

AIM: Huffman Coding using Message Passing Interface

Definition of Huffman Coding :

Huffman coding assigns codes to characters such that the length of the code depends on the relative frequency or the weight of the corresponding character. Huffman codes are of variable-length, and prefix-free (no code is a prefix of any other). Any prefix-free binary code can be visualized as a binary tree with the encoded characters stored at the leaves.

Huffman coding tree or **Huffman tree** is a full binary tree in which each leaf of the tree corresponds to a letter in the given alphabet.

Define the weighted path length of a leaf to be its weight times its depth. The Huffman tree is the binary tree with minimum external path weight, i.e., the one with the minimum sum of weighted path lengths for the given set of leaves. So the goal is to build a tree with the minimum external path weight.

For Example :

BAGGAGE \longrightarrow 100 11 0 0 11 0 101

Major Steps :

- 1) Prepare the frequency table
- 2) Construct the binary tree
- 3) Extract Huffman code from the tree

Algorithm:

$N \leftarrow |C|$

$Q \leftarrow C$

For $i=0$ to $n-1$

Do

 Allocate a new node z

 Left $[z] \leftarrow x \leftarrow \text{EXTRACT-MIN}(Q)$


 Right $[z] \leftarrow y \leftarrow \text{EXTRACT-MIN}(Q)$

$f[z] \leftarrow f[x] + f[y]$

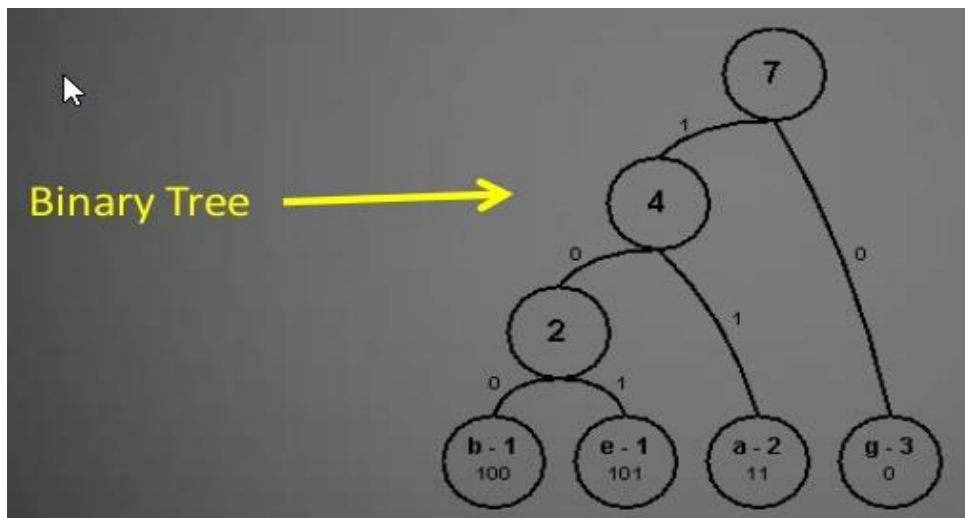
 INSERT (Q, Z)

END FOR

Return EXTRACT-MIN (Q)



| Alphabets | Frequencies |
|-----------|-------------|
| b | 1 |
| a | 2 |
| g | 3 |
| e | 1 |



Time Complexity:

$O(n^3)$

Where each iteration requires $O(n^2)$ time to determine lowest weight (by sorting) and $O(n)$ for n iterations.

Two parts of implementation:

- 1) Sort the tree every time
- 2) After binary tree is created, get the code

Problems:

- 1) Number of processes are more or less than the number of elements
 - If the number of processes is more than required, then extra processes must have to idealize
 - If the number of processes is less than required, then they have to work for multiple times
- 2) Barriers are required
- 3) Unexpected waiting times may occur

Parallel implementation:

As there are a lot of dependencies in the creation of the tree, tree implementation in parallel is difficult. Only sorting can be implemented in parallel.

