ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ НА C#. ФАКТЫ И ГИПОТЕЗЫ

В. А. Биллиг

Тверской государственный технический университет, факультет ИТ

Аннотация

Современные процессоры стали многоядерными. Процесс увеличения вычислительной мощности компьютеров за счет увеличения числа процессоров, числа ядер у каждого процессора, будет только прогрессировать. Изменение «железа» не может не сказываться на изменении «софта». Как следствие, параллельные вычисления становятся одним из главных направлений развития современного программирования.

При программировании на C# параллельные вычисления поддерживаются механизмом потоков, создаваемых операционной системой. В программах на C# можно создать поток – объект класса Thread и связать с ним определенный фрагмент кода. Кажется естественным, что при создании программного объекта класса Thread операционная система создает физический поток, который и будет выполнять код при запуске потока на выполнение. Так, например, происходит с файловыми объектами, - создание файлового объекта в программе приводит к созданию физического файла.

В работе показано, в каких ситуациях «естественная» семантика имеет место, а в каких ситуациях создание программного потока не приводит к созданию физического потока. Даются рекомендации по оптимизации уровня распараллеливания. Рассматриваются ситуации ограничения распараллеливания в рекурсивных методах.

Проведенные исследования позволили обнаружить ситуацию, когда взаимодействие двух важных механизмов – потоков и анонимных методов приводит к некорректной работе. Сконструированы примеры, демонстрирующие некорректную работу анонимных методов при распараллеливании.

Ключевые слова: C#, параллельные вычисления, многопоточное программирование, поток, рекурсия, анонимный метод

Annotation

Modern processors are multi-core processors. Increasing computing power by increasing the number of processors and the number of cores in each processor will only be progressing. Changing of hard leads to changing of soft. As a result, parallel computing becomes one of the main directions of modern programming.

When programming in C#, parallel computing is supported by the mechanism of threads created by the operating system. You can create a thread object of the Thread class and associate a certain piece of code with it. It seems natural that when you create a program object of class Thread, the operating system creates a physical

thread, which will execute the code. For example, it occurs with file objects - creating a file object in the program leads to the creation of a physical file.

In the present article, the situations in which the "natural" semantics takes place are shown, as well as the situations in which creation of program thread does not create the physical thread. We provide recommendations for optimizing the degree of parallelization in recursive methods and in methods without recursion.

The research made it possible to detect the situation when interaction of two important mechanisms - threads and anonymous methods - leads to incorrect work. The constructed examples show incorrect work of anonymous methods when paralleling.

Keywords: C#, parallel computing, multi-thread programming, thread, recursion, anonymous method.

Параллельные вычисления особенно необходимы, когда задача требует высокой производительности компьютера. Распараллеливание по данным – один из главных приемов параллельного программирования, позволяющий ускорить обработку больших данных. В конечном счете, проблема сводится к распараллеливанию циклов. В данной работе анализируются результаты экспериментальных исследований. Приводятся факты, выдвигаются гипотезы, даются рекомендации по организации распараллеливания по данным. О некоторых других проблемах параллельных вычислений при программировании на C# можно прочесть в [1].

Неоднозначная семантика. Случай, когда создание программного потока не приводит к созданию физического потока

Пусть в нашей программе есть цикл:

```
for(int i = 0; i < N; i++)
    Body(i);
```

Предположим, что итерации цикла – Body(i) независимы, так что цикл может быть распараллелен. Наиболее эффективно в этом случае в качестве инструмента распараллеливания использовать класс Parallel и заменить обычный оператор цикла на статический метод Parallel.For. Некоторым недостатком является то, что в этом случае мы не управляем уровнем распараллеливания, - не можем задать число создаваемых потоков, параллельно выполняющих итерации цикла. По этой или другим причинам для распараллеливания цикла зачастую используется класс Thread, предоставляющий программисту больше возможностей управления распараллеливанием.

