Fall 2017 ME759 Final Project Report
University of Wisconsin-Madison

Parallelizing OFDM based communication

Rahul Nayar
Ayush Gupta
Pavan Holla

December,2017

Abstract

Design of OFDM based communication systems often involve running simulations on millions of packets, usually taking days. Speeding up these simulations will help in faster calculations for packet error rate and also help testing algorithms faster and thereby achiever faster full system simulations where throughput also needs be assessed for higher OSI layers. We design and accelerate an OFDM based transmitter using OpenMP and CUDA. We parallelize individual sub modules of the OFDM transmitter. Our results show that the workload is highly parallelizable across processors.

Intro to OFDM

OFDM (Orthogonal Frequency Division Multiplexing) is a communication scheme used in LTE and Wifi networks. It involves mapping a stream of data bits(frame/packet used interchangeably) to several orthogonal frequency channels. Before transmitting the bits on air, the bits pass through several processing stages to improve the reliability of wireless transmission.

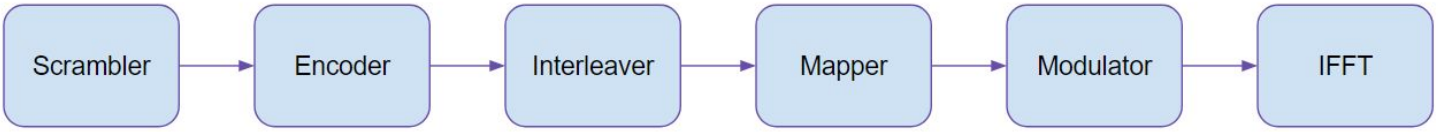


Fig 1 shows the transmission pipeline for Wifi 802.11a

Scrambler

The incoming bit stream/frame from a higher layer is passed through the scrambler. The scrambler XOR's the incoming bit stream with a sequence generated from a 7 bit LFSR(Linear Feedback Shift Register). The 7 bit LFSR is initialized randomly at the beginning of every frame. The 7-bit LFSR generates a repeating sequence of 128 numbers. Scrambling ensures that there is no contiguous stream of 0's transmitted on air because the electronics for wireless reception require that the transmitted bits toggle.

