УДК 004.432

ИНСТРУМЕНТЫ РАЗРАБОТКИ ПАРАЛЛЕЛЬНЫХПРОГРАММ НА ЯЗЫКЕ C#

Казачёнок М.С., Прихожий А.А.
Белорусский национальный технический университет, г. Минск

Начиная с версии .NET 4.0 в Microsoft ввели новый подход к разработке многопоточных приложений, предусматривающий применение библиотеки параллельного программирования, которая называется TPL (TaskParallelLibrary),основной функционал которой располагается в пространстве имен System.Threading.Tasks [1]. Эта библиотека совершенствует многопоточное программирование двумя основными способами:

- во-первых, упрощает создание и применение многих потоков.
- во-вторых, позволяет автоматически использовать несколько процессоров.

Проще говоря, TPL предоставляет возможности для автоматического масштабирования приложений с целью эффективного использования ряда доступных процессоров. В данной библиотеке есть два класса объектов, Taski Parallel, которые мы аналізіруем в этой работе.

Класс Taskпозволяет в значительной степени упростить написание параллельного кода, без необходимости работы непосредственно с потоками или пулом потоков. Класс Taskотличается от классаThreadтем, что он является абстракцией, представляющей асинхронную операцию, а в классе Threadинкапсулируется поток исполнения.

Для определения и запуска задачи можно использовать различные способы. Первый способ – создание объекта Task и вызов его метода Start, представленного на рисунке 1. В качестве параметра объект Task принимает делегат Action, которыйможет быть, например, лямбда-выражением, как в данном случае. То есть, при выполнении задачи, представленной на рисунке 1, будет выполнен метод, который вызывается в лямбда-выражении. А метод Start() собственно запускает задачу.

```
Task task = new Task (() => ParallelMethod ());
task.Start ();
```

Рисунок 1 –Первый способ создания Task

Второй способ заключается в использовании статического метода Task.Factory.StartNew(), представленного на рисунке 2. Этот метод также в качестве параметра принимает делегат Action, который указывает, какое действие будет выполняться. При этом метод сразу же запускает задачу. В качестве результата метод возвращает объект запущенной задачи.

```
Task task = Task.Factory.StartNew (() => ParallelMethod ());
```

Рисунок 2 –Второй способ создания Task

Третий способ определения и запуска задач предусматривает использование статического метода Task.Run() представленного на рисунке 3. Метод Task.Run() также в качестве параметра может принимать делегат Action, описывающий выполняемое действие и возвращающий объект Task.

```
Task task = Task.Run (() => ParallelMethod ());
```

Рисунок 3 –Третий способ создания Task

Класс Parallelпозволяет выполнять параллельное программирование на основе задач, не прибегая к управлению задачами явным образом. Этот класс поддерживает набор методов, которые позволяют выполнять итерации по коллекции данных в параллельном режиме. К таким методам относятся [2]:

- Parallel.Invoke – запускает массив делегатов параллельно;
- Parallel.ForEach – выполняет параллельный эквивалент цикла foreach;
- Parallel.For – выполняет параллельный эквивалент цикла for.

Все три метода блокируют управление до окончания выполнения всех действий. При возникновении необработанного исключения в каком-то из потоков, оставшиеся рабочие потоки прекращают выполнение и вызывают исключение – AggregationExeption.

