Практическая работа № 7

2 часть

Пример оптимизации параллельного приложения с использованием Currency Visualizer

1. Создадим консольное приложение и назовем его, к примеру, "VisualizerConsoleApplication":

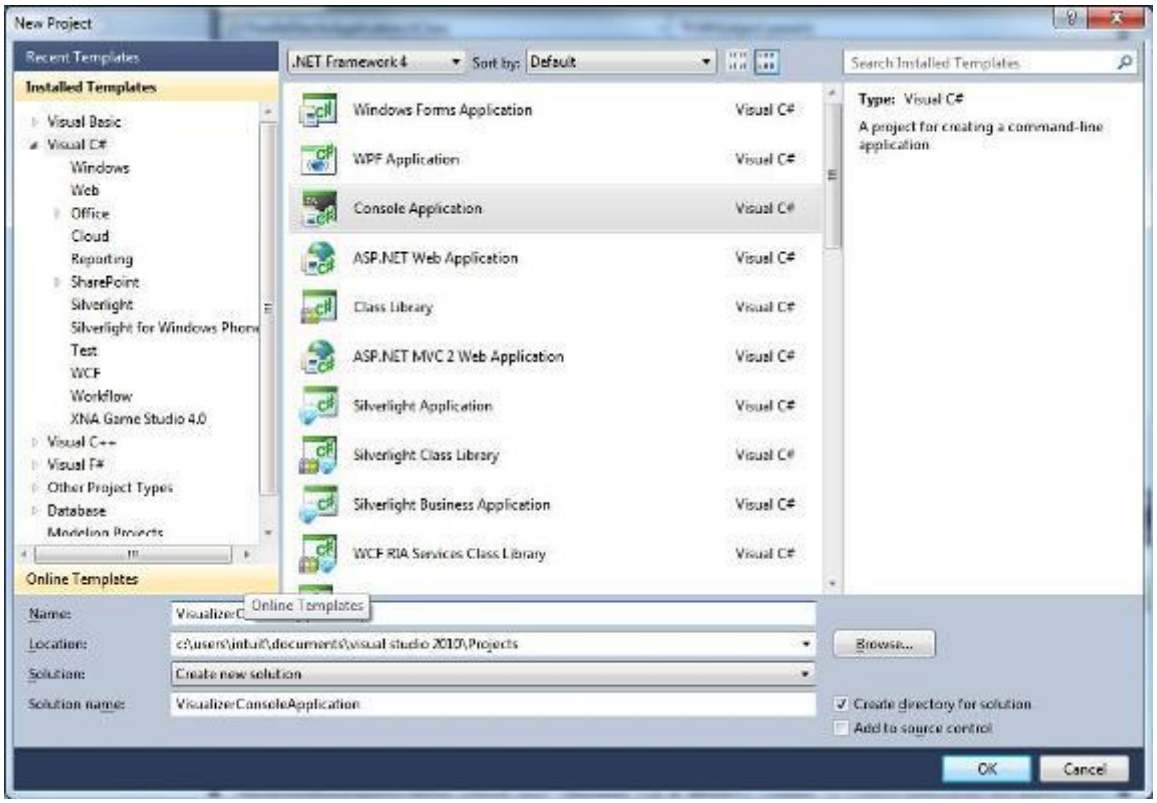

Рис. 1

2. Добавим в начало кода выражения: System.Threading и System.Threading.Task, для работы с потоками. В итоге должно получиться следующее:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace VisualizerConsoleApplication
{
   class Program
   {
      static void Main(string[] args)
      {
      }
   }
}
```

3. После чего, добавим в класс Program два статичных метода:
   - Метод DoSomething() в котором будет вызываться метод Thread.SpinWait();
   - Метод ParallelLoop(), в котором перебираются числа от 0 до 1000 (созданные с помощью метода Enumerable.Range()) с использованием цикла foreach. В цикле foreach вызовем метод DoSomething().

В итоге должно получиться следующее:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
namespace VisualizerConsoleApplication
{
   class Program
   {
      static void DoSomething()
      {
         Thread.SpinWait(int.MaxValue / 10);
      }

      static void ParallelLoop()
      {
         var numbers = Enumerable.Range(0, 1000);
         foreach (var number in numbers)
         {
            DoSomething();
         }
      }

      static void Main(string[] args)
      {
      }
   }
}
```

4. Теперь необходимо вызвать из метода Main() - метод ParallelLoop(), в отдельном потоке. Код будет выглядеть следующим образом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
namespace VisualizerConsoleApplication
{
```

