



**Ejecución de aplicaciones distribuidas en el cluster
del Centro de Apoyo Tecnológico**

Índice de contenido

Introducción.....	2
Descripción del sistema.....	3
Paralelización manual de un problema.....	4
Análisis del problema no paralelizado.....	4
Una solución distribuida.....	5
Ejecutando aplicaciones con el gestor de trabajos OpenPBS.....	6
Introducción:.....	6
Ejecución de programas desde línea de comando.....	6
Modificando el comportamiento del comando scasub.....	6
Consultando el estado de nuestros trabajos.....	7
Eliminando programas de la cola.....	7
Ejecución de programas desde el gestor scadestop.....	7
Aplicaciones distribuidas con MPI.....	8
Introducción:.....	8
Desarrollo de aplicaciones distribuidas con MPICH.....	8
Ejemplo sencillo de programación en C con MPICH.....	8
Compilación de programas Fortran con MPICH.....	9
Compilación de programas C++ con MPICH.....	9
Otro ejemplo de programación distribuida con MPICH.....	9
Un caso práctico: Cálculo distribuido de una integral numérica.....	9
Ejecución de programas MPICH de forma óptima.....	10
Ejecución de programas MPICH desde el gestor scadestop.....	10
Desarrollo de aplicaciones distribuidas con LAM-MPI.....	10
Ejemplo sencillo de programación en C con LAM-MPI.....	10
Compilación de programas Fortran con LAM-MPI.....	11
Compilación de programas C++ con LAM-MPI.....	11
El ejemplo mensajes_MPI.c, ejecutando con LAM-MPI.....	12
Cálculo distribuido de una integral numérica con LAM-MPI.....	12
Referencias:.....	13
Código fuente de los ejemplos empleados en esta guía:.....	14
Código de hello_mpi.c.....	14
Código de hello_mpi.f (Fortran).....	14
Código de hello_mpi.cc (C++).....	14
Código de mensajes_MPI.c.....	14
Código de integral_mpi.c.....	16

Introducción

La presente guía es una modesta presentación de "Hidra", el sistema multiprocesador adquirido por el Centro de Apoyo Tecnológico de la Universidad Rey Juan Carlos.

Su objetivo es dar a conocer parte de los programas instalados en el sistema y realizar una breve introducción al desarrollo de aplicaciones distribuidas, en las cuales un conjunto de ordenadores cooperan para resolver un problema. Veremos dos maneras diferentes de conseguir este tipo de resultados: mediante división del problema en partes independientes que pueden resolverse sin necesidad de desarrollar nuevos programas y mediante el desarrollo de aplicaciones distribuidas que se comunican mediante paso de mensajes con la librería MPI.

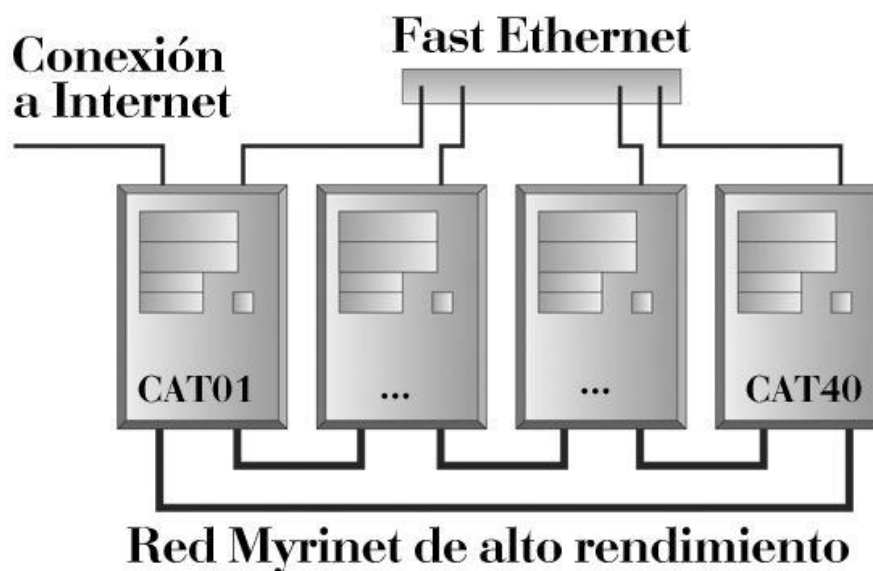
Por último, también debe ser este documento una invitación al uso del sistema por cualquier persona, así como una puesta a su disposición de los administradores del mismo, D. José Miguel Espadero (j.espadero@escet.urjc.es) y D. Carlos Gómez (cgomez@escet.urjc.es)

Descripción del sistema

Comenzaremos este documento dando una breve descripción del sistema, con el fin de poder sacar el máximo partido del mismo. El nombre escogido para el sistema ha sido **hidra**, en referencia a su naturaleza de procesamiento colectivo.