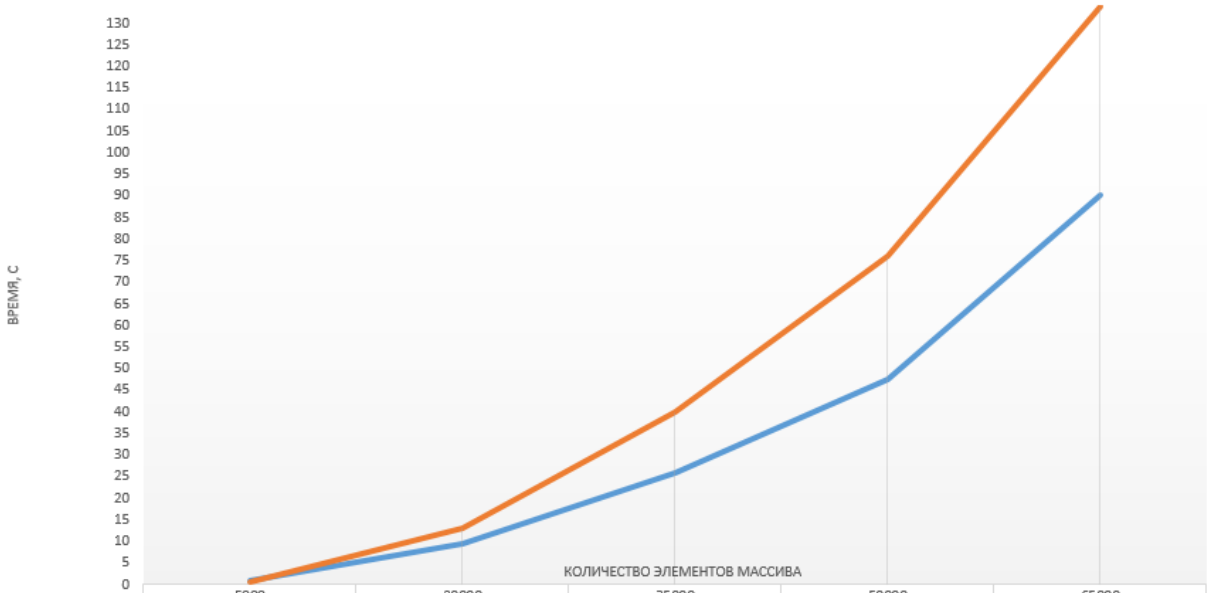
Пример использования метода Parallel.Forпредставлен на рисунке 4. Первый параметр – начальный индекс цикла, второй параметр – конечный индекс цикла, третий параметр передает делегат (Action), который вызывается один раз за итерацию.

```
Parallel.For (0, array.Length, i => array[i] = rand.Next (1, 101));
```

Рисунок 4 –Пример синтаксиса Parallel.For

Для сравнения времени выполнения параллельной и последовательной версий программы смоделирована следующая задача. Необходимо создать массив и заполнить его случайными значениями от 1 до 100, после это нужно отсортировать массив пузырьковым методом. Зависимость времени выполнения последовательной и параллельной программ представлена в виде графиков на рисунке 5. На горизонтальной оси указана длина массива, на вертикальной оси указано время выполнения. Рисунок 5 показывает, что при небольшой длине массива (приблизительно менее 7000 элементов) последовательный код работает быстрее, чем параллельный. При количестве элементов 7000 и более,

параллельный код начинает выигрывать у последовательного, причем чем больше элементов содержит массив, тем больше разница между временем параллельного выполнения и временем последовательного выполнения программы.



	5000	20000	35000	50000	65000
Parallel	0,799	9,469	25,949	47,453	90,026
Simple	0,685	13,089	39,8374	75,974	133,732
Simple/Parallel	0,86	1,38	1,54	1,6	1,49

Рисунок 5 – График выполнения последовательной и параллельной версий программы

На основе выполненного анализа, можно сделать вывод о том, что использование инструментов параллельной разработки .NET может как уменьшить время выполнения программы, так и увеличить его. Необходимо уметь распознавать, когда стоит распараллеливать программу, а когда нет.

Литература

1. Троелсен, Э., Джепикс, Ф. Язык программирования C# 7 и платформы .NET и .NET Core, 8-е изд.: Пер. с англ. — СПб.: ООО “Диалектика”, 2018 — 1328 с.

2. Албахари, Д., Албахари, Б. C# 7.0. Справочник. Полное описание языка.: Пер. с англ. — СпБ.: ООО “Альфа-книга”, 2018. — 1024 с.

3. Скит, Д. C# для профессионалов: тонкости программирования, 3-е изд.: Пер. с англ. — М.: ООО “И.Д. Вильямс”, 2014. – 608 с.

4. Голдштейн, С., Зурбалев, Д., Флатков, И., и др. Оптимизация приложений на платформе .NET. – Пер. с англ. Кисилев А.Н. – М. ДМК Пресс, 2014 – 524 с.