```
class Program
{
   static void DoSomething()
   {
      Thread.SpinWait(int.MaxValue / 10);
   }
   static void ParallelLoop()
   {
      var numbers = Enumerable.Range(0, 1000);
      foreach (var number in numbers)
      {
         DoSomething();
      }
   }
   static void Main(string[] args)
   {
      new Thread(new ThreadStart(ParallelLoop)).Start();
      Console.ReadLine();
   }
}
}
```

5. Далее, необходимо запустить Currency Visualizer. Для этого в меню Debug выбрать пункт "Start Performance Analysis"или можно использовать сочетание клавиш Alt+F2 (Для использования Currency Visualizer, VIsual Studio 2010 должна быть запущена с правами администратора)

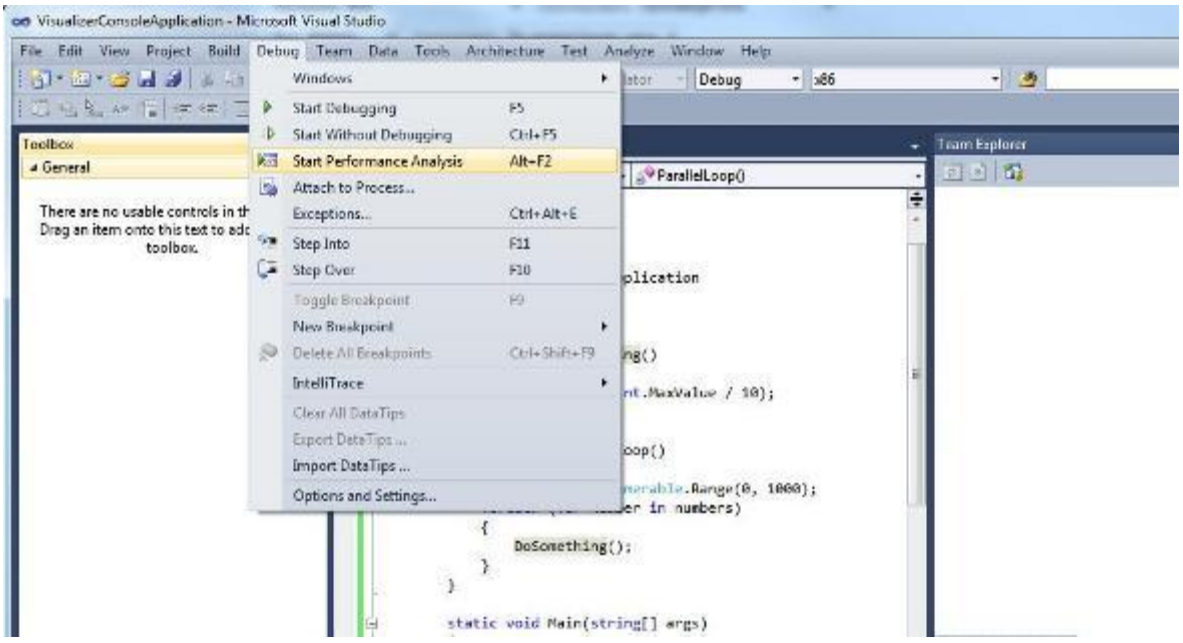

Рис. 2

6. Запустится окно мастера. В нем необходимо выбрать пункт Concurrency и выбрать в нем два пункта "Collect resource contention data" и "Visualize the behavior of a multithreaded application" и для продолжения работы мастера кнопку Next:

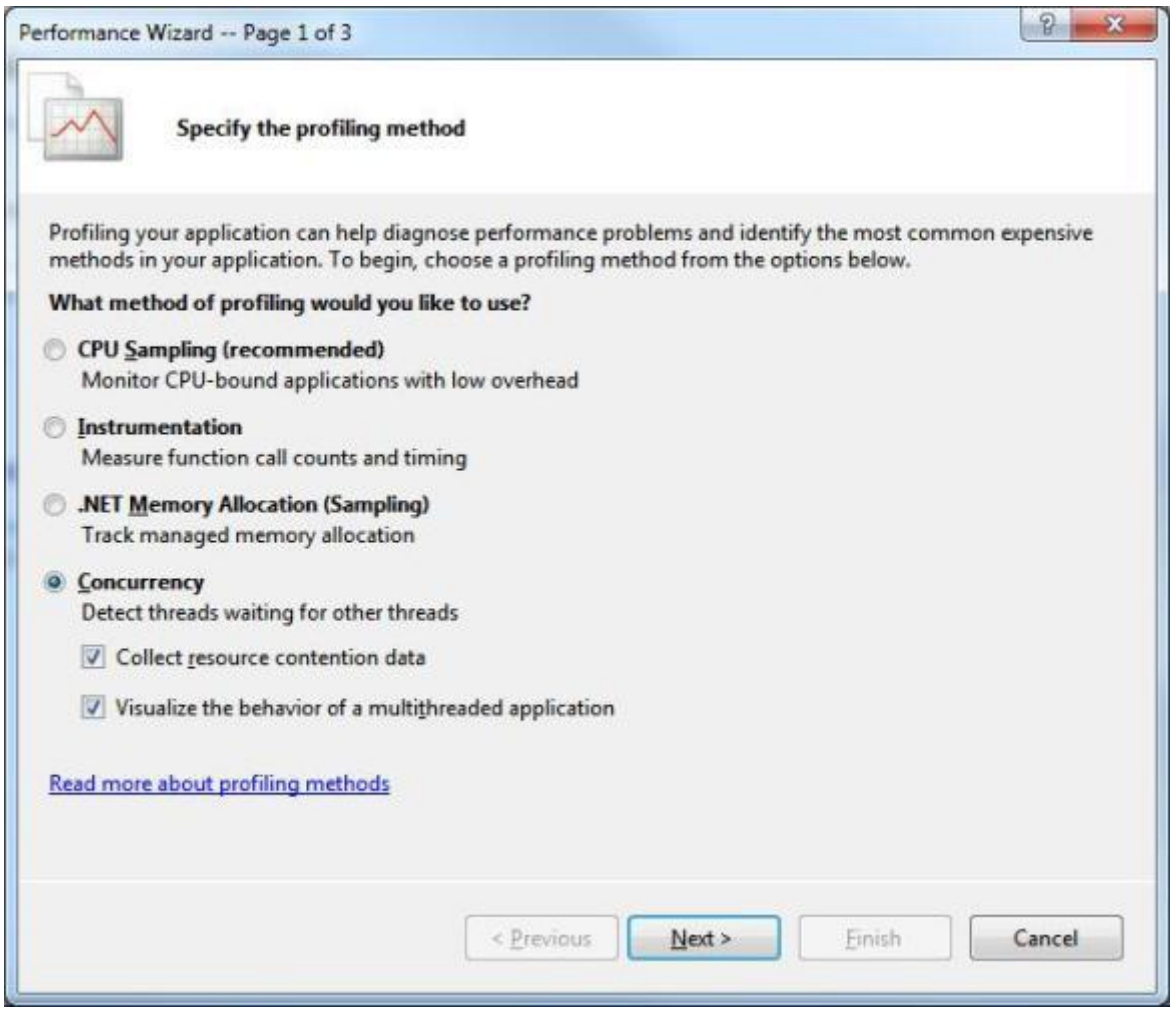

Рис. 3

7. Далее, выбираем проекты для анализа на производительность (если их несколько) и жмем кнопку "Next":

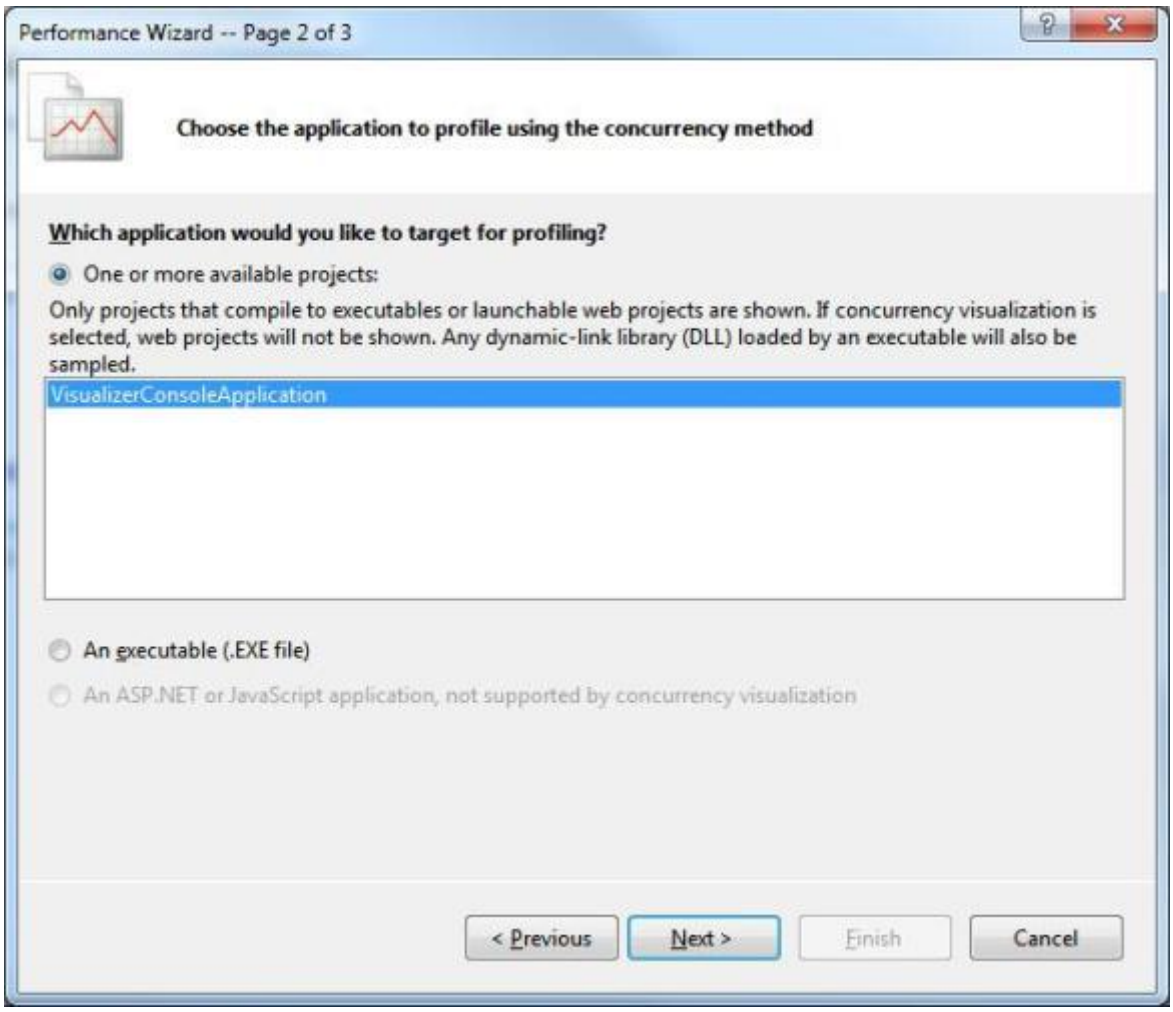

Рис. 4

8. На последнем шаге мастера, выберите пункт "Launch profiling