In this exercise we still study time measurement, vector processing and parallel processing. This is a very demanding exercise.

## Excercise 13a (Extra exercise, performance measuring, 0.5p)

Discrete Fourier transform (DFT) from time domain to frequency domain is a useful tool for various digital signal processing (DSP) tasks[1]. It is defined as

$$f_k = \sum_{j=0}^{N-1} x_j e^{-\frac{2\pi i j k}{N}}$$

where $x_j$ is the $j$th input signal sample (like x[j]), $f_k$ is the $k$th frequency component, and $N$ is the number of samples used in the transformation process. DFT takes $N$ real or complex input samples $x_j$ and produces $N$ complex output samples $f_k$. The number of multiplications needed in this transform is the number of input samples squared, i.e. complexity of this transform is $O(N^2)$. This makes the DFT unpractical for large values of $N$. If the number of points is $N=2^n$ (this is called the radix-2 –algorithm), the equation can be modified to

$$f_k = \sum_{j=0}^{N/2-1} x_{2j} e^{-\frac{2\pi i j k}{N/2}} + e^{\frac{2\pi i}{N}} \sum_{j=0}^{N/2-1} x_{2j+1} e^{-\frac{2\pi i}{N/2}}$$

where we first take $N/2$ samples discrete Fourier transform from the even index numbered samples (e.g. x[0], x[2], x[4], …) and then another discrete Fourier transform from the odd numbered samples (e.g. x[1], x[3], x[5], …), and finally combining the results by multiplying the latter transform by $e^{\frac{2\pi i k}{N}}$ and adding results together. This is an example of divide-and-conquer algorithm design technique. With this technique the complexity of the discrete Fourier transform can be shown to be $O(N\log_2 N)$ that is much smaller (than the original transform) when $N$ is large. Therefore, implementation based on this concept is called a Fast Fourier Transform (FFT).

You don't need to bother yourself implementing this FFT; a straightforward recursive implementation of the latter equation is given here (and in a file fft.cpp, available from Ovi).

```
typedef complex<double> cx;
// radix-2 in-place FFT, n must be 2^k (e.g. 2,4,8,16…)
void fft(int n, cx x[]) {
        const cx J(0, 1);
        const double PI = 3.1415926536;
        // check the trivial case
        if (n == 1)
                return;

        // perform two sub-transforms
        int n2 = n/2; // size of sub-transform
        cx *xe = new cx[n2];
        cx *xo = new cx[n2];
        for (int i = 0; i < n2; i++) { // perform n/2 DIF 'butterflies'
                xe[i] = x[i] + x[i+n2];                      // even subset
                xo[i] = (x[i] - x[i+n2])*exp(-J*(2*PI*i/n)); // odd subset
        }
        fft(n2, xe);
```

[1] See Proakis, J., G., Manolakis, D., G.: "Digital Signal Processing, Principles, Algorithms and Applications", Prentice-Hall, 2007

```
        fft(n2, xo);

        // construct the result vector
        for (int k = 0; k < n2; k++) {
                x[2*k]   = xe[k]; // even k
                x[2*k+1] = xo[k]; // odd k
        }

        delete[] xe; delete[] xo;
}
```

Your task is to write a program that measures the execution time (in ms) of the given Fast Fourier transform implementation (function `void fft(int n, cm x[])`) using the high performance timer available in C++. In order to make your execution time measurements reliable, you need to run the function more than once (30 times, for example). Store those individual measurements to a suitable C++ container, discard 10 largest values[2] and calculate their mean and standard deviation[3]. Don't use loops in your statistical values calculation, use containers and algorithms instead.

Mean μ and standard deviation σ can be calculated using the equations (we are using unbiased sample variance here)

$$\mu = \frac{x_1 + x_2 + \cdots + x_N}{N}, \sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu)^2}$$

Hint: high_resolution_clock, accumulate, for_each, lambda function

## Extra excercise 13b (Extra exercise, parallelization, 0.5p)

Warning! This is not an easy task.
In the divide-and-conquer technique the algorithm is structured as follows

1. split the original data to two pieces A and B
2. do the operation for the smaller data A
3. do the operation for the smaller data B
4. combine the results

Because A and B data sets are independent, phases 2 and 3 can be done easily in parallel.