Если мы хотим управлять числом создаваемых в программе потоков – объектов класса Thread, то исходный цикл разумно разбить на два цикла, введя дополнительный параметр – число групп, на которые разбивается исходный интервал изменения параметра цикла. Причина такого разбиения понятна. Физическое создание N потоков явно неразумно при больших значениях N, поскольку накладные расходы на создание и удаление потоков могут не только съесть весь выигрыш от распараллеливания, но и ухудшить результаты.

Семантически эквивалентная замена исходного цикла двумя циклами сложнее одиночного цикла:

```
int m = N/k_group;
int groups = (N % k_group == 0) ? k_group : k_group + 1;
int start, finish;
for( int k = 0; k < groups; k++)
{
        start = k*m;
        finish = (start + m < N)? start + m : N;
        for(int i = start; i < finish; i++)
                Body(i)
}
```

Сложность конструкции компенсируется тем, что появился управляемый параметр groups! Теперь можно распараллелить только внешний цикл, создав массив потоков по числу групп. Вот как может выглядеть параллельная версия этого цикла, использующая массив потоков:

```
int groups = (N % K_groups == 0)? K_groups : K_groups +1;
        Thread[] threads = new Thread[groups];
        for(int k = 0; k < groups; k++)
        {
           threads[k] = new Thread(Body);
           threads[k].Start(k);
        }
        for (int k = 0; k < groups; k++)
           threads[k].Join();
```

Метод Body, передаваемый конструктору каждого потока, выглядит так:

```
    void Body(object group)
    {
      int k = (int)group;
        int m = N / K_groups;
        int start = k * m;
      int gr = N % K_groups == 0 ? K_groups : K_groups + 1;
      int finish = (k != gr - 1) ? start + m - 1 : N - 1;
        BodyM(start, finish);
    }
```

Методу Body передается номер группы. Зная N и число групп, можно рассчитать начальный и конечный индексы группы, а затем вызвать метод BodyM, выполняющий итерации исходного цикла в заданном интервале.

Такова классическая схема распараллеливания цикла. Идея понятна. Исходный цикл разбивается на группы с равным числом итераций в группе за исключением, возможно, последней группы. Создается массив потоков по числу групп. Каждая группа итераций запускается в отдельном потоке. Когда все потоки заканчивают работу, завершает работу цикл.

Возникает естественный вопрос, каково оптимальное число групп? Как влияет выбор на время вычислений? Следует ли связывать это число с характеристиками компьютера, на котором будут производиться вычисления? Можно, например, предположить, что оптимальное значение параметра groups следует выбирать равным числу виртуальных процессоров компьютера. Так для моего компьютера с четырьмя ядрами, на каждом из которых одновременном можно запускать два потока, верно ли, что лучшие результаты достигаются, когда при распараллеливании в программе создаются восемь потоков – объектов класса Thread? Эксперименты опровергают это предположение.

Приведу результаты времени вычислений для задачи сортировки целочисленного массива, содержащего 200 000 элементов. Время работы последовательной версии Insert сортировки (сортировки вставкой) на таком массиве составило 64,9 секунды. Время работы параллельной версии в зависимости от числа групп (следовательно, от числа создаваемых программных потоков) приведено в таблице 1:

N (групп)	2	4	8	10	20	100	200	400	600	1000	2000	20000
T (секундах)	16,5	5,0	2,0	1,7	0,9	0,24	0,18	0,17	0,22	0,26	0,44	5,65

Таблица 1. Время сортировки массива в зависимости от уровня распараллеливания

Прежде чем перейти к анализу этой таблицы, несколько слов о применяемом алгоритме. В последовательном варианте применяется классический вариант сортировки вставкой. В параллельном варианте исходная задача разбивается на k подзадач – массив разбивается на k групп, каждая их которых сортируется последовательным алгоритмом вставки. После этого для получения окончательно отсортированного массива проводится слияние k отсортированных групп. При слиянии используется параллельный пирамидальный алгоритм. Слияние каждой пары массивов ведется в отдельном потоке.

