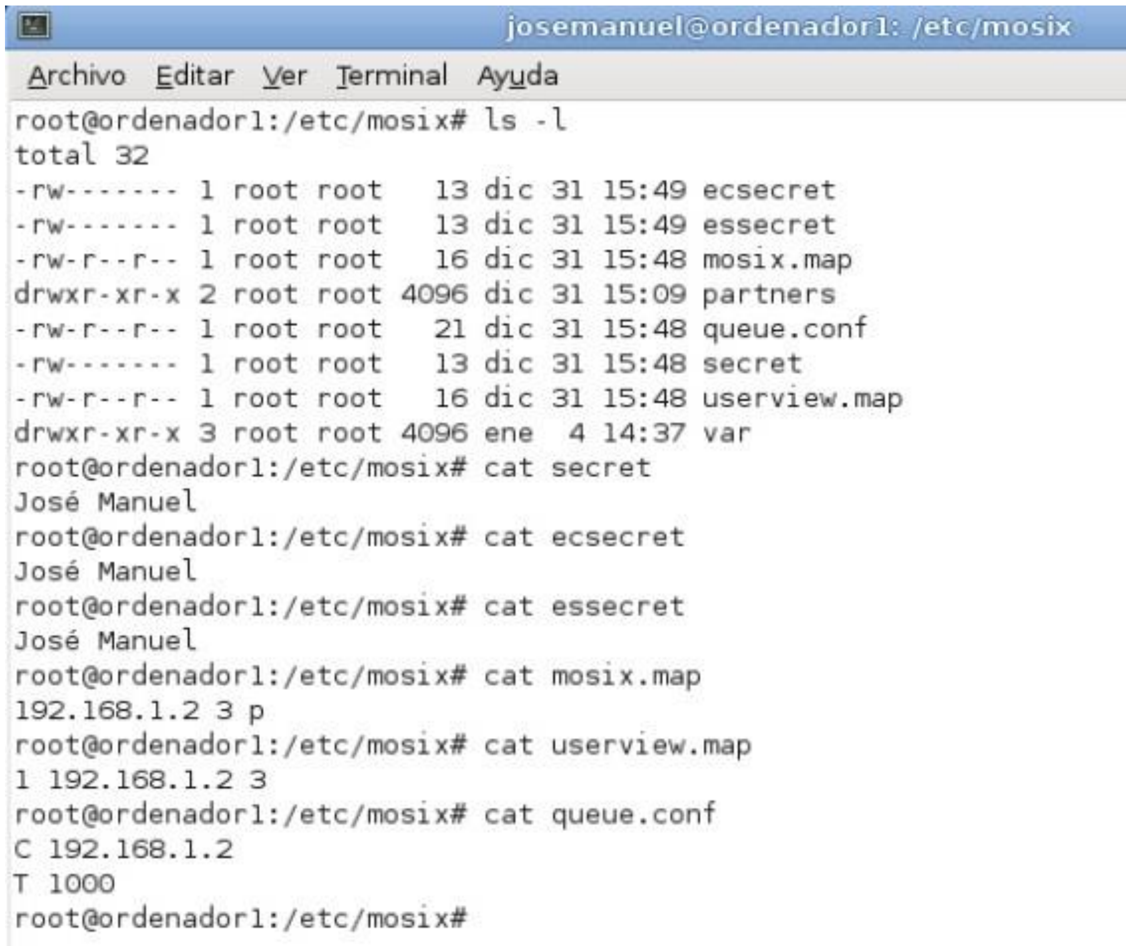
Pero hay que asegurarse de que lo siguiente no está configurado:

```
CONFIG_HEADERS_CHECK"
   'Kernel hacking' ---> 'Run 'make headers_check' when building vmlinux'
```

6. Compilar el kernel:
```
make bzImage
make modules    (si estas utilizando modulos del kernel)
```
7. Instalar el kernel:
```
make modules_install
make install
```

Configuración:

El script `mosconf` guia paso a paso en el proceso de configuración. Este script es normalmente llamado por `mosix.install` y lo que hace es editar los archivos de configuración:



La parte más importante de la configuración, es la de indicar, que nodos pertenecen al clúster.

Con los archivos editados de esta manera tenemos el clúster configurado con 3 nodos que son: 192.168.1.2, 192.168.1.3 y 192.168.1.4 esto está escrito en el archivo `/etc/mosix.map` con una notación reducida: Se indica que hay 3 nodos consecutivos y que el 192.168.1.2 es el primer nodo. Esta notación es equivalente a poner cada IP consecutiva de nodo una por línea (este fichero de configuración admite estas dos maneras de representar que nodos que pertenecen al clúster).

Es necesario editar los archivos: `secret, ecsecret` y `essecret` añadiendo una palabra secreta para hacer seguras las comunicaciones entre los nodos. En este caso he puesto las tres palabras iguales: `José Manuel` (recordemos que todos los pasos de la instalación y configuración deben hacerse igual en todos los nodos)

Opcionalmente se pueden utilizar números de nodo en vez de sus IPs , para indicar que número tiene asociado cada nodo hay que editar el archivo `/etc/userview.map` , en este caso , con el archivo editado de esta manera, el nodo 192.168.1.2 tiene asociado el número 1, el 192.168.1.3 el 2 y el 192.168.1.4 el 3, la notación utilizada es como en el caso del archivo `/etc/mosix.map`.

El archivo `/etc/queue.conf` se utiliza para definir cual es el nodo utilizado que hará de gestor de cola y sus políticas, en este caso está configurado para que el gestor de cola sea el primer nodo .
Para que el kernel 3.7.0 parcheado con Mosix y compilado pueda arrancarse, hay que situarse en el directorio `/boot` y ejecutar la siguiente instrucción:

```
mkinitramfs –v 3.7.0 –o intrd.img-3.7.0
```

Esta instrucción sirve para crear el archivo `initrd.img-3.7.0` necesario para el arranque del kernel 3.7.0

A continuación hay que editar el archivo de configuración del gestor de arranque, en nuestro caso es Grub:

`/boot/grub/grub.cfg`

```
### BEGIN /etc/grub.d/10_linux ###
menuentry 'Debian GNU/Linux, with Linux 2.6.32-5-amd64' --class debian --class gnu-linux --class gnu --class os {
        insmod part_msdos
        insmod ext2
        set root='(hd0,msdos1)'
        search --no-floppy --fs-uuid --set bc5df9eb-98f2-41bb-a0d8-2bb33e0c1108
        echo    'Loading Linux 2.6.32-5-amd64 ...'
        linux   /boot/vmlinuz-2.6.32-5-amd64 root=UUID=bc5df9eb-98f2-41bb-a0d8-2bb33e0c1108 ro  quiet
        echo    'Loading initial ramdisk ...'
        initrd  /boot/initrd.img-2.6.32-5-amd64
}
menuentry 'Debian GNU/Linux, with Linux 3.7.0 con el parche de Mosix' --class debian --class gnu-linux --class gnu --class os {
        insmod part_msdos
        insmod ext2
        set root='(hd0,msdos1)'
        search --no-floppy --fs-uuid --set bc5df9eb-98f2-41bb-a0d8-2bb33e0c1108
        echo    'Loading Linux 3.7.0 con el parche de Mosix'
        linux   /boot/vmlinuz-3.7.0 root=UUID=bc5df9eb-98f2-41bb-a0d8-2bb33e0c1108 ro quiet
        echo    'Loading initial ramdisk ...'
        initrd  /boot/initrd.img-3.7.0
}
### END /etc/grub.d/10_linux ###
```

Estas son las líneas que hay que editar para poner los valores correctos:

```
echo   'Loading Linux 3.7.0 con el parche de Mosix'
linux  /boot/vmlinuz-3.7.0 root=UUID=bc5df9eb-98f2-41bb-a0d8-2bb33e0c1108 ro quiet
echo   'Loading initial ramdisk ...'
initrd /boot/initrd.img-3.7.0
```

La linea `linux` indica donde está el kernel de Linux y en nuestro caso es: `/boot/vmlinuz-3.7.0`.

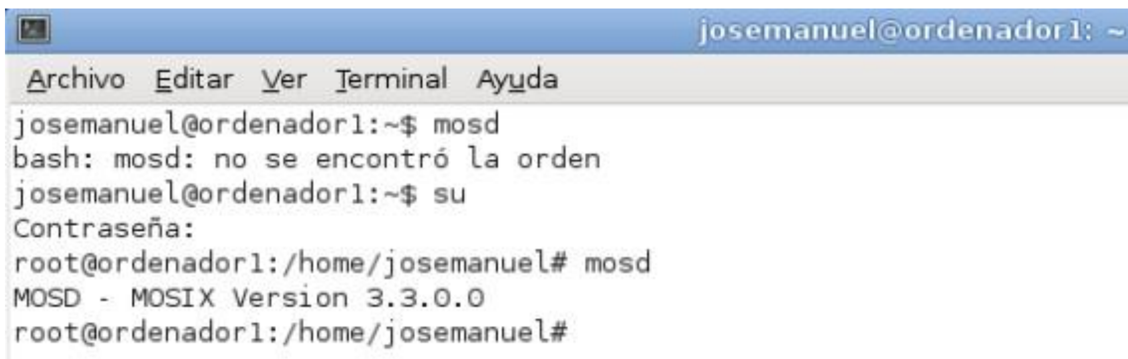
La línea `initrd` indica donde está el archivo que hemos generado en el paso anterior, este archivo contiene una imagen de memoria necesaria para el arranque del núcleo 3.7.0

La línea `menuentry` se puede modificar para poner el título que se desee, este título saldrá en el menú de inicio al arrancar el sistema.

Una vez realizados todos los pasos de instalación y configuración descritos, se debe reiniciar el sistema:

Al iniciar el ordenador podemos elegir arrancar con el nuevo kernel 3.7.0 MOSIX o con el que teníamos anteriormente.

Una vez iniciado el sistema operativo se puede arrancar el servicio MOSIX ejecutando la instrucción `mosd` con el usuario `root`.



El mensaje que imprime por pantalla `mosd` indica que MOSIX está arrancado y funcionando correctamente.

Todos los pasos del proceso de instalación y configuración hay que hacerlos igual en todos los nodos que forman el clúster (utilizando la misma versión de MOSIX y Kernel). Una vez hecho esto y hayamos iniciado el servicio MOSIX mediante la instrucción `mosd` en todos los nodos, ya tenemos el clúster preparado para ejecutar programas los cuales podrán migrar entre los nodos.

MOSIX en funcionamiento

A continuación se muestran pruebas de uso del clúster utilizando programas que simulan la realización de cálculos intensivos.

En la imagen hay tres consolas abiertas en el `ordenador1` con el usuario `josemanuel`:

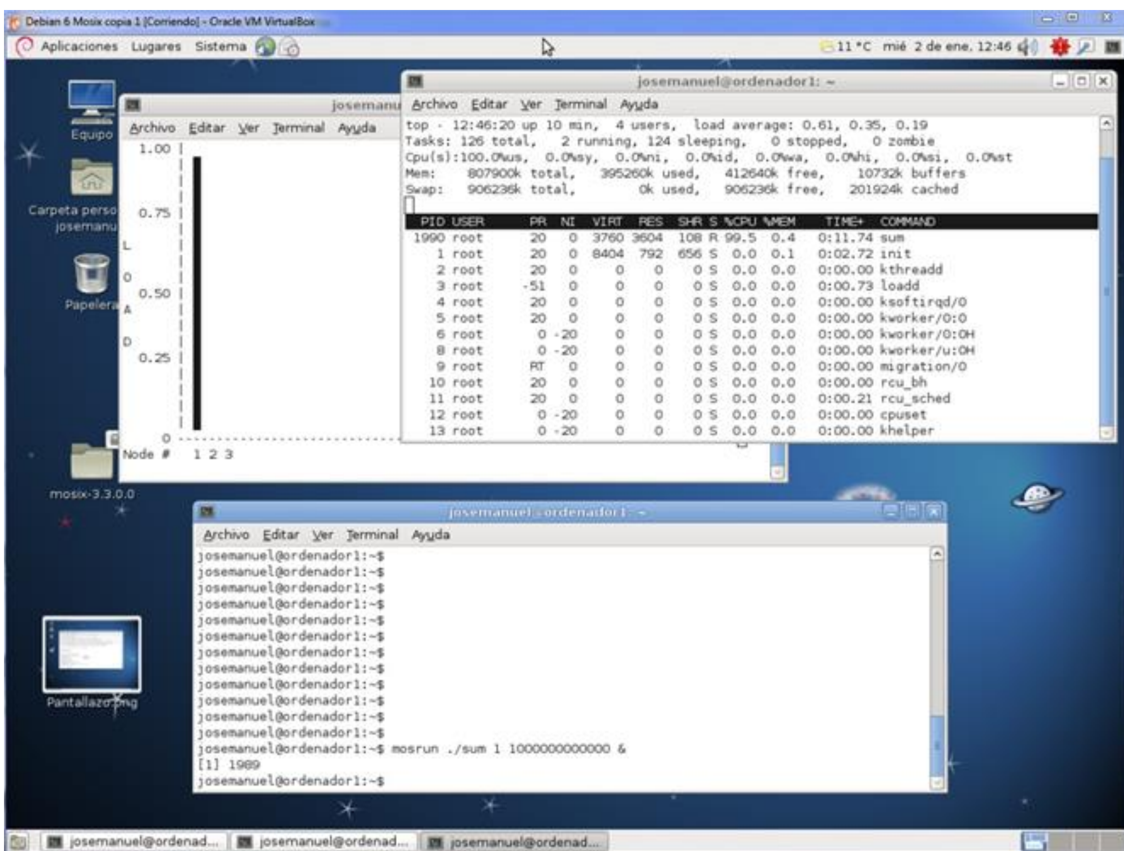
En una de las consolas, se ha ejecutado la instrucción `mosmon` que ilustra la carga de cada nodo del clúster en tiempo real.

Todas las instrucciones de MOSIX empiezan por “mos”.

En otra de las consolas se está ejecutando el comando de Linux `top`.

