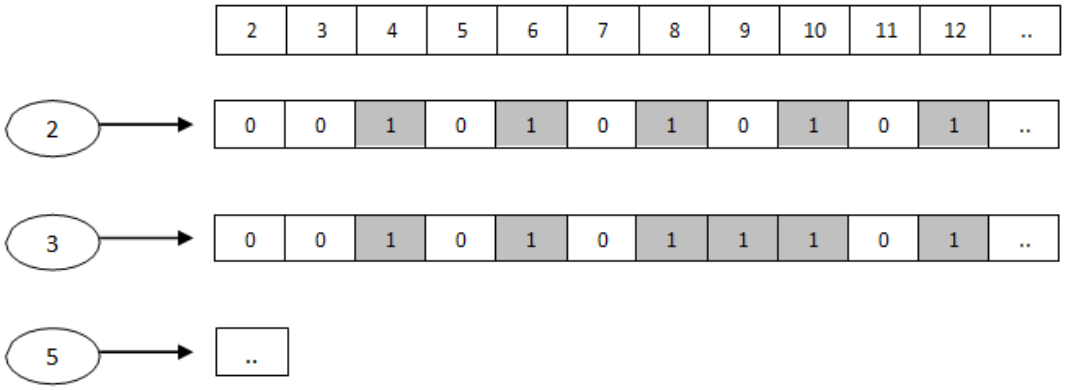
Laboratory work №2.

Finding prime numbers

Tasks: to implement sequential and parallel algorithms for finding prime numbers; to analyze the performance of algorithms for different amounts of data, different number of threads; to calculate the acceleration and efficiency of algorithms; to draw conclusions about the feasibility of using parallel algorithms and the need for synchronization.

The sequential algorithm "Swelto Eratosthena"

The algorithm consists of sequential interaction of already known simple numbers, starting with a two, and checking the decomposability of all the numbers of the range (*m*, *n*] to the specified simple number *m*. In the first step, the number *m* = 2 is selected, the decomposability of the number of ranges (2, *n*] is checked for a two. The numbers that are divided into two are marked as composite and do not participate in further analysis. The following uncommon (simple) number will be *m* = 3, etc.



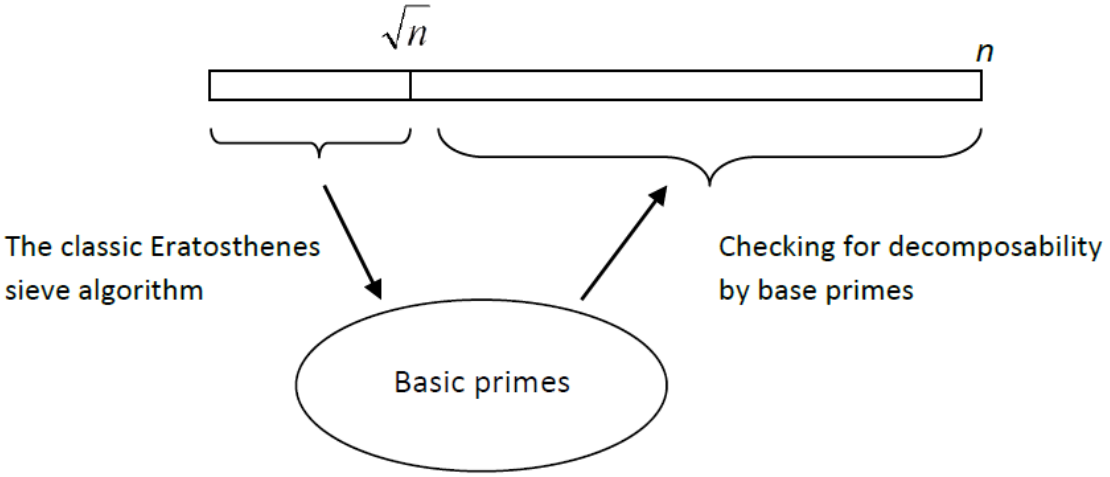
At the same time, it is enough to check the decomposability of numbers on simple numbers in the interval $(2,\sqrt{n}]$ For example, in the range from 2 to 20, we check all the numbers for decomposability 2, 3.