Как показывают результаты вычислений, так реализованный параллельный алгоритм позволил сократить время решения при правильно выбранном уровне распараллеливания более чем в триста раз! Такое прекрасное ускорение является «странным» фактом и требует отдельного объяснения. Дело в том, что алгоритм сортировки вставкой имеет сложность $O(N^2)$. При сокращении размера массива в группе в k раз, время сортировки одной группы сокращается в k^2 раз. Конечно, из-за ограниченного числа физических потоков лишь немногие группы сортируются параллельно. Кроме того, приходится тратить время на слияние k отсортированных групп в единый

отсортированный массив и, возможно, на создание k потоков. Тем не менее, как показывают результаты эксперимента, при обработке массива из двухсот тысяч элементов алгоритмом с квадратичной сложностью можно получить значительный выигрыш при распараллеливании алгоритма.

Выигрыш во времени в триста раз достигается благодаря совместному действию двух факторов – разбиения задачи на подзадачи меньшей размерности и возможности решения подзадач в отдельных потоках. Основной эффект достигается за счет разбиения задачи на подзадачи. Эффект от распараллеливания зависит от числа ядер компьютера. Приведу результаты эксперимента, где разбиение на подзадачи не сопровождается созданием потоков для подзадач.

N (групп)	2	4	8	10	20	100	200	400	600	1000	2000	20000
T (секундах) С потоками	20,9	6,7	2,6	2,2	1,1	0,29	0,21	0,18	0,21	0,30	0,55	6,1
T (секундах) Без потоков	41,9	20,9	10,5	8,4	4,2	0,88	0,48	0,28	0,22	0,16	0,11	0,19

Таблица 2. Время сортировки массива в зависимости от числа групп

При распараллеливании алгоритма Insert усложнение, связанное с введением групп, вполне естественно. Без распараллеливания такое усложнение также дает эффект, но кажется, что в такой ситуации лучше использовать эффективные алгоритмы со сложностью NlogN

Более интересно рассмотреть два других «странных» факта, связанных с распараллеливанием:

Оптимальное значение N достигается в точке N = 400, что далеко от предполагаемого значения N = 8 (число виртуальных процессоров компьютера).

При значениях N = 20000, что превосходит всякое «разумное» задание числа потоков для компьютера с четырьмя ядрами, время работы возрастает незначительно. Это означает, что накладные расходы, связанные с созданием физических потоков, практически не появляются.

Гипотеза и рекомендации

Описать точную семантику связи между программными объектами класса Thread и физическими объектами – потоками операционной системы по

результатам наблюдений затруднительно. Можно лишь высказать некоторую гипотезу и рекомендации, согласующиеся с результатами наблюдений:

При работе с массивом программных потоков создание конструктором класса Thread элемента массива - объекта класса Thread не означает создания физического объекта – потока операционной системы.

Программный поток проецируется на существующий физический поток из пула потоков операционной системы.

Прямые наблюдения за числом потоков в диспетчере задач подтверждают эту гипотезу.

Для эффективного распараллеливания число программных потоков следует задавать намного больше числа виртуальных процессоров компьютера, на котором производятся вычисления. Если обозначить число виртуальных процессоров компьютера через V, то оптимальное число программных потоков равно k*V, где коэффициент k находится в диапазоне [10, 25].

Число создаваемых программных потоков не следует задавать больше чем k * V.

