# Assignment No.: 01

**Title:** Implement Binary Search Tree using C/C++/Java.

**Aim**: Using Divide and Conquer Strategies design a cluster/Grid of BBB or Rasberi pi or Computers in network to run a function for Binary Search Tree using C /C++/ Java/Python/ Scala

**Objectives:**

- To understand working of Cluster of BBB (BeagleBone Black)
- To understand Divide and Conquer strategy
- To implement Binary Search

**Theory:**

STEPS FOR CREATING CLUSTER OF BEAGLEBONE

Configuring the BeagleBone : Once the hardware is set up and a machine is connected to the network, Putty or any other SSH client can be used to connect to the machines. The default hostname to connect to using the above image is ubuntu-armhf. My first task was to change the hostname. I chose to name mine beaglebone1, beaglebone2 and beaglebone3. First I used the hostname command:

**sudo hostname beaglebone1**

Next I edited /etc/hostname and placed the new hostname in the file. The next step was to hard code the IP address for so I could probably map it in the hosts file. I did this by editing /etc/network/interfaces to tell it to use static IPs. In my case I have a local network with a router at 192.168.1.1. I decided to start the IP addresses at 192.168.1.51 so the file on the first node looked like this:

```
iface eth0 inet static
address 192.168.1.51
netmask 255.255.255.0
network 192.168.1.0
broadcast 192.168.1.255
gateway 192.168.1.1
```

It is usually a good idea to pick something outside the range of IPs that your router might assign if you are going to have a lot of devices. Usually you can configure this range on your router. With this done, the final step to perform was to edit /etc/hosts and list the name and IP address of each node that would be in the cluster. My file ended up looking like this on each of them:

```
127.0.0.1    localhost
192.168.1.51  beaglebone1
192.168.1.52  beaglebone2
192.168.1.53  beaglebone3
```

**Creating a Compute Cluster With MPI**

After setting up all 3 BeagleBones, I was ready to tackle my first compute project. I figured a good starting point for this was to set up MPI. MPI is a standardized system for passing messages between machines on a network. It is powerful in that it distributes programs across nodes so each instance has access to the local memory of its machine and is supported by several languages such as C, Python and Java. There are many versions of MPI available so I chose MPICH which I was already familiar with. Installation was simple, consisting of the following three steps:

**sudo apt-get update**

**sudo apt-get install gcc**

**sudo apt-get install libcr-dev mpich2 mpich2-doc**

MPI works by using SSH to communicate between nodes and using a shared folder to share data. The first step to allowing this was to install NFS. I picked beaglebone1 to act as the master node in the MPI cluster and installed NFS server on it:

**sudo apt-get install nfs-client**

With this done, I installed the client version on the other two nodes:

**sudo apt-get install nfs-server**

Next I created a user and folder on each node that would be used by MPI. I decided to call mine hpcuser and started with its folder:

**sudomkdir /hpcuser**

Once it is created on all the nodes, I synced up the folders by issuing this on the master node:

**echo "/hpcuser *(rw,sync)" | sudo tee -a /etc/exports**

Then I mounted the master's node on each slave so they can see any files that are added to the master node:

**sudo mount beaglebone1:/hpcuser /hpcuser**

To make sure this is mounted on reboots I have edited /etc/fstab and added the following:

**beaglebone1:/hpcuser /hpcusernfs**

Finally I created the hpcuser and assigned it the shared folder:

**sudouseradd -d /hpcuserhpcuser**

With network sharing set up across the machines, I installed SSH on all of them so that MPI could communicate with each:

**sudo apt-get install openssh-server**

The next step was to generate a key to use for the SSH communication. First I switched to the hpcuser and then used ssh-keygen to create the key.

