Prácticas COS – Curso 2022-23 – Sesión 3

Ejecución de aplicaciones mediante Condor

Trabajaremos con el ordenador del laboratorio, arrancando en Linux. Tras iniciar sesión en la máquina del seminario, iniciaremos sesión remota ssh en el clúster *cac*:

```
ssh cosxx@cac1
Password: xx202223cos
```

Copiaremos los archivos de la práctica a una carpeta de nuestro directorio *home* y nos situaremos en dicha carpeta:

```
cp –r /tmp/practicasCOS/condor-files $HOME/prac3-condor
cd $HOME/prac3-condor
```

Condor es un planificador de trabajos o tareas *(Job Scheduler)* diseñado para aprovechar al máximo la potencia de cálculo de un conjunto de nodos de cómputo que forman un clúster. Hay disponible una versión de dominio público (http://www.cs.wisc.edu/condor) que es la que tenemos configurada e instalada en el clúster *cac*. Dispone también de una versión de pago para empresas. Hay otros planificadores de código abierto como por ejemplo OpenPBS (Portable Batch System, http://www.openpbs.org)

Condor ofrece un mecanismo de encolado de trabajos, permite establecer la política de planificación, establecer prioridades y manejar y monitorizar todos los recursos disponibles en el clúster. Los usuarios envían sus trabajos a *Condor* y este los mete en una cola, elige cuando y donde ejecutarlos en función de la política de planificación y de las prioridades asignadas, monitoriza su progreso e informa al usuario de su finalización.

Puede manejar desde trabajos secuenciales a trabajos paralelos MPI o PVM. En esta práctica vamos a usar *Condor* como planificador de trabajos secuenciales y desde el punto de vista de un usuario, no de un administrador.

Los pasos a seguir cuando queremos enviar un trabajo a *Condor* son los siguientes:

1) **Preparar el trabajo para que se ejecute sin intervención del usuario (batch job).** El programa ejecutado por *Condor* no es capaz de realizar entradas/salidas interactivas. Hay que crear un fichero que contendrá los datos de entrada necesarios e indicarle en que ficheros queremos que almacene los resultados y los posibles mensajes de error.
2) **Seleccionar el universo Condor.** Dispone de cinco entornos de ejecución. Cada entorno de ejecución de llama *Universo*. Hay dos universos para trabajos serie *(Standard y Vanilla)*, uno para trabajos paralelos *MPI*, uno para trabajos paralelos *PVM* y uno para aplicaciones *Grid*.
3) **Crear fichero de descripción para el envío del trabajo.** Este fichero contiene información sobre el nombre y la ubicación del fichero que queremos ejecutar, que universo usar, los ficheros a usar como *stdin, stdout, stderr*, preferencias sobre el/los nodos que debería/n ejecutar el programa y a qué dirección de correo electrónico enviar un email cuando finalice la ejecución. Se puede indicar cuantas veces queremos ejecutar un programa y elegir diferentes datos de entrada para cada ejecución.

4) **Enviar el trabajo.** Enviar el trabajo a *Condor* con el comando *condor_submit*.

Una vez enviado *Condor* se encarga de su ejecución. El usuario puede ver el progreso de la ejecución con los comandos *condor_q y condor_status*. También puede cambiar la prioridad asignada al trabajo, eliminarlo de la cola, etc.

Vamos a ver a continuación un sencillo fichero de descripción de trabajo.

Este fichero encola dos copias de un programa llamado *matematica*. La primera copia se ejecuta en el directorio *run_1* y la segunda en el directorio *run_2*. Para ambas copia el fichero de entrada *stdin* será *test.data, stdout* será *salida.out* y *stderr* será *salida.error*. Cada uno de estos ficheros aparecerá en los dos directorios de trabajo.

```
# Ejemplo de fichero de descripción de trabajo
# Usa varios directorios para organizar los datos

universe = vanilla
executable = matematica

# Podemos introducir algunos argumentos en la línea de órdenes

arguments = -solver matrix

input = test.data
output = salida.out
stderr = salida.error
log = salida.log

initialdir = run_1
queue
initialdir = run_2
queue
```

El universo elegido es *vanilla*. Si no se especifica el universo por defecto es *standard*. Ambos universos permiten ejecutar trabajos secuenciales (no paralelos). El universo *standard* impone algunas restricciones al tipo de trabajos soportados y *vanilla* no impone ninguna restricción. Con *queue* se introduce el trabajo en la cola.

