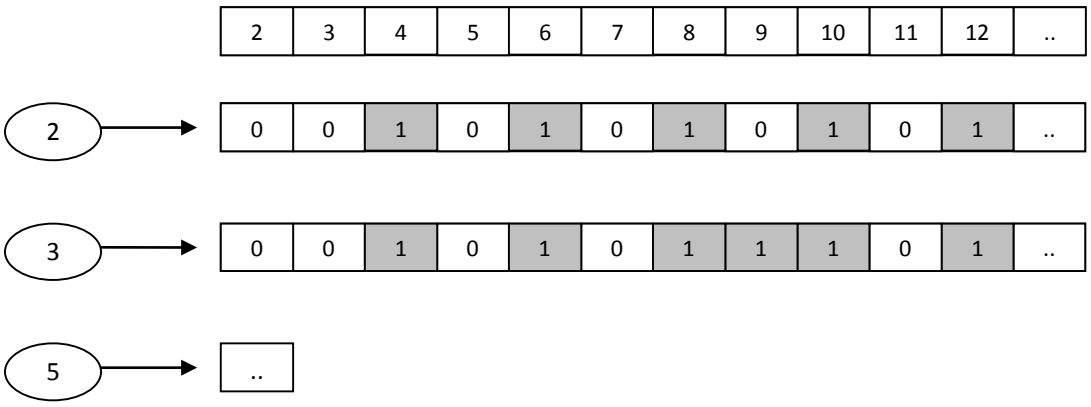
Лабораторная работа №2.
Поиск простых чисел

Задачи: реализовать последовательный и параллельные алгоритмы поиска простых чисел; выполнить анализ быстродействия алгоритмов при разном объеме данных, разном числе потоков; рассчитать ускорение и эффективность выполнения алгоритмов; сделать выводы о целесообразности применения параллельных алгоритмов и необходимости использования синхронизации.

Последовательный алгоритм «Решето Эратосфена».

Алгоритм заключается в последовательном переборе уже известных простых чисел, начиная с двойки, и проверке разложимости всех чисел диапазона (*m, n*] на найденное простое число *m*. На первом шаге выбирается число *m* = 2, проверяется разложимость чисел диапазона (*2, n*] на 2-ку. Числа, которые делятся на двойку, помечаются как составные и не участвуют в дальнейшем анализе. Следующим непомеченным (простым) числом будет *m* = 3, и так далее.



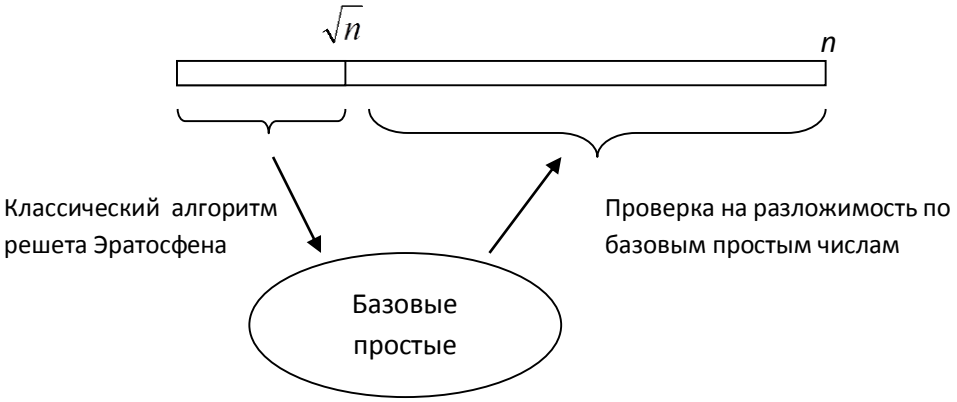
При этом достаточно проверить разложимость чисел на простые числа в интервале (*2,* $\sqrt{n}$]. Например, в интервале от 2 до 20 проверяем все числа на разложимость 2, 3.

Модифицированный последовательный алгоритм поиска

В последовательном алгоритме «базовые» простые числа определяются поочередно. После тройки следует пятерка, так как четверка исключается при обработке двойки. Последовательность нахождения простых чисел затрудняет распараллеливание алгоритма. В модифицированном алгоритме выделяются два этапа:

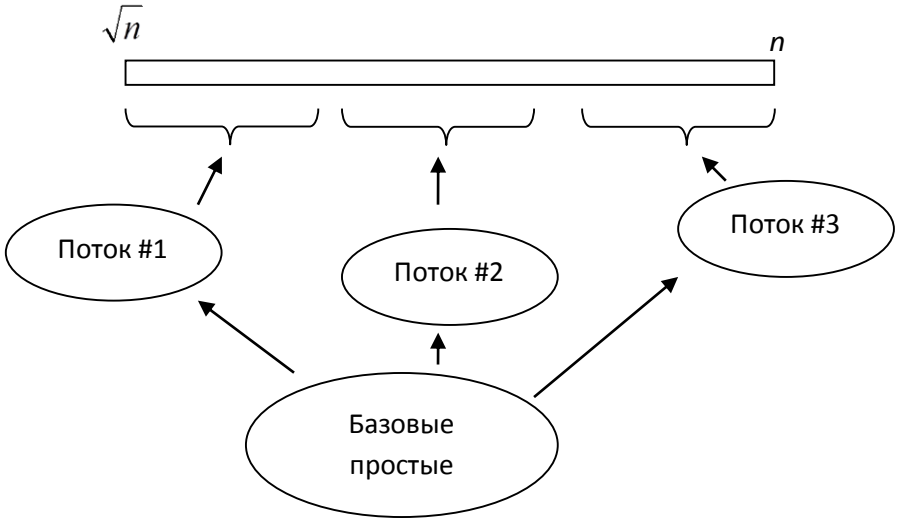
1-ый этап: поиск простых чисел в интервале от 2… $\sqrt{n}$ с помощью классического метода решета Эратосфена (базовые простые числа).

2-ой этап: поиск простых чисел в интервале от $\sqrt{n}$ …*n*, в проверке участвуют базовые простые числа, выявленные на первом этапе.



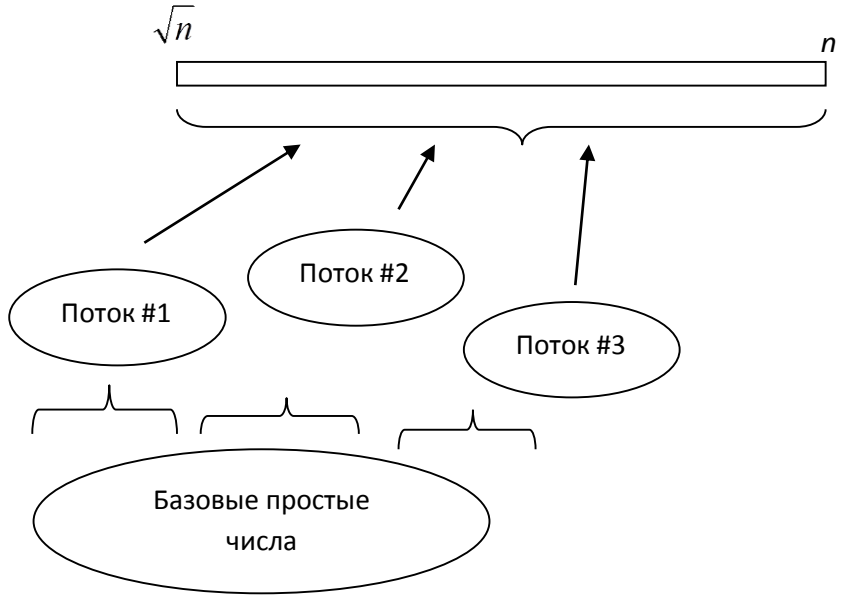
На первом этапе алгоритма выполняется сравнительно небольшой объем работы, поэтому нецелесообразно распараллеливать этот этап. На втором этапе проверяются уже найденные базовые простые числа. Параллельные алгоритмы разрабатываются для второго этапа.

Параллельный алгоритм №1: декомпозиция по данным



Идея распараллеливания заключается в разбиении диапазона $\sqrt{n}$ …*n* на равные части. Каждый поток обрабатывает свою часть чисел, проверяя на разложимость по каждому базовому простому числу.

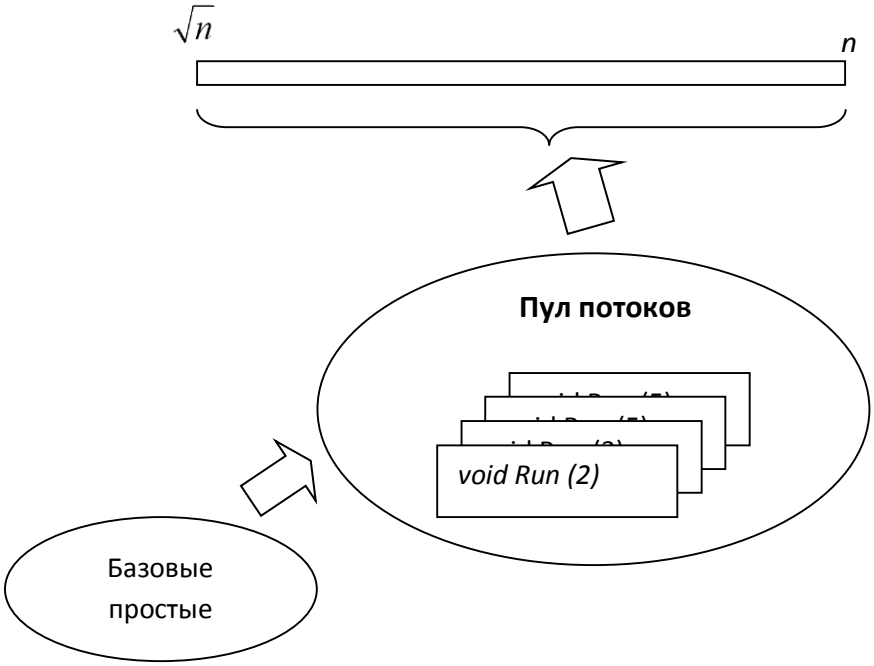
Параллельный алгоритм №2: декомпозиция набора простых чисел



В этом алгоритме разделяются базовые простые числа. Каждый поток работает с ограниченным набором простых чисел и проверяет весь диапазон $\sqrt{n}$ …*n.*

Параллельный алгоритм №3: применение пула потоков

Применение пула потоков позволяет автоматизировать обработку независимых рабочих элементов. В качестве рабочих элементов предлагается использовать проверку всех чисел диапазона от $\sqrt{n}$ …*n* на разложимость по одному базовому простому числу.



