Задание 1. Поиск минимального значения в векторе

Необходимо найти минимальное значение среди элементов вектора. Для этого воспользуемся тремя подходами:

- Перебор значений в одном потоке
- Распараллеливание по данным в цикле:
  - С обеспечением синхронизации за счет помещения операции сравнения элементов в критическую секцию
  - С использованием редукции оператором `min`

Описание подходов

Все описанные алгоритмы находятся в модуле `vectorMinValue`.

Перебор значений в одном потоке

Реализация данного подхода содержится в методе `FindMinSingleThread`. Алгоритм предельно прост - сравниваем все элементы, пока не найдем среди них минимальный.

```
int FindMinSingleThread(int *vector, int size) {
    int *end = vector + size;
    int min = INT_MAX;
    for (int *start = vector; start < end; start++) {
        if (min > *start) {
            min = *start;
        }
    }
    return min;
}
```

Синхронизация за счет критической секции

Данный подход отличается от предыдущего наличием распараллеливания по данным в цикле. Чтобы обеспечить синхронизацию потоков, операция сравнения текущего элемента с текущим минимумом помещена в критическую секцию.

```
int FindMinWithForLoopParallelism(int *vector, int size) {
    int *end = vector + size;
    int min = INT_MAX;
#pragma omp parallel for shared(vector, end, min) default(none)
    for (int *start = vector; start < end; start++) {
        if (min > *start) {
#pragma omp critical
            if (min > *start) {
                min = *start;
            }
        }
```

```
    }
    return min;
}
```

Редукция

Третий подход заключается в использовании операции редукции вместе с оператором min.

```
int FindMinWithReduction(int *vector, int size) {
    int minValue;
    int start;
#pragma omp parallel for shared(vector, size) private(start)
reduction(min: minValue) default(none)
    for (start = 0; start < size; start++) {
        minValue = vector[start];
    }
    return minValue;
}
```

Сравнение эффективности алгоритмов

Для сравнения алгоритмов были произведены замеры времени их работы на массивах, состоящих из 10000, 10000000 и 100000000 элементов. Было проведено 30 экспериментов, их результаты сохранены в файле output.csv. Первые 10 строк таблицы представлены ниже.

In [ ]:
```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.ticker import FormatStrFormatter
%matplotlib inline

dataset = pd.read_csv("output.csv", sep=';')
array_sizes = {10000: "small", 10000000: "medium", 100000000: "large"}
dataset = dataset.astype({'method': 'category',  'array_size': 'category'})
dataset['array_size'] = dataset['array_size'].replace(array_sizes)
print(dataset.head(10))
```

```
             method array_size  elapsed_time
0            single      small        0.0150
1  critical_section      small        0.2538
2         reduction      small        0.0043
3            single     medium       12.2329
4  critical_section     medium        3.1207
5         reduction     medium        3.3028
6            single      large      125.5371
7  critical_section      large       28.2935
8         reduction      large       33.2001
9            single      small        0.0141
```

Рассчитаем среднее время работы каждого из описанных подходов при каждом из имеющихся размеров массивов.

