ECE 4822: Engineering Computation IV

Homework No. 6: Parallel DSP Filters Using OpenMP

By Shahzad Khan

The goal of this assignment is to implement both an FIR and IIR filter and parallelize them using OpenMP.

The Finite Impulse Response (FIR) filter is a digital filter whose output is a weighted sum of finite number of past inputs. Which means that it takes an input signal, and outputs will eventually settle to 0 given infinite time, hence the finite in the name. The equation for an FIR filter is the following

$$y[n] = \sum_{i=0}^{N} b_i * x[n-i]$$

$x[n]$ = input signal

$y[n]$= output signal

N= filter order

b_i= coefficient of the filter

The advantages of FIR filters are that they are inherently stable which means no matter what coefficients are given the output won't explode to infinity. The coefficients are the main part of FIR filters, because depending on what values are given the filter can change. There are a couple ways to calculate the coefficients for the filter. For example, to calculate the ideal impulse response for a low pass filter we can use the $sin(x)$ function

$$h_d[n] = \frac{\sin(\omega_c n)}{\pi n}$$

And to calculate the cut-off frequency we can use $\omega_c = 2\pi * \left(\frac{f_c}{f_s}\right)$ where f_s is our sampling frequency and f_c being our desired cutoff frequency in Hz

But a problem accurses because $sinc$ function is infinite we must convert it to a discrete time so we can use it in our program.

The way we can do this is by centering our waves.

$$n = n - \frac{N-1}{2}$$

So, our low pass filter is because

$$h_d[n] = \sin\frac{\left(w_c\left(n - \frac{N-1}{2}\right)\right)}{\pi\left(n - \frac{N-1}{2}\right)}$$

But this causes a ripple effect called Gibbs phenomenon, but for our cases it doesn't really matter so we can use a rectangular windowing $w[n] = 1$

So now using this we calculate our coefficients, there is also another way by $\frac{1}{tabs}$ this method allows you to quickly calculate the coefficients for a low pass filter but doesn't allow you to choose a cut-off frequency.

So doing that we can get a coefficient of $[0.2, 0.2, 0.2, 0.2, 0.2]$

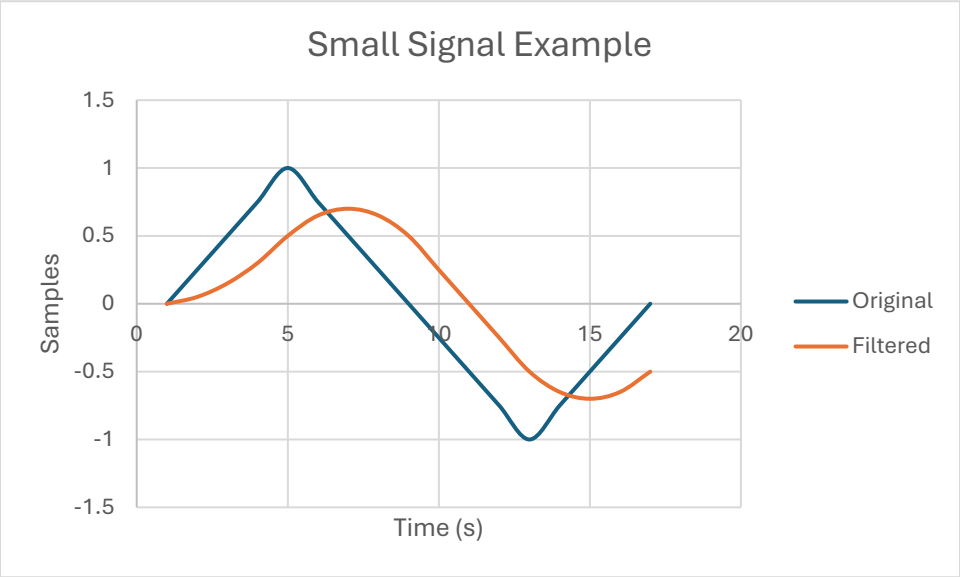

Figure 1

The coefficient that we are using is a moving average where we can see I input a triangle wave, and the filter cuts it off smoothing the whole signal out.

