M1 - Project Proposal

Parallelization of the Cooley-Tukey algorithm

Krishna Kinnal (13CO254), Sunil Saini (13CO147)

Problem statement

A fast Fourier transform (FFT) algorithm computes the discrete Fourier transform (DFT) of a sequence, or its inverse. Fourier analysis converts a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa. An FFT rapidly computes such transformations by factorizing the DFT matrix into a product of sparse (mostly zero) factors. As a result, it manages to reduce the complexity of computing the DFT from O $(n\hat{2})$, which arises if one simply applies the definition of DFT, to O (n logn), where n is the data size.

The CooleyTukey algorithm, named after J.W. Cooley and John Tukey, is the most common fast Fourier transform (FFT) algorithm. It re-expresses the discrete Fourier transform (DFT) of an arbitrary composite size N = N1N2 in terms of N1 smaller DFTs of sizes N2, recursively, to reduce the computation time to O(N log N) for highly composite N (smooth numbers).

In this project we aim to parallelize the Cooley-Tukey algorithm using OpenMP constructs, allowing the user to specify the size of the 1D array to be computed and the number of OpenMP threads to be launched in order to perform the parallel computation.

Objectives

The objectives to achieve are:

1. Write code that computes the Cooley-Tukey FFT of a 1D array serially
2. Parallelize it using OpenMP constructs
3. Achieve speed performance by optimizing

Project Timeline

February - Began research on project topic and defined the project proposal. Found out that there is no existing implementation of an OpenMP-parallelized version of the Cooley-Tukey Radix-2 algorithm in specific. Hence, we chose this as our project topic.

March - Began work on the project by working on the serial version of the FFT algorithm

April - Worked on the parallelization of the code by using OpenMP constructs

Work Distribution

Sunil Saini - Working on the serial version of the FFT code.
Krishna Kinnal - Working on implementing a parallel version of the code.

Progress after the proposal is presented in this page precisely.

First Important Milestone

The first milestone achieved in the project, and probably the most important one, was the implementation of the serial code of the Cooley-Tukey algorithm. This was implemented in C++. For the implementation, the definition of multiple functions like BitReversal(), FillArray(), CalcFFT(), GetTimeElapsed(), GetTwiddleFactors() was needed. This modular structure makes the code more readable and understandable.

First the 1-D array is filled with random numbers using the FillArray() function that fills the array with relevant time domain values.

Since the Cooley-Tukey Radix-2 algorithm is being implemented in this case, the array is divided into two parts, and the data is converted into a bitstream. The first half is retained as it is but the second bitstream is reversed in order to perform the calculation. This is done using the BitReversal() fucntion.

For calculating the Twiddle factors, we make use of the GetTwiddleFactors() function, and these factors are required to call the CalcFFT() function in order to calculate the frequency domain value.

Partially Completed Second Important Milestone

For this we had to decide which parallel programming model to use for optimizing the performance of the code. We decided on OpenMP for the following reasons:-

1. Easy to implement
2. Not many changes required to the original algorithm or the code in general
3. Shared memory model
4. Number of threads can be set dynamically based on system hardware

Objectives achieved for the duration of the project are presented in this page.

Important Milestones Achieved

We successfully parallelized the code using OpenMP. Following are some of the stats when we ran the code for an array size of 1024 for different number of OpenMP threads

1. 1 thread = 0.519248 seconds
2. 2 threads = 0.705714 seconds
3. 4 threads = 0.20991083 seconds
4. 8 threads = 0.1860775 seconds (best performance)
5. 16 threads = 0.3017322 seconds

The results show that the performance is best when the number of threads launched when it is equal to the number of cores present. In our case, we ran this is on a 4-core machine with hyperthreading, resulting in a total of 8 virtual cores. Hence the performance for 8 threads is the best. As you increase the number of threads, the performance will decrease since time is wasted on thread synchronization and worksharing. Hence, it is best to run 8 threads in my case, or a number equal to the number of cores on your system.