Universidad Nacional de General Sarmiento

CLUSTER DE COMPUTADORAS

Una breve introducción

LIC. MARIANO VARGAS
2023

Contenido

Introducción a Clúster de computadoras:.....2
SOFTWARE (Cronos).....2
Munge.....2
Introducción a Slurm:.....3
Comandos básicos de Slurm.....3
srun - programar comandos.....4
squeue - ver trabajos programados/encolados.....4
scancel - cancelar un trabajo encolado.....5
sbatch - programar un batch script.....5
Open Mpi.....6
Mpiexec.....6
Y llega el hola mundo.....6
Creamos un script.....7
Ejecutamos el job.....7
Programando en Python MPI.....8
Creamos nuestro primer programa en Python.....8
Creamos el Script.....9

Introducción a Clúster de computadoras:

Un cluster de computadoras es una colección de computadoras individuales (nodos) interconectadas que trabajan juntas como si fueran una sola entidad de cómputo. Estas computadoras están diseñadas para funcionar de manera coordinada para realizar tareas de procesamiento intensivo, cálculos científicos, simulaciones, análisis de datos, renderización, y una amplia variedad de aplicaciones que requieren una gran capacidad de procesamiento. Los clusters de computadoras se utilizan en entornos de alto rendimiento (HPC), en la nube, en centros de datos y en la investigación científica.

Características:

- **Rendimiento Escalable**: Los clusters permiten aumentar el rendimiento de manera escalable agregando más nodos según sea necesario. Esto permite abordar tareas computacionalmente intensivas que serían imposibles de procesar en una sola computadora.
- **Disponibilidad y Tolerancia a Fallos**: Los clusters pueden estar configurados para ser altamente disponibles y tolerantes a fallos. Si un nodo falla, otros nodos pueden asumir la carga de trabajo para garantizar la continuidad del servicio.
- **División de Tareas**: Los trabajos pueden dividirse en múltiples nodos para su procesamiento en paralelo, lo que acelera significativamente el tiempo de ejecución.
- **Eficiencia Energética**: Los clusters pueden estar diseñados con hardware más eficiente en términos de energía para tareas de cómputo de alta densidad. (Cronos se enmarca en el concepto de Green Computing).
- **Flexibilidad y Adaptabilidad**: Los clusters son flexibles y se pueden configurar para adaptarse a una amplia variedad de aplicaciones y necesidades.

Sin embargo, gestionar un clúster de computadoras y coordinar el uso eficiente de sus recursos puede ser una tarea compleja. Es acá donde entra en juego un software de gestión de recursos como Slurm (Simple Linux Utility for Resource Management).

Green computing, también conocido como "green IT" o "Tecnologías de la Información Verde", es un enfoque que se centra en la utilización eficiente y sostenible de la tecnología de la información y las comunicaciones (TIC) con el objetivo de reducir su impacto ambiental y promover prácticas más respetuosas con el medio ambiente en la industria de la tecnología y la informática. El término "green" en este contexto se refiere a la sostenibilidad y la preocupación por el medio ambiente.

SOFTWARE (Cronos)

Munge

Munge es una herramienta de seguridad diseñada específicamente para entornos de clusters de computadoras y sistemas de gestión de trabajos como Slurm. Su función principal es proporcionar un mecanismo seguro y eficiente para la autenticación de usuarios y procesos dentro del clúster.

Características:

- **Autenticación Segura**: En un clúster de computadoras, es esencial asegurarse de que solo usuarios y procesos legítimos tengan acceso a los recursos y que no se produzca acceso no autorizado. Munge proporciona una capa adicional de seguridad al autenticar a los usuarios y procesos de manera segura y eficiente.

- **Evitar Suplantación de Identidad**: Munge evita la suplantación de identidad, lo que significa que garantiza que los procesos que se ejecutan en el clúster sean realmente lo que dicen ser. Esto evita que los usuarios maliciosos o no autorizados intenten aprovecharse del sistema haciéndose pasar por otros usuarios o procesos.

