Borrador. Práctica opcional: Despliegue de un clúster con herramientas IaC y ejecución de NPB en diferentes proveedores cloud

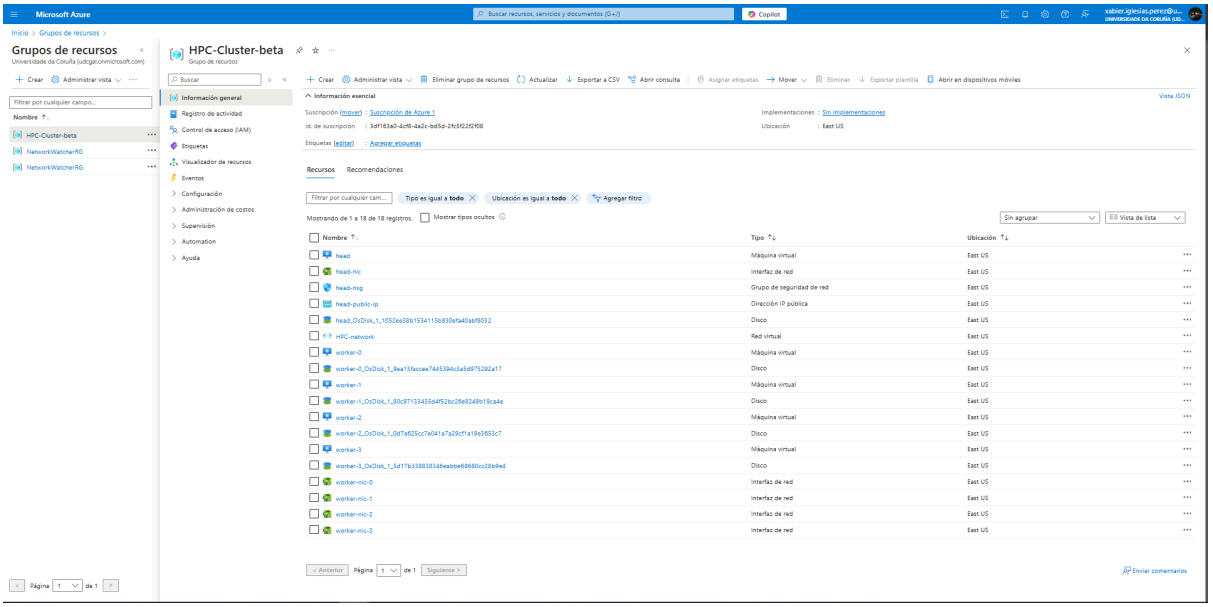
Xabier Iglesias Pérez (xabier.iglesias.perez@udc.es)

