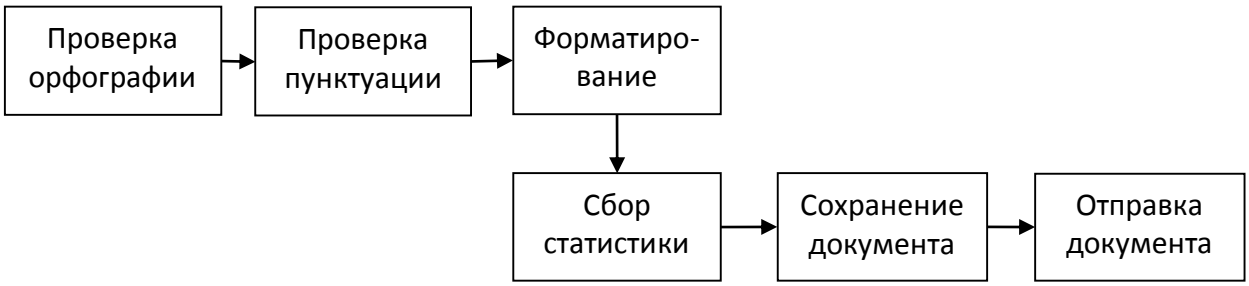
2. Проблемы разработки параллельных приложений

Основными этапами разработки параллельных приложений являются: декомпозиция, выявление информационных зависимостей между подзадачами, масштабирование подзадач и балансировка нагрузки для каждого процессора.

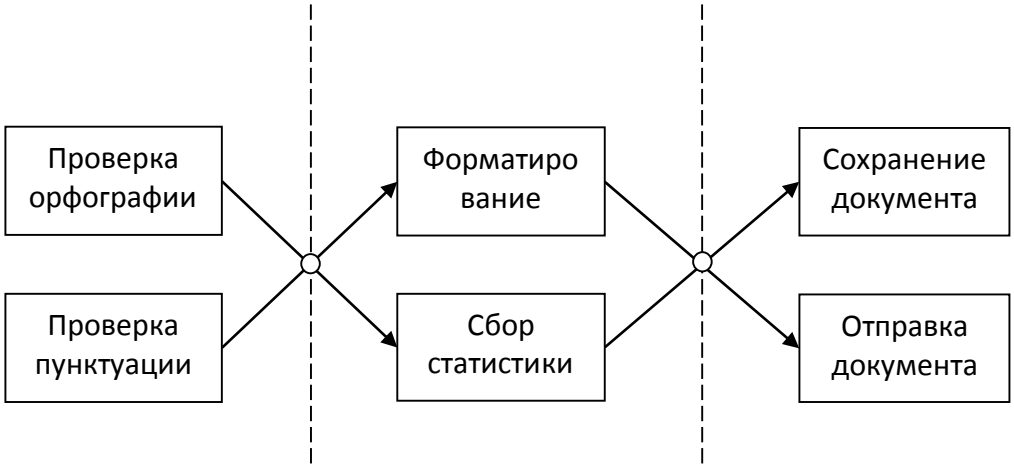
Декомпозиция

Под декомпозицией понимается разбиение задачи на относительно независимые части (подзадачи). Декомпозиция задачи может быть проведена несколькими способами: по заданиям, по данным, по информационным потокам.

Декомпозиция по заданиям (функциональная декомпозиция) предполагает присвоение разным потокам разных функций. Например, приложение выполняющее правку документа включает следующие функции: проверка орфографии `CheckSpelling`, проверка пунктуации `CheckPuncto`, форматирование текста в соответствие с выбранными стилями Format, подсчет статистики по документу `CalcStat`, сохранение изменений в файле `SaveChanges` и отправка документа по электронной почте `SendDoc`.



Функциональная декомпозиция разбивает работу приложения на подзадачи таким образом, чтобы каждая подзадача была связана с отдельной функцией. Но не все операции могут выполняться параллельно. Например, сохранение документа и отправка документа выполняются только после завершения всех предыдущих этапов. Форматирование и сбор статистики могут выполняться параллельно, но только после завершения проверки орфографии и пунктуации.