- **Integración con Slurm y Clústeres**: Munge se integra estrechamente con sistemas de gestión de trabajos como Slurm, lo que facilita la autenticación de trabajos y procesos dentro del entorno del clúster. *Slurm utiliza Munge para asegurarse de que solo los trabajos autorizados se ejecuten y se asignen recursos*.
- **Tokens de Autenticación**: Munge utiliza tokens de autenticación que son generados de manera segura y que contienen información de autenticación codificada. Estos tokens son utilizados por Slurm y otros componentes del clúster para verificar la autenticidad de los procesos y garantizar que tengan permiso para acceder a los recursos.
- **Comunicación Segura**: Munge se utiliza en la comunicación entre nodos del clúster y con el controlador de Slurm para garantizar que la información de autenticación sea confiable y no pueda ser interceptada o manipulada por terceros.

Munge es esencial en entornos de clusters de computadoras y sistemas de gestión de trabajos como Slurm para garantizar la autenticación segura de usuarios y procesos, prevenir la suplantación de identidad y asegurar que solo los trabajos autorizados se ejecuten en el clúster. Proporciona una capa adicional de seguridad que es fundamental para la operación segura y confiable de clústeres de cómputo de alto rendimiento.

Introducción a Slurm:

Slurm (Simple Linux Utility for Resources Management), es un sistema de gestión de tareas (*scheduler*) y de clústeres (nodos o servidores de cómputo). Los servicios de procesamiento de CRONOS están gestionados por este servicio.

Slurm como sistema de gestión tiene tres tareas clave:

- Asignar a los usuarios acceso exclusivo o no exclusivo a nodos de cómputo durante un tiempo determinado para que puedan ejecutar sus tareas.
- Proporciona un framework que permite iniciar, ejecutar y supervisar el trabajo.
- Se encarga de arbitrar la disputa de recursos, administrando una cola de tareas pendientes.

Por ejemplo, se reservarán los recursos particulares que necesite un programa en concreto, y se pondrá en espera si no están disponibles en un momento dado. Los usuarios envían trabajos para que se ejecuten, y solo tienen que esperar a que finalice, sin preocuparse de si el trabajo se ha realizado en uno, dos o tres nodos de cómputo.

Comandos básicos de Slurm

Slurm proporciona varias herramientas útiles de línea de comandos que usaremos para interactuar con el clúster. Iniciar sesión en su nodo maestro/de inicio de sesión que configuramos la última vez:

```
ssh clusteruser@cunodo00
```

El primer comando que veremos es sinfo. Esto es bastante sencillo, solo proporciona información sobre el clúster:

```
$ sinfo
PARTITION   AVAIL   TIMELIMIT   NODES   STATE   NODELIST
cronos*     up      infinite    6       mix     cunodo[0-5]
```

Acá tenemos el nombre de la partición, si se puede usar, el límite de tiempo predeterminado, el número de nodos y sus estados. El estado “mix” se produce cuando un nodo tiene un trabajo en ejecución, pero todavía tiene algunos recursos disponibles. (Por ejemplo, cuando solo se usa 1 núcleo).

Opciones Comunes:

- -N o --nodes: Muestra información detallada sobre los nodos del clúster, incluyendo su estado (disponible, ocupado, inactivo, etc.), la cantidad de CPUs, GPUs, la memoria disponible y otros recursos.
- -P o --partition: Filtra la salida para mostrar información específica de una partición particular.
- -l o --long: Proporciona una salida más detallada que incluye información adicional sobre cada nodo.
- -o o --format: Permite especificar el formato de salida, lo que le permite personalizar la información que se muestra.

Información que Puede Mostrar sinfo:

- Estado de los Nodos: sinfo muestra si los nodos están disponibles (alloc), ocupados (alloc, comp), inactivos (drain), sin respuesta (drain*), deshabilitados (down), en mantenimiento (maint), etc.
- Particiones: Muestra información sobre las particiones disponibles en el clúster, incluyendo sus nombres, estados y características asociadas.
- Recursos Disponibles: sinfo proporciona detalles sobre los recursos disponibles en los nodos, como el número de CPUs, GPUs, la memoria total y libre, y otros recursos personalizados.