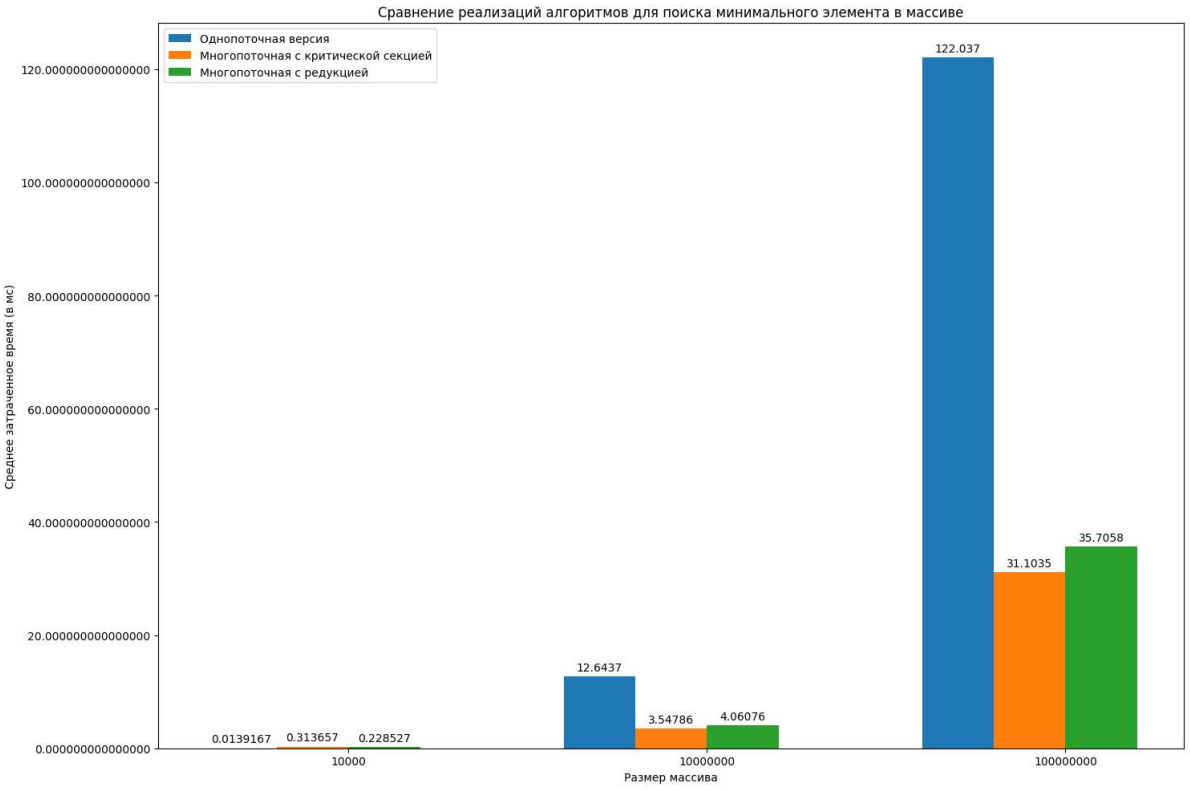
In [ ]:
```
means_for_single_thread = dataset[dataset['method'] == 'single'][['array_size', 'el
```

```
means_for_critical_section = dataset[dataset['method'] == 'critical_section'][['arr
means_for_reduction = dataset[dataset['method'] == 'reduction'][['array_size', 'ela
```

In [ ]:
```
def visualize(ylabel, title, data):
    labels = array_sizes.keys()
    x = np.arange(len(labels))
    width = 0.2

    fig, ax = plt.subplots(figsize=(15,10))
    rects1 = ax.bar(x - 3*width/2, data['single'], width, label='Однопоточная верси
    rects2 = ax.bar(x - width/2, data['critical_section'], width, label='Многопоточ
    rects3 = ax.bar(x + width/2, data['reduction'], width, label='Многопоточная с р

    ax.set_ylabel(ylabel)
    ax.set_title(title)
    ax.set_xlabel('Размер массива')
    ax.set_xticks(x, labels)
    ax.yaxis.set_major_formatter(FormatStrFormatter('%.15f'))
    ax.legend()

    ax.bar_label(rects1, padding=3)
    ax.bar_label(rects2, padding=3)
    ax.bar_label(rects3, padding=3)

    fig.tight_layout()
```

Визуализируем данные. Построим гистограмму среднего времени работы каждого из подходов для каждого из доступных размеров массивов.

In [ ]:
```
mean_data = dict(zip(dataset['method'].unique(), [
                means_for_single_thread['elapsed_time'], means_for_critical_sectior
visualize('Среднее затраченное время (в мс)', 'Сравнение реализаций алгоритмов для
```



Легко заметить, что на массивах небольшого размера однопоточная версия работает лучше многопоточных, однако на массивах больших размеров существенно им

уступает. В то же время оба многопоточных подхода работают с примерно одинаковой скоростью.