

Trabajo Computación Paralela y distribuida

Un problema de transporte en un isekai.

Integrantes: Javier Alejandro Augusto Roco
Jaime Orellana Olivares
Carlos Quinteros Gonzalez
Sección: 411
Profesor: Sebastián Salazar Molina
Fecha: Martes 2 de julio del 2025

Índice

Introducción	3
Descripción del ejercicio	4
Técnicas utilizadas	5
Procesamiento de datos con Dask	5
Procesamiento lazy y pipeline de operaciones	6
Reducción y agregación de resultados	7
Procesamiento de conteos masivos	7
Visualización y funciones auxiliares	7
Control del rendimiento	7
Resolución del ejercicio	8

Introducción

En la actualidad, la computación paralela y distribuida constituye un pilar fundamental para la resolución eficiente de problemas que implican grandes volúmenes de datos y cálculos complejos. Desde el modelamiento científico hasta la minería de datos, el procesamiento paralelo permite aprovechar al máximo el poder de cómputo disponible mediante la división del trabajo entre múltiples procesadores o núcleos de ejecución.

El ejercicio presentado se enmarca en este contexto: se nos solicita procesar una vasta base de datos que describa la identidad, residencia y destinos frecuentes de 100 millones de habitantes de un mundo ficticio llamado Eldoria. El volumen de información, sumado a la complejidad de las consultas —cálculo de estadísticas demográficas, análisis de movilidad y estratificación social—, hace inviable una solución puramente secuencial si se desea alcanzar tiempos de respuesta razonables.

Para abordar este desafío, se aplican principios de programación paralela, que consisten en dividir el problema en subtarefas que pueden resolverse simultáneamente, y en algunos casos distribuir la carga de trabajo entre distintos hilos o procesos. Entre las técnicas utilizadas destacan la paralelización de datos (data parallelism), donde los datos son fragmentados y procesados en bloques independientes, y la paralelización funcional (task parallelism), que permite ejecutar funciones diferentes en paralelo sobre la información.

El desarrollo de esta solución no sólo implica la correcta implementación de algoritmos paralelos, sino también el manejo eficiente de estructuras de datos compartidas, la sincronización entre procesos y la minimización de cuellos de botella que puedan degradar el rendimiento. Así, este proyecto no es sólo un ejercicio académico, sino un ejemplo representativo de los retos que enfrentan los sistemas modernos en la era del procesamiento masivo de datos.

En las siguientes secciones, se detalla la estrategia de paralelización, los lenguajes y bibliotecas utilizados, las decisiones de diseño y los aspectos clave que permitieron transformar un problema computacionalmente costoso en una solución escalable y eficiente.

Descripción del ejercicio

La solución desarrollada aborda el problema planteado mediante un enfoque de procesamiento paralelo orientado a datos, combinando técnicas de paralelización funcional cuando es necesario aplicar distintas operaciones sobre la misma partición de datos. Este enfoque permite procesar grandes volúmenes de información de manera eficiente, aprovechando la capacidad de múltiples núcleos de CPU.

El programa se estructura en los siguientes componentes principales:

Carga y partición de datos

- Se implementa un mecanismo de lectura eficiente del archivo de texto delimitado por punto y coma (;), considerando que contiene millones de registros.
- Tras la carga inicial, los datos se dividen en particiones de tamaño equilibrado, que se distribuyen entre los distintos hilos o procesos de ejecución. Esta división permite que cada trabajador procese su bloque de registros de manera autónoma.

Procesamiento paralelo de registros

- Cada partición se procesa en paralelo para extraer los campos requeridos:
 - Identificador único.
 - .- Especie.
 - .- Género.
 - .- Fecha de nacimiento.
 - .- Código postal de origen.
 - .- Código postal de destino.
 - En esta fase, se calculan de manera concurrente:
 - .- Conteo por estrato social.
 - .- Cálculo de la edad actual del ciudadano.
 - .- Clasificación de edades por rangos definidos (menor de 18, 18–35, etc.).
 - .- Conteo de viajes entre origen y destino.

Técnicas utilizadas

Procesamiento de datos con Dask

La principal técnica aplicada en este programa es el uso de la biblioteca Dask, una herramienta de *computación paralela* y *procesamiento distribuido* en Python.