srun- programar comandos

El comando srun es muy útil, se utiliza para ejecutar directamente un comando en la cantidad de nodos/núcleos que desee. Probémoslo:

```
$ srun --nodes=3 hostname
cunodo00
cunodo01
cunodo02
```

Ejecutamos el comando hostname en 3 nodos. Esto es diferente a ejecutarlo en 3 núcleos, que pueden estar todos en el mismo nodo:

```
$ srun --ntasks=3 hostname
cunodo01
cunodo01
cunodo01
```

Ntasks se refiere al número de procesos. Este es efectivamente el número de núcleos en los que se debe ejecutar el comando. Estos no están necesariamente en máquinas diferentes. Slurm simplemente toma los siguientes núcleos disponibles.

También podemos combinar los dos:

```
$ srun --nodes=2 --ntasks-per-node=3 hostname
node1
node2
node2
node1
node2
node1
```

Esto ejecuta el comando en 2 nodos y lanza 3 tareas por nodo, efectivamente 6 tareas.

squeue- ver trabajos programados/encolados

Cuando empiece a ejecutar trabajos cada vez más largos y complejos, es útil comprobar su estado. Para hacer esto, ejecutar el comando *squeue*. De forma predeterminada, muestra todos los trabajos enviados por todos los usuarios y sus estados:

```
$ squeue
```

```
JOBID PARTITION  NAME     USER ST    TIME  NODES NODELIST(REASON)
 609 cronos     24.sub.s pi    R     10:16     1 node2
```

La mayor parte de esta información se explica por sí misma. Lo único que observaremos es la columna ST, que es el estado del trabajo. R significa que el trabajo se está ejecutando. Lista completa de códigos de estados.

scancel- cancelar un trabajo encolado

Una vez que se ha programado/encolado un trabajo, se puede cancelar mediante el comando scancel:

```
$ scancel 609
```

(Donde 609 es el JOBID que desea cancelar) Tené en cuenta que solo puede cancelar trabajos iniciados por su usuario.

sbatch- programar un batch script

sbatch es realmente el corazón del planificador Slurm. Es lo que usamos con más frecuencia cuando queremos programar un job para que se ejecute en el clúster. Este comando toma una serie de flags y configuraciones, así como un archivo de shell. Ese archivo de shell se ejecuta una vez y los recursos solicitados (nodos /núcleos/etc.) se ponen a su disposición. Hagamos un job básico como ejemplo.

El archivo Batch
Nuestro job comienza con la definición de un archivo por lotes/batch . Este archivo batch suele ser un script bash que ejecuta nuestro trabajo, sin embargo, se ve un poco diferente. Crearemos el archivo /clusterfs/directorio/holamundo.sh:

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --partition=<partition name>
cd $SLURM_SUBMIT_DIR
echo "Hola Mundo!" > holamundo.txt
```

El archivo comienza con un numeral. Esto es necesario, ya que le dice a Slurm cómo ejecutar tu trabajo. A esto le siguen una serie de flags que adoptan la siguiente forma:

```
#SBATCH <flag>
```

Estos flags simplifican cualquier parámetro que se pueda pasar al comando sbatch. Estos son casi idénticos a los que usa el comando srun, pero con una diferencia principal: los trabajos no son automáticamente relanzados en cada nodo núcleo especificado.

Más bien, cada trabajo se ejecuta en el primer núcleo del primer nodo al que se le asignó, pero el trabajo tiene acceso a los otros nodos que ha solicitado. Más sobre eso más tarde.

El cd `$SLURM_SUBMIT_DIR` garantiza que nuestro trabajo se está ejecutando en cualquier directorio desde el que se envió. En nuestro caso, este es `/clusterfs/directorio/`.

Ahora, podemos decirle a Slurm que encole y ejecute nuestro trabajo:

```
$ sbatch ./holamundo.sh
Submitted batch job 639
```

Dado que nuestro trabajo es muy simple, debe realizarse básicamente de inmediato. Si todo ha ido según lo planeado, deberíamos ver el archivo /clusterfs/holamundo.txt que creamos.

Salida

Notaremos que el trabajo no genera nada en el shell, lo cual tiene sentido. Si tuvo un trabajo en ejecución durante 4 horas, no es muy útil tener una terminal abierta todo el tiempo para obtener resultados. En su lugar, Slurm genera un error estándar y una salida estándar en un archivo con el formato slurm-XXX.out, donde XXX es el número de identificación del trabajo.

Open Mpi

OpenMPI es una implementación de código abierto de la interfaz de paso de mensajes. Un MPI es un software que conecta procesos que se ejecutan en varias computadoras y les permite comunicarse mientras se ejecutan. Esto es lo que permite que un solo script ejecute un trabajo distribuido en varios nodos del clúster.

MPI es una especificación que permite a los programadores escribir programas que pueden ejecutarse en paralelo en múltiples nodos de un clúster y comunicarse entre sí.

Características de Open MPI:

- **Programación Paralela**: Open MPI está diseñado para admitir la programación paralela, lo que significa que permite a los desarrolladores escribir programas que se ejecutan en múltiples nodos de un clúster y trabajan juntos para resolver tareas computacionalmente intensivas de manera más rápida y eficiente.
- **Comunicación entre Procesos**: MPI proporciona un conjunto de funciones y bibliotecas que permiten a los procesos en diferentes nodos comunicarse entre sí. Esto es fundamental para la coordinación de tareas en paralelo y el intercambio de datos en un clúster.
- **Portabilidad**: Open MPI es compatible con una amplia variedad de arquitecturas de hardware y sistemas operativos, lo que significa que los programas escritos con Open MPI pueden ejecutarse en diferentes tipos de clústeres sin necesidad de cambios significativos en el código.
- **Escalabilidad**: Open MPI es altamente escalable y puede manejar clústeres de diferentes tamaños, desde sistemas pequeños con unos pocos nodos hasta supercomputadoras con miles de nodos.
- **Rendimiento**: Open MPI está optimizado para obtener un alto rendimiento en términos de velocidad de comunicación entre nodos y eficiencia de recursos.
- **Biblioteca de Rutinas MPI**: Open MPI proporciona una implementación completa de la especificación MPI, lo que significa que incluye una amplia gama de funciones y rutinas que los programadores pueden utilizar para desarrollar aplicaciones paralelas.
- **Código Abierto**: Open MPI es de código abierto, lo que significa que el código fuente está disponible para que los desarrolladores lo utilicen, modifiquen y distribuyan de acuerdo con las licencias de código abierto.

Mpiexec

Es un comando que forma parte de Open MPI y se utiliza para lanzar y ejecutar programas paralelos escritos utilizando la interfaz de paso de mensajes (MPI) en clústeres de computadoras y sistemas de alto rendimiento.

```
mpiexec --version
```

Con este comando chequeamos la versión de OpenMpi, podés probarlo en el nodo maestro del cluster.

Y llega el hola mundo

Vamos a crear un programa en C que crea un clúster MPI con los recursos que SLURM asigna a nuestro trabajo. Luego, llamará a un comando print en cada proceso.

Cree el archivo /clusterfs/hello_mpi.c con el siguiente contenido:

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

```
int main(int argc, char** argv)
{
   int node;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &node);
   printf("Hola mundo desde el Nodo %d!\n", node);
   MPI_Finalize();
}
```

Acá, incluimos la biblioteca `mpi.h` proporcionada por OpenMPI. Luego, en la función principal, inicializamos el clúster MPI, obtenemos el número del nodo en el que se ejecutará el proceso actual, imprimimos un mensaje y cerramos el clúster MPI.

Necesitamos compilar nuestro programa C para ejecutarlo en el clúster. Sin embargo, a diferencia de un programa C normal, no usaremos `gcc` como es de esperar. En cambio, OpenMPI proporciona un compilador que vinculará automáticamente las bibliotecas MPI.

Debido a que necesitamos usar el compilador proporcionado por OpenMPI, vamos a tomar una instancia de shell de uno de los nodos:

```
login1$ srun --pty bash
node1$ cd /clusterfs
node1$ mpicc hola_mpi.c (mpicc -o archejectuable loquesea.c)
node1$ ls
a.out* hello_mpi.c
node1$ exit
```

El archivo a.out es el programa compilado que ejecutará el clúster.

Creamos un script

Ahora, crearemos el script de envío que ejecuta nuestro programa en el clúster. Cree el archivo /clusterfs/sub_mpi.sh:

```
#!/bin/bash

cd $SLURM_SUBMIT_DIR

# Print the node that starts the process
echo "Master node: $(hostname)"

# Run our program using OpenMPI.
# OpenMPI will automatically discover resources from SLURM.
mpirun a.out
```

Ejecutamos el job

Ejecutamos el trabajo enviándolo a SLURM y solicitando un par de nodos y procesos:

```
$ cd /clusterfs
$ sbatch --nodes=3 --ntasks-per-node=2 sub_mpi.sh
Submitted batch job 1211
```

Esto le dice a SLURM que obtenga 3 nodos y 2 núcleos en cada uno de esos nodos. Si tenemos todo funcionando correctamente, esto debería crear un clúster MPI con 3 nodos. Suponiendo que esto funcione, deberíamos ver algunos resultados en nuestro archivo `slurm-XXX.out`:

```
Master node: node1
Hello World from Node 0!
Hello World from Node 1!
```

```
Hello World from Node 2!
Hello World from Node 3!
Hello World from Node 4!
Hello World from Node 5!
```

Programando en Python MPI

Finalmente, para probar nuestras nuevas instalaciones de OpenMPI y Python, vamos a realizar un trabajo rápido de Python que usa OpenMPI. Para interactuar con OpenMPI en Python, usaremos una biblioteca fantástica llamada mpi4py.

Para nuestra demostración, usaremos uno de los programas de demostración en el repositorio de mpi4py. Vamos a calcular el valor de pi (el número) en paralelo.

Creamos nuestro primer programa en Python

Como se mencionó anteriormente, vamos a utilizar uno de los programas de demostración proporcionados en el repositorio de mpi4py.repo. Sin embargo, debido a que lo ejecutaremos a través del scheduler Slurm, debemos modificarlo para que no requiera ninguna entrada del usuario. Cree el archivo /clusterfs/calc-pi/calculate.py:

```
from mpi4py import MPI
from math   import pi as PI
from numpy  import array

def comp_pi(n, myrank=0, nprocs=1):
    h = 1.0 / n
    s = 0.0
    for i in range(myrank + 1, n + 1, nprocs):
        x = h * (i - 0.5)
        s += 4.0 / (1.0 + x**2)
    return s * h

def prn_pi(pi, PI):
    message = "pi is approximately %.16f, error is %.16f"
    print  (message % (pi, abs(pi - PI)))

comm = MPI.COMM_WORLD
nprocs = comm.Get_size()
myrank = comm.Get_rank()

n    = array(0, dtype=int)
pi   = array(0, dtype=float)
mypi = array(0, dtype=float)

if myrank == 0:
    _n = 20 # Enter the number of intervals
    n.fill(_n)
comm.Bcast([n, MPI.INT], root=0)
_mypi = comp_pi(n, myrank, nprocs)
mypi.fill(_mypi)
comm.Reduce([mypi, MPI.DOUBLE], [pi, MPI.DOUBLE],
            op=MPI.SUM, root=0)
if myrank == 0:
    prn_pi(pi, PI)
```

Este programa dividirá el trabajo de calcular nuestra aproximación de pi a la cantidad de procesos que le proporcionamos. Luego, imprimirá el valor calculado de pi, así como el error del valor almacenado de pi.

MPI_Comm_rank es una función importante en la biblioteca de paso de mensajes (MPI) y se utiliza para determinar el identificador (rango) de un proceso dentro de un comunicador MPI específico en un programa paralelo desarrollado con Open MPI u otras implementaciones de MPI. Esta función se utiliza comúnmente para que cada proceso sepa su posición en el comunicador MPI y pueda realizar tareas específicas en función de ese conocimiento. A continuación, se explica cómo funciona MPI_Comm_rank en Open MPI:

Sintaxis de la Función:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- `MPI_Comm comm`: Este es el comunicador MPI en el que se desea determinar el rango del proceso. Un comunicador MPI define un grupo de procesos que pueden comunicarse entre sí.
- `int *rank`: Esta es una referencia a una variable en la que se almacenará el rango del proceso dentro del comunicador MPI `comm`.

Funcionamiento:

1. Cuando se llama a MPI_Comm_rank, cada proceso en el comunicador comm proporciona su identificador de rango, que es único para cada proceso dentro de ese comunicador.
2. La función MPI_Comm_rank asigna este identificador de rango al proceso y lo almacena en la variable rank que se pasa como argumento.
3. Después de llamar a MPI_Comm_rank, cada proceso tendrá su propio valor de rango. Estos rangos se asignan de manera secuencial, comenzando desde 0. Es decir, el primer proceso en el comunicador tendrá un rango de 0, el segundo tendrá un rango de 1 y así sucesivamente.

Ejemplo de Uso en C:

Supongamos que tenemos un programa MPI en el que varios procesos están trabajando juntos. Cada proceso llama a `MPI_Comm_rank` y almacena su rango en una variable. Luego, pueden utilizar estos rangos para realizar tareas específicas según su posición en el comunicador. Por ejemplo:

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

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

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

    printf("Soy el proceso %d de un total de %d procesos.\n", rank,
size);

    MPI_Finalize();
    return 0;
}
```

