MPI IMPLEMENTATION ON FINITE IMPULSE RESPONSE

High-Performance Computing Project Report

Problem Statement: Parallel simulation of moving average finite impulse response filter

Faculty guide: Dr. Noor Mahammad

By,

PALETI KRISHNASAI

CED18I039

Hardware Configuration:

PU NAME: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz
Number of Sockets: 1
Cores per Socket: 4
Threads per core: 2
L1d cache: 128 KiB
L1i cache: 128 KiB
L2 cache: 1 MiB
L3 cache: 8 MiB

```
paleti@paleti-Lenovo-ideapad-330-15ICH:~$ likwid-topology
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz
CPU type:       Intel Coffeelake processor
CPU stepping:   10
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:                1
Cores per socket:       4
Threads per core:       2
--------------------------------------------------------------------------------
HWThread        Thread          Core            Socket          Available
0               0               0               0               *
1               0               1               0               *
2               0               2               0               *
3               0               3               0               *
4               1               0               0               *
5               1               1               0               *
6               1               2               0               *
7               1               3               0               *
--------------------------------------------------------------------------------
Socket 0:               ( 0 4 1 5 2 6 3 7 )
--------------------------------------------------------------------------------
********************************************************************************
Cache Topology
********************************************************************************
Level:                  1
Size:                   32 kB
Cache groups:           ( 0 4 ) ( 1 5 ) ( 2 6 ) ( 3 7 )
--------------------------------------------------------------------------------
Level:                  2
Size:                   256 kB
Cache groups:           ( 0 4 ) ( 1 5 ) ( 2 6 ) ( 3 7 )
--------------------------------------------------------------------------------
Level:                  3
Size:                   8 MB
Cache groups:           ( 0 4 1 5 2 6 3 7 )
--------------------------------------------------------------------------------
********************************************************************************
NUMA Topology
********************************************************************************
NUMA domains:           1
--------------------------------------------------------------------------------
Domain:                 0
Processors:             ( 0 1 2 3 4 5 6 7 )
Distances:              10
Free memory:            3546.2 MB
Total memory:           7831.84 MB
--------------------------------------------------------------------------------
```

```
*************************************************
Graphical Topology
*************************************************
Socket 0:
+-----------------------------------------------+
| +--------+ +--------+ +--------+ +--------+ |
| |  0 4   | |  1 5   | |  2 6   | |  3 7   | |
| +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ |
| |  32 kB | |  32 kB | |  32 kB | |  32 kB | |
| +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ |
| | 256 kB | | 256 kB | | 256 kB | | 256 kB | |
| +--------+ +--------+ +--------+ +--------+ |
| +-------------------------------------------+ |
| |                   8 MB                    | |
| +-------------------------------------------+ |
+-----------------------------------------------+
```

No of nodes : 12 (4 for each as written in the machine file).

INTRODUCTION

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. This is in contrast to infinite impulse response (IIR) filters, which may have internal feedback and may continue to respond indefinitely (usually decaying).

The impulse response (that is, the output in response to a Kronecker delta input) of an N^{th}-order discrete-time FIR filter lasts exactly *N + 1* samples (from first nonzero element through last nonzero element) before it then settles to zero.

For a causal discrete-time FIR filter of order *N*, each value of the output sequence is a weighted sum of the most recent input values:

$$\begin{aligned} y[n] &= b_0 x[n] + b_1 x[n-1] + \cdots + b_N x[n-N] \\ &= \sum_{i=0}^{N} b_i \cdot x[n-i], \end{aligned}$$

- $x[n]$ is the input signal,
- $y[n]$ is the output signal,
- N is the filter order; an N^{th}-order filter has $N+1$ terms on the right-hand side
- b_i is the value of the impulse response at the *i*'th instant for $0 \leq i \leq N$ of an N^{th}-order FIR filter. If the filter is a direct form FIR filter then b_i is also a coefficient of the filter.

