Assignment No.: 05

Title: Implement Booth Multiplication algorithm.

Aim: Build a small compute cluster using Raspberry Pi/BBB modules to implement booth Multiplication algorithm.

Objectives:

To understand working of Cluster of BBB (BeagleBone Black)

Implement booths multiplication algorithm.

Theory:

STEPS FOR CREATING CLUSTER OF 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 byediting/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
```

Booth Multiplication algorithm:: Booth's algorithm examines adjacent pairs of bits of the N-bit multiplier Y in signed two's complement representation, including an implicit bit below the least significant bit, y -1 = 0. For each bit y i , for i running from 0 to N- 1, the bits y i and y i-1 are considered. Where these two bits are equal, the product accumulator P is left unchanged.

Where y i = 0 and y i-1 = 1, the multiplicand times 2 i is added to P; and where y i = 1 and y i-1 = 0, the multiplicand times 2 i is subtracted from P. The final value of P is the signed product.

The multiplicand and product are specified; typically, these are both also in two's complement representation, like the multiplier, but any number system that supports addition and subtraction will work as well.

As stated here, the order of the steps is not determined. Typically, it proceeds from LSB to MSB, starting at i = 0; the multiplication by 2 i is then typically replaced by incremental shifting of the P accumulator to the right between steps; low bits can be shifted out, and subsequent additions and subtractions can then be done just on the highest N bits of P.

There are many variations and optimizations on these details.

The algorithm is often described as converting strings of 1's in the multiplier to a high-order +1 and a low- order –1 at the ends of the string. When a string runs through the MSB, there is no high-order +1, and the net effect is interpretation as a negative of the appropriate value.

Algorithm :

Implementation:

Booth's algorithm can be implemented by repeatedly adding (with ordinary unsigned binary addition) one of two predetermined values A and S to a product P, then performing a rightward arithmetic shift on P.

Let m and r be the multiplicand and multiplier, respectively; and let x and y represent the number of bits in m and r.

1. Determine the values of A and S, and the initial value of P. All of these numbers should have a length equal to (x + y + 1).

    1. A: Fill the most significant (leftmost) bits with the value of m. Fill the remaining (y + 1)bits with zeros.

    2. S: Fill the most significant bits with the value of (−m) in two's complement notation. Fill the remaining (y + 1) bits with zeros.

    3. P: Fill the most significant x bits with zeros. To the right of this, append the value of r. Fill the least significant (rightmost) bit with a zero.

2. Determine the two least significant (rightmost) bits of P.

    1. If they are 01, find the value of P + A. Ignore any overflow.

    2. If they are 10, find the value of P + S. Ignore any overflow.

    3. If they are 00, do nothing. Use P directly in the next step.

    4. If they are 11, do nothing. Use P directly in the next step.

3. Arithmetically shift the value obtained in the 2nd step by a single place to the right. Let P now equal this new value.

4. Repeat steps 2 and 3 until they have been done y times. Drop the least significant (rightmost) bit from P. This is the product of m and r.

Mathematical Model:

Let, S be the System Such that,

A={S, E, I, O, F, DD, NDD, F_min, F_fri, CPU_Core, Mem_Shared, success, failure} Where,

S= Start state,

E= End State,

I= Set of Input

O= Set of Out put

F =Set of Function

DD=Deterministic Data

NDD=Non Deterministic Data

F_Min=Main Function

F_Fri= Friend Function

CPU_Core= No of CPU Core.

Mem_Shared=Shared Memory.

Function:

1) Splitting Function = this function is used for splitting unsorted list.

2) Sorting Function = this function is used for sorting list.

3) Binary Search = this function apply binary search on sorted list.

Success Case: It is the case when all the inputs are given by system are entered correctly. Failure Case: It is

the case when the input does not match the validation Criteria.

Conclusion: Thus we have implemented booths algorithm using small cluster of BBB.

Steps to be followed for successful exection of program

Prerequisites:

First disable the firewall on the machine used for running server code.

Check the IP addres and port number in socket's connect() and bind() functions in all the files.

Make sure all the code have the IP and port number of the machine running the server code.

Steps:

Run the server code first and enter the multiplicand an multiplier in integer base 10 format.(The usual numbers like -5, 3 ,4...)

Run the multiplicand code next on another machine.

Run the multiplier code on yet another machine.

1. The multiplicand code will display the values of A and S. It will also display the packed string which is sent back to the server.

2. The multiplier code will display the value of P

3. The server code will display the connection received from both the clients. Finally, it also outputs the answer in binary format.

Code:

server_booth.py

