The Fast Fourier Transform (FFT) is one of the basic algorithms in CS, known by everybody who knows anything about CS.
The Fast Fourier Transform (FFT) is one of the basic algorithms in CS, known by everybody who knows anything about CS.

The Discrete Fourier Transform (DFT) takes a sequence of $n$ (complex) numbers and returns a sequence of $n$ numbers.
The Fast Fourier Transform (FFT) is one of the basic algorithms in CS, known by everybody who knows anything about CS.

The Discrete Fourier Transform (DFT) takes a sequence of $n$ (complex) numbers and returns a sequence of $n$ numbers.

If the input numbers represent a signal, the DFT values represent the constituent frequencies of that signal.
The Fast Fourier Transform (FFT) is one of the basic algorithms in CS, known by everybody who knows anything about CS.

The Discrete Fourier Transform (DFT) takes a sequence of $n$ (complex) numbers and returns a sequence of $n$ numbers.

If the input numbers represent a signal, the DFT values represent the constituent frequencies of that signal.

$$y_k = \sum_{j=0}^{n-1} x_j e^{-2\pi ijk/n}, \text{ for } 0 \leq k < n$$

The $n$ values $x_i$ are input; the $n$ values $y_i$ are output.
Parallel Algorithms

FFT

This has two obvious elements of parallelism:
This has two obvious elements of parallelism:

- each $y_k$ can be computed independently, for a $n$-way parallelism
This has two obvious elements of parallelism:

- each $y_k$ can be computed independently, for a $n$-way parallelism
- each summation can be done as a tree, for a $\log n$-way parallelism
Parallel Algorithms

FFT

This has two obvious elements of parallelism:

- each $y_k$ can be computed independently, for a $n$-way parallelism
- each summation can be done as a tree, for a $\log n$-way parallelism
- taking total time $O(\log n)$ on $O(n^2)$ processors
Parallel Algorithms

FFT

This has two obvious elements of parallelism:

- each $y_k$ can be computed independently, for a $n$-way parallelism
- each summation can be done as a tree, for a $\log n$-way parallelism
- taking total time $O(\log n)$ on $O(n^2)$ processors

But, instead let us look at a sequential divide and conquer version
Parallel Algorithms

FFT

This sum can be computed as presented: summing $n$ values for each of $n$ values $y_k$, thus taking time $O(n^2)$
Parallel Algorithms

FFT

This sum can be computed as presented: summing $n$ values for each of $n$ values $y_k$, thus taking time $O(n^2)$

However, if $n$ is even, then we get a nice recursive presentation by splitting the sum into evens and odds

$$y_k = \sum_{j=0}^{n-1} x_j e^{-2\pi ijk/n}$$

$$= \sum_{j=0}^{n/2-1} x_{2j} e^{-2\pi i(2j)k/n} + \sum_{j=0}^{n/2-1} x_{2j+1} e^{-2\pi i(2j+1)k/n}$$

$$= \sum_{j=0}^{n/2-1} x_{2j} e^{-2\pi ijk/(n/2)} + e^{-2\pi ik/n} \sum_{j=0}^{n/2-1} x_{2j+1} e^{-2\pi ijk/(n/2)}$$
Parallel Algorithms

FFT

This is just two half-size DFTs
This is just two half-size DFTs

For $n$ a power of 2 we can repeat recursively, leading to the \textit{Fast Fourier Transform}, a way to implement the DFT
This is just two half-size DFTs

For $n$ a power of 2 we can repeat recursively, leading to the *Fast Fourier Transform*, a way to implement the DFT

In fact, the FFT is an unwinding of the recursion into an iteration that runs slightly faster, but is harder to understand
This is just two half-size DFTs

For \( n \) a power of 2 we can repeat recursively, leading to the **Fast Fourier Transform**, a way to implement the DFT

In fact, the FFT is an unwinding of the recursion into an iteration that runs slightly faster, but is harder to understand

The FFT takes sequential time \( O(n \log n) \), which is a huge improvement over \( O(n^2) \); e.g., for \( n = 1,000,000 \), this is about 20,000,000 against 1,000,000,000,000
This is just two half-size DFTs

For $n$ a power of 2 we can repeat recursively, leading to the *Fast Fourier Transform*, a way to implement the DFT

In fact, the FFT is an unwinding of the recursion into an iteration that runs slightly faster, but is harder to understand

The FFT takes sequential time $O(n \log n)$, which is a huge improvement over $O(n^2)$; e.g., for $n = 1,000,000$, this is about 20,000,000 against 1,000,000,000,000

But, for our purposes, we can see this as a simple divide and conquer, thus easily parallelisable
Parallel Algorithms

FFT

The parallelisation of the FFT works in a way very similar to what we have seen before and has complexity $O(\log n)$ on $O(n)$ processors, and $O(\log p + (n/p) \log(n/p))$ on $p$ processors.
The parallelisation of the FFT works in a way very similar to what we have seen before and has complexity $O(\log n)$ on $O(n)$ processors, and $O(\log p + (n/p) \log(n/p))$ on $p$ processors.

As the FFT is such an important algorithm, much has been written about it and its parallel variants, in particular matching it to the various kinds of hardware (SIMD, pipeline, shared memory, etc.)
Parallel Algorithms
And So On

There are very many other parallel algorithms: just think of the large literature on sequential algorithms that exists
There are very many other parallel algorithms: just think of the large literature on sequential algorithms that exists.

We have just looked at a couple, but everything that you have done in the past sequentially will probably have a parallel counterpart.
There are very many other parallel algorithms: just think of the large literature on sequential algorithms that exists.

We have just looked at a couple, but everything that you have done in the past sequentially will probably have a parallel counterpart.

Some algorithms will map best to shared memory, some distributed, some SIMD, and so on.
Parallel Algorithms
And So On

There are very many other parallel algorithms: just think of the large literature on sequential algorithms that exists.

We have just looked at a couple, but everything that you have done in the past sequentially will probably have a parallel counterpart.

Some algorithms will map best to shared memory, some distributed, some SIMD, and so on.

Some will be sensitive to the topology of the architecture (full connect, torus, etc.), others work well regardless.
There are very many other parallel algorithms: just think of the large literature on sequential algorithms that exists.

We have just looked at a couple, but everything that you have done in the past sequentially will probably have a parallel counterpart.

Some algorithms will map best to shared memory, some distributed, some SIMD, and so on.

Some will be sensitive to the topology of the architecture (full connect, torus, etc.), others work well regardless.

Still more will not work well in parallel at all.
There are very many other parallel algorithms: just think of the large literature on sequential algorithms that exists.

We have just looked at a couple, but everything that you have done in the past sequentially will probably have a parallel counterpart.

Some algorithms will map best to shared memory, some distributed, some SIMD, and so on.

Some will be sensitive to the topology of the architecture (full connect, torus, etc.), others work well regardless.

Still more will not work well in parallel at all.

Exercise. Look some up!