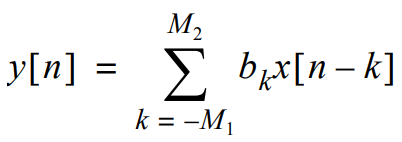
### **Introduction to FIR :**

In signal processing, a finite impulse response (FIR) filter is a filter whose impulse response (or response to any finite length input) is of finite duration, because it settles to zero in finite time. If you put in an impulse, that is, a single “1” sample followed by many “0” samples, zeroes will come out after the “1” sample has made its way through the delay line of the filter.

The term finite impulse response arises because the filter output is computed as a weighted, finite term sum, of past, present, and perhaps future values of the filter input, i.e.,



where both M1 and M2 are finite.

An FIR filter is designed by finding the coefficients and filter order that meet certain specifications, which can be in the time domain (e.g. a matched filter) and/or the frequency domain (most common). Matched filters perform a cross-correlation between the input signal and a known pulse shape. The FIR convolution is a cross-correlation between the input signal and a time-reversed copy of the impulse response. Therefore, the matched filter's impulse response is "designed" by sampling the known pulse-shape and using those samples in reverse order as the coefficients of the filter.

One of the simplest FIR filters we may consider is a 3-term moving average filter of the form

y[n] = (x[n+1] + x[n] + x[n-1])

An FIR filter is based on a feed-forward difference equation as demonstrated by example above

* Feed-forward means that there is no feedback of past or future outputs to form the present output, just input related terms.

Impulse response is the reaction of any dynamic system in response to some external change. E.g. a ball suspended by a spring, when wacked with a bat, responds with a harmonic motion of gradually decreasing amplitude.

### **Proposed method :**

Considering the same 3-term moving average filter for our project. Calculating y[i] for all i[0, N] will require the input of i, i-1 and i+1. Since the output has no dependence on any other output, the parallelization can be directly done by using MPI library. Correctly passing the arguments and distributing the task will do the job. The length of array is taken to be 1 Million. Further for the first and last discrete value, separate calculation is done by the master itself as they have a 3rd term missing. Load balancing has been done to make each worker get equal work and the extra work is done by the Master.

### **Appendix - C Code :**

/\*

High Performance Computing Project

Finite Impulse Response using MPI

- Akshay Kumar (CED15I031)

The FIR function under consideration is

y[n] = (x[n+1] + x[n] + x[n-1]) / 3

\*/

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#define ARRAY\_SIZE 100000 // Array Size

#define MASTER 0

#define FROM\_MASTER 1

#define FROM\_WORKER 2

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

{

int numtasks,

taskid,

numworkers,

source,

dest,

mtype,

aver\_size,

extra,

offset,

i, rc;

double x[ARRAY\_SIZE],

y[ARRAY\_SIZE];

srand (time (NULL));

MPI\_Status status;

MPI\_Init (&argc, &argv);

MPI\_Comm\_rank (MPI\_COMM\_WORLD, &taskid);

MPI\_Comm\_size (MPI\_COMM\_WORLD, &numtasks);

if (numtasks < 2)

{

printf ("need atleast two mpi tasks, quitting...\n");

MPI\_Abort (MPI\_COMM\_WORLD, rc);

exit (1);

}

numworkers = numtasks - 1;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* master task \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if (taskid == MASTER)

{

printf ("FIR Filter (3 point Average) has started with %d tasks.\n", numtasks);

printf ("\n\t\t\tInitializing Input Array (x) started...\n");

for (i = 0; i < ARRAY\_SIZE; i++)

x[i] = rand () % 256;

printf ("\n\t\t\tInitializing Input Array (x) finished...\n\n");

// calculating terms for sending data and task distribution

aver\_size = (ARRAY\_SIZE - 2) / numworkers;

extra = (ARRAY\_SIZE - 2) % numworkers;

offset = 0;

mtype = FROM\_MASTER;

// the first and last element being exceptions are handled by master only

// also extra work handled by master for Load Balancing

y[0] = (x[0] + x[1]) / 3;

y[ARRAY\_SIZE - 1] = (x[ARRAY\_SIZE - 2] + x[ARRAY\_SIZE -1]) / 3;

for (i = 1; i <= extra; i++)

{

y[i] = (x[i-1] + x[i] + x[i+1]) / 3;

}

offset = extra + 1;

// sending matrix data to the worker tasks

for (dest = 1; dest <= numworkers; dest++)

{

printf ("Sending %d terms of 'y' to task %d offset = %d\n", aver\_size, dest, offset);

MPI\_Send (&offset, 1, MPI\_INT, dest, mtype, MPI\_COMM\_WORLD);

MPI\_Send (&aver\_size, 1, MPI\_INT, dest, mtype, MPI\_COMM\_WORLD);

MPI\_Send (&x[offset - 1], aver\_size + 2, MPI\_DOUBLE, dest, mtype, MPI\_COMM\_WORLD);

offset = offset + aver\_size;

}

// reveive results from worker tasks

mtype = FROM\_WORKER;

for (source = 1; source <= numworkers; source++)

{

MPI\_Recv (&offset, 1, MPI\_INT, source, mtype, MPI\_COMM\_WORLD, &status);

MPI\_Recv (&aver\_size, 1, MPI\_INT, source, mtype, MPI\_COMM\_WORLD, &status);

MPI\_Recv (&y[offset], aver\_size, MPI\_DOUBLE, source, mtype, MPI\_COMM\_WORLD, &status);

printf ("Received results from task %d\n", source);

}

// Print input and output

printf ("|\*\*\*\*\*\*\*\*\*\*INPUT\*\*\*\*\*\*\*\*\*\*||\*\*\*\*\*\*\*\*\*OUTPUT\*\*\*\*\*\*\*\*\*\*|\n");

for (i = 0; i < ARRAY\_SIZE; i++)

{

printf ("| x [%6d] = %10.2f || y [%6d] = %10.2f |\n", i, x[i], i, y[i]);

}

printf ("Done.\n");

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* worker task \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if (taskid > MASTER)

{

mtype = FROM\_MASTER;

MPI\_Recv (&offset, 1, MPI\_INT, MASTER, mtype, MPI\_COMM\_WORLD, &status);

MPI\_Recv (&aver\_size, 1, MPI\_INT, MASTER, mtype, MPI\_COMM\_WORLD, &status);

MPI\_Recv (&x, aver\_size + 2, MPI\_DOUBLE, MASTER, mtype, MPI\_COMM\_WORLD, &status);

// printf("%d %d\n", taskid, offset);

for (i = 1; i <= aver\_size; i++)

{

y[i + offset - 1] = (x[i - 1] + x[i] + x[i + 1]) / 3;

}

mtype = FROM\_WORKER;

MPI\_Send (&offset, 1, MPI\_INT, MASTER, mtype, MPI\_COMM\_WORLD);

MPI\_Send (&aver\_size, 1, MPI\_INT, MASTER, mtype, MPI\_COMM\_WORLD);

MPI\_Send (&y[offset], aver\_size, MPI\_DOUBLE, MASTER, mtype, MPI\_COMM\_WORLD);

}

MPI\_Finalize ();

}

### **Execution Screenshots :**

