Algoritmos paralelos

Los algoritmos paralelos son una categoría de algoritmos diseñados para aprovechar las capacidades de procesamiento paralelo de las computadoras modernas, permitiendo la ejecución simultánea de múltiples tareas. A medida que los procesadores multinúcleo y los sistemas de computación distribuida se han vuelto más comunes, el desarrollo y la implementación de algoritmos paralelos se ha vuelto esencial para mejorar el rendimiento y la eficiencia de muchas aplicaciones.

Conceptos fundamentales

El campo de los algoritmos paralelos es esencial para el avance de machine learning (ML) y deep learning (DL). La capacidad de procesar grandes volúmenes de datos y ejecutar complejas operaciones matemáticas de manera eficiente es crucial para el desarrollo y la implementación de modelos de aprendizaje automático y profundo.

Descomposición de datos

En ML y DL, la descomposición de datos es una técnica fundamental. Consiste en dividir grandes conjuntos de datos en fragmentos más pequeños que pueden ser procesados simultáneamente por múltiples unidades de procesamiento. Esta técnica es particularmente útil en la fase de entrenamiento de modelos de DL, donde los datos de entrenamiento pueden ser distribuidos entre diferentes GPUs o nodos de un clúster de computación.

Descomposición de tareas

La descomposición de tareas implica dividir un algoritmo en partes más pequeñas, cada una de las cuales puede ejecutarse en paralelo. En DL, esto puede aplicarse a la arquitectura del modelo, donde diferentes capas de una red neuronal profunda pueden ser computadas simultáneamente, o en algoritmos de optimización donde múltiples instancias de una función objetivo pueden evaluarse en paralelo.

Granularidad y overhead

La granularidad se refiere al tamaño de las tareas individuales en las que se divide un problema. En el contexto de ML y DL, una granularidad adecuada puede influir significativamente en el rendimiento del algoritmo paralelo.

- Granularidad fina: Implica dividir el problema en tareas muy pequeñas. Aunque esto puede aprovechar mejor la paralelización, el overhead de comunicación entre tareas puede ser alto, lo que reduce la eficiencia general.
- Granularidad gruesa: Divide el problema en tareas más grandes, reduciendo el overhead de comunicación. Esto es común en sistemas donde las tareas individuales son relativamente independientes y no requieren comunicación frecuente.

Sincronización y comunicación

- Sincronización: La sincronización asegura que las tareas paralelas se coordinen adecuadamente, especialmente cuando comparten recursos. En ML y DL, esto es crucial para operaciones como la actualización de parámetros del modelo en algoritmos de entrenamiento distribuidos.

- Locks y semáforos: Se utilizan para controlar el acceso a recursos compartidos, garantizando que no se produzcan condiciones de carrera.
- Barreras: Aseguran que todas las tareas lleguen a un cierto punto antes de continuar, lo que es esencial en el entrenamiento sincronizado de redes neuronales distribuidas.

Comunicación

La comunicación eficiente entre tareas es esencial para el rendimiento de algoritmos paralelos en ML y DL. Existen dos modelos principales:

- Memoria compartida: Las tareas acceden a un espacio de memoria común. Este modelo es eficiente pero puede ser problemático debido a las condiciones de carrera.
- Pasaje de mensajes: Las tareas se comunican enviando y recibiendo mensajes. Este modelo es más adecuado para sistemas distribuidos, como el entrenamiento de modelos en clústeres de computación.

Técnicas de paralelización en machine learning y deep learning

Paralelización de datos

La paralelización de datos es ampliamente utilizada en DL, donde los datos de entrenamiento se dividen entre múltiples GPUs o nodos. Cada unidad de procesamiento entrena una copia del modelo en su subconjunto de datos y luego combina los gradientes calculados para actualizar los parámetros del modelo global.

Paralelización de modelos

En redes neuronales muy grandes, la paralelización de modelos divide el modelo en diferentes partes que se distribuyen entre varias GPUs. Por ejemplo, diferentes capas de una red pueden ser asignadas a diferentes GPUs, permitiendo que cada GPU procese su parte del modelo en paralelo.

Pipeline de datos

El pipeline de datos es una técnica donde las operaciones de preprocesamiento y entrenamiento se solapan. Mientras una parte de los datos está siendo procesada, otra parte está siendo utilizada para entrenar el modelo. Esto maximiza el uso de recursos y reduce el tiempo de inactividad.

Entrenamiento distribuido

El entrenamiento distribuido es una técnica donde múltiples nodos colaboran para entrenar un modelo. Existen dos enfoques principales:

- Entrenamiento sincronizado: Todos los nodos deben completar su cálculo antes de que los parámetros del modelo sean actualizados.
- Entrenamiento asincronizado: Los nodos actualizan los parámetros del modelo de manera independiente, lo que puede conducir a una convergencia más rápida pero también introduce desafíos de coherencia.

Algoritmos paralelos en machine learning y deep learning

Paralelización de datos (data parallelism)

La paralelización de datos es una técnica ampliamente utilizada en ML y DL, donde los datos se dividen en fragmentos y se procesan simultáneamente en múltiples unidades de procesamiento. Esta técnica es especialmente efectiva en el entrenamiento de modelos de DL, donde grandes conjuntos de datos se distribuyen entre varias GPUs o nodos de un clúster.

Ejemplos:

- Entrenamiento de redes neuronales: Los datos de entrenamiento se dividen entre varias GPUs, cada una de las cuales calcula los gradientes para su subconjunto de datos. Luego, los gradientes se combinan para actualizar los parámetros del modelo global.
- Gradient Boosting Machines (GBM): En GBM, los árboles se construyen secuencialmente, pero los cálculos de gradientes y actualizaciones pueden paralelizarse.

Paralelización de modelos (model parallelism) En redes neuronales muy grandes, la paralelización de modelos divide el modelo entre múltiples GPUs. Cada GPU se encarga de una parte del modelo, lo que permite manejar arquitecturas que de otro modo no cabrían en la memoria de una sola GPU.

Ejemplos:

- Redes neuronales convolucionales (CNNs): Las diferentes capas de una CNN pueden ser asignadas a distintas GPUs para procesarse en paralelo.
- Redes neuronales recursivas (RNNs): Las secuencias de datos largas se pueden dividir entre múltiples GPUs, donde cada GPU procesa una subsecuencia.