**su - hpcuser**

**sshkeygen-t rsa**

When performing this step, for simplicity you can keep the passphrase blank. When asked for a location, you can keep the default. If you want to use a passphrase, you will need to take extra steps to prevent SSH from prompting you to enter the phrase. You can use ssh-agent to store the key and prevent this. Once the key is generated, you simply store it in our authorized keys collection:

**cd .ssh**

**cat id_rsa.pub >>authorized_keys**

I then verified that the connections worked using ssh:

**ssh hpcuser@beaglebone2**

**Testing MPI**

Once the machines were able to successfully connect to each other, I wrote a simple program on the master node to try out. While logged in as hpcuser, I created a simple program in its root directory /hpcuser called mpi1.c. MPI needs the program to exist in the shared folder so it can run on each machine. The program below simply displays the index number of the current process, the total number of processes running and the name of the host of the current process. Finally, the main node receives a sum of all the process indexes from the other nodes and displays it:

```
#include <mpi.h>
#include <stdio.h>
int main(intargc, char* argv[])
{
int rank, size, total;
char hostname[1024];
gethostname(hostname, 1023);
MPI_Init(&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Reduce(&rank, &total, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
printf("Testing MPI index %d of %d on hostname %s\n", rank, size, hostname);
if (rank==0)
   {
printf("Process sum is %d\n", total);
   }
MPI_Finalize();
return 0;
}
```

Next I created a file called machines.txt in the same directory and placed the names of the nodes in the cluster inside, one per line. This file tells MPI where it should run:

```
beaglebone1
beaglebone2
beaglebone3
```

With both files created, I finally compiled the program using mpicc and ran the test:

mpicc mpi1.c -o mpiprogram

mpiexec -n 8 -f machines.txt. /mpiprogram

This resulted in the following output demonstrating it ran on all 3 nodes:

```
Testing MPI index 4 of 8 on hostname beaglebone2
Testing MPI index 7 of 8 on hostname beaglebone2
Testing MPI index 5 of 8 on hostname beaglebone3
Testing MPI index 6 of 8 on hostname beaglebone1
Testing MPI index 1 of 8 on hostname beaglebone2
Testing MPI index 3 of 8 on hostname beaglebone1
Testing MPI index 2 of 8 on hostname beaglebone3
Testing MPI index 0 of 8 on hostname beaglebone1
Process sum is 28
```

**Divide and Conquer strategy**: The most well-known algorithm design strategy, given a function to compute on n inputs, the divide and conquer strategy consists of: 1. Divide the problem into two or smaller sub-problems. That is splitting the inputs into k distinct subsets, $1 \leq k \leq n$, yielding k sub-problems. 2. Conquer the sub-problems by solving them recursively. 3. Combine the solutions to the sub-problems into the solutions for the original problem. 4. If the sub-problems are relatively large, then divide Conquer is applied again. 5. If the sub-problems are small, then sub-problems are solved without splitting.

**Algorithm for Binary Search**:

1. Algorithm Bin search(a,n,x)

2. //Given an array a[1:n] of elements in non-decreasing

3. //order, n>=0,determine whether „x" is present and

4. // if so, return „j" such that x=a[j]; else return 0.

5.{ low:=1;high:=n;

6. while(low<=high) do

7. {mid:=[(low+high)/2];

8. if(x<a[mid])then high;

9. else if(x>a[mid]) then low=mid+1;

10. else return mid; }

```
11. return 0;}
```

**Mathematical Model**:

Binary Search is used for searching of elements from given data.

Following parameters are used for Binary search tree:

M= {s, e, i, o, n, F, Success, Failure}

s = Start state = inserting numbers to array to be sorted in BST manner.

e = End state = key element found & displayed.

i is the set of input elements and key element.

o is the required output i.e. position of the key element.

n = Number of processes = {hostname of processors in a file}

F is the set of functions required for BST search = {f1, f2}

f1 = {Sort the given input in the form of Binary Search Tree}

f2 = {Searching of the key element}

i= {a1, a2, a3,……., an}. = Array of elements to be sorted in BST.

o={Postion of the key element}

Success – Sorted list of elements by BST and keyt element found..

Failure-ϕ

**Conclusion**: We have studied the concept of divide and conquer strategy and binary search tree algorithm is implemented on Grid of BBB.