MOVING AVERAGE FIR FILTER ANALYSIS

A moving average filter is a very simple FIR filter. It is sometimes called a boxcar filter, especially when followed by decimation. The filter coefficients, $b_0, \ldots, b_N$, are found via the following equation:

$$b_i = \frac{1}{N+1}$$

To provide a more specific example, we select the filter order:

$$N = 2$$

The impulse response of the resulting filter is:

$$h[n] = \frac{1}{3}\delta[n] + \frac{1}{3}\delta[n-1] + \frac{1}{3}\delta[n-2]$$

Parallel Code [MPI] : [Commented to explain parallelization]

```c
/*
Author : Paleti Krishnasai CED18I039
Simulation of N-order moving average FIR filter
    N : order of the filter
    n : instance
    filter equation : output_signal[n] = (input_signal[n-1] +
input_signal[n] + input_signal[n1+1]) / (N+1)

    The FIR function under consideration is
        y[n] = (x[n+1] + x[n] + x[n-1]) / 6
*/

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

#define ARRAY_SIZE  100000
#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];
    double start,end;
    srand (time (NULL));
```

```
    MPI_Status status;

    MPI_Init (&argc, &argv);
    start = MPI_Wtime();
    MPI_Comm_rank (MPI_COMM_WORLD, &taskid);
    MPI_Comm_size (MPI_COMM_WORLD, &numtasks);

    if (numtasks < 2) // catch if less than 2 master-slave
    {
        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)\n");
        printf ("\n started...\n");
        for (i = 0; i < ARRAY_SIZE; i++)
            x[i] = rand () % 256;
        printf ("\n 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
        // also extra work handled by master for Load Balancing
        y[0] = (x[0] + x[1]) * 0.142857143;
        y[ARRAY_SIZE - 1] = (x[ARRAY_SIZE - 2] + x[ARRAY_SIZE -1]) *
0.142857143;
```

```
        for (i = 1; i <= extra; i++)
        {
            y[i] = (x[i-1] + x[i] + x[i+1]) * 0.142857143;
        }
        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;
        }

        // receive 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]);
        }
```

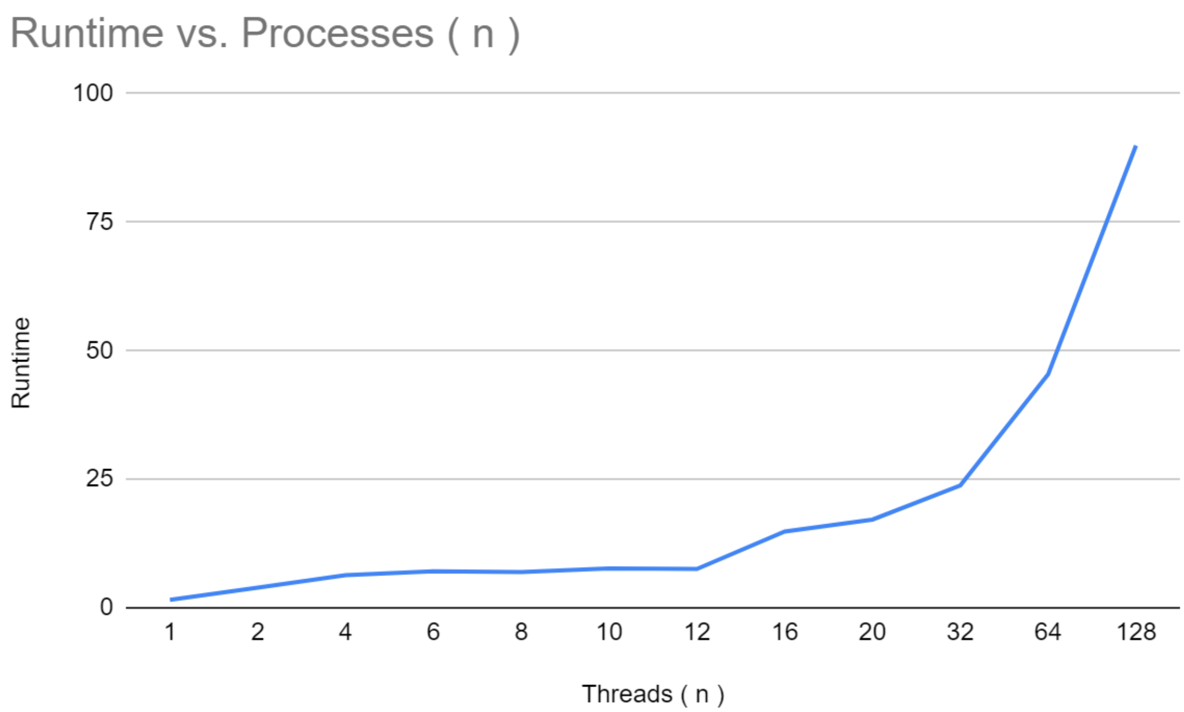
```
        printf ("Done.\n");
        end = MPI_Wtime();
        printf("Time_Stamp = %f\n", end - start);
    }

    /******************************************* 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]) *
0.142857143;
        }

        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 ();
}
```

