Lab Course : Distributed Data Analytics

Exercise Sheet 1 - Group 2

Krithika Murugesan (277537)

Distributed Computing with Message Passing Interface

Message Passing Interface API provided by python, mpi4py is studied in this exercise. It allows easy implementation of parallel computation. Communication between processes is done by *Comm.Recv* and *Comm.Send* functions.

Exercise 0 : Explain your system

The hardware and software resources available are listed below,

Hardware Setup	
Processor	Intel® Core™ i5-7200U CPU @ 2.50GHz × 4
No. of cores	4
RAM	8 GB
OS	Ubuntu 17.10
Software Version	
Anaconda	conda 4.5.0
Python	Python 3.6.2

Exercise 1 : Basic Parallel Vectors Operations with MPI

Two vectors are available, parallel processing has to be implemented in addition and average of the two vectors. Here, process with rank 0 is considered as the root process, it initializes the data and distributes them equally to all the available processes. The remaining workers perform the operations on the data available and send the result back to the root process. Where the final answer is got by concatenating the outputs from various workers. Point to point communication is used (send and receive) to relay information between the processes. The following logic is used to equally divide the data among the workers.

```
startIndex = int((i-1)*(len(data1)/(size-1)))
endIndex = int(((len(data1)/(size-1))*i))
##print("Index",startIndex,endIndex)
#May cause index out of range due to index strting from 0, but we are counting from 1
if endIndex > len(data1):
    endIndex = int(len(data1))
```

The start and the end index are dependent on the rank of the current worker, the start index is a multiple of the worker rank and the division possible(vector size/processes). Since the process with rank 0 is used as the root process, i-1 is used. The end Index may

have an additional count as the indexing starts from 0, while the size starts from 1, the last two lines ensures that the end index is out bound.

The time taken by the processes is as follows,

Vector size : 10^4	Time required					
Workers	Process0	Process1	Process2	Process3	Total	With Overhead
1	0.00214				0.00214	0.00214
2	0.00108	0.00079			0.00079	0.00187
3	0.00134	0.00054	0.00049		0.00052	0.00186
4	0.00193	0.00052	0.00046	0.00047	0.00048	0.00241
Vector size : 10^5	Time required					
Workers	Process0	Process1	Process2	Process3	Total	With Overhead
1	0.00571				0.00571	0.00571
2	0.00883	0.00604			0.00604	0.01487
3	0.01242	0.00421	0.00384		0.00403	0.01645
4	0.01264	0.00279	0.00275	0.00265	0.00273	0.01537
Vector size : 10^6	Time required					
Workers	Process0	Process1	Process2	Process3	Total	With Overhead
1	0.03684				0.03684	0.03684
2	0.09438	0.06749			0.06749	0.16187
3	0.09658	0.03542	0.03366		0.03454	0.13112
4	0.12679	0.02901	0.02830	0.02607	0.02779	0.15458
Vector size : 10^7	Time required					
Workers	Process0	Process1	Process2	Process3	Total	With Overhead
1	0.50588				0.50588	0.50588
2	1.00845	0.73746			0.73746	1.74591
3	1.00839	0.38684	0.41807		0.40246	1.41085
4	1.26531	0.29352	0.27964	0.26451	0.27922	1.54453

The total time is just the average of the time taken at each process without considering the buffer time. With overload time includes the send and error delays caused, It can be seen that increasing the number of processes the time required for computation decreases. For average the times are similar,

Vector size : 10^5	Time required					
Workers	Process0	Process1	Process2	Process3	Total	With Overhead
1	0.00255				0.00255	0.00255
2	0.00355	0.00214			0.00214	0.00569
3	0.00354	0.00150	0.00128		0.00139	0.00493
4	0.00490	0.00102	0.00083	0.00083	0.00089	0.00579
Vector size : 10^6	Time required					
Workers	Process0	Process1	Process2	Process3	Total	With Overhead
1	0.03103				0.03103	0.03103
2	0.03472	0.03472			0.03472	0.06944
3	0.04488	0.01265	0.01132		0.01199	0.05687
4	0.04514	0.00868	0.00843	0.00707	0.00806	0.05320
Vector size : 10^7	Time required					
Workers	Process0	Process1	Process2	Process3	Total	With Overhead
1	0.19773				0.19773	0.19773
2	0.35901	0.21012			0.21012	0.56913
3	0.44188	0.11164	0.12235		0.11699	0.55887
4	0.44217	0.08171	0.08076	0.06736	0.07661	0.51878

The time for process 0 is increasing as it has to send data to more processes. For higher size of vectors, memory error occurs even with the use of sparse matrices.

