Paralelismo

Si está trabajando en problemas complejos y de gran escala que requieren informática de alto rendimiento (HPC), es posible que se pregunte cómo elegir la mejor arquitectura para sus necesidades.

Los sistemas HPC se pueden clasificar en dos tipos principales: ***memoria compartida (Shared memory)*** y sistemas de ***memoria distribuida (Distributed memory)***.

Paralelismo de memoria compartida (threading)

En el *Paralelismo de memoria compartida* o *shared-memory parallelism (SM)*, las aplicaciones logran el paralelismo al ejecutar más de un subproceso a la vez en los núcleos dentro de un nodo. Cada uno de los subprocesos ve y manipula la misma memoria que asignó el proceso inicial. Los subprocesos son ligeros y rápidos y procesan de forma independiente partes del trabajo. Con el paralelismo SM, **los trabajos están limitados a la cantidad total de memoria y núcleos en un nodo**.

Para ejecutar una aplicación con subprocesos correctamente a través de Slurm, deberá especificar una serie de restricciones de Slurm. Por ejemplo, para iniciar un trabajo de Slurm para su aplicación con subprocesos que utilizará 16 subprocesos, haga lo siguiente:

```
#!/bin/bash
#SBATCH -J my_job
# Numero de Nodos y cores por nodo
#SBATCH -N 1
#SBATCH --cpu-per-task=16
#SBATCH --mem=45000
#SBATCH --constraint=sandy
#SBATCH --partition=batch

module load GCC/12.2.0
srun ./myapp
```

Si opta por no usar *srun* para iniciar su aplicación, debe agregar esto a su script de trabajo de Slurm: `export OMP_NUM_THREADS=16` Este número debe coincidir con *--cpus-per-task*.

Paralelismo de memoria distribuida (MPI)

En el *paralelismo de memoria de distribuida* o *distributed-memory parallelism (DM)*, una aplicación logra el paralelismo al ejecutar varias instancias de sí misma en varios nodos para resolver el problema. A cada instancia se le asigna su propia porción de memoria virtual y se comunica con otras instancias a través de una interfaz de paso de mensajes como MPI.

Por ejemplo, para usar 4 nodos y 16 procesadores por nodo,

```
#!/bin/bash
#SBATCH -J my_job
#SBATCH -N 4
#SBATCH --tasks=16
#SBATCH --mem=28000
#SBATCH --constraint=sandy
#SBATCH --partition=batch

module load   intel/2021b   intel-compilers/2021.4.0   impi/2021.4.0
srun ./myapp
```

Paralelismo híbrido: DM + SM

En el paralelismo híbrido, las aplicaciones logran el paralelismo con el uso de subprocesos y tareas MPI. Este tipo de paralelización combina las características de los dos modelos anteriores, lo que permite que un programa ejecute subprocesos en varios nodos del clúster.

Los trabajos híbridos pueden ejecutarse potencialmente de manera más eficiente (consumiendo menos memoria y escalando más) al reducir la sobrecarga de comunicación MPI de un trabajo. Esto se puede hacer sustituyendo hilos livianos por rangos MPI.

Para configurar trabajos para usar DM+SM, primero asegúrese de que su aplicación lo admita.

```
#!/bin/bash
#SBATCH –cpus-per-task=12  #same as -c 12
#SBATCH –ntasks=4          #same as -n 4
#SBATCH –ntasks-per-node=2

module load gompi/2022b
srun ./myapp
```

Los trabajos de DM pueden afectar el rendimiento debido a los pasos de comunicación de MPI cuando se ejecuta la aplicación en demasiados nodos.

Por ejemplo, si tiene un trabajo de DM que funciona mejor con 64 cores, puede solicitar que su trabajo se ejecute soólo en 4 nodos con *nodes=4*. Por supuesto, al hacer esto, es probable que su trabajo tarde más en comenzar debido a la restricción adicional del trabajo, pero podría valer la pena para sus análisis. Lo siguiente mejor que puede hacer es pedirle a Slurm que use preferiblemente 4 nodos, pero si no es posible, use 5 o 6. Debe especificar esto con *--nodes=4-6* Sin especificar *--nodes*, su trabajo de MPI podría ejecutarse en cualquier cantidad de nodos en el clúster.