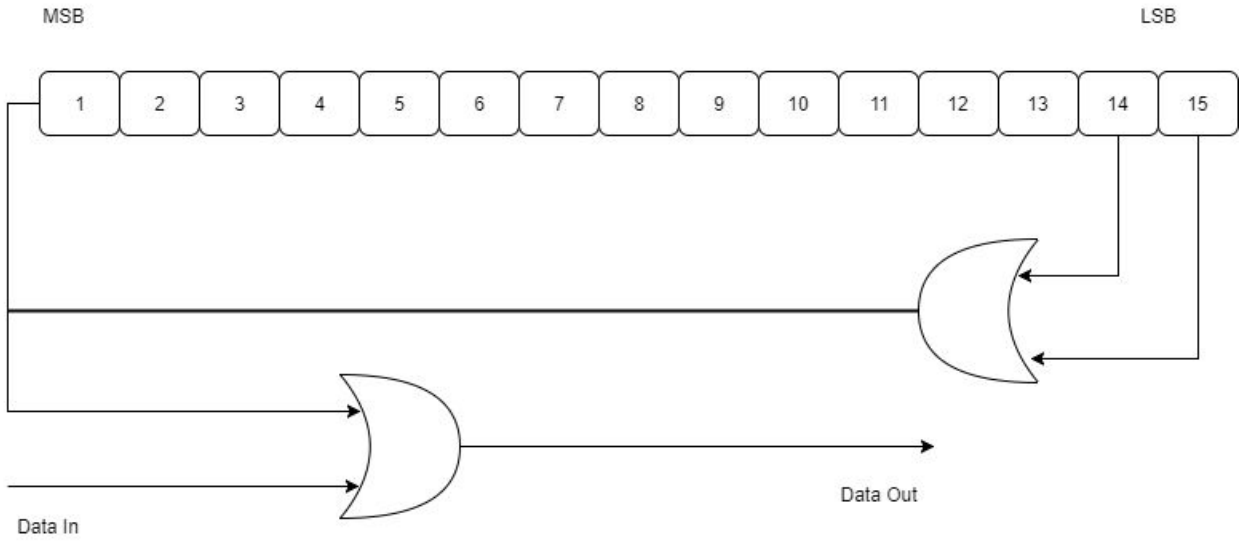


Fig 2 shows logic implementing Scrambler

Encoder

The scrambler feeds into the encoder, which is responsible for repeating bits in the input frame. For every bit in the scrambled frame, the encoder produces 2 bits (for a rate ½ convolutional encoder). The rate ½ encoder is a set of two FIR filters, each of which produces an output bit. In 802.11, the encoder is composed of two 6th order FIR filters with polynomials octal(131,171)

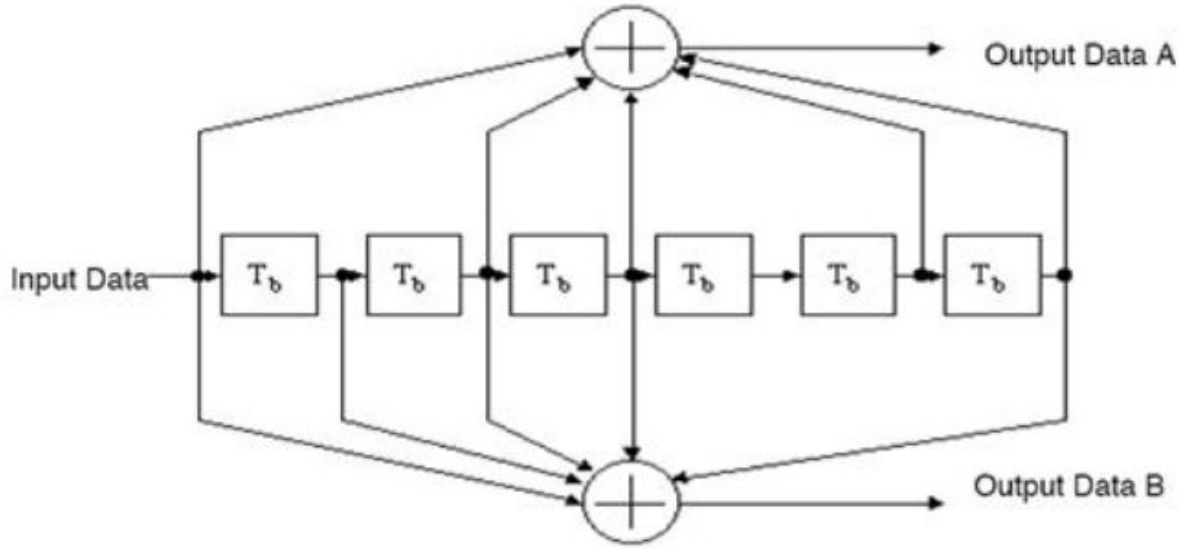


Fig 3 shows Encoding logic

Mapper

Until the encoder, we dealt with frames. However, starting from the mapper, the encoded frame is broken into symbols.The coded data is grouped into a number of bits per symbol (128 bits per symbol in our implementation). This is performed by the mapper.

[Bit 1 Bit 2 Bit 3 ….. Bit 126 Bit 127 Bit 128] [Bit 129 Bit 130 …. Bit 256]
[Symbol 1] [Symbol 2]

Fig 4 Mapping 128 bits to a Symbol

Interleaver

Each symbol is then block interleaved with two permutations. The first permutation maps adjacent coded bits onto nonadjacent subcarriers/frequencies. The incoming symbol is arranged in a 16 x 8 matrix row-wise and read out column-wise.