En este ejemplo, cada proceso obtiene su rango (rank) en el comunicador MPI_COMM_WORLD (el comunicador por defecto que incluye a todos los procesos) y lo imprime. Esto permite que cada proceso identifique su posición en el grupo y realice tareas específicas según su rango.

Creamos el Script

Podemos ejecutar nuestro trabajo usando el scheduler. Solicitaremos algunos núcleos del clúster y SLURM preconfigurará el entorno MPI con esos núcleos. Luego, simplemente ejecutamos nuestro programa Python usando OpenMPI. Creemos el archivo de envío `/clusterfs/calc-pi/sub_calc_pi.sh`:

```
#!/bin/bash
```

```
#SBATCH --ntasks=4
cd $SLURM_SUBMIT_DIR
mpiexec -n 4 python3 calculate.py
```

Aquí, usamos la bandera `--ntasks`. Donde el indicador `--ntasks-per-node` solicita una cierta cantidad de núcleos para cada nodo, el indicador `--ntasks` solicita un número específico de núcleos en total. Debido a que usamos MPI, podemos tener núcleos en todas las máquinas. Por lo tanto, solo podemos solicitar la cantidad de núcleos que queramos. En este caso, pedimos 6 núcleos.

Para ejecutar el programa actual, usamos `mpiexec` y le decimos que tenemos 6 núcleos. Le decimos a OpenMPI que ejecute nuestro programa Python.

Tengamos en cuenta que puede ajustar el número de núcleos para que sea mayor/menor como desee. Solo asegúremonos de cambiar el indicador `mpiexec -n ##` *para que coincida.*

Finalmente, podemos ejecutar el script:

```
$ cd /clusterfs/calc-pi
$ sbatch sub_calc_pi.sh
Submitted batch job 1215
```

¡Eureka!

El cálculo debería tomar sólo un par de segundos en el clúster. Cuando se complete el trabajo (recuerde, puede monitorearlo con `squeue`), deberíamos ver algo de salida en el archivo `slurm - ####. Out:`

```
$ cd /clusterfs/calc-pi
$ cat slurm-1215.out
pi is approximately 3.1418009868930934, error is 0.0002083333033003
```

Podemos modificar el programa para calcular un valor más preciso de pi aumentando el número de intervalos en los que se ejecuta el cálculo. Haga esto modificando el archivo `calculate.py`:

```
if myrank == 0:
    _n = 20         # change this number to control the intervals
    n.fill(_n)
```

Por ejemplo, aquí está el cálculo ejecutado en 500 intervalos:

```
pi is approximately 3.1415929869231265, error is 0.0000003333333334
```

Bibliografia

https://lsi2.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Comm_rank&idioma=en

https://proteus.ugr.es/docs/slurm/

https://dokuwiki.ccad.unc.edu.ar/doku.php?id=pagina_principal

https://slurm.schedmd.com/