Guía para la confección del apartado "Descripción del enfoque HPC"

Esta guía te ayudará a recolectar la información necesaria para especificar los requerimientos de cómputo de tu programa para tu solicitud IPAC. Recomendamos seguir adicionalmente la guia del llamado:

https://www.argentina.gob.ar/sites/default/files/ipac_-_guia_para_la_presentacion_de_proyectos_2024.pdf .
Puede resultar útil también consultar la wiki de UNC Supercómputo: https://wiki.ccad.unc.edu.ar/.

Nota: Recordar decir en qué repositorio se publicarán y liberarán los datos.

C. Plan Computacional

Pensando en los objetivos científicos, hacé una tabla con las corridas que realizarás tal como se explica en la guía de la convocatoria. Dependiendo de la performance de tus códigos (ver D2), buscá adecuar los objetivos científicos y los recursos que utilizarás en cada corrida para que tu software tenga una performance aceptable. Por ejemplo, si al aumentar el número de cpus tu código corre muy por debajo del ideal (ej. el doble de cpus idealmente reduciría el tiempo a la mitad, ver D2.h), analizá la posibilidad de replantear las corridas. El balance de tiempo requerido, número de cpus y tamaño de tu problema, es algo que solo vos podés decidir, tratá de justificar en el plan tu elección y hacer compromisos entre estas tres variables.

Dependiendo de los objetivos científicos, hay que considerar las diferentes etapas que se realizarán en el cluster, y cuáles afuera en otros recursos. Dependiendo de ello considerá los análisis de datos, la generación de condiciones iniciales, el pre/postprocesado de los datos generados, antes de que se eliminen del cluster. Estos tiempos también deben ser considerados en esta tabla, ya que de esta manera se garantiza la factibilidad del proyecto.

D.1. Descripción del código

Describí de la manera más breve posible pero precisa cuál es la función de tu código, qué simula o analiza y cómo lo hace. Si disponés de documentación buscá allí toda la información que pueda ser útil. Si tu código es de uso extendido en tu comunidad científica buscá trabajos donde se utilice, si se corrió en equipamiento de HPC eso suele mencionarse en los agradecimientos. Describí acá cuán extendido es su uso o en qué centros de cómputo se ha utilizado (si es el caso). Si lo has corrido en UNC Supercómputo decilo aquí, junto con la

referencia a tus trabajos previos con el código. Podés calcular la cantidad de *horas-core* que tu grupo ejecutó en Serafín, por ejemplo, y darlo como antecedente, por ejemplo:

```
$ sreport --cluster=rome cluster -p AccountUtilizationByUser -t Hours
Users=username1,username2,username3,...,usernameN start=2021-01-01
end=2025-01-01
```

Si querés otro cluster, podés cambiar la opción `--cluster=` por:

- Serafín: `rome`
- MendietaF2: `ivb`
- Eulogia: `knl`
- Mulatona: `bdw`

Si tu código tiene página web o repositorio git indicalo aquí.

D2.a Lenguaje de programación

Indicá acá el lenguaje de programación de tu código. Si no estás seguro de esto y el código a utilizar dispone de un repositorio de git, en la página principal del proyecto suele darse la información sobre los lenguajes de programación utilizados (en GitHub, panel derecho donde dice *Languages*; en GitLab, panel derecho donde dice *Project information*).

D2.b Paradigma de paralelización utilizado

- **¿Usás MPI? (Message Passing Interface):**
  - Para saber si tu programa utiliza MPI, identificá las secciones paralelas del código que envían y reciben mensajes entre procesos. Suelen haber llamadas a funciones que comienzan con "`MPI_`".
    Más precisamente **buscá funciones de inicialización y finalización de MPI,** estas funciones indican que el programa está utilizando MPI, en el directorio de tu código:

    ```
    $ grep -n --color "MPI_Init" *.*
    $ grep -n --color "MPI_Finalize" *.*
    ```

    Si tu programa usa el comando `mpirun`, entonces usa MPI.

- **¿Usás OpenMP? (Open Multi-Processing):**
  - Buscá en tu código C o C++ directivas que empiezan con `#pragma omp`. Si tenés un código Fortran, buscá líneas que comiencen con`!$OMP`. Estas directivas indican que se usa OpenMP.
    Por ejemplo en el directorio de tu código corré:

```
$ grep --color -r -E "#pragma omp|OMP" *.*
```

- **GPUs (procesamiento en unidades gráficas):**
    - Verificá si tu programa utiliza bibliotecas como CUDA, OpenCL o *frameworks* como *TensorFlow/PyTorch*.
    - Podés revisar en qué partes del código se ejecutan en la GPU y cuáles en la CPU.
    - Puedes usar herramientas como `nvidia-smi` o `nvtop` para monitorear el uso de GPUs durante la ejecución. Si hay actividad, entonces usa la GPU.

D2.c y D2.d Compilador y librerías

Informá acá qué compilador utilizás normalmente para compilar tu programa. ¿Necesitás de otras librerías para correrlo? Una manera de ver esto es revisar los módulos que necesitás cargar al compilar y correr en algún cluster de UNC Supercómputo por ejemplo. Si tenés experiencia utilizando Spack o cualquier otro administrador de paquetes en tu home podés mencionarlo acá en caso que sepas también como instalar tus propias librerías.