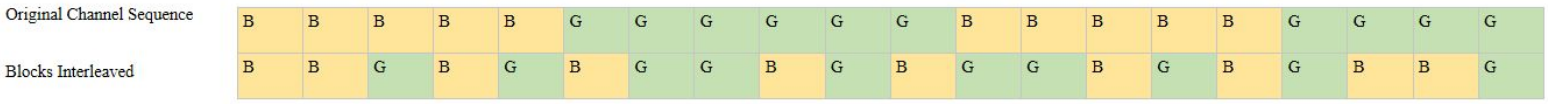


Fig 5 Interleaving

In this manner, adjacent coded(for simplicity, we can replace the term ‘coded’ by duplicated) bits are transmitted on channels far away from each other so that if one channel fails, the other still succeeds.

The second permutation performed by the interleaver ensures that adjacent coded bits are distributed between less and more significant bits of the modulation constellation. In simpler terms it conditionally swaps bits that are adjacent to each other after the first permutation.

Modulator

The modulator receives the interleaved symbol and groups adjacent bits into complex numbers. For example, the incoming stream [1, 0, 1, 1 , 0, 0], gets mapped to [1 + 0i , 1+1i ,0 + 0 i]. The 802.11 modulators can potentially group multiple bits into a single complex number, however, in our implementation, we perform QPSK modulation (Map two bits into 1 complex number).

IFFT

The final step in our implementation is the IFFT, which performs an FFT on modulated symbol. Each symbol (64 complex numbers from the modulator) gets mapped to a time domain waveform that is 64 samples long. These 64 samples are then transmitted on air one after another.

Implementation and Scaling details

Frame Generator

This module generates a randomized 32kB frame. The pseudo code is as follows:

```
for(int ii=0; ii<frame_size; ii++)
   frame[ii]=rand()%2;
```

We had some trouble parallelizing this in OpenMP. Once we realized rand() was not thread safe and switched to using ***rand_r()***, we observed linear scale up with the number of processors.For the CUDA implementation, we used the cuRAND library. Each block receives 128 threads, with each thread generating a bit. 128 is a safe thread count since the number of bits transmitted is guaranteed to be a multiple of 128 in 802.11a.

Scrambler

This module performs an xor operation with a predefined sequence generated from an LFSR.

```
 for(int ii=0; ii<frame_size; ii++)  {
     frame[ii] ^= xor_sequence[ii%128];
}
```

We observe a linear scale up with number of processors in OpenMP. The CUDA implementation has 128 threads in each block, because the scrambling sequence repeats after 128 bits. This makes sure that all memory accesses are coalesced.

Encoder

Our encoder is composed of two FIR filters. If we had a sixth order FIR filter, which implements the equation

Y[n] = X[n] ^X[n-6] , where n=1:1000
Then, we could potentially split the workload into
Y[n] = X[n] ^X[n-6] , where n=1:500, initialize X[-5:0] =0
Y[n] = X[n] ^X[n-6] , where n=501:1000, bypass X[495:500] from the scrambler for initialization
Our encoder also linearly scales with the number of processors. Our CUDA implementation is similar to the averaging filter used in problem 2 of HW9 and we use shared memory to speed up our implementation.

Mapper and Interleaver

We combined the mapper and interleaver operation into a single block. Since the interleaver operates on a per symbol basis, we parallelized the interleaver on a per symbol basis. A symbol

is defined as a set of 128 contiguous encoded bits in our implementation. Our OpenMP implementation of the interleaver defined in 802.11a is as follows:

```
int NCBPS =128; // Number of coded bits per symbol
  #pragma omp parallel for num_threads(N) schedule(static)
  for (int symbol_no =0; symbol_no< num_symbols; symbol_no++)  {
    for(int k=0 ; k < NCBPS; k++)  {
      int i = (NCBPS/16) * (k % 16) + (floor(k/16)) ; //first permutation, interleave frequencies
      int j = s * floor(i/s) + ((int)(i + NBPSC- floor(16 * i/NCBPS)) % s); //second permutation, swap
adjacent bits
      interleaved_data[symbol_no*NCBPS + j] = data[symbol_no*NCBPS + k];
    }
  }
```

We observe that the implementation scales linearly in OpenMP. In CUDA, we considered coalescing memory accesses, but realized that bank conflicts will certainly occur for certain configurations of OFDM transmission. Moreover, the interleaver was one of the least time consuming operations, therefore we did not optimize it further for CUDA.