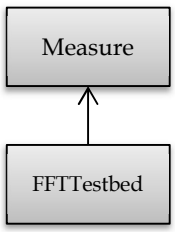
Make the given fast Fourier transform function concurrent using C++ high-level parallel interface and compare the performance of this implementation to the original non-parallel implementation. Do the comparison with multiple values of *N*, e.g. from 2048 to 262144, you may also export your measurement data to the file (csv format) where it can be read to a spread-sheet application (like MS-Excel).

---

[2] Because in multitasking operating system, a task switch can appear in the middle of the function timing, errors tend always be larger than the actual execution time of the function measured. Therefore largest measurement errors are always those largest values.

[3] Standard deviation σ is a useful tool to determine how reliable the sampled mean value is. E.g. with a 95% probability the measured value μ is within a range [μ-2σ, μ+2σ]. See Yates, R., D., Goodman, D., J.: "Probability and Stochastic Processes, A friendly introduction for electrical and computer engineers", Wiley, 2005

Construct your program using object oriented design rules. You should have two classes; Measure class who has a function `void measure(double &mean, double &stdev);` which is responsible to run the function to be tested, measure the execution time and doing this a couple of times in order to calculate statistical information.




The other class, FFTTestbed inherits measurement functions from the Measure class and gives the function to be measured to the Measure class. You should be able to use your measurement system like this

```
const int n = 2048;
double mean, stdev;
void fft(int n, cm x);

FFTTestBed *p = new FFTTestBed(fft, n);
p->measure(mean, stdev);
delete p;
```

Now your measurement system (class Measure) is versatile; you can use it to evaluate different type of functions. FFTTestbed is the cover class to maintain the necessary environment (e.g. setup input data for the function, testing the results, etc.) needed for running the function. If you want to test other functions, you create another class that inherits the class Measure, so you are reusing the class Measure.

In order to reduce the measurement errors (because we are using multitasking operating system which can make task switches during the performance measurement) take 30 measurement and discard 10 worst time measurements. Result of the comparison depends on the hardware used, but one simulation run (Intel i5-661, 3.33 GHz, Windows 7) can be like this:

```
       N        Unopt      Unopt_br     Single    Single_br    Parallel    Paral_br
    2048:    0(± 0)ms    0(± 0)ms    0(± 0)ms    0(± 0)ms    0(± 0)ms    0(± 0)ms
    4096:    0(± 0)ms    0(± 0)ms    0(± 0)ms    0(± 0)ms    0(± 0)ms    0(± 0)ms
    8192:    3(± 9)ms    4(±10)ms    3(± 9)ms    1(± 4)ms    1(± 4)ms    0(± 0)ms
   16384:   10(± 0)ms   10(± 0)ms   10(± 0)ms    8(± 8)ms    5(±10)ms    3(± 9)ms
   32768:   22(± 8)ms   22(± 8)ms   21(± 4)ms   17(± 9)ms   11(± 6)ms    9(± 6)ms
   65536:   50(± 0)ms   50(± 0)ms   50(± 0)ms   38(± 9)ms   29(± 6)ms   19(± 7)ms
  131072:  110(± 0)ms  110(± 0)ms  106(±10)ms   80(± 0)ms   63(± 9)ms   38(± 5)ms
  262144:  231(± 4)ms  236(±10)ms  230(± 0)ms  171(± 4)ms  139(±12)ms   82(± 1)ms
Measurements took 67s
```