Декомпозиция по информационным потокам выделяет подзадачи, работающие с одним типом данных. В рассматриваемом примере могут быть выделены следующие подзадачи:

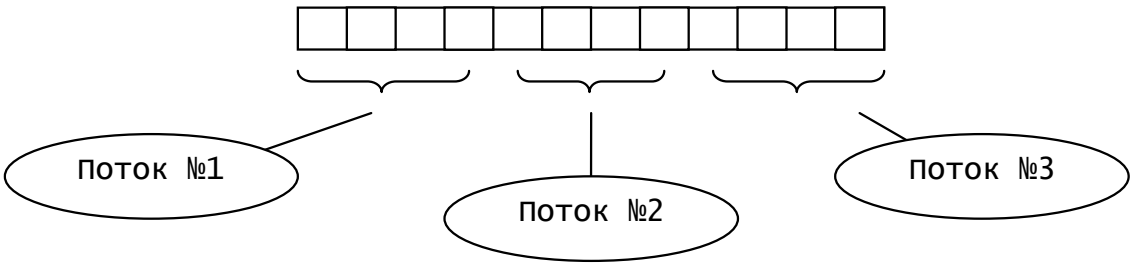
1) Работа с черновым документом (орфография и пунктуация):
2) Работа с исправленным документом (форматирование и сбор статистики);
3) Работа с готовым документом (сохранение и отправка);

При декомпозиции по данным каждая подзадача работает со своим фрагментом данных, выполняя весть перечень действий. В рассматриваемом примере декомпозиция по данным может применяться к задачам, допускающим работу с фрагментом документа. Таким образом, функции CheckSpelling, CheckPuncto, CalcStat, Format объединяются в одну подзадачу, но создается несколько экземпляров этой подзадачи, которые параллельно работают с разными фрагментами документа. Функции SaveChanges и SendDoc составляют отдельные подзадачи, так как не могут работать с частью документа.

Декомпозиция по данным

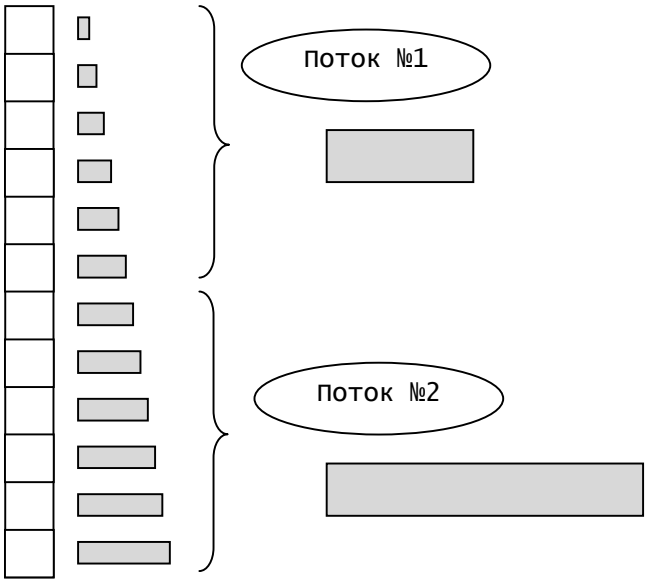
При декомпозиции по данным каждая подзадача выполняет одни и те же действия, но с разными данными. Во многих задачах, параллельных по данным, существует несколько способов разделения данных. Например, задача матричного умножения допускает разделение по строчкам – каждый поток обрабатывает одну или несколько строчек матрицы, по столбцам – каждый поток обрабатывает один или несколько столбцов, а также по блокам заданного размера.

Два основных принципа разделения данных между подзадачами – статический и динамический. При статической декомпозиции фрагменты данных назначаются потокам до начала обработки и, как правило, содержат одинаковое число элементов для каждого потока. Например, разделение массива элементов может осуществляться по равным диапазонам индекса между потоками (`range partition`).