Modulator

The modulator groups adjacent bits into a single complex number. Once again, just like the interleaver, the modulator works on a symbol level granularity. Thus, it makes sense to draw parallelization boundaries at symbols. Our OpenMP pseudo code for the modulator is as follows.

```
int NCBPS =128; // Number of coded bits per symbol
#pragma omp parallel for num_threads(N) schedule(static)
  for(int symbol_num=0; symbol_num< size/NCBPS; symbol_num++)  {
    for(int ii=0; ii<  NCBPS ; ii+=1)  {
      modulated_data[symbol_num][ii] =  Complex (2*data[2*ii+symbol_num*NCBPS] -1 ,2 *
data[2*ii + 1 + symbol_num*NCBPS] -1 );
    }
  }
```

We observed linear scaling with the modulator as well on OpenMP.

IFFT

We did not write our own implementation of the IFFT algorithm. Instead, we used an implementation of fft available at https://rosettacode.org/wiki/Fast_Fourier_transform . Since

802.11 requires that the FFT be performed on a per symbol basis, parallelizing across symbols was our easiest option. It is possible to parallelize the IFFT itself, but since most OFDM symbol IFFT's are less than 1024 samples long, the overhead associated with dispatching threads for each symbol would be detrimental. We could potentially use a vectorized implementation of the IFFT, but we do not quantify vectorization gains in this project.

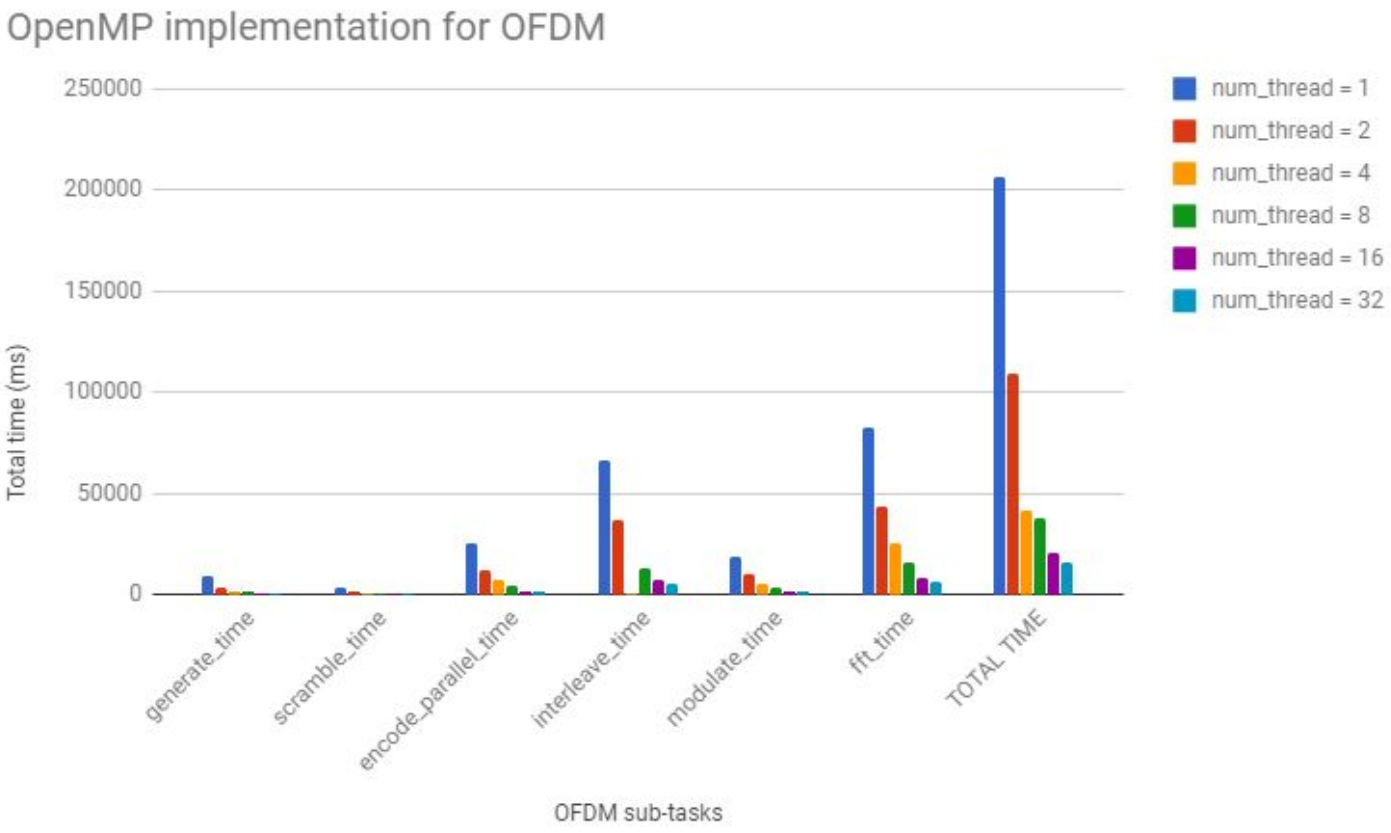
Parallelization of the IFFT across symbols gives us linear gains in OpenMP. For the CUDA implementation, we used the cuFFT library.