Неоднозначная семантика. Случай, когда создание программного потока приводит к созданию физического потока

Мы рассмотрели ситуацию, когда в программе создается большой массив потоков. Но разрастание числа потоков не обязательно связано с заданием массива потоков. Так, для рассматриваемой нами задачи сортировки массивов применяются методы сортировки со сложностью O(N * log N), которые, как правило, являются рекурсивными. В классическом последовательном варианте сортировки слиянием исходный массив разбивается на две половинки, каждая из которых сортируется независимо и рекурсивно. Отсортированные половинки сливаются в единый отсортированный массив. Алгоритм допускает естественное распараллеливание. Для сортировки половинок создаются два потока, каждая половинка сортируется в своем потоке. Из-за рекурсии число потоков растет экспоненциально с ростом уровня рекурсии. Экспоненциальный рост следует ограничить. Для ограничения возрастания числа потоков можно применить следующий подход, - новые потоки не создавать, когда длина сортируемого массива становится меньше заданной величины Limit.

Рассмотрим задачу сортировки целочисленного массива, содержащего 10000000 (десять миллионов) элементов. В последовательном варианте применялся классический рекурсивный алгоритм сортировки слиянием. Время работы последовательной версии на таком массиве составило 4,2 секунды. Заметьте, на больших массивах последовательный алгоритм сортировки слиянием существенно выигрывает не только у последовательного, но и у параллельного алгоритма сортировки вставкой.

В параллельном варианте сортировки слиянием рост числа потоков ограничивался, как было сказано, введением константы Limit. Приведу результаты эксперимента:

Limit	$4*10^6$	$2*10^6$	10^6	10^5	10^4	10^3
T (секундах)	1,45	1,24	1,36	1,66	4,67	Выход за пределы памяти

Таблица 3. Время сортировки массива методом слияния в зависимости от уровня распараллеливания

Гипотеза и рекомендации

Создание потоков в рекурсивном методе имеет ту особенность, что новые потоки создаются внутри потока, еще не завершившего свою работу. В такой ситуации семантика работы с потоками меняется. Создание нового программного потока внутри потока, не завершившего работу, приводит к созданию нового физического потока.

Прямые наблюдения за числом потоков в диспетчере задач подтверждают эту гипотезу.

Программные потоки следует создавать только на верхних уровнях рекурсии. Для нашего алгоритма оптимальный уровень рекурсии p, на котором следует прекратить создание потоков, можно вычислить из соотношения $2^p = V$, где V – число виртуальных процессоров.

Если не ограничить разумным способом создание потоков, то можно не только проиграть во времени вычислений в сравнении с последовательным вариантом, но и не получить решения из-за возникновения исключительной ситуации – выхода за пределы допустимой памяти при создании новых потоков. В нашем эксперименте такая ситуация возникает при значении Limit, равном 1000.

При оптимальном задании уровня распараллеливания параллельный алгоритм выигрывает у последовательного алгоритма, - время сокращается примерно в три раза. Не столь эффективное распараллеливание, как в случае алгоритма сортировки методом вставки, объясняется двумя факторами:

Сортировка слиянием имеет сложность, близкую к линейной, поэтому здесь не достигается тот выигрыш на подзадачах, характерный для алгоритма со сложностью O(N2).

Добавляются накладные расходы, связанные с созданием физических потоков.

Стоит заметить, что параллельный алгоритм сортировки слиянием выигрывает у последовательного алгоритма только для массивов большого размера. Например, при сортировке массива в 10 000 элементов применять параллельный алгоритм слияния не рекомендуется, поскольку он проиграет последовательному алгоритму по той причине, что выигрыш за счет распараллеливания вычислений будет меньше проигрыша, связанного с накладными расходами на организацию физических потоков.

Экспериментальная проверка гипотез и рекомендаций требует проведения более широких исследований. В качестве первого шага рассмотрим два других алгоритма сортировки: алгоритм пузырьковой сортировки со сложностью O(N2) и алгоритм быстрой сортировки со средней сложностью O(N * log N).