Pipeline parallelism

El pipeline parallelism divide las operaciones de preprocesamiento y entrenamiento en una serie de etapas secuenciales, donde cada etapa se ejecuta en paralelo. Esto maximiza la utilización de recursos y minimiza el tiempo de inactividad.

Ejemplo:

- Transformers: En modelos como BERT, diferentes capas de autoatención pueden procesarse en paralelo usando pipeline parallelism, mejorando la eficiencia del entrenamiento.

Algoritmos de optimización paralelos

Los algoritmos de optimización paralelos son esenciales para acelerar el entrenamiento de modelos complejos en ML y DL. Métodos como el descenso de gradiente estocástico (SGD) se pueden paralelizar fácilmente.

Ejemplos:

- SGD paralelo: Los datos se dividen entre varios nodos, cada uno calculando los gradientes de forma independiente. Luego, los gradientes se agregan para actualizar los parámetros del modelo global.

- Métodos de Monte Carlo: Los métodos de Monte Carlo se pueden paralelizar evaluando múltiples muestras simultáneamente, útil en optimización estocástica y aprendizaje por refuerzo.

Algoritmos paralelos en sistemas distribuidos

MapReduce

MapReduce es un modelo de programación que permite el procesamiento y la generación de grandes conjuntos de datos mediante un modelo de pares clave-valor. Este modelo distribuye el trabajo en dos fases principales: Map y Reduce, facilitando la paralelización de tareas de procesamiento de datos en un clúster.

Ejemplo:

- Procesamiento de Logs: Los logs se dividen en fragmentos y se distribuyen entre varios nodos para el procesamiento paralelo. Los resultados parciales se combinan en la fase de reducción para generar la salida final.

Clustering (cluster computing)

La computación en grupos utiliza múltiples nodos de un clúster para ejecutar tareas paralelas. Frameworks como Apache Hadoop y Apache Spark permiten la ejecución eficiente de tareas distribuidas.

Ejemplos:

- Análisis de grandes datos (big data analytics): Herramientas como Spark permiten procesar y analizar grandes volúmenes de datos en paralelo, utilizando operaciones como transformaciones y acciones distribuidas.

Programación de pasaje de mensajes (Message Passing Interface, MPI)

MPI es un estándar para la programación paralela que permite a los desarrolladores escribir programas que pueden ejecutarse en sistemas distribuidos, facilitando la comunicación entre nodos mediante el paso de mensajes.

Ejemplo:

- Simulaciones científicas: Las simulaciones que requieren grandes cantidades de cálculo pueden distribuirse entre múltiples nodos utilizando MPI para comunicarse y sincronizarse.

Algoritmos distribuidos

Los algoritmos distribuidos están diseñados para sistemas donde los componentes son nodos autónomos que interactúan mediante comunicación de red. Estos algoritmos se utilizan para resolver problemas donde la información y el control están distribuidos.

Ejemplos:

- Algoritmos de Consenso: En sistemas distribuidos, los algoritmos de consenso como Paxos y Raft aseguran que todos los nodos acuerden un valor único, crucial para la consistencia de los datos.

- Algoritmos de elección de líder: Estos algoritmos determinan un líder entre un grupo de nodos, utilizado en sistemas distribuidos para coordinar actividades.

Algoritmos paralelos en la nube

Funciones como servicio (function as a service, FaaS)

FaaS permite ejecutar fragmentos de código en respuesta a eventos sin la necesidad de gestionar servidores. Las plataformas en la nube como AWS Lambda, Google Cloud Functions y Azure Functions facilitan la paralelización y el escalado automático de tareas.

Ejemplo:

- Procesamiento de imágenes: Las imágenes subidas a un almacenamiento en la nube pueden desencadenar funciones Lambda que procesan las imágenes en paralelo, aplicando filtros o análisis.
- **Infraestructura como servicio (IaaS)**

IaaS proporciona recursos de computación virtualizados que pueden ser escalados según las necesidades del usuario. Esto permite la ejecución paralela de algoritmos de ML y DL en instancias de máquinas virtuales.

Ejemplo:

- Entrenamiento de modelos en EC2: Utilizando instancias EC2 con múltiples GPUs, los modelos de DL pueden ser entrenados en paralelo, reduciendo significativamente el tiempo de entrenamiento.

Kubernetes y contenedores

Kubernetes es una plataforma de orquestación de contenedores que permite gestionar y escalar aplicaciones en contenedores. Los contenedores proporcionan un entorno aislado y consistente para la ejecución de aplicaciones, facilitando la paralelización y el despliegue en la nube.

Ejemplo:

- Microservicios: Los microservicios que componen una aplicación se despliegan en contenedores, y Kubernetes gestiona la distribución y escalado de estos contenedores, permitiendo una ejecución paralela eficiente.

Plataformas de machine learning en la nube

Plataformas como AWS SageMaker, Google AI Platform y Azure Machine Learning proporcionan herramientas para la construcción, entrenamiento y despliegue de modelos de ML y DL en la nube. Estas plataformas soportan la paralelización automática de tareas.

Ejemplo:

- Entrenamiento distribuido en SageMaker: SageMaker distribuye automáticamente el entrenamiento de modelos en múltiples nodos y GPUs, optimizando el uso de recursos y acelerando el proceso de entrenamiento.

Almacenamiento y procesamiento de datos en la nube

El almacenamiento y procesamiento de grandes volúmenes de datos es fundamental para ML y DL. Servicios como Amazon S3, Google Cloud Storage y Azure Blob Storage permiten almacenar datos de manera distribuida y acceder a ellos de forma paralela.

Ejemplo:

- Procesamiento de datos en BigQuery: Google BigQuery permite la ejecución de consultas SQL en grandes conjuntos de datos almacenados en la nube, procesando los datos en paralelo para obtener resultados rápidamente.

Implementaciones

Entrenamiento de redes neuronales con datos divididos entre varias GPUs

El entrenamiento de redes neuronales a gran escala se puede acelerar significativamente mediante la paralelización de datos. En este enfoque, el conjunto de datos de entrenamiento se divide entre varias GPUs. Cada GPU calcula los gradientes para su subconjunto de datos y, posteriormente, estos gradientes se combinan para actualizar los parámetros del modelo global. Este método se conoce como Data Parallelism.

Implementación:

- Los datos de entrada se dividen en fragmentos y se distribuyen entre las GPUs disponibles.
- Cada GPU realiza el cálculo de gradientes en su subconjunto de datos.
- Los gradientes calculados por cada GPU se combinan (generalmente mediante una operación de reducción, como all-reduce) para actualizar los parámetros del modelo global.