Hidra está formado por 40 ordenadores (o nodos) tipo PC, con las siguientes características:

- Procesador AMD Athlon XP a 2Ghz
- 512 Mb de memoria DDR-RAM
- Disco duro de 92 Gb. en RAID compartido por todos los nodos por NFS
- Disco duro de 40 Gb. independiente para cada nodo.
- Doble conexión de red interna, Myrinet y Fast Ethernet
- Sistema operativo RedHat Linux 7.3 recompilado con optimizaciones para Athlon
- Librerías de comunicación mediante paso de mensajes MPICH y LAM-MPI
- Gestor de trabajos OpenPBS



La principal característica que distingue a Hidra de otros sistemas similares es la doble red de comunicaciones: una red Myrinet de muy alto rendimiento (especialmente indicada para paso de mensajes entre aplicaciones distribuidas), y una red Fast-Ethernet (para el resto de tareas administrativas). Los nodos tienen un nombre dentro de la red Myrinet del tipo cat01, cat02, ... cat40 y otro nombre para la interfaz Fast-Ethernet del tipo hypercat01, hipercat02, ..., hypercat40. Además, el nodo cat01 responde al nombre **hidra.escet.urjc.es** para las conexiones al cluster desde Internet.

El directorio **/home/** es un RAID de discos de 92 Gb compartido por todos los nodos mediante NFS, por lo que no es necesario copiar los programas o ficheros de datos de un nodo a otro, la compartición de los directorios de los usuarios se realiza de forma automática y transparente entre todos los nodos.

La instalación del sistema operativo RedHat se ha hecho de forma casi completa, habiendo recompilado la mayoría de los programas y librerías para optimizar su ejecución en el procesador Athlon. Hidra dispone de los compiladores de GNU de C, C++ y Fortran77, así como el resto de herramientas de desarrollo y aplicaciones estándar de Unix: Latex, Octave, etc... En caso de necesitar la instalación de un programa, puede realizarla en su propio directorio de usuario o consultar con los administradores la posibilidad de realizar una instalación compartida.

Recuerde que Hidra es una máquina que permite que varios usuarios puedan estar trabajando de forma simultánea sin ningún problema, compartiendo los recursos de la máquina. Por favor, tenga en cuenta los siguientes normas:

- Evite sobrecargar algún nodo (especialmente cat01) enviando repetidamente ejecuciones de programas pesados. En caso de duda, lea el capítulo "Ejecución de aplicaciones bajo el gestor de trabajos OpenPBS" para saber como aprovechar los recursos de manera eficiente.
- Recuerde que es decisión suya el permitir o denegar el acceso a sus ficheros al resto de usuarios. Si quiere proteger su directorio de accesos indeseados ejecute el comando:

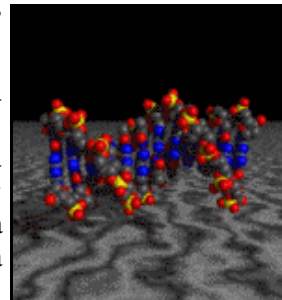
```
chmod -R go-rwx $HOME
```
- No utilice el disco del sistema para almacenar datos que no necesite, o copias de seguridad. Un uso racional del espacio en disco evitará la necesidad de activación de cuotas para los usuarios.

Por último, recuerde que los administradores del sistema pueden llegar a ser sus mejores amigos a la hora de sacar el mayor partido al mismo. No dude en consultarnos cualquier duda que le surja acerca del funcionamiento del mismo.

Paralelización manual de un problema

Este es un sencillo ejemplo de como aprovechar el cluster para resolver de forma paralela un problema sin necesidad de emplear programas especialmente diseñado para sistemas multiprocesador.

El problema que analizaremos y paralelizaremos será la generación de una animación que representa una cadena de ADN que gira sobre su propio eje. La animación consta de 120 fotogramas generados de manera **independiente**, lo cual permite la paralelización empleando técnicas de muy alto nivel. Cada fotograma se ha renderizado mediante el conocido programa [povray](#) y posteriormente se han combinado en un sólo fichero con la librería [ImageMagick](#).