Now moving to how I parallelized the code. OpenMp makes this process simple on my end. The first problem we must deal with is with a large file size, so the solution to that is to use a circular buffer to feed the signal in blocks one at a time. To do this we first calculate the number of blocks

```
const int BLOCK_SIZE = 1024; // Number of samples per block

int numBlocks = (samples.size() + BLOCK_SIZE - 1) / BLOCK_SIZE;
```

Then we can use OpenMp to parallelize each block, and since the FIR filter doesn't use any feedback loop, we can do this

```
// Parallel block processing
#pragma omp parallel for
 for (int b = 0; b < numBlocks; ++b) {
  int start = b * BLOCK_SIZE;
  int end = std::min(start + BLOCK_SIZE, (int)samples.size());

  // History buffer for this block
  std::vector<float> history(n, 0.0f);
  int circIndex = 0;

  // Initialize history from previous block if not first block
  if (b > 0) {
   int prevEnd = start;
   int copyStart = std::max(0, prevEnd - (n-1));
   int copyCount = prevEnd - copyStart;
   for (int i = 0; i < copyCount; ++i)
    history[i] = samples[copyStart + i];
   circIndex = copyCount % n;
  }

  // Process block
  for (int i = start; i < end; ++i) {
   output[i] = fir_filter_circ(samples[i], coef.data(), n, history.data(), &circIndex);
  }
 }
```

We use a history variable to keep track of where we are in the block and then we get our fully parallelized FIR filter

We can also work with a file that is around 1GB and timing this we get a time of **5.123 (s)**

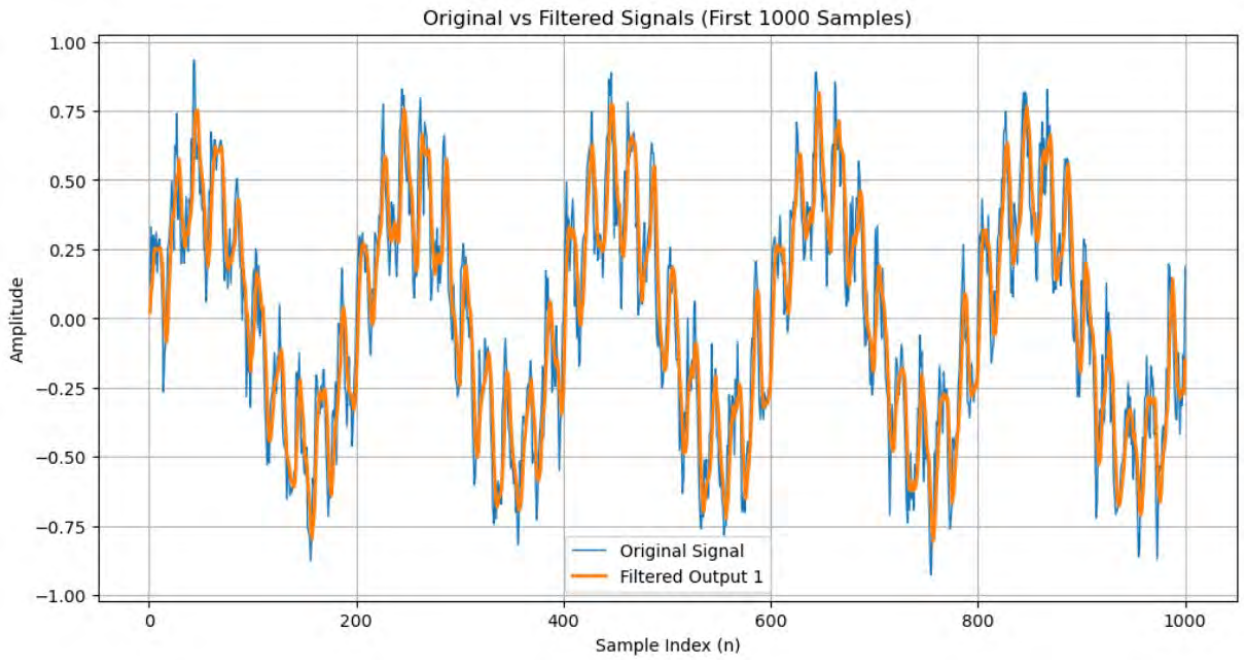

Figure 2

This is using our 0.2 moving average coefficient, but I also calculated some more for a cutoff frequency of $1000Hz$