Version 0.1 - GIT: https://github.com/TretornESP/HPCN-TERRAFORM

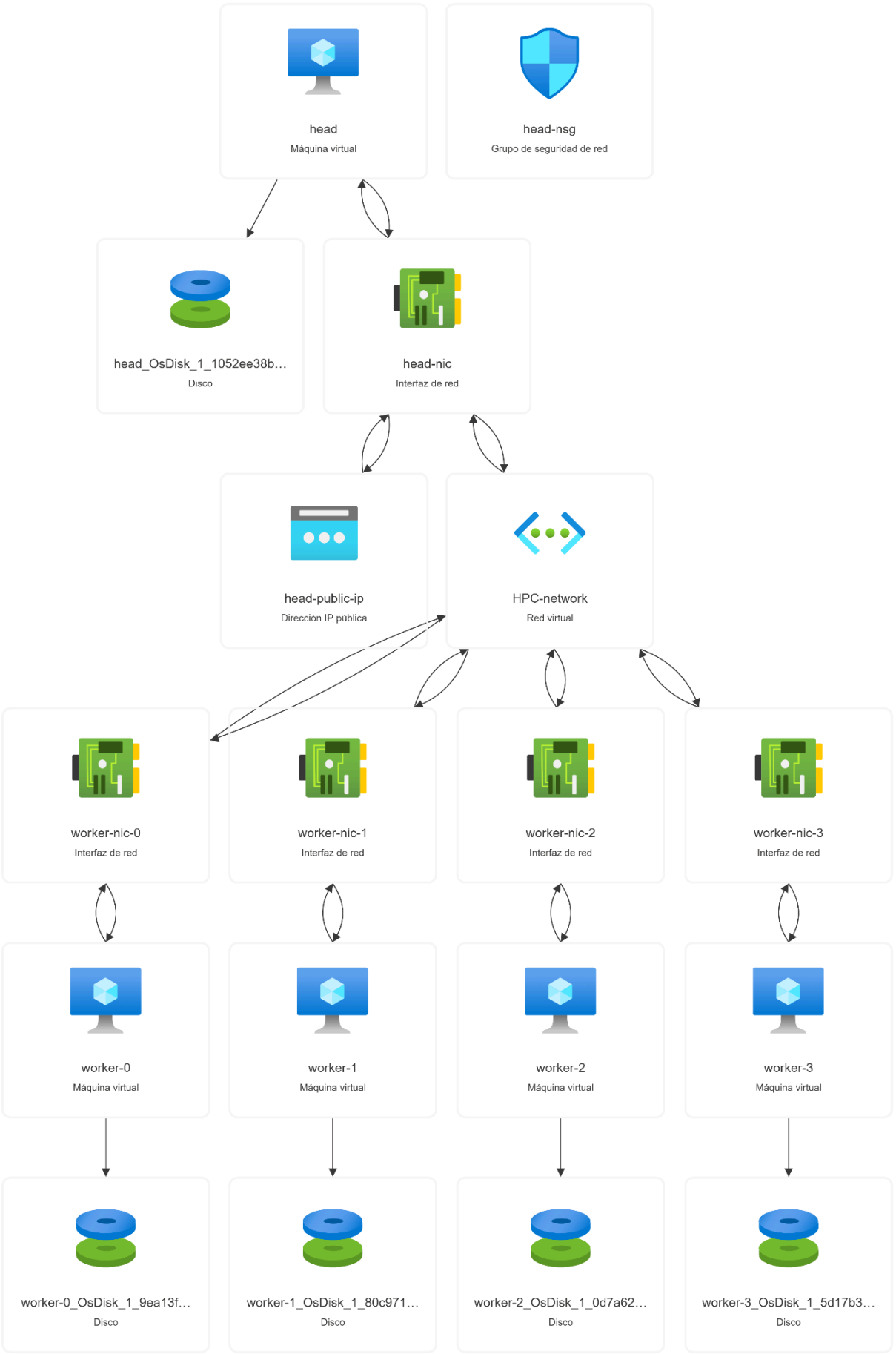
1. Introducción

Este proyecto es la continuación natural de la práctica de esta asignatura sobre clústeres virtuales en AWS. Siendo similar en su planteamiento, la principal diferencia es que el objetivo de aprendizaje es familiarizarse con otras plataformas de computación en la nube así cómo con tecnologías de *infraestructura como código (IaC)*. Para ello hemos creado un nuevo módulo de automatismos que enlaza con el fichero *execute.sh* de la práctica anterior. y que, mediante Terraform, es capaz de desplegar un clúster semejante en diferentes nubes. A la fecha de redacción de este borrador hemos ejecutado pruebas exitosas en Azure.

Cómo evidencia adjuntamos unas breves capturas:



Captura 1. Pantalla de visor de recursos agregada de Azure.


head
Máquina virtual
head-nsg
Grupo de seguridad de red
head_OsDisk_1_1052ee38b…
Disco
head-nic
Interfaz de red
head-public-ip
Dirección IP pública
HPC-network
Red virtual
worker-nic-0
Interfaz de red
worker-nic-1
Interfaz de red
worker-nic-2
Interfaz de red
worker-nic-3
Interfaz de red
worker-0
Máquina virtual
worker-1
Máquina virtual
worker-2
Máquina virtual
worker-3
Máquina virtual
worker-0_OsDisk_1_9ea13f…
Disco
worker-1_OsDisk_1_80c971…
Disco
worker-2_OsDisk_1_0d7a62…
Disco
worker-3_OsDisk_1_5d17b3…
Disco

