Augusdi的专栏

攀登技术的高峰，我与大家共勉！

转 OpenMP3.0的新特性Task指令基础

2013年04月16日 10:52:54 阅读数：4824

从OpenMP3.0开始，OpenMP增加了task指令，这是OpenMP3.0中最激动人心的一个新特性。

本文的”术语“大多数是根据个人理解用词，不保证用词准确性。

（1）task基础

OpenMP Tutorials: 任务构造https://computing.llnl.gov/tutorials/openMP/#Task

从功能上说：

1. The TASK construct defines an explicit task, which may be executed by the encountering thread, or deferred for execution by any other thread in the team.
2. The data environment of the task is determined by the data sharing attribute clauses.
3. Task execution is subject to task scheduling - see the OpenMP 3.0 specification document for details.

任务构造定义一个显式的任务，可能会被遇到的线程马上执行，也可能被延迟给线程组内其他线程来执行。任务的执行，依赖于OpenMP的任务调度。

语法：

```
                untied
                default (shared | none)
                mergeable
                private (list)
                firstprivate (list)
                shared (list)
    structured_block
```

先来理解task的基本使用再来分析其子句的含义。

task，简单的理解，就是定义一个任务，线程组内的某一个线程来执行此任务。和工作共享结构很类似，我们都知道，for也是某一个线程执行某一个迭代，如果把每一个迭代看成一个task，那么就是task的工作方式了，在for只能用于循环迭代的基础上，OpenMP提供了sections构造，用于构造一个sections，然后里面定义一堆的section，每一个section被一个线程去执行，这样，每一个section也类似于for的每一次迭代，只是使用sections会更灵活，更简单，但是其实，for和sections在某种程度上是可以转换的，用下面的例子来看一个使用sections和for指令分别执行“三个”任务的例子：

```
#include <omp.h>

#define TASK_COUNT      3

void task1(int p)
{
        printf("task1, Thread ID: %d, task: %d\n", omp_get_thread_num(), p);
}

void task2(int p)
{
        printf("task2, Thread ID: %d, task: %d\n", omp_get_thread_num(), p);
}

void task3(int p)
{
        printf("task3, Thread ID: %d, task: %d\n", omp_get_thread_num(), p);
}
```

```
22
23  #pragma omp parallel num_threads(2)
24          {
25  #pragma omp sections
26                  {
27  #pragma omp section
28                          task1(10);
29  #pragma omp section
30                          task2(20);
31  #pragma omp section
32                          task3(1000);
33                  }
34          }
35
36  #pragma omp parallel num_threads(2)
37          {
38  #pragma omp for
39                  for(int i = 0;i < TASK_COUNT; i ++)
40                  {
41                          if(i == 0)
42                                  task1(10);
43                          else if (i == 1)
44                                  task2(20);
45                          else if (i == 2)
46                                  task3(1000);
47                  }
48          }
49
50          return 0;
51  }
```

当然，这个程序不是这里要讨论的重点，只是为了说明for和sections的一些类似之处，或者其实可以理解为sections其实是for的展开形式，适合于少量的“任务”，并且适合于没有迭代关系的“任务”，很显然，上面的例子适合用sections去解决，因为本身是三个任务，不存在迭代的关系，三个任务和循环迭代变量没

复制

```
#include <omp.h>

void task(int p)
{
        printf("task, Thread ID: %d, task: %d\n", omp_get_thread_num(), p);
}

#define N               3
void init(int*a)
{
        for(int i = 0;i < N;i++)
                a[i] = i + 1;
}

int main(int argc, _TCHAR* argv[])
{
        int a[N];
        init(a);

#pragma omp parallel num_threads(2)
        {
#pragma omp sections
                {
#pragma omp section
                        task(a[0]);
#pragma omp section
                        task(a[1]);
#pragma omp section
                        task(a[2]);
                }
        }

#pragma omp parallel num_threads(2)
        {
```

```
37                  {38                         task(a[i]);
39                  }
40          }
41
42          return 0;
43  }
```

这个例子，很容易理解了，把一个数组内的每一个元素“并行”的传递给task()函数，执行一个“任务”。同样，for和sections都能解决，但是如果N太大了，比如N是100，那sections就为难了，这里要说明的是：sections不能使用嵌套的形式，比如：

```
1                           for(int i = 0;i < N;i++)
2                           {
3   #pragma omp section
4                                   task(a[0]);
5                           }
```

这样是不行的，section只能显式的，直接的在sections里面书写，可以理解为”静态“的。继续研究这个例子，假设现在的需求是对如下的代码进行并行化：

```
1   for(int i = 0;i < N; i=i+a[i])
2                   {
3                           task(a[i]);
4                   }
```