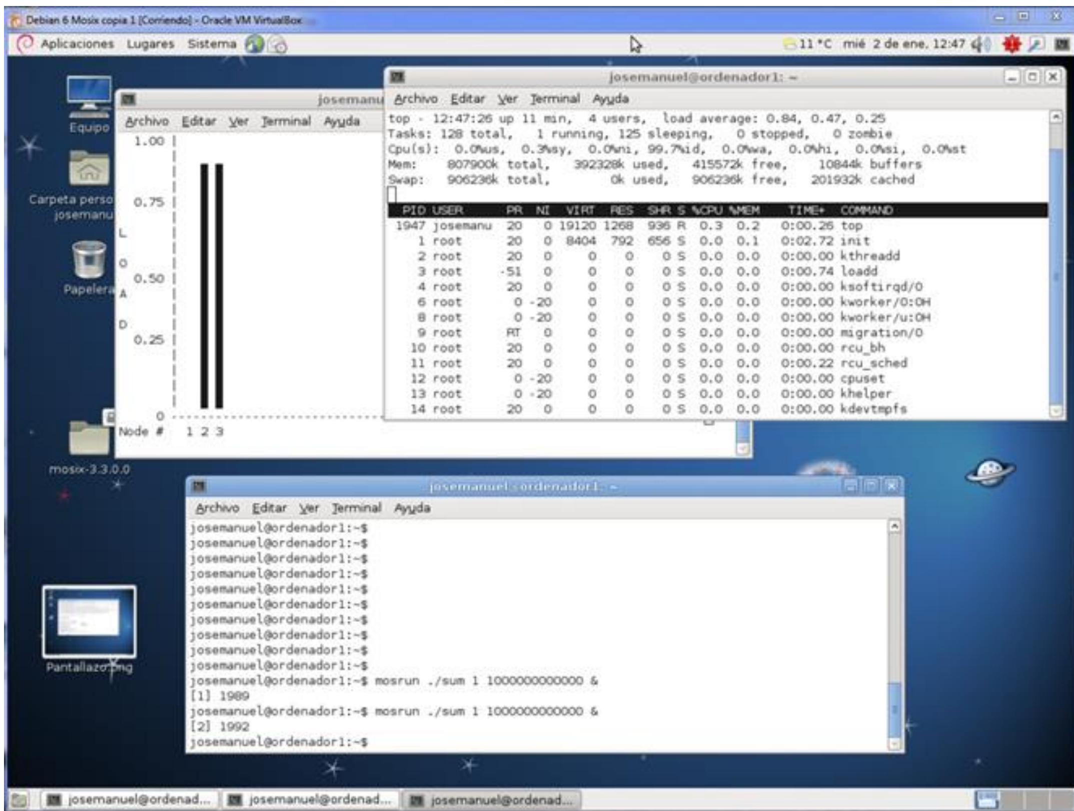
En la tercera consola se lanza el programa `sum` (utilizado en el apartado “beneficios del cómputo distribuido”). El programa `sum` se arranca mediante el comando `mosrun` para que pueda migrar entre los nodos del clúster si es necesario.

Al final de la instrucción `mosrun` está el carácter “&” el cual indica que la instrucción se ejecute en segundo plano:

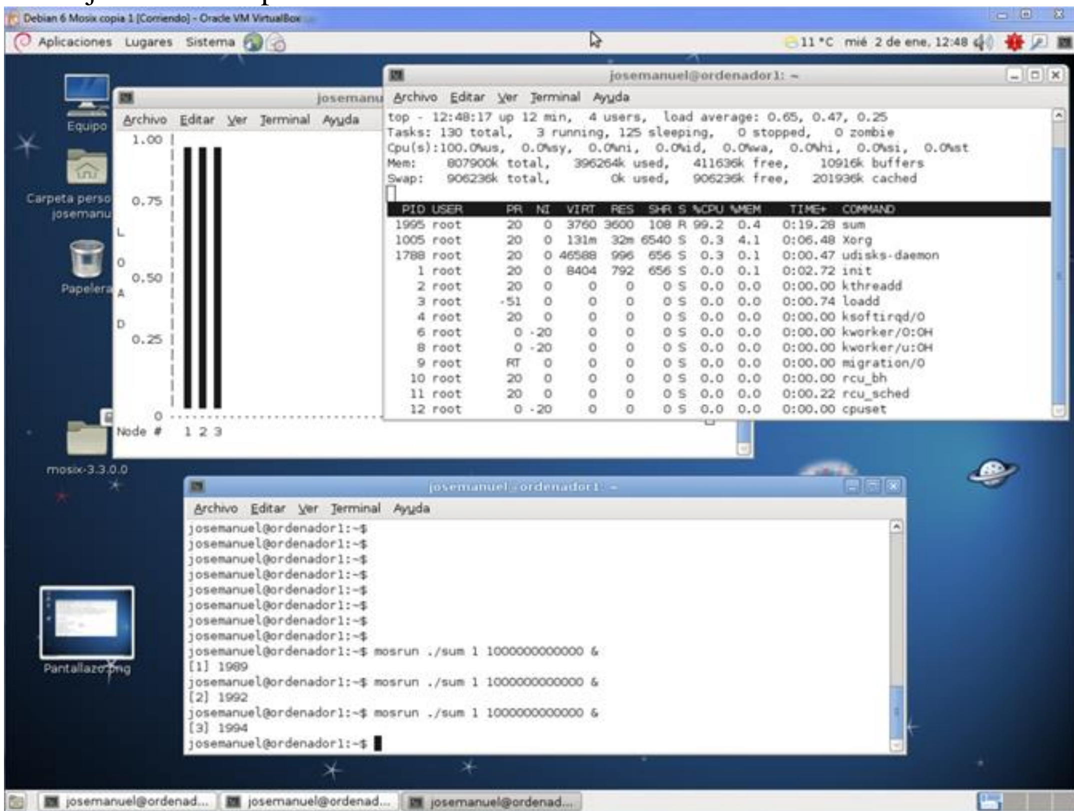


Se puede apreciar, gracias al comando `mosmon` , la carga de trabajo de los nodos. En este caso, el nodo 1 es el que está ejecutando el programa `sum`, se está ejecutando en el mismo nodo que ha lanzado la tarea.

Como la instrucción anterior se ejecuta en segundo plano, después se puede invocar la ejecución de otro proceso sum igual que en el paso anterior:



Si se lanza un segundo proceso sum con el comando `mosrun`, se puede apreciar que los dos procesos `sum` migran: Uno del ordenador1 al ordenador2 y el otro del ordenador1 al ordenador3. Si se ejecuta un 3er proceso `sum`:



Después de ejecutar el 3er proceso sum con el comando `mosrun`, hay 3 procesos sum repartidos entre los 3 nodos del clúster, cada proceso se ejecuta en un nodo. La carga está equilibrada entre los nodos.

Ejecución del comando top en el ordenador2:

```
josemanuel@ordenador2: ~
Archivo  Editar  Ver  Terminal  Ayuda
top - 12:50:00 up 11 min,  2 users,  load average: 0.94, 0.48, 0.25
Tasks: 122 total,   2 running, 120 sleeping,   0 stopped,   0 zombie
Cpu(s):  1.3%us,  0.7%sy, 98.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:    807900k total,   383880k used,   424020k free,    10820k buffers
Swap:   906236k total,        0k used,   906236k free,   199224k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2221 root      39  19  3756 1608    0 R 95.8  0.2   2:41.07 mosremoted
 1254 root      20   0  131m  32m 6304 S  2.3  4.1   0:02.44 Xorg
 2076 josemanu  20   0  160m  11m 8900 S  0.7  1.4   0:00.15 metacity
 2089 josemanu  20   0  356m  19m  14m S  0.3  2.5   0:00.72 nautilus
 2149 josemanu  20   0  223m  13m   9m S  0.3  1.7   0:00.29 gnome-terminal
    1 root      20   0  8404  788  656 S  0.0  0.1   0:02.54 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root     -51   0     0    0    0 S  0.0  0.0   0:00.39 loadd
    4 root      20   0     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/0
    6 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 kworker/0:0H
    8 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 kworker/u:0H
    9 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
   10 root      20   0     0    0    0 S  0.0  0.0   0:00.00 rcu_bh
   11 root      20   0     0    0    0 S  0.0  0.0   0:00.22 rcu_sched
   12 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 cpuset
   13 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 khelper
```

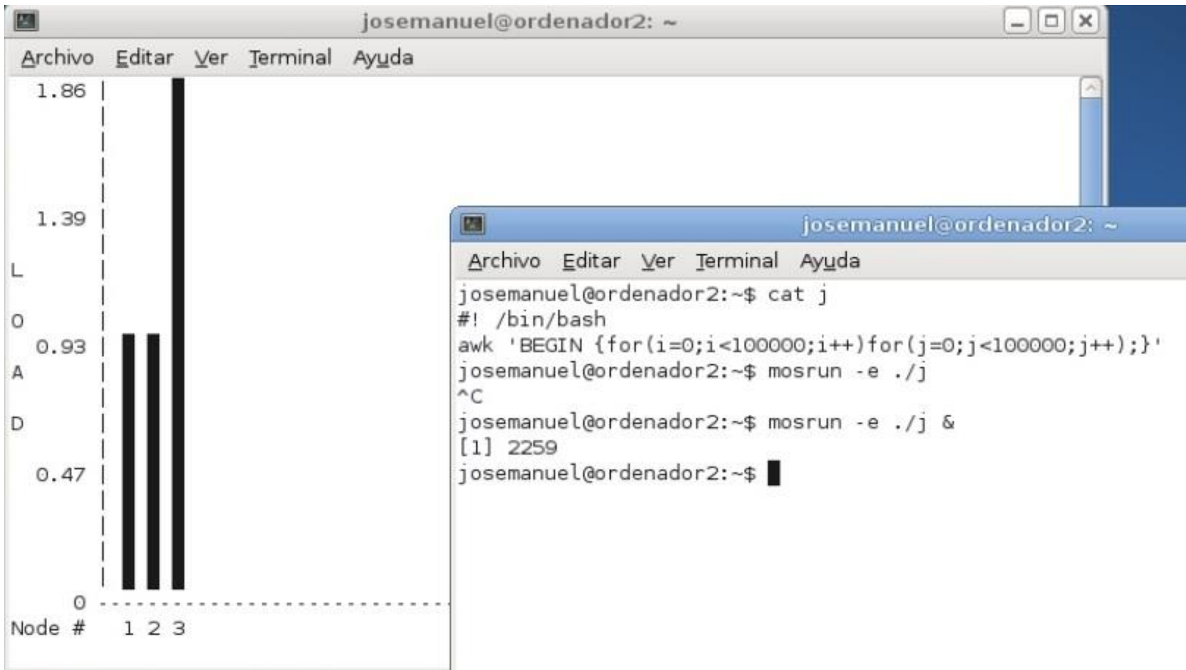
Ejecución del comando `top` en el ordenador3:

```
josemanuel@ordenador3: ~
Archivo  Editar  Ver  Terminal  Ayuda
top - 12:51:07 up 10 min,  2 users,  load average: 0.98, 0.59, 0.28
Tasks: 123 total,   2 running, 121 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  0.2%sy, 74.8%ni, 24.9%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:    807900k total,   384544k used,   423356k free,    10700k buffers
Swap:   906236k total,        0k used,   906236k free,   200076k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1917 root      39  19  3756 1608    0 R 99.4  0.2   3:47.27 mosremoted
  985 root      20   0  130m  31m 6092 S  0.3  4.0   0:02.38 Xorg
    1 root      20   0  8404  788  656 S  0.0  0.1   0:02.72 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root     -51   0     0    0    0 S  0.0  0.0   0:00.74 loadd
    4 root      20   0     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/0
    6 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 kworker/0:0H
    8 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 kworker/u:0H
    9 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
   10 root      20   0     0    0    0 S  0.0  0.0   0:00.00 rcu_bh
   11 root      20   0     0    0    0 S  0.0  0.0   0:00.20 rcu_sched
   12 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 cpuset
   13 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 khelper
   14 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kdevtmpfs
   15 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 netns
   16 root      20   0     0    0    0 S  0.0  0.0   0:00.00 bdi-default
   17 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 kintegrityd
```

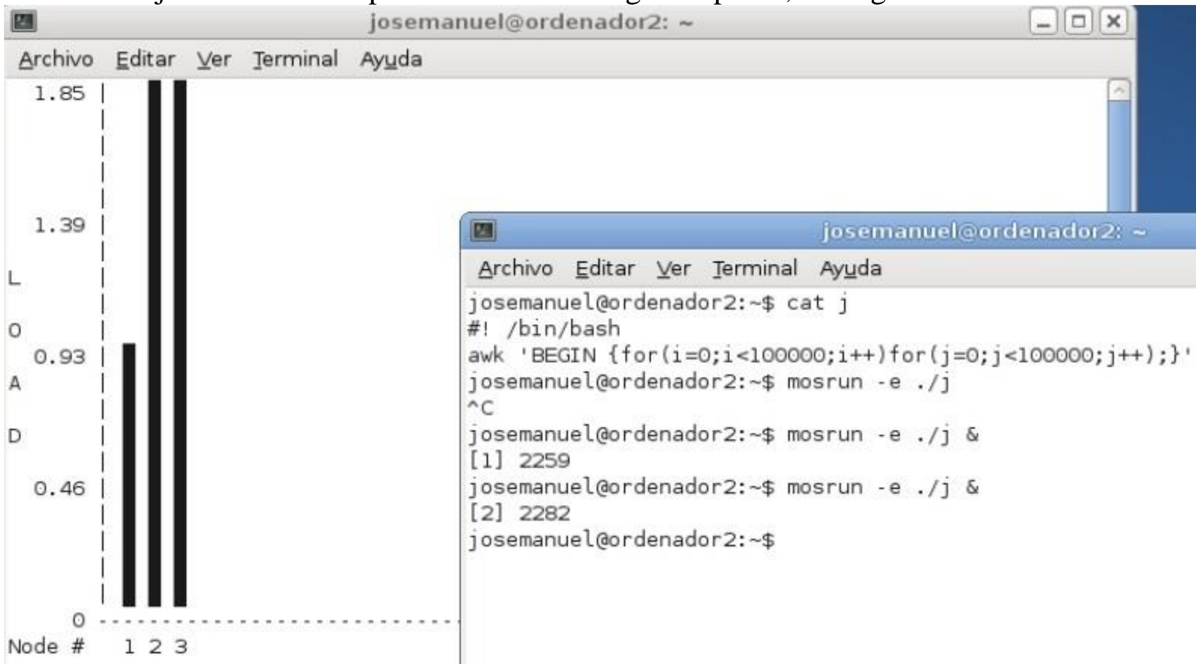
La ejecución de las tareas migradas las ejecuta el proceso `mosremoted` en el nodo local.

Siguiendo los pasos anteriores se tienen los 3 procesos `sum`, cada uno ejecutándose en un nodo.