D2.e Requerimientos de memoria por núcleo

Determina la memoria usada por cada núcleo

1. **Ejecutá tu programa y medí su uso de memoria:**
    - Utilizá el comando `top` o `htop` para observar el uso de memoria (columna `RES`) mientras tu programa se ejecuta.
    - Filtrá específicamente los procesos asociados a tu usuario.
2. **Calculá la memoria por núcleo:**
    - Si tu programa usa 4 núcleos y consume 8 GiB de memoria total RES, entonces el requerimiento por núcleo es:

      Memoria por núcleo = Memoria total / Número de núcleos

    - Un valor razonable es de 1 GiB por núcleo, aunque puede ser menor. En pocos casos es de 2 GiB o más por núcleo.
    - Clementina XXI tiene para cada nodo 64 núcleos y 512 GiB de RAM, asi que tiene 8 GiB por núcleo, un valor más que holgado[1].

[1] A survey of application memory usage on a national supercomputer: An analysis of memory requirements on ARCHER, Andy Turner*, Simon McIntosh-Smith.

D2.f Requerimientos de memoria virtual

Los sistemas Linux/UNIX no permiten que un proceso (programa en ejecución) acceda a la memoria física de manera directa. Para esto se utilizan métodos indirectos que mapean la memoria física con la memoria que el programa realmente ve o memoria virtual. La memoria virtual que ocupa tu programa usualmente (VIRT) es mucho mayor que la memoria física que ocupa gracias a algunas técnicas de uso eficiente de la memoria. **Se puede saber la memoria virtual que ocupa tu programa viendo la columna VIRT en htop,** ese es el número que debés informar acá.

D2.g Capacidad de reinicio de los cálculos

Establecé acá cada cuanto tiempo tu programa guarda en disco los datos necesarios para poder reiniciarse, en caso de un corte de luz o por decisión de los administradores, por ejemplo. La mayoría de los códigos de simulación, suelen salvar el estado del programa en el disco cada cierta cantidad de tiempo a fin de facilitar correr desde el último punto de salvado en caso de interrumpirse la ejecución. Si tu código es propio, es importante que tenga esta capacidad, que se logra simplemente guardando en cada salida los datos necesarios para reiniciarlo desde ese punto.

En el caso de corridas de arreglos de simulaciones monocore, si se van salvando los resultados de las corridas anteriores ya es suficiente como estrategia de reinicio de los cálculos, pero debe ser especificado.

D2.h Escalabilidad del código

¿Cuánto aumenta la velocidad de tu código, o equivalentemente cuánto disminuye el tiempo de ejecución, cuando se utilizan más CPUs/GPUs? Si tu código es un paquete muy utilizado en la comunidad seguramente dispone de estos gráficos. Buscá información al respecto e incluí estos gráficos en esta sección. Si tu código es código propio podés medir los tiempos de ejecución variando el número de cpus, en potencias de 2 en un mismo nodo. Si sos usuario de UNC Supercómputo utilizá la guía de ejecución de trabajos en slurm https://wiki.ccad.unc.edu.ar/empezar/slurm.html. Si utilizás openmp podés variar el número de cpus que utilizás variando el número de hilos de ejecución: e.g. 2,4,8,16,etc (cambiando la variable cpus-per-task). Si utilizas mpi podés variar el número de tareas (variable ntasks). Tené en cuenta que en MPI, dependiendo del número de tareas y el número de hilos podés utilizar varios nodos. Para ver la velocidad revisá los archivos .log generados por tu código, o en su defecto mirá los tiempos en que tus trabajos se ejecutaron, utilizando por ejemplo:

```
$ sacct -j <job_id> --format=JobID,JobName,Elapsed,State
```

¿Por qué se pide esto? Tomemos por ejemplo el software GROMACS, en una configuración particular con un nodo supongamos que simula 10ns/day y con dos nodos hace 8ns/day. En este caso no resulta conveniente usar dos nodos, porque **se espera que al doble de hardware dé el doble de velocidad**. Esto último se conoce como *perfect scaling*. La derivada de esto se llama **eficiencia** y es 1 en caso de que la relación cantidad de cómputo vs. velocidad sea lineal. Eficiencias de 0.8 son razonables, eficiencias de 1 son una utopía y eficiencias por debajo de 0.5 indican que ya no tiene sentido realizar corridas con mayor cantidad de recursos (el problema no escala).

Aunque el documento no lo dice, habría que indicar que al menos se están usando procesadores completos (32 núcleos) o GPUs completas, ya que granularidades menores no resultan convenientes para una máquina de esta escala.

Si tenés un barrido de parámetros monocore, entonces el scaling es perfecto, podés pedir 64 núcleos, o sea un nodo entero, e internamente lanzar 64 procesos.

E. Plan de gestión de datos

¿Cuánto almacenamiento requieren tus corridas? ¿Cómo vas a mover los datos a tu computadora una vez que concluya tu proyecto? Da detalles de cuántos archivos y su tamaño aproximado.

Cómo saber el tamaño total que se genera:

```
$ du -h ~/proyecto/
```

Cómo saber cuántos archivos se generan:

```
$ ls -1R ~/proyecto | wc -l
```