Для применения пула потоков необходимо загрузить рабочие элементы вместе с необходимыми параметрами в очередь пула потоков:

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

`Run` – метод обработки всех чисел диапазона $\sqrt{n}$ …n на разложимость простому числу `basePrime[i]`.

Выполнение рабочих элементов осуществляется автоматически после добавления в пул потоков. Не существует встроенного механизма ожидания завершения рабочих элементов, добавленных в пул потоков. Поэтому вызывающий поток (метод Main) должен контролировать завершение либо с помощью средств синхронизации (например, сигнальных сообщений), либо с помощью общих переменных и цикла ожидания в методе Main.

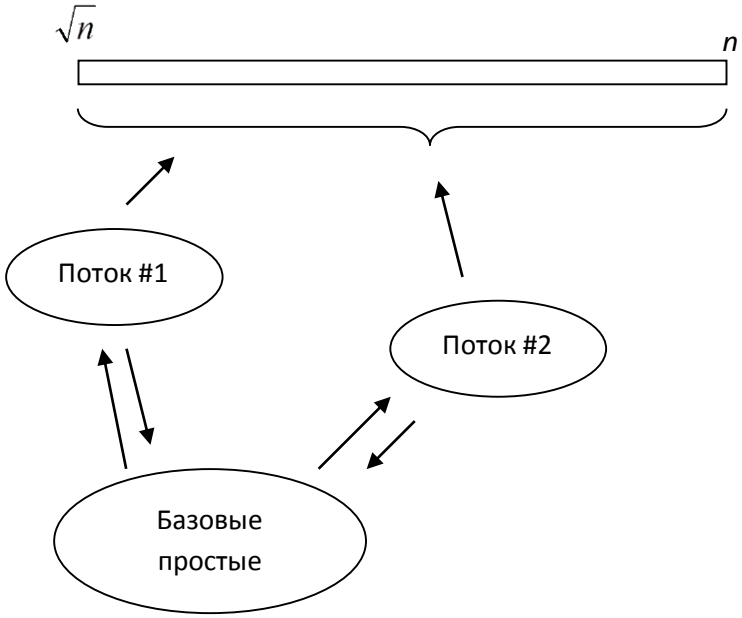
Применение сигнальных сообщений может быть реализовано следующим образом:

```
static void Main()
{
     // Поиск базовых простых
     ..
     int[] basePrime = ..

     // Объявляем массив сигнальных сообщений
     ManualResetEvent [] events =
          new ManualResetEvent [basePrime.Length];

     // Добавляем в пул рабочие элементы с параметрами
     for(int i=0; i<basePrime.Length; i++)
     {
          events[i] = new ManualResetEvent(false);
          ThreadPool.QueueUserWorkItem(Run,
                         new object[] {basePrime[i], events[i]})
     }
     // Дожидаемся завершения
     WaitHandle.WaitAll(events);

     // Выводим результаты
     ..
}
static void Run(object o)
{
     int prime = (int)((object[])o)[0];
```

```
    ManualResetEvent ev = ((object[])o)[1] as ManualResetEvent;
    // Обработка чисел на разложимость простому числу prime
    ..
    ev.Set();
}
```

Параллельный алгоритм №4: последовательный перебор простых чисел

Идея алгоритма заключается в последовательном переборе базовых простых чисел разными потоками. Каждый поток осуществляет проверку всего диапазона на разложимость по определенному простому числу. После обработки первого простого числа поток не завершает работу, а обращается за следующим необработанным простым числом.



Для получения текущего простого числа поток выполняет несколько операторов:

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

В этой реализации существует разделяемый ресурс – массив простых чисел. При одновременном доступе к ресурсу возникает проблема гонки данных. Следствием этой проблемы являются: лишняя обработка, если несколько потоков одновременно получают одно и то же число; пропущенная задача - потоки, получив одно число, последовательно

увеличивают текущий индекс; исключение «Выход за пределы массива», когда один поток успешно прошел проверку текущего индекса, но перед обращением к элементу массива, другой поток увеличивает текущий индекс.

Для устранения проблем с совместным доступом необходимо использовать средства синхронизации (критические секции, атомарные операторы, потокобезопасные коллекции).

Критическая секция позволяет ограничить доступ к блоку кода, если один поток уже начал выполнять операторы секции:

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

где `sync_obj` – объект синхронизации, идентифицирующий критическую секцию (например, строковая константа).