El resultado puede verse en varios formatos: [\[gif\]](#) [\[minigif\]](#) [\[mng\]](#) [\[mpeg\]](#)

Análisis del problema no paralelizado

El script que genera los fotogramas y genera la animación es el siguiente:

```
#!/bin/sh

#Generamos 120 fotogramas tamaño 512x512
for ang in `seq -f "%03g" 1 3 360` ; do
  povray dna.ini +oimg$ang.png +k$ang
done

#Combinamos los fotogramas en el fichero adn.mpg
convert -quality 85 img*.png adn.mpg

#Eliminamos los ficheros temporales
rm img*.png
```

Para generar cada fotograma se ha escrito un fichero (dna.pov) describiendo la geometría de la molécula y otro (dna.ini) con los parámetros como el ancho y alto de la imagen. Estos ficheros se introducen como entrada del programa povray, así como un par de argumentos para indicar el nombre del fichero de salida (argumento +o) y el número de grados que giramos la molécula en el fotograma (argumento +k) . Por ejemplo, para generar el fotograma de la molécula rotada 27 grados, podemos usar:

```
povray dna.ini +oimg027.png -k27
```

Una vez generados todos los fotogramas, se realiza la combinación en un sólo fichero .mpg mediante el comando convert, indicando cual es la calidad del resultado final:

```
convert -quality 85 img*.png adn.mpg
```

El tiempo de ejecución del script anterior en un nodo del cluster es:

Tarea	Programa	Duración
Generación de los fotogramas	povray	9m24s
Compresión mpeg	convert	1m14s
Total		10m38s

Puede observarse que la mayor parte del tiempo se consume en la generación de los fotogramas, por lo que resulta interesante intentar acortar este proceso. Para ello, podemos tratar el problema de la generación de los fotogramas como 120 problemas independientes y dividir la resolución de los mismos entre varios nodos del cluster.

Una solución distribuida

La manera más sencilla de repartir las 120 ejecuciones del programa povray entre los nodos del cluster consiste en emplear un gestor de trabajos, que se encarga de repartir automáticamente las tareas entre todos los nodos del cluster. Un gestor de trabajos se encarga de buscar nodos del cluster que no se encuentren ejecutando ningún programa y encargarle la ejecución de un trabajo, de manera que la generación de los fotogramas puede realizarse de manera solapada en el tiempo. La descripción completa del gestor de trabajos instalado (Scali OpenPBS) se realizará en otra parte del documento, pero por ahora podemos adelantar que el comando "mágico" que realiza esta tarea se llama **scasub**.

Para ejecutar un comando en un nodo cualquiera del cluster, sin importar cual de ellos realiza la tarea, sólo hay que escribir:

```
scasub comando
```

Podemos emplear scasub para adaptar la primera solución para que use todos los nodos del cluster, por ejemplo:

```
#!/bin/sh
#Generamos 120 fotogramas tamaño 512x512
for ang in `seq -f "%03g" 1 3 360` ; do \
    scasub povray dna.ini +oimg$ang.png +k$ang ;\
done

#Espera a que aparezca el ultimo fotograma
while [ ! -f img357.png ] ; do
    sleep 3
done

#Combinamos los fotogramas en el fichero adn.mpg
convert -quality 85 img*.png adn.mpg

#Eliminamos los ficheros temporales
rm img*.png
```

El tiempo de ejecución del anterior script depende del número de nodos disponibles en el sistema, pero asegura que la respuesta se dará siempre en el menor tiempo posible al aprovechar al máximo todos los recursos del mismo. Una ejecución con los 40 nodos del cluster presentes tarda:

Tarea	Programa	Duración
Generación de los fotogramas	povray	0m15s
Compresión mpeg	convert	1m17s
Total		1m32s

Puede observarse que no merece la pena realizar nuevas optimizaciones sobre la generación de fotogramas, puesto que la mayor parte del tiempo lo consume ahora el comando de compresión mpeg. Como este comando realiza una sola ejecución, no es posible dividirlo en subproblemas empleando la misma técnica que hemos empleado para la primera parte del script.

Ejecutando aplicaciones con el gestor de trabajos OpenPBS

Introducción:

Un gestor de trabajos (o gestor de colas) es un sistema de distribución y ejecución de trabajos batch, es decir, programas que se ejecutan de modo desatendido, sin interactuar con el usuario. Su misión consiste en ejecutar trabajos aprovechando al máximo los recursos del cluster y garantizando el uso equitativo del sistema para todos los usuarios.

