Parallel Programming: Practice 7

May 13 – 17, 2024

Answers must be delivered in a single document (pdf format). This document must include the full name of all group members. If the exercise is an implementation exercise, the source code can be delivered in separate .py files within a compressed archive.

Exercise 1

The Allgather function is a collective operation that collects data from all processes in a communication and distributes it to all processes, including the process that provided the original data. In other words, Allgather gathers data from all processes and distributes it back to all processes.

In terms of syntax and parameters, the Allgather function is similar to Gather, but with the key difference that data is distributed to all processes, not just the root process.

Python Objects

```
datarecv = comm.allgather(datasend)
```

Buffered data (e.g. numpy arrays)

```
comm.AllGather(datasend, datareceived)
```

The allgather operation could be seen as a gather operation followed by a broadcast.

In exercise 4, which uses the Scatterv function, to know how much data is distributed to each processor, process 0 has calculated the size vector and has sent that information to the rest of the processors.

- Modify exercise 4 so that each process calculates the number of elements it should receive and all processors have that information.

Exercise 2

So far, we have seen the Scatterv function in its simplest version

```
comm.Scatterv([global_array, tamanyos], local_array, root = 0)
```

However, it may be the case that we do not want to distribute all the elements of the vector or we do not want the distribution to begin at element 0. In these cases, the syntax must be used:

```
comm.Scatterv([global_array, sizes, offset,
MPI.DOUBLE],local_array, root = 0)
```

In this case, in addition to indicating the array to be distributed (array_global) and the number of elements that each processor will receive (sizes), another array (displacement) must be specified with the initial position of the elements within the original vector and the type of the elements. elements to send (MPI.INT or MPI.DOUBLE)

In the same way that gather is the inverse of scatter, MPI offers us Gatherv as the inverse of Scatterv. We will use Gatherv when the number of elements to collect from each process is different (that is, not all processes send the same number of data). The parameters for the Gatherv primitive are similar to those of the Scatterv, exchanging origin and destination:

```
comm.Gatherv(array_local, [array_global, tamanyos], root = 0)
```

```
comm.Gatherv(array_local, [array_global, sizes,
offset, MPI.DOUBLE], root = 0)
```

To practice these functions we are going to assume an execution with four processes, P2 distributes data from vector B (of 16 integers) as follows: to P0: B[3], B[4], B[5]; to P1: B[7], B[8]; a P2: B[10]; and to P3: B[12], B[13], B[14], B[15]. After that, each process adds 100 to the received elements, and, finally, the final data is collected in P2, in the same initial positions of vector B.

- Complete the program below to perform the described functionalities: At first, P2 must initialize the vector to B[i]
  = i, and, at the end, print the new vector B.

  The program should print:

  ```
  B in rank=2 after the calculation

   0 1 2 103 104 105 6 107 108              9 110 11 112 113 114 115
  ```

```
from mpi4py import MPI import
numpy

comm = MPI.COMM_WORLD
```

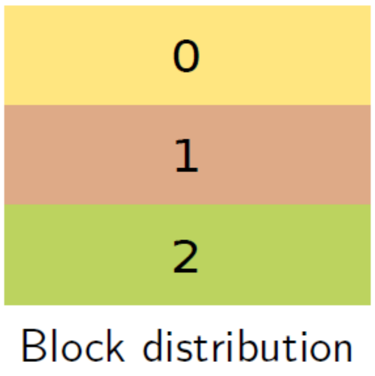
```
rank     =   comm.Get_rank()
size     =   comm.Get_size()


if rank == 2:
        B=numpy.empty(16, dtype='i') for i in
        range(16):
                B[i] = i
else:
        B=None

# Scattering of B from P2


# Local calculation


# Gathering of B in P2


# Print results
if rank == 2:
        print("\n B in rank=2 after the calculation \n") for i in range(16):

                print("%4d" % B[i], end="") print("\n\n")
```