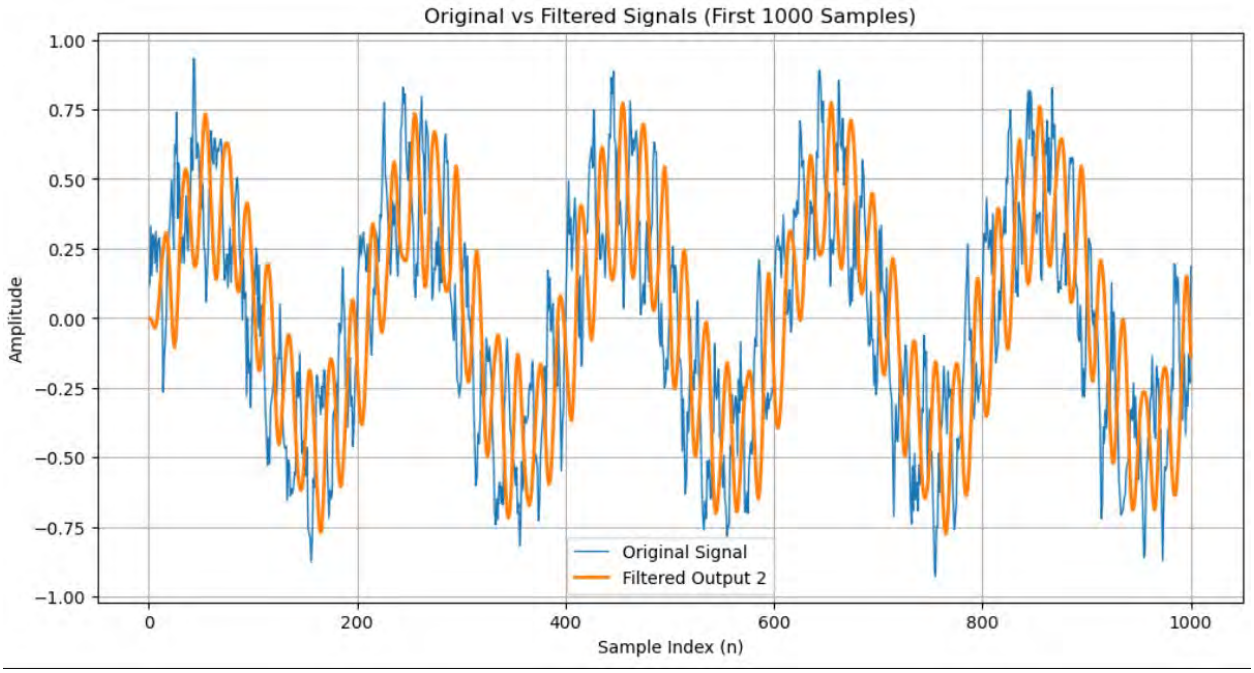

Figure 3

We can see here that that signal is filtered even more, and after playing both the filtered audio there is a noticeable difference between the sound. So, this shows my filter does work as intended and works for large files.

The Infinite Impulse Response (IIR) another digital filter that as the name suggests is similar to the FIR filter but with a couple key differences well the FIRs response is finite and given infinite time will reach 0, the IIRs response is infinite and because of the recursive nature of the IIR, any impulse that goes through the signal never really fades and effects the entire signal. The equation for an IIR filter is stated

$$y[n] = \sum_{i=0}^{P} b_i x[n-i] + \sum_{i=1}^{Q} a_i y[n-i]$$

P = Feed-Forward Filter

b_i = Feed-Forward Coefficient

Q = Feed-Back Filter

a_i= Feed-Back Coefficient

$x[n]$=input signal

$y[n]$ = output signal

As seen from the equation the IIR filter can be split up into parts the first part which can be boiled down to an FIR filter and then the feedback loops giving the filter its recursive properties. But this comes with a tradeoff, well the FIR filter is inherently stable no matter what, the IIR filter has a chance of becoming unstable when the poles that are set are outside the unit circle