N групп	2	4	8	10	20	100	200	400	1000	2000	20000
T(sec) Insert	16,2	5,0	2,0	1,8	0,85	0,25	0,18	0,17	0,25	0,45	5,1
T(sec) Bubble	29,9	7,8	2,9	2,5	1,3	0,3	0,19	0,17	0,25	0,47	8,9

Таблица 4. Сравнение пузырьковой сортировки и сортировки вставкой

Для обоих методов оптимальное значение достигается в одной и той же точке (число групп N = 400). Оба графика имеют одну и ту же тенденцию, подтверждающую ранее сделанные выводы. Можно лишь заметить, что распараллеливание для пузырьковой сортировки дает больший эффект, чем для сортировки вставкой. Так для последовательных вариантов «пузырек» проигрывает «вставке» в три раза, а при оптимальном распараллеливании время сортировки обоими методами уравнивается.

Таким образом, мы получили некое подтверждение того, что наша гипотеза и рекомендации имеют место не только для одного конкретного алгоритма, а имеют более значимый характер.

Справедливы ли наши утверждения для рекурсивных методов, когда потоки создаются внутри работающего потока? Давайте рассмотрим еще один, наиболее употребляемый рекурсивный метод быстрой сортировки Хоара. Вот результаты экспериментов, позволяющие сравнить два параллельных варианта сортировки массива из 10 миллионов элементов.

Limit	$4 * 10^6$	$2 * 10^6$	10^6	10^5	10^4	10^3
T(sec) Merge	1,49	1,23	1,3	1,7	4,6	Memory out
T(sec) Quick	1,0	0,79	0,70	0,73	1,15	Memory out

Таблица 5. Сравнение сортировки слиянием и быстрой сортировки

Как всегда, сортировка Хоара и в параллельном варианте выигрывает у всех своих конкурентов. Но тенденции зависимости времени сортировки от уровня распараллеливания для обоих методов одинаковы и оптимум достигается в одной и той же точке, когда число потоков равно 8, что в данном случае совпадает с числом виртуальных процессоров компьютера.

Некорректная работа анонимных методов в потоках

Анонимные методы – мощный и полезный механизм. Это константы в мире методов. Потоку в момент его создания необходимо передать метод, выполняемый потоком. Крайне удобно во многих ситуациях задавать именно константный, анонимный метод. Помимо прочего, это позволяет преодолевать синтаксические ограничения, накладываемые на метод, передаваемый потоку. Такой метод, как известно, должен быть процедурой без параметров или иметь один параметр универсального типа Object. При использовании анонимного метода эти ограничения легко обходятся.

К сожалению, использование анонимного метода при работе с потоками может приводить к некорректной работе. Хуже того, некорректность может не приводить к появлению исключительных ситуаций, результаты вычислений будут не верны, но работа приложения будет продолжаться. Хуже того, некорректность может не всегда проявляться, в простых ситуациях, характерных для тестирования, все может работать нормально, а с ростом сложности вычислений некорректность может проявляться.

Пример некорректного «анонима» при работе с объектами класса Thread

Для подтверждения факта некорректной работы анонимного метода в потоках сконструируем достаточно простой пример. Создается массив из N потоков. При создании потока с номером k ему передается анонимный метод, который добавляет целое число k в элемент массива с номером k. При корректной работе сумма элементов этого массива равна N * (N -1) /2.

Приведу текст этого метода:

```
/// <summary>
/// Потоки с анонимным методом
```

```
/// Пример некорректной работы
/// </summary>
/// <returns>true при корректной работе</returns>
public bool Test_ThreadsWithAnonym()
{
   threads = new Thread[N];
   for (int i = 0; i < N; i++)
      mas[i] = 0;
   for (int k = 0; k < N; k++)
   {
      threads[k] = new Thread(() =>
      {
         mas[k] += k;

      });
      threads[k].Start();
   }
   for (int i = 0; i < N; i++)
      threads[i].Join();
   double S = 0;
   for (int i = 0; i < N; i++)
      S += mas[i];
   return S - N * (N - 1) / 2.0 < 1e-7;
}
```

При запуске этого примера проявляются ошибки двух типов:

Некоторые элементы массива не заполняются. Это означает, что некоторые потоки threads[i] не создаются. С другой стороны, некоторые потоки threads[i]создаются дважды (как правило, соседние с пропущенными потоками), так что элементы массива получают удвоенное значение. Такой эффект, например, наблюдался при задании N равном 10000. При малых значениях N, как правило, элементы заполняются корректно.