对于这样的需求，OpenMP的for指令也是无法完成的，因为for指令在进行并行执行之前，就需要”静态“的知道任务该如何划分，而上面的i=i+a[i]，在运行之前，是无法知道有那些迭代，需要如何进行划分，因为其迭代的循环依赖于数组a里面保存的值。那么对于这样的循环，该如何并行？最关键的是，从语义上，这个循环是明显可以进行并行的。这就是之所以OpenMP3.0提供task的原因了。

在此，先总结一下for和sections指令的”缺陷“：无法根据运行时的环境动态的进行任务划分，必须是预先能知道的任务划分的情况。

（2）task的基本使用（说明：当前我的VS2010不支持OpenMP3.0的task指令，下面的测试内容无法使用VS运行，可以采用支持OpenMP的task特性的编译器运行）

```
#include <omp.h>

void task(int p)
{
        printf("task, Thread ID: %d, task: %d\n", omp_get_thread_num(), p);
}

#define N               50
void init(int*a)
{
        for(int i = 0;i < N;i++)
                a[i] = i + 1;
}

int main(int argc, _TCHAR* argv[])
{
        int a[N];
        init(a);

#pragma omp parallel num_threads(2)
        {
#pragma omp single
                {
                        for(int i = 0;i < N; i=i+a[i])
                        {
#pragma omp task
                                task(a[i]);
                        }
                }
        }

        return 0;
}
```

那么，task和前面的for和sections的区别在于：task是“动态”定义任务的，在运行过程中，只需要使用task就会定义一个任务，任务就会在一个线程上去执行，那么其它的任务就可以并行的执行。可能某一个任务执行了一半的时候，或者甚至要执行完的时候，程序可以去创建第二个任务，任务在一个线程上去执行，一个动态的过程，不像sections和for那样，在运行之前，已经可以判断出可以如何去分配任务。而且，task是可以进行嵌套定义的，可以用于递归的情况等等。

总结task的使用：task主要适用于不规则的循环迭代（如上面的循环）和递归的函数调用。都是无法使用for来完成的情况。

（3）显示任务和隐式任务（implicit&explicit）

task的作用就是创建一个显式的任务，那么什么是隐式的任务呢？OpenMP的任务分为显式和隐式两种，根据我的个人理解，分析下面的代码：

```
#pragma omp parallel num_threads(2)
        {
#pragma omp single
                {
                        for(int i = 0;i < N; i=i+a[i])
                        {
#pragma omp task
                                task(a[i]);
                        }
                        task(1000);
 }
}
```

其中task(1000);就属于一个隐式的任务。因为执行完for后，会执行这一个任务，而上面的任务可能也会同时执行。

（4）task的嵌套

A task construct may be nested inside an outer task, but the task region of the inner task is not a part of the task region of the outer task.

任务构造结构可以嵌套在另一个task结构中，但是内部的task结构并不属于外部的task区域的一部分。

```
#pragma omp task
{
task(a[i]);
#pragma omp task
task(a[i]);
}
```

When an if clause is present on a task construct, and the if clause expression evaluates to false, an undeferred task is generated, and the encountering thread must suspend the current task region, for which execution cannot be resumed until the generated task is completed. Note that the use of a variable in an if clause expression of a task construct causes an implicit reference to the variable in all enclosing constructs
如果给一个task使用了if子句，如果if子句的表达式是false，会生成一个不延迟的任务，这样，遇到这个task的当前线程必须挂起当前的task区域，直到当前的任务完成之后才会恢复。个人理解，当前线程挂起，那么这个task是不是由其它的线程去执行呢，还是就是当前的这个线程执行这个任务?
final子句:
When a final clause is present on a task construct and the final clause expression evaluates to true, the generated task will be a final task. All task constructs encountered during execution of a final task will generate final and included tasks. Note that the use of a variable in a final clause expression of a task construct causes an implicit reference to the variable in all enclosing constructs.
如果给task使用了final子句，如果final表达式的值为true，生成的任务是一个终结任务。所有任务遇到终结任务执行的时候会生成终结和包含的任务。PS：不太理解！
default、private 、firstprivate、shared等数据属性子句就不多说了。
mergeable、untied等参考OpenMP Spec: http://www.openmp.org/mp-documents/OpenMP3.1.pdf
（5）其它:
和task相关的还有一些其他深入一些的内容，包括上面的task指令的子句也有一些需要深入理解，在这里就不讨论了。在以后的学习过程中再总结。毕竟，task是OpenMP的一个新的特性，也是很有用的。这里主要是理解task的基本使用，

参考:
http://emonkey.blog.sohu.com/165038545.html
http://wikis.sun.com/display/openmp/Using+the+Tasking+Feature（里面是几个使用task的best practice，挺好的文章）

个人分类： OpenMP

想对作者说点什么？ 我来说一句

lingjiexiong0344 2017-08-02 16:16:11 #2楼