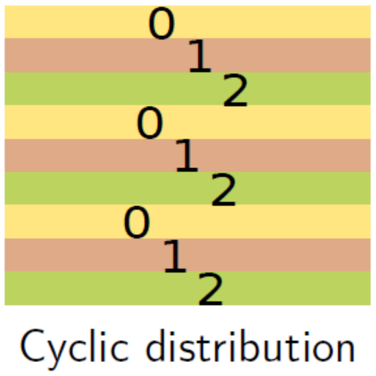
Exercise 3

To practice the use of mpi we are going to choose a problem with high computational cost such as the calculation of the PI number, in which a large number of repetitive calculations are carried out. Next, you have a sequential code to calculate an approximation of the PI number.

```
import math

def    compute_pi(n):
       h = 1.0/n
       s = 0.0

       for i in range(n):
              x = h * (i + 0.5) s += 4.0 / (1.0 + x**2)
              return s * h


n =     100
pi =      compute_pi(n)
error = abs(pi - math.pi)

print ("pi is approximately %.16f, error is %.16f" % (pi, error))
```

- Modify the exercise so that the calculation is done in parallel with p processors. You will need to divide the number of repetitions between the processors involved. Use a cyclical distribution to achieve best results. Finally you will have to combine the partial results obtained by each processor to obtain the final PI result.





Exercise 4

Modify the previous exercise to use Numpy. Remember that you can convert a Python int or float to a numpy scalar value with the following expressionx = numpy.array(x).

Exercise 5

Another example of an algorithm that can be parallelized with MPI is the algorithm that generates and visualizes the Julia set (it can also be done with the Mandelbrot set). The Julia set is a set of points in the complex plane that exhibit certain mathematical properties. These points are calculated iteratively using a specific function and a color is assigned to each point based on certain criteria. The shape and pattern of the Julia set depend on the function used and the initial values   (seeds) given to the calculations. The Julia set is commonly visualized as a set of points in the complex plane, where each point is colored according to the number of iterations required for a specific sequence of calculations to diverge to infinity or remain bounded. Points inside the Julia set are typically colored black or a specific color, while points outside the set are colored differently depending on how quickly they diverge.

Here is a basic example of how to generate and display the Julia array in Python using the matplotlib library. This code will generate and display the Julia assembly with a specific parameter setting. You can experiment with different values   of c, as well as adjust other parameters such as xmin, xmax, ymin, ymax, width, height, and max_iter to get different visualizations of the Julia set.

```
import numpy as np
import matplotlib.pyplot as plt

def julia_set(z, c, max_iter):
        for i in range(max_iter):
                if abs(z) > 2:
                        return i
                z = z**2 + c
        return max_iter

def      generate_julia_set(xmin,                    xmax,       ymin,        ymax,
width, height, c, max_iter):
        x_vals = np.linspace(xmin, xmax, width) y_vals =
        np.linspace(ymin, ymax, height) julia = np.zeros((height, width))


        for i in range(width):
                for j in range(height):
                        x   =   x_waltz[i]
                        and=   y_waltz[j]
                        z = complex(x, y)
                        julia[j, i] = julia_set(z, c, max_iter)

        return julia

def       plot_julia_set(julia_set,xmin,                        xmax,        ymin,
ymax):
        plt.imshow(julia_set,                 cmap='hot', extent=(xmin,
xmax, ymin, ymax))
        plt.colorbar()
        plt.title("Set                      of    Julia")
        plt.xlabel("Part                 real")
        plt.ylabel("Part                 imaginary")
        plt.show()

if __name__ == "__main__":
        xmin, xmax, ymin, ymax = -2, 2, -2, 2 width, height = 500,
        500
        c = complex(-0.7, 0.27015) max_iter = 300


        Julia        =     generate_julia_set(xmin,                  xmax,       ymin,
ymax, width, height, c, max_iter)
        plot_julia_set(julia, xmin, xmax, ymin, ymax)
```

- Parallelize the calculation of the Julia set using MPI, to do this we will divide the complex plane into regions and assign each region to a process. Each process will be responsible for computing a part of the Julia set and then the results will be combined to form the full image.

Note: You must make modifications to the following functions:

- generate_julia_set so that each process calculates the Julia set on the rows that correspond to it. In this case it uses a distribution by blocks. Each process will calculate its start row and end row and perform the for loop on those rows.

- plot_julia_set to collect the partial julia sets computed by each process. Furthermore, once all the partial sets have been assembled into the final set, only processor 0 should display the image.