Parallelization of a digit recognition program

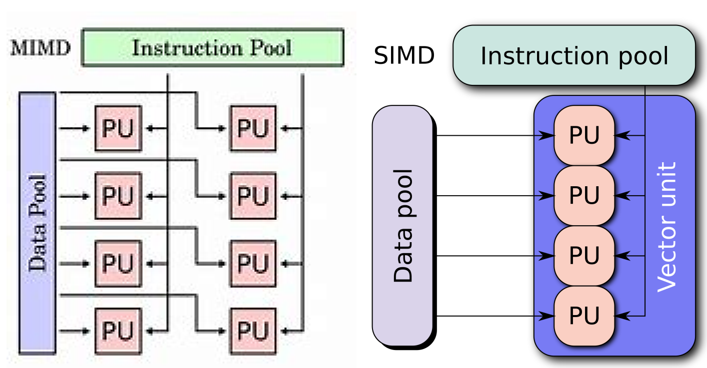
---Using python MPI---

Course : Parallel and distributed computing

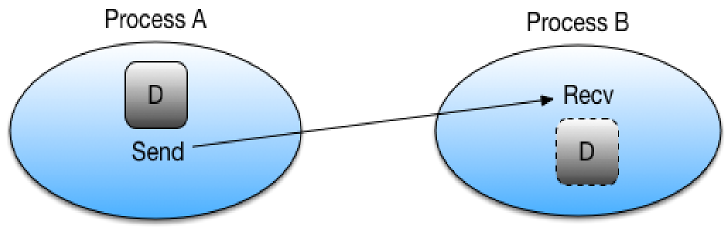
Mahima M Rao
6th Sem
Pes2201800257
Pesu ECE

MPI - Message passing interface

- MPI for Python provides an object oriented approach to message passing which grounds on the standard MPI-2 C++ bindings. The interface was designed with focus in translating MPI syntax and semantics of standard MPI-2 bindings for C++ to Python.
- Some basics of MPI
    - Commutators ---> Initiates communication between Process (nodes,host,computer cluster) Via message passing between the nodes .
    - Works for both SIMD and MIMD

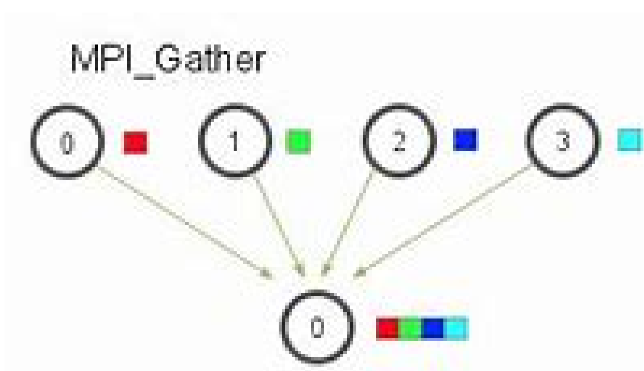


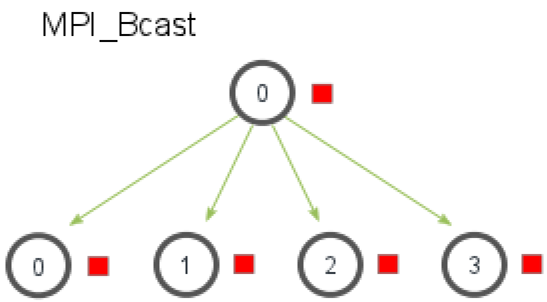
-
    - Point to point communication : -
        - Using send and receive objects
        - Can be blocking and non blocking depends on object