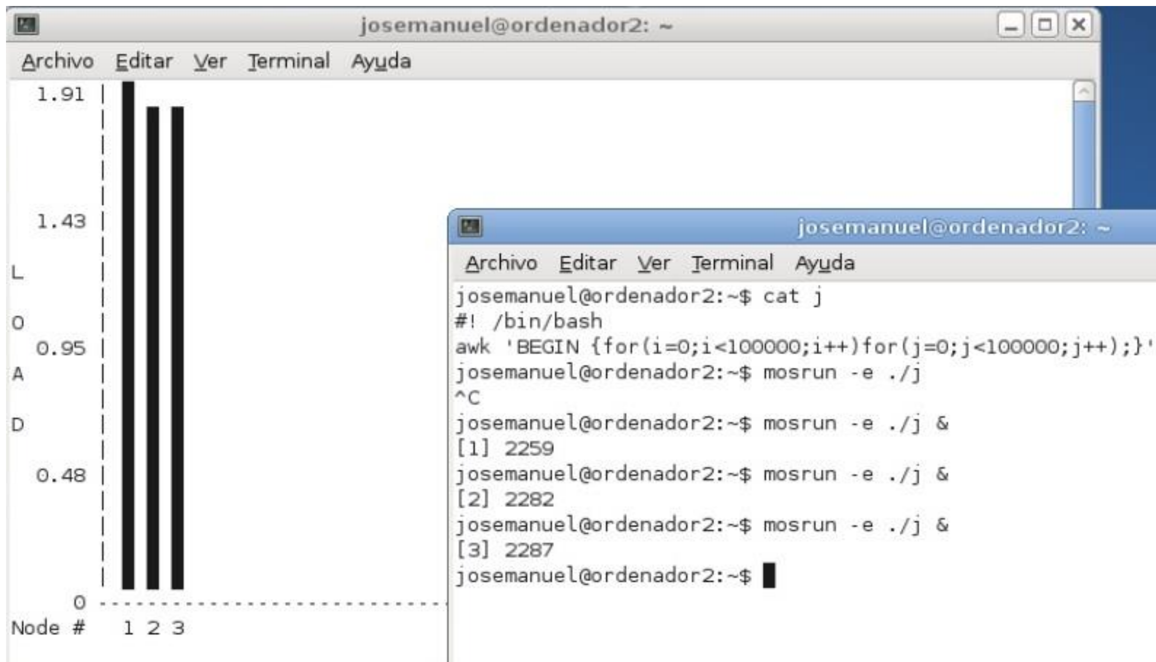
Si después se ejecuta en el ordenador2 el comando AWK que se muestra en la imagen, en segundo plano, la carga de cada nodo es la siguiente:



Si además ejecutamos un 2º proceso AWK en segundo plano, la carga de cada nodo es:

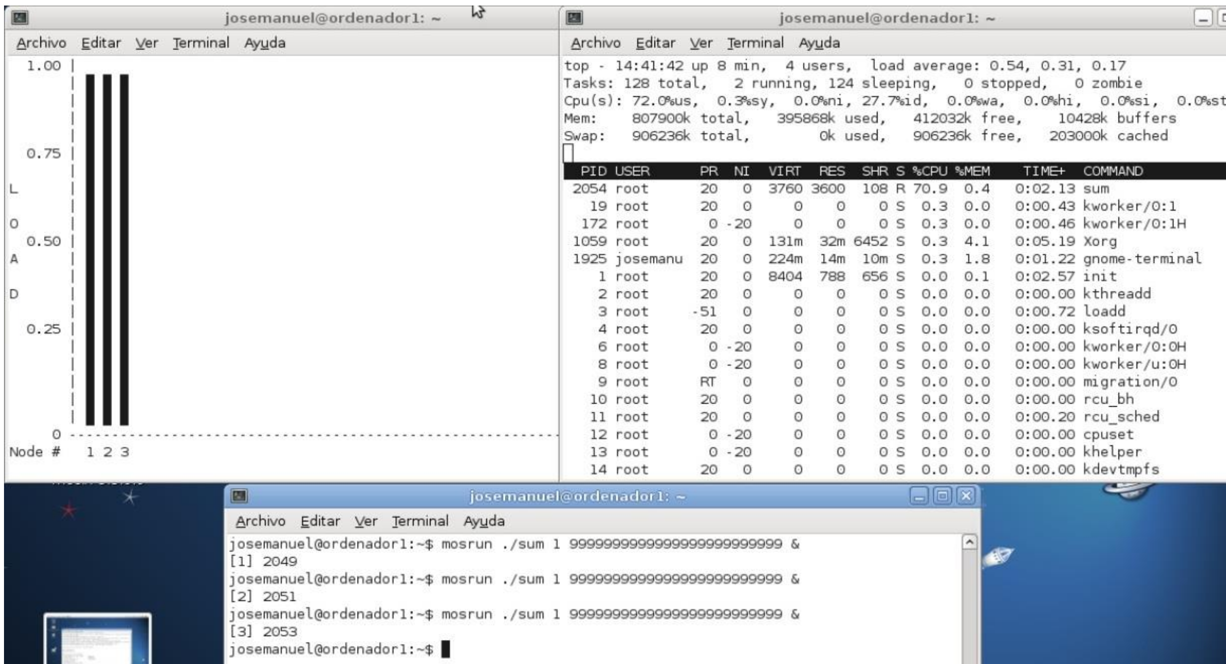


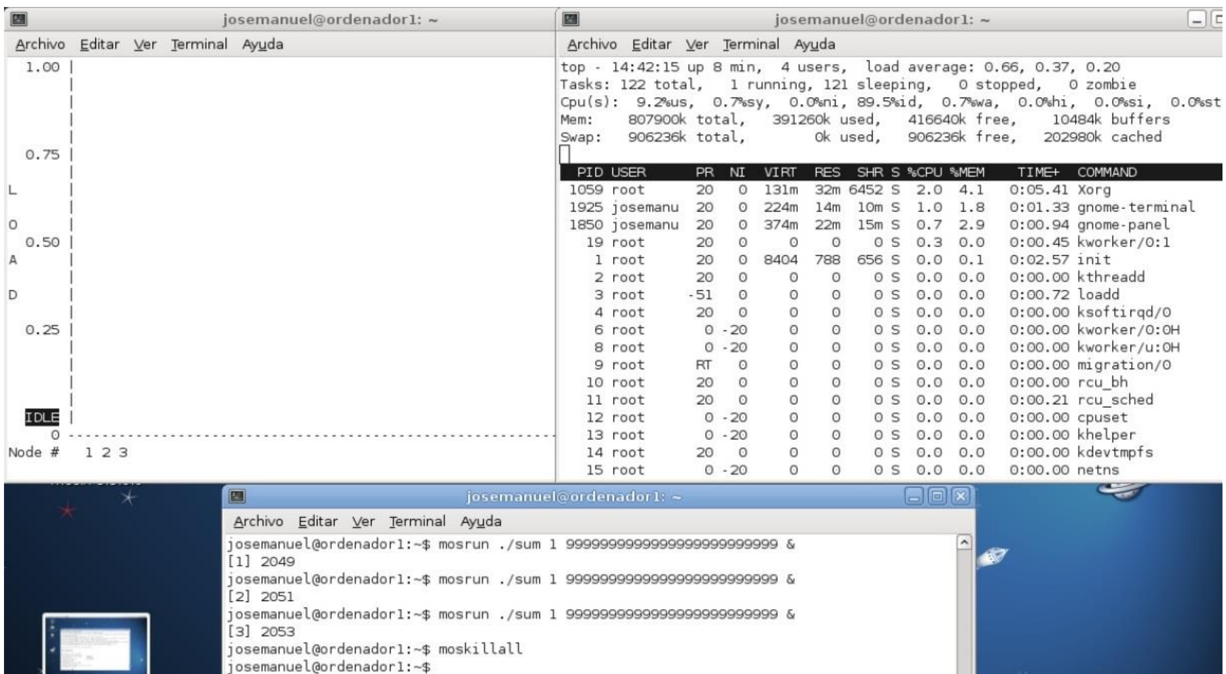
Finalmente si ejecutamos un 3er proceso AWK en segundo plano, la carga de cada nodo es:



Si se ejecutan con el comando `mosrun`, los 3 procesos `sum` (desde el ordenador1) y los 3 procesos AWK (desde el ordenador2) todos en segundo plano y simultáneamente, la carga se equilibra entre los 3 nodos de clúster automáticamente y de forma transparente para el usuario.

Con el comando `moskillall` se finaliza la ejecución de todos los procesos MOSIX lanzados desde el nodo donde se invoca la instrucción:





Pruebas de compresión de música en MOSIX

Compresión de las 15 canciones de un CD de música utilizando MOSIX:

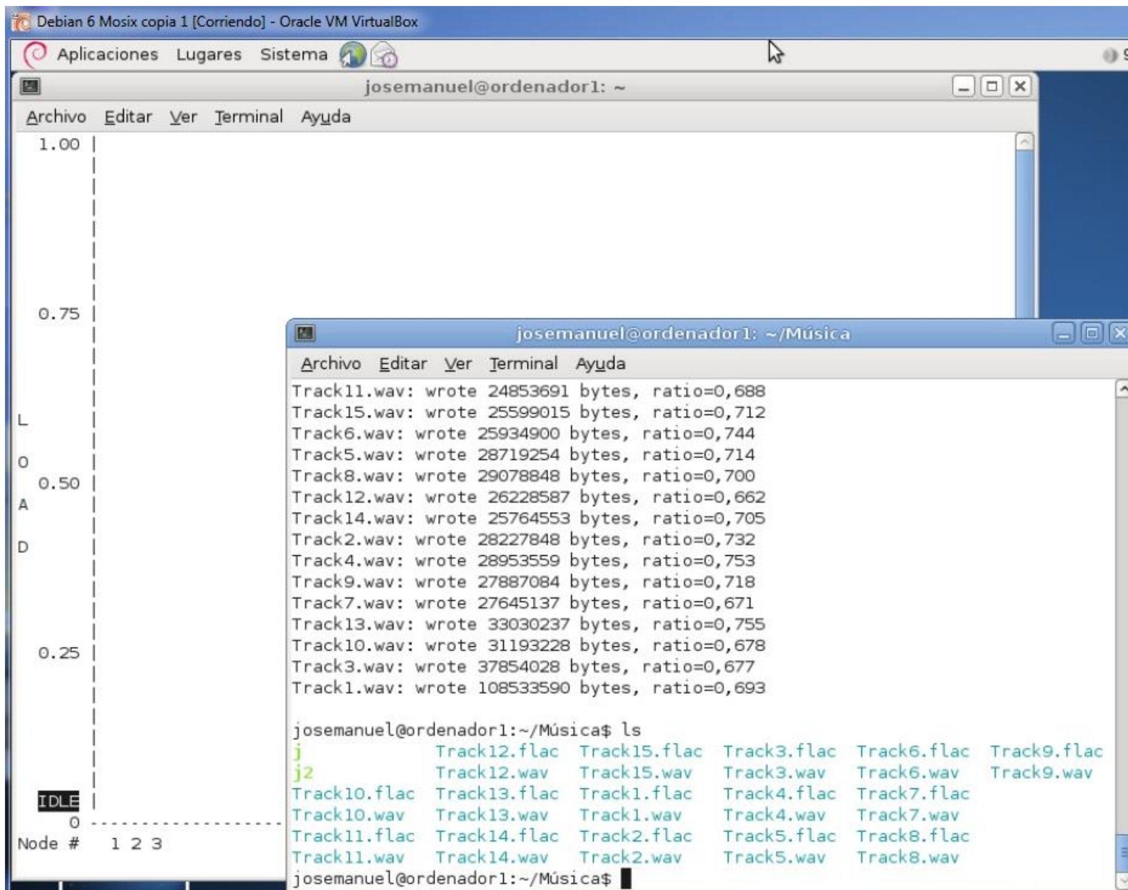
```
josemanuel@ordenador1: ~/Música
Archivo  Editar  Ver  Terminal  Ayuda
josemanuel@ordenador1:~/Música$ ls
j            Track11.wav  Track14.wav  Track2.wav  Track5.wav  Track8.wav
j2           Track12.wav  Track15.wav  Track3.wav  Track6.wav  Track9.wav
Track10.wav  Track13.wav  Track1.wav   Track4.wav  Track7.wav
josemanuel@ordenador1:~/Música$
```

La compresión se puede hacer ejecutando este script:

```
#! /bin/bash
for x in Tra*.wav
do
        mosrun flac -8 $x &
done
```

```
josemanuel@ordenador1: ~
Archivo  Editar  Ver  Terminal  Ayuda
2.66
2.00
L
O  1.33
A
D
0.67
0
Node #   1 2 3
```

```
josemanuel@ordenador1: ~/Música
Archivo  Editar  Ver  Terminal  Ayuda
Track7.wav: WARNING: skipping unknown sub-chunk 'LIST' (use --keep-foreign-metad
ata to keep)

flac 1.2.1, Copyright (C) 2000,2001,2002,2003,2004,2005,2006,2007  Josh Coalson
flac comes with ABSOLUTELY NO WARRANTY.  This is free software, and you are
welcome to redistribute it under certain conditions.  Type `flac' for details.

Track8.wav: WARNING: skipping unknown sub-chunk 'LIST' (use --keep-foreign-metad
ata to keep)

flac 1.2.1, Copyright (C) 2000,2001,2002,2003,2004,2005,2006,2007  Josh Coalson
flac comes with ABSOLUTELY NO WARRANTY.  This is free software, and you are
welcome to redistribute it under certain conditions.  Type `flac' for details.

Track3.wav: 0% complete, ratio=0,239Track9.wav: WARNING: skipping unknown sub-ch
unk 'LIST' (use --keep-foreign-metadata to keep)
Track11.wav: wrote 24853691 bytes, ratio=0,688
Track15.wav: wrote 25599015 bytes, ratio=0,712
Track6.wav: wrote 25934900 bytes, ratio=0,744
Track5.wav: wrote 28719254 bytes, ratio=0,714
Track8.wav: wrote 29078848 bytes, ratio=0,700
Track12.wav: wrote 26228587 bytes, ratio=0,662
Track14.wav: wrote 25764553 bytes, ratio=0,705
Track13.wav: 92% complete, ratio=0,763
```