```
# take values A,S from clients and run the main loop for calculating P

import socket

def addition(op1, op2):
        # length of P and A and S is same.
        result=""
        carry="0"
        for i in range(len(op1)-1, -1, -1):         #run reverse loop
                if op1[i]=="1" and op2[i]=="1":
                        if carry=="1":
                                result="1"+result
                                carry="1"
                        else:                                   #carry = 0
                                result="0"+result
                                carry="1"

                elif op1[i]=="0" and op2[i]=="0":
                        if carry=="1":
                                result="1"+result
                                carry="0"
                        else:                                   #carry = 0
                                result="0"+result
                                carry="0"

                elif op1[i]=="0" and op2[i]=="1":
                        if carry=="1":
                                result="0"+result
                                carry="1"
                        else:                                   #carry = 0
                                result="1"+result
                                carry="0"

                else:                                           # 1, 0
                        if carry=="1":
                                result="0"+result
                                carry="1"
                        else:                                   #carry = 0
                                result="1"+result
                                carry="0"
        return result

s = socket.socket()         # Create a socket object
s.bind(("192.168.6.80", 9001))        # Bind to the port


M=int(input("Enter a multiplicant:"))
R=int(input("Enter a  multiplier:"))
M, R=bin(M), bin(R)
print "Binary representation: ", M, R

s.listen(2)                     # Now wait for client connection.
client, addr = s.accept()     # Establish connection with client.
print 'Got connection from', addr

client2, addr2 = s.accept()
print 'Got connection from', addr2

'''
Send the value of both. Client will return A, S. It will also return length_R as first param.
<Length_R>A<A>S<S>
```

```
Send the value of length of R and value of R. Client will return P.      P<P>
'''
client.send(M+" "+R)

AS=client.recv(1024)  # recv A, S
index_A=AS.index("A")
index_S=AS.index("S")
A=AS[index_A+1:index_S]
S=AS[index_S+1:]

length_R=int(AS[:index_A])
client2.send(str(length_R)+" "+R)
P=client2.recv(1024)  # recv P
index_P=P.index("P")
P=P[index_P+1:]
P_length=len(P)

#we've got A,S,P in strings
for i in range(length_R):

        last_two_digits=P[P_length-2:P_length]

        if last_two_digits == "01":
                #add A in P and store that in P and ignore overflows
                P=addition(A, P)
        elif last_two_digits == "10":
                #add S in P aND store the result in P and IGNORE OVerflows
                P=addition(S, P)

        #print "After addn", P
        #arithmetic right shift (copy the sign bit as well). Start looping from the right
most digits
        P=P[0]+P[0:P_length-1]

P=P[:P_length-1]
print P


client _multiplier.py
# calculate value P and send it back to server.

import socket

def twos_comp(binM):
        S = [int(x) for x in binM]
        flag = 0
        for i in range(len(S)-1, -1, -1):
                if flag==1:
                        #invert
                        if S[i]==1:
                                S[i]=0
                        else:
                                S[i]=1
                        continue

                if S[i]==1:
                        flag=1
        return S

s = socket.socket()         # Create a socket object
s.connect(("192.168.6.80", 9001))
temp=s.recv(1024)
```

```
temp=temp.split()
length_R, R= int(temp[0]), temp[1]

if R[0]=="-":
        R=R[3:]
        #origR=R
        R=twos_comp(R)
        R=[str(x) for x in R]
        R=''.join(R)
        for i in range (length_R-len(R)):
                R="1"+R
else:
        R=R[2:]
        for i in range (length_R-len(R)):
                R="0"+R
        #flag_R=2