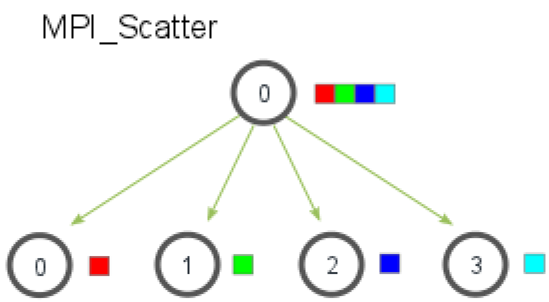


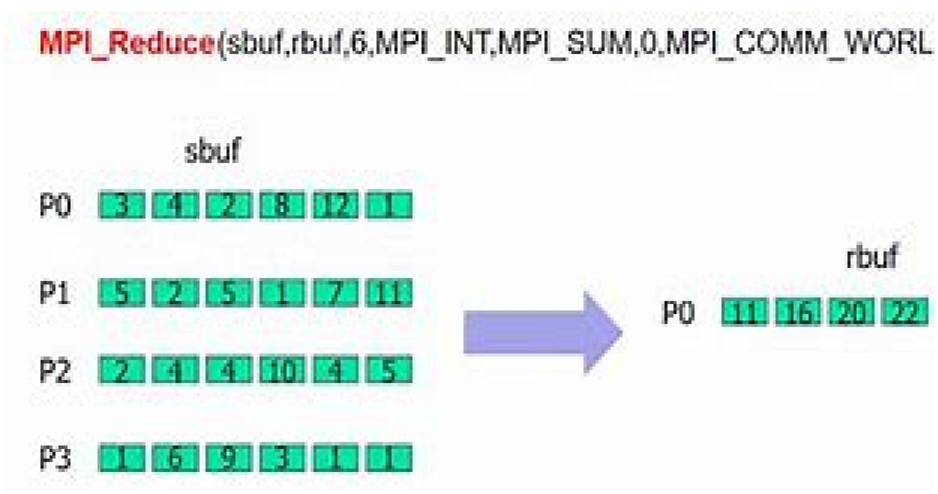
Collective Communication

- Barrier synchronization across all group members.
- Global communication functions
  - Broadcast data from one member to all members of a group.
  - Gather data from all members to one member of a group.
  - Scatter data from one member to all members of a group.
- Global reduction operations such as sum, maximum, minimum, etc
- All to all , many to many , all to one , allgather , allreduce etc







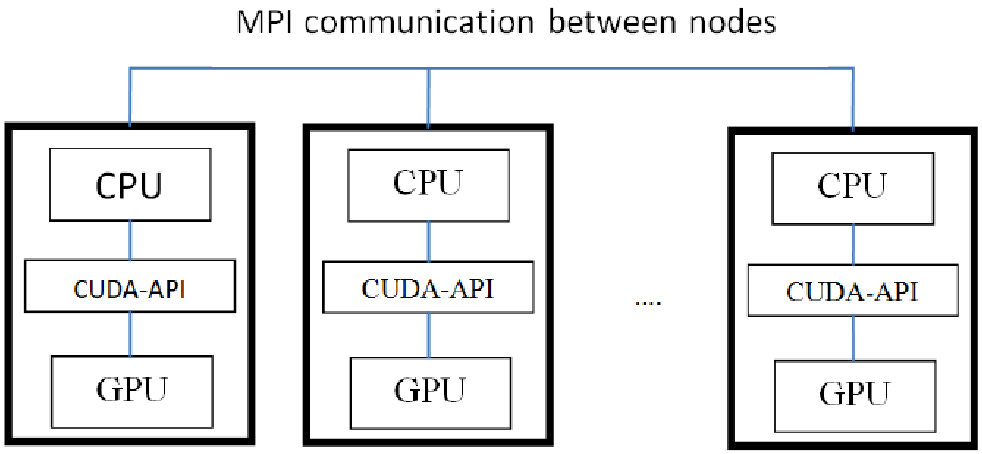


Novel paradigms are becoming more and more common:

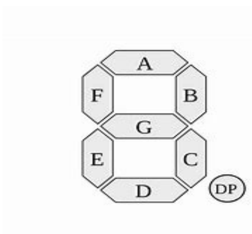
MPI+OpenMP

MPI+GPU, etc --- This can be achieved with the CUDA support for python using “cupy” class