Dask permite manejar grandes volúmenes de datos que no caben en memoria RAM (out-of-core computing), dividiendo el dataset en *particiones* y procesándolas en paralelo en múltiples hilos o procesos.

Las características clave de Dask que se están usando aquí son:

- **Dask DataFrame (dd.read_csv)**
 - .- Equivalente a un pandas DataFrame, pero internamente se compone de muchos DataFrames de pandas más pequeños (peticiones).
 - .- Cada partición puede procesarse independientemente en distintos workers.
 - .- Definimos **blocksize="256 MB"** que indica el tamaño máximo que puede tener cada bloque de lectura, lo que controla cuánta RAM usa cada partición.
- **Persistencia**
 - .- El método **.persist()** guarda en memoria los resultados parciales de las operaciones previas (en lugar de recalcularlos cada vez), mejorando eficiencia.
 - compute()
 - .- Al final de la cadena de transformaciones diferidas (*lazy evaluation*), se invoca **.compute()** para materializar el resultado.
 - .- Este es el momento en que Dask reparte la carga de trabajo y ejecuta las tareas en paralelo.

Scheduler multi-hilo

```
dask.config.set(scheduler='threads', num_workers=2)
```

Esto configura el scheduler de Dask para usar un planificador basado en hilos, con dos *workers* paralelos.

- Scheduler de hilos (**threads**): Cada tarea se ejecuta en un *thread* del mismo proceso.
- Ventajas: Bajo costo de creación y comunicación entre hilos, adecuado para operaciones principalmente de I/O

Procesamiento *lazy* y *pipeline* de operaciones

```
df = dd.read_csv(...)
```

```
df["EDAD"] = ...
```

```
df = df.dropna(...)
```

```
resumen = df.groupby(...).size().reset_index()
```

```
resumen = resumen.rename(...).persist()
```

- Estas operaciones se planifican en un grafo de dependencias.
- No se ejecutan inmediatamente, sino que se guardan como un *pipeline* de tareas.
- Cuando se llama `.compute()` o `.persist()`, Dask:
 1. Divide el grafo en bloques de trabajo.
 2. Los reparte entre los *workers* en paralelo.
 3. Combina los resultados parciales.

Reducción y agregación de resultados

La agregación de estadísticas (promedio, mediana, conteos) utiliza operaciones de reducción:

- **Reducción simple:** `groupby().size()`, `groupby().sum()`
- **Aplicaciones personalizadas con `apply()`:**
 1. `edad_promedio` calcula la media ponderada por cantidad.
 2. `mediana_ponderada()` ordena y computa la mediana acumulada.

Procesamiento de conteos masivos

`top_10000_pueblos_con_mas_viajes()`

- **Map:** Dask lee en paralelo las columnas de origen y destino (`CP ORIGEN` y `CP DESTINO`).
- **Concatenación:** Se unen ambas columnas en una serie común (`pueblos`).
- **Reduce:** Se realiza `value_counts()` para contar ocurrencias de cada poblado.
- **Compute:** Materializa los conteos en un DataFrame pandas.
- **Sort y exportación:** Ordena y guarda el top 10.000 en CSV.

Visualización y funciones auxiliares

Aunque la visualización con `matplotlib` no es paralela, el preprocesamiento de los datos sí lo es:

- Antes de graficar las pirámides, se computan los datos de población por edad y género en Dask.
- Solo en la fase final (cuando se materializan los resultados) se transfieren a pandas.

Control del rendimiento

`imprimir_tiempo()` ,Permite medir tiempos de ejecución total y parciales, un aspecto crítico en entornos de procesamiento distribuido.

Resolución del ejercicio

¿Cuántas personas pertenecen a cada estrato social?

```
conteo = df["CP ORIGEN"].str[0].value_counts().compute().sort_index()

print("\n¿Cuántas personas por estrato social?")

for estrato, cantidad in conteo.items():

    print(f" Estrato {estrato}: {cantidad:,} personas")
```

Qué hace:

- Toma el primer dígito del código postal (CP ORIGEN), que representa el estrato social.
- Cuenta cuántos registros hay de cada estrato.
- Imprime la cantidad de personas por estrato.