Debian 6 Mosix copia 1 [Corriendo] - Oracle VM VirtualBox
Aplicaciones Lugares Sistema
josemanuel@ordenador1: ~
Archivo Editar Ver Terminal Ayuda
1.00
0.75
L O A D
0.50
0.25
IDLE
0
Node # 1 2 3
josemanuel@ordenador1: ~/Música
Archivo Editar Ver Terminal Ayuda
Track11.wav: wrote 24853691 bytes, ratio=0,688
Track15.wav: wrote 25599015 bytes, ratio=0,712
Track6.wav: wrote 25934900 bytes, ratio=0,744
Track5.wav: wrote 28719254 bytes, ratio=0,714
Track8.wav: wrote 29078848 bytes, ratio=0,700
Track12.wav: wrote 26228587 bytes, ratio=0,662
Track14.wav: wrote 25764553 bytes, ratio=0,705
Track2.wav: wrote 28227848 bytes, ratio=0,732
Track4.wav: wrote 28953559 bytes, ratio=0,753
Track9.wav: wrote 27887084 bytes, ratio=0,718
Track7.wav: wrote 27645137 bytes, ratio=0,671
Track13.wav: wrote 33030237 bytes, ratio=0,755
Track10.wav: wrote 31193228 bytes, ratio=0,678
Track3.wav: wrote 37854028 bytes, ratio=0,677
Track1.wav: wrote 108533590 bytes, ratio=0,693
josemanuel@ordenador1:~/Música$ ls
j Track12.flac Track15.flac Track3.flac Track6.flac Track9.flac
j2 Track12.wav Track15.wav Track3.wav Track6.wav Track9.wav
Track10.flac Track13.flac Track1.flac Track4.flac Track7.flac
Track10.wav Track13.wav Track1.wav Track4.wav Track7.wav
Track11.flac Track14.flac Track2.flac Track5.flac Track8.flac
Track11.wav Track14.wav Track2.wav Track5.wav Track8.wav
josemanuel@ordenador1:~/Música$

Pruebas de compresión de música MPICH2

Compresión de las 15 canciones de un CD de música utilizando MPI:

jmr.sh | comprimir.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main( int argc, char *argv[] ){
    int numprocs,myid;
    char cadena[30];
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    printf("\nHola desde el proceso: %d de %d\n",myid,numprocs);
//  printf(argv[myid+1]);
    sprintf(cadena,"flac %s",argv[myid+1] );
    system("hostname");
    system(cadena);
//  printf(cadena);
    printf("\n\n\n\n");
    MPI_Finalize();
    return 0;
}
```

jmr.sh | comprimir.c

```bash
#! /bin/bash
for x in Tra*.wav
do
        y=$y" "$x
        n=$(expr $n + 1)
done

mpiexec -np $n comprimir $y
```

```
josemanuel@ordenador1:~/jm/mpich2$ ./jmr.sh

Hola desde el proceso: 9 de 15
Hola desde el proceso: 10 de 15
Hola desde el proceso: 12 de 15
Hola desde el proceso: 0 de 15
Hola desde el proceso: 3 de 15
Hola desde el proceso: 4 de 15
Hola desde el proceso: 5 de 15
Hola desde el proceso: 2 de 15
Hola desde el proceso: 1 de 15
Hola desde el proceso: 8 de 15
Hola desde el proceso: 13 de 15
Hola desde el proceso: 11 de 15
Hola desde el proceso: 7 de 15
```

```
Hola desde el proceso: 14 de 15
Hola desde el proceso: 6 de 15
ordenador1
ordenador1
ordenador1
ordenador1
ordenador1
ordenador2
ordenador2
ordenador2
ordenador2
ordenador2
ordenador3
ordenador3
ordenador3
ordenador3
ordenador3
flac 1.2.1, Copyright (C) 2000,2001,2002,2003,2004,2005,2006,2007  Josh Coalson
flac comes with ABSOLUTELY NO WARRANTY.  This is free software, and you are
welcome to redistribute it under certain conditions.  Type `flac' for details.
Track13.wav: wrote 33076473 bytes, ratio=0,756
Track4.wav: wrote 28980058 bytes, ratio=0,753
Track15.wav: wrote 25646674 bytes, ratio=0,714
Track8.wav: 93% complete, ratio=0,7165
Track11.wav: wrote 24905397 bytes, ratio=0,689
Track2.wav: 71% complete, ratio=0,7344
Track14.wav: wrote 25820761 bytes, ratio=0,706
Track6.wav: wrote 25962130 bytes, ratio=0,745
Track9.wav: wrote 27902472 bytes, ratio=0,719
Track12.wav: 74% complete, ratio=0,653
Track5.wav: wrote 28742936 bytes, ratio=0,715
Track8.wav: wrote 29161556 bytes, ratio=0,702
Track7.wav: wrote 27696202 bytes, ratio=0,672
Track10.wav: wrote 31341133 bytes, ratio=0,681
Track2.wav: wrote 28342509 bytes, ratio=0,735
Track12.wav: wrote 26288203 bytes, ratio=0,663
Track3.wav: wrote 37928109 bytes, ratio=0,678
Track1.wav: wrote 109238246 bytes, ratio=0,698


josemanuel@ordenador1:~/jm/mpich2$ ls *.wav
Track10.wav  Track13.wav  Track1.wav  Track4.wav  Track7.wav
Track11.wav  Track14.wav  Track2.wav  Track5.wav  Track8.wav
Track12.wav  Track15.wav  Track3.wav  Track6.wav  Track9.wav
josemanuel@ordenador1:~/jm/mpich2$ ls *.flac
Track10.flac  Track13.flac  Track1.flac  Track4.flac  Track7.flac
Track11.flac  Track14.flac  Track2.flac  Track5.flac  Track8.flac
Track12.flac  Track15.flac  Track3.flac  Track6.flac  Track9.flac
josemanuel@ordenador1:~/jm/mpich2$
```

Tiempos dedicados a la compresión:

Tiempo que se tarda en comprimir las 15 canciones del CD:

MPICH2: 45 segundos

MOSIX: 1 minuto 40 segundos

Secuencialmente: 1 minuto 29 segundos

Para comprimir las canciones secuencialmente hay que ejecutar el script:

```
josemanuel@ordenador1:~/Música/a$ cat j
#! /bin/bash
for x in Tra*.wav
do
        flac -8 $x
done
```

Compresión de música con archivos grandes:

Compresión de las 15 canciones de un CD de música + 2 canciones de gran tamaño, en total 17 canciones:

```
-rw-r--r-- 1 josemanuel josemanuel  46021698 ene  2 19:04 Track10.wav
-rw-r--r-- 1 josemanuel josemanuel  36129186 ene  2 19:04 Track11.wav
-rw-r--r-- 1 josemanuel josemanuel  39640722 ene  2 19:04 Track12.wav
-rw-r--r-- 1 josemanuel josemanuel  43740258 ene  2 19:04 Track13.wav
-rw-r--r-- 1 josemanuel josemanuel  36569010 ene  2 19:04 Track14.wav
-rw-r--r-- 1 josemanuel josemanuel  35929266 ene  2 19:04 Track15.wav
-rw-r--r-- 1 josemanuel josemanuel 782993130 ene 20 17:04 Track16.wav
-rw-r--r-- 1 josemanuel josemanuel 782993130 ene 20 17:06 Track17.wav
-rw-r--r-- 1 josemanuel josemanuel 156598626 ene  2 19:03 Track1.wav
-rw-r--r-- 1 josemanuel josemanuel  38544690 ene  2 19:03 Track2.wav
-rw-r--r-- 1 josemanuel josemanuel  55914210 ene  2 19:03 Track3.wav
-rw-r--r-- 1 josemanuel josemanuel  38474130 ene  2 19:03 Track4.wav
-rw-r--r-- 1 josemanuel josemanuel  40205202 ene  2 19:03 Track5.wav
-rw-r--r-- 1 josemanuel josemanuel  34856754 ene  2 19:03 Track6.wav
-rw-r--r-- 1 josemanuel josemanuel  41214210 ene  2 19:03 Track7.wav
-rw-r--r-- 1 josemanuel josemanuel  41522322 ene  2 19:04 Track8.wav
-rw-r--r-- 1 josemanuel josemanuel  38829282 ene  2 19:04 Track9.wav
```

Tiempo que se tarda en comprimir las 15 canciones del CD + 2 canciones de gran tamaño:

MPICH2: 1 minuto 35 segundos

MOSIX: 2 minutos 4 segundos

Secuencialmente: 2 minuto 12 segundos

Compresión de 6 canciones de gran tamaño en MOSIX:

```
josemanuel@ordenador1:~/Música/a$ ls -l
total 6748116
-rwxr-xr-x 1 josemanuel josemanuel         54 ene 27 13:05 j
-rwxr-xr-x 1 josemanuel josemanuel         55 ene 27 13:06 j2
-rwxr-xr-x 1 josemanuel josemanuel         62 ene 27 13:05 j3
-rw-r--r-- 1 josemanuel josemanuel 1150542450 ene 27 12:59 Track1.wav
-rw-r--r-- 1 josemanuel josemanuel 1150542450 ene 27 13:03 Track2.wav
-rw-r--r-- 1 josemanuel josemanuel 1150542450 ene 27 13:04 Track3.wav
-rw-r--r-- 1 josemanuel josemanuel 1150542450 ene 27 13:18 Track4.wav
-rw-r--r-- 1 josemanuel josemanuel 1150542450 ene 27 13:19 Track5.wav
-rw-r--r-- 1 josemanuel josemanuel 1150542450 ene 27 13:20 Track6.wav


josemanuel@ordenador1:~/Música/a$ cat j3
#! /bin/bash
for x in Tra*.wav
do
      mosrun flac -8 $x &
done
josemanuel@ordenador1:~/Música/a$ ./j3


. . .


Track2.wav: wrote 31193228 bytes, ratio=0,678

josemanuel@ordenador1:~/Música/a$ ls -l
total 6931116
-rwxr-xr-x 1 josemanuel josemanuel         54 ene 27 13:05 j
-rwxr-xr-x 1 josemanuel josemanuel         55 ene 27 13:06 j2
-rwxr-xr-x 1 josemanuel josemanuel         62 ene 27 13:05 j3
-rw-r--r-- 1 root       josemanuel   31193228 ene 27 12:59 Track1.flac
-rw-r--r-- 1 josemanuel josemanuel 1150542450 ene 27 12:59 Track1.wav
-rw-r--r-- 1 root       josemanuel   31193228 ene 27 13:03 Track2.flac
-rw-r--r-- 1 josemanuel josemanuel 1150542450 ene 27 13:03 Track2.wav
-rw-r--r-- 1 root       josemanuel   31193228 ene 27 13:04 Track3.flac
-rw-r--r-- 1 josemanuel josemanuel 1150542450 ene 27 13:04 Track3.wav
-rw-r--r-- 1 root       josemanuel   31193228 ene 27 13:18 Track4.flac
-rw-r--r-- 1 josemanuel josemanuel 1150542450 ene 27 13:18 Track4.wav
-rw-r--r-- 1 root       josemanuel   31193228 ene 27 13:19 Track5.flac
-rw-r--r-- 1 josemanuel josemanuel 1150542450 ene 27 13:19 Track5.wav
-rw-r--r-- 1 root       josemanuel   31193228 ene 27 13:20 Track6.flac
-rw-r--r-- 1 josemanuel josemanuel 1150542450 ene 27 13:20 Track6.wav
josemanuel@ordenador1:~/Música/a$
```

Tiempo que se tarda en comprimir 6 canciones de gran tamaño en MOSIX:

Secuencialmente: 46 segundos

Utilizando MOSIX: 30 segundos

Se puede comprobar experimentalmente, que para que utilizar MOSIX salga a cuenta, hay que ejecutar programas que hagan cálculos intensivos durante periodos largos de tiempo. En las pruebas realizadas se puede apreciar que cuanto mayor es el tiempo de cómputo necesario para comprimir cada canción, más beneficioso es el uso de MOSIX.

Para tareas que tardan poco tiempo, MOSIX no sale a cuenta, ya que la migración de procesos y las entradas/salidas desde el nodo local (mediante la red) tiene un coste elevado de tiempo.

Se puede comprobar experimentalmente que utilizando MPICH2, el tiempo de computo tiende a dividirse entre el número de nodos (en estas pruebas no llega a dividirse entre 3). Hay que tener en cuenta que 2 de los 3 nodos están accediendo a los ficheros mediante la red a través de "Network File System (NFS)" y esto tiene un coste de tiempo.

También hay que tener en cuenta que: Todas las pruebas que se explican en estos textos se están ejecutando en un clúster de 3 máquinas virtuales, las cuales se están ejecutando en una máquina real de 2 núcleos.

El tiempo que se tarda en comprimir las 15 canciones del CD de música en la máquina real (Intel Core i3 con 2 núcleos a 3.3 GHz) en la cual se han realizado todas las pruebas es:

- Secuencialmente: 1 minuto 5 segundos.

- Todas a la vez en segundo plano: 34 segundos.

- De dos en dos: 34 segundos

Apagando nodos mientras ejecutan procesos MOSIX

En MOSIX se pueden añadir y quitar ordenadores del clúster en caliente sin problemas. La forma correcta de apagar el servicio MOSIX de un nodo, es invocando la instrucción: `mosctl shutdown` en dicho nodo. Al hacer esto, todos los procesos migran a sus respectivos nodos (*home-nodes*) y se apaga el servicio MOSIX en el nodo en el cual se invoca la instrucción (esto se puede comprobar experimentalmente).

También se puede comprobar experimentalmente que: Si se apaga alguna de las máquinas de forma normal, mediante la instrucción: `shutdown -h now` , todos los procesos migran a sus respectivos nodos (*home-nodes*) y el servicio MOSIX se apaga correctamente y si el ordenador se vuelve a encender y se invoca la instrucción: `mosd` , el nodo vuelve a estar disponible en el clúster y los procesos pueden volver a migrar a este nodo. En el caso de hacer `mosctl shutdown` , para que el nodo vuelva a estar disponible hay que reiniciar el ordenador e invocar la instrucción `mosd`.

En un clúster MOSIX: Se puede arrancar los programas desde cualquier nodo del clúster pero normalmente hay un nodo (el nodo de acceso) en el cual los usuarios se logan y arrancan los procesos. De esta forma: Si un nodo (que no sea el de acceso) se apaga, los procesos que se estaban ejecutando en dicho nodo migran a sus respectivos *home-nodes* antes de apagarse y pueden continuar ejecutándose y migrar a otro nodo.

Checkpoints en MOSIX:

En MOSIX se pueden hacer *checkpoints*. Cuando se hace un *checkpoint* de un proceso, se guarda la imagen del proceso en un archivo y después se puede recuperar y continuar con los cálculos desde el mismo punto.

Los *checkpoints* se pueden disparar añadiendo al comando `mosrun` , los modificadores: `-C nombre_archivo  -A numero_de_minutos`. Con el modificador `-C` se indica el nombre del archivo donde se guardará el *checkpoint* y el modificador `-A` indica cada cuantos minutos se hará un *checkpoint*.

Por ejemplo, si se invoca la instrucción `mosrun -C miCheckPoint -A 1 programa` , el programa se ejecutará y cada minuto se guardará un *checkpoint* del proceso, de manera que el primer *checkpoint* se guardará en el archivo `miCheckPoint.1` , el segundo *checkpoint* se guardará en el archivo `miCheckPoint.2` , el tercer *checkpoint* se guardará en el archivo `miCheckPoint.3` y así sucesivamente.

Se puede continuar la ejecución de un programa desde el punto en el cual se guardó un *checkpoint*, esto se hace invocando la instrucción: `mosrun -R miCheckPoint` , donde `miCheckPoint` es el nombre de archivo en el cual está guardado el *checkpoint* del proceso que queremos recuperar.

También se pueden disparar los *checkpoints* desde dentro del programa utilizando la interfaz de *checkpoints* de MOSIX. (No requiere el uso de ninguna librería adicional)

A continuación se muestra un ejemplo extraído de la página web oficial de MOSIX:

El siguiente programa realiza 100 unidades de trabajo y utiliza el argumento “unidad de *checkpoint*” para disparar un *checkpoint* una vez acabada dicha unidad de trabajo, el nombre de archivo de *checkpoint* (checkpointfile) es utilizado para guardar el *checkpoint*

