# Отчёт о выполнении лаб. работы «Реализация многопоточного FFT»

Олег Иващенко, МСУ201, февраль 2021

Возьмём в качестве основы классическую реализацию FFT, например https://ru.wikibooks.org/wiki/

```
public class FFT
{
    private static Complex w(int k, int N)
    {
        if (k % N == 0) return 1;
        double arg = -2 * Math.PI * k / N;
        return new Complex(Math.Cos(arg), Math.Sin(arg));
    }

    // Исходная функция FFT (использует рекурсивный подход "сверху-вниз")
    public static Complex[] fft(Complex[] x)
    {
        Complex[] X;
        int N = x.Length;
        if (N == 2)
        {
            X = new Complex[2];
            X[0] = x[0] + x[1];
            X[1] = x[0] - x[1];
        }
        else
        {
            Complex[] x_even = new Complex[N / 2];
            Complex[] x_odd = new Complex[N / 2];
            for (int i = 0; i < N / 2; i++)
            {
                x_even[i] = x[2 * i];
                x_odd[i] = x[2 * i + 1];
            }
            Complex[] X_even = fft(x_even);
            Complex[] X_odd = fft(x_odd);
            X = new Complex[N];
            for (int i = 0; i < N / 2; i++)
            {
                X[i] = X_even[i] + w(i, N) * X_odd[i];
                X[i + N / 2] = X_even[i] - w(i, N) * X_odd[i];
            }
        }
        return X;
    }
}
```

Отметим простоту и наглядность данной реализации. Здесь исходный массив разделяется на два – содержащие чётные и нечётные элементы исходного, а затем рекурсивным образом вычисляется fft от этих массивов – вплоть до массивов размера 2x2.

С помощью класса Parallel библиотеки TPL можно «распараллелить» и ускорить данный алгоритм буквально в «несколько строк кода»:

```
// параллельная fft, до 2x раз быстрее
public static Complex[] pfft(Complex[] x, int level=1, int forkLevel=1)
{
// ...фрагмент вырезан
        Complex[] X_even = null, X_odd = null;
        if (level < forkLevel)
        {
            Parallel.Invoke(
                () => X_even = pfft(x_even, level + 1, forkLevel),
                () => X_odd = pfft(x_odd, level + 1, forkLevel));
        }
```

```
        else
        {
            X_even = fft(x_even);
            X_odd = fft(x_odd);
        }
// ...фрагмент вырезан

    }
    return X;
}
```

Здесь для вычисления fft от вложенных массивов запускаются потоки до уровня рекурсии не более log2(кол-во CPU). Таким способом удаётся ускорить алгоритм в 2 раза. (Если исходное 16x fft выполнялось за 16+2*8+4*4+8*2 = 64 шага, то распараллеливание рекурсивных вызовов даёт 16+8+4+2= 30 шагов)…

Для реализации настоящего «распараллеливания» предлагается алгоритм, не использующий рекурсию. Он выполняет последовательные fft «снизу вверх», слой за слоем – начиная с размер «окна» 2, 4, 8 и так далее.