¿Qué porcentaje de la población pertenece a cada estrato social?

```
total = conteo.sum()

print("\n¿Qué porcentaje representa cada estrato?")

for estrato, cantidad in conteo.items():

    porcentaje = (cantidad / total) * 100

    print(f" Estrato {estrato}: {porcentaje:.2f}%")
```

Qué hace:

- Suma todas las personas (**total**).
- Calcula el porcentaje de cada estrato sobre el total.
- Muestra el porcentaje por estrato.

¿Cuál es la edad promedio según cada especie y género?

```
edad_promedio = resumen_pd.groupby(["ESPECIE", "GENERO"]).apply(

    lambda g: (g["EDAD"] * g["CANTIDAD"]).sum() / g["CANTIDAD"].sum()

).round(2)

print("\nEdad Promedio por Especie y Género:")

print(edad_promedio)
```

Qué hace:

- Agrupa por especie y género.

- Calcula el promedio ponderado de la edad ($\text{edad} \times \text{cantidad} / \text{total}$).
- Redondea a 2 decimales.
- Imprime el promedio.

¿Cuál es la edad mediana según cada especie y género?

```
edad_mediana=resumen_pd.groupby(["ESPECIE","GENERO"]).apply(media_n_ponderada)
```

```
print("\nEdad Mediana por Especie y Género:")
```

```
print(edad_mediana)
```

Qué hace:

- Aplica la función `media_n_ponderada()` a cada grupo.
- La función ordena las edades y calcula la mediana acumulada según cantidad.

Función usada:

```
def media_n_ponderada(grupo):  
  
    grupo = grupo.sort_values("EDAD")  
  
    grupo["ACUM"] = grupo["CANTIDAD"].cumsum()  
  
    total = grupo["CANTIDAD"].sum()  
  
    return grupo.loc[grupo["ACUM"] >= total / 2, "EDAD"].iloc[0]
```

¿Qué proporción de la población tiene menos de 18 años, entre 18–35, 36–60 y más de 60 según especie y género?

`proporcion_por_rango_etario(resumen_pd)`

Qué hace:

- Clasifica cada edad en un rango etario con la función interna `categorizar_edad()`.
- Agrupa por especie, género y rango etario.
- Calcula el porcentaje sobre el total por especie y género.

Fragmento clave de la función:

```
def categorizar_edad(edad):
```

```
    if edad < 18:
```

```
        return "Menor de 18"
```

```
    elif edad <= 35:
```

```
        return "18-35"
```

```
    elif edad <= 60:
```

```
        return "36-60"
```

```
    else:
```

```
        return "60+"
```

Se imprime por pantalla un resumen porcentual por especie y género.

¿Cuál es la pirámide de edades de la población según especie y género?

`graficar_piramides(resumen_pd)`

Qué hace:

- Filtra y agrupa las cantidades de hombres y mujeres por edad y especie.
- Genera un gráfico de barras horizontales con `matplotlib`.
 - Lado izquierdo: hombres (valores negativos).
 - Lado derecho: mujeres.

Salida:

Muestra un gráfico interactivo en pantalla por cada especie.

¿Cuál es el índice de dependencia?

`calcular_indice_dependencia_total(resumen_pd)`

Qué hace:

- Cuenta cuántas personas tienen:
 - Menos de 15 años (`menores_15`).
 - Más de 64 años (`mayores_64`).
 - Entre 15 y 64 años (`edad_trabajo`).

Donde:

$$\text{Índice} = (\text{menores} + \text{mayores}) / \text{edad_trabajo} \times 100$$

Muestra el porcentaje.

Determine los 10.000 poblados con más viajes

`top_10000_pueblos_con_mas_viajes()`

Qué hace:

- Lee en paralelo las columnas **CP ORIGEN** y **CP DESTINO**.
- Combina ambas listas de códigos postales en una sola columna (**poblado_id**).
- Cuenta cuántas veces aparece cada poblado.
- Ordenar por frecuencia descendente.
- Guarda en el archivo **top_pueblos.csv**.
- Muestra los 10 principales.