Implementation of Huffman coding in parallel using MPI :

Source code:

```
#include<stdio.h>
#include<mpi.h>
#include<stddef.h>
#include<stdlib.h>
#include<string.h>
int n=50, bsize=50, size, rank;
struct List
{
    char code[15];
    int inst;
    int k;
    float pri;
}first[200];
void cmpswap(struct List l[]);
void sort(int strt,int end);
void crtList(int n);
void getCode();
void print(int end);
char processor_name[MPI_MAX_PROCESSOR_NAME];
int main (int argc, char* argv[])
{
    int i=0, namelen;
    MPI_Status status;
    MPI_Init (&argc, &argv);    /* starts MPI */
    MPI_Comm_rank (MPI_COMM_WORLD, &rank); /* get current process id */
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(processor_name, &namelen);
    if(rank==0)
    {
        bsize=n;
        crtList(n);
```

```

        print(n);
    }
    i=0;
    while(i<n-1)
    {
        MPI_Barrier(MPI_COMM_WORLD);
        sort(2*i,n+i);//parrallel
        MPI_Barrier(MPI_COMM_WORLD);
        if(rank==0)
        {
            printf("\n strt:%d  end:%d",2*i,n+i+1);
            print(n+i);
            strcpy(first[i+n].code,"");
            first[i+n].pri=first[2*i].pri+first[2*i+1].pri;
            first[i+n].inst=i+n+1;
            printf("\ninst:%d  %f inst:%d  %f inst:%d  %f\n",first[2*i].inst,first[2*i].pri,first[2*i+1].inst,first[2*i+1].pri,first[i+n].inst,first[i+n].pri);
        }
        MPI_Barrier(MPI_COMM_WORLD);
        i++;
    }
    MPI_Barrier(MPI_COMM_WORLD);
    sort(2*i,n+i);
    MPI_Barrier(MPI_COMM_WORLD);
    if(rank==0)
    {
        getCode();
    }
    MPI_Finalize();
    if(rank==0)
    {
        printf("\n\n final\n");
        print(n+i);
    }
    return 0;
}

```

```

void getCode()
{
    int i,k=2*(n-2);
    strcat(first[2*n-1].code,"");
    for(i=2*(n-1);i>=0;i--)
    {
        if(first[i].inst>n)
        {
            strcpy(first[k].code,first[i].code);
            strcpy(first[k+1].code,first[i].code);
            strcat(first[k].code,"0");
            strcat(first[k+1].code,"1");
            k=k-2;
        }
    }
}

```

```

void cmpswap(struct List l[])
{
    struct List t;
    if(l[0].pri>l[1].pri)
    {
        t.inst=l[0].inst;
        strcpy(t.code,l[0].code);
        t.pri=l[0].pri;
        l[0].inst=l[1].inst;
        strcpy(l[0].code,l[1].code);
        l[0].pri=l[1].pri;
        l[1].inst=t.inst;
        strcpy(l[1].code,t.code);
        l[1].pri=t.pri;
    }
}

```

```

void sort(int strt,int end)
{
    struct List l[2];
    int in,li,flag=0,out,j,k,i,b=0,x=0;
    for(j=strt;j<end;)

```

```

{

if(rank==0)
{
    if(j%2==0)
    {
        k=strt;
        while(k<end-1)
        {
            for(i=1; (i<size && k<end-1);i=i+1)
            {
                first[k].k=first[k+1].k=k;
                MPI_Send( &first[k],sizeof(struct List)*2,MPI_BYTE,i,0,
                MPI_COMM_WORLD);//send obj
                k+=2;
            }
            k=k-2*(i-1);
            for(i=1;(i<size&&k<end-1);i++)
            {
                MPI_Status status;
                MPI_Recv(l,sizeof(struct List)*2,MPI_BYTE,
                MPI_ANY_SOURCE,0, MPI_COMM_WORLD, &status);
                printf("\nMaster:%s", processor_name);
                first[l[0].k].inst=l[0].inst;
                strcpy(first[l[0].k].code,l[0].code);
                first[l[0].k].pri=l[0].pri;
                first[l[0].k+1].inst=l[1].inst;
                strcpy(first[l[0].k+1].code,l[1].code);
                first[l[0].k+1].pri=l[1].pri;
                k+=2;
            }
        }
    }
    else
    {
        k=strt+1;
        while(k<end-1)