Code

```
#include<stdio.h>
#include<mpi.h>
#include<stdlib.h>

struct BSTNode
{
   int data;
   struct BSTNode *left;
   struct BSTNode *right;
};

//Inserting element in BST
struct BSTNode *Insert(struct BSTNode *root, int data)
{
   if(root == NULL)
   {
       root = (struct BSTNode *) malloc (sizeof(struct BSTNode));

       if(root == NULL)
       {
           printf("Memory Error");
           return;
       }
       else
       {
           root -> data = data;
           root -> left = root -> right = NULL;
       }
   }
   else
   {
       if(data < root -> data)
           root -> left = Insert(root -> left, data);
       else if(data > root -> data)
           root -> right = Insert(root -> right, data);
   }
   return root;
}
//int p=0
//Inorder
void inorder(struct BSTNode *root,int *arr){
    if(root){
         inorder(root -> left,arr);
        printf("%d\t", root -> data);
 //        arr[p] = root -> data;
   //      p++;
         inorder(root -> right,arr);
    }
}

int main(int argc,char *argv[])
{

   int a[10] = {1,6,8,3,5,2,4,9,7,0};
   int i,rank,size,b[10],search;

   printf("\n Insert the key element to be searched : ");
   scanf("%d",&search);
   printf("\n You entered : %d\n",search);

   MPI_Request request;
   MPI_Status status;
```

```
/*  int flag;
  int flag1=0;
  int flag2=0;
  */
  MPI_Init(&argc,&argv);
       MPI_Comm_rank(MPI_COMM_WORLD,&rank);
       MPI_Comm_size(MPI_COMM_WORLD,&size);

       MPI_Scatter(&a,5,MPI_INT,&b,5,MPI_INT,0,MPI_COMM_WORLD);

       if(rank == 0)
       {

          struct BSTNode *root_1 = NULL;
          for(i=0;i<5;i++)
          {
               root_1=Insert(root_1, b[i]);
          }
          if (root_1 != NULL)
          {
               printf("\nInorder at rank-%d processor:\t",rank);
           inorder(root_1,b);
       }
       int flag=0;
       int flag1=0;
       MPI_Irecv(&flag,1,MPI_INT,1,3,MPI_COMM_WORLD,&request);
       while(root_1)
       {
           if(flag ==1)
           {
               break;
           }
           if(search == root_1 -> data)
           {
               printf("\nkey %d found at rank-%d processor\n",search,rank);
               flag1=1;
               MPI_Send(&flag1,1,MPI_INT,1,2,MPI_COMM_WORLD);
               break;
           }
           else if(search > root_1 -> data)
               root_1 = root_1 -> right;
           else
               root_1 = root_1 -> left;
       }
       MPI_Send(&flag1,1,MPI_INT,1,2,MPI_COMM_WORLD);
       MPI_Wait(&request,&status);
       if(flag ==0 && flag1 ==0)
       {
           printf("\nKey %d not found\n",search);
       }
       }

       if(rank == 1)
       {
          struct BSTNode *root_2 = NULL;
          for(i=0;i<5;i++)
          {
              root_2=Insert(root_2, b[i]);
          }
          if (root_2 != NULL)
          {
              printf("\nInorder at rank-%d processor:\t",rank);
            inorder(root_2,b);
       }
```

```
        int flag=0;
        int flag1=0;
        MPI_Irecv(&flag,1,MPI_INT,0,2,MPI_COMM_WORLD,&request);
        while(root_2)
        {
            if(flag ==1)
            {
                break;
            }
            if(search == root_2 -> data)
            {
                printf("\nkey %d found at rank-%d processor\n",search,rank);
                flag1=1;
                MPI_Send(&flag1,1,MPI_INT,0,3,MPI_COMM_WORLD);
                break;
            }
            else if(search > root_2 -> data)
                root_2 = root_2 -> right;
            else
                root_2 = root_2 -> left;
        }
        MPI_Send(&flag1,1,MPI_INT,0,3,MPI_COMM_WORLD);
        }

        MPI_Finalize();
        return 0;
}
```

Machinefile

```
ub0:20

ub1:20
```

Output

```
mpiu@DB21:/mirror/mpiu/CL4 prog/A-1$ mpicc bst_mpi.c



mpiu@DB21:/mirror/mpiu/CL4 prog/A-1$ mpiexec -f machinefile -n 4 ./a.out


 Insert the key element to be searched :

 You entered : 0


 Insert the key element to be searched :

 You entered : 0


 Insert the key element to be searched :

 You entered : 0
```

```
6


 Insert the key element to be searched :
 You entered : 6


Inorder at rank-0 processor:
Inorder at rank-1 processor: 0      2      4      7      9
key 0 found at rank-1 processor
1      3      5      6      8
key 6 found at rank-0 processor
mpiu@DB21:/mirror/mpiu/CL4 prog/A-1$
```