Conclusion

Based on timing analysis of OFDM simulation, "interleave" and "fft" processing took about 70% of total processing time. We know from Amdahl's law that we can get significant speedup by parallelizing this 70% of the processing time of the OFDM simulation. Therefore, we first focussed on parallelising "fft" and "interleave" stages. We also subsequently parallelised the other stages of OFDM operation. Using the OpenMP library to parallelise we observed a speedup of 13x with 32 threads. The results of our parallelization operation for different number of threads are shown below. As discussed earlier, these results show linear speedup for each of the stages as the number of threads increases.

	num_thread = 1	num_thread = 2	num_thread = 4	num_thread = 8	num_thread = 16	num_thread = 32
generate_time	9612.032	3337.18016	2036.43904	1212.47744	751.8464	743.56224
scramble_time	3462.76864	1926.4512	1105.3568	694.17984	415.094272	313.778176
encode_parallel_time	25085.3376	12528.7936	7054.6432	4368.62464	2037.92896	1393.41312
interleave_time	66365.44	37118.2592	407.645	12848.0256	7487.488	5134.2848
modulate_time	18691.1232	10072.6784	5852.8256	3502.11584	1839.64672	1621.29408
fft_time	83014.656	43927.7568	25112.0128	15434.1376	7982.9504	6396.7232
TOTAL TIME(ms)	206231.3574	108911.1194	41568.92244	38059.56096	20514.95475	15603.05562