```

```

        {
            for(i=1;i<size&&k<end-1;i=i+1)
            {
                first[k].k=first[k+1].k=k;
                MPI_Send(&first[k],sizeof(struct List)*2,MPI_BYTE,i,0,
                MPI_COMM_WORLD);
                MPI_Status status;
                k+=2;
            }
            k=k-2*(i-1);
            for(i=1;i<size &&k<end-1;i++)
            {
                MPI_Status status;
                MPI_Recv(l,sizeof(structList)*2,MPI_BYTE
                ,MPI_ANY_SOURCE,0, MPI_COMM_WORLD, &status);
                printf("\nMaster:%s", processor_name);

                first[l[0].k].inst=l[0].inst;
                strcpy(first[l[0].k].code,l[0].code);
                first[l[0].k].pri=l[0].pri;

                first[l[0].k+1].inst=l[1].inst;
                strcpy(first[l[0].k+1].code,l[1].code);
                first[l[0].k+1].pri=l[1].pri;
                k+=2;
            }
        }
    }
}
else
{
    if(rank<=end/2)
    {
        li=strt+j%2;
        while(end-li>1)
        {
            if(rank<=(end-li)/2)

```

```

        {
            MPI_Status status;
            MPI_Recv(1,sizeof(struct List)*2,MPI_BYTE,0,0,
            MPI_COMM_WORLD, &status);
            printf("\nchild::%s", processor_name);

            cmpswap(1);
            MPI_Send(1,sizeof(structList)*2,MPI_BYTE,0,0,
            MPI_COMM_WORLD);
        }

        if(size<=end)
        {
            li=li+2*(size-1);//chk for more than 9;

        }
        else
        {
            break;
        }
    }

}

MPI_Barrier(MPI_COMM_WORLD);
j++;
}

}

void crtList(int n)
{
    int i;
    float g;
    for(i=0;i<n;i++)
    {
        strcpy(first[i].code,"");
        first[i].inst=i+1;
        first[i].pri=((float)((200-i)))/10;
        printf("Enter Inst priority for inst %f :",first[i].pri);
    }
}

```

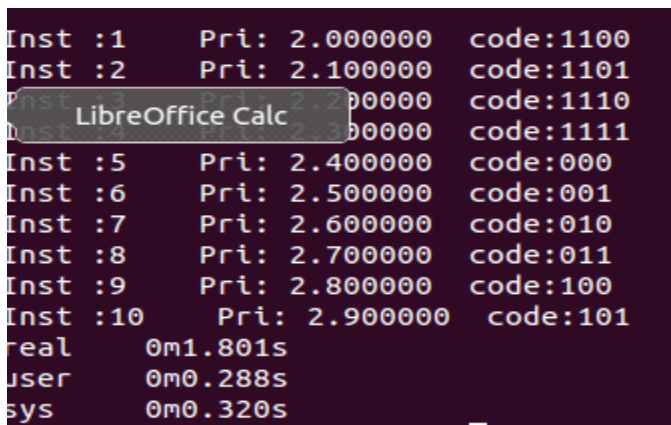


```

}
void print(int end)
{
    int i;
    printf("\n\n");
    for(i=0;i<end;i++)
    {
        printf("\nInst :%d  Pri: %f  code:%s",first[i].inst,first[i].pri,first[i].code);
    }
}

```

Output:



```

Inst :1      Pri: 2.000000  code:1100
Inst :2      Pri: 2.100000  code:1101
Inst :3      Pri: 2.200000  code:1110
Inst :4      Pri: 2.300000  code:1111
Inst :5      Pri: 2.400000  code:000
Inst :6      Pri: 2.500000  code:001
Inst :7      Pri: 2.600000  code:010
Inst :8      Pri: 2.700000  code:011
Inst :9      Pri: 2.800000  code:100
Inst :10     Pri: 2.900000  code:101
real        0m1.801s
user        0m0.288s
sys         0m0.320s

```

Applications of Huffman Coding:

- ZIP(multichannel compression including text and other data types)
- JPEG-Joint Photographic Experts Group
- MPEG-Moving Picture Experts Group (only upto 2 layers)
- Used in Stenganography for JPEG carrier compression.

Conclusion:

Huffman coding is required for compression of data so that it could be transmitted over the internet and other transmission channel properly.