К счастью (или к несчастью) возникает еще и исключительная ситуация – индекс выходит за допустимые пределы. Это означает, что некорректно работает условие окончания цикла.

Если отказаться от анонимного метода и перейти к именованному методу, то все работает корректно. Вот текст, демонстрирующий работу с именованным методом:

```
public bool Test_ThreadsWithNamedMethods()
{
   threads = new Thread[N];
   for (int i = 0; i < N; i++)
      mas[i] = 0;
   for (int k = 0; k < N; k++)
```

```
        {
            threads[k] = new Thread(Cort);
            threads[k].Start(k);
        }
        for (int i = 0; i < N; i++)
            threads[i].Join();
        double S = 0;
        for (int i = 0; i < N; i++)
            S += mas[i];
        return S - N * (N - 1) / 2.0 < 1e-7;
    }
    void Cort(object par)
    {
        int k = (int)par;
        mas[k] += k;
    }
```

А что, если для распараллеливания использовать прекрасный метод Parallel.For:

```
    /// <summary>
    /// Parallel.For с анонимным методом
    /// Работает корректно в тестируемых примерах
    /// </summary>
    /// <returns>true при корректной работе</returns>
    public bool Test_ParallelWithAnonym()
    {
        for (int i = 0; i < N; i++)
            mas[i] = 0;
        Parallel.For(0, N, (k) =>
        { mas[k] += k; });
        double S = 0;
        for (int i = 0; i < N; i++)
            S += mas[i];
        return S - N * (N - 1) / 2.0 < 1e-7;
    }
```

В данном примере для всех протестированных значений N метод Parallel.For в сочетании с анонимным методом корректно работал. Может быть, такое сочетание всегда корректно работает? Возможно, что так! По крайней мере, мне не удалось сконструировать пример, где взаимодействие метода Parallel.For в сочетании с анонимным методом приводило к некорректной работе. Пример, который я пытался сконструировать, работал некорректно, но из-за ошибки в конструкции, приводящей к гонке данных.

Итоги

1. При обработке «больших данных» целесообразно использовать параллельные вычисления. Данные, зачастую, можно разбить на группы, обрабатывать их независимо и параллельно. Затем результаты обработки групп объединяются в общее решение. Помимо того, что обработка групп может вестись параллельно, разбиение задачи на подзадачи меньшей размерности может приводить к дополнительному эффекту, ускоряющему решение общей задачи.
2. Если метод обработки не предполагает рекурсию, то для распараллеливания можно использовать наряду с классом Parallel и массив потоков – объектов класса Thread. Как показывают наши исследования создание элементов этого массива – программных потоков не приводит к созданию физических потоков. Как следствие, оптимальное для распараллеливания число групп следует выбирать намного больше числа виртуальных процессоров компьютера.
3. Для рекурсивных методов обработки при распараллеливании следует создавать потоки только на верхних уровнях рекурсии. Создание новых программных потоков внутри потока, не завершившего обработку, приводит к созданию физических потоков, что требует существенных накладных расходов, как по времени, так и по памяти.
4. При распараллеливании весьма удобным механизмом является механизм анонимных методов. К сожалению, как показывают проведенные исследования, взаимодействие этих двух важных механизмов может приводить к некорректной работе, по крайней мере, при непосредственной работе с потоками.

ЛИТЕРАТУРА

1. В.А. Биллиг, «Параллельные вычисления и многопоточное программирование», Москва – Тверь, ИНТУИТ – ЗАО НИИ ЦПС, 2013 г.