Modified sequential algorithm

In the consecutive algorithm, the basic simple numbers are determined alternately. After the triple, the five follows, as the four is excluded when processing twos. The sequence of finding prime numbers makes it difficult to parallelize the algorithm. Two stages are allocated in the modified algorithm:

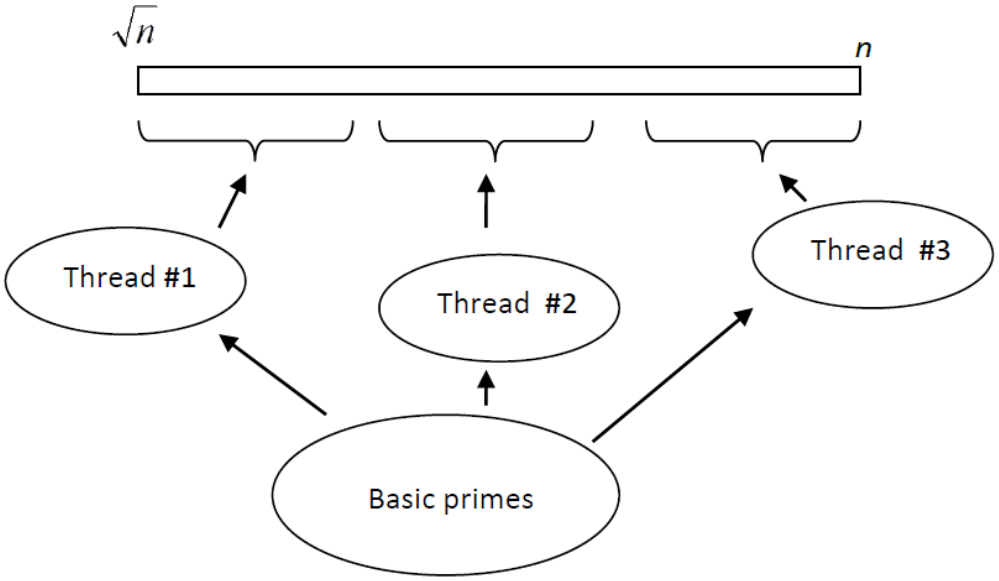
The first stage: the search for simple numbers in the range from 2 ... $\sqrt{n}$ with the help of the classical method of Eratosthenes solution (basic simple numbers).

The second stage: search for prime numbers in the range from $\sqrt{n}$ ... *n*, the test involves the basic primes identified in the first stage.



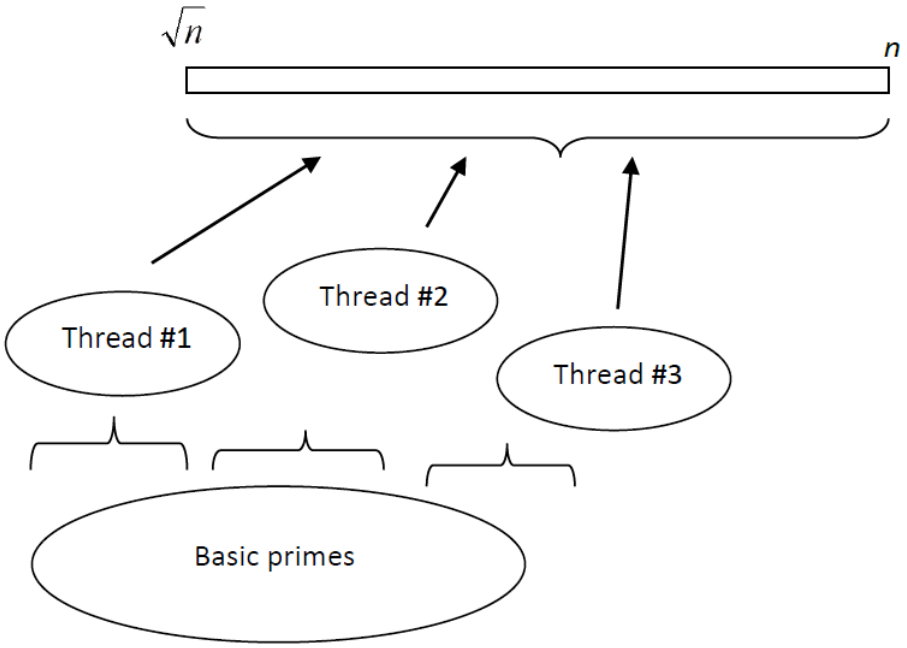
At the first stage of the algorithm, a relatively small amount of work is performed, so it is inappropriate to parallel this stage. At the second stage, the basic simple numbers are checked. Parallel algorithms are developed for the second stage.

