OpenMP 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

```
palett@palett-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                    | |
| +-------------------------------------------+ |
+-----------------------------------------------+
```

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 [OpenMP] :

```
/*
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)
*/

// g++ Norder_gl.cpp -O0 -fopenmp -lGL -lGLU -lglut -lGLEW -o Norder_gl



#include <bits/stdc++.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <GL/glew.h>
#include <GL/freeglut.h>
#include <GL/glut.h>
#include <math.h>
#include <time.h>
#include <omp.h>

using namespace std;

#define N 6 // filter order
#define size 200000 // size of 1-D signal

long double input_signal[size], output_signal[size]; // store in heap
int h;
void changeViewPort(int w, int h)
{
  if(w>=h)
     glViewport(w/2-h/2, 0, h, h);
  else
     glViewport(0, h/2-w/2, w, w);
```

```
}

void myinit(void)
{
  glClearColor(0.8,0.8,0.8,0.0);
  glMatrixMode(GL_PROJECTION);
  gluOrtho2D (0.0,200000.0,0.0,10000.0);
  glMatrixMode(GL_MODELVIEW);
}

void filter()
{
  myinit();
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  float startTime, endTime,execTime;
  long double random_input;

  // initialize input signal using random values
  for (int i = 0; i < size; i++)
  {
     random_input = static_cast <long double> (rand());
     input_signal[i] =random_input;
  }
  startTime = omp_get_wtime();
  // filter equation

     output_signal[0] = ( (input_signal[0] + input_signal[1]) ) * 0.142857143; // edge case 1
     #pragma omp parallel private (h) shared (input_signal,output_signal)
     {
        #pragma omp for
        for ( h = 1; h < size-1; h++)
        {
           output_signal[h] = ( (input_signal[h-1] + input_signal[h] + input_signal[h+1]) ) *
0.142857143; // divide by N+1

                    glColor3f(0,0,0);
                    glPointSize(3);
                    glBegin(GL_POINTS);
```

```
                    glVertex2f(h,(output_signal[h])/100000);
                    glEnd();
                    glFlush();
                    glutSwapBuffers();

                }

            }
            output_signal[size - 1] = ( (input_signal[size - 2] + input_signal[size - 1]) )* 0.142857143; // edge
case 2

  endTime = omp_get_wtime();
  execTime = endTime - startTime;
  cout << execTime << endl;
  //glFlush();
  //glutSwapBuffers();
  glutLeaveMainLoop();
}


int main (int argc,char** argv)
{
  srand (time(0));

  glutInit(&argc,argv);
  glutInitDisplayMode(GLUT_DOUBLE| GLUT_RGB );
  glutInitWindowSize(1000, 1000);
  glutInitWindowPosition(200, 200);


  glutCreateWindow("CED18I039");
  glutDisplayFunc(filter);


  glutReshapeFunc(changeViewPort);
  GLenum err = glewInit();
  if (GLEW_OK != err) {
     fprintf(stderr, "GLEW error");
```

```
    return 1;
  }
  glutMainLoop();


  return 0;
}
```

Compilation and Execution:

To enable OpenMP environment, use *-fopenmp* flag while compiling using g++.
-O0 flag is used to disable compiler optimizations.

```
g++ Norder_gl.cpp -O0 -fopenmp -lGL -lGLU -lglut -lGLEW -o Norder_gl
```

Then,
export OMP_NUM_THREADS= no of threads for parallel execution.
./Norder_gl

Critical part and how the code is parallelized

```
#pragma omp parallel private (h) shared (input_signal,output_signal)
    {
        #pragma omp for
        for ( h = 1; h < size-1; h++)
        {
            output_signal[h] = ( (input_signal[h-1] + input_signal[h] + input_signal[h+1]) ) *
0.142857143; // divide by N+1

                    glColor3f(0,0,0);
                    glPointSize(3);
                    glBegin(GL_POINTS);
                    glVertex2f(h,(output_signal[h])/100000);
                    glEnd();
                    glFlush();
                    glutSwapBuffers();

        }

    }
```