Captura 2. Arquitectura del clúster generado con terraform

```
azureuser@head:/nfs/mpi/npbinaries$ sinfo -N
NODELIST   NODES PARTITION STATE
worker-0       1      aws* alloc
worker-1       1      aws* idle
worker-2       1      aws* idle
worker-3       1      aws* idle
azureuser@head:/nfs/mpi/npbinaries$ scontrol show node worker-0
NodeName=worker-0 Arch=x86_64 CoresPerSocket=2
   CPUAlloc=2 CPUEfctv=2 CPUTot=2 CPULoad=1.21
   AvailableFeatures=(null)
   ActiveFeatures=(null)
   Gres=(null)
   NodeAddr=10.0.1.9 NodeHostName=worker-0 Version=22.05.11
   OS=Linux 6.1.0-28-cloud-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.119-1 (2024-11-22)
   RealMemory=3927 AllocMem=0 FreeMem=2362 Sockets=1 Boards=1
   State=ALLOCATED+DYNAMIC_NORM ThreadsPerCore=1 TmpDisk=0 Weight=1 Owner=N/A MCS_label=N/A
   Partitions=aws
   BootTime=2025-01-10T11:52:54 SlurmdStartTime=2025-01-10T11:57:50
   LastBusyTime=2025-01-10T12:10:44
   CfgTRES=cpu=2,mem=3927M,billing=2
   AllocTRES=cpu=2
   CapWatts=n/a
   CurrentWatts=0 AveWatts=0
   ExtSensorsJoules=n/s ExtSensorsWatts=0 ExtSensorsTemp=n/s

azureuser@head:/nfs/mpi/npbinaries$
```

Captura 3. Output de los comandos sinfo y scontrol sobre el head node

2. Configuración experimental

En este caso, si bien nos guiamos por un criterio de similitud con respecto a la práctica, estamos limitados a un máximo de 10 vcpus, con lo que nuestra configuración será de 4 nodos con 2vcpus para los workers y 4gb de RAM. Mientras que el head node consumirá las 2vcpus y 4 gb de ram restantes.

Atendiendo a criterios de coste emplearemos instancias generalistas del tipo B2s (B=Economica con capacidad de burst, 2=2vcpus, s=almacenamiento de alto rendimiento) que se corresponden con un Intel Xeon E5-2673 a 2.4GHz y un único dominio NUMA. Es curioso mencionar que Azure no posee una oferta tan granular de nodos de cómputo cómo lo hace AWS.

Para esta primera prueba todas las instancias se encontrarán en la zona de disponibilidad East US. Todos los resultados son la media de 5 ejecuciones.

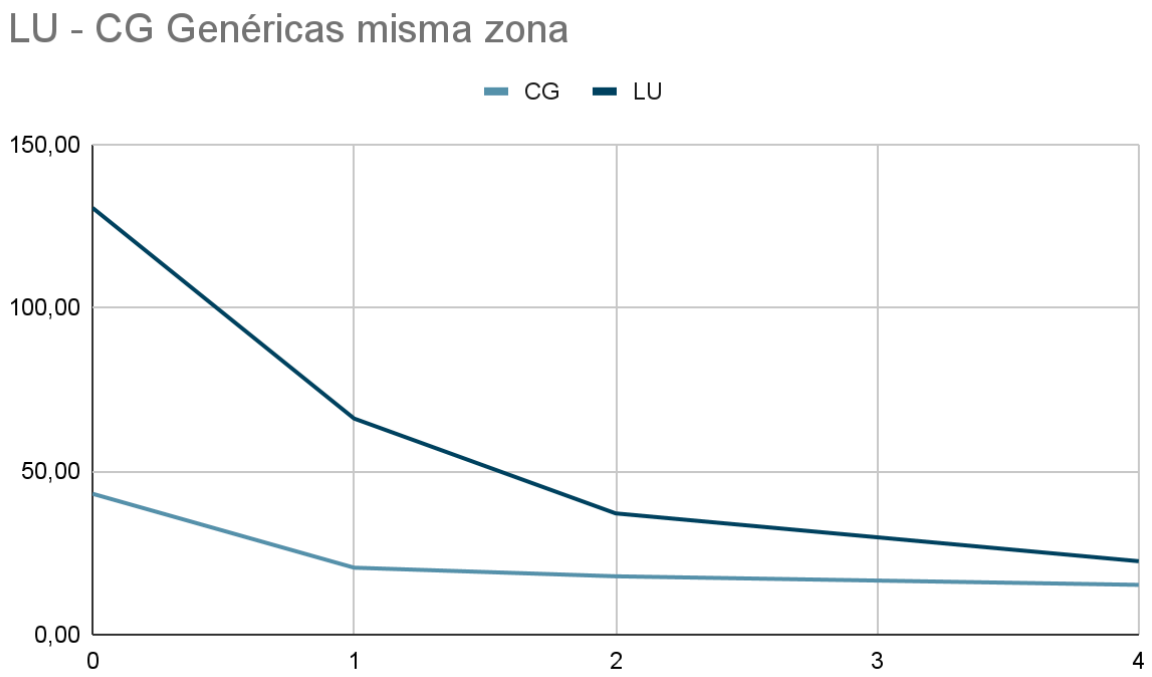
En lo que al sistema operativo se refiere, emplearemos Debian 12 vanilla sobre el que desplegaremos SLURM 22.05 y OpenMPI 4.1.4. Al igual que en la práctica emplearemos el script de automatismos execute.sh, a través de la interfaz runner.sh