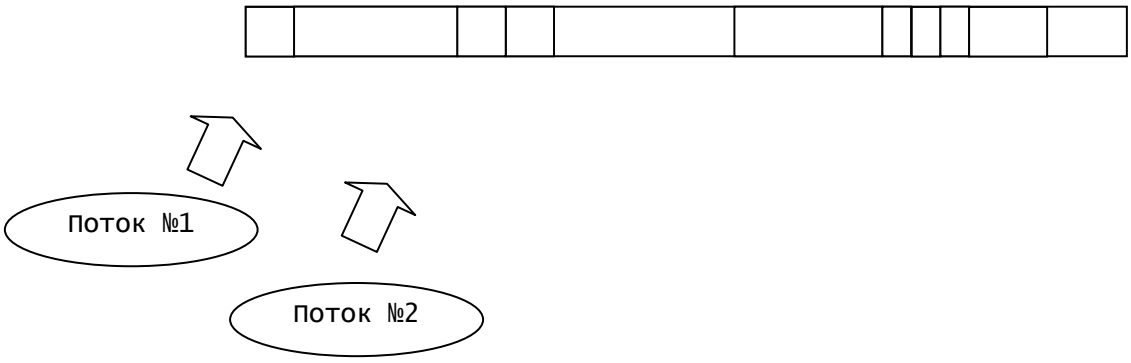
Основным достоинством статического разделения является независимость работы потоков (нет проблемы гонки данных) и, как следствие, нет необходимости в средствах синхронизации для доступа к элементам. Эффективность статической декомпозиции снижается при разной вычислительной сложности обработки элементов данных.

Например, вычислительная нагрузка обработки i-элемента может зависеть от индекса элемента.



Разделение по диапазону приводит к несбалансированности нагрузки разных потоков. Несбалансированность нагрузки снижает эффективность распараллеливания. В некоторых случаях декомпозиция может быть улучшена и при статическом разбиении, когда заранее известно какие элементы обладают большей вычислительной трудоемкостью, а какие меньшей. Например, в случае зависимости вычислительной трудоемкости от индекса элемента, применение круговой декомпозиции выравнивает загруженность потоков. Первый поток обрабатывает все четные элементы, второй поток обрабатывает все нечетные.

В общем случае, когда вычислительная сложность обработки элементов заранее не известна, сбалансированность загрузки потоков обеспечивает динамическая декомпозиция.



При динамической декомпозиции каждый поток, участвующий в обработке, обращается за блоком данных (порцией). После обработки блока данных поток обращается за следующей порцией. Динамическая декомпозиция требует синхронизации доступа потоков к структуре данных. Размер блока определяет частоту обращений потоков к структуре. Некоторые алгоритмы динамической декомпозиции увеличивают размер блока в процессе обработки. Если поток быстро обрабатывает элементы, то размер блока для него увеличивается.

Масштабирование подзадач

Свойство масштабируемости заключается в эффективном использовании всех имеющихся вычислительных ресурсов. Параллельное приложение должно быть готово к тому, что завтра оно будет запускаться на системе с большими вычислительными возможностями. Свойство масштабируемости приложения тесно связано с выбранным алгоритмом решения задачи. Один алгоритм очень хорошо распараллеливается, но только на две подзадачи, другой алгоритм позволяет выделить произвольное число подзадач.

Обязательным условием масштабируемости приложения является возможность параметризации алгоритма в зависимости от числа процессоров в системе и в зависимости от текущей загруженности вычислительной системы. Такая параметризация позволяет изменять число выделяемых подзадач при конкретных условиях выполнения алгоритма. Современные платформы параллельного программирования предоставляют средства для автоматической балансировки нагрузки (пул потоков).

Проблема гонки данных

Потоки одного процесса разделяют единое адресное пространство, что упрощает взаимодействие подзадач (потоков), но требует обеспечения согласованности доступа к общим структурам данных.

Проблема гонки данных возникает при следующих условиях:

1) Несколько потоков работают с разделяемым ресурсом.

2) Конечный результат зависит от очередности выполнения командных последовательностей разных потоков.

Для иллюстрации проблемы гонки данных рассмотрим следующий фрагмент. Два потока выводят на экран значение общей переменной Msg.

```
// Код потока №1
(1)  Msg = "I'm thread one";
(2)  Console.WriteLine("Thread #1: " + Msg);

// Код потока №2
(3)  Msg = "I'm thread two";
(4)  Console.WriteLine("Thread #2: " + Msg);
```

Переменная Msg является общей – изменение переменной в одном потоке будут видны в другом потоке. При параллельной работе потоков вывод определяется конкретной последовательностью выполнения операторов.