Gradient Boosting Machines (GBM)

El algoritmo GBM construye árboles de decisión secuencialmente, donde cada árbol intenta corregir los errores cometidos por los árboles anteriores. Sin embargo, los cálculos de gradientes y actualizaciones de los parámetros pueden paralelizarse.

Implementación:

- Inicia con una predicción base.
- Cada árbol se construye en función del gradiente de la pérdida respecto a la predicción actual.
- Aunque la construcción de árboles es secuencial, los cálculos de gradientes y ajustes pueden realizarse en paralelo.

Redes neuronales convolucionales (CNNs)

Las CNNs son particularmente adecuadas para la paralelización debido a la naturaleza independiente de las convoluciones en diferentes filtros y capas. En Model Parallelism, diferentes capas de una CNN pueden ser asignadas a distintas GPUs para procesarse en paralelo.

Implementación:

- Divide las capas de la red entre las GPUs.
- Cada GPU realiza el forward pass de su conjunto de capas.
- Las GPUs calculan los gradientes y los combinan para actualizar los parámetros.

Redes neuronales recursivas (RNNs)

Las RNNs son utilizadas para datos secuenciales y pueden ser paralelizadas mediante la división de secuencias largas entre múltiples GPUs. Cada GPU procesa una subsecuencia, lo que mejora la eficiencia del entrenamiento.

Implementación:

- Divide la secuencia de entrada en subsecuencias.
- Cada GPU procesa su subsecuencia.
- Los resultados de cada subsecuencia se combinan para la actualización del modelo.

Transformers

Los modelos de transformers, como BERT, utilizan múltiples capas de autoatención que pueden ser paralelizadas utilizando Pipeline Parallelism. Esto implica que diferentes capas de autoatención se procesan en paralelo, mejorando la eficiencia del entrenamiento.

Implementación:

- Divide las capas del transformer en etapas.
- Procesa cada etapa en paralelo.
- Asegura que las etapas se sincronizan correctamente.

SGD paralelo

El descenso de gradiente estocástico (SGD) puede paralelizarse dividiendo los datos de entrenamiento entre varios nodos. Cada nodo calcula los gradientes de forma independiente y luego se combinan para actualizar los parámetros del modelo global.

Implementación:

- Los datos de entrenamiento se dividen entre los nodos.
- Cada nodo calcula los gradientes para su subconjunto de datos.
- Los gradientes se combinan para actualizar los parámetros del modelo global.

Métodos de Monte Carlo

Los métodos de Monte Carlo son utilizados en optimización estocástica y aprendizaje por refuerzo. Pueden ser paralelizados evaluando múltiples muestras simultáneamente.

Implementación

- Genera múltiples muestras simultáneamente.
- Evalua las muestras en paralelo.
- Combina los resultados para la actualización del modelo.

Entrenamiento de redes neuronales con datos divididos entre varias GPUs.

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

# Configuración del dispositivo
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
num_gpus = torch.cuda.device_count()

# Modelo de ejemplo
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(784, 256)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(256, 10)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# Creación de un dataset de ejemplo
x = torch.randn(10000, 784)
y = torch.randint(0, 10, (10000,))
dataset = TensorDataset(x, y)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)

# División del modelo entre múltiples GPUs
model = SimpleNN().to(device)
if num_gpus > 1:
    model = nn.DataParallel(model)

optimizer = optim.SGD(model.parameters(), lr=0.01)
criterion = nn.CrossEntropyLoss()

# Entrenamiento del modelo
for epoch in range(10):
    for data, target in dataloader:
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
    print(f"Epoca {epoch+1}, Perdida: {loss.item()}")

import tensorflow as tf
from tensorflow.keras import layers, models, optimizers, losses
```

```
from tensorflow.data import Dataset

# Configuración del dispositivo
device = '/GPU:0' if tf.config.list_physical_devices('GPU') else
'/CPU:0'
num_gpus = len(tf.config.list_physical_devices('GPU'))

# Modelo de ejemplo
class SimpleNN(models.Model):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = layers.Dense(256, activation='relu')
        self.fc2 = layers.Dense(10)

    def call(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        return x

# Creación de un dataset de ejemplo
x = tf.random.normal((10000, 784))
y = tf.random.uniform((10000,), maxval=10, dtype=tf.int32)
dataset = Dataset.from_tensor_slices((x, y)).shuffle(10000).batch(32)

# División del modelo entre múltiples GPUs
with tf.device(device):
    model = SimpleNN()

if num_gpus > 1:
    strategy = tf.distribute.MirroredStrategy()
    with strategy.scope():
        model = SimpleNN()
        optimizer = optimizers.SGD(learning_rate=0.01)
        criterion =
losses.SparseCategoricalCrossentropy(from_logits=True)
else:
    optimizer = optimizers.SGD(learning_rate=0.01)
    criterion = losses.SparseCategoricalCrossentropy(from_logits=True)

# Compilación del modelo
model.compile(optimizer=optimizer, loss=criterion)

# Entrenamiento del modelo
for epoch in range(10):
    for data, target in dataset:
        with tf.GradientTape() as tape:
            output = model(data, training=True)
            loss = criterion(target, output)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients,
```

```
model.trainable_variables))
    print(f"Epoca {epoch+1}, Perdida: {loss.numpy()}")
```

Para ilustrar cómo paralelizar los cálculos de gradientes en GBM, usaremos lightgbm que soporta paralelización de manera nativa.

