Типовые модели параллельных приложений

Существуют следующие распространенные модели параллельных приложений:

- модель делегирования («управляющий-рабочий»);
- сеть с равноправными узлами;
- конвейер («производители-потребители»);

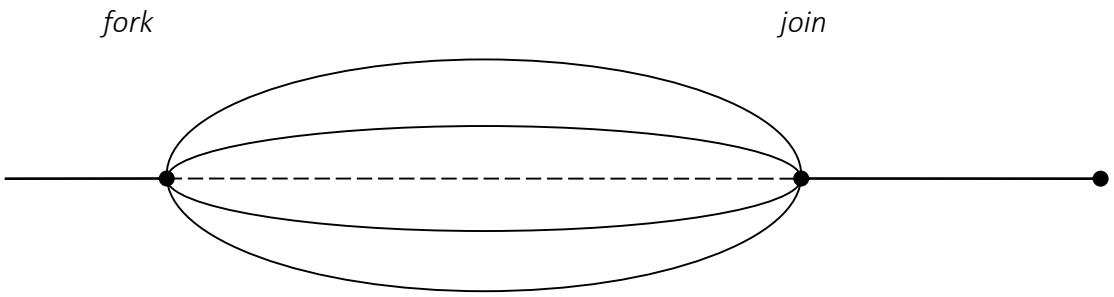
Каждая модель характеризуется собственной декомпозицией работ, которая определяет, кто отвечает за порождение подзадач и при каких условиях они создаются, какие информационные зависимости между подзадачами существуют.

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	Нет центрального узла, все потоки участвуют в обработке.
Конвейер	Конвейерный подход применяется для поэтапной обработки потока входных данных. Входные данные обрабатываются строго последовательно.

Модель делегирования

В модели делегирования выделяется один центральный поток (менеджер, управляющий, мастер) и несколько рабочих потоков. Управляющий поток запускает рабочие потоки, передает им все необходимые данные, контролирует работу и обрабатывает результаты после их завершения.

К модели делегирования относится так называемая схема «fork-join». Этап “fork” (разветвление) – делегирование полномочий рабочим потокам: создание, запуск, передача параметров. Этап “join” (присоединение) – ожидание завершения работы потоков.



Самым простым способом реализации схемы `fork-join` является применение шаблона `Parallel.Invoke`:

```
Parallel.Invoke(worker1, worker2, worker3);
```

Шаблон параллельно запускает рабочие элементы и блокирует основной поток до завершения работы.

Если в главном потоке выполняется какая-либо работа параллельно с рабочими потоками, то можно использовать объекты Task и встроенные механизмы ожидания.

```
Task t1 = Task.Factory.StartNew(..);
Task t2 = Task.Factory.StartNew(..);
Task t3 = Task.Factory.StartNew(..);
// Параллельная работа центрального узла
fManager();
// Ожидаем завершения работы
Task.WaitAll(t1, t2, t3);
// Обрабатываем результаты
fResults();
```

Объект синхронизации `CountdownEvent` позволяет выполнять координацию работы управляющего и рабочих потоков. Объект обладает внутренним счетчиком, который устанавливается при инициализации объекта. При завершении работы или достижении какого-либо этапа рабочий поток вызывает метод `Signal`, уменьшающий внутренний счетчик. Когда внутренний счетчик становится равным нулю, объект сигнализирует потоку, ожидающему с помощью метода `Wait`.

```
static void Main()
{
     int N = 5;
     CountdownEvent ev = new CountdownEvent(N);
     Thread[] workers = new Thread[N];
     for(int i=0; i<N; i++)
     {
          int y = i;
          workers[i] = new Thread(() => {
               DoSomeWork1(y);
```