Если операторы первого потока выполняются до операторов второго потока, т.е. при последовательности (1) – (2) – (3) – (4), то мы получаем:

```
Thread #1: I'm thread one
Thread #2: I'm thread two
```

Если же в выполнение операторов одного потока вмешаются операторы другого потока, например, при последовательности (1) – (3) – (2) – (4), то получим следующее:

```
Thread #1: I'm thread two
Thread #2: I'm thread two
```

Проблема гонки данных возникает не только при выполнении нескольких операторов, но и при выполнении одного оператора. Рассмотрим следующий случай. Оба потока выполняют один оператор над общей переменной x типа int:

```
x = x + 5
```

Данный оператор предполагает выполнение следующих действий:

```
загрузить значение операндов в регистры процессора
осуществить суммирование
записать результат по адресу переменной x
```

При выполнении командных последовательностей одного потока на одном исполнительном устройстве в работу может вмешаться другой поток. Конечный результат зависит от очередности выполнения командных последовательностей. Если потоки осуществляют суммирование последовательно, то получаем конечный результат: 10. Если потоки осуществляют суммирование одновременно, то раннее изменение будет затерто поздним:

Действия	Поток №1	Поток №2
Загрузка операндов	0	0
Вычисление	5	5
Запись результатов	5	
		5
Значение переменной	5	5

Еще одной «ловушкой» в многопоточной обработке является работа с динамическими структурами данных (списки, словари). Добавление и удаление элементов в динамические структуры данных осуществляется с помощью одного метода:

```
list.Add("New element");
dic.RemoveKey("keyOne");
```

Реализация методов включает несколько действий. Например, при добавлении элемента в список необходимо записать элемент по текущему индексу (указателю) и сдвинуть индекс на следующую позицию. Для корректного осуществления добавления элемента одним потоком необходимо, чтобы другие потоки дождались завершения манипуляций со списком.

```
// Добавление элемента в массив по текущему индексу
data[current_index] = new_value;
current_index++;
```

Для решения проблемы гонки данных необходимо обеспечить взаимно исключительный доступ к тем командным последовательностям, в которых осуществляется работа с разделяемым ресурсом. Взаимная исключительность означает, что в каждый момент времени с ресурсом работает только один поток, другие потоки блокируются в ожидании завершения работы первого потока. Фрагмент кода, к которому должен быть обеспечен взаимно исключительный доступ, называется критической секцией.

Проблема гонки данных не всегда возникает при работе с общей переменной. Например, в следующем фрагменте два потока перед завершением осуществляют изменение разделяемой переменной, а третий поток читает это значение.

```
// Общая переменная
bool b = false;
//Поток №1
void f1()
```

```
{
     DoSomeWork1();
     b = true;
}

//Поток №2
void f2()
{
     DoSomeWork2();
     b = true;
}

//Поток №3
void f3()
{
     while(!b) ;
     DoSomeWork3();
}
```

В этом примере третий поток в цикле «ожидает» завершения хотя бы одного потока. Проблемы гонки данных не возникает, несмотря на работу трех потоков с общей переменной. Порядок выполнения потоков не влияет на конечный результат. Изменения, вносимые потоками, не противоречат друг другу.

Проблемы синхронизации

Решение проблемы гонки данных требует применения средств синхронизации, позволяющих обеспечить взаимно-исключительный доступ к критической секции. Применение синхронизации гарантирует получение корректных результатов, но снижает быстродействие приложения. Чем больше размер критических секций в приложении, тем больше доля последовательного выполнения и ниже эффективность от распараллеливания. Для повышения быстродействия размер критической секции должен быть предельно минимальным – только те операторы, последовательность которых не должна прерываться другим потоком.

В следующем фрагменте каждый поток сохраняет в разделяемом массиве data данные из файла. Для обеспечения согласованного доступа к разделяемому ресурсу используются средства синхронизации.

```
// Поток №1
<Вход в критическую секцию>
     StreamReader sr = File.OpenText("file" + ThreadNum);
     newValue = GetValue(sr);
     data[cur_index] = newValue;
     cur_index++;
     sr.Close();
<Выход из критической секции>
```

Размер критической секции в этом фрагменте не оправдано большой. Действия по подготовке данных для сохранения (открытие файла, уникального для каждого потока; поиск и чтение необходимой информации) могут быть вынесены за критическую секцию:

```
 // Поток №1
StreamReader sr = File.OpenText("file" + ThreadNum);
newValue = GetValue(sr);
<Вход в критическую секцию>
     data[cur_index] = newValue;
     cur_index++;
<Выход из критической секции>
sr.Close();
```

Современные платформы для параллельного программирования, в том числе и среда Framework .NET, предлагают широкий выбор средств синхронизации. В каждой задаче применение того или иного инструмента синхронизации будет более эффективным. Например, для многопоточного увеличения разделяемого счетчика могут использоваться средства организации взаимно-исключительного доступа (объекты Monitor, Mutex), сигнальные сообщения (AutoResetEvent, ManualResetEventSlim), двоичные семафоры (Semaphore). Но максимально эффективным будет использование неблокирующих атомарных операторов (Interlocked.Increment).