```
import lightgbm as lgb
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Generación de un dataset de ejemplo
X, y = make_classification(n_samples=10000, n_features=20,
n_classes=2)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)

# Creación del dataset de LightGBM
train_data = lgb.Dataset(X_train, label=y_train)
test_data = lgb.Dataset(X_test, label=y_test)

# Configuración del modelo
params = {
    'objective': 'binary',
    'metric': 'binary_logloss',
    'boosting_type': 'gbdt',
    'num_leaves': 31,
    'learning_rate': 0.05,
    'feature_fraction': 0.9,
    'n_jobs': -1  # Usar múltiples núcleos para la paralelización
}

# Entrenamiento del modelo
bst = lgb.train(params, train_data, valid_sets=[test_data],
num_boost_round=100)
```

Redes Neuronales Convolucionales (CNNs)

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms

# Configuración del dispositivo
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
num_gpus = torch.cuda.device_count()

# Modelo de ejemplo (CNN)
class SimpleCNN(nn.Module):
    def __init__(self):
```

```
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3)
        self.fc1 = nn.Linear(64 * 24 * 24, 128)  # Ajustar las
dimensiones
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)  # Cambia a esta línea para aplanar
correctamente
        x = self.fc1(x)
        x = self.fc2(x)
        return x

# Carga de datos de ejemplo (MNIST)
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
train_dataset = datasets.MNIST('./data', train=True, download=True,
transform=transform)
train_loader = torch.utils.data.DataLoader(train_dataset,
batch_size=64, shuffle=True)

# División del modelo entre múltiples GPUs
model = SimpleCNN().to(device)
if num_gpus > 1:
    model = nn.DataParallel(model)

optimizer = optim.SGD(model.parameters(), lr=0.01)
criterion = nn.CrossEntropyLoss()

# Entrenamiento del modelo
for epoch in range(10):
    for data, target in train_loader:
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
    print(f"Epoca {epoch+1}, Perdida: {loss.item()}")

import tensorflow as tf
from tensorflow.keras import layers, models, optimizers, losses
from tensorflow.keras.datasets import mnist

# Configuración del dispositivo
```

```
device = '/GPU:0' if tf.config.list_physical_devices('GPU') else
'/CPU:0'
num_gpus = len(tf.config.list_physical_devices('GPU'))

# Modelo de ejemplo (CNN)
class SimpleCNN(models.Model):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = layers.Conv2D(32, kernel_size=3,
activation='relu')
        self.conv2 = layers.Conv2D(64, kernel_size=3,
activation='relu')
        self.flatten = layers.Flatten()
        self.fc1 = layers.Dense(128, activation='relu')
        self.fc2 = layers.Dense(10)

    def call(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.fc2(x)
        return x

# Carga de datos de ejemplo (MNIST)
(x_train, y_train), (_, _) = mnist.load_data()
x_train = x_train.reshape(-1, 28, 28, 1).astype('float32') / 255.0
y_train = y_train.astype('int32')

dataset = tf.data.Dataset.from_tensor_slices((x_train,
y_train)).shuffle(10000).batch(64)

# División del modelo entre múltiples GPUs
with tf.device(device):
    model = SimpleCNN()

if num_gpus > 1:
    strategy = tf.distribute.MirroredStrategy()
    with strategy.scope():
        model = SimpleCNN()
        optimizer = optimizers.SGD(learning_rate=0.01)
        criterion =
losses.SparseCategoricalCrossentropy(from_logits=True)
else:
    optimizer = optimizers.SGD(learning_rate=0.01)
    criterion = losses.SparseCategoricalCrossentropy(from_logits=True)

# Compilación del modelo
model.compile(optimizer=optimizer, loss=criterion)
```

```
# Entrenamiento del modelo
for epoch in range(10):
    for data, target in dataset:
        with tf.GradientTape() as tape:
            output = model(data, training=True)
            loss = criterion(target, output)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients,
model.trainable_variables))
    print(f"Epoca {epoch+1}, Perdida: {loss.numpy()}")
```

Transformers

```
import torch
import torch.nn as nn
import torch.optim as optim
from transformers import BertModel, BertTokenizer

# Configuración del dispositivo
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
num_gpus = torch.cuda.device_count()

# Modelo de ejemplo (BERT)
class SimpleBERT(nn.Module):
    def __init__(self):
        super(SimpleBERT, self).__init__()
        self.bert = BertModel.from_pretrained('bert-base-uncased')
        self.fc = nn.Linear(768, 2)  # Suponiendo una tarea de
clasificación binaria

    def forward(self, x):
        x = self.bert(x)[0]
        x = self.fc(x[:, 0, :])
        return x

# Tokenización de ejemplo
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
texts = ["Example sentence fo BERT model."] * 10
inputs = tokenizer(texts, return_tensors='pt', padding=True,
truncation=True)

# División del modelo entre múltiples GPUs
model = SimpleBERT().to(device)
if num_gpus > 1:
    model = nn.DataParallel(model)

optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

# Entrenamiento del modelo
```

```
for epoch in range(10):
    optimizer.zero_grad()
    outputs = model(inputs['input_ids'].to(device))
    loss = criterion(outputs, torch.tensor([1]*10).to(device))
    loss.backward()
    optimizer.step()
    print(f"Epoca{epoch+1}, Entrenamiento: {loss.item()}")

import tensorflow as tf
from transformers import TFBertModel, BertTokenizer
from tensorflow.keras import layers, optimizers, losses

# Configuración del dispositivo
device = '/GPU:0' if tf.config.list_physical_devices('GPU') else
'/CPU:0'
num_gpus = len(tf.config.list_physical_devices('GPU'))

# Modelo de ejemplo (BERT)
class SimpleBERT(tf.keras.Model):
    def __init__(self):
        super(SimpleBERT, self).__init__()
        self.bert = TFBertModel.from_pretrained('bert-base-uncased')
        self.fc = layers.Dense(2)  # Suponiendo una tarea de
clasificación binaria

    def call(self, inputs):
        bert_output = self.bert(inputs)[0]
        cls_output = bert_output[:, 0, :]
        output = self.fc(cls_output)
        return output

# Tokenización de ejemplo
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
texts = ["Example sentence for BERT model."] * 10
inputs = tokenizer(texts, return_tensors='tf', padding=True,
truncation=True)

# División del modelo entre múltiples GPUs
if num_gpus > 1:
    strategy = tf.distribute.MirroredStrategy()
    with strategy.scope():
        model = SimpleBERT()
        optimizer = optimizers.Adam(learning_rate=0.001)
        criterion =
losses.SparseCategoricalCrossentropy(from_logits=True)
else:
    model = SimpleBERT()
    optimizer = optimizers.Adam(learning_rate=0.001)
    criterion = losses.SparseCategoricalCrossentropy(from_logits=True)
```

```
# Compilación del modelo
model.compile(optimizer=optimizer, loss=criterion)

# Etiquetas de ejemplo
labels = tf.constant([1] * 10)

# Entrenamiento del modelo
for epoch in range(10):
    with tf.GradientTape() as tape:
        outputs = model(inputs['input_ids'])
        loss = criterion(labels, outputs)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients,
model.trainable_variables))
    print(f"Epoca {epoch+1}, Entrenamiento: {loss.numpy()}")
```