Where in it sends data to gpu calling out treads

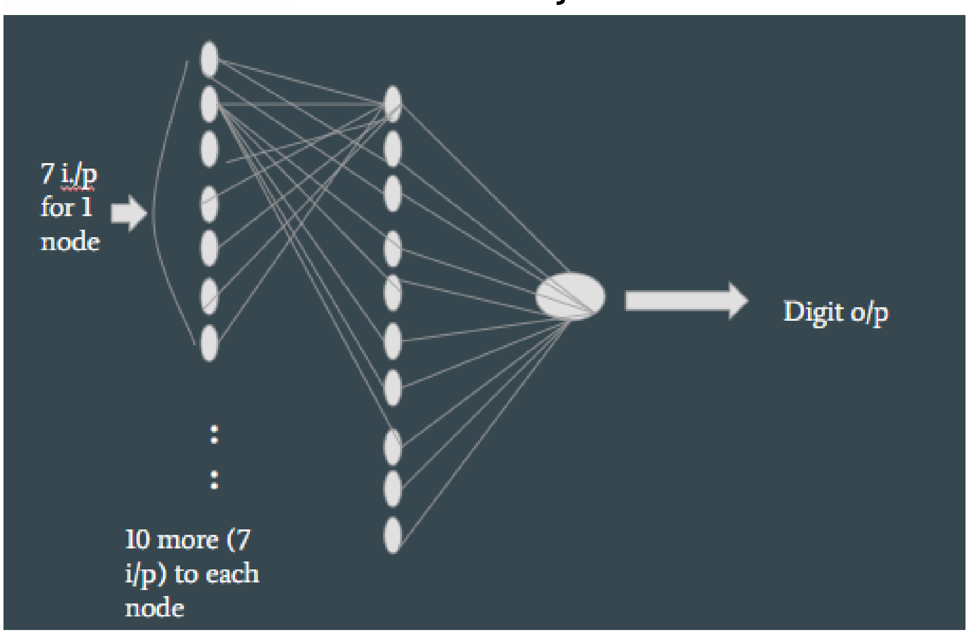


Digit recognition using perceptron learning algorithm

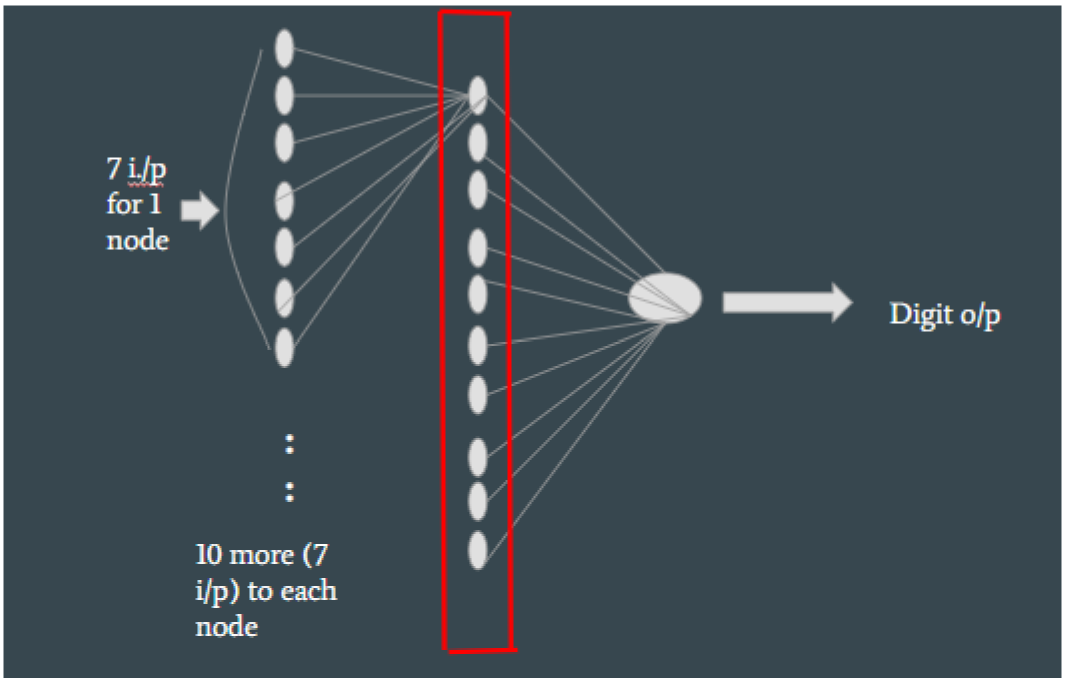


→ This is the seven segment display

Consider the fully connected network



First - training all 10 nodes parallelly



Code:

```
import mpi4py
from mpi4py import MPI
import time

digit_R=[[0,0,0,0,0,0,0],
        [1,1,0,0,0,0,0],
        [1,0,1,1,0,1,1],
        [1,1,1,0,0,1,1],
        [0,1,0,0,1,0,1],
        [0,1,1,0,1,1,1],
        [0,1,1,1,1,1,1],
```

```
        [1,1,0,0,0,0,0],
        [1,1,1,1,1,1,1],
        [1,1,1,0,1,1,1]]
```

y = [[1,0,0,0,0,0,0,0,0,0],
[0,1,0,0,0,0,0,0,0,0],
[0,0,1,0,0,0,0,0,0,0],
[0,0,0,1,0,0,0,0,0,0],   % expected output
[0,0,0,0,1,0,0,0,0,0],
[0,0,0,0,0,1,0,0,0,0],
[0,0,0,0,0,0,1,0,0,0],
[0,0,0,0,0,0,0,1,0,0],
[0,0,0,0,0,0,0,0,1,0],
[0,0,0,0,0,0,0,0,0,1]]

```
# Make a prediction with weights
def predict(row,weights):
        activation = weights[0]
        #print(len(row))
        for i in range(len(row)):
                activation += weights[i + 1] * row[i]
        return 1.0 if activation >= 0.0 else 0.0


# Estimate Perceptron weights using stochastic gradient descent
def train_weights(train,y, l_rate, n_epoch):
        weights = [0.0 for i in range(len(train[0])+1)]
        cost=[]
        for epoch in range(n_epoch):
                sum_error = 0.0
                for row,i in zip(train,y):
                        prediction = predict(row, weights)