3. Experimentos

En este primer borrador hemos ejecutado las cargas de trabajo CG - Conjugate Gradient y LU - Lower-Upper Gauss-Seidel Solver sobre instancias de cómputo genéricas sobre la misma zona de disponibilidad.

Hemos avanzado desde la versión secuencial 1 nodo / 1 core hasta la configuración con el máximo paralelismo que nos permiten los límites de cuota actuales 4 nodos / 8 cores) y hemos obtenido los siguientes resultados.

Para esta primera prueba hemos logrado las siguientes tablas de tiempos medios.



CG

Inst. Generales	1 nodo (1 core, control)	1 nodo (2 cores)	2 nodos (4 cores)	4 nodos (8 cores)
Tiempos	43,076	20,464	17,832	15,212
Speedup	1	2,10	2,42	2,83
Ef. Paralela	1	1,05	0,61	0,35

LU

Inst. Generales	1 nodo (1 core, control)	1 nodo (2 cores)	2 nodos (4 cores)	4 nodos (8 cores)
Tiempos	130,540	66,090	37,078	22,448
Speedup	1	1,98	3,52	5,82
Ef. Paralela	1	0,99	0,88	0,73

4. Conclusiones

Estamos pendientes de la comparativa con instancias dedicadas y en otras zonas de disponibilidad, así cómo en otros proveedores cloud. De momento podemos decir que la eficiencia paralela en el caso del benchmark LU es similar al de las instancias optimizadas de AWS. Mientras que en el benchmark CG es mucho más pobre, con lo que podemos afirmar que Azure nos ofrece máquinas potentes en los *tiers* más bajos pero con una arquitectura de interconexión muy pobre. *Esta es una conclusión aventurada hasta que tengamos datos en distintas zonas de disponibilidad.*

5. Terraform, azure y dificultades

Este es el apartado más diferencial de todo el trabajo, en este caso hemos dedicado un tiempo significativo a aprender a utilizar la herramienta Terraform para crear despliegues agnósticos de la plataforma. Es decir, en la versión final este proyecto será capaz de desplegar el clúster en Azure, GCP, AWS, Digital Ocean, etc con mínimos cambios. Para ello hemos adaptado el fichero de CloudFormation original, manteniendo la estructura básica aunque con algunos cambios resultantes del uso de un sistema operativo cómo Debian.

La principal dificultad ha surgido en las exigencias para el uso del plugin de cgroups, un cgroup (control group) es un mecanismo que permite limitar el uso de CPU, memoria y operaciones de entrada/salida que puede realizar un conjunto de procesos. Cómo podemos imaginarnos slurm hace un uso exhaustivo de esta funcionalidad (en su configuración por defecto) sin embargo, la versión disponible en la compilación básica y la ofertada por el kernel no son compatibles a simple vista, con lo que debemos compilarla. Tras mucho investigar descubrimos que el script de makefile lo hará de forma automática si tiene las librerías de desarrollo de dbus (cosa curiosa porque en principio no tienen nada que ver, dbus es un subsistema para IPC). Además existe un bug en la versión de slurm comentado en su sitio web según el cual el servicio de systemd no podrá arrancar si no encuentra el fichero cgroup.conf (aunque este se encuentre vacío). Para solventarlo en la etapa de despliegue copiaremos el fichero de ejemplo a la carpeta $SLURM_HOME/etc/

Finalmente hemos encontrado lo que creemos que es un bug en la implementación original, la línea del yaml SlurmdSpoolDir=/var/spool/slurm/d debería ser SlurmdSpoolDir=/var/spool/slurm/ a secas (o para ser más correctos terminológicamente SlurmdSpoolDir=/var/spool/slurmd/).

Por último el mecanismo de notificación de estado del nodo head a los workers se ha simplificado, ahora los workers esperarán un tiempo fijo a que se monte el servidor nfs y a partir de entonces recibirán mensajes desde el head a través de un fichero compartido /nfs/headnode_started.

6. Cómo ejecutar:

De momento, seguir los pasos especificados en el README.md.