Parallel algorithm №1: decomposition by data



The idea of parallelization consists of splitting the range $\sqrt{n}$ ... n on equal parts. Each stream processes its part of the numbers by checking on decomposability for each basic simple number.

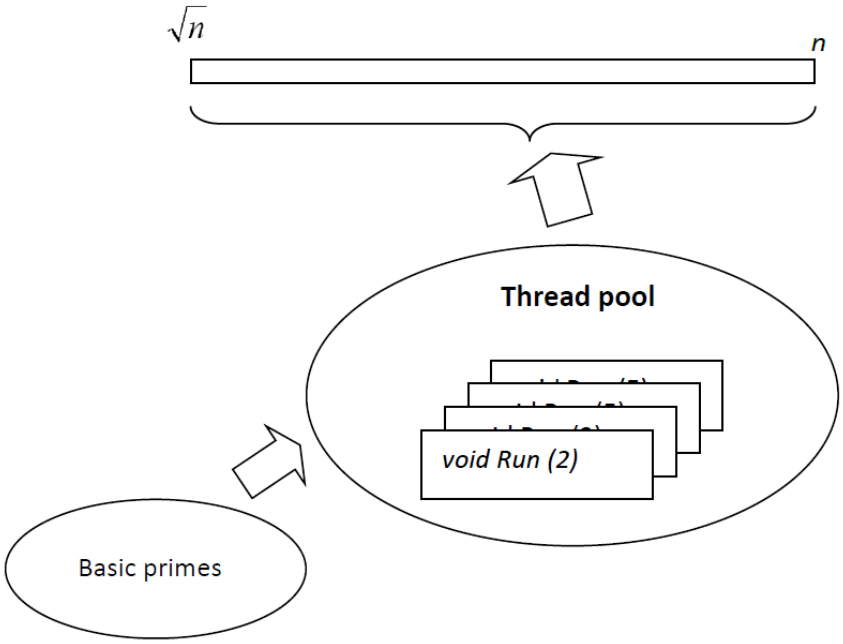
Parallel algorithm №2: decomposition of a set of prime numbers



In this algorithm, basic simple numbers are divided. Each thread works with a limited set of prime numbers and checks the entire range $\sqrt{n}$ ... n

Parallel algorithm №3: thread pool application

Using a thread pool allows you to automate the processing of independent work items. As working elements, it is proposed to use the check of all numbers in the range from $\sqrt{n}$ ... n for decomposability by a single base prime number



To apply a thread pool, you must load the work items along with the required parameters into the thread pool queue:

```
for (int i = 0; i < basePrime.Length; i++)
{
      ThreadPool.QueueUserWorkItem(Run, basePrime[i]);
}
```

`Run` – method for processing all numbers in the range $\sqrt{n}$ ... *n* for decomposability to a prime `basePrime[i]`

Work Items are executed automatically after they are added to the thread pool. There is no built-in mechanism for waiting for the completion of work items added to the thread pool. Therefore, the calling thread (*Main* method) must control completion either using synchronization tools (such as signaling messages) or by using shared variables and a waiting loop in *Main* method.