after the wizard finish", после чего нажмите кнопку"Finish":

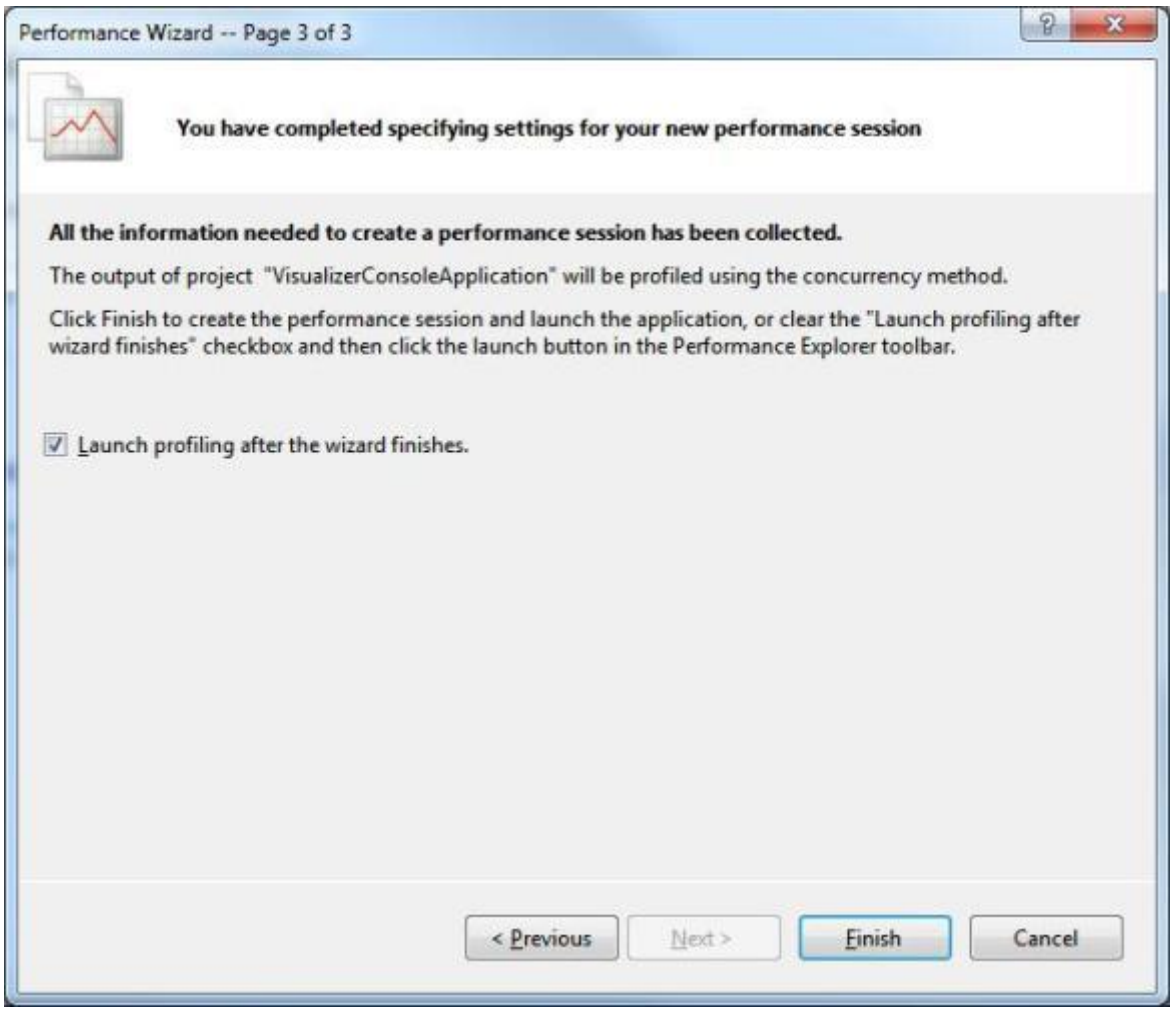

Рис. 5.

9. После чего запустится окно программы (в нашем случае это консоль) и процесс профилирования. Остановите процесс профилирования приложения с помощью кнопки "Stop Profiling" через 10-15 секунд:

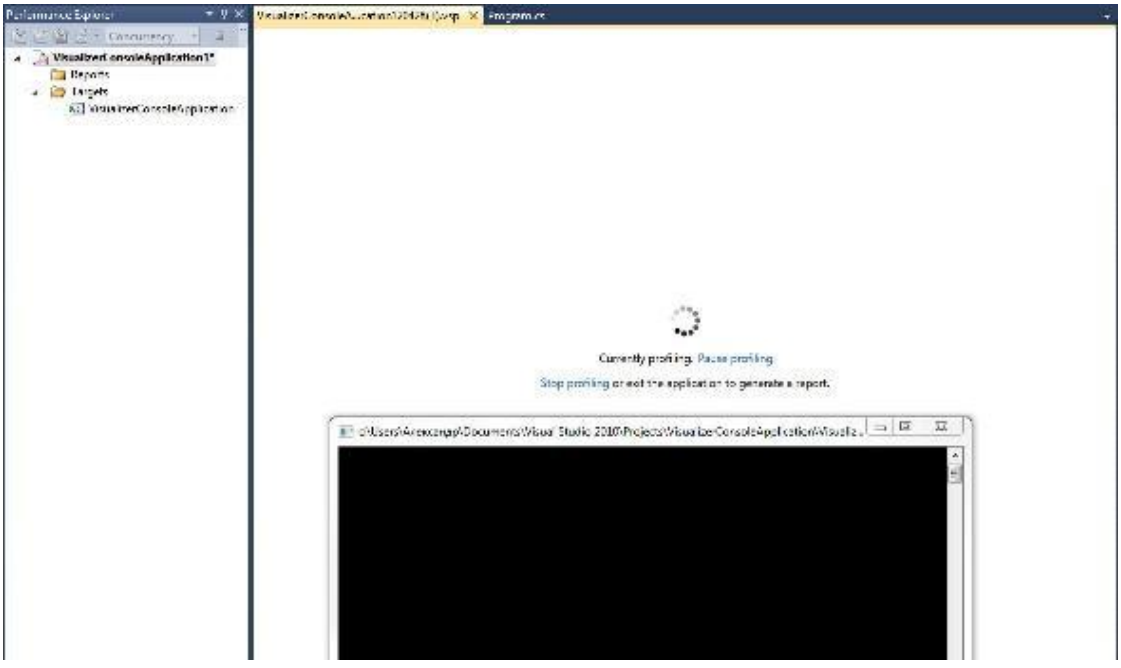

Рис. 6

10. После завершения процесса анализа производительности, отобразится окно с различными отчетами:

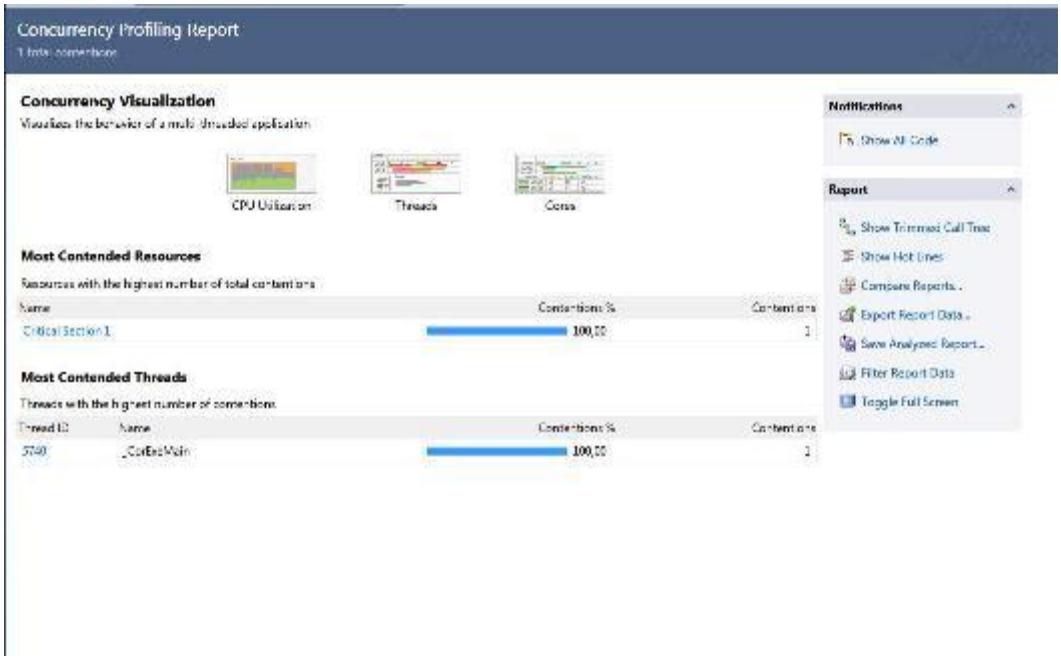

Рис. 7

11. Выберем из списка "Cores". Отобразиться график, который показывает дисбаланс нагрузки на логические ядра процессора:

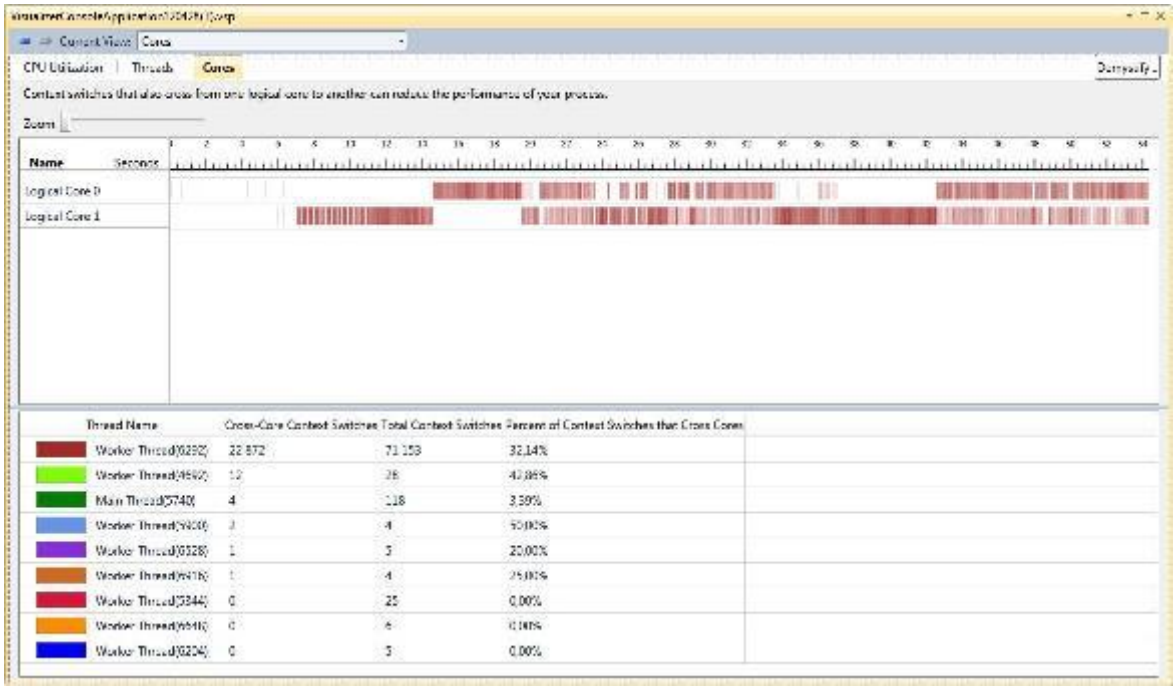

Рис. 8

12. Для того что бы сбалансировать нагрузку на ядра процессора модифицируем метод ParallelLoop() - заменим цикл foreach наParallel.ForEach:

```
static void ParallelLoop()
  {
      var numbers = Enumerable.Range(0, 1000);
      Parallel.ForEach(numbers, (number) =>
      {
```

```
            DoSomething();
        });
    }
```

13. Повторно запустим профилирование. На диаграмме видно, что теперь каждое ядро процессора имеет сбалансированную нагрузку, кроме того каждое ядро выполняет различные потоки:

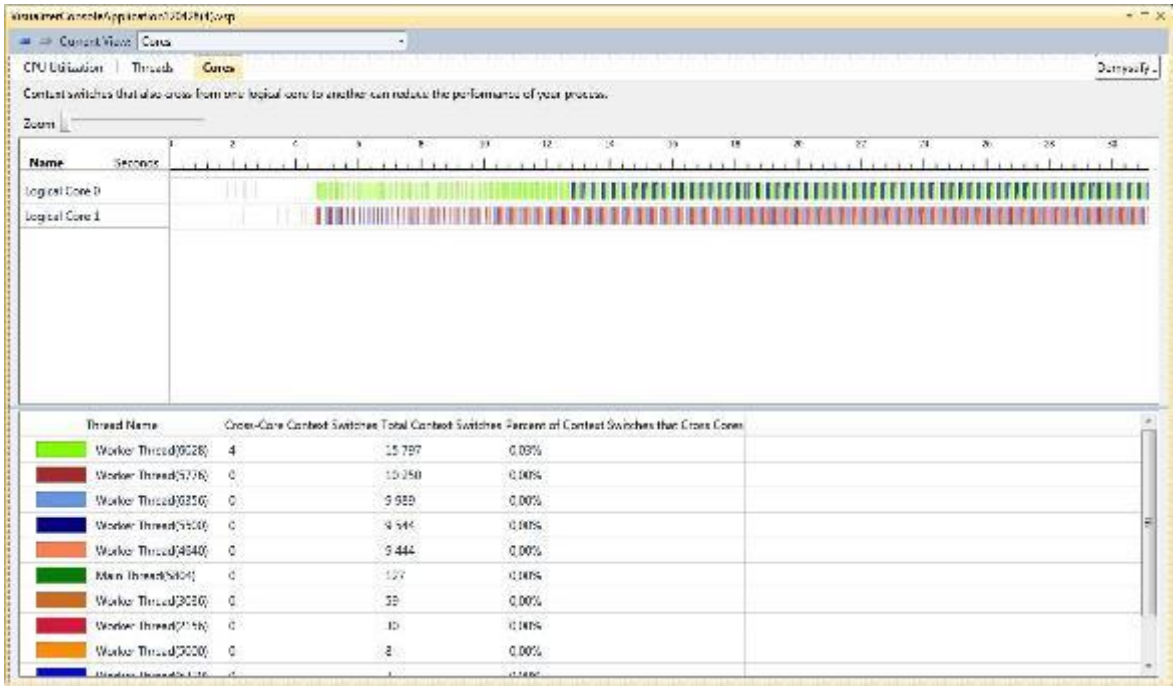

Рис. 9

14. Выберите вкладку "CPU Utilization". На данном графике отображается использование процессора с работающим приложением (в нашем случае приложение использует 54%):

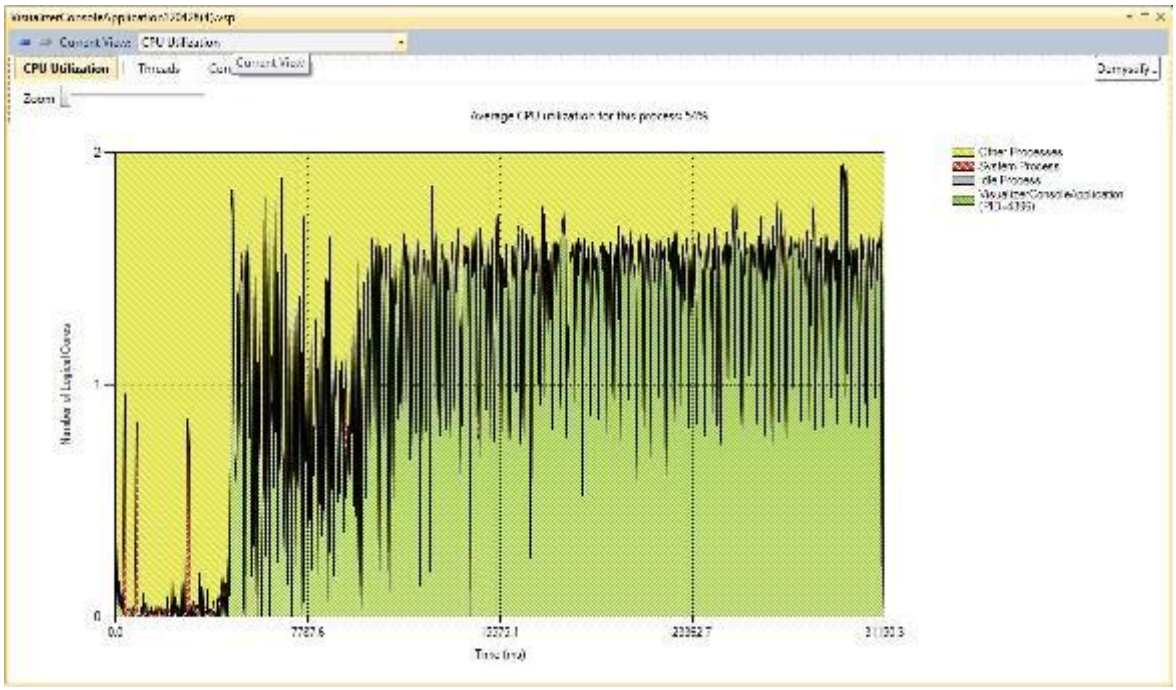

Рис. 10

15. Теперь, добавим в приложение два идентичных потока и один объект sync (который будет использоваться для синхронизации lock):