Código

```
proyectorfinal.py 4 ●
C: > Users > joo21 > Downloads > proyectorfinal.py > top_10000_pueblos_con_mas_viajes
1  # Importación de librerías principales
2  import dask
3  import dask.dataframe as dd
4  from datetime import datetime
5  import pandas as pd
6  import matplotlib.pyplot as plt
7  import time
8
9  # Configuración de Dask para usar hilos con 2 workers
10 dask.config.set(scheduler='threads', num_workers=2)
11
12 # ===== Funciones Auxiliares =====
13
14 # Mide e imprime el tiempo transcurrido
15 def imprimir_tiempo(mensaje, inicio):
16     print(f"\n{mensaje}: {time.time() - inicio:.2f} segundos")
17
18 # Calcula la mediana ponderada de edades en un grupo
19 def mediana_ponderada(grupo):
20     # Ordenar por edad
21     grupo = grupo.sort_values("EDAD")
22     # Acumular cantidad por edad
23     grupo["ACUM"] = grupo["CANTIDAD"].cumsum()
24     total = grupo["CANTIDAD"].sum()
25     # Buscar el punto donde el acumulado alcanza la mitad del total
26     return grupo.loc[grupo["ACUM"] >= total / 2, "EDAD"].iloc[0]
27
```

```
proyectofinal.py 4
C: > Users > joo21 > Downloads > proyectofinal.py > top_10000_pueblos_con_mas_viajes
19 def mediana_ponderada(grupo):
20     return grupo.loc[grupo["ACUM"] >= total / 2, "EDAD"].iloc[0]
21
22 # Calcula y muestra la proporción de población por rango etario
23 def proporcion_por_rango_etario(df):
24     # Función interna para categorizar la edad
25     def categorizar_edad(edad):
26         if edad < 18:
27             return "Menor de 18"
28         elif edad <= 35:
29             return "18-35"
30         elif edad <= 60:
31             return "36-60"
32         else:
33             return "60+"
34
35     # Crear columna con rango etario
36     df["RANGO_ETARIO"] = df["EDAD"].apply(categorizar_edad)
37     # Total de población por especie y género
38     totales = df.groupby(["ESPECIE", "GENERO"])["CANTIDAD"].sum().rename("TOTAL")
39     # Conteo por especie, género y rango
40     rangos = df.groupby(["ESPECIE", "GENERO", "RANGO_ETARIO"])["CANTIDAD"].sum().rename("CUENTA")
41     # Combinar resultados
42     resultado = pd.merge(rangos.reset_index(), totales.reset_index(), on=["ESPECIE", "GENERO"])
43     # Calcular porcentaje
44     resultado["%"] = (resultado["CUENTA"] / resultado["TOTAL"] * 100).round(2)
45
46     # Imprimir resultados por especie y género
47     print("\nProporción por rango etario (%):")
48     for (especie, genero), grupo in resultado.groupby(["ESPECIE", "GENERO"]):
49         print(f"\n{especie} - {genero}")
50         for _, fila in grupo.iterrows():
51             print(f"    {fila['RANGO_ETARIO']}: {fila['%']}")
52
53
54
55
56
57
58
```

```
proyectorfinal.py 4
C: > Users > joo21 > Downloads > proyectorfinal.py > top_10000_pueblos_con_mas_viajes
29 def proporcion_por_rango_etario(df):
57     print(f" {fila['RANGO_ETARIO']}: {fila['%']}%")
58
59 # Genera pirámides de edad por especie
60 def graficar_piramides(df):
61     df = df.copy()
62     df["EDAD"] = df["EDAD"].astype(int)
63     especies = df["ESPECIE"].unique()
64
65     for especie in especies:
66         # Filtrar datos por especie
67         datos = df[df["ESPECIE"] == especie]
68         edades = sorted(datos["EDAD"].unique())
69
70         # Conteo por género
71         hombres = datos[datos["GENERO"] == "MACHO"].groupby("EDAD")["CANTIDAD"].sum()
72         mujeres = datos[datos["GENERO"] == "HEMBRA"].groupby("EDAD")["CANTIDAD"].sum()
73
74         # Preparar valores negativos (hombres) y positivos (mujeres)
75         valores_h = [-hombres.get(e, 0) for e in edades]
76         valores_m = [mujeres.get(e, 0) for e in edades]
77
78         # Crear gráfico
79         plt.figure(figsize=(10, 6))
80         plt.barh(edades, valores_h, label="Masculino", color="blue")
81         plt.barh(edades, valores_m, label="Femenino", color="pink")
82         plt.xlabel("Población")
83         plt.ylabel("Edad")
84         plt.title(f"PIRÁMIDE DE EDAD - {especie}")
85         plt.legend()
86         plt.tight_layout()
87         plt.show()
88
```



```
proyectofinal.py 4
C: > Users > joo21 > Downloads > proyectofinal.py > top_10000_pueblos_con_mas_viajes

88
89 # Calcula el índice de dependencia de la población
90 def calcular_indice_dependencia_total(df):
91     menores_15 = df[df["EDAD"] < 15]["CANTIDAD"].sum()
92     mayores_64 = df[df["EDAD"] > 64]["CANTIDAD"].sum()
93     edad_trabajo = df[(df["EDAD"] >= 15) & (df["EDAD"] <= 64)]["CANTIDAD"].sum()
94
95     if edad_trabajo == 0:
96         print("\nNo hay población en edad de trabajar.")
97         return
98
99     indice = ((menores_15 + mayores_64) / edad_trabajo) * 100