El gestor que se ha instalado en hidra es OpenPBS, y se encuentra integrado dentro del software Scali Universe. OpenPBS es capaz de ejecutar programas que corran en una sola máquina o en varias (mediante MPICH). En caso de que un usuario necesite ejecutar un programa, OpenPBS se ocupa automáticamente de buscar un nodo del cluster que no esté siendo utilizado, y realizar dicha ejecución en el nodo. De igual manera, en caso de querer ejecutar una aplicación distribuida, se buscará el número necesario de nodos para realizar la ejecución. En caso de que actualmente no pueda atenderse la ejecución de un programa por estar todos los nodos ocupados, OpenPBS introduce nuestra petición en una cola en espera de que se liberen los recursos necesarios.

A continuación, veremos como ejecutar programas realizando las peticiones a través del sistema de colas de OpenPBS.

Ejecución de programas desde línea de comando

El comando que debemos emplear para ejecutar aplicaciones en un nodo cualquiera del cluster es **scasub**. La manera más simple de ejecutar un programa sin importar el nodo que realice el trabajo es escribir simplemente:

```
scasub nombre_programa
```

Con lo cual el programa se ejecutará en el primer nodo libre del cluster o, en caso de que todos estén ocupados, se esperará a que alguno de ellos quede libre. Como respuesta a nuestro comando, scasub escribirá un número identificador del trabajo (Job ID o JID) y el nombre de la máquina que ejecuta el servidor (por ejemplo *76.cat01.escet.urjc.es*). Es importante recordar el número de nuestro trabajo para poder consultar el estado de nuestra petición.

Al terminar la ejecución de nuestro programa, scasub creará dos ficheros llamados **nombre_programa.o76** y **nombre_programa.e76** que contendrán el resultado de la salida estándar (stdout) y error estándar (stderr) respectivamente. El número en que termina el nombre del fichero (76) es el JID de nuestra petición.

Modificando el comportamiento del comando scasub

El comando scasub admite modificadores que permiten ajustar el comportamiento del mismo. La lista completa de opciones se puede consultar con el comando "man scasub", Siendo las siguientes las más comunes:

- -i nombre_fichero : Nombre del fichero que será usado como entrada estándar (stdin)
- -o nombre_fichero: Nombre del fichero donde se volcará la salida estándar (stdout) en lugar del nombre por defecto.
- -e nombre_fichero: Nombre del fichero donde se volcará la salida de error estándar (stderr) en lugar del nombre por defecto.
- -m email : Solicita que se envíe un aviso por correo electrónico cuando termine la ejecución de nuestro programa.

- -maxtime nn : Encarga al sistema que mate nuestro programa si su ejecución tarda más de nn minutos.
- -mpich : Indica que nuestro programa emplea la librería mpich de paso de mensajes, por lo que es necesario ejecutarlo mediante el lanzador mpirun.mpich (ver sección de aplicaciones MPI)
- -np num_nodos: En caso de que nuestra aplicación emplee la librería mpich, se indica el número de nodos que deben ejecutar nuestro programa. Usar sólo junto con el parámetro -mpich.
- -nodes nn : Solicita que se obligue la ejecución del trabajo en el nodo nn. También es posible indicar una lista de nodos separada por comas, en caso de ejecuciones con -mpich.

Consultando el estado de nuestros trabajos

Podemos emplear el comando **qstat** para solicitar un listado de los trabajos que están ejecutando o en espera de ser ejecutados. Por ejemplo, supongamos que ejecutamos la siguiente secuencia de comandos:

```
scasub ./programa1
scasub -mpich .np 10 ./programa2
scasub -mpich -np 30 ./programa3
qstat
```

Y el resultado del comando qstat es el siguiente:

Job id	Name	User	Time Use	S	Queue
104.cat01	programa2	jespa	0	R	scali_exec
105.cat01	programa3	jespa	0	Q	scali_exec

La salida del comando indica que programa2 tiene el JID 104 y se encuentra en ejecución (**R**), mientras que el programa3 está encolado (**Q**) en espera de que 30 nodos del cluster queden libres. El programa1 no aparece en el listado porque ya ha terminado su ejecución. Puede encontrar más información sobre el comando qstat mediante su página del manual (man qstat)

Eliminando programas de la cola

En caso de que deseemos eliminar un encargo de la cola de trabajos podemos usar el comando **qdel**, indicando el número del trabajo que deseamos eliminar. Por ejemplo, para eliminar el trabajo 105 del listado anterior:

```
qdel 105
```

Ejecución de programas desde el gestor scadestop

Quien prefiera realizar las ejecuciones de sus aplicaciones desde el entorno gráfico **scadesktop**, deberá realizar los siguientes pasos:

1. Ejecute el programa scadesktop y seleccione el cluster que desea gestionar.
2. Active la acción de menú : Run->'Submit Job'
3. Escriba la ruta completa del ejecutable de la aplicación en el campo 'Command Line'
4. Si su aplicación usa la librería MPICH, indique el número de nodos que necesita en el campo 'Number of Nodes' y añada el modificador **-mpich** en el campo 'Command Line', antes del nombre de su programa.
5. Pulse el botón 'Submit'

Aplicaciones distribuidas con MPI

Introducción:

El presente documento pretende realizar una breve introducción a la librería MPI y guiar al usuario en la compilación y ejecución de programas paralelos que empleen dicha librería. MPI es una especificación estándar de una librería que permite comunicar varios procesos mediante paso de mensajes, independientemente de si los procesos ejecutan en la misma máquina o no.

Hidra contiene dos implementaciones del estándar MPI:

- LAM-MPI es la implementación estándar que puede encontrarse en todas las distribuciones de GNU/Linux, lo cual permite ejecutar en hidra aplicaciones paralelas que hayan sido probadas anteriormente en otras configuraciones. Es posible encontrar documentación y futuras actualizaciones de la librería en la página <http://www.lam-mpi.org/>
- MPICH es una implementación de MPI optimizada para entornos homogéneos y myrinet, lo que proporciona un mayor rendimiento en el paso de mensajes entre nodos. La página del proyecto MPICH es <http://www-unix.mcs.anl.gov/mpi/mpich/>

De las dos implementaciones disponibles, la más recomendada es MPICH, tanto por su rendimiento (muy superior a LAM) como por facilidad de uso. La razón por la que se ha decidido dejar ambas implementaciones disponibles es porque LAM está más extendida, por lo que es posible encontrar alguna aplicación que haga uso forzoso de ella.

A continuación, veremos por separado como ejecutar programas con una u otra implementación de MPI. Hay que destacar, que no es necesario modificar el código fuente de los programas para adaptarlo a cada una, aunque el proceso de compilación y ejecución difiere entre las mismas.

Desarrollo de aplicaciones distribuidas con MPICH

A continuación veremos varios ejemplos sencillos de desarrollo de aplicaciones distribuidas con MPICH.

Ejemplo sencillo de programación en C con MPICH

El primer ejemplo que compilaremos es la versión distribuida del famoso "Hola Mundo", que puede encontrarse al final de este documento con el nombre de `hello_mpi.c`. Para ejecutar el programa hay que seguir los siguientes pasos:

1. Compilar el fichero `hello_mpi.c` empleando el comando `/opt/scali/contrib/mpich/bin/mpicc` (en lugar del típico `cc`), o bien `mpicc.mpich`

```
mpicc.mpich hello_mpi.c -o hello_mpi
```

2. Ejecutar nuestro programa con el comando `/opt/scali/contrib/mpich/bin/mpirun`, o bien `mpirun.mpich`. Podemos indicar cual es el número de procesos que hay que lanzar con la opción `-np`, ocupándose MPICH de repartir los procesos entre los nodos del cluster. Por ejemplo, para lanzar 4 procesos:

```
mpirun.mpich -np 4 ./hello_mpi
```

La salida del programa debe ser algo parecido a esto:

```
Soy el nodo 0 de un cluster de 4
Soy el nodo 1 de un cluster de 4
Soy el nodo 2 de un cluster de 4
Soy el nodo 3 de un cluster de 4
```

Compilación de programas Fortran con MPICH

En caso de que nuestro programa esté escrito en Fortrán, el único cambio necesario es cambiar el nombre del compilador para emplear `/opt/scali/contrib/mpich/bin/mpif77`, o bien `mpif77.mpich`, como puede verse en el siguiente ejemplo:

```
mpif77.mpich hello_mpi.f -o hello_mpi
mpirun.mpich -np 4 ./hello_mpi
```

La salida del programa debe ser similar a la del ejemplo anterior.

Compilación de programas C++ con MPICH

Cuando nuestra aplicación se halla realizado en lenguaje C++ debemos emplear `/opt/scali/contrib/mpich/bin/mpiCC`, o bien `mpiCC.mpich`, como puede verse en el siguiente ejemplo:

```
mpiCC.mpich hello_mpi.cc -o hello_mpi
mpirun.mpich -np 4 ./hello_mpi
```

Otro ejemplo de programación distribuida con MPICH

El segundo ejemplo presentado sirve para comprobar la comunicación entre procesos mediante paso de mensajes bloqueantes mediante los comandos `MPI_Send()` y `MPI_Recv`. El código `mensajes_MPI.c` contiene un programa en el cual los nodos forman una cadena circular en la que se van comunicando un número entero. El primer nodo del cluster decrementa el número cada vez que pasa por sus manos, y todo el proceso se repite mientras el número pasado sea mayor que cero.