Para ilustrar cómo paralelizar el SGD, utilizamos `torch.distributed` para implementar el paralelismo de datos.

```
import torch
import torch.distributed as dist
import torch.multiprocessing as mp
from torch.nn.parallel import DistributedDataParallel as DDP
import torch.nn as nn
import os

def setup(rank, world_size):
    os.environ['MASTER_ADDR'] = 'localhost'
    os.environ['MASTER_PORT'] = '12355'
    dist.init_process_group("gloo", rank=rank, world_size=world_size)

def cleanup():
    dist.destroy_process_group()

class ToyModel(nn.Module):
    def __init__(self):
        super(ToyModel, self).__init__()
        self.net1 = nn.Linear(10, 10)
        self.relu = nn.ReLU()
        self.net2 = nn.Linear(10, 5)

    def forward(self, x):
        return self.net2(self.relu(self.net1(x)))

def demo_basic(rank, world_size):
    print(f"Proceso {rank} empezando...")
    setup(rank, world_size)
    print(f"Proceso {rank} ha completado la configuración.")

    try:
```

```
        model = ToyModel().to(rank)
        ddp_model = DDP(model, device_ids=[rank])

        loss_fn = nn.MSELoss()
        optimizer = torch.optim.SGD(ddp_model.parameters(), lr=0.001)

        dataset = torch.randn(20, 10)
        targets = torch.randn(20, 5)

        for epoch in range(10):
            optimizer.zero_grad()
            outputs = ddp_model(dataset.to(rank))
            loss = loss_fn(outputs, targets.to(rank))
            loss.backward()
            optimizer.step()
            if rank == 0:
                print(f"Epoca {epoch+1}, Perdida: {loss.item()}")

    except Exception as e:
        print(f"Proceso {rank} encontró un error: {e}")
    finally:
        cleanup()
        print(f"Proceso {rank} ha terminado la limpieza.")

def run_demo(demo_fn, world_size):
    processes = []
    for rank in range(world_size):
        p = mp.Process(target=demo_fn, args=(rank, world_size))
        p.start()
        processes.append(p)
    for p in processes:
        p.join()

if __name__ == "__main__":
    world_size = 2
    run_demo(demo_basic, world_size)

import tensorflow as tf
from tensorflow.keras import layers, models, optimizers, losses
import numpy as np
import os

# Configuración del entorno
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

# Modelo de ejemplo (TensorFlow)
class ToyModel(tf.keras.Model):
    def __init__(self):
        super(ToyModel, self).__init__()
        self.net1 = layers.Dense(10, activation='relu')
```

```
        self.net2 = layers.Dense(5)

    def call(self, x):
        x = self.net1(x)
        return self.net2(x)

def demo_basic(strategy):
    print("Empezando entrenamiento con TensorFlow...")

    # Configuración del modelo dentro de la estrategia
    with strategy.scope():
        model = ToyModel()
        optimizer = optimizers.SGD(learning_rate=0.001)
        loss_fn =
losses.MeanSquaredError(reduction=tf.keras.losses.Reduction.NONE)
        global_batch_size = 4 * strategy.num_replicas_in_sync

    # Datos de ejemplo
    dataset = np.random.randn(20, 10).astype(np.float32)
    targets = np.random.randn(20, 5).astype(np.float32)

    # Creación del dataset de TensorFlow
    train_dataset = tf.data.Dataset.from_tensor_slices((dataset,
targets)).batch(global_batch_size)

    @tf.function
    def train_step(inputs, targets):
        with tf.GradientTape() as tape:
            predictions = model(inputs, training=True)
            per_example_loss = loss_fn(targets, predictions)
            loss = tf.nn.compute_average_loss(per_example_loss,
global_batch_size=global_batch_size)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients,
model.trainable_variables))
        return loss

    for epoch in range(10):
        total_loss = 0.0
        num_batches = 0
        for batch in train_dataset:
            inputs, targets = batch
            loss = strategy.run(train_step, args=(inputs, targets))
            total_loss += strategy.reduce(tf.distribute.ReduceOp.SUM,
loss, axis=None)
            num_batches += 1
        print(f"Epoca {epoch+1}, Perdida: {total_loss / num_batches}")

    print("Entrenamiento completado.")
```

```
def run_demo():
    # Creación de la estrategia para múltiples GPUs
    strategy = tf.distribute.MirroredStrategy()
    demo_basic(strategy)

if __name__ == "__main__":
    run_demo()
```

Métodos de Monte Carlo

```
import torch
import torch.multiprocessing as mp

def monte_carlo_simulation(num_simulations):
    count_inside = 0
    for _ in range(num_simulations):
        x, y = torch.rand(2)
        if x**2 + y**2 <= 1.0:
            count_inside += 1
    return count_inside

def parallel_monte_carlo(num_simulations, num_processes):
    with mp.Pool(num_processes) as pool:
        results = pool.map(monte_carlo_simulation, [num_simulations //
num_processes] * num_processes)
    return sum(results) / num_simulations * 4

if __name__ == "__main__":
    num_simulations = 1000000
    num_processes = 4
    pi_estimate = parallel_monte_carlo(num_simulations, num_processes)
    print(f"El valor de Pi estimado: {pi_estimate}")
```

Ejercicios

1 . Implementa un modelo de red neuronal simple y entrenarlo utilizando paralelización de datos en múltiples GPUs.

Instrucciones:

- Crea un modelo de red neuronal simple (por ejemplo, una red completamente conectada).
- Utiliza torch.nn.DataParallel para paralelizar el entrenamiento del modelo en múltiples GPUs.
- Entrena el modelo en el conjunto de datos MNIST.

Tareas

1. Implementa y completa el pseudocódigo.
2. Ejecuta el entrenamiento en un sistema con múltiples GPUs.

3. Analiza el rendimiento y la aceleración obtenida.

```
## Tus respuestas
```

2 . Implementa un modelo de GBM utilizando lightgbm y realizar el entrenamiento utilizando múltiples núcleos de CPU.

Instrucciones:

- Crea un conjunto de datos sintético.
- Utiliza lightgbm para entrenar un modelo de GBM, configurando la paralelización con múltiples núcleos de CPU.
- Evalua el rendimiento del modelo.