Для работы с динамическими списками можно использовать как обычные коллекции с теми или иными средствами синхронизации, так и конкурентные коллекции с встроенной неблокирующей синхронизацией.

Проблемы кэшируемой памяти

Наличие кэшируемой памяти увеличивает быстродействие обработки данных, но усложняет работу системы при многопоточной обработке. Неоптимальная работа с кэшируемой памятью может сильно снизить эффективность параллельной обработки.

Кэш-память каждого процессора (ядра процессора) наполняется данными, необходимыми для работы потока, выполняющегося на этом процессоре. Если потоки работают с общими данными, то на аппаратном уровне должна обеспечиваться согласованность содержимого кэш-памяти. Изменение общей переменной в одном потоке, сигнализирует о недействительности значения переменной, загруженной в кэш-память другого процессора. При этом необходимо сохранить значение переменной в оперативной памяти и обновить кэш-память других процессоров. Большая интенсивность изменений общих переменных, которые используются в нескольких потоках, приводит к большому числу ошибок кэш-памяти (кэш-промахи) и увеличению накладных расходов, связанных с обновлением кэш-памяти.

Распространенной проблемой кэш-памяти является так называемое ложное разделение данных (false sharing). Проблема связана с тем, что потоки работают с разными переменными, которые в оперативной памяти расположены физически близко. Дело в

том, что в кэш-память загружается не конкретная переменная, а блок памяти (строка кэша), содержащая необходимую переменную. Размер строки кэша может составлять 64, 128, 512 байт. Если в одной строке кэша расположены несколько переменных, используемых в разных потоках, то в кэш-память каждого процессора будет загружена одна и та же строка. При изменении в одном потоке своей переменной, содержимое кэш-памяти других процессоров считается недействительным и требует обновления.

В качестве иллюстрации проблемы ложного разделения рассмотрим следующий фрагмент. В программе объявлена структура, содержащая несколько полей.

```
struct data
{
     int x;
     int y;
}
```

Первый поток работает только с полем x, второй поток работает только с полем y. Таким образом, разделения данных и проблемы гонки данных между потоками нет. Но последовательное расположение в памяти структуры data, приводит к тому, что в кэш-память одного и другого процессора загружается строка размером 64 байт, содержащая значения и поля x (4 байта), и поля y (4 байта). При изменении поля в одном потоке происходит обновление строки кэша в другом потоке.

```
// Поток №1
for(int i=0; i<N; i++)
     data1.x++;

// Поток №2
for(int i=0; i<N; i++)
     data1.y++;
```

Чтобы избежать последовательного расположения полей x и y в памяти, можно использовать дополнительные промежуточные поля.

Другой подход заключается в явном выравнивании полей в памяти с помощью атрибута `FieldOffsetAttribute`, который определен в пространстве `System.Runtime.InteropServices`:

```
// Явное выравнивание в памяти
[StructLayout(LayoutKind.Explicit)]
struct data
{
     [FieldOffset(0)] public int x;
     [FieldOffset(64)] public int y;
}
```

При достаточно большом значении N, разница в быстродействии кода с разделением кэша и без разделения может достигать 1.5 – 2 раз.

Все же самым эффективным решением при независимой обработке полей структуры будет применение локальных переменных внутри каждого потока. Разница в быстродействии может достигать нескольких десятков.

Модели параллельных приложений

Существуют следующие распространенные модели параллельных приложений:

- модель делегирования («управляющий-рабочий»);
- сеть с равноправными узлами;
- конвейер;
- модель «производитель-потребитель»

Каждая модель характеризуется собственной декомпозицией работ, которая определяет, кто отвечает за порождение подзадач и при каких условиях они создаются, какие информационные зависимости между подзадачами существуют.

Модель	Описание
Модель делегирования	Центральный поток («управляющий») создает «рабочие» потоки и назначает каждому из них задачу. Управляющий поток ожидает завершения работы потоков, собирает результаты.
Модель с равноправными узлами	Все потоки имеют одинаковый рабочий статус.
Конвейер	Конвейерный подход применяется для поэтапной обработки потока входных данных.
Модель «производитель-потребитель»	Частный случай конвейера с одним производителем и одним потребителем.