Para ejecutar el ejemplo `mensajes_MPI.c` es necesario escribir:

```
mpicc.mpich mensajes_mpi.c -o mensajes_mpi
mpirun.mpich -np 4 ./mensajes_mpi
```

Un caso práctico: Cálculo distribuido de una integral numérica

A continuación presentamos un ejemplo práctico de programación distribuida, que calcula el resultado de una integral aprovechando el número de nodos del cluster que tengamos a nuestra disposición. La solución propuesta en el ejemplo `integral_mpi.c`, que calcula la integral de la función $\text{seno}(x)$ dividiendo el intervalo de integración entre todos los nodos, de forma que cuantos más nodos tengamos más rápida será la respuesta.

La compilación y ejecución del programa es idéntica a los ejemplos anteriores, salvo que necesita la librería matemática (flag `-lm`) para poder evaluar la función `seno`. Podemos ejecutar el programa con el comando `time` para verificar la diferencia de tiempos al variar el número de nodos.

Para ejecutar la aplicación empleando MPICH:

```
mpicc.mpich integral_mpi.c -o integral_mpi -lm
time mpirun.mpich -np 1 ./integral_mpi
time mpirun.mpich -np 4 ./integral_mpi
time mpirun.mpich -np 10 ./integral_mpi
```

Ejecución de programas MPICH de forma óptima con OpenPBS

Aunque en los ejemplos anteriores se ha empleado el comando `mpirun.mpich` para la ejecución de aplicaciones paralelas con MPICH, no es la manera más recomendable, sino que es mejor realizar las ejecuciones a través del gestor de trabajos OpenPBS. La razón por la cual no se recomienda emplear directamente `mpirun.mpich` es que este lanzador usa siempre los nodos 01 a nn para lanzar sus procesos, por lo que la ejecución de varias aplicaciones paralelas competirán por el uso de los nodos con numeración más baja del cluster. Para evitar esto, se recomienda emplear el comando **scasub con la opción -mpich**, lo que permite optimizar el uso de los recursos entre todos los usuarios. Así, para ejecutar la aplicación anterior, se recomienda usar:

```
mpicc.mpich integral_mpi.c -o integral_mpi -lm
scasub -mpich -np 1 ./integral_mpi
scasub -mpich -np 4 ./integral_mpi
scasub -mpich -np 10 ./integral_mpi
```

Ejecución de programas MPICH desde el gestor scadestop

Quien prefiera realizar las ejecuciones de sus aplicaciones desde el entorno gráfico **scadesktop**, deberá realizar los siguientes pasos:

1. Primero debe compilar su programa empleando las librerías MPICH tal como se ha explicado en los apartados anteriores, es decir, mediante los compiladores `mpicc.mpich`, `mpiCC.mpich` o `mpif77.mpich`, según el lenguaje en que se haya escrito la aplicación.
2. Ejecute el programa `scadesktop` y seleccione el cluster deseado (por ejemplo "Hidra") haciendo doble click sobre el icono del mismo.
3. Introduzca su nombre de usuario y contraseña para acceder al cluster
4. Seleccione los nodos sobre los que desea ejecutar la aplicación. Recuerde que puede pulsar la tecla control para añadir nuevos nodos a la selección actual.
5. Ejecute la acción de menú : Run->'MPICH program'
6. Escriba la ruta completa del ejecutable de la aplicación en el campo 'MPI program' y pulse el botón 'Run program'. Recuerde que la aplicación se ejecutará sobre los nodos seleccionados en el paso 4.

Desarrollo de aplicaciones distribuidas con LAM-MPI

A diferencia de MPICH, la ejecución con el sistema LAM-MPI necesita la inicialización de un entorno de ejecución por parte del usuario, lo que complica algo más su uso. Si por alguna razón se prefiriese emplear LAM-MPI, estos son los pasos que hay que seguir para compilar los ejemplos anteriores.

