**DFS**

from multiprocessing import Process, Array

class Graph:

def \_\_init\_\_(self, num\_nodes):

self.adj\_list = [[] for \_ in range(num\_nodes)]

def add\_edge(self, src, dest):

self.adj\_list[src].append(dest)

self.adj\_list[dest].append(src)

def dfs(graph, visited, v, traversal\_order):

if not visited[v]:

visited[v] = 1

traversal\_order.append(v)

for adj in graph.adj\_list[v]:

dfs(graph, visited, adj, traversal\_order)

def parallel\_dfs(graph, start):

num\_nodes = len(graph.adj\_list)

visited = Array('i', [0] \* num\_nodes)

traversal\_order = []

# Start parallel DFS traversal from all vertices

processes = []

for v in range(num\_nodes):

if v != start:

p = Process(target=dfs, args=(graph, visited, v, traversal\_order))

processes.append(p)

p.start()

# Start DFS traversal from the start vertex

dfs(graph, visited, start, traversal\_order)

# Wait for all processes to finish

for p in processes:

p.join()

# Print DFS traversal

print("Parallel DFS from vertex", start, ":", traversal\_order)

if \_\_name\_\_ == "\_\_main\_\_":

num\_nodes = 6

graph = Graph(num\_nodes)

# Add edges

graph.add\_edge(5, 1)

graph.add\_edge(5, 3)

graph.add\_edge(1, 2)

graph.add\_edge(1, 4)

graph.add\_edge(3, 4)

graph.add\_edge(2, 0)

graph.add\_edge(4, 0)

start\_vertex = int(input("Enter the start vertex: ")) # Take input for the starting vertex

# Perform parallel DFS

parallel\_dfs(graph, start\_vertex)

**BFS**

from multiprocessing import Process, Array, Queue

class Graph:

def \_\_init\_\_(self, num\_nodes):

self.adj\_list = [[] for \_ in range(num\_nodes)]

def add\_edge(self, src, dest):

self.adj\_list[src].append(dest)

self.adj\_list[dest].append(src)

def bfs(graph, start, visited, traversal\_order, queue):

queue.put(start)

visited[start] = 1

while not queue.empty():

v = queue.get()

traversal\_order.append(v)

for adj in graph.adj\_list[v]:

if not visited[adj]:

visited[adj] = 1

queue.put(adj)

def parallel\_bfs(graph, start):

num\_nodes = len(graph.adj\_list)

visited = Array('i', [0] \* num\_nodes)

traversal\_order = []

queue = Queue()

# Start parallel BFS traversal from all vertices

processes = []

for v in range(num\_nodes):

if v != start:

p = Process(target=bfs, args=(graph, v, visited, traversal\_order, queue))

processes.append(p)

p.start()

# Start BFS traversal from the start vertex

bfs(graph, start, visited, traversal\_order, queue)

# Wait for all processes to finish

for p in processes:

p.join()

# Print BFS traversal

print("Parallel BFS from vertex", start, ":", traversal\_order)

if \_\_name\_\_ == "\_\_main\_\_":

num\_nodes = 6

graph = Graph(num\_nodes)

# Add edges

graph.add\_edge(0, 1)

graph.add\_edge(0, 2)

graph.add\_edge(1, 3)

graph.add\_edge(1, 4)

graph.add\_edge(2, 5)

start\_vertex = int(input("Enter the start vertex: ")) # Take input for the starting vertex

# Perform parallel BFS

parallel\_bfs(graph, start\_vertex)

**BUBLE SORT**

import multiprocessing

def bubble\_sort(arr):

n = len(arr)

for i in range(n):

for j in range(0, n-i-1):

if arr[j] > arr[j+1]:

arr[j], arr[j+1] = arr[j+1], arr[j]

def parallel\_bubble\_sort(arr):

# Split the array into chunks for parallel processing

num\_chunks = multiprocessing.cpu\_count()

chunk\_size = max(1, len(arr) // num\_chunks) # Ensure chunk size is at least 1

chunks = [arr[i:i + chunk\_size] for i in range(0, len(arr), chunk\_size)]

# Create processes for each chunk

processes = []

for chunk in chunks:

process = multiprocessing.Process(target=bubble\_sort, args=(chunk,))

process.start()

processes.append(process)