Tras esta pequeña introducción vamos a aprender cómo se lanzar programas con diferentes configuraciones de datos de entrada.

1ª parte.

En primer lugar, vamos a ejecutar múltiples instancias del conocido benchmark *dhrystone* mediante *Condor*.

- Los archivos están ubicados en la carpeta `dhrystone`. El ejecutable se generaría mediante la orden:

```
gcc dhry_1.c dhry_2.c –o dhry
```

  Sin embargo, el ejecutable está ya disponible en las carpetas correspondientes.

  Tenemos tres archivos de descripción de trabajos:

  - Lanzamiento de un proceso.
    Carpeta `basic`, archivo de descripción de trabajo `dhry.cmd`, archivo de entrada `dhry.in`.
    Analizar el contenido de los archivos y encolar para su ejecución mediante la orden:

```
condor_submit dhry.cmd
```

  - Lanzamiento de varios procesos con datos en múltiples directorios.
    Carpeta `mult-dir`, archivo de descripción de trabajo `dhryN.cmd`, archivos de entrada `dhry.in` ubicados en dos carpetas `run_1 y run_2`.
    Analizar el contenido de los archivos y encolar para su ejecución mediante mediante la orden:

```
condor_submit dhryN.cmd
```

  - Lanzamiento de varios procesos con múltiples ficheros de datos.
    Carpeta `nn`, archivo de descripción de trabajo `dhrynn.cmd`, archivos de entrada `in.0` a `in.4`.
    Analizar el contenido de los archivos y encolar para ejecución mediante la orden:

```
condor_submit dhrynn.cmd
```

  Cada una de las ejecuciones de un programa tiene asociado un número proceso, comenzando con el número proceso 0. En este último ejemplo podéis observar el uso de la macro *$(Process)* que *Condor* expande con el número de proceso asociado a esa ejecución del programa. Os será de utilidad en apartados posteriores.

- En cada uno de los casos, comprobaremos el estado de los trabajos en el sistema:

```
condor_status –submitters
condor_status –run
condor_q
condor_q 132.0 -analyze
```

  donde 132.0 sería el trabajo del que solicitamos información. Debéis usar el número que *Condor* asigna a vuestro trabajo.

- Cambiar la prioridad de algún trabajo. Por ejemplo:

```
condor_prio –p –15 132.0
```

- Borrar algún trabajo. Por ejemplo:

```
condor_rm 132.0
```

- Obtener el histórico de trabajos:

```
condor_history
```

- Analizar el resultado obtenido en los archivos `.log` y `.out`

2ª parte.

En el directorio `dlx` tenemos disponible un simulador del procesador DLX, tal y como lo define Hennessy y Patterson en su libro de texto. El programa acepta varios parámetros de configuración (relacionados con la salida de resultados y el diseño concreto del procesador a simular), así como el archivo que contiene el programa DLX a ejecutar. Disponemos de los programas *ordena.s* y *ordena3.s* escritos en ensamblador del procesador DLX.

- Escribir el archivo de descripción de trabajos necesario para simular un programa de ordenación de un vector, con diversos parámetros de configuración:

```
dlx -s final -d p -c p -f ordena.s
dlx -s final -d c -c p -f ordena.s
dlx -s final -d p -c t -f ordena.s
dlx -s final -d c -c t -f ordena.s
dlx -s final -d p -c 3 -f ordena3.s
dlx -s final -d c -c 3 -f ordena3.s
```

- Ejecutar los trabajos mediante *Condor* y comprobar en los resultados obtenidos que el vector queda ordenado en la memoria.

3ª parte.

En el directorio `dagman` tenemos disponible un programa para calcular integrales definidas, en este caso de la función sin(x) (`intsin.c`). Para compilarlo, utilizamos la orden:

```
gcc intsin.c -lm -o intsin
```

El programa acepta tres parámetros: límite inicial, límite final y número de intervalos. Por ejemplo:

```
intsin 0 6.28 1000
```

También se pueden pasar los parámetros por la entrada estándar.