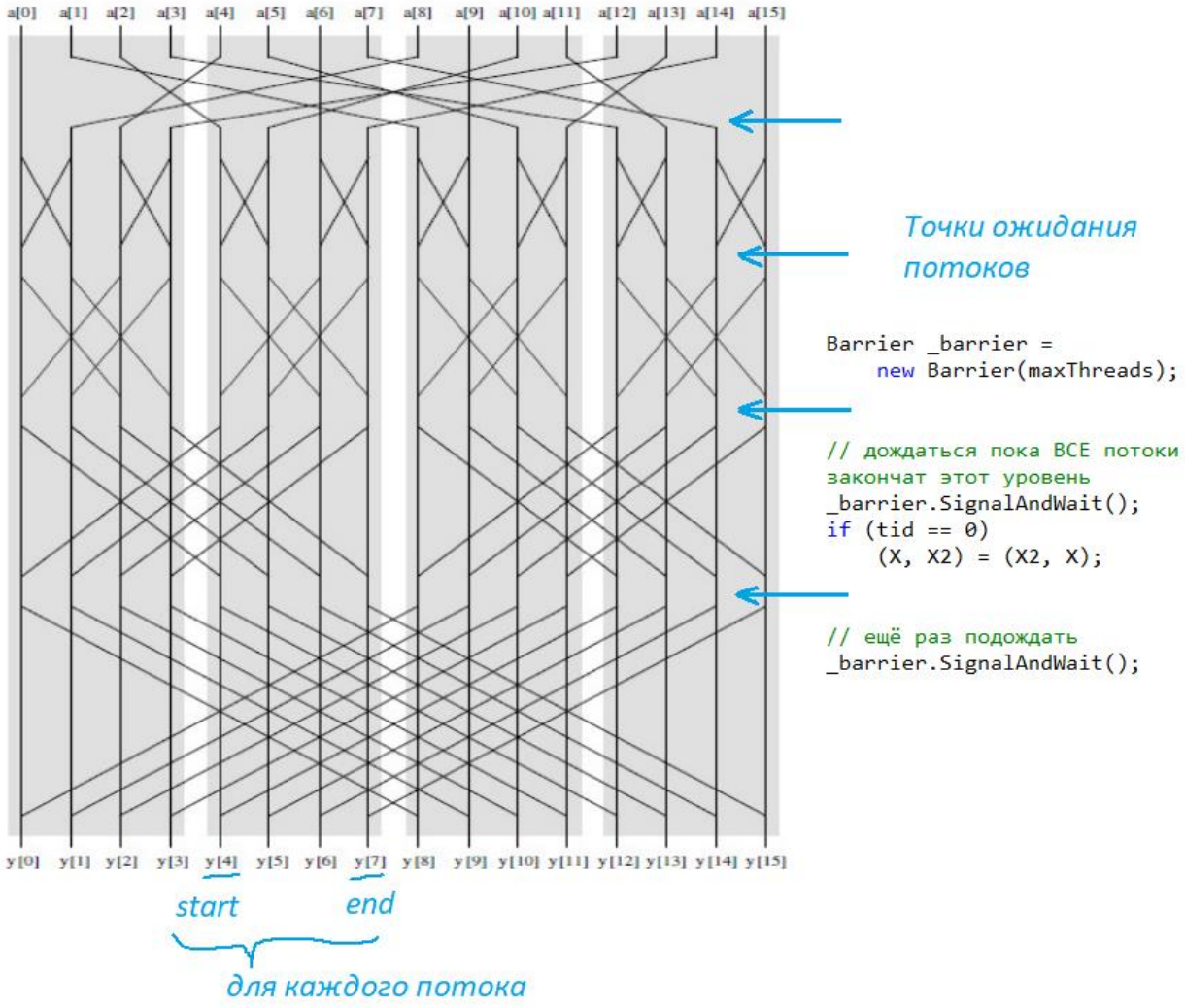
```
// FFT снизу-вверх, без рекурсии
public static Complex[] fft2(Complex[] x)
{
    int N = x.Length, high_bit = N / 2;
    Complex[] X = new Complex[N];
    Complex[] X2 = new Complex[N];

    for (int n = 2; n <= N; n *= 2)
    {
        for (int i = 0; i < N; i += n)
        {
            for (int j = i; j < i + n / 2; j++)
            {
                if (n == 2)
                {
                    int jj = Reverse(j, high_bit);
                    int jj1 = Reverse(j + 1, high_bit);
                    X2[j] = x[jj] + x[jj1];
                    X2[j + 1] = x[jj] - x[jj1];
                }
                else
                {
                    X2[j] = X[j] + w(j, n) * X[j + n / 2];
                    X2[j + n / 2] = X[j] - w(j, n) * X[j + n / 2];
                }
            }
        }
        (X, X2) = (X2, X);
    }
    return X;
}
```

Для распараллеливания опять воспользуемся *Parallel.For*, и в каждом потоке будем выполнять операции от элемента с индексом *start* до элемента с индексом *end*.

```
// границы потока не пересекают блок. пропускаем!
if (i >= end || (i + n) < start) continue;
    for (int j = i; j < i + n / 2; j++) // индекс элемента блока
    {
        if (j < start || j >= end) continue;
```

Самый тонкий момент в этом подходе – это необходимость дождаться окончания работы всех потоков перед началом работы в следующим «слое» - с новым окном размера 2x. Самым простым и логичным является использование объекта класса Barrier, который решает эту задачу одной строкой кода:

```
_barrier.SignalAndWait();    // дождаться пока ВСЕ потоки закончат этот уровень
```




В итоге получается параллельная реализация fft, ускоряющая вычисление преобразования Фурье в 5 раз на 6 ядрах процессора.


Исходный код лаб. работы: https://github.com/ivashchenko/hse-labs/tree/master/FftLab2

Ссылки:


[1] Parallel Fast Fourier Transform

[2] http://www.themobilestudio.net/the-fourier-transform-part-12