The above part of the code is the parallizable part from the serial code.
Here we are simulating the moving average finite impulse response filter where in as there is a constant flow of input signal the corresponding output signal is generated, albeit with a delay as each output is dependent on consecutive inputs.
This piece of code simulates the functionality by plotting each point on a glut canvas as it is generated.
The number of output signal components that are paralllized are size - 2 , as the first and the last signal component is not parallizable as it doesnt fit into the general equation.
The equation is also optimized to reduce the number of operations by removing generalization by making it order specific, ie., 1/N+1 is calculated for order 6 and value is written in the equation.
The pragma enables each value to be computed using a thread using the fork - join paradigm.
Further parallization was not found. Another approach tried is to calculate the output as soon as the input signal is being generated, but this way there would be several

misses in the output signal. Hence the approach mentioned above has been chosen. This code can be used on proprietary datasets (not available open source) with varying filter orders.

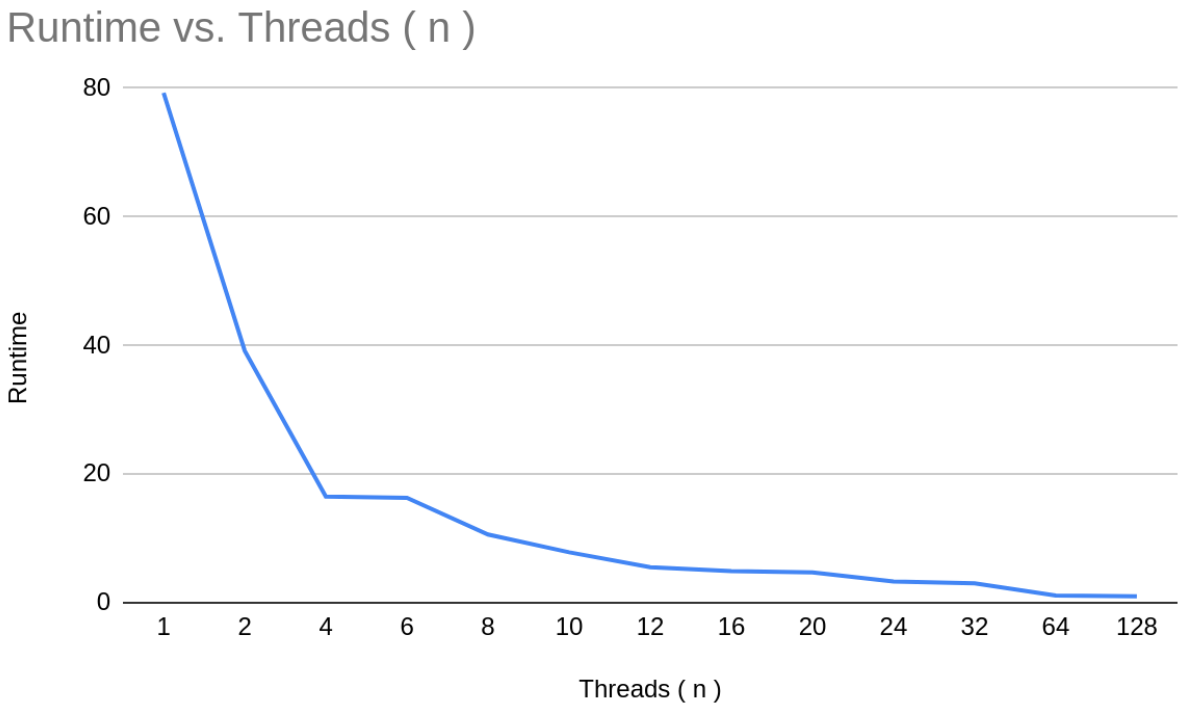
The output is plotted by scaling it down by a factor of 10^5 as the values are too to plot on a glut canvas.

