Распараллеливание методов

Шокуров Антон В.
shokurov.anton.v@yandex.ru
http://машинноезрение.рф

26 марта 2018 г.

Версия: 0.10

Аннотация

В данной заметке показано как распараллеливать методы. Показано на примере подсчета суммы массива. Используется библиотека pthread.

Предварительная версия!

1 Распараллеливание

Распараллеливание позволяет повысеть скорость работы программы за счет того, что код фактически работает одновременно на несколько ядрах.

1.1 Подготовка кода

В качестве примера рассмотрим метод, который суммирует элементы массива.

Исходная версия кода Рассмотрим метод вычисления суммы элементов массива. В обычной реализации это будет стандартный цикл в котором очередной элемент добавляется к итоовой переменной:

```
1 int n;
2 //...Считываем значение n, т.е. количество элементов
3 double *data = (double *)malloc( sizeof(double) * n )
4 //...Считываем значения элементов массива...
5 //теперь вычисляем сумму элементов массива.
6 double sum = 0;
7 int i;
```

```
8  for( i = 0; i < n; i++)
9    sum += data[i];
10 //Сумма посчитана, она в sum
```

Обособление При распараллеливание каждая версия кода живет своей отдельной жизнью. Поэтому для распараллеливания код нужно привести к соответствующему виду. Для этого нужно понять какие имено переменные нужны для коректного выполнения метода. Также слудет понять, что является результатом, т.е. возвращаемом значением. В данном случае, нужен указатель на данные и их количества. Результатом работы метода является переменая отвечающая за сумму.

Проще всего начать с создания функции, которая имеет соответсвующие аргументы. И далее переписать предыдущий код следующим образом:

```
1  double do_work( double *data, int n)
2  {
3    int i;
4    double sum = 0;
5    for( i = 0; i < n; i++)
6      sum += data[i];
7    return sum;
8  }
9
10 //....некий код...
11
12 int n;
13 //...Считываем значение n, т.е. количество элементов
14 double *data = (double *)malloc( sizeof(double) * n )
15 //...Считываем значения элементов массива...
16 //теперь вычисляем сумму элементов массива.
17 double sum;
18 sum = do_work( data, n);
19 //Сумма посчитана, она в sum
```

Структура После такого преобразования можно обособить уже сами даные в отдельную структуру:

```
1  struct _my_data_;
2  typedef _my_data_ mydata;
```

```
3
4  struct _my_data_
5  {
6    double *data;
7    int n;
8
9    double sum;//! Возвращаемое значение
10 };
```

Отмечу, что переменная i не попала в структуру так как она является локальной, т.е. она задается и используется самим методов. Она не нужна для задания работы. Также отмечу, что результат сохранен в структуре, а не возвращается.

И переписать соответственно код:

```
1  void do_work( mydata *work)
2  {
3    int i;
4    double sum = 0;
5    for( i = 0; i < n; i++)
6      sum += data[i];
7    work->sum = sum;
8  }
9
10 //....некий код...
11
12 int n;
13 //...Считываем значение n, т.е. количество элементов
14 double *data = (double *)malloc( sizeof(double) * n )
15 //...Считываем значения элементов массива...
16 //теперь вычисляем сумму элементов массива.
17 mydata work;
18 work.data = data;
19 work.n = n;
20 do_work( &data );
21 //Сумма посчитана, она в data->sum.
```

Теперь нужно научится выполнять задания почастям.

По частям Распараллеливание осуществляется за счет того, что одновременной делаются вычисления над разыми частями задания. В данном случае, над частями массива. Часть массива характеризуется началом и концом.

Для изображения такое деление могло бы выглядить как прямоугольник изображения. Такие прямоугольники накрывают все изображения без перекрытия.

Конкретно, предполагается, что массив делится на несколько частей. Вообще говоря, равных, и каждая часть вычилсяется по отдельности. Далее итоговый ответ собирается из разных частей.