Tareas:

1. Implementa y completa el pseudocódigo.
2. Ajusta los parámetros del modelo para mejorar la precisión.
3. Evalua el rendimiento del modelo en términos de tiempo de entrenamiento y precisión.

```
## Tu respuesta
```

3 . Implementa una CNN y entrena utilizando paralelización de modelos en múltiples GPUs.

Instrucciones:

- Define una arquitectura CNN.
- Utiliza torch.nn.DataParallel para paralelizar el modelo en múltiples GPUs.
- Entrena el modelo en el conjunto de datos CIFAR-10.

Tareas:

1. Implementa y completa el pseudocódigo.
2. Ejecuta el entrenamiento en un sistema con múltiples GPUs.
3. Evalua la precisión del modelo y el tiempo de entrenamiento.

```
## Tu respuesta
```

4 .Implementa un modelo transformer simple y entrena utilizando pipeline parallelism en múltiples GPUs.

Instrucciones:

- Define una arquitectura de transformer.
- Utiliza torch.nn.DataParallel para paralelizar las capas del transformer en múltiples GPUs.
- Entrena el modelo en un conjunto de datos de procesamiento de lenguaje natural.

Tareas

1. Implementa y completa el pseudocódigo.
2. Ejecuta el entrenamiento en un sistema con múltiples GPUs.

3. Analiza el rendimiento y la precisión del modelo.

```
## Tu respuesta
```

5 . Implementa el algoritmo SGD paralelo en un entorno distribuido utilizando torch.distributed.

Instrucciones:

- Configura un entorno distribuido utilizando torch.distributed.
- Divide los datos de entrenamiento entre múltiples nodos.
- Implementa el algoritmo SGD paralelo donde cada nodo calcula los gradientes de forma independiente y luego los gradientes se agregan para actualizar los parámetros del modelo global.

Tareas

1. Implementa y completa el pseudocódigo.
2. Ejecuta el entrenamiento en un entorno distribuido con múltiples nodos.
3. Evalua el rendimiento del modelo y el tiempo de entrenamiento.

```
## Tus respuestas
```

6 . Implementa un algoritmo de Monte Carlo para la estimación de Pi y paralelizarlo utilizando multiprocessing.

Instrucciones:

- Escribe un algoritmo de Monte Carlo para estimar Pi.
- Utiliza multiprocessing para paralelizar la generación de muestras y la evaluación.
- Compara el tiempo de ejecución en comparación con una implementación secuencial.

```
import multiprocessing as mp
import random
import time

def monte_carlo_pi_part(n):
    count = 0
    for _ in range(n):
        x, y = random.random(), random.random()
        if x**2 + y**2 <= 1.0:
            count += 1
    return count

def parallel_monte_carlo_pi(total_samples, num_processes):
    pool = mp.Pool(processes=num_processes)
    samples_per_process = [total_samples // num_processes] *
num_processes
    counts = pool.map(monte_carlo_pi_part, samples_per_process)
    pi_estimate = sum(counts) / total_samples * 4
    return pi_estimate
```

```
if __name__ == "__main__":
    total_samples = 10**7
    num_processes = 4

    start_time = time.time()
    pi_estimate = parallel_monte_carlo_pi(total_samples,
num_processes)
    end_time = time.time()

    print(f"Pi estimado: {pi_estimate}")
    print(f"Tiempo tomado: {end_time - start_time} en segundos")
```

Tareas

1. Implementa y completa el pseudocódigo.
2. Ejecuta la estimación de Pi utilizando diferentes números de procesos.
3. Evalua el tiempo de ejecución y la precisión de la estimación.

```
## Tu respuesta
```

Procesamiento de logs

El procesamiento de logs en sistemas distribuidos es una tarea esencial para la gestión de grandes volúmenes de datos generados por aplicaciones y sistemas. La clave para procesar logs de manera eficiente es dividir los datos en fragmentos y distribuir estos fragmentos entre varios nodos para procesamiento paralelo. Posteriormente, los resultados parciales se combinan en una fase de reducción para generar la salida final.

El procesamiento de logs implica las siguientes etapas:

- División de datos: Los logs se dividen en fragmentos más pequeños que pueden ser procesados en paralelo.
- Distribución de tareas: Estos fragmentos se distribuyen entre múltiples nodos en un clúster.
- Procesamiento paralelo: Cada nodo procesa su fragmento de manera independiente, aplicando filtros, transformaciones y agregaciones necesarias.
- Reducción y agregación: Los resultados parciales de cada nodo se combinan para obtener la salida final, que puede ser un resumen de los logs o informes detallados.

```
from concurrent.futures import ThreadPoolExecutor, as_completed
import os

def process_log_fragment(log_fragment):
    # Implementar el procesamiento del fragmento de log
    processed_data = []
    for line in log_fragment:
        if "ERROR" in line:
            processed_data.append(line)
    return processed_data
```

```
def read_log_file(file_path, chunk_size=1024):
    with open(file_path, 'r') as f:
        while True:
            lines = f.readlines(chunk_size)
            if not lines:
                break
            yield lines

def combine_results(results):
    combined = []
    for result in results:
        combined.extend(result)
    return combined

def process_logs_in_parallel(log_file_path, num_workers=4):
    log_fragments = read_log_file(log_file_path)
    results = []
    with ThreadPoolExecutor(max_workers=num_workers) as executor:
        future_to_fragment = {executor.submit(process_log_fragment,
fragment): fragment for fragment in log_fragments}
        for future in as_completed(future_to_fragment):
            results.append(future.result())
    combined_results = combine_results(results)
    return combined_results

if __name__ == "__main__":
    log_file_path = 'large_log_file.log'
    processed_logs = process_logs_in_parallel(log_file_path)
    for log in processed_logs:
        print(log)
```

El uso de frameworks como Hadoop MapReduce o Apache Spark facilita este tipo de procesamiento distribuido.

```
from pyspark.sql import SparkSession

# Crear una sesión de Spark
spark = SparkSession.builder.appName("Log Processing").getOrCreate()

# Leer los logs desde un archivo
logs = spark.read.text("hdfs://path/to/logs")

# Función de procesamiento de logs
def process_log(line):
    # Procesamiento personalizado del log
    return line

# Aplicar la función de procesamiento a cada línea del log
processed_logs = logs.rdd.map(process_log)
```

```
# Guardar los resultados procesados
processed_logs.saveAsTextFile("hdfs://path/to/processed_logs")

spark.stop()
```

Análisis de grandes datos (Big Data Analytics)

El análisis de grandes datos implica el procesamiento y análisis de volúmenes masivos de datos para extraer información significativa. Herramientas como Apache Spark permiten realizar este tipo de análisis en paralelo, utilizando un modelo de programación distribuido.