The IIR is also different because you cannot calculate the coefficients from the frequency response, we must design poles first and then compute them. So usually when we are computing these filters in the s-plane we must make sure that the real part is not positive, this is because the equation is given as this

$$x(t) = e^{\sigma} * e^{j\omega t}$$

When σ is positive, instead of the real part acting as a decay, the real part acts like growth, this causes the exploding effect.
But because we are acting in the discrete domain, we must convert this as well. So, the equation then becomes

$$x[n] = r^n * e^{jn\theta}$$

Now we see that r^n is our growth or decay coefficient and if n is greater than 1 the function once again explodes because it acts as a growth instead of a decay which makes our filter stable.

We can use this equation to find the poles of a first order low pass filter

$$z = 1 + \frac{1 + \frac{sT}{2}}{1 - \frac{sT}{2}}$$

Transfer Function^

$$H(z) = \frac{b_0}{1 - a_1 z^{-1}}$$

When we solve for the denominator = 0 so we can find the poles

$$z = a_1$$

Then we can find the poles using the cutoff and sampling frequency

$$a_1 = \frac{1 - \alpha}{1 + \alpha}, \quad \alpha = \frac{\omega_c}{f_s 2}$$

If we want a cutoff of $1000Hz$ with a $f_c = 10{,}000Hz$ we can calculate the cutoff

$$2 * \pi * \frac{1000}{10000} = 0.6283 \frac{rad}{sample}$$

Now plugging those into our transfer function we get

$$z = \frac{1 + \frac{sT}{2}}{1 - \frac{sT}{2}} = \frac{1 - 0.31415}{1 + 0.31415} = \frac{0.68585}{1.31415} = 0.522$$