```
#include < stdlib.h>
#include < unistd.h>
#include < string.h>
#include < stdio.h>
#include < fcntl.h>
#include < sys/stat.h>
#include < sys/types.h>

//Estableciendo el archivo checkpoint desde dentro del proceso
//Esto también se puede hacer con el argumento -C de mosrun

int setCheckpointFile(char *file) {
     int fd;

     fd = open("/proc/self/checkpointfile", 1|O_CREAT, file);
     if (fd == -1) {
        return 0;
     }
     return 1;

}
// Disparando un checkpoint desde dentro del proceso
int triggerCheckpoint() {
     int fd;
     fd = open("/proc/self/checkpoint", 1|O_CREAT, 1);
     if(fd == -1) {
```

```
        fprintf(stderr, "Error doing self checkpoint \n");
        return 0;
     }
     printf("Checkpoint was done successfully\n");
     return 1;
}

int main(int argc, char **argv) {
     int j, unit, t;
     char *checkpointFileName;
     int checkpointUnit = 0;

     if(argc < 3) {
        fprintf(stderr, "Usage %s < checkpoint-file> < unit> \n", argv[0]);
        exit(1);
     }

     checkpointFileName = strdup(argv[1]);
     checkpointUnit = atoi(argv[2]);
     if(checkpointUnit < 1 || checkpointUnit > 100) {
        fprintf(stderr, "Checkpoint unit should be > 0 and < 100\n");
        exit(1);
     }

     printf("Checkpoint file: %s\n", checkpointFileName);
     printf("Checkpoint unit: %d\n", checkpointUnit);



//Estableciendo el archivo checkpoint desde dentro del proceso
// esto también se puede hacer con el argumento –C de mosrun

     if(!setCheckpointFile(checkpointFileName)) {
        fprintf(stderr, "Error setting the checkpoint filename from within the
process\n");
        fprintf(stderr, "Make sure you are running this program via mosrun\n");
        return 0;
     }


//Bucle principal que se ejecuta 100 veces(100 unidades),cambiar si se desea que
se ejecuten mas unidades
     for( unit = 0; unit < 100 ; unit++ ) {
        // Se consume tiempo de cpu  (simulando la ejecución de una aplicación)




//Cambiar el numero de abajo para que cada bucle consuma mas o menos tiempo
        for( t=0, j = 0; j < 1000000 * 500; j++ ) {
          t = j+unit*2;
       }
       printf("Unit %d done\n", unit);

//Disparando un checkpoint desde dentro del proceso
       if(unit == checkpointUnit) {
          if(!triggerCheckpoint())
             return 0;
          }
       }
       return 1;
}
```

Para compilar: **gcc -o checkpoint_demo checkpoint_demo.c**
Para ejecutar: **mosrun checkpoint_demo**

Ejecución:

mosrun ./checkpoint_demo ccc 5

```
Checkpoint file: ccc
Checkpoint unit: 5
Unit 0 done
Unit 1 done
Unit 2 done
Unit 3 done
Unit 4 done
Unit 5 done
Checkpoint was done successfully
Unit 6 done
Unit 7 done
Unit 8 done
 ^C
El programa dispara un checkpoint después de la unidad 5. El checkpoint se
guarda en ccc.1. Después de la unidad 8 la ejecución del programa fue
interrumpida.
```

```
Para continuar: > mosrun -R ccc.1
Checkpoint was done successfully
Unit 6 done
Unit 7 done
Unit 8 done
Unit 9 done
Unit 10 done
...
```

El programa continua desde el punto en el cual se disparó el *checkpoint*.

Los *checkpoints* disparados desde el programa son una buena solución: Cuando se ejecutan programas de cálculo intensivo en los cuales se ejecutan bucles donde, en cada iteración se dedica una gran cantidad de tiempo de cálculo. En estos casos al terminar cada iteración se puede hacer un *checkpoint* para guardar el trabajo realizado hasta el momento.

Conclusiones y comparativa MPICH2 / MOSIX

Cuando lo que se desea es la máxima eficiencia en tiempo de cómputo, la mejor solución es MPICH2 pero esta solución requiere que los programas estén programados especialmente utilizando la librería `mpi.h` (Paralelización de programas).

En la prueba de compresión de música, el coste de la programación MPI fue relativamente bajo y el beneficio en tiempo de cómputo fue alto.

En estos textos salen programas relativamente sencillos pero la paralelización de programas puede ser una tarea compleja.Sin embargo utilizando MOSIX:

-No es necesario la paralelización de programas utilizando librerías especiales como `mpi.h`

-No es necesario modificar los programas ni utilizar librerías especiales ni copiar archivos en los nodos remotos.

-Los usuarios pueden *logarse* en cualquier nodo y ejecutar sus programas sin saber en qué nodo se están ejecutando, la migración de procesos es automática y transparente para el usuario.

MOSIX es adecuado para ejecutar tareas de computo intensivo (cálculos de larga duración) con bajo/moderado uso de entrada/salida pero para tareas que hacen uso intensivo de entrada/salida no es una buena solución.

Otra cosa a tener en cuenta es que en MOSIX se pueden hacer *checkpoints* de los procesos en ejecución. Cuando se hace un *checkpoint* de un proceso: Se guarda la imagen del proceso en un archivo y después se puede recuperar y continuar con los cálculos desde el mismo punto.

Además, en MOSIX se pueden apagar los nodos en caliente y los procesos que estaban ejecución en dichos nodos pueden migrar a otro nodo y continuar ejecutándose. Y si el nodo se vuelve a encender, este puede volver a estar disponible para el clúster.

Ventajas de utilizar máquinas virtuales

Se puede comprobar experimentalmente que:

-Al utilizar máquinas virtuales (Virtual Box en este caso) tenemos la opción de pausar las máquinas y tomar instantáneas y guardarlas. Y estas se pueden recuperar en el futuro para continuar con los cálculos.

-En MPICH2 y PVM en ambos casos: Si se pone en pausa alguna de las máquinas virtuales, esto no produce ningún error: Al quitar el pause continua la ejecución normalmente.

-Si se hace esto mismo en MOSIX: Se imprimen por pantalla mensajes de error y no se ejecutan las todas tareas correctamente.

El uso de máquinas virtuales es una buena solución cuando estas se ejecutan en una máquina real con igual o más núcleos reales que máquinas virtuales.

Si lo que se desea es: Ejecutar tareas en el menor tiempo de cómputo, lo mejor es utilizar máquinas reales.

Mare Nostrum III

Introducción

A continuación se muestran unos extractos de documentación obtenida mediante la página oficial del BSC:

Una **supercomputadora** o superordenador es una máquina con una capacidad de cálculo muy superior al resto. La base de los superordenadores actuales son chips y procesadores como los que utilizamos en los ordenadores de casa, pero en gran cantidad y conectados entre ellos para poder multiplicar su potencia.
Los superordenadores fueron introducidos en la década de los **sesenta**, diseñados principalmente por el ingeniero y matemático **Seymor Cray** en la compañía Control Data Corporation (CDC).

En el campo científico los superordenadores son parte imprescindible en multitud de estudios. Estos son algunos ejemplos de las investigaciones que se pueden hacer gracias a estas máquinas:

o Recopilar datos del **clima** pasado y actual y predecir el clima futuro
o Estudiar el **sol** y el clima espacial
o Simular cómo un **tsunami** afecta una determinada costa o ciudad
o Simular explosiones de estrellas **supernovas** en el espacio
o Probar la **aerodinámica** de los aviones
o Simular diferentes procesos que se producen en el cuerpo humano y otros organismos vivos: por ejemplo, cómo se pliegan las proteínas y cómo esto puede afectar a las personas que padecen enfermedades como el **Alzheimer**, el asma, la diabetes o diferentes tipos de cáncer.
o Simular explosiones **nucleares**, accidentes, catástrofes naturales, limitando la necesidad de hacer pruebas reales.

Dos veces al año, en junio y en noviembre, se hace pública la lista del **Top500**, el ranking de los superordenadores más potentes del mundo. MareNostrum llegó a situarse como la supercomputadora más potente de Europa y la cuarta del mundo en el año 2006

Otra lista importante: **The Green 500**, que analiza la relación entre capacidad de cálculo y consumo de energía. El supercomputador MinoTauro del BSC-CNS llegó a ser en noviembre de 2011 la máquina más eficiente energéticamente de Europa y la séptima del mundo.

El BSC-CNS y MareNostrum es Centro pionero de la supercomputación en España. Su especialidad es la computación de altas prestaciones (HPC, por sus siglas en inglés) y su misión es doble: desarrollar investigación propia y ofrecer infraestructuras y conocimiento en supercomputación a la comunidad científica y la sociedad.
Es Centro de Excelencia Severo Ochoa, gestiona la Red Española de Supercomputación (RES) y es miembro de primer nivel de la infraestructura de investigación europea PRACE (Partnership for Advanced Computing in Europe).
Como centro de investigación, cuenta con más de 300 científicos, divididos en cuatro grandes áreas: ciencias de la computación, ciencias de la vida, ciencias de la tierra y aplicaciones computacionales en ciencia e ingeniería.
Estos científicos desarrollan sus propias investigaciones y participan en proyectos de otros centros, facilitando el uso de la HPC con su experiencia en el desarrollo de aplicaciones aptas para este tipo de computación. Los especialistas en HPC son, hoy en día, un eslabón imprescindible para proyectos científicos de las más diversas especialidades.
Como centro de servicios, dispone de varios supercomputadores y repositorios de datos de gran capacidad. El superordenador MareNostrum, con una capacidad de más de un Petaflop/s, es el emblema del centro. MinoTauro ha sido reconocido por su arquitectura energéticamente eficiente.

A continuación se muestra el software que debe estar instalado en el Mare Nostrum III y en todos los supercomputadores de la RES (Red Española de Supercomputación):

Software available in MareNostrum

You may request that any of this software may be installed in any RES-Red Española de Supercomputación machine

ABINIT	ADA	AMBER	ANT
ARPREC	ATLAS	AUTOCONF	AUTOMAKE
BIOPERL	BLAS	BLAST	BLASTZ
BLENDER	BOOST	CDO	CEPBATOOLS
CFITSIO	CHAPEL	CHARMM	CLOUDY
CLUSTAL	CMAKE	CNS	CP2K
CPMD	CPPUNIT	CRYSTAL	DAP
DDD	DDT	DL_POLY	DYNINST
ECLIPSE	ESMF	ESSL	F2C
fastDNAml	FELIB	FERRET	FFTW
FTPS	GADGET	GAMESS	GAMESS-UK
GCC	GLPK	GMP	GNUPLOT
GOTO	GRACE	GRACOS	GRID-S
GROMACS	GSL	GTK+	H4H5TOOLS
HDF4	HDF5	HEALPIX	HMMER
HYPHY	IBM XL Compilers	IHPCT	IMAGEMAGICK
IOAPI	IOTRACK	JASPER	LAMMPS
LAPACK	LIBTOOL	LP_SOLVE	MASS
MCNP	METIS	MM5	MMTSB
MOLDEN	MPIBLAST	MPICH	MPICH2
MPIP	MPQC	MrBayes	MUMPS
MX	NAMD	NCARG	NCDAP
NCL	NCVIEW	NETCDF	NUMPY
NWCHEM	OCTAVE	OCTOPUS	OMNIORB
ONLA	OPENMPI	P-NETCDF	PAML
PAPI	PARAVIEW	PARMETIS	PARMS
PAVE	PCASUITE	PEPC	PERL
PESSL	PETSC	PHYLOBAYES	PNG
PORT	POVRAY	PTRAJ	pyMPI
PYTHON	Q	QD	QT4
R	RAxML	SBML	SCALAPACK
SCALASCA	SCONS	SIESTA	SLEPC
SPARSEKIT	SPRNG	SWIG	T-COFFEE
TCL	UDUNITS	VALGRIND	VAMPIR
VASP	VIS5D	VISIT	VMD
VTK	WRF	WSMP	WXGTK
XCrySDen			

En esta imagen, obtenida en la página oficial del BSC, se puede apreciar que todos los supercomputadores de las RES deben tener instalado MPICH2.

Extractos de la guía de usuario del MN3:

A continuación se muestra unos extractos de la guía de usuario del MN3 traducidos al castellano:

Sistema operativo: Linux SuSe 11 SP2

Conectarse mediante *Secure Shell*

La manera de *logarse* en el Mare Nostrum es mediante *Secure Shell* a través de los nodos de acceso:

```
mn1.bsc.es
mn2.bsc.es
```

ejemplo:

```
> ssh -l username mn1.bsc.es
username's password:
Last login: Fri Sep 2 15:07:04 2011 from XXXX
username@login1 ~]$
```

Sistemas de archivos

Hay 3 tipos diferentes de almacenamiento disponibles en cada nodo.

-Sistema de archivos raíz: Es el sistema de archivos donde reside el sistema operativo.

-Sistema de archivos GPFS (*General Paralel File System*): GPFS es un sistema de archivos en red distribuido al cual pueden acceder todos los nodos.

-Disco duro local: Cada placa tiene un disco duro interno.

Sistema de archivos raíz

El sistema de archivos raíz donde se encuentra el sistema operativo no reside en la placa, es un sistema de archivos NFS montado desde uno de los servidores.

Sistema de archivos GPFS

El IBM *General Parallel File System* (GPFS) es un sistema de archivos de disco compartido de alto rendimiento que proporciona acceso rápido y seguro desde todas las placas del clúster al sistema de ficheros global. GPFS permite a aplicaciones paralelas tener acceso simultáneo a un conjunto de archivos desde cualquier nodo que tenga montado el sistema de archivos GPFS y proporciona un alto nivel de control sobre todas las operaciones del sistema de archivos. Para la mayoría de trabajos se recomienda el uso de estos sistemas de archivos, porque GPFS proporciona entrada/salida de alto rendimiento porque puede acceder a bloques de datos de archivos individuales a través de múltiples discos en múltiples dispositivos de almacenamiento y leer /escribir estos bloques en paralelo. Además GPFS puede leer o escribir bloques grandes de datos en una sola operación de entrada/salida, de este modo se minimiza la sobrecarga.

Estos son los sistemas de archivos GPFS disponibles desde todos los nodos en el Mare Nostrum III:

`/gpfs/home`: Este sistema de archivos contiene los directorios home de todos los usuarios. Al logarte en Mare Nostrum III, empiezas en tu directorio home por defecto. Todos los usuarios tienen su propio directorio home para guardar los ejecutables, sus fuentes y sus datos personales.

`/gpfs/projects`: Además del directorio home, hay un directorio en `/gpfs/projects` para cada grupo de usuarios del Mare Nostrum III. Por ejemplo, el grupo `bsc01` tendrá el directorio `/gpfs/proejects/bsc01` listo para usar. Este espacio está destinado a almacenar datos que necesitan ser compartidos entre los usuarios del mismo grupo o proyecto.

`/gpfs/scratch`: Cada usuario del Mare Nostrum III tendrá un directorio sobre `/gpfs/scratch`, se debe utilizar este espacio para almacenar los archivos temporales de las tareas durante su ejecución

`/apps`: En este sistema de archivos residen las aplicaciones y librerías que tienen que estar instaladas en Mare Nostrum III. Son aplicaciones disponibles para uso general.

Disco duro local

Todas las placas tienen un disco duro local que puede utilizarse como un espacio scratch local para almacenar archivos temporales durante la ejecución de una tarea de usuario. Este espacio está montado en el directorio `/scratch`. El espacio del sistema de archivos `/scratch` es de 500 GB. Todos los datos almacenados en estos discos duros locales en las placas de cómputo no están disponible desde los nodos de login. Los datos de los discos duros locales no se eliminan automáticamente, cada tarea tiene que eliminar sus datos al finalizar.

Ejecutando tareas

`LSF` es la utilidad utilizada en el Mare Nostrum III para el soporte de procesamiento por lotes, por lo que todas las tareas deben ejecutarse a través de ella.

Enviando tareas

Una tarea es la unidad de ejecución para `LSF`. Una tarea está definida por un archivo de texto que contiene un conjunto de directivas que describen la tarea y los comandos a ejecutar. Hay un límite de 3600 bytes para el tamaño de este archivo de texto.

Comandos `LSF`:

Estas son las directivas básicas para enviar tareas:

```
bsub < job_script
```

Envía un script de tarea al sistema de cola.

```
bjobs [-w] [-X] [-l job_id]
```

Muestra todas las tareas enviadas.

```
bkill <job_id>
```

Elimina la tarea del sistema de cola, cancelando la ejecución de los procesos, si están todavía ejecutándose.

Directivas de tareas:

Una tarea debe contener una serie de directivas para informar al sistema de procesamiento por lotes sobre las características de la tarea. Estas directivas aparecen como comentarios en el script de tarea, con la sintaxis siguiente:

```
#BSUB –option value
```

```
#BSUB –J job_name
```

El nombre de la tarea

```
#BSUB –q debug
```

Esta cola está dirigida para pequeños tests, hay un límite de 1 tarea por usuario, usando más de 64 CPUs (4 nodos) y una hora como máximo.

```
#BSUB –W HH:MM
```

El límite de tiempo. Este es un campo obligatorio y se debe establecer con un valor mayor que el tiempo de ejecución real de la tarea y menor que el límite de tiempo concedido al usuario. Tanscurrido este periodo de tiempo la tarea será eliminada.

```
#BSUB –cwd pathname
```

El directorio de trabajo de la tarea (donde la tarea se ejecutará). Si no se especifica, es el directorio de trabajo actual en el momento que la tarea fue enviada.

```
#BSUB -e/-eo file
```

El nombre del archivo que recoge la salida estándar de error (stderr output) de la tarea. Se puede utilizar %J para job_id. La opción –e añadirá el archivo, -eo reemplazará el archivo.

```
#BSUB -o/-oo file
```

El nombre del archivo que recoge la salida estándar (stdout) de la tarea. La opción –o añadirá el archivo, -oo reemplazará el archivo.

```
#BSUB –n number
```

El número de procesos a arrancar.

```
#BSUB -R"span[ptile=number]"
```

Número de procesos asignados a un nodo.

```
man bsub
```

Manual del comando `bsub`

Ejemplos:

Ejemplo para una tarea secuencial:

```
#!/bin/bash
#BSUB -n 1
#BSUB -oo output_%J.out
#BSUB -eo output_%J.err
#BSUB -J sequential
#BSUB -W 00:05
./serial.exe
```

La tarea será enviada invocando:

```
bsub < ptest.cmd
```

Ejemplo para una tarea secuencial usando OpenMP:

```
#!/bin/bash
#BSUB -n 1
#BSUB -oo output_%J.out
#BSUB -eo output_%J.err
#BSUB -J sequential_OpenMP
#BSUB -W 00:05
export OMP_NUM_THREADS=16
./serial.exe
```

Ejemplo para una tarea paralela:

```
#!/bin/bash
#BSUB -n 128
#BSUB -o output_%J.out
#BSUB -e output_%J.err
# In order to launch 128 processes with 16 processes per node:
#BSUB -R"span[ptile=16]"
#BSUB -x # Exclusive use
#BSUB -J parallel
#BSUB -W 02:00

# You can choose the parallel environment through modules
module load intel openmpi
mpirun ./wrf.exe
```

Ejemplo para una tarea paralela usando threads:

```
#!/bin/bash
# The total number of MPI processes:
#BSUB -n 128
#BSUB -oo output_%J.out
#BSUB -eo output_%J.err
# It will allocate 4 MPI processes per node:
#BSUB -R"span[ptile=4]"
#BSUB -x # Exclusive use
#BSUB -J hybrid
#BSUB -W 02:00
# You can choose the parallel environment through
# modules
module load intel openmpi
# 4 MPI processes per node and 16 cpus available
# (4 threads per MPI process):
export OMP_NUM_THREADS=4
mpirun ./wrf.exe
```

Entorno de software

Compiladores C

En el clúster puedes encontrar estos compiladores de C / C++ :

```
icc / icpc -> Intel C/C++ Compilers version 13.0
man icc
man icpc

gcc /g++ -> GNU Compilers for C/C++, Version 4.3.4
man gcc
man g++
```

Estos son los tamaños por defecto de los tipos de datos standard de C / C++ en Mare Nostrum III :

Type	*Length (bytes)*
bool (c++ only)	1
char	1
wchar_t	4
short	2
int	4
long	8
float	4
double	8
long double	16

Paralelismo de memoria distribuida

Para compilar programas MPI se recomienda utilizar: `mpicc`, `mpicxx` para codigo fuente C y C++. Primero se necesita elegir el entorno paralelo: `module load openmpi` / `module load impi`. De este modo se incluyen todas las librerías necesarias para construir aplicaciones MPI sin tener que especificar todos los detalles a mano.

```
mpicc a.c -o a.exe
mpicxx a.C -o a.exe
```

Paralelismo de memoria compartida

Las directivas OpenMP están todas soportadas por los compiladores de C y C++ de Intel. Para usarlas hay que añadir `-openmp` a la instrucción de compilación.