Deseamos repartir el cálculo entre todos los nodos que *Condor* tiene disponibles. Para ello, repartiremos el intervalo a integrar en subintervalos (50 por ejemplo) y lanzaremos el cálculo de cada subintervalo a ejecución en *Condor*. *Condor* lanzará a ejecución 50 copias del programa *intsin* distribuidas entre todos los nodos que tiene disponibles y por último, una vez finalizadas todas las ejecuciones, sumaremos todos los resultados.

Para ello, disponemos del programa (`prep.c`) que, a partir de los límites inferior y superior, número de intervalos y número de procesos, genera los ficheros de entrada para `intsin`. Por ejemplo, la orden:

```
prep 0 6.28 1000 50 in
```

genera 50 ficheros que comienzan por "in", entre los que se habrá repartido el intervalo a integrar.

Analizar el contenido del fichero *prep.c*

Por otra parte, el programa `sum.c` suma el contenido (numérico) de los archivos que le pasamos como parámetro. Por ejemplo, si hacemos:

```
sum 50 out
```

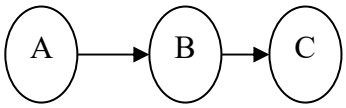
obtendrá la suma del contenido de los 50 ficheros que comienzan por *out*.

Analizar el contenido del fichero *sum.c*.

Se pide:

- En primer lugar, escribir un archivo de descripción de trabajo *Condor* (`intsin.condor`) para ejecutar el programa de integración numérica `intsin` para 50 intervalos. Los archivos de entrada los generaremos directamente mediante el programa `prep`, sin utilizar *Condor*.
  Una vez tengamos el archivo de descripción de trabajo, debemos encolarlo para su ejecución. Cuando todos los procesos hayan terminado, utilizaremos el programa `sum` (sin utilizar *Condor*) para sumar los resultados parciales, comprobando que el resultado es el esperado.

- Otra opción es utilizar el planificador *Dagman* de *Condor*, que sirve para lanzar trabajos con dependencias. En este caso, pretendemos ejecutar una carga representada por el siguiente grafo de dependencias:



  Siendo:

  - "A" el programa `prep` que genera los ficheros de entrada para el programa "B.
  - "B" el programa de integración numérica cuya función a integrar es sin(x) (`intsin.c`)
  - "C" el programa que suma el contenido (numérico) de los archivos generados por "B".

  Los límites de integración, el número de intervalos y número de procesos se almacenarán en las variables de entorno LI, LS, ITER y JOBS, respectivamente. Por ejemplo:

```
export LI=0
export LS=6.28
export ITER=1000
export JOBS=50
```

  Escribir un fichero de descripción de trabajo para ejecutarlo con el planificador `Dagman`. Bucar información en internet sobre la estructura que debe tener este fichero.

  Los ficheros `prep.condor` y `sum.condor` ya contienen la descripción de trabajo para ejecutar los programas *prep* y *sum*, respectivamente. El fichero `intsin.condor` es el que hemos escrito con la descripción de trabajo para ejecutar el programa `intsin`.

  Una vez preparado el archivo `dagman`, lo ejecutaríamos con el planificador `dagman`, mediante la orden:

```
condor_submit_dag myjob.dagman
```

NOTAS:
-El valor de una variable de entorno se puede leer en el fichero de descripción de Condor mediante `$ENV(VARIABLE)`.

-Para que el programa `sum` pueda abrir los archivos cuyo nombre le pasamos como parámetro, teniendo en cuenta que todas las máquinas comparten un disco mediante NFS, deberíamos especificar la ruta absoluta para los nombres de los ficheros. Por ejemplo:

```
sum 5 $ENV(PWD)/out
```

Resumen de las órdenes de usuario de Condor

Command	Description
condor_checkpoint	Checkpoint jobs running on the specified hosts
condor_compile	Create a relinked executable for submission to the Standard Universe
condor_history	View log of Condor jobs completed to date
condor_hold	Put jobs in the queue in hold state
condor_prio	Change priority of jobs in the queue
condor_qedit	Modify attributes of a previously submitted job
condor_q	Display information about jobs in the queue
condor_release	Release held jobs in the queue
condor_reschedule	Update scheduling information to the central manager
condor_rm	Remove jobs from the queue
condor_run	Submit a shell command-line as Condor job
condor-status	Display status of the Condor pool
condor_submit_dag	Manage and queue jobs within a specified DAG for interjob dependences
condor_submit	Queue jobs for execution
condor_userlog	Display and summarize job statistics from job log files