En el contexto de Big Data, se utilizan dos tipos de operaciones principales:

- Transformaciones: Operaciones que crean un nuevo conjunto de datos a partir de un existente, como map, filter y reduceByKey.
- Acciones: Operaciones que devuelven un valor al controlador de Spark, como count, collect y saveAsTextFile.

Apache Spark distribuye los datos y las operaciones entre múltiples nodos, permitiendo un procesamiento eficiente a gran escala.

```
from pyspark.sql import SparkSession

# Crear una sesión de Spark
spark = SparkSession.builder \
    .appName("Big Data Analytics") \
    .getOrCreate()

# Leer datos de ejemplo
data = [("Checha", 34), ("Aco", 45), ("Tomo", 29)]
columns = ["Name", "Age"]
df = spark.createDataFrame(data, columns)

# Realizar transformaciones
df_filtered = df.filter(df.Age > 30)
df_grouped = df_filtered.groupBy("Name").count()

# Ejecutar una acción
df_grouped.show()

# Detener la sesión de Spark
spark.stop()
```

Programación de pasaje de mensajes (Message Passing Interface, MPI)

MPI es un estándar para la programación paralela en sistemas distribuidos que permite la comunicación eficiente entre nodos mediante el paso de mensajes. MPI es ampliamente utilizado en aplicaciones de alto rendimiento donde la comunicación y la sincronización entre procesos son esenciales.

Algoritmos distribuidos

Los algoritmos distribuidos están diseñados para sistemas donde los componentes son nodos autónomos que interactúan mediante comunicación de red. Estos algoritmos resuelven problemas donde la información y el control están distribuidos.

Dos ejemplos importantes son los algoritmos de consenso y los algoritmos de elección de líder.

Algoritmos de consenso

- Paxos y Raft: Estos algoritmos aseguran que todos los nodos en un sistema distribuido acuerden un valor único, crucial para la consistencia de los datos.

Algoritmos de elección de líder

- Determinan un líder entre un grupo de nodos, utilizado para coordinar actividades en sistemas distribuidos.

```
## Ejemplo simple: Raft
import random
import threading

class Node:
    def __init__(self, node_id):
        self.node_id = node_id
        self.state = "follower"
        self.votes = 0

    def start_election(self):
        self.state = "candidate"
        self.votes = 1  # vote for self
        print(f"Nodo {self.node_id} inicia una eleccion")

    def receive_vote(self):
        self.votes += 1
        print(f"Nodo {self.node_id} recive un voto")

nodes = [Node(i) for i in range(5)]
leader_elected = threading.Event()

def run_node(node):
    while not leader_elected.is_set():
        if node.state == "follower" and random.random() < 0.05:
            node.start_election()
            for peer in nodes:
                if peer != node:
                    peer.receive_vote()
            if node.votes > len(nodes) / 2:
                node.state = "leader"
                leader_elected.set()
                print(f"Nodo {node.node_id} es elegido como lider")

threads = [threading.Thread(target=run_node, args=(node,)) for node in
nodes]
```

```
for t in threads:
    t.start()
for t in threads:
    t.join()
```

Algoritmo de elección de líder

```
import threading

class LeaderElection:
    def __init__(self, num_nodes):
        self.num_nodes = num_nodes
        self.nodes = [i for i in range(num_nodes)]
        self.leader = None
        self.lock = threading.Lock()

    def elect_leader(self):
        with self.lock:
            if self.leader is None:
                self.leader = max(self.nodes)
                print(f"Nodo {self.leader} es elegido como lider")

def node_task(election):
    election.elect_leader()

election = LeaderElection(num_nodes=5)
threads = [threading.Thread(target=node_task, args=(election,)) for _
in range(election.num_nodes)]
for t in threads:
    t.start()
for t in threads:
    t.join()
```

Ejercicios

1 . Implementa un sistema de procesamiento de logs utilizando el paradigma MapReduce para dividir los logs, procesarlos en paralelo y combinar los resultados.

Instrucciones:

- Divide el archivo de logs en fragmentos.
- Implementa la función map para procesar cada fragmento de logs y filtrar las líneas que contienen "ERROR".
- Implementa la función reduce para combinar los resultados parciales.

```
from concurrent.futures import ThreadPoolExecutor, as_completed

def map_function(log_fragment):
    # Filtrar líneas con "ERROR"
    return [line for line in log_fragment if "ERROR" in line]
```

```
def reduce_function(mapped_data):
    # Combinar los resultados de todos los mappers
    return [line for sublist in mapped_data for line in sublist]

def read_log_file(file_path, chunk_size=1024):
    with open(file_path, 'r') as f:
        while True:
            lines = f.readlines(chunk_size)
            if not lines:
                break
            yield lines

def process_logs_in_parallel(log_file_path, num_workers=4):
    log_fragments = read_log_file(log_file_path)
    results = []
    with ThreadPoolExecutor(max_workers=num_workers) as executor:
        future_to_fragment = {executor.submit(map_function, fragment):
fragment for fragment in log_fragments}
        for future in as_completed(future_to_fragment):
            results.append(future.result())
    combined_results = reduce_function(results)
    return combined_results

if __name__ == "__main__":
    log_file_path = 'large_log_file.log'
    processed_logs = process_logs_in_parallel(log_file_path)
    for log in processed_logs:
        print(log)
```

Tareas

1. Implementa y completa el código base.
2. Ejecuta el procesamiento de logs en un sistema con múltiples hilos.
3. Evalua el rendimiento y la eficiencia del procesamiento paralelo.

```
## Tu respuesta
```

2 . Implementa un análisis de grandes datos utilizando Apache Spark para procesar y analizar un conjunto de datos grande.

Instrucciones

- Configura una sesión de Spark.
- Lea un conjunto de datos grande.
- Realiza transformaciones y acciones sobre los datos.

Tareas:

1. Implementa el código base para este ejercicio.
2. Ejecuta el análisis de datos en un entorno de Spark.

3. Evalua la escalabilidad y eficiencia del procesamiento de datos en Spark.

```
## Tu respuesta
```

3 . Implementa el algoritmo de K-Means para clustering utilizando paralelización en múltiples hilos o procesos.

