Computer Algorithm Lab

Q] Write a suitable example, analyze the difference between point-to-point communication and collective communication

Daulatrao A. Patil

2020BTEIT00034
CA LAB ESE

Platform: *Linux*
Language: *Python*
Execution: *python 2020BTEIT00034.py len_of_array*

If argument len_of_array is not provided then it will set the length to 100000

Code:

```
'''

PRN: 2020BTEIT00034

Name: Daulatrao Patil

'''



import random, time, sys

import multiprocessing

from multiprocessing import Process, Pipe
```

```
def main():
    """
    This is the main method, where we:
    -generate a random list.
    -time a sequential quicksort on the list.
    -time a parallel quicksort on the list.
    """
    N = 100000
    if len(sys.argv) > 1:  #the user input a list size.
        N = int(sys.argv[1])


    n = multiprocessing.cpu_count()


    # print(multiprocessing.cpu_count())


    #We want to sort the same list, so make a backup.
    backup_list = [random.random() for x in range(N)]


    #Sequential quicksort a copy of the list.
    main_list = list(backup_list)               #copy the list
    start_time = time.time()                    #start time
    main_list = quicksort(main_list)            #quicksort the list
    elapsed_time = time.time() - start_time     #calculate time
```

```
    if not isSorted(main_list):
        print('😞 quicksort did not sort the list. oops. 😞\n')
        return

    print('🔷 Sequential quicksort: %f sec\n' % elapsed_time)

    #So that cpu usage shows a lull.
    time.sleep(1)
    print('\tPreparing the processors... 👍')
    time.sleep(1)
    print('\tIntializing the Pipes... 👍\n')
    time.sleep(1)

    #Parallel quicksort.
    main_list = list(backup_list)

    if n > multiprocessing.cpu_count():
        print("\t🔥 System Overloading 🔥\n\t⛔ Processors not available!!!
⛔\n\t✋ Aborting the mission ✋")
        return

    start_time = time.time()
```

```
    #Instantiate a Pipe so that we can receive the process's response.
    pconn, cconn = Pipe()

    #Instantiate a process that executes quicksortParallel on the entire
list.
    p = Process(target=quicksortParallel, args=(main_list, cconn, n))
    p.start()

    main_list = pconn.recv()
    #Blocks until there is something (the sorted list) to receive.

    p.join()
    elapsed_time = time.time() - start_time

    if not isSorted(main_list):
        print('😞 quicksortParallel did not sort the list. oops. 😞\n')

    print('🔷 Parallel quicksort: %f sec' % elapsed_time)

def quicksort(list_array):
    """
    Quicksort implementation, return a new sorted version of the input
list.
    """
```

```
    if len(list_array) <= 1:
        return list_array
    pivot = list_array.pop(random.randint(0, len(list_array)-1))

    return quicksort([x for x in list_array if x < pivot]) + [pivot] +
quicksort([x for x in list_array if x >= pivot])

def quicksortParallel(list_array, conn, procNum):
    """
    Partition the list, then quicksort the left and right sides in
parallel.
    """

    if procNum <= 1 or len(list_array) <= 1:
        #In the case of len(list) <= 1, quicksort will immediately return
anyway.
        conn.send(quicksort(list_array))
        conn.close()
        return

    pivot = list_array.pop(random.randint(0, len(list_array)-1))

    leftSide = [x for x in list_array if x < pivot]
    rightSide = [x for x in list_array if x >= pivot]
```

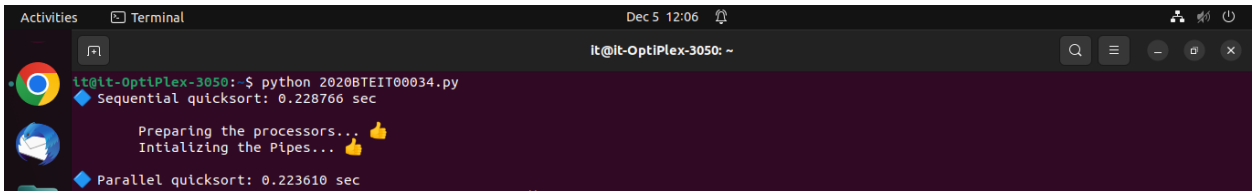
```
    #Creat a Pipe to communicate with the left subprocess
    pconnLeft, cconnLeft = Pipe()

    #Create a leftProc that executes quicksortParallel on the left
half-list.
    leftProc = Process(target=quicksortParallel, args=(leftSide, cconnLeft,
procNum - 1))

    #Again, for the right.
    pconnRight, cconnRight = Pipe()
    rightProc = Process(target=quicksortParallel, args=(rightSide,
cconnRight, procNum - 1))

    #Start the two subprocesses.
    leftProc.start()
    rightProc.start()

    #Our answer is the concatenation of the subprocesses answers, with the
pivot in between.
    conn.send(pconnLeft.recv() + [pivot] + pconnRight.recv())
    conn.close()

    #Join our subprocesses.
    leftProc.join()
    rightProc.join()
```

```
def isSorted(list):
   """
  Checking if the list is sorted or not.
   """
   for i in range(1, len(list)):
       if list[i] < list[i-1]:
           return False
   return True



#Call the main method if run from the command line.
if __name__ == '__main__':
   main()
```

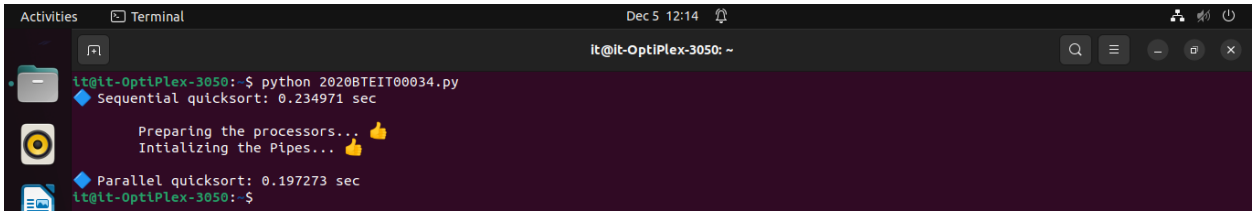
Output

Output of the code for different numbers of processors.

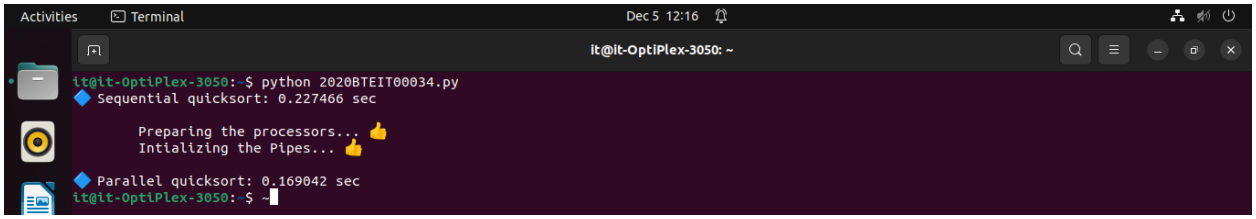
P = 2



P = 3



P = 4 (Max)



Comparison

Output No	QuickSort	Parallel Quicksort
1 -> P = 2	T = 0.228766	T = 0.223610
2 -> P = 3	T = 0.234971	T = 0.197273
3 -> P = 4 (max)	T = 0.227466	T = 0.169042

Conclusion

1. Point-to-Point Communication and Collective Communication is demonstrated using example of Sequential Quicksort v/s Parallel Quicksort
2. As we increase the number of processors the time eventually decreases for parallel processing of quicksort.
3. Implementation is done in python.