```
icc -openmp -o exename filename.c
icpc -openmp -o exename filename.C
```

De esta manera se puede también mezclar código MPI + OpenMP .

A continuación se muestran unas explicaciones sobre estos extractos de la guía de usuario del Mare Nostrum III:

OpenMP es una API que define una interfaz para la programación multihilo en sistemas de memoria compartida.

En los ejemplos de ejecución de tareas de la guía de usuario del MN3, se muestra el script que se le pasa al gestor de tareas `LSF` para ejecutar la tarea llamada `hybryd` en la cual se combina: El paso de mensajes MPI con la programación multihilo OpenMP (o threads).

Los threads que lanza un proceso pueden compartir la memoria que hay disponible para la CPU donde se ejecutan. A esto, en la guía de usuario del MN3, se le ha llamado: "Paralelismo de memoria compartida" porque se puede paralelizar programas mediante la programación multihilo gracias a la memoria compartida. También se puede paralelizar programas mediante MPI gracias al paso de mensajes entre procesos y además se puede combinar el paso de mensajes con la capacidad de compartir memoria a nivel local: Los procesos pueden comunicarse mediante MPI y además, también pueden lanzar threads para tener varios hilos de ejecución y/o tener concurrencia entre procesamiento y Entrada/Salida. A la paralelización de programas mediante el paso de mensajes entre procesos, en la guía de usuario del MN3 se le ha llamado "Paralelismo de memoria distribuida".

Al gestor de tareas `LSF` hay que indicarle: Cuantos procesos se ejecutan, cuantos procesos se ejecutan en cada nodo y cuantos threads lanza cada proceso. Así, el gestor de tareas `LSF` puede repartir de forma adecuada los procesos entre los nodos del clúster con el objetivo de aprovechar al máximo los recursos.

Al programar y al definir las directrices de ejecución de las tareas hay que tener en cuenta las características del hardware para poder conseguir un rendimiento óptimo. Hay que tener en cuenta que: En el MN3, cada nodo dispone de 2 CPU´s de 8 núcleos (16 núcleos en cada nodo). Los threads que se ejecutan en el mismo nodo pueden compartir la memoria entre ellos.

En los ejemplos de ejecución de tareas de la guía de usuario del MN3, se muestra la ejecución de la tarea llamada `hybrid`, en la cual se le indica al gestor de tareas: Que se ejecuten 128 procesos MPI, que se ejecuten 4 procesos en cada nodo y también se indica que cada proceso MPI lanza 4 threads. De este modo se lanzan 16 hilos de ejecución (threads) en cada nodo. Como cada nodo tiene 16 núcleos, pueden ejecutarse los 16 hilos de ejecución simultáneamente en paralelo.

Documentación del MN3:

A continuación se muestra la documentación del Mare Nostrum III obtenida mediante la página oficial del BSC. (Traducida al castellano y se ha dejado la apariencia idéntica):

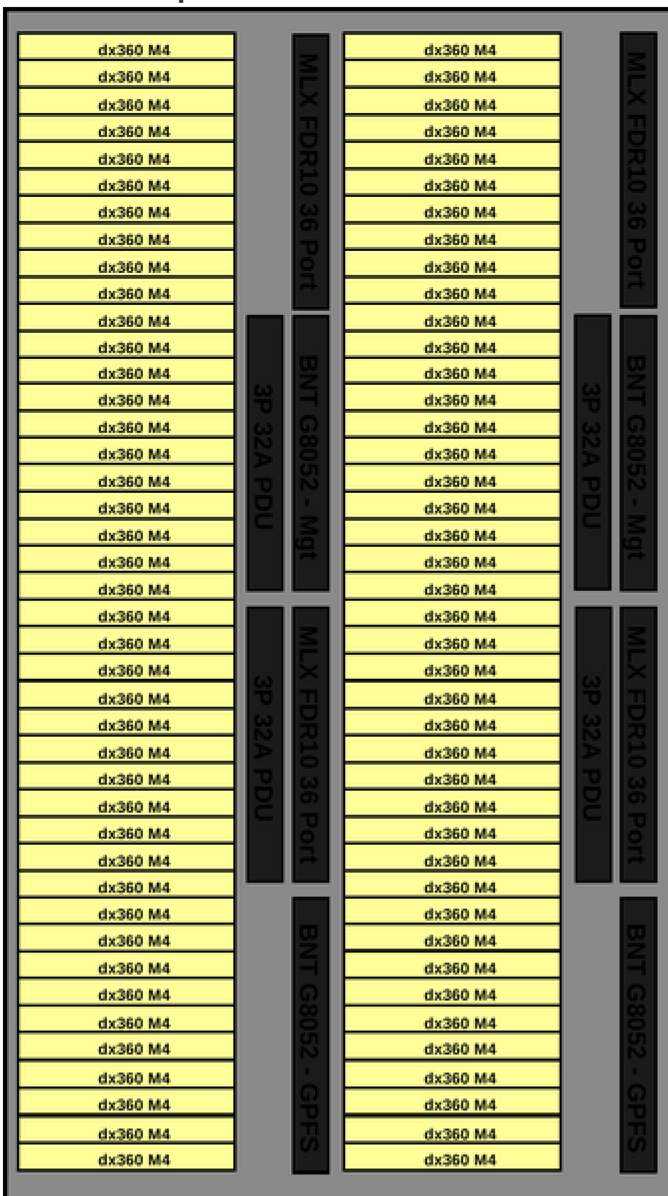
Racks de cómputo

El Mare Nostrum III tiene 36 racks dedicados a hacer cálculos. Estos racks tienen un total de 48.448 núcleos Intel SandyBridge con frecuencia 2,6 GHz y 94.625 TB de memoria total.

En total, cada rack tiene un total de 1.344 núcleos y 2.688 GB de memoria.

El rendimiento máximo por rack es 27,95 Tflops y el máximo consumo es 28,04 KW.

Rack de cómputo



Cada rack de cómputo IBM iDataPlex está compuesto por:

- 84 nodos de cómputo IBM dx360 M4
- 4 Switches Mellanox 36-port Managed FDR10 IB
- 2 Racks de Swhitches BNT RackSwitch G8052F (Red para la administración)
- 2 Racks de Swhitches BNT RackSwitch G8052F (Red para el GPFS)
- 4 Unidades de distribución de alimentación

Según el rendimiento por rack:

- 2.60 GHz x 8 flops/ciclo (AVX) = 20.8 Gflops/núcleo
- 16 núcleos x 20.8 Gflops/núcleo = 332.8 Gflops/nodo
- 84 nodos x 298.64 Gflops/nodo = 27.95 Tflops/rack

Chasis

Los nodos de cómputo dx360 M4 están agrupados en chasis de 2 unidades, teniendo 42 columnas de chasis de 2 unidades. Cada chasis de 32 unidades tiene 2 fuentes de alimentación de 900W independientes para que, si una falla, la otra continue funcionando.

Nodo de cómputo

Los nodos de cómputo son la última generación de servidores IBM System X: : iDataPlex dx360 M4. Estos nodos están basados en la tecnología Intel Xeon (R) y ofrecen alto rendimiento, flexibilidad y eficiencia energética.

Cada nodo de cómputo está compuesto por:

- 2 procesadores de 8 núcleos Intel Xeon ES-2670 a 2.6GHz,20MB de memoria de caché con rendimiento máximos de 332.8 Gflops por nodo
- 8 DIMM´s de 4GB DDR3 1.5V 1600MHz. Disponiendo 32GB por nodo y 2 GB por núcleo.En términos de ancho de banda de memoria, esta es la mejor configuración para el acceso a la memoria para los procesadores Intel E5-26xx EP (4 canales de memoria por chip)
- Disco duro local: IBM 500 GB 7.2K 6Gbps NL SATA 3.5
- Tarjeta de red MPI: Mellanox ConnectX-3 Dual Port QDR/FDR10 Mezz Card.
- 2 Targetas de red Ethernet Gigabit (red de administración y GPFS)

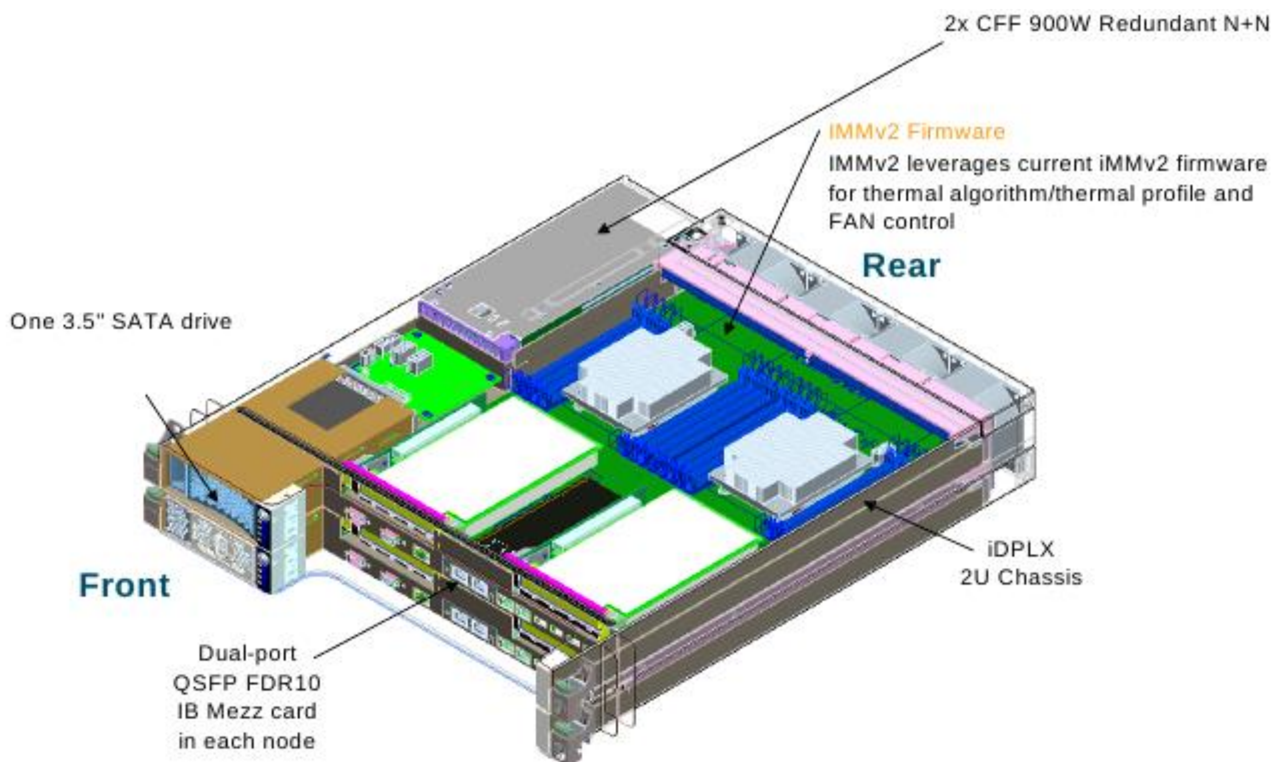
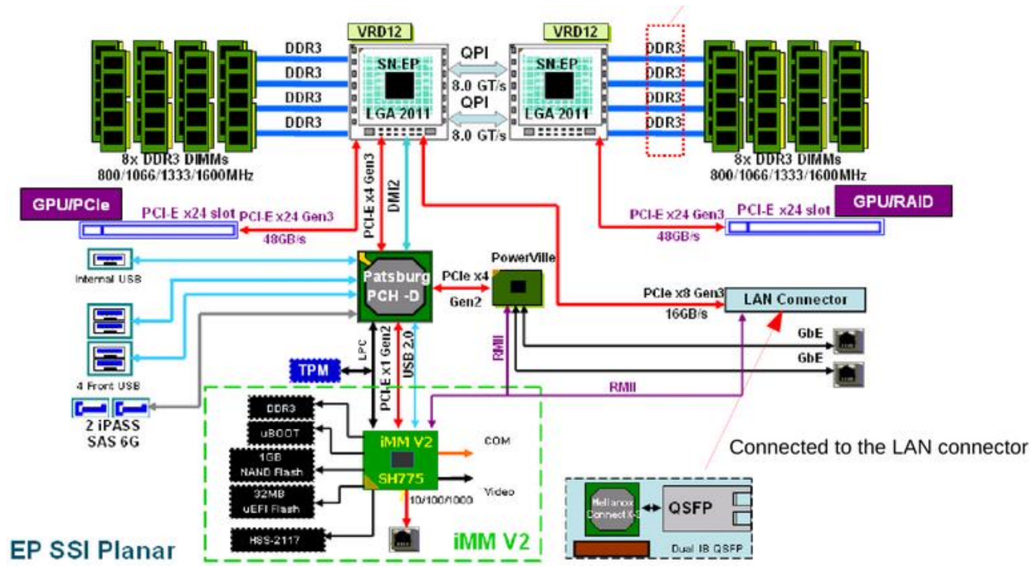


Diagrama de bloques del nodo de cómputo dx360 M4:



Racks Infiniband

Los 3,028 nodos de cómputo están interconectados mediante una red de conexión de alta velocidad: Infiniband FDR10. Los diferentes nodos están interconectados mediante cables de fibra óptica y Switches Infiniband Core 648-port FDR10.

Parte delantera

Parte trasera

4 de los racks del Mare Nostrum están dedicados a elementos de red los cuales permiten interconectar los diferentes nodos conectados a la red Infiniband.

Las principales características del switch Infiniband Core Mellanox 648-port son:

• De 100 ns a 300 ns de latencia de selección
• Rutero basado en hardware
• Control de congestión
• Aseguramiento de Calidad de servicio
• Sensores de temperatura y monitores de voltage
• Auto negociación de la velocidad de los puertos
• Ancho de banda biseccional total para todos los puertos
• Bandejas de ventilación intercambiables “en caliente”
• Indicadores LED de estado de los puertos y del sistema
• RoHS-5 compatible

A continuación se muestran unas imágenes extraídas de la guía de usuario del IBM dx360 M4 y de la guía de implementación obtenidas mediante la página oficial de IBM:

En las siguientes 2 imágenes se muestra el interior de un nodo (*blade,* placa) dx360 M4 :

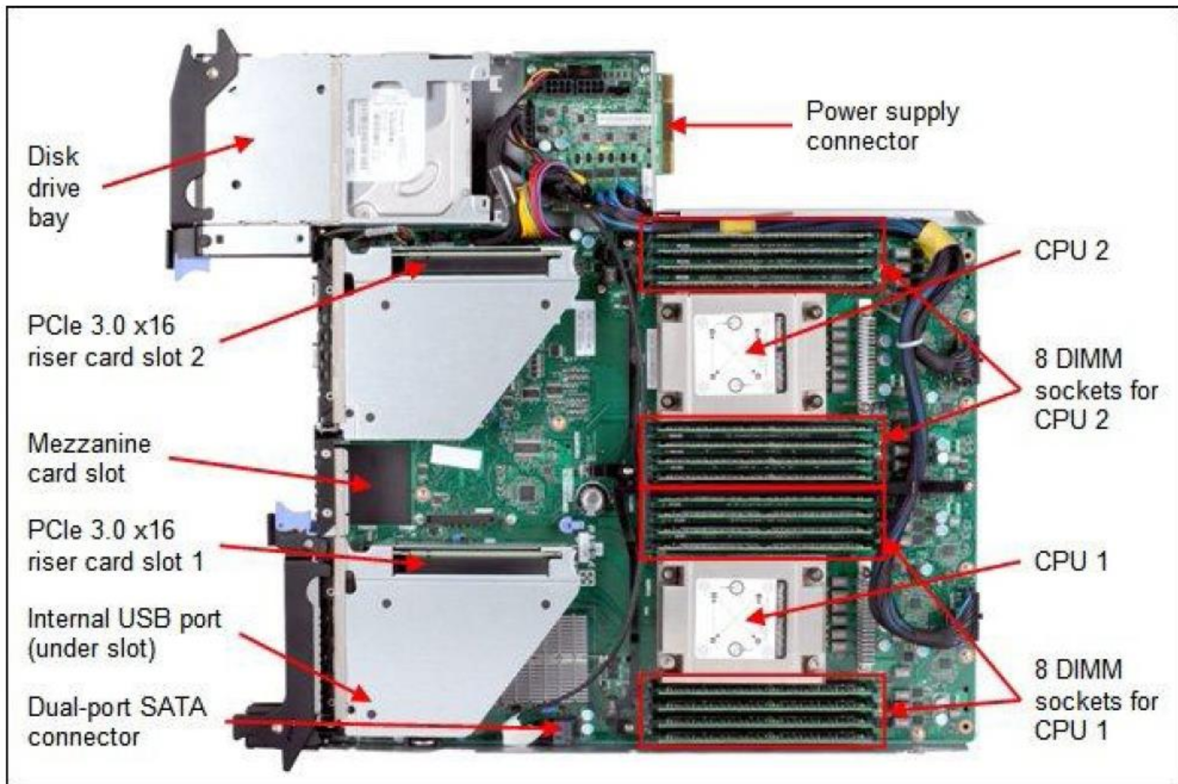


Figure 4. Inside view of the iDataPlex dx360 M4

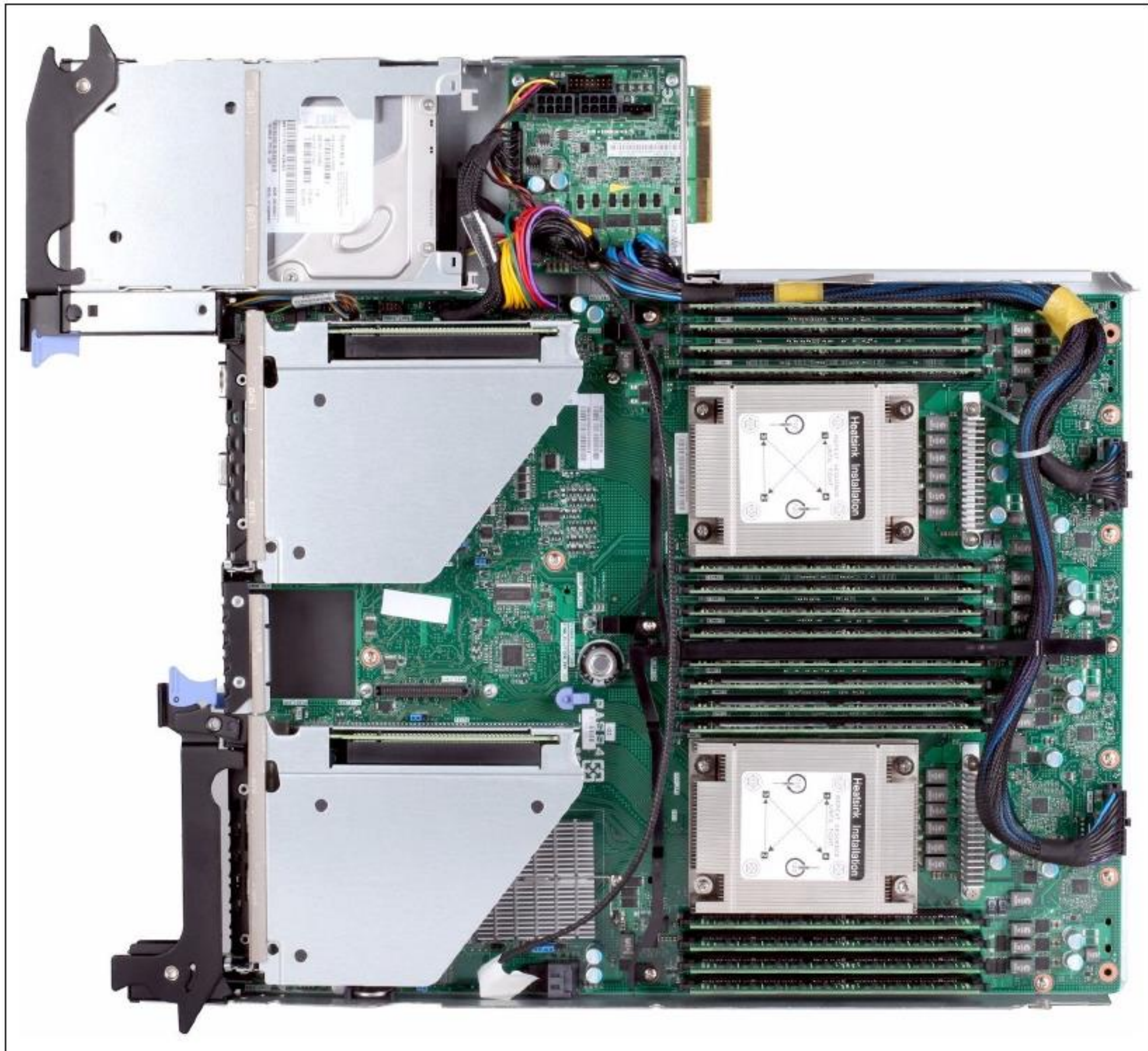


Figure 4-4 dx360 M4 system board

En la siguiente imagen se muestra los controles y conectores que tiene el dx360 M4:

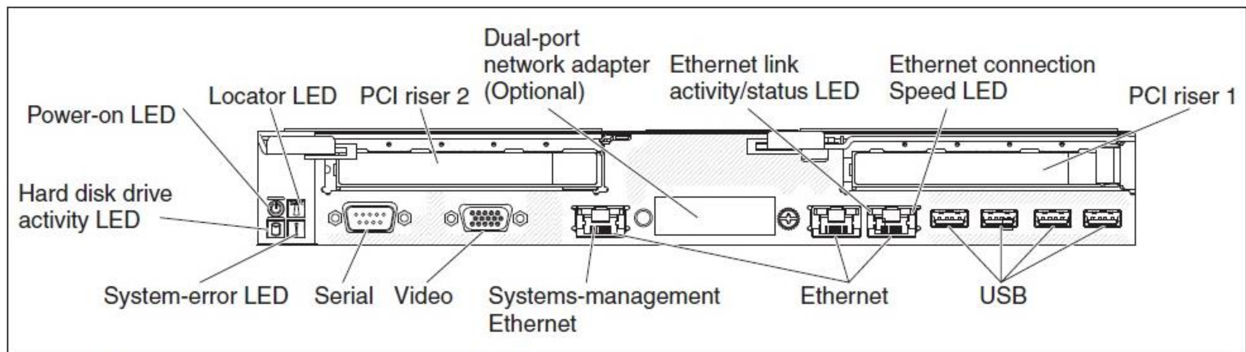


Figure 4-3 dx360 M4 controls and connectors

En la siguiente imagen se muestra el panel de operaciones del dx360 M4:

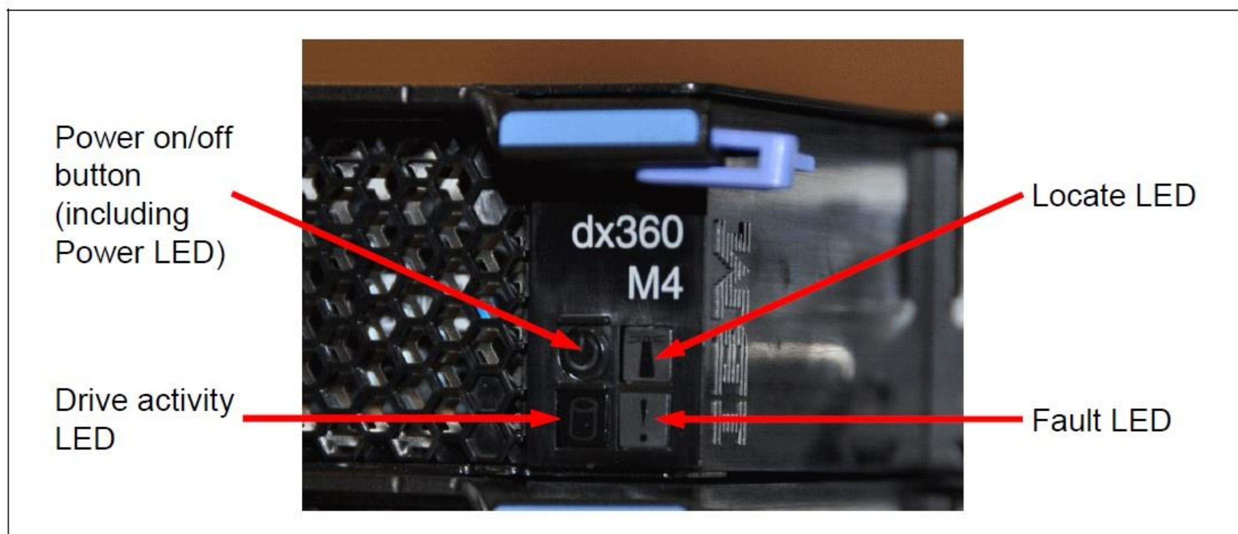


Figure 4-11 iDataPlex dx360 M4 operator panel

En la siguiente imagen se muestran dos dx360 M4 instalados en un chasis de 2 unidades:

Figure 4-2 Two air-cooled iDataPlex dx360 M4 servers installed in a 2U chassis

En la siguiente imagen se muestra como es un *rack* en el cual se instalan las *blades* dx360 M4:

Figure 1-1 iDataPlex 100U rack

En la siguiente imagen se muestra como es un *rack* donde se instalan las *blades* y su distribución:

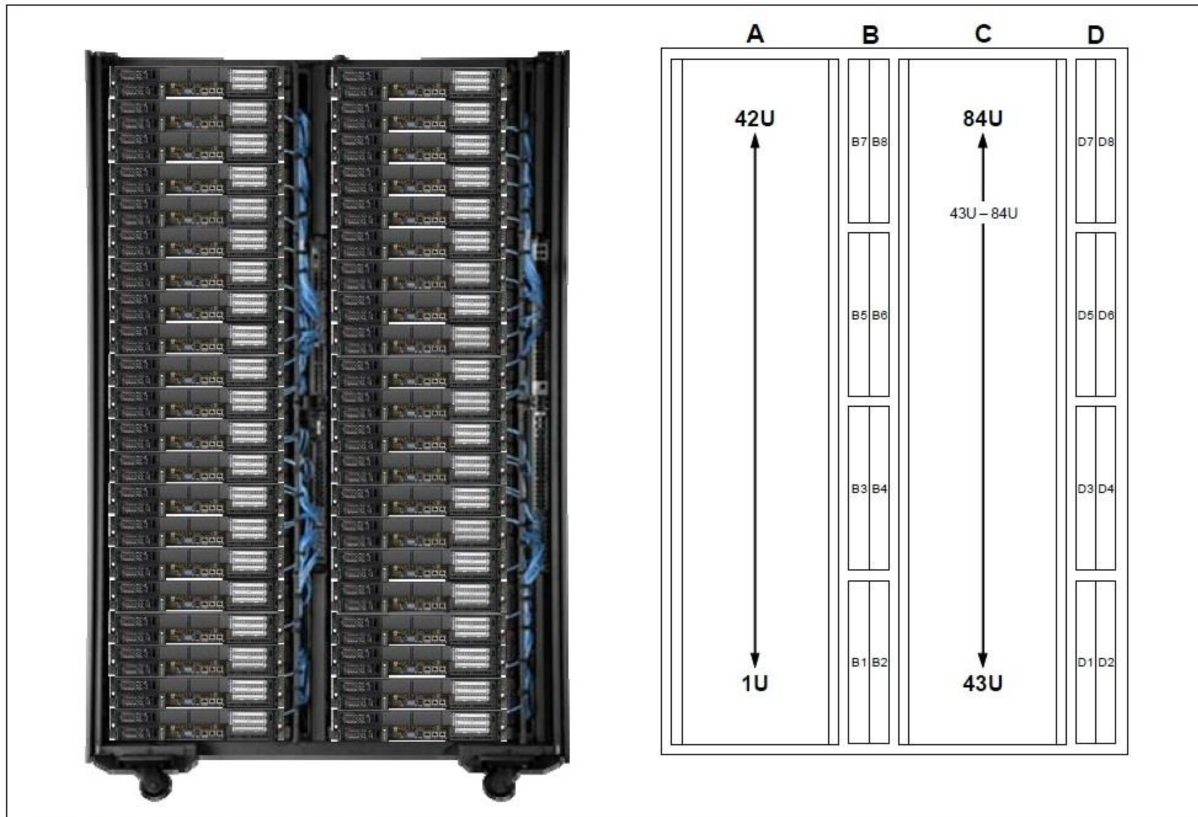


Figure 5-2 iDataPlex rack numbering scheme

Grid computing

Los requisitos de cómputo necesarios para ciertas aplicaciones son tan grandes que requieren miles de horas para poder ejecutarse en entornos de *clusters*. Tales aplicaciones han promovido la generación de ordenadores virtuales en red, *metacomputers* o *grid computers*. 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 manera y con excelentes resultados. Ejemplos de su aplicación son experimentos como el I-WAY networking (el cual conecta superordenadores de 17 sitios diferentes) en América del Norte, o DataGrid, CrossGrid en Europa o IrisGrid en España. Estos *metacomputers o grid computers* tienen mucho en común con los sistemas paralelos y distribuidos (SPD), pero también difieren en aspectos importantes. Si bien están conectados por redes, éstas 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 *metacomputing* 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 el *metacomputing* puede formarse con la base de los SPD, para éstos es necesario la creación de nuevas herramientas, mecanismos y técnicas.