Exercise 2: Parallel Matrix Vector multiplication using MPI

Here, a matrix and vector has to be multiplied. This has to be parallelized among the different processes. The vector has to be broadcast to all workers, so comm.send is used for this, the logic used to broadcast the matrix to the processors is similar to the one used above. The start and end index are used for the rows and all the columns are sent to each process. The rows and vectors are multiplied, this result is concatenated at the root and stored. The time taken for the different combinations are

Vector size : 10^2	Time required					
Workers	Process0	Process1	Process2	Process3	Total	With Overhead
1	0.01145				0.01145	0.01145
2	0.00064	0.00048			0.00048	0.00112
3	0.00075	0.00032	0.00042		0.00037	0.00112
4	0.00110	0.00034	0.00026	0.00025	0.00028	0.00138
Vector size : 10^3	Time required					
Workers	Process0	Process1	Process2	Process3	Total	With Overhead
1	0.02476				0.02476	0.02476
2	0.03095	0.01734			0.01734	0.04829
3	0.07993	0.01749	0.01572		0.01661	0.09654
4	0.11302	0.01643	0.01645	0.01523	0.01604	0.12906
Vector size : 10^4	Time required					
Workers	Process0	Process1	Process2	Process3	Total	With Overhead
1	1.34457				1.34457	1.34457
2	3.05988	1.66250			1.66250	4.72238
3	3.81990	0.89464	0.96455		0.92960	4.74950
4	4.13849	0.67999	0.66682	0.65570	0.66750	4.80599

The time can be seen to decrease with the number of workers. Here also, the time overhead can be seen at the root node. Scatter and bcast will have better efficiency than this send and receive due to the time required in synchronizing the workers. The code snippet is

```
def broadcast(matrix,vector,size):
    for i in range(1,size):

        #Evenly spacing the vector among the available workers
        startIndex = int((i-1)*(matrix.shape[0]/(size-1)))
        endIndex = int(((matrix.shape[0]/(size-1))*i))
        ##print("Index",startIndex,endIndex)
        if endIndex > matrix.shape[0]:
            endIndex = int(matrix.shape[0])
        ##print("Data Sent :",data[startIndex:endIndex])
        tic = time.time()
        comm.send(matrix[startIndex:endIndex,],dest = i)
        comm.send(vector,dest = i)
        temp = comm.recv(source = i)
        product.append(temp.tolist())
        print("Time without overhead ",time.time()-tic)
    #print("Result",sum(product,[]))


def process():
#Consider worker 0 to initialize the data and broadcast to other workers
    if rank == 0:
        matrix = np.random.rand(n,n)
        #print("Matrix",matrix)
        vector = np.random.rand(n)
        #print("Vector",vector)
        broadcast(matrix,vector,size)

    else:
        matrixRecv = comm.recv(source = 0)
        vectorRecv = comm.recv(source = 0)
        part = np.matmul(matrixRecv,vectorRecv)
        comm.send(part,dest = 0)
```

Exercise 3: Parallel Matrix Operation using MPI

Matrix multiplication can be done in parallel in several ways. The rows of one matrix can be scattered and the other matrix is broadcast, the multiplication is done at the workers and gather is used to combine the results at the root. ScatterV and GatherV are used as scatter and gather functions can only work when the number of the rows in the matrix is divisible by the number of workers available. Here the root process can also be used for computation unlike in the previous two code snippets. So the overhead time will not make a huge difference in this implementation.

```
#Scattering matrix A
comm.Scatterv([sendbuf,splitI, dispI,MPI.DOUBLE], matrixA, root=0)
#Bcasting matrix B
matrixB = comm.bcast(matrixB, root = 0)


comm.barrier()
#Performing matrix multiplication at all processes including root
temp = np.matmul(matrixA,matrixB)
#Gathering the results at root process
comm.Gatherv(temp,[pdt,split0,disp0,MPI.DOUBLE], root=0)

comm.barrier()
toc = MPI.Wtime()
```

The time required for the processes are shown below, it can be seen that the with number of processes time is decreasing and the process is getting more efficient.

Vector size : 10^2	Time required				
Workers	Process0	Process1	Process2	Process3	Total
1	0.03341				0.03341
2	0.01656	0.04761			0.03209
3	0.03546	0.03258	0.02287		0.03030
4	0.00781	0.00108	0.02720	0.08094	0.02926
Vector size : 10^3	Time required				
Workers	Process0	Process1	Process2	Process3	Total
1	0.25981				0.25981
2	0.27848	0.29942			0.28895
3	0.26671	0.27586	0.25485		0.26536
4	0.25971	0.25270	0.25813	0.24722	0.25268

The matrix 10^4 was throwing memory error therefore cannot be executed.