Augusdi的专栏

攀登技术的高峰，我与大家共勉！

转 OpenMP中数据属性相关子句详解(3): reduction子句

2013年04月16日 11:00:57 阅读数：7668

reduction的作用：A private copy for each list variable is created for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.
reduction子句为变量指定一个操作符，每个线程都会创建reduction变量的私有拷贝，在OpenMP区域结束处，将使用各个线程的私有拷贝的值通过制定的操作符进行迭代运算，并赋值给原来的变量。
reduction的语法为recutioin(operator:list)和其他的数据属性子句不一样的是多了一个operator参数。由于最后会进行迭代运算，所以不是所有的运算符都能作为reduction的参数，而且，迭代运算需要一个初始值，不是所有的操作符需要有相同的初始值，一般而言，常见的reduction操作符的初始值为：+(0),*(1),-(0),&~(0),|(0),^(0),&&(1),||(0)，当然，这不是必须的，比如叠加运算的初始值，可以是任意值，只是表达的含义不一样而已，但是对于某些操作符，有些初始值是没有什么意义的，比如乘法迭代如果初始值为0没有什么意义，结果肯定是0了！

典型的使用reduction的例子，就是迭加（求和）操作了：

```
#include <omp.h>
#define COUNT 10

int main(int argc, _TCHAR* argv[])
{
        int sum = 100;          // Assign an initial value.
#pragma omp parallel for reduction(+:sum)
        for(int i = 0;i < COUNT; i++)
```

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！ 登录 注册

```
        }12         printf("Sum: %d\n",sum);
13
14          return 0;
15  }
```

这个例子就是对0到COUNT进行求和，由于初始值为100，所以还会加一个100，如果只是为了求和，只需要初始值为0即可。使用reduction可以避免数据竞争的发生，将上面例子的COUNT修改为一个比较大的值，如果不使用reduction，会发现有数据竞争导致结果不一致，使用reduction后，每次都能得到正确的结果。

reduction的使用是比较简单的，主要还是需要理解上面说到的“初始值”，第一个理解是这里的100这样的初始值，这是并行区域外的初始值，会在最后计算到迭代结果中，那么还有一个隐含的初始值，就是我们知道，使用了reduction，那么每个线程都会构造一个reduction变量的线程副本，那么其值为多少呢？从上面的例子可以看出，其初始值就是0，如果初始值都是100，那么结果应该是100会被加线程数目的次数。**初始值的确定方法**就是上面提到的：**+(0),*(1),-(0),&~(0),|(0),^(0),&&(1),||(0)**。

所以，理解reduction的工作过程：

（1）进入并行区域后，team内的每个新的线程都会对reduction变量构造一个副本，比如上面的例子，假设有四个线程，那么，进入并行区域的初始化值分别为：sum0=100,sum1 = sum2 = sum3 = 0.为何sum0为100呢？因为主线程不是一个新的线程，所以不需要再为主线程构造一个副本（没有找到官方这样的说法，但是从理解上，应该就是这样工作的，只会有一个线程使用到并行区域外的初始值，其余的都是0）。

（2）每个线程使用自己的副本变量完成计算。

（3）在退出并行区域时，对所有的线程的副本变量使用指定的操作符进行迭代操作，对于上面的例子，即sum' = sum0'+sum1'+sum2'+sum3'.

（4）将迭代的结果赋值给原来的变量（sum），sum=sum'.

注意：

reduction只能用于标量类型（int、float等等）；

reduction只用于一个区域构造或者工作共享构造的结构中，并且，在这个区域中，reduction的变量只能被用于类似如下的语句：

```
1  x = x op expr
2  x = expr op x (except subtraction)
```

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！

登录 注册

```
5 | ++x  6 | x--
7 | --x
```

说明：经过测试，其实不符合这一规则的时候，编译运行都不会有问题，有些甚至也是可以解释清楚为什么结果是这样的，但是无论如何，一般使用reduction的时候，都是一些迭代的情况，语义应该是很清楚的情况。看下面的一个“错误”的例子：

```
1  #define COUNT 10
2
3  int main(int argc, _TCHAR* argv[])
4  {
5          int sum = 100;          // Assign an initial value.
6  #pragma omp parallel for reduction(+:sum)
7          for(int i = 0;i < COUNT; i++)
8          {
9                  sum += i;
10                 sum = 1;
11         }
12         printf("Sum: %d\n",sum);
13
14         return 0;
15 }
```

输出结果为104（4核机器）。这个例子，sum=1;这个表达式是不应该出现的，但是如果就这么些，编译运行都没问题，而且，这个结果甚至也算是预料中的。每一个线程计算结束后，其sum的值都是1，四个线程，然后初始值是100，所以最后结果是104。:) 无论如何，即使可以解释得通，相信也没有这样使用的场合，至少，不要依赖于这样的实现的结果。从这个错误的例子，反过来，我发现上面的关于”理解reduction的工作过程“似乎不太完全正确，其中第一步，进入并行区域后，初始值为”sum0=100,sum1 = sum2 = sum3 = 0“，如果这样，只是一个初始值，那么计算后，在这个例子里，所有线程的sum都是1，结果应该为4才对。所以看来，实际的理解应该是，主线程也会创建一个副本变量，其初始值也为0，在最后迭代的时候，是用sum原来的值和每个线程的副本进行计算。过程大概如下：

（1）sum=100

（2）进入并行区域，创建4个线程的4个副本：sum0=sum1=sum2=sum3=0;

（3）计算完成后，得到sum0',sum1',sum2',sum3'

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！

登录 注册

总之，具体编译器是如何实现的并不重要，关键是理解reduction是如何工作的。

个人分类： OpenMP

OpenMp之reduction求和

Openmp中的reduction求和

想对作者说点什么？ 我来说一句

OpenMP之求和（用section分块完成）

// Sum_section.cpp : 定义控制台应用程序的入口点。 //section功能：; //1.指定其内部的代码划分给线程中某个线程，不同的section由不同的线程执行; //2.将一个...

he_xiang_ 2014-09-24 11:02:27 阅读数：1293

并行计算—OpenMP—并行区域法求和

// OpenMP1.cpp : 定义控制台应用程序的入口点。 //使用并行区域方法进行求和。 #include "stdafx.h" #include #include #include #...

LY_624 2016-10-25 13:59:02 阅读数：601

OpenMP

OpenMP 2008-8-10 version 1.0 1 简介 www.openmp.org GNU的gomp项目； Include ； 编译参数-fopenmp打...

horizons_kong 2016-12-27 14:42:32 阅读数：299

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！ 登录 注册