Ejemplo sencillo de programación en C con LAM-MPI

Para compilar y ejecutar el programa `hello_mpi.c` hay que seguir los siguientes pasos:

1. Compilar el fichero `hello_mpi.c` empleando el comando **mpicc** (en lugar del típico `cc`)

```
mpicc hello_mpi.c -o hello_mpi
```

2. Escribir un fichero de configuración con el nombre de los nodos que queremos que ejecuten el programa. Por ejemplo, para lanzarlo en los nodos `cat01` a `cat04` escribiríamos en el fichero **nodos_1-4.txt**:

```
cat01
cat02
cat03
cat04
```

3. Ejecutar el comando **lamboot** con el fichero de configuración anterior, de forma que se crea la infraestructura de comunicación entre los nodos.

```
lamboot nodos_1-4.txt
```

4. Ejecutar nuestro programa con el comando **mpirun**. Podemos indicar cual es el número de procesos que hay que lanzar con la opción **-np**, ocupándose MPI de repartir los procesos entre los nodos indicados mediante el comando lamboot. Por ejemplo, para lanzar 4 procesos:

```
mpirun -np 4 ./hello_mpi
```

La salida del programa debe ser algo parecido a esto:

```
Soy el nodo 0 de un cluster de 4
Soy el nodo 1 de un cluster de 4
Soy el nodo 2 de un cluster de 4
Soy el nodo 3 de un cluster de 4
```

5. Una vez que hayamos terminado de ejecutar programas distribuidos debemos destruir la infraestructura creada empleando el comando **wipe** con el mismo fichero de configuración empleado en el comando lamboot:

```
wipe nodos_1-4.txt
```

NOTA: Es muy **IMPORTANTE** realizar el paso 5 una vez terminada la ejecución de programas, puesto que de lo contrario los nodos quedan marcados como ocupados para futuras configuraciones de LAM-MPI.

Compilación de programas Fortran con LAM-MPI

En caso de que nuestro programa esté escrito en Fortran en lugar de en C, el único cambio necesario para la compilación y ejecución del programa consiste en emplear el compilador **mpif77** en lugar del mpicc. Por ejemplo, para compilar y ejecutar el programa hello_mpi.f debemos tener preparado el fichero nodos_1-4.txt como en el ejemplo anterior y emplear los siguientes comandos:

```
mpif77 -I/usr/include hello_mpi.f -o hello_mpi
lamboot nodos_1-4.txt
mpirun -np 4 ./hello_mpi
wipe nodos_1-4.txt
```

Nota: Es importante incluir la opción **-I/usr/include** al compilador para que encuentre los ficheros de cabecera necesarios para compilar con MPI

Compilación de programas C++ con LAM-MPI

En caso de que nuestro programa esté escrito C++, debemos emplear el compilador **mpiCC**:

```
mpiCC hello_mpi.cc -o hello_mpi
lamboot nodos_1-4.txt
mpirun -np 4 ./hello_mpi
wipe nodos_1-4.txt
```

El ejemplo mensajes_MPI.c, ejecutando con LAM-MPI

Para compilar y ejecutar el ejemplo mensajes_MPI.c, deben ejecutarse los siguientes comandos.

```
mpicc mensajes_mpi.c -o mensajes_mpi
lamboot nodos_1-4.txt
mpirun -np 4 ./mensajes_mpi
wipe nodos_1-4.txt
```

Nota: Es posible emplear este ejemplo para comprobar la diferencia de eficiencia en la comunicación de mensajes entre LAM y MPICH. Para ello, modifique el fuente para aumentar el número de mensajes (por ejemplo, un millón) y mida el tiempo de ejecución entre ambas implementaciones.

Cálculo distribuido de una integral numérica con LAM-MPI

La compilación y ejecución del programa integra_mpi.c es idéntica a los ejemplos anteriores, salvo que necesita la librería matemática (flag -lm).

```
mpicc integral_mpi.c -o integral_mpi -lm
lamboot nodos_1-10.txt
time mpirun -np 1 ./integral_mpi
time mpirun -np 4 ./integral_mpi
time mpirun -np 10 ./integral_mpi
wipe nodos_1-10.txt
```

Referencias:

A continuación le indicamos algunas referencias generales que pueden servir de complemento al contenido de la presente guía.

- Curso de Introducción a Linux : <http://cila.gulic.org/libro.php>
- The Advanced Bash-Scripting Guide : <http://en.tldp.org/LDP/abs/html/>
- The povray documentation page : <http://www.povray.org/documentation/>
- Página oficial de OpenPBS: <http://www.openPBS.org/>
- Página de documentación de Scali: <http://www.scali.com/index.php?content=170>
- Tutorial de MPI en castellano: <tutmpi.pdf>
- Página oficial del estándar MPI: <http://www-unix.mcs.anl.gov/mpi/>
- Página del proyecto LAM-MPI: <http://www.lam-mpi.org/>
- Página del proyecto MPICH: <http://www-unix.mcs.anl.gov/mpi/mpich/>