                        error = i - prediction
                        sum_error += error**2
                        weights[0] = weights[0] + l_rate * error   #bias update
                        for i in range(len(row)):
                                weights[i + 1] = weights[i+1] + l_rate * error * row[i]
#weight update
                #cost.extend([sum_error])
        return weights
```

```
def perceptron_learning_algorithm(logic,y,name) :
    l_rate = 0.1
    n_epoch = 30
    print("----------------------%s--------------------" %(name))
    weights = train_weights(logic,y,l_rate, n_epoch)
    print("Trained weight = %s" %(weights))
    for row,i in zip(logic,y):
        prediction = predict(row, weights)
        print("Expected=%d, Predicted=%d" % (i, prediction))
    if rank == 0 :
        stop_time = time.time()
        print("Time consumed = ",stop_time - start_time)
    return(weights)


if __name__ == "__main__":

    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()                  % Parallelized part

    start_time = time.time()
    print(rank,size)

    r = perceptron_learning_algorithm(digit_R,y[rank],rank)
    a = comm.gather(r,root=0)

    print(a)
```

Results :

(mpi) C:\Users\mahim\Desktop\k mean>mpiexec -np 10 python mip_prog.py

Here i have used 10 process to parallely execute for training 10 different set of inputs
Each process are run independently

```
7 10
----------------------7--------------------
```