# Wait for all processes to finish

for process in processes:

process.join()

# Merge sorted chunks

sorted\_arr = merge\_chunks(chunks)

return sorted\_arr

def merge\_chunks(chunks):

sorted\_arr = []

while any(chunks):

min\_val = float('inf')

min\_index = -1

for i, chunk in enumerate(chunks):

if chunk and chunk[0] < min\_val:

min\_val = chunk[0]

min\_index = i

sorted\_arr.append(chunks[min\_index].pop(0))

return sorted\_arr

if \_\_name\_\_ == "\_\_main\_\_":

arr = [6, 4, 25, 1, 2, 11, 9]

print("Original array:", arr)

sorted\_arr = parallel\_bubble\_sort(arr)

print("Sorted array bubble:", sorted\_arr)

**MERGE SORT**

import concurrent.futures

import multiprocessing

def merge\_sort(arr):

if len(arr) <= 1:

return arr

mid = len(arr) // 2

left\_half = arr[:mid]

right\_half = arr[mid:]

# Recursively sort the left and right halves

left\_half = merge\_sort(left\_half)

right\_half = merge\_sort(right\_half)

# Merge the sorted halves

return merge(left\_half, right\_half)

def merge(left, right):

merged = []

left\_index = right\_index = 0

# Merge the left and right subarrays

while left\_index < len(left) and right\_index < len(right):

if left[left\_index] < right[right\_index]:

merged.append(left[left\_index])

left\_index += 1

else:

merged.append(right[right\_index])

right\_index += 1

# Append remaining elements from left and right subarrays

merged.extend(left[left\_index:])

merged.extend(right[right\_index:])

return merged

def parallel\_merge\_sort(arr):

# Split the array into chunks for parallel processing

num\_chunks = multiprocessing.cpu\_count()

chunk\_size = max(1, len(arr) // num\_chunks) # Ensure chunk size is at least 1

chunks = [arr[i:i + chunk\_size] for i in range(0, len(arr), chunk\_size)]

# Use ThreadPoolExecutor to perform parallel merge sort

with concurrent.futures.ThreadPoolExecutor() as executor:

sorted\_chunks = list(executor.map(merge\_sort, chunks)) # Convert generator to list

# Merge the sorted chunks sequentially

sorted\_arr = merge\_chunks(sorted\_chunks)

return sorted\_arr

def merge\_chunks(chunks):

sorted\_arr = chunks[0]

for chunk in chunks[1:]:

sorted\_arr = merge(sorted\_arr, chunk)

return sorted\_arr

if \_\_name\_\_ == "\_\_main\_\_":

arr = [640, 34, 2555, 12, 22, 11, 90]

print("Original array:", arr)

sorted\_arr = parallel\_merge\_sort(arr)

print("Sorted array merge:", sorted\_arr)

**MIN,MAX,AVG**

import concurrent.futures

import os

def parallel\_reduction(data, operation):

num\_threads = min(len(data), os.cpu\_count())

chunk\_size = (len(data) + num\_threads - 1) // num\_threads

# Divide data into chunks

chunks = [data[i:i+chunk\_size] for i in range(0, len(data), chunk\_size)]

# Perform reduction on each chunk in parallel

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

results = list(executor.map(operation, chunks))

# Final reduction on aggregated results from all threads

return operation(results)

def parallel\_min(chunk):

return min(chunk)

def parallel\_max(chunk):

return max(chunk)

def parallel\_sum(chunk):

return sum(chunk)

def parallel\_average(chunk):

return sum(chunk) / len(chunk)

if \_\_name\_\_ == "\_\_main\_\_":

data = [1, 5, 3, 7, 9, 2, 4, 6, 8, 10]

# Min operation

min\_value = parallel\_reduction(data, parallel\_min)

print("Min:", min\_value)

# Max operation

max\_value = parallel\_reduction(data, parallel\_max)

print("Max:", max\_value)

# Sum operation

sum\_value = parallel\_reduction(data, parallel\_sum)

print("Sum:", sum\_value)

# Average operation

avg\_value = parallel\_reduction(data, parallel\_average)

print("Average:", avg\_value)