OpenMP/Multithreading vs. MPI

Si bien hay varias formas de solicitar una cierta cantidad de núcleos de CPU para su programa, observe la siguiente distinción:

- **Solicitar *ntask* (nº) cores CPU para MPI (procesos distintos)**

Estas tareas se pueden distribuir en varios nodos de cómputo.

```
#SBATCH --ntask= <ntask>
```

- **Solicitar *nodes ntask-per-node* (nº) cores CPU para MPI (procesos distintos) en el mismo nodo**

Esto distribuye sobre *X* nodos de cómputo *Y* tareas.

```
#SBATCH --nodes= <nodes>
#SBATCH --ntask-per-node= <ntas-per-node>
```

- **Solicitar *cpu-per-task* (nº) de CPU cores para aplicaciones multithread (eg. OpenMP)**

--cpus-per-task" especifica cuántas CPU puede usar cada tarea. ¡Estos siempre se asignarán dentro de un sólo nodo de cómputo, nunca a varios nodos!

```
#SBATCH --cpus-per-task= <cpus-per-task>
```

Algunos consejos

Para una aplicación MPI simple, use

```
--ntasks,
esto usa memoria distribuida (entre nodos),
requiere MPI.
```

Para una aplicación simple de OpenMP/multiproceso, use

```
--ntasks=1
--cpus-per-task=<cpu_per-task>
Esto usa Memoria compartida (dentro de un solo nodo)."
```

Para una aplicación híbrida, use:

```
--ntasks=<no of nodes>
--cpus-per-task=<no of cores per node>,
Esto usa SM and DM y requiere MPI.
```

La opción SBATCH *--ntasks-per-core=#* sólo es adecuada para nodos de cómputo que tengan HyperThreading habilitado en hardware/BIOS

En TeideHPC, **el HyperThreading está desactivado por defecto**, es decir `#SBATCH --ntasks-per-core=1`

Haz tus propios test de escalabiliddad

Comience con 2 nodos y estudie los resultados y aumente la cantidad de nodos o tareas poco a poco.

Ejemplo básico: *usar srun or no usarlo*

```
#!/bin/bash
#SBATCH --ntasks=8
## more options
echo hello
```

Esto siempre generará una línea, porque el script sólo se ejecuta en uno de los nodos

```
#!/bin/bash

#SBATCH --ntasks=8

## more options
srun echo hello
```

srun hace que el script ejecute su comando en los nodos trabajadores y, como resultado, debería obtener 8 líneas de saludo.

Ejemplo básico:*Usar ntask y una aplicación de 1 thread.*

- Usando el valor por defecto de *ntasks=1*

```
#!/bin/bash

#SBATCH --ntasks=1

srun sleep 10 &
srun sleep 12 &
wait
```

El número de tareas predeterminado se especificó en 1 y, por lo tanto, la segunda tarea no puede comenzar hasta que la primera tarea haya finalizado. Este trabajo tardará en alrededor de 22 segundos. Desglosando esto:

`sacct -j1425 --format=JobID,Start,End,Elapsed,NCPUS`

```
       JobID             Start               End    Elapsed      NCPUS
------------ ------------------- ------------------- ---------- ----------
1425       2023-07-13T20:51:44 2023-07-13T20:52:06   00:00:22          1
1425.batch 2023-07-13T20:51:44 2023-07-13T20:52:06   00:00:22          1
1425.0     2023-07-13T20:51:44 2023-07-13T20:51:56   00:00:12          1
1425.1     2023-07-13T20:51:56 2023-07-13T20:52:06   00:00:10          1
```

Aquí la tarea 0 comenzó y finalizó (en 12 segundos) seguida de la tarea 1 (en 10 segundos). Para hacer un tiempo de usuario total de 22 segundos.

- Usando *ntasks=2*

```
#!/bin/bash

#SBATCH --ntasks=2

srun --ntasks=1 sleep 10 &
```

```
srun --ntasks=1 sleep 12 &
wait
```

Ejecutando el mismo comando sacct como se especificó anteriormente:

`sacct -j 515064 --format=JobID,Start,End,Elapsed,NCPUS`

```
JobID             Start               End    Elapsed      NCPUS
------------ ------------------- ------------------- ---------- ----------
515064       2023-07-13T21:34:08 2023-07-13T21:34:20   00:00:12         2
515064.batch 2023-07-13T21:34:08 2023-07-13T21:34:20   00:00:12         2
515064.0     2023-07-13T21:34:08 2023-07-13T21:34:20   00:00:12         1
515064.1     2023-07-13T21:34:08 2023-07-13T21:34:18   00:00:10         1
```

Más ejemplos.

Visite nuestro repositorio en github https://github.com/hpciter/user_codes.git