Trained weight = [-0.1, 0.1, 0.1, -0.1, -0.1, -0.1, -0.1, -0.1]
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=1, Predicted=1
Expected=0, Predicted=0
Expected=0, Predicted=0
None **% here 7 is expected 7 is predicted and this is executed by 7th rank process Similarly others**
5 10
----------------------5--------------------
Trained weight = [-0.2, -0.6, -0.1, 0.2, -0.6, 0.0, 0.2, 0.0]
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=1, Predicted=1
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
None
3 10
----------------------3--------------------
Trained weight = [-0.20000000000000004, 0.1, 0.0, 0.1, -0.1, -0.2, 0.1, 0.0]
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=1, Predicted=1
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0

None
2 10
----------------------2--------------------
Trained weight = [-0.1, 0.0, -0.1, 0.0, 0.1, 0.0, 0.0, 0.0]
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=1, Predicted=1
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
None
6 10
----------------------6--------------------
Trained weight = [-0.30000000000000004, -0.7999999999999999, -0.2, 0.1, 0.6, -0.1, 0.1, -0.1]
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=1, Predicted=1
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
None
1 10
----------------------1--------------------
Trained weight = [-0.2, -0.1, 0.0, -0.1, -0.1, 0.0, -0.1, -0.1]
Expected=0, Predicted=0
Expected=1, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0

```
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
None
4 10
----------------------4---------------------
Trained weight = [-0.1, -0.1, 0.0, -0.2, 0.0, 0.1, -0.2, 0.1]
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=1, Predicted=1
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
None
9 10
----------------------9---------------------
Trained weight = [-0.5, 0.20000000000000004, -0.1, 0.0, -0.2, 0.6, 0.0, -0.2]
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=1, Predicted=1
None
8 10
----------------------8---------------------
Trained weight = [-0.6, 0.20000000000000004, 0.10000000000000003,
-0.20000000000000004, 0.6, 0.4, -0.20000000000000004, -0.20000000000000004]
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
```

```
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=1, Predicted=1
Expected=0, Predicted=0
None
0 10
----------------------0--------------------
Trained weight = [0.0, -0.1, -0.1, 0.0, 0.0, 0.0, 0.0, 0.0]
Expected=1, Predicted=1
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Time consumed =  0.0065081119537353526
```

[[0.0, -0.1, -0.1, 0.0, 0.0, 0.0, 0.0, 0.0], [-0.2, -0.1, 0.0, -0.1, -0.1, 0.0, -0.1, -0.1], [-0.1, 0.0, -0.1, 0.0, 0.1, 0.0, 0.0, 0.0], [-0.20000000000000004, 0.1, 0.0, 0.1, -0.1, -0.2, 0.1, 0.0], [-0.1, -0.1, 0.0, -0.2, 0.0, 0.1, -0.2, 0.1], [-0.2, -0.6, -0.1, 0.2, -0.6, 0.0, 0.2, 0.0], [-0.30000000000000004, -0.7999999999999999, -0.2, 0.1, 0.6, -0.1, 0.1, -0.1], [-0.1, 0.1, 0.1, -0.1, -0.1, -0.1, -0.1, -0.1], [-0.6, 0.20000000000000004, 0.10000000000000003, -0.20000000000000004, 0.6, 0.4, -0.20000000000000004, -0.20000000000000004], [-0.5, 0.20000000000000004, -0.1, 0.0, -0.2, 0.6, 0.0, -0.2]]

Second - parallelizing the calculation of activation function

By performing :

Y =Y+ X*W ---in all levels

I,e parallelizing predict function as its accessed many times

Code :

% parallelized predict function

```
def predict(row,weights,rank,comm):

   a0 = weights[0]

   w = np.array(weights[1:len(row)+1])
   r = np.array(row[0:len(row)])
   a = w*r

   a1 =  comm.allreduce(a,op=MPI.SUM)


   a = sum(a1) + a0

   return 1.0 if a >= 0.0 else 0.0
```

Results

Sequential : time consumed : 0.37s

First parallel :Ran it using 10 process : 0.0043 to 0.006s

Second Parallel : 0.34 s -- using 1 process --optimal , if used more than that due to extensive process communication runtime increases.

Combining both First and Second parallel : 1.12s again the same reason , run time decreases as number of nodes increases ---> **gives out good speed up for larger input vectors**