Instrucciones

- Implementa la inicialización de centroides.
- Asigna puntos de datos a los centroides más cercanos en paralelo.
- Recalcula los centroides en paralelo.
- Repite hasta la convergencia.

```
import numpy as np
from concurrent.futures import ThreadPoolExecutor

def initialize_centroids(data, k):
    indices = np.random.choice(data.shape[0], k, replace=False)
    return data[indices]

def assign_clusters(data, centroids):
    clusters = {}
    for x in data:
        closest_centroid = np.argmin([np.linalg.norm(x - centroid) for
centroid in centroids])
        if closest_centroid not in clusters:
            clusters[closest_centroid] = []
        clusters[closest_centroid].append(x)
    return clusters

def update_centroids(clusters):
    return [np.mean(clusters[k], axis=0) for k in clusters]

def kmeans_parallel(data, k, num_workers=4, max_iters=100):
    ## Completa
```

Tareas

1. Implementa y completa el código base.
2. Ejecuta el algoritmo de K-Means en un conjunto de datos grande.
3. Evalua la convergencia y la calidad del clustering.

```
## Tu respuesta
```

4 . Implementa un programa MPI que sincroniza múltiples procesos utilizando barreras.

Instrucciones

- Configura un entorno MPI.
- Implementa la sincronización de procesos con MPI_Barrier.

- Haz que cada proceso realice una tarea antes y después de la barrera.

Tareas:

1. Implementa y completa el código base.
2. Ejecuta el programa en un entorno MPI con múltiples procesos.
3. Evalua la sincronización y el comportamiento de los procesos.

```
## Tu respuesta
```

5 . Implementa un programa MPI que utiliza MPI_Allreduce para realizar operaciones de reducción entre todos los procesos.

Instrucciones:

- Configura un entorno MPI.
- Implementa un cálculo distribuido donde cada proceso genera un valor.
- Utiliza MPI_Allreduce para sumar los valores generados por todos los procesos.

Tareas:

1. Implementa y completa el código base.
2. Ejecuta el programa en un entorno MPI con múltiples procesos.
3. Evalua la eficiencia y corrección del cálculo distribuido.

```
## Tu respuesta
```

Ejercicios de repaso

1 . Implementa el algoritmo de Merge Sort usando paralelización para dividir y vencer.

Instrucciones:

- Implementa la función de merge sort que divide el problema en sub-problemas.
- Paraleliza la división de los sub-problemas utilizando concurrent.futures.ThreadPoolExecutor.
- Ejecuta el algoritmo de Merge Sort en un conjunto de datos grande.
- Evalua la eficiencia y tiempo de ejecución del algoritmo paralelo en comparación con la versión secuencial.

```
## Tu respuesta
```

2 . Implementa un algoritmo de reducción para sumar los elementos de una matriz utilizando paralelización.

Instrucciones:

- Divide la matriz en fragmentos.
- Calcula la suma de cada fragmento en paralelo.
- Combina los resultados parciales en una suma total.

```
import numpy as np
from concurrent.futures import ThreadPoolExecutor

def sum_matrix_part(matrix_part):
    return np.sum(matrix_part)

def parallel_sum_matrix(matrix, num_workers=4):
    rows, cols = matrix.shape
    chunk_size = rows // num_workers
    futures = []

    with ThreadPoolExecutor(max_workers=num_workers) as executor:
        for i in range(num_workers):
            start_row = i * chunk_size
            end_row = (i + 1) * chunk_size if i != num_workers - 1
else rows
            futures.append(executor.submit(sum_matrix_part,
matrix[start_row:end_row, :]))

    total_sum = sum(f.result() for f in futures)
    return total_sum

# Datos de ejemplo
matrix = np.random.rand(1000, 1000)
total_sum = parallel_sum_matrix(matrix)
print("Suma total de matriz:", total_sum)
```

Tareas

1. Implementa y completa el código base.
2. Ejecuta el algoritmo de reducción en una matriz grande.
3. Evalua la eficiencia y tiempo de ejecución del algoritmo paralelo en comparación con la versión secuencial.

```
## Tu respuesta
```

3 .Implementa un algoritmo de barrido paralelo para buscar un elemento en un conjunto de datos.

Instrucciones:

- Divide el conjunto de datos en fragmentos.
- Busca el elemento en cada fragmento en paralelo.
- Combina los resultados parciales.

```
import concurrent.futures

def find_element_in_part(data_part, element):
    return element in data_part
```

```
def parallel_find_element(data, element, num_workers=4):
    chunk_size = len(data) // num_workers
    futures = []

    with
concurrent.futures.ThreadPoolExecutor(max_workers=num_workers) as
executor:
        for i in range(num_workers):
            start_idx = i * chunk_size
            end_idx = (i + 1) * chunk_size if i != num_workers - 1
else len(data)
            futures.append(executor.submit(find_element_in_part,
data[start_idx:end_idx], element))

    found = any(f.result() for f in futures)
    return found

# Datos de ejemplo
data = list(range(10000))
element = 5000
found = parallel_find_element(data, element)
print(f"Elemento {element} encontrado:", found)
```

Tareas

1. Implementa y completa el código base.
2. Ejecuta el algoritmo de búsqueda en un conjunto de datos grande.
3. Evalua la eficiencia y tiempo de ejecución del algoritmo paralelo en comparación con la versión secuencial.

```
## Tu repuesta
```

4 . Implementa la búsqueda en anchura (BFS) en un grafo utilizando paralelización.

Instrucciones:

- Implementa la estructura de datos del grafo.
- Implementa el algoritmo BFS.
- Paraleliza el procesamiento de los nodos en cada nivel del BFS.
- Ejecuta el algoritmo BFS en un grafo grande.
- Evalua la eficiencia y tiempo de ejecución del algoritmo paralelo en comparación con la versión secuencial.

```
## Tu respuesta
```

5 . Implementa la búsqueda en profundidad (DFS) en un grafo utilizando paralelización.

Instrucciones:

- Implementa la estructura de datos del grafo.

- Implementa el algoritmo DFS.
- Paraleliza el procesamiento de los nodos en la recursión DFS.
- Ejecuta el algoritmo DFS en un grafo grande.
- Evalua la eficiencia y tiempo de ejecución del algoritmo paralelo en comparación con la versión secuencial.

```
## Tu respuesta
```