```
static void Main(string[] args)
        {
           new Thread(new ThreadStart(ParallelLoop)).Start();

           object sync = new object();
           new Thread(new ThreadStart(() =>
           {
              lock (sync)
            {
                 Thread.Sleep(2000);
              }

           })).Start();

           new Thread(new ThreadStart(() =>
           {
              lock (sync)
              {
                 Thread.Sleep(2000);
              }

           })).Start();

           Console.ReadLine();
        }
```

Примечание: ключевое слово lock не позволит одному потоку войти в раздел кода в тот момент, когда в нем находится другой поток.

16. Запустим повторно профилирование. После профилирования перейдем во вкладку "Threads":

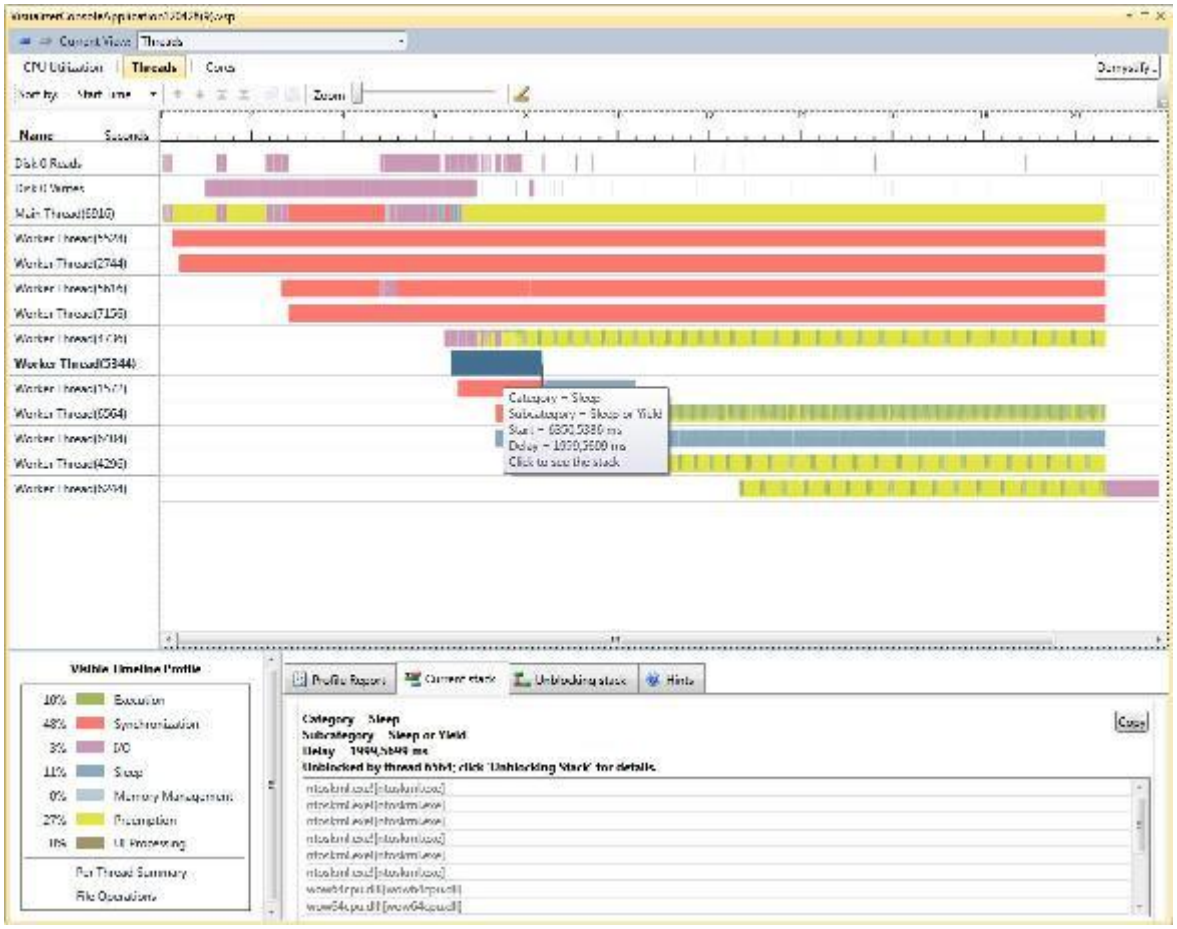

Рис. 11

На диаграмме можно видеть два зависимых потока, один из потоков запускается с сегмента синхронизации (Thread.Sleep(2000)).

Листинг кода программы:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
namespace VisualizerConsoleApplication
{
   class Program
   {
      static void DoSomething()
      {
         Thread.SpinWait(int.MaxValue / 10);
      }

      static void ParallelLoop()
      {
```

```
            var numbers = Enumerable.Range(0, 1000);
            Parallel.ForEach(numbers, (number) =>
            {
                DoSomething();
            });
        }
        static void Main(string[] args)
        {
            new Thread(new ThreadStart(ParallelLoop)).Start();
            object sync = new object();
            new Thread(new ThreadStart(() =>
            {
                lock (sync)
                {
                    Thread.Sleep(2000);
                }

            })).Start();

            new Thread(new ThreadStart(() =>
            {
                lock (sync)
                {
                    Thread.Sleep(2000);
                }

        })).Start();

            Console.ReadLine();

        }
    }
}
```

Вопросы ответить в тетради (прочитать файл Visualizer):

1. **Что такое Currency Visualizer?**
2. **Перечислить возможности Currency Visualizer.**
3. **Виды отчетов, которые можно получить с помощью Currency Visualizer.**
4. **Что содержит в себе представление потоков?**