La evolución en el tiempo del número de ordenadores y del número de redes indica la pauta de la siguiente arquitectura para computación de alto rendimiento:
Cómputo distribuido en Internet *o grid computing* (GC) *o metacomputing*.

Grid computing es una nueva tecnología emergente, cuyo objetivo es compartir recursos en Internet de manera uniforme, transparente, segura, 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), un *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.

Resumen y conclusiones

La computación de alto rendimiento es fundamental e imprescindible para la investigación y se lleva a cabo mediante supercomputadores (o superordenadores). Los supercomputadores actuales se basan en un conjunto de computadores de potencia similar a los que podríamos tener en casa pero con la diferencia de que estos computadores están interconectados por una red de altas prestaciones constituyendo un Clúster.
A la hora conseguir la mayor potencia de cómputo, sale más a cuenta económicamente interconectar un conjunto de computadores y utilizar el paso de mensajes entre procesos que tener una máquina con un procesador muy rápido (esta no es buena solución por culpa de las limitaciones tecnológicas) también sale más a cuenta que tener una Máquina de Memoria Compartida, es decir, muchos procesadores con muchos núcleos cada uno y con mucha memoria compartida entre todos los procesadores. En una MMC se pueden ejecutar procesos con muchísimos hilos de ejecución (threads) consiguiendo gran capacidad de cálculo gracias a la paralelización de memoria compartida pero hay que tener en cuenta que en una MMC solo se puede utilizar threads y tiene el problema de la escalabilidad, además es un hardware complejo por lo cual es caro, en cambio, interconectando un conjunto de ordenadores más sencillos y utilizando paso de mensajes entre procesos (Paralelismo de memoria distribuida) podemos conseguir una potencia de cálculo que económicamente no sería posible de otra manera, además no limita el uso de threads, aunque a nivel local, para poder aprovechar nodos multiprocesador y/o multinúcleo y/o tener concurrencia entre procesamiento y entrada/salida. Esta solución tiene el problema de la latencia en las comunicaciones pero gracias a los avances en este campo es la mejor solución, además tiene como punto a favor la escalabilidad y la relación rendimiento/coste.

Este trabajo es un estudio de soluciones *Clustering* actuales para la computación de alto rendimiento:

Se empieza estudiando el clúster Beowulf y la interfaz de paso de mensajes *Message Passing Interface*. Se estudia también *Parallel Virtual Machine* por ser tecnología anterior a MPI que existe y se puede utilizar pero se explica cómo introducción para posteriormente estudiar MPI siendo este el foco principal. Este tipo de soluciones *clustering* (MPI y PVM) requieren de la paralelización de programas utilizando librerías de paralelización o de paso de mensajes.

La paralelización de programas puede ser una tarea compleja, por este motivo también se estudia otro tipo de solución actual disponible para la computación de alto rendimiento: MOSIX, solución que consiste en la modificación del núcleo del sistema operativo para añadirle, a este, funcionalidades de *clustering*. Esta solución es transparente para el usuario y no requiere la paralelización de programas, ni utilizar ninguna biblioteca especial y es adecuada para la ejecución de tareas de computo intensivo con bajo/moderado uso de entrada/salida pero para la ejecución de tareas que hacen uso intensivo de entrada/salida no es una buena solución.

Finalmente se examina el superordenador Mare Nostrum III del Centro de Súper Computación de Barcelona. La guía de usuario del MN3 muestra como parte esencial el uso de MPI y OpenMP, entendiendo que la interfaz de paso de mensajes MPI y la programación multihilo (interfaz OpenMP) son esenciales a la hora de elaborar programas que se ejecutan en supercomputadores actuales y que el paso de mensajes entre procesos y la programación multihilo son esenciales a la hora de aprovechar los recursos del MN3.

Bibliografía y referencias

Bibliografía:

Administración avanzada de GNU/Linux , Autores: Josep Jorba Esteve, Remo Suppi Boldrito, UOC (Universitat Oberta de Catalunya).

Referencias:

http://www.debian.org/ Página oficial de Debian GNU/Linux

http://www.csm.ornl.gov/pvm/ Página oficial de PVM

http://www.mpich.org/ Página oficial de MPICH

http://www.mosix.cs.huji.ac.il/ Página oficial de MOSIX

http://www.bsc.es/ Página oficial del Centro de Súper Computación de Barcelona

http://www.globus.org/ Página oficial de Globus

http://www.ibm.com/es/es/ Página oficial de IBM

Apuntes de la Universidad de Valladolid

http://lsi.ugr.es/jmantas/pdp/teoria/teoria.php (visitado el 17/03/2013)

Apuntes de la Universidad de Granada de la asignatura “programación distribuida y paralela”

Glosario

Clúster: Conjunto de ordenadores en una LAN que trabajan con un objetivo común.

Grid: Agrupacion de ordenadores unidos por redes de área extensa (WAN). Clúster de clústers.

LAN: *Local Area Network*, red de área local.

WAN: *Wide Area Network*, red de área extensa.

HPC: *High Performance Computing*, Computación de alto rendimiento.

APD: Aplicaciones paralelas / distribuidas.

Clúster Beowulf: Es una arquitectura multiordenador que puede ser utilizada para aplicaciones paralelas/distribuidas (APD).

SSH: *Secure Shell*, shell segura.Protocolo de acceso seguro a la shell mediante la red.

NFS: *Network File System*, sistema de archivos en red.

RSA: Algoritmo de cifrado.

Nodo: Ordenador que forma parte del clúster.

Archivo tubería FIFO (*First In First Out*): Es un tipo de fichero especial que permite a procesos independientes comunicarse. Un proceso abre el fichero FIFO para escribir, y otro para leer.

Proceso: Programa en ejecución.

Thread: Hilo de ejecución, también llamado proceso ligero.

MPS: *Message Passing System*, Sistema donde se utiliza el paso de mensajes.

SHM: *Shared Memory Machine*, Máquina de memoria compartida.

RMI: Programación orientada a objetos distribuida JAVA.

PVM: *Parallel Virtual Machine*, máquina virtual paralela.

MPI: *Message Passing Interface*, interfaz de paso de mensajes.

MPICH2: Implementación del standard MPI 2.

MOSIX: *Multicomputer Operating System for Unix*, sistema operativo multicomputador para Unix.

Home-node: Nodo al cual pertenece el proceso MOSIX.

Launching-node: Nodo desde el cual se invocó la instrucción mosrun.

BSC: *Barcelona Super Computing Center*, Centro de Súper Computación de Barcelona.

RES: Red Española de Supercomputación.

MN3: Mare Nostrum III. Supercomputador del BSC.

GPFS: *General Parallel File System*, sistema de archivos de alto rendimiento que permite el acceso en paralelo a los datos.

LSF: Gestor de tareas utilizado en el supercomputador MN3.

Paralelismo de memoria distribuida: Paralelización de programas mediante el uso de paso de mensajes.

Paralelismo de memoria compartida: Paralelización de programas mediante el uso de la memoria compartida.

OpenMP: Interfaz para la programación multihilo.

Blade: Placa, los nodos de cómputo del MN3 son placas que están instaladas en *racks*.

Rack: Armario, estantería donde están instaladas las *blades.*