Código fuente de los ejemplos empleados en esta guía:

Código de hello_mpi.c

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

int main (int argc, char **argv)
{
    int rank;
    int size;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf ("Soy el nodo %d de un cluster de %d\n", rank, size);
    MPI_Finalize ();
    return 0;
}
```

Código de hello_mpi.f (Fortran)

```
program hello

    implicit none
    include 'mpif.h'
    integer rank,size,ierr

    call mpi_init(ierr);
    call mpi_comm_rank(MPI_COMM_WORLD,rank,ierr);
    call mpi_comm_size(MPI_COMM_WORLD,size,ierr);
    write (*,'(A,I3,A,I3)') 'Soy el nodo ',rank,' de ',size
    call mpi_finalize(ierr);

end
```

Código de hello_mpi.cc (C++)

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

int main(int argc, char *argv[])
{
    MPI::Init(argc, argv);

    int rank = MPI::COMM_WORLD.Get_rank();
    int size = MPI::COMM_WORLD.Get_size();

    cout << "Soy el nodo " << rank << " de " << size << endl;

    MPI::Finalize();
}
```

Código de mensajes_MPI.c

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

int main(int argc, char *argv[])
{
    MPI_Status status;
    int num, rank, size, tag, next, from;

    /* Start up MPI */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    /* Arbitrarily choose 201 to be our tag. Calculate the */
    /* rank of the next process in the ring. Use the modulus */
    /* operator so that the last process "wraps around" to rank */
    /* zero. */
    tag = 201;
    next = (rank + 1) % size;
    from = (rank + size - 1) % size;

    /* If we are the "console" process, get a integer from the */
    /* user to specify how many times we want to go around the */
    /* ring */
    if (rank == 0) {
        if (argc>1)
            num = atoi (argv[1]);
        else
            num = 5;

        printf("Process %d sending %d to %d\n", rank, num, next);
        MPI_Send(&num, 1, MPI_INT, next, tag, MPI_COMM_WORLD);
    }

    /* Pass the message around the ring. The exit mechanism works */
    /* as follows: the message (a positive integer) is passed */
    /* around the ring. When it passes rank 0, it is decremented. */
    /* When each processes receives the 0 message, it passes it on */
    /* to the next process and then quits. By passing the 0 first, */
    /* every process gets the 0 message and can quit normally. */
    while (1) {

        MPI_Recv(&num, 1, MPI_INT, from, tag, MPI_COMM_WORLD, &status);
        printf("Process %d received %d\n", rank, num);

        if (rank == 0) {
            num--;
            printf("Process 0 decremented num\n");
        }

        printf("Process %d sending %d to %d\n", rank, num, next);
        MPI_Send(&num, 1, MPI_INT, next, tag, MPI_COMM_WORLD);

        if (num == 0) {
            printf("Process %d exiting\n", rank);
            break;
        }
    }

    /* The last process does one extra send to process 0, which needs */
    /* to be received before the program can exit */
    if (rank == 0)
        MPI_Recv(&num, 1, MPI_INT, from, tag, MPI_COMM_WORLD, &status);

    /* Quit */
    MPI_Finalize();
    return 0;
}
```

Código de integral_mpi.c

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

int main (int argc, char **argv)
{
    MPI_Status status;
    int rank, size;
    double ancho, a, b, x, acum;
    double delta = 0.00000001;
    int num_iter;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    /* Dividimos el intervalo [0 .. PI) entre todos los procesos, */
    /* Calculamos los límites inferior y superior de este proceso */
    ancho = M_PI / size;
    a = rank * ancho;
    b = (rank + 1) * ancho;

    /* Mensaje informativo */
    printf ("nodo %d calculando intervalo [%f .. %f)\n", rank, a, b);

    /* Calculamos la integral de la función entre a y b */
    /* Empleamos el método de Riemann con intervalo delta*/
    acum = 0;
    for (x = a; x < b; x += delta )
        acum += delta * sin (x);

    /* Todos los procesos envían su resultado al proceso 0, que
     * realiza la acumulación y escribe su valor en pantalla */
    if (rank > 0)
    {
        MPI_Send (&acum, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
    }
    else
    {
        int i;
        double data;
        /* El nodo 0 recibe todas las respuesta y acumula */
        for (i = 1; i < size; i++)
        {
            MPI_Recv (&data, 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD, &status);
            acum += data;
        }

        /* Imprimimos el resultado total */
        printf ("El resultado de la integral entre 0 y PI es: %f\n", acum);
    }

    MPI_Finalize ();
    return 0;
}
```