100     print(f"\nÍndice de Dependencia (total): {indice:.2f}%")
101
102 # Procesa los 10.000 poblados con más viajes
103 def top_10000_pueblos_con_mas_viajes():
104     print("\nProcesando los 10.000 pueblos con más viajes...")
105
106     # Lectura paralela de columnas de origen y destino
107     viajes = dd.read_csv(
108         "eldoria.csv",
109         sep=';',
110         quotechar='"',
111         dtype=str,
112         usecols=["CP ORIGEN", "CP DESTINO"],
113         blocksize="256MB"
114     )
115
116     # Concatenar ambos campos en una sola columna
117     pueblos = dd.concat([viajes["CP ORIGEN"], viajes["CP DESTINO"]]).rename("poblado_id")
118     # Conteo de ocurrencias por poblado
119     conteo = pueblos.value_counts().compute().reset_index()
120     conteo.columns = ["poblado_id", "frecuencia"]
121
```

```
proyectorfinal.py 4
C: > Users > joo21 > Downloads > proyectorfinal.py > top_10000_pueblos_con_mas_viajes
103 def top_10000_pueblos_con_mas_viajes():
120     conteo.columns = [ 'poblado_id', 'frecuencia' ]
121
122     # Seleccionar top 10.000
123     top_10000 = conteo.sort_values(by="frecuencia", ascending=False).head(10000)
124     # Guardar resultados en CSV
125     top_10000.to_csv("top_pueblos.csv", index=False)
126
127     # Mostrar los 10 primeros
128     print("\n--- Top 10.000 Pueblos con más Viajes ---")
129     print(top_10000.head(10))
130
131     # ===== PROCESO PRINCIPAL =====
132
133     if __name__ == "__main__":
134         inicio = time.time()
135
136         # Lectura paralela de datos principales
137         df = dd.read_csv(
138             "eldoria.csv",
139             sep=';',
140             quotechar='\"',
141             dtype=str,
142             usecols=["CP ORIGEN", "ESPECIE", "GENERO", "FECHA NACIMIENTO"],
143             blocksize="256MB"
144         )
145
146         # Conteo por estrato social (primer dígito de CP ORIGEN)
147         conteo = df["CP ORIGEN"].str[0].value_counts().compute().sort_index()
148         print("\n¿Cuántas personas por estrato social?")
149         for estrato, cantidad in conteo.items():
150             print(f"  Estrato {estrato}: {cantidad:,} personas")
151
```

```
proyectofinal.py 4
C: > Users > joo21 > Downloads > proyectofinal.py > top_10000_pueblos_con_mas_viajes

151
152 # Porcentaje por estrato social
153 total = conteo.sum()
154 print("\n¿Qué porcentaje representa cada estrato?")
155 for estrato, cantidad in conteo.items():
156     porcentaje = (cantidad / total) * 100
157     print(f"  Estrato {estrato}: {porcentaje:.2f}%")
158
159 # Convertir fecha de nacimiento en datetime
160 df["FECHA NACIMIENTO"] = dd.to_datetime(df["FECHA NACIMIENTO"].str.split("T").str[0], errors="coerce")
161 hoy = datetime.now()
162 # Calcular edad en años
163 df["EDAD"] = ((hoy - df["FECHA NACIMIENTO"]).dt.days // 365)
164
165 # Eliminar registros con valores faltantes
166 df = df.dropna(subset=["ESPECIE", "GENERO", "EDAD"])
167
168 # Agrupación por especie, género y edad
169 resumen = df.groupby(["ESPECIE", "GENERO", "EDAD"]).size().reset_index()
170 resumen = resumen.rename(columns={0: "CANTIDAD"}).persist()
171 # Convertir en pandas DataFrame
172 resumen_pd = resumen.compute()
173 # Normalizar nombres de género
174 resumen_pd["GENERO"] = resumen_pd["GENERO"].str.upper().str.strip()
175
176 # Mostrar tiempo total
177 imprimir_tiempo("Tiempo total Dask", inicio)
178
179 # Cálculo de edad promedio
180 edad_promedio = resumen_pd.groupby(["ESPECIE", "GENERO"]).apply(
181     lambda g: (g["EDAD"] * g["CANTIDAD"]).sum() / g["CANTIDAD"].sum()
182 ).round(2)
183 # Cálculo de edad mediana
184 edad_mediana = resumen_pd.groupby(["ESPECIE", "GENERO"]).apply(mediana_ponderada)
185
```

```
proyectofinal.py 4
C: > Users > joo21 > Downloads > proyectofinal.py > top_10000_pueblos_con_mas_viajes
180     edad_promedio = resumen_pd.groupby(["ESPECIE", "GENERO"]).apply(
181         lambda g: (g["EDAD"] * g["CANTIDAD"]).sum() / g["CANTIDAD"].sum()
182     ).round(2)
183     # Cálculo de edad mediana
184     edad_mediana = resumen_pd.groupby(["ESPECIE", "GENERO"]).apply(mediana_ponderada)
185
186     # Imprimir resultados de edades
187     print("\nEdad Promedio por Especie y Género:")
188     print(edad_promedio)
189     print("\nEdad Mediana por Especie y Género:")
190     print(edad_mediana)
191
192     # Calcular proporción por rango etario
193     proporcion_por_rango_etario(resumen_pd)
194
195     # Calcular índice de dependencia
196     calcular_indice_dependencia_total(resumen_pd)
197
198     # Graficar pirámides de edad
199     graficar_piramides(resumen_pd)
200
201     # Obtener top 10.000 pueblos
202     top_10000_pueblos_con_mas_viajes()
203
```

