Laboratory work №1.
Multithreaded processing of array elements

Tasks

1. Implement sequential processing of array elements, for example, multiplying array elements by a number. The number of array elements given as parameter *N*.
2. Implement multithreaded processing of array elements, using array splitting into an equal amount of parts. The number of threads given as parameter *M*.
3. Complete analysis of the effectiveness of multithreaded computing using different parameter values *N* (10, 100, 1000, 100000) and *M* (2, 3, 4, 5, 10). Give results in tabular form.
4. Complete effectiveness analysis with complicating the processing of each array element.
5. Explore efficiency of range separation with an uneven computational complexity of processing array elements.
6. Explore efficiency of parallelism with circular separation of elements. Compare it with partition by range efficiency.

Methodical guidelines

In this work explores parallelization effectiveness of independent processing of vector elements. In the first task, as processing method might be chosen one of mathematical transformation of vector elements method:

```
for(int i = 0; i < a.Length; i++)
     b[i] = Math.Pow(a[i], 1.789);
```

Multithread processing possible in C# with Thread objects. On a multi-core system, multithreading leads to parallelism. Classes for working with threads placed in `System.Threading` namespace.

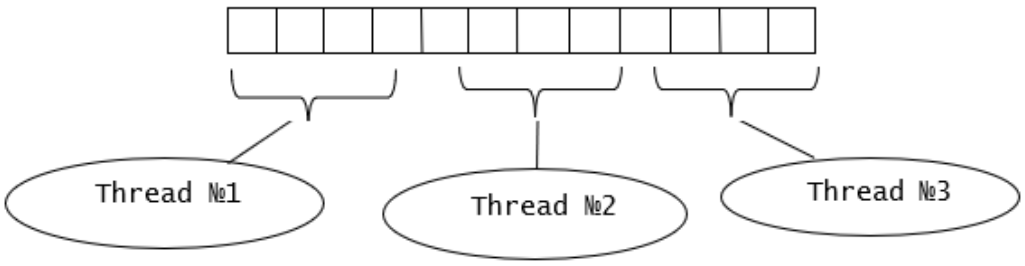
To create a thread need to specify the name of the workflow method, which could be declared in: another class, a class with an entry point as a static method, or as a lambda-expression. The thread method is either not takes any arguments or takes an argument as `object` type. A thread is started by calling Start() method.

```
class Program
{
     static void Run(object some_data)
     {
          int m = (int) some_data;
          ..
     }
     static void Main()
     {
          ..
          Thread thr = new Thread(Run);
          thr.Start(some_data);
     }
}
```

You can wait until the threads are finished using Join method :

```
thr1.Join(); thr2.Join();
```

In the thread method, it is essential to provide the possibility of partition the range 0…(N-1) into the number of threads. On thread startup as an argument is passed either «thread index» which determines array range computing in that thread, or starting and ending array index.



Multithreaded computation will be parallel if the system uses several processors (processor cores). The number of processors can be found using the property:

```
System.Environment.ProcessorCount
```

Besides, parallel computing may be implemented with the TPL (Task Parallel Library). Library classes placed in System.Threading.Tasks namespace. Parallel operations computing over loop elements executed with Parallel.For method:

```
Parallel.For(0, a.Length, i =>
   { b[i] = Math.Pow(a[i], 1.789); });
```

For performance analysis of coherent and parallel computation could be used variables with `DateTime` type. For example,

```
DateTime dt1, dt2;
dt1 = DateTime.Now;
// Computation_method_call;
dt2 = DateTime.Now;
TimeSpan ts = dt2 – dt1;
Console.WriteLine("Total time: {0}", ts.TotalMilliseconds);
```

You can also use the object Stopwatch of System.Diagnostics: namespace

```
Stopwatch sw = new Stopwatch();
sw.Start();
// Computation_method_call;
sw.Stop();
TimeSpan ts = sw.Elapsed;
Console.WriteLine("Total time: {0}", ts.TotalMilliseconds);
```

When do evaluate performance, it's necessary to take into account that the execution time of the algorithm depends on many parameters. Therefore it is desirable to assess average turnaround time with multiple algorithm runs, excluding the first "warm-up" run.

Parallel algorithm efficiency depends heavily on the array elements, number of threads, math function complexity, etc. It should be taken into account that with a small volume of array elements, the overhead associated with organizing multithreaded processing exceeds the gain from parallel processing. With sequentially executing primitive loop processing, performance is achieved by optimal use of the cache memory.

When analyzing the dependence of speed on the number of threads, the number of processor cores should be taken into account. An increase in the number of threads beyond the capabilities of a computing system leads to thread competition and poor performance.

The complication of processing array elements is proposed to be implemented using the internal loop. For example,

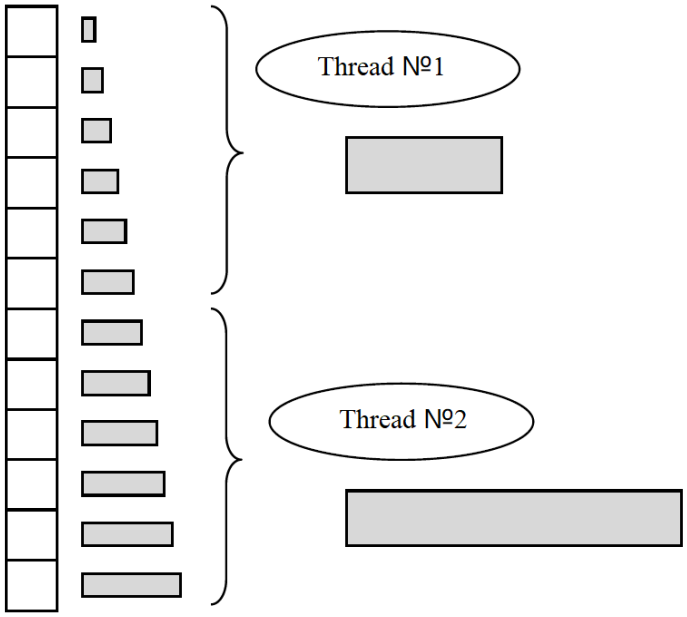
```
for(int i = 0; i < a.Length; i++)
{
     // i-element processing
     for(int j = 0; j < K; j++)
          b[i] += Math.Pow(a[i], 1.789);
}
```

K is a difficult parameter. Increasing that parameter, we observe an increase in the efficiency of parallel processing with a smaller array of numbers.

At each iteration, in the considered processing methods computational load relatively flat. In cases where the computational load depends on element index, array partitioning into equal ranges may not be efficient. Consider the following processing option:

```
for(int i=0; i<a.Length; i++)
{
      // I-element processing
      for(int j=0; j < i; j++)
            b[i] += Math.Pow(a[i], 1.789);
}
```

The computational burden when processing an i-element depends on the `i` index. Processing the initial elements of an array takes less time indifference with last elements processing. Separation of data by range leads to unbalanced loading of threads and decreases the efficiency of parallelization.



One possible approach to equalization of thread loading is the circular decomposition method. In the case of two threads, we receive a scheme where the first thread handle all even elements and the second thread handles all uneven. Implement circle decomposition for *N* threads (*N* > 2).