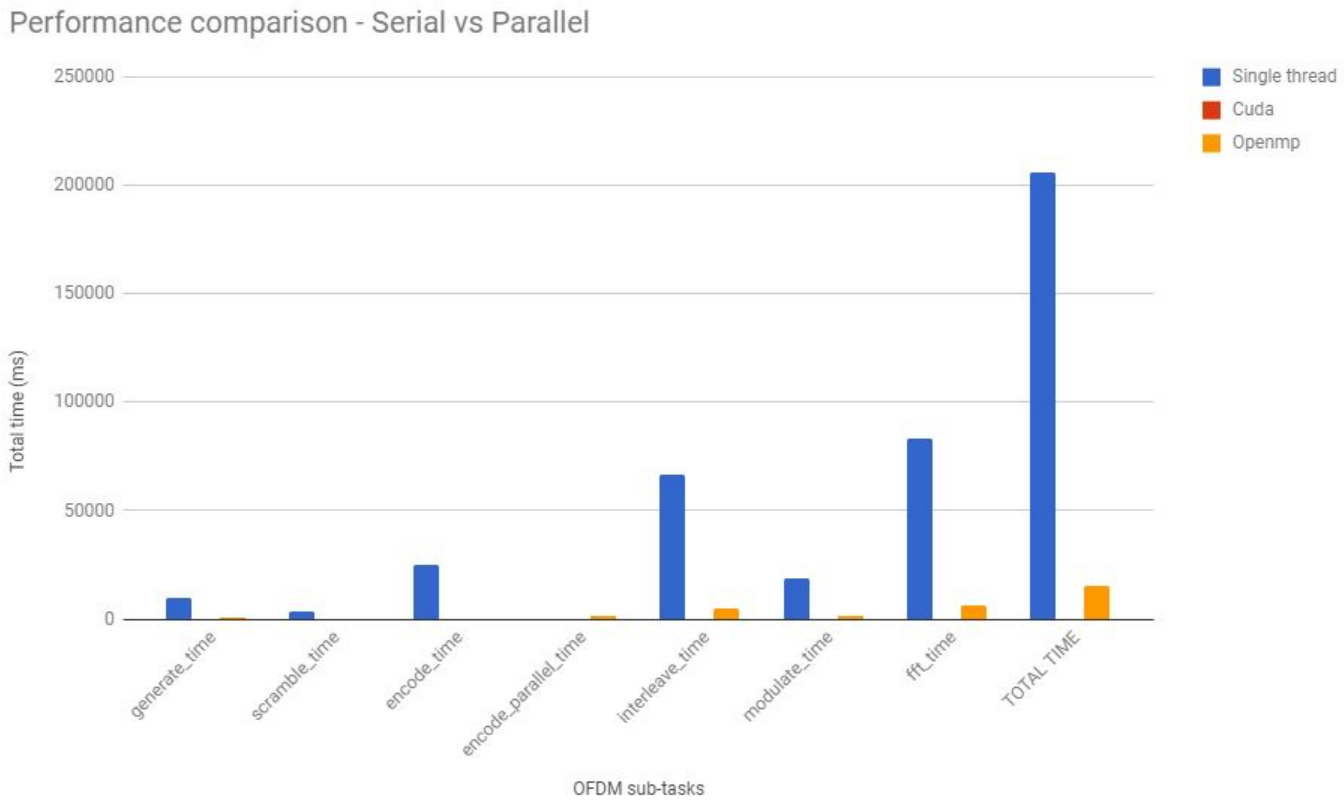
Table 1: Scaling analysis for different Stages of OFDM on OpenMP



Graph: Scaling analysis for different Stages of OFDM on OpenMP

After running simulations on OpenMP and getting significant speedup, we were convinced about our theorised speedup on CUDA. OFDM simulations on CUDA gave us a speed up of 900x compared to single thread and 70x compared to best OpenMP implementation using 32 threads. Results of our timing analysis are shown in the table below.

	Single thread	CUDA	OpenMP
generate_time	9612.032	151.033	743.56224
scramble_time	3462.76864	14.4685	313.778176
encode_time	25085.3376	0	0
encode_parallel_time	0	18.4605	1393.41312
interleave_time	66365.44	14.3562	5134.2848
modulate_time	18691.1232	19.8832	1621.29408
fft_time	83014.656	7.37702	6396.7232
TOTAL TIME(ms)	206231.3574	225.57842	15603.05562



Graph: Scaling analysis for different Stages of OFDM on OpenMP/CUDA. CUDA is 70x faster than OpenMP with 32 threads

References

- https://rosettacode.org/wiki/Fast_Fourier_transform
- http://www.teletopix.org/cdma/how-interleaving-improves-data-transmission-systems/
- https://www.amazon.com/Next-Generation-Wireless-LANs-802-11ac/dp/1107016762
- http://www.rfwireless-world.com/source-code/MATLAB/scrambler-descrambler-matlab-code.html
- http://flylib.com/books/en/2.92.1.83/1/
- http://acarus.uson.mx/docs/CUDA-5.5/CUFFT_Library.pdf