P = []
for i in range(2*length_R + 1):
        P.append(0)

print "check length of P: ", P

for i in range(len(R)):
        P[length_R+i]=int(R[i])

P=[str (x) for x in P]
P="".join(P)
print P
s.send("P"+P)
```

Client_multiplicand.py

```
# calculate values A, S and send it back to server.

import socket

def twos_comp(binM):
        S = [int(x) for x in binM]
        flag = 0
        for i in range(len(S)-1, -1, -1):
                if flag==1:
                        #invert
                        if S[i]==1:
                                S[i]=0
                        else:
                                S[i]=1
                        continue

                if S[i]==1:
                        flag=1
        return S

s = socket.socket()         # Create a socket object
s.connect(("192.168.6.80", 9001))
temp=s.recv(1024)
temp=temp.split()
M, R=temp[0], temp[1]
origM, origR="", ""
Max_length=0

flag,flag_R=0,0                     # flag=1: -M, flag=2: M. flag_R=1: -R, flag_R=2: R
```

```
if M[0]=="-":
        M=M[3:]
        origM=M
        M=twos_comp(M)
        M=[str(x) for x in M]
        M=''.join(M)
        flag=1
else:
        M=M[2:]
        flag=2

if R[0]=="-":
        R=R[3:]
        origR=R
        R=twos_comp(R)
        R=[str(x) for x in R]
        R=''.join(R)
        flag_R=1
else:
        R=R[2:]
        flag_R=2

if len(M)>= len(R):
        padding=len(M)-len(R)+1 #+1 for sign bit
        if flag==1:
                M="1"+M
        else:
                M="0"+M
        for i in range (padding):
                if flag_R==1:
                        R="1"+R
                else:
                        R="0"+R
        Max_length=len(M)
else:
        padding=len(R)-len(M)+1
        if flag_R==1:
                R="1"+R
        else:
                R="0"+R
        for i in range (padding):
                if flag==1:
                        M="1"+M
                else:
                        M="0"+M
        Max_length=len(R)

print M, R

#now calc A, S using the length of M and R and 1 (lenM+lenR+1)
A = []
for i in range(len(M)+len(R)+1):
        A.append(0)

for i in range(len(M)):
        A[i]=int(M[i])
A=[str(x) for x in A]
print "A: ", A
#A is ready at this point

if flag==1:             # orignal M was -ve. So we need origM with the minus sign eliinated
        for i in range(Max_length-len(origM)):
                origM="0"+origM
        S=[str(x) for x in origM]
```

```
else:
        S=twos_comp(M)

for i in range(len(M)+len(R)+1-len(S)):
        S.append(0)

S=[str(x) for x in S]

#S is ready at this point
print "S: ", S

#pack the A ans S in a buffer string
Send_AS= str(len(R))+"A"+''.join(A) #secret- length of both operands is same. So u can re-
place R with M
Send_AS += "S"+''.join(S)
print Send_AS

#send the A and S to server and the job here is done
s.send(Send_AS)
```

OUTPUT

```
Server side :

root@Student-301:/home/student/Documents/A-5 Booth's Algorithm# python server_booth.py

Enter a multiplicant:5

Enter a  multiplier:2

Binary representation:  0b101 0b10

Got connection from ('192.168.6.67', 36176)

Got connection from ('192.168.6.79', 36224)

00001010

root@Student-301:/home/student/Documents/A-5 Booth's Algorithm#
```

```
Client1 :

root@student-302:/home/student/Documents# python client_multiplier.py check length of P:  [0,
0, 0, 0, 0, 0, 0, 0, 0]

000000100

root@student-OptiPlex-3010:/home/student/Documents#
```

Client2 :

```
root@student-303:/home/student/Documents# python client_multiplicand.py
0101 0010
A:  ['0', '1', '0', '1', '0', '0', '0', '0', '0']
S:  ['1', '0', '1', '1', '0', '0', '0', '0', '0']
4A010100000S101100000
root@student-303:/home/student/Documents#
```