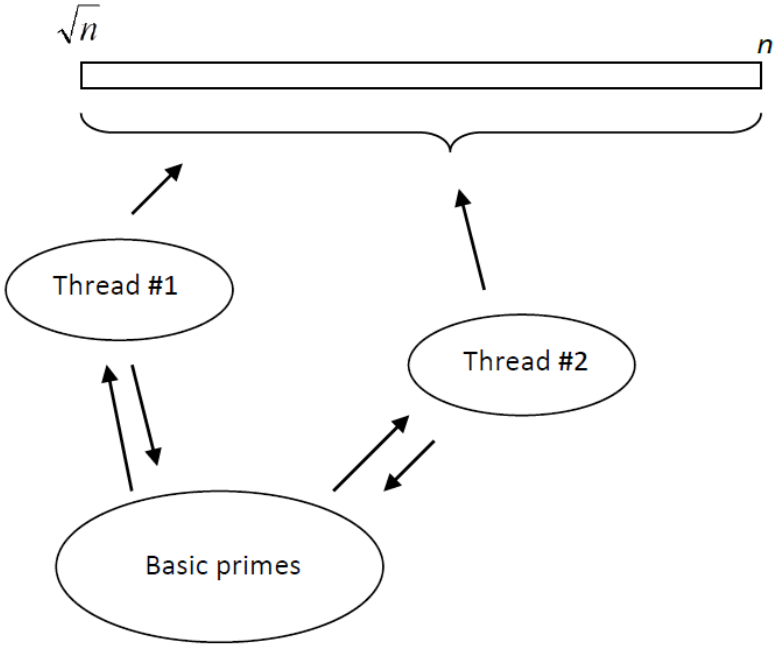
The use of signal messages can be implemented as follows:

```
static void Main()
{
      // Search for basic simple numbers
      ..
      int[] basePrime = ..
      // Declaring an array of signal messages
      ManualResetEvent[] events =
      new ManualResetEvent[basePrime.Length];
      // Adding work items with parameters to the pool
      for (int i = 0; i < basePrime.Length; i++)
      {
            events[i] = new ManualResetEvent(false);
            ThreadPool.QueueUserWorkItem(Run,
            new object[] { basePrime[i], events[i] })
      }
      // Waiting for completion
      WaitHandle.WaitAll(events);
      // Output the results
      ..
}
```

```
static void Run(object o)
{
      int prime = (int)((object[])o)[0];
      // Processing numbers for decomposability to the base prime number
      ManualResetEvent ev = ((object[])o)[1] as ManualResetEvent; //
Обработка чисел на разложимость простому числу prime
      ..
      ev.Set();
}
```

Parallel algorithm №4: sequential iteration of primes

The idea of the algorithm is to sequentially iterate through the base primes by different threads. Each thread checks the entire range for decomposability by a certain prime number. After processing the first prime number, the thread does not terminate but calls for the next raw prime number.



To get the current prime number, the thread executes several statements:

```
while (true)
{
      if (current_index >= basePrime.Length)
            break;
      current_prime = basePrime[current_index];
      current_index++;
      // Обработка текущего простого числа
      ..
}
```

In this implementation, there is a shared resource - an array of prime numbers. When simultaneously accessing the resource, the problem of data racing occurs. The consequence of this problem is:

- excess treatment, if several streams simultaneously receive the same number;
- the missed task – streams, having received one number, consistently increase the current index;
- Exception "Out out the array" when one thread has successfully checked the current index, but before referring to the array element, another thread increases the current index.

To solve problems with sharing, it is necessary to use synchronization tools (critical sections, atomic operators, thread-safe collections).

The critical section allows you to restrict access to the code block if one stream has already started to execute the section operators:

```
lock (sync_obj)
{
      критическая_секция
}
```

`sync_obj` is a synchronization object identifying a critical section (for example, a string constant).