PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC3533 — Computación de Alto Rendimiento — 2025-2

Tarea 2

Jueves 13 de Noviembre de 2025

En grupos de tres o cuatro personas, desarrollar y entregar el informe respectivo para la actividad siguiente.

Plazo de Entrega : 4 de Diciembre 2025

1. **[100pts]** *K-means Distribuido con MPI + Numba*

   En esta tarea, queremos implementar el algoritmo de *clustering* k-means en Python utilizando: paralelización híbrida con MPI (mpi4py) para comunicación entre procesos y Numba para paralelización con threads dentro de cada proceso.

   El algoritmo minimiza la inercia:

   $$J = \sum_{i=1}^{k} \sum_{x \in C_i} ||x - \mu_i||^2$$

   donde C_i es el conjunto de puntos en el cluster de puntos i y μ_i es el centroide del cluster i, definido como el promedio de los puntos en dicho *cluster*:

   $$\mu_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$$

   Algoritmo iterativo:

   (i) **Assignment:** Asignar cada punto al cluster más cercano.

   (ii) **Update:** Recalcular los k centroides como promedio de puntos asignados.

   (iii) Repetir hasta un máximo de iteraciones, o cuando los centroides no se desplacen más que una cierta tolerancia.

   (a) **[30pts]** Implementar las siguientes funciones con las especificaciones dadas:

   (i) `generate_distributed_data(n_total, d, k, seed)`: Cada proceso MPI genera independientemente su porción de datos (n/p puntos) sin comunicación. Usar semilla `seed + rank` para reproducibilidad.

   (ii) Funciones con Numba usando `@njit(parallel=True)`:

   - `compute_distances(data, centroids)`: Calcular distancias euclidianas con `prange`.
   - `assign_labels(distances)`: Asignar cada punto al cluster más cercano.
   - `compute_local_sums(data, labels, k)`: Acumular sumas por cluster.

   (iii) Comunicación MPI:

   - `comm.Bcast(centroids, root=0)`: Distribuir centroides iniciales.
   - `comm.Allreduce(local_sums, global_sums, op=MPI.SUM)`: Reducción global para recalcular.

(b) **[20pts]** Ejecutar el código en el cluster asignado al curso con las siguientes especificaciones:
Recursos del cluster:

- Partición: `hpc-iic3533`
- Nodos: Ahsoka y Ventress (2 nodos)
- CPUs máximos: 8 por nodo (16 CPUs totales).
- Tiempo máximo: 15 minutos por *job*.
- RAM: 4 GB por usuario.

Configuración del entorno con Miniconda:
Antes de ejecutar los experimentos, instalar Miniconda y crear un entorno virtual con las dependencias necesarias:

```
# 1. Descargar e instalar Miniconda en ~/miniconda3
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh -b -p ~/miniconda3

# 2. Crear entorno virtual para el proyecto
~/miniconda3/bin/conda create -n kmeans_env -y

# 3. Activar el entorno
source ~/miniconda3/bin/activate kmeans_env

# 4. Instalar paquetes necesarios
~/miniconda3/bin/conda install -c conda-forge numpy numba mpi4py -y

# 5. Verificar instalacion
python -c "import numpy; import numba; import mpi4py; print('OK')"
```

Instrucciones de uso del cluster:
Para información detallada sobre cómo conectarse al cluster, ejecutar jobs con SLURM, y mejores prácticas, consultar la documentación oficial y la Clase 19b:

`https://github.com/Sysadmins-CENIA/Cluster-IALab-Docs/wiki`

El código debe ejecutarse utilizando SLURM con la partición `hpc-iic3533`. En cada script SLURM, es necesario:

- Activar el entorno conda: `source ~/miniconda3/bin/activate kmeans_env`
- Configurar número de nodos (`--nodes`)
- Configurar número de tareas MPI (`--ntasks`)
- Configurar CPUs por tarea para threads Numba (`--cpus-per-task`)
- Configurar variable de entorno: `export NUMBA_NUM_THREADS=$SLURM_CPUS_PER_TASK`

Ejemplo de script SLURM:

```
#!/bin/bash
#SBATCH --job-name=kmeans_mpi
#SBATCH --partition=hpc-iic3533
#SBATCH --ntasks=8
#SBATCH --cpus-per-task=2
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --time=00:15:00

# Activar entorno conda
source ~/miniconda3/bin/activate kmeans_env

# Configurar threads Numba
export NUMBA_NUM_THREADS=$SLURM_CPUS_PER_TASK

# Ejecutar
mpirun -n 8 python kmeans_distributed.py
```

(c) [**30pts**] Probar **cinco configuraciones diferentes** de paralelización híbrida:

(i) **Configuración 0:** 1 proceso MPI $\times$ $\{1, 2, 4, 8\}$ threads Numba = $\{1, 2, 4, 8\}$ CPUs.

(ii) **Configuración 1:** 2 procesos MPI $\times$ $\{1, 2, 4, 8\}$ threads Numba = $\{2, 4, 8, 16\}$ CPUs.

(iii) **Configuración 2:** 4 procesos MPI $\times$ $\{1, 2, 4\}$ threads Numba = $\{4, 8, 16\}$ CPUs.

(iv) **Configuración 3:** 8 procesos MPI $\times$ $\{1, 2\}$ threads Numba = $\{8, 16\}$ CPUs.

(v) **Configuración 4:** 16 procesos MPI $\times$ 1 thread Numba = 16 CPUs.

Para cada configuración, medir tiempos de ejecución con `MPI.Wtime()` (usar `comm.Barrier()` antes y después). Presentar resultados en tabla comparativa.

(d) [**20pts**] Identificar y analizar:

(i) ¿Cuál de las configuraciones híbridas (item d) presenta mejor rendimiento? Justificar.

(ii) ¿Cuál es el principal cuello de botella del algoritmo?

Especificaciones técnicas:

- Tamaño mínimo del problema: $n = 4{,}000{,}000$ muestras, $d = 20$ features.
- El dataset debe generarse de forma distribuida (sin Scatter inicial).
- Todos los experimentos deben ejecutarse en el cluster usando SLURM.
- Precaución con el uso máximo de memoria por usuario.

Entregar: (por Canvas)

- Código fuente: `kmeans_distributed.py`
- Scripts SLURM utilizados.
- Informe (PDF) con gráficos, tablas de resultados y análisis.
- Outputs del cluster.