Of course! Let’s break this down carefully:

**✨ Code Explanation:**

**1. Import libraries**

import numpy as np

from concurrent.futures import ThreadPoolExecutor

* numpy is used for fast array operations like sum, min, and max.
* ThreadPoolExecutor is used to **simulate parallelism** by running functions in different threads.

**2. parallel\_reduce Function**

def parallel\_reduce(arr, operation, num\_threads=4):

* Takes:
  + arr → the input array
  + operation → what you want to perform ('sum', 'min', or 'max')
  + num\_threads → how many threads you want (default 4)

**3. Dividing the Array**

chunk\_size = len(arr) // num\_threads

* Divides the array into nearly equal parts.

**4. Processing Each Chunk**

def process\_chunk(chunk):

if operation == 'sum':

return np.sum(chunk)

elif operation == 'min':

return np.min(chunk)

elif operation == 'max':

return np.max(chunk)

* Each thread will work on **one chunk**:
  + Find its **sum**, **min**, or **max**.

**5. Creating Threads**

with ThreadPoolExecutor(max\_workers=num\_threads) as executor:

futures = []

for i in range(num\_threads):

start = i \* chunk\_size

end = (i + 1) \* chunk\_size if i != num\_threads - 1 else len(arr)

chunk = arr[start:end]

futures.append(executor.submit(process\_chunk, chunk))

* Using a **ThreadPoolExecutor**:
  + Submits tasks to run process\_chunk() for each part of the array.

**6. Collecting Results**

for future in futures:

results.append(future.result())

* As threads complete, their partial results (sum, min, max) are collected.

**7. Final Reduction**

if operation == 'sum':

return np.sum(results)

elif operation == 'min':

return np.min(results)

elif operation == 'max':

return np.max(results)

* After all threads finish:
  + **Partial results** are combined to get **final output**.

**3. Main Program**

arr = np.random.randint(1, 100, size=20)

* Generates a random array of 20 numbers between 1 and 99.

**4. Calling parallel\_reduce**

parallel\_sum = parallel\_reduce(arr, 'sum')

parallel\_min = parallel\_reduce(arr, 'min')

parallel\_max = parallel\_reduce(arr, 'max')

parallel\_avg = parallel\_sum / len(arr)

* Perform **sum**, **min**, **max**, and **average** operations in parallel.

**5. Printing Results**

print(f"Parallel Sum: {parallel\_sum}")

print(f"Parallel Min: {parallel\_min}")

print(f"Parallel Max: {parallel\_max}")

print(f"Parallel Average: {parallel\_avg:.2f}")

* Shows the output.

**📚 Theory Related to This Practical**

**➡️ What is Parallel Reduction?**

* **Reduction** means combining a set of elements into a **single result** (like sum, min, max).
* **Parallel Reduction** means doing this **in parallel** — using multiple threads or processors to make it faster.

Example:  
Instead of summing all elements one-by-one,  
split the array into parts,  
sum parts **simultaneously**,  
and then combine partial sums.

**➡️ Operations Performed:**

| **Operation** | **Meaning** |
| --- | --- |
| Sum | Add all elements |
| Min | Find the smallest element |
| Max | Find the largest element |
| Average | (Sum of all elements) / (Number of elements) |

**➡️ Why use Threads?**

* **Threads** allow **multiple parts** of the program to run **at the same time**.
* Reduces total execution time on **multi-core processors**.
* Especially useful for **large arrays**.

**➡️ Limitations:**

* Python has a **GIL (Global Interpreter Lock)** → true parallelism with threads isn't possible for CPU-heavy tasks.
* **Better performance** can be achieved with **multiprocessing** or in languages like **C++** with **OpenMP**.
* For small arrays, threading overhead can **slow down** instead of speeding up.

**🧠 Final Summary:**

| **Topic** | **Details** |
| --- | --- |
| Technique | Parallel Reduction |
| Language | Python |
| Parallelism | Simulated using ThreadPoolExecutor |
| Operations | Sum, Min, Max, Average |
| Advantage | Faster for large arrays |
| Limitation | Python GIL restricts real parallelism with threads |

**🚀 Quick Example:**

Suppose array = [1, 2, 3, 4] and using 2 threads:

* Thread 1 → works on [1,2] → sum = 3
* Thread 2 → works on [3,4] → sum = 7
* Final sum = 3 + 7 = 10

**✅ That's the full explanation!**

Would you also like me to provide a **ready-to-submit practical journal write-up** (with Aim, Apparatus, Code, Output, Result, and Conclusion)? 📔  
(Just say **"yes"** and I'll prepare it!)