So, our pole would be $a_1 = 1 - 0.522$

$$b_0 + b_1 = 1 - a_1 = 1 - 0.522 = 0.478$$

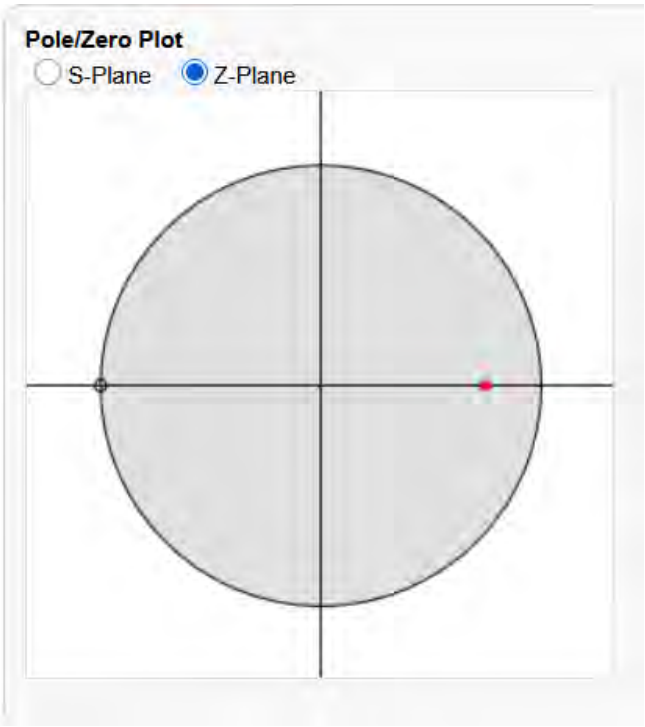

Figure 4

So, our coefficient for the lowpass filter with a cutoff of $1000Hz$ would be

$$D = [1, -0.522]$$

$$N = [0.239, 0.239]$$

The parallelization of the IIR filter is a little more complex. Since there is a feedback loop integrated. We must make sure that that we can use history from our previous iteration to use when we do our current calculation.

```
// Precompute histories for each block
std::vector<std::vector<float>> xHistories(numBlocks, std::vector<float>(bCoef.size(), 0.0f));
std::vector<std::vector<float>> yHistories(numBlocks, std::vector<float>(aCoef.size()-1, 0.0f));



// Copy last samples from previous block for history initialization
for(int b = 1; b < numBlocks; ++b){
 int prevEnd = b*BLOCK_SIZE;
 int copyCountX = std::min((int)bCoef.size()-1, prevEnd);
 int copyCountY = std::min((int)aCoef.size()-1, prevEnd);

 for(int i = 0; i < copyCountX; ++i)
  xHistories[b][i] = samples[prevEnd - copyCountX + i];

 for(int i = 0; i < copyCountY; ++i)
  yHistories[b][i] = output[prevEnd - copyCountY + i]; // initially zero for first iteration
}
```

Then we can parallelize our code with our precomputed history blocks

```
// Process each block in parallel
#pragma omp parallel for
for(int b = 0; b < numBlocks; ++b){
int start = b * BLOCK_SIZE;
int end = std::min(start + BLOCK_SIZE, (int)samples.size());

auto& xHist = xHistories[b];
auto& yHist = yHistories[b];

for(int i = start; i < end; ++i){
output[i] = iir_filter(samples[i], bCoef, aCoef, xHist, yHist);
}
}
```

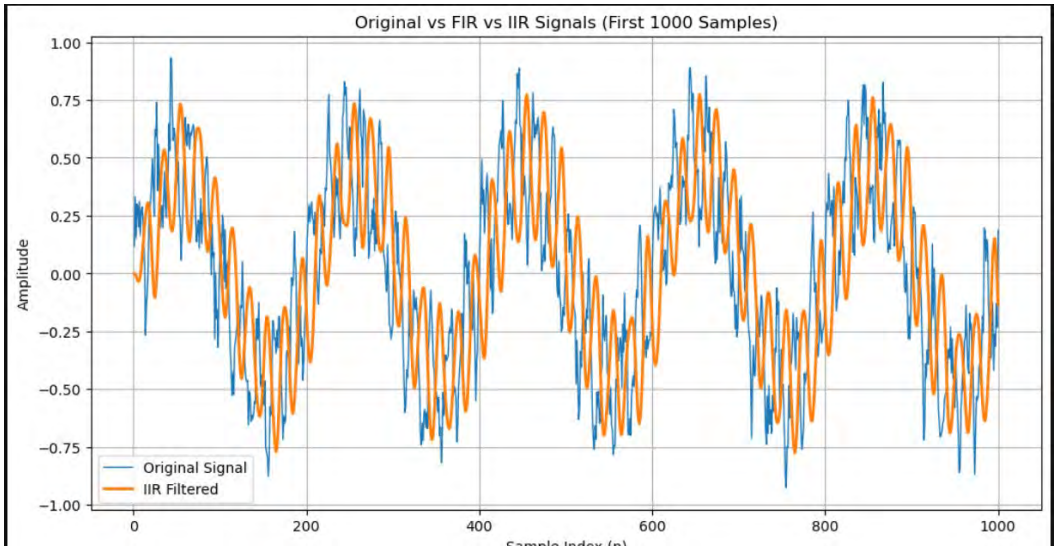

Figure 5

Above we see the output of the IIR filter compared to noise signal, and we can see that it does filter out the higher frequency and spikes well.

The program runs with a 1GB file in **4.780s.**

In conclusion, through this homework I was able to use OpenMP to parallelize both an FIR and IIR filter. Through this I was able to gain a better understanding of both filters as well as the challenges with parallelizing functions that utilize a feedback loop. As well as how digital filters work and the challenges with their implementation such as taking into account how the data will be fed to the function, and my biggest take away is how filters tie back to the rest of my classes and it allowed for me to get a better understanding of basic analog signal filtering going on in my classes currently and how we can convert then to discrete version, an example is the stability of a IIR filter, and how every analog filter we are using is an IIR filter and the difference between the s and z planes