Conclusiones

En este proyecto logramos desarrollar un programa que permite procesar de forma eficiente un conjunto de datos muy grande relacionado con la población y los viajes en el Reino de Eldoria. Para ello, utilizamos la librería Dask, que facilitó la paralelización de las tareas y la gestión de la información en particiones que se procesaron en varios hilos de manera simultánea.

Durante el trabajo, implementamos distintas técnicas de procesamiento paralelo:

- Dividimos los datos en bloques y aplicamos operaciones de conteo, agrupación y reducción de forma distribuida.
- Utilizamos evaluación diferida para optimizar el rendimiento, ejecutando todo el flujo de operaciones sólo cuando era necesario con `.compute()`.
- Calculamos métricas complejas, como el promedio y la mediana ponderada de edades, además de proporciones por rango etario y el índice de dependencia, de forma completamente automatizada.
- Finalmente, generamos gráficos de pirámides de edad que ayudan a visualizar mejor la distribución poblacional, y también identificamos los 10.000 poblados con más viajes registrados.

Una de las principales ventajas del enfoque que adoptamos es que permite manejar datos que serían muy difíciles de procesar con pandas puro, sin requerir infraestructuras muy costosas. Además, la sintaxis de Dask es bastante similar a pandas, lo que facilita la transición.

En conclusión, consideramos que el proyecto cumple con todos los requerimientos planteados en el enunciado y que el uso de procesamiento paralelo fue fundamental para lograr tiempos de respuesta aceptables. Este trabajo también nos permitió comprender de manera práctica cómo se aplican los conceptos de computación paralela y distribuida en problemas reales donde el volumen de datos es un desafío principal.