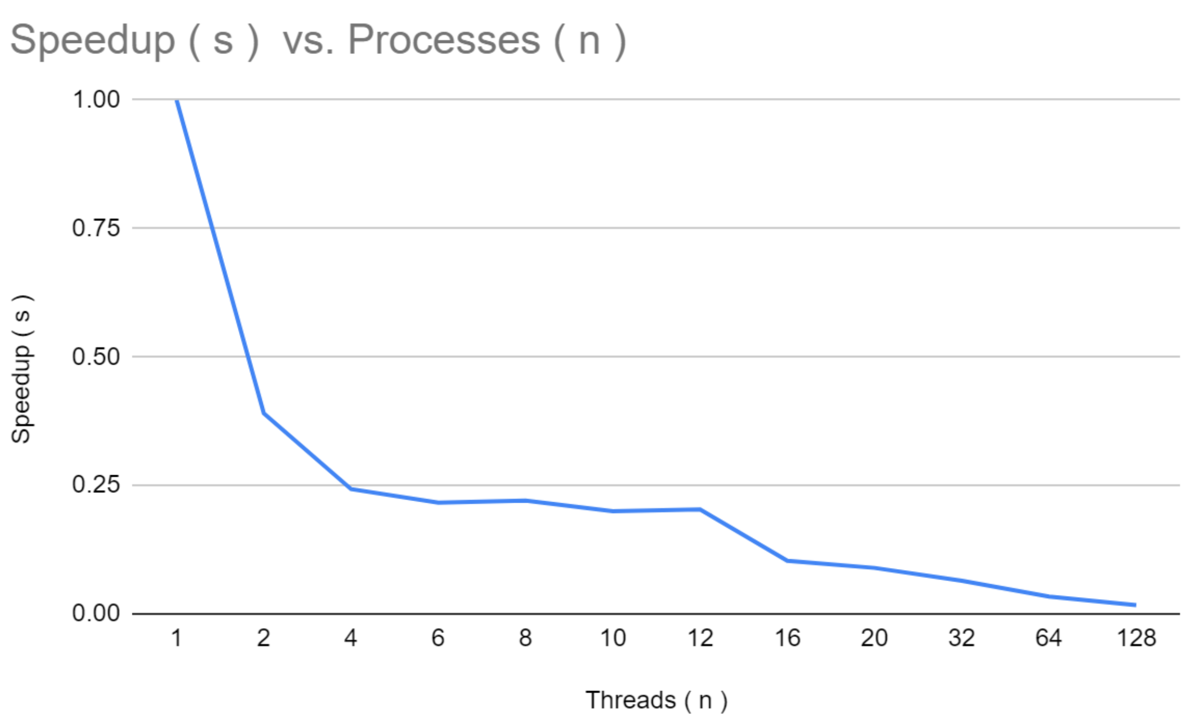
Observations:

Processes (n)	Runtime	Speedup (s)	Parallelization Fraction	1 - 1/s	1 - 1/n
1	1.534658	1			
2	3.93219	0.3902807341	-3.124516342	-1.562258171	0.5
4	6.320199	0.2428179872	-4.157748067	-3.11831105	0.75
6	7.096568	0.2162535468	-4.349041936	-3.624201614	0.8333333333
8	6.954492	0.2206714739	-4.036140951	-3.531623332	0.875
10	7.673172	0.2000030757	-4.444359011	-3.99992311	0.9
12	7.541329	0.2034996749	-4.269832106	-3.914012764	0.9166666667
16	14.84191	0.1034003036	-9.249228254	-8.671151488	0.9375
20	17.158033	0.08944253692	-10.71617122	-10.18036266	0.95
32	23.798167	0.06448639511	-14.97511935	-14.50714687	0.96875
64	45.428122	0.03378211408	-29.05545447	-28.601463	0.984375
128	89.931172	0.01706480596	-58.05368332	-57.60013892	0.9921875

Speed up can be found using the following formula,
$S(n)=T(1)/T(n)$
where, $S(n)$ = Speedup for thread count 'n'
$T(1)$ = Execution Time for Thread count '1' (serial code)
$T(n)$ = Execution Time for Thread count 'n' (serial code)

Parallelization Fraction can be found using the following formula,
$S(n)=1/((1 - p) + p/n)$
where, $S(n)$ = Speedup for thread count 'n'
n = Number of threads
p = Parallelization fraction


Runtime vs. Processes (n)
100
75
50
25
0
Runtime
1
2
4
6
8
10
12
16
20
32
64
128
Threads (n)


Speedup (s) vs. Processes (n)
1.00
0.75
0.50
0.25
0.00
Speedup (s)
1
2
4
6
8
10
12
16
20
32
64
128
Threads (n)

Inference:

(Note: Execution time, graph, and inference will be based on hardware configuration)

- Since MPI is a distributed memory architecture, the communication overhead between nodes causes the parallel code to run slower compared to serial code (running in 1 node or only in master).

Output screenshot:

```
ubuntuhpc@c01:~/mirror$ mpirun -n 128 -f machinefile ./fir
FIR Filter (3 point Average) has started with 128 tasks.

                        Initializing Input Array (x)

 started...

 finished...

|------------INPUT----------||-------------OUTPUT--------------|
| x [      0] =     198.00 || y [      0] =      34.00 |
| x [      1] =      40.00 || y [      1] =      38.71 |
| x [      2] =      33.00 || y [      2] =      37.00 |
| x [      3] =     186.00 || y [      3] =      56.86 |
| x [      4] =     179.00 || y [      4] =      66.29 |
| x [      5] =      99.00 || y [      5] =      45.86 |
| x [      6] =      43.00 || y [      6] =      42.14 |
| x [      7] =     153.00 || y [      7] =      52.29 |
| x [      8] =     170.00 || y [      8] =      64.57 |
| x [      9] =     129.00 || y [      9] =      74.57 |
| x [     10] =     223.00 || y [     10] =      53.14 |
| x [     11] =      20.00 || y [     11] =      39.29 |
| x [     12] =      32.00 || y [     12] =       7.43 |
| x [     13] =       0.00 || y [     13] =      31.29 |
| x [     14] =     187.00 || y [     14] =      30.86 |
| x [     15] =      29.00 || y [     15] =      32.86 |
| x [     16] =      14.00 || y [     16] =      21.00 |
| x [  99994] =     192.00 || y [  99994] =      57.29 |
| x [  99995] =     147.00 || y [  99995] =      60.14 |
| x [  99996] =      82.00 || y [  99996] =      51.71 |
| x [  99997] =     133.00 || y [  99997] =      35.29 |
| x [  99998] =      32.00 || y [  99998] =      30.86 |
| x [  99999] =      51.00 || y [  99999] =      11.86 |
Done.
```

[OpenGL plotting is not implemented in MPI and CUDA projects as OpenGL installation proved to be difficult in virtual machines and colab]