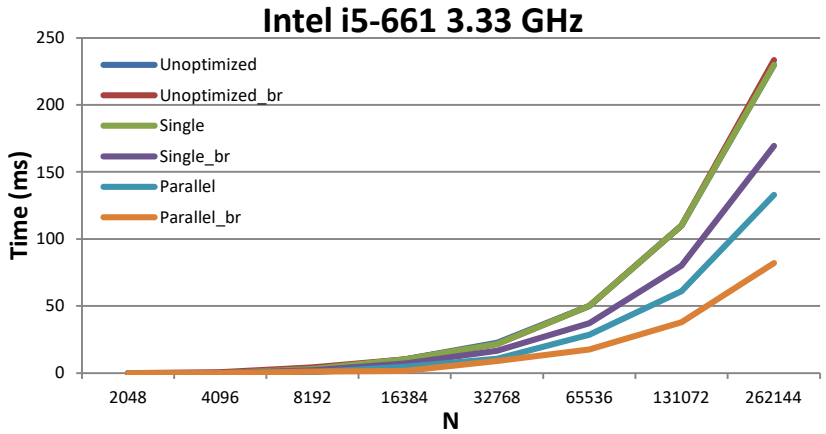
The number in parenthesis is the 95% confidence interval (standard deviation multiplied by two) and _br ending means special variation to the FFT-function where no dynamic memory allocation is needed.

From the table you can immediately find out that the execution time is really $N \times \log N$-dependent on the size of the input vector to be transformed ($N$) because $(262144 \times \log 262144)/(131072 \times \log 131072) = 2{,}12$ and $231/110 = 2{,}10$.

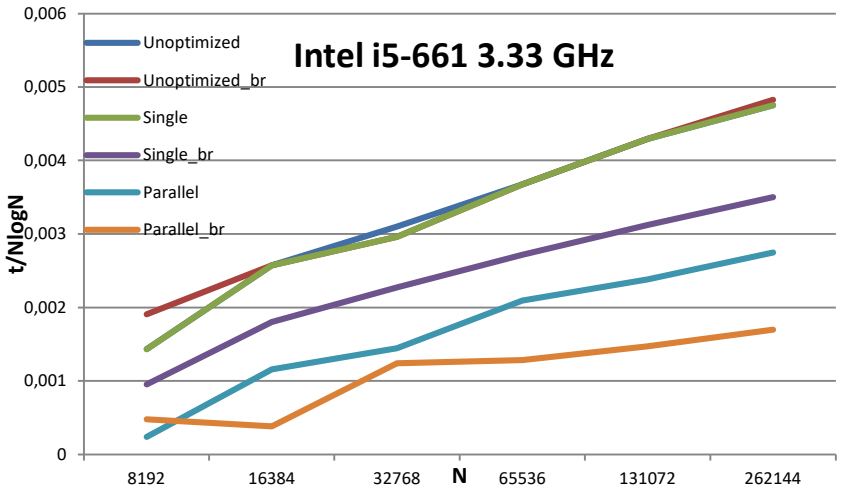
Optimizing the recursive FFT function using Visual Studio 2012 optimizer does not decrease the execution time of unoptimized recursive FFT (231ms) compared to optimized version (230ms). Compiler's optimization process is dependent on the code; unoptimized recursive FFT without dynamic memory allocation has a running time of 236 ms and optimized version has a running time of 171 ms, an improvement of 28%.

Parallelization of the optimized single threaded recursive FFT (230ms) decreases the execution time to 139ms, 40% reduction in execution speed. Theoretical 50% reduction (i5-661 processor has two cores) cannot be obtained in practice because there are overhead in the parallelization process (thread creation/deletion times, etc.).

Importing .csv type file to MS-Excel enables graph plotting utilities.




Notice that the horizontal axis is exponential, therefore the curve seems to be exponential, even it is of $N \times \log N$ type. If we scale the vertical axis by $N \times \log N$, the curve should be constant. Curves are not exactly constant because there are some additional overhead (memory handling with very large memory blocks) whose proportion to the total execution time increases a little when vectors are larger.




Hint: future, async, virtual function