В структуру добавлены дополнительные переменные для указаия конкретного подзадания, т.е. части массива. Конкретно, добавлены переменные start и end (не включительно).

```
1  struct _my_data_;
2  typedef _my_data_ mydata;
3
4  struct _my_data_
5  {
6    double *data;
7    int n;
8
9    //подзадание определяется:
10   int start, end;//подиапазон
11
12   double sum;//! Возвращаемое значение
13 };
```

Далее как и раньше, считыаем элементы массива. Например, из файла.

```
1  int n;
2  //...Считываем значение n, т.е. количество элементов
3  double *data = (double *)malloc( sizeof(double) * n )
4  //...Считываем значения элементов массива...
```

После считывания подготавливаем задание для каждой подчасти массива.

```
1  int k;//количество частей...
2  // k считываем....
3  mydata *work = (my_data *)malloc( sizeof(mydata) * k );
4  int i;
5  for( i = 0; i < k; i++)
6  {
7    work.data = data;
8    work.n = n;
9
10   work.start = n * i;
```

```
11    work.start /= k;//начало
12    work.end = n * ( i + 1 );
13    work.end /= k;//конец
14  }
```

Упражение. Подумай почему именно такие формулы используются для вычисления start и end.

Теперь вызываем эти подзадания. Это фактически можно рассматривать как отложенный вызов.

```
1  for( i = 0; i < k; i++)
2  {
3    // запускаем ioe задание
4    do_work( &work[i] );
5  }
6  //для каждой из подчастей сумма посчитана.
```

```
1  //теперь собираем сумму элементов массива.
2  double sum = 0;
3  for( i = 0; i < k; i++)
4  {
5    sum += work[i].sum;
6  }
7  //итоговая сумма в sum.
```

Обобщение Исходя из неких тонкостей принято прототип функции (т.е. тип аргумента и возвращаемого значения) делать единым. Так, аргумент делается указателем на void, аналогично с возвращаемым значением. Тогда функиця do_work доделывается так:

```
1  void *do_stuff(void *d)
2  {
3    mydata *data = (mydata*)d;
4    do_work( data );
5    return 0;
6  }
```

При вызове соответственно делается:

```
1  for( i = 0; i < k; i++)
2  {
```

```
3   // запускаем ioe задание
4   void *tmp = (void*)&work[i];
5   do_stuff( tmp );
6 }
7 //для каждой из подчастей сумма посчитана.
```

Параллельная работа Теперь нужно "отложеный вызов" превратить в параллельный. Для этого нужно вызвать некий системынй вызов, который это обеспечит. Стандартный способ заключается применении библиотеки pthread.

Для её использования нужке объект этоё библиотеки отвечающей за параллельную работу кода "нить": pthread_t. Этот объект нужно вставить в нашу структуру, так как она как раз у нас и отвечает за состояние некого исполнителя. Тогда структура будет такой:

```
1  struct _my_data_;
2  typedef _my_data_ mydata;
3
4  struct _my_data_
5  {
6    pthread_t id;
7    double *data;
8    int n;
9
10   //подзадание определяется:
11   int start, end;//подиапазон
12
13   double sum;//! Возвращаемое значение
14 };
```

Отмечу, что в начала файла следует поместить достаточное количество загаловочных файлов. Как минимум pthread.h Полный список лучше посмотреть по комманде man pthread_create, там где показан пример.

Для запуска кода в паралель используется функция pthread_create. Тогда код будет выгляеть так:

```
1 for( i = 0; i < k; i++)
2 {
3   // запускаем ioe задание
4   void *tmp = (void*)&work[i];
5   pthread_create( &work[i].id, 0, do_stuff, tmp );
6 }
```

```
7 //для каждой из подчастей сумма посчитана.
```

Замечу что как прототип функции do_stuff, так и переменной tmp уже какие надо.

В отличии от отложенного вызова нельзя сразу взять и собрать каждую из подсумм. Для начала нужно дождатся когда разные нити завершат работу. Для этого вызывается системынй вызов pthread_join.

```
1 //пока нет гарантий, что подсуммы посчитаны.
2 for( i = 0; i < k; i++)
3 {
4   pthread_join( &work[i].id, 0).
5 }
6 //для каждой из подчастей сумма посчитана.
```

Теперь можно суммировать подсуммы как и раньше:

```
1 //теперь собираем сумму элементов массива.
2 double sum = 0;
3 for( i = 0; i < k; i++)
4 {
5   sum += work[i].sum;
6 }
7 //итоговая сумма в sum.
```