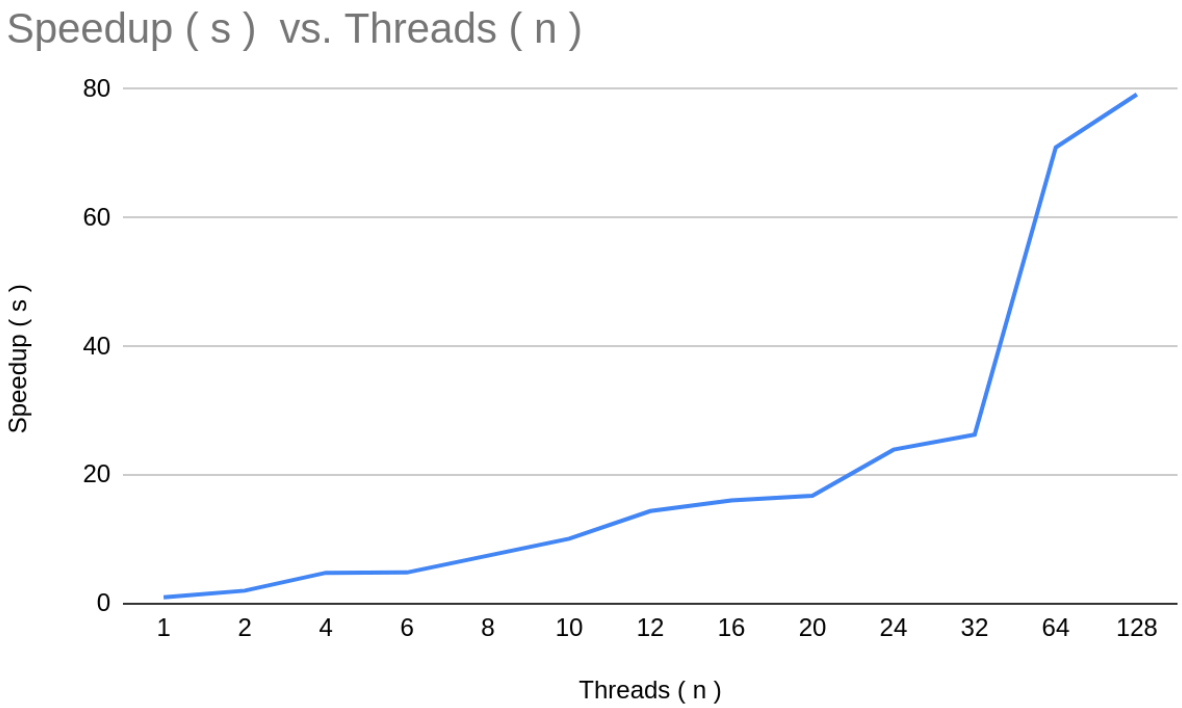
Observations:

Threads (n)	Runtime	Speedup (s)	Parallelization Fraction
1	79.2881	1	
2	39.1943	2.022949766	1.011344704
4	16.4932	4.807320593	1.055978556
6	16.3184	4.858815815	0.9530262423
8	10.6094	7.473382095	0.9899334561
10	7.8457	10.10593064	1.00116467
12	5.50684	14.39811217	1.015141582
16	4.93262	16.07423641	1.00030789
20	4.72656	16.77501185	0.9898816037
24	3.30957	23.95722103	0.9999223635
32	3.01758	26.27539286	0.9929719511
64	1.11719	70.97100762	1.0015591
128	1.00098	79.21047374	0.9951500164

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. Threads (n)
80
60
40
20
0
Runtime
1
2
4
6
8
10
12
16
20
24
32
64
128
Threads (n)


Speedup (s) vs. Threads (n)
80
60
40
20
0
Speedup (s)
1
2
4
6
8
10
12
16
20
24
32
64
128
Threads (n)

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

- The runtime decreases as the number of threads increases.
- The speedup increases with the number of threads.
- This is because as the number of threads increase, the computational intensity (the size of the 1-D signal) of the problem is shared over several threads which execute parallely.