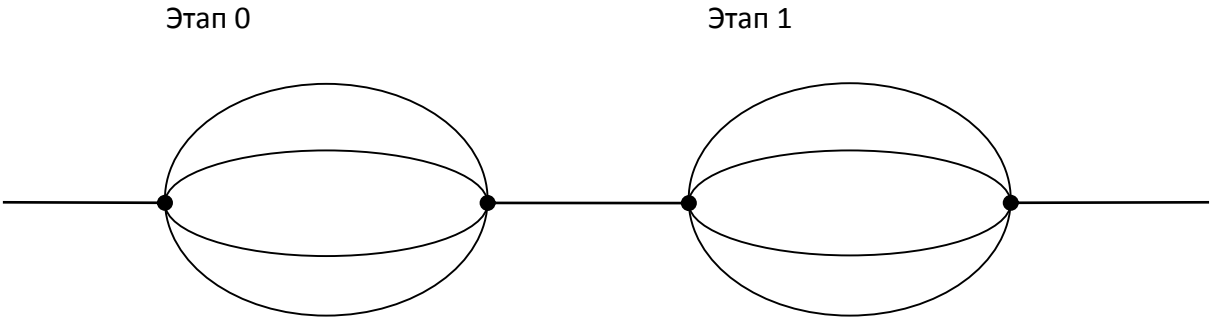
```
                ev.Signal();
                DoSomeWork2(y);
                });
          workers[i].Start();
     }
     while(!ev.IsSet)
     {
          ev.Wait(TimeSpan.FromSeconds(5));
          Console.WriteLine("{0}-рабочих закончили",
                                        ev.CurrentCount);
          SaveCurrentResults();
     }

}
```

При инициализации объекта указывается начальное значение внутреннего счетчика объекта. Каждый рабочий поток при завершении работы уменьшает значение счетчика с помощью метода `Signal`. Уменьшение счетчика осуществляется потокобезопасно. Когда значение счетчика уменьшится до нуля, объект синхронизации сигнализирует ожидающему главному потоку, при этом главный поток разблокируется.

Кроме блокирующего ожидания с помощью метода `Wait`, также можно использовать перегрузку, принимающую в качестве аргумента интервал в миллисекундах, в течение которого поток блокируется. Текущее значение внутреннего счетчика можно узнать с помощью свойства `CurrentCount`, свойство `IsSet` позволяет определить установлен ли сигнал или нет.

Объект `Barrier` инкапсулирует барьерную синхронизацию с возможностью нескольких этапов. При барьерной синхронизации несколько участников могут параллельно выполнять свою работу. При завершении работы участники дожидаются остальных. После завершения работы всех участников выполняется финальный обработчик на данном этапе. После завершения работы финального обработчика начинается следующий этап, и работа участников возобновляется.



При инициализации объекта указывается число участников барьерной синхронизации и делегат, вызываемый в конце каждого этапа.

```
Barrier bar = new Barrier(3, (bar) =>
     {
          Console.WriteLine("Phase: {0}",
                    bar.CurrentPhaseNumber);
     });

Action worker = () => {
          Work1();
          bar.SignalAndWait();
          Work2();
          bar.SignalAndWait();
          Work3();
     };
var w1 = worker; var w2 = worker; var w3 = worker;
Parallel.Invoke(w1, w2, w3);
```

Метод `SignalAndWait` сигнализирует о завершении работы данным участником и блокирует поток до завершения работы всех участников. Объект Barrier позволяет изменять число участников в процессе работы.

Рекурсивные алгоритмы относятся к моделям делегирования. В качестве примера рассмотрим алгоритм быстрой сортировки. Алгоритм рекурсивно разбивает диапазон чисел на два диапазона в соответствии с выбранным ведущим элементом – левый диапазон содержит только числа, меньшие или равные ведущему элементу; правый диапазон содержит числа, большие ведущего элемента. Распараллеливание алгоритма сводится к одновременному выполнению обработки левого и правого диапазона.

```
static void ParallelSort(T[] data, int startIndex, int endIndex,
                         IComparer<T> comparer,
                         int minBlockSize=10000) {

     if (startIndex < endIndex) {
      // мало элементов – выполняем последовательную сортировку
          if (endIndex - startIndex < minBlockSize) {
             // Последовательная сортировка
             Array.Sort(data, startIndex,
                         endIndex - startIndex + 1, comparer);
          } else {
            // Определяем ведущий элемент
             int pivotIndex = partitionBlock(data, startIndex,
                                  endIndex, comparer);
            // обрабатываем левую и правую часть
             Action leftTask = () =>
                     {
                        ParallelSort(data, startIndex,
                              pivotIndex - 1, comparer,
                             depth + 1, maxDepth, minBlockSize);
                     });
             Action rightTask = () =>
                     {
```

```
                        ParallelSort(data, pivotIndex + 1,
                              endIndex, comparer,
                            depth + 1, maxDepth, minBlockSize);
                    });
                    // wait for the tasks to complete
                Parallel.Invoke(leftTask, rightTask);
            }
        }
}

// Осуществляем перераспределение элементов
static int partitionBlock(T[] data, int startIndex,
                          int endIndex, IComparer<T> comparer) {

     // Ведущий элемент
     T pivot = data[startIndex];
     // Перемещаем ведущий элемент в конец массива
     swapValues(data, startIndex, endIndex);
     // индекс ведущего элемента
     int storeIndex = startIndex;
     // цикл по всем элементам массива
     for (int i = startIndex; i < endIndex; i++) {
     // ищем элементы меньшие или равные ведущему
          if (comparer.Compare(data[i], pivot) <= 0) {
          // перемещаем элемент и увеличиваем индекс
               swapValues(data, i, storeIndex);
               storeIndex++;
          }
     }
     swapValues(data, storeIndex, endIndex);
     return storeIndex;
}
// Обмен элементов
static void swapValues(T[] data,
                    int firstIndex, int secondIndex) {

     T holder = data[firstIndex];
     data[firstIndex] = data[secondIndex];
     data[secondIndex] = holder;
}

static void Main(string[] args) {
     // generate some random source data
     Random rnd = new Random();
     int[] sourceData = new int[5000000];
     for (int i = 0; i < sourceData.Length; i++) {
          sourceData[i] = rnd.Next(1, 100);
     }

     QuickSort(sourceData, new IntComparer());
}
```

Основная проблема рекурсивных алгоритмов заключается в снижении эффективности при большой глубине рекурсии. Для ограничения рекурсивного разбиения множества данных применяется пороговая величина MinBlock. Если число элементов в блоке незначительно, то выполняется нерекурсивная сортировка (пузырьковая или сортировка со вставками). Распараллеливание быстрой сортировки приводит к еще одному источнику накладных расходов – рекурсивное порождение задач, конкурирующих за рабочие потоки пула. При использовании пользовательских потоков (работа с объектами Thread) конкуренция будет фатальной – рекурсивное порождение потоков приводит к значительным накладным расходам. Для контроля степени параллелизма применяют несколько подходов. Самый простой способ заключается в контроле глубины рекурсии – если глубина рекурсии превышает некий порог, то выполняется последовательная быстрая сортировка.

```
static void ParallelSort(T[] data, int startIndex, int endIndex,
                         IComparer<T> comparer,
                         int minBlockSize=10000,
                         int depth = 0,
                         int MaxDepth)
{
     // Последовательная сортировка
     if (endIndex - startIndex < minBlockSize)
          InsertionSort(data, startIndex, endIndex, comparer);
     else
     {
          // Определяем ведущий элемент
          int pivotIndex = partitionBlock(data, startIndex,
                                          endIndex, comparer);
          // обрабатчик левой части
          Action leftTask = () =>
                      {
                          ParallelSort(data, startIndex,
                                pivotIndex - 1, comparer,
                               depth + 1, maxDepth, minBlockSize);
                      });
          // обработчик правой части
          Action rightTask = () =>
                      {
                          ParallelSort(data, pivotIndex + 1,
                                endIndex, comparer,
                               depth + 1, maxDepth, minBlockSize);
                      });

          if(depth >= MaxDepth)
          {
               leftTask();
               rightTask();
          }
          else
          {
               Parallel.Invoke(leftTask, rightTask);
```

```
            }

      }

}
```

Глубина рекурсии является простым критерием, но не оптимальным. При плохом выборе ведущего элемента, блоки будут неравномерными, и глубина рекурсии для обработки каждого блока будет различной. Если правая часть содержит мало элементов, то обработка правой части будет завершена достаточно быстро. Поэтому распараллеливание обработки левой части может осуществляться и при большей глубине, чем задано параметром `MaxDepth`. Реализовать «адаптивный» параллелизм можно с помощью разделяемого счетчика фактических выполняющихся параллельных вызовов. При параллельном запуске быстрой сортировки - счетчик увеличивается, при завершении параллельных вызовов – счетчик уменьшается. Изменения счетчика необходимо выполнять атомарно с помощью методов `Interlocked.Increment`, `Interlocked.Decrement`.

```
static int parallelCalls;
static void ParallelSort(T[] data, int startIndex, int endIndex,
                         IComparer<T> comparer,
                         int minBlockSize=10000)
{
     // Последовательная сортировка
     if (endIndex - startIndex < minBlockSize)
          InsertionSort(data, startIndex, endIndex, comparer);
     else
     {
          // Определяем ведущий элемент
          int pivotIndex = partitionBlock(data, startIndex,
                                          endIndex, comparer);
          // обработчик левой части
          Action leftTask = () =>
                      {
                          ParallelSort(data, startIndex,
                               pivotIndex - 1, comparer,
                              depth + 1, maxDepth, minBlockSize);
                      });
          // обработчик правой части
          Action rightTask = () =>
                      {
                          ParallelSort(data, pivotIndex + 1,
                                endIndex, comparer,
                              depth + 1, maxDepth, minBlockSize);
                      });

          if (parallelCalls > MaxParallelCalls)
          {
               leftTask();
               rightTask();
```

```
            }
            else
            {
                  Interlocked.Increment(ref parallelCalls);
                  Parallel.Invoke(leftTask, rightTask);
                  Interlocked.Decrement(ref parallelCalls);
            }

      }

}
```

Максимальное число параллельных вызовов `MaxParallelCalls` можно выбирать в зависимости от числа ядер вычислительной системы:

```
MaxParallelCalls = System.Environment.ProcessorCount / 2;
```

Таким образом, для двуядерной системы разрешается один параллельный вызов двух методов.

Модель с равноправными узлами

В модели с равноправными узлами все потоки (или задачи) участвуют в обработке; центрального узла нет. Работа узлов может осуществляться параллельно. Задачи, в которых декомпозиция осуществляется по данным, являются примером модели с равноправными узлами. Для реализации алгоритмов с параллелизмом по данным можно использовать средства TPL, автоматически осуществляющие декомпозицию и агрегацию результатов с учетом возможностей вычислительной системы и текущей загруженностью. В эту группу входят шаблоны `Parallel.For, Parallel.ForEach`, а также технология PLINQ.

В следующем примере рассматривается матричное умножение с декомпозицией по строкам первой матрицы. Каждый рабочий поток (подзадача) оперирует с одной или несколькими строками первой матрицы и всей второй матрицей. Таким образом, каждый поток вычисляет соответствующие строки результирующей матрицы. Для эффективного разделения строк матрицы можно использовать шаблон Parallel.For.

```
      Parallel.For(0, N, i => {
            for(int j=0; j<N; j++)
                  for(int k=0; k<N; k++)
                        C[i,j] += A[i,k] * B[k, j];
            });
```

В модели с равноправными узлами работа каждого участника может включать несколько последовательных этапов. Примером многоэтапной обработки является шаблон Map/Reduce. Обработка элементов включает следующие этапы:

1. Map. Обработка элементов, формирование для каждого элементы пары «ключ-значение», группировка элементов по ключам.

2. Reduce. Обработка сгруппированных элементов и выполнение для каждой группы заданной редукции.

Примером шаблона Map/Reduce является задача подсчёта встречаемости слов в тексте. Операция Map создает для каждого уникального слова пару «ключ-значение», ключ соответствует слову, значение равно единице. Все слова группируются. Операция Reduce вычисляет количество слов в каждой группе.

```
// Операция Map
    // Генерируем пары ключ-значение (word, 1)
    ILookup<string, int> map =
        words.AsParallel().ToLookup(p => p, k => 1);

// Операция Reduce
     // Вычисляем встречаемость слов
     // Отбираем с частотой встречаемости больше 1
    var reduce = from IGrouping<string, int> wordMap
                 in map.AsParallel()
                 where wordMap.Count() > 1
                 select new { Word = wordMap.Key,
                              Count = wordMap.Count() };
// Отображение результатов
    foreach (var word in reduce)
        Console.WriteLine("Word: '{0}'; Count: {1}",
            word.Word, word.Count);
    Console.ReadLine();
```

Для повышения эффективности алгоритм можно переписать в виде одного PLINQ-запроса:

```
var files =
     Directory.EnumerateFiles(dirPath, "*.txt").AsParallel();

var counts = files
     .SelectMany(f =>
          File.ReadLines(f).SelectMany(line =>
               line.Split(delimeters)))
     .GroupBy(w => w)
     .Select(g => new {Word = g.Key, Count = g.Count()});
```

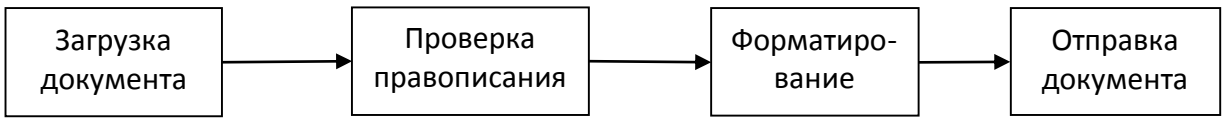
В первой строке инициализируется перечислимый список файлов с расширением *.txt в директории `dirPath`. Список файлов представляет собой тип `ParallelQuery<File>`, поэтому все запросы выполняются параллельно. Первый запрос `SelectMany` формирует общий список слов из всех файлов. Для разделения строк файла на слова используется массив разделителей `delimeters`. Оператор `GroupBy` осуществляет группировку слов, последний оператор Select для каждой группы формирует безымянный тип с полями `Word` и `Count`.

Модель конвейера

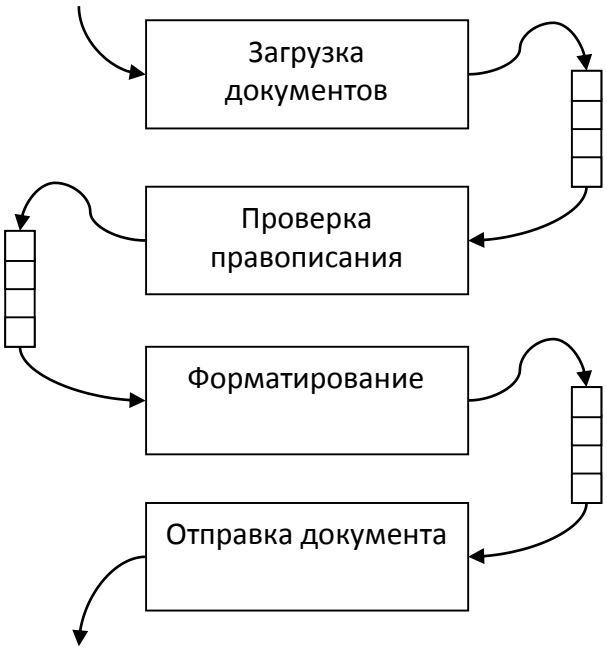
В модели конвейерной обработки (`pipelines`) поток обрабатываемых данных проходит через несколько этапов. Прохождение этапов осуществляется строго последовательно. Параллелизм достигается за счет одновременной обработки разных элементов на разных этапах.

Если последовательность элементов заранее определена и порядок обработки элементов не важен, то для распараллеливания эффективнее использовать модель с равноправными узлами (шаблон `MapReduce`, шаблон `Parallel.For`). Конвейерная обработка возникает при работе с последовательными потоками данных (потоки событий, потоки видеосигналов, потоки изображений), а также при многоэтапной обработке элементов последовательности в строго заданном порядке.

Рассмотрим конвейер, обрабатывающий документы. Обработка документов включает следующие этапы: загрузка документа (из файла, из почтового сервера и т.д.), проверка правописания, форматирование, отправка документа. Обработка документов выполняется строго последовательно, то есть первым отправляется тот документ, который загружен первым. Но при организации конвейера обработка разных документов на разных этапах может осуществляться параллельно.



Реализацию конвейерной обработки можно реализовать с помощью задач и конкурентных очередей. Каждая задача реализует этап конвейера, очереди выступают буферами, накапливающими элементы.



Последовательный алгоритм обработки изображений выглядит следующим образом:

```
while(bWorking)
{
     doc = LoadDoc(..);
     spelledDoc = CheckSpelling(doc);
     formattedDoc = FormatDoc (spelledDoc);
     Send(formattedDoc);
     nDoc++;
}
```

Для реализации конвейера необходимо ввести буферы, предназначенные для параллельной работы, и оформить этапы в виде асинхронных задач:

```
var downloads = new BlockingCollection<MyDoc>(limit);
var checked = new BlockingCollection<MyDoc>(limit);
var formatted = new BlockingCollection<MyDoc>(limit);

var factory =
         new TaskFactory(TaskCreationOptions.LongRunning,
                         TaskContinuationOptions.None);

var loadTask = factory.StartNew(() =>
               LoadingDocs(downloads));

var checkTask = factory.StartNew(() =>
               CheckingDocs(downloads, checked));

var filterTask = factory.StartNew(() =>
               FilteringDocs(checked, filtered));
```

```
var sendTask = factory.StartNew(() =>
         SendingDocs(formatted.GetConsumingEnumerable()));

Task.WaitAll(loadTask, checkTask, filterTask, sendTask);
```

Задачи создаются с параметром `LongRunning`, чтобы с каждым этапом был связан собственный поток, и планировщик не оценивал производительность выполнения задач. Задачи читают элементы из одной очереди и пишут в другую. Если во входной очереди нет элементов, то поток блокируется для ожидания поступления элементов. Если выходная очередь переполнена, то поток, осуществляющий